Modelling generalization and other class-to-parent relationships

Laurence Tratt[1], Tony Clark[1], Andy Evans[2]
[1]King's College London, [2]University of York

July 1, 2002

Abstract

The UML contains the concepts of generalization and realization between two modelling elements intending to match the OO concepts of subclassing and type conformance. With particular focus on the concept of generalization we show that: by choosing a single (non-precisely defined) semantics, these concepts are too narrowly defined to model real-world variations; they are incapable of expressing the more advanced functionality implicit in their theoretical counterparts. We present the outline of a solution to these problems whereby, in order to support variability and generality of these concepts, we create a series of templates which can be combined to create modelling languages with the appropriate semantics.

1 Introduction

Many programming languages in use today derive their object orientated features from those found in Simula [DN66]; the most important of these derived features being classes and the incremental definition of subclasses. The UML [OMG01], as the de facto modelling language, also has classes; subclassing in the UML is known as generalization and is a relationship between two classes (we call such relationships class-to-parent relationships).

There are two significant issues with generalization as it stands. Firstly, as with most other parts of the UML definition, generalization is defined imprecisely using natural language. This causes confusion as it is very difficult to be sure of the exact result of a generalization relationship. Secondly, only a single semantics – no matter its preciseness – is given: there is no scope for variation. The UML is used to model a wide array of end-user programming languages and systems and this lack of flexibility leads to undesirable solutions such as forcing developers to have a local 'understanding' of what generalization means in a specific model. Related to generalization is the UML concept of realization which forces a class to conform to another class-like elements methods without actually inheriting any of those methods. Realization also suffers from the same problems as generalization. Various real world programming languages have other types of class-to-parent relationships.

It is widely recognized in the UML community that UML needs to become a 'family of languages': that is that a single semantics can never be sufficient. Our aim in this paper is to present a mechanism which allows us to create differing types of class-to-parent relationships. In order to achieve this goal, we sketch out the problem space in section 1.1, give a formal semantics for UML generalization in section 2.1 and our actual mechanism in section 3.

An issue which needs to be addressed is what relevance such class-to-parent relationships have to the world of modelling, particularly as we take most of our motivation from programming languages. Even discounting the obvious rejoinder that much modelling is of systems intended for specific programming languages, our answer is simple: many of the needs in modelling were identified from programming languages, and it seems churlish to ignore advances in programming language theory and practise. If programming language designers and theorists have given particular thought to an area – as indeed they have to the area of inheritance – we feel that such thought should be acknowledged and the work, where relevant, be absorbed in a sensible way into modelling. We believe that inheritance is one such area.

The results of our research are as follows: we are able to show the flaws in the current definitions of UML generalization and realization, as well as why no one single definition could ever hope to address all these flaws; we present a mechanism which allows different modelling languages to have different variations on generalization and realization as well as having completely new types of class-to-parent relationships.

1.1 Differing concepts

One of the chief difficulties in discussing issues such as generalization and inheritance is the wide variety of meanings these words have been associated with. Arguably, inheritance has been the most severely abused term of all: most frequently it is associated with subclassing, although subclassing and inheritance are not necessarily the same thing [AC96]. In order to make our intended meanings for these terms clear, and to sketch out the problem space we wish to work in, we give a number of examples of differing relationships between a class and its parents often in relation to the UML and to real world programming languages. We make a number of points during these examples which we return to later in the paper.

Figure 1(a) shows simple subclassing from multiple parents as found in languages such as C++ [Str97]. This example in C++ or the UML, for example, will see C contain two methods $m(p : \mathit{Int})$ and $m(p : \mathit{String})$; with any given call to a method m with a single parameter, it is possible to differentiate which of the two methods is to be called since the types *Int* and *String* are in no way related. In Python [vR01], on the other hand, C will contain only the method from A since it is interested solely in the name of the method as opposed to how many arguments it has[1] and chooses which particular method to inherit on a left-to-right depth-first basis. Eiffel [Mey92] renders this example invalid as it stands, and insists that each element in a class must have a unique name.

Figure 1(b) is much trickier as both of the methods m in A and B have the same signature. We have already seen that Python will choose the m from A. Whereas C++ and UML could safely inherit both m methods in figure 1(a), as both methods in figure 1(b) have the same signature, it is not possible to

[1] Python is untyped but as it ignores the number of arguments, typing information would not affect this selection criteria.

differentiate which of the two methods is to be called when a message is passed to an object. C++ solves this by a scheme know as 'qualified naming' [Str87]: in our example, C will contain two methods $A::m(p : Int)$ and $B::m(p : Int)$. The UML takes a different approach and would render the example invalid as it stands; if, however, C had contained a method $m(p : Int)$ then it would have been a valid UML model.

Figure 1(c) shows a class C subclassing a class A; in a language with contravariant method overriding, C will contain a single method $m(P : X) : Y$. In the UML, for example, which deals only with invariant method signatures it will contain $m(P : X) : Y$ and $m(P : Y) : X$ as inherited from A.

Figure 1(d) shows a class A being conformed to (in UML parlance realized by) a class C. The intention is that C is forced to contain a method $m(p : Int)$ with a method body: m is not inherited from A. Although we do not model it here, this is a close cousin of a class 'implementing' an interface in Java [GJSB00].

Figure 1(e) is something that is not currently accommodated by UML notation or in the UML definition, and so we have invented some notation to try to make the point: the intention is that C subclasses A but renames the attribute a to b in the process. Renaming is present in languages such as Eiffel and Sather [SO99] and is these languages solution to the problem of names clashing when a subclass has multiple parents: name clashes are illegal, but one can always rename the offending elements.

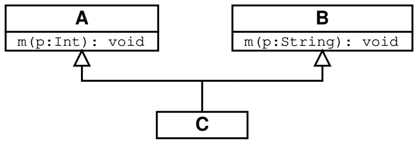
Figure 1(f) is also not currently accommodated by UML notation or in the UML definition, and as before we have invented some new notation. In this example, C is not intended to be a subclass of A; instead it merely inherits (to avoid confusion we call this 'selects') the attribute b from A. This also means that C is not a subtype of A and renders it very different from our other examples. This effect can be achieved in languages such as Python.

1.2 Defining models of class to parent relationships

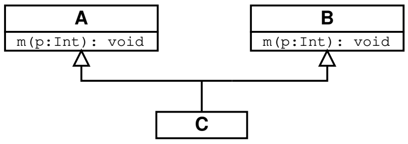
In this paper, we define a number of different types of class-to-parent relationships. In order to present the salient points of the discussion in a form that they can be most easily appreciated and tersely presented, we model a rather simpler world than that found in the UML. In our model we have only classes (with class-to-parent relationships), methods and attributes; classes contain both methods and attributes.

We use a modelling semantics as found in e.g. [CEK02] to precisely define the various different class-to-parent relationships. Essentially, one defines in separate packages: an abstract syntax; a resolved syntax, with constraints on the valid modelling elements; and a semantic mapping, which describes the valid mappings of the abstract syntax onto the resolved syntax. All constraints are written in OCL [OMG97]. For simplicity, we use the same abstract syntax, resolved syntax and semantic mapping packages throughout this paper although in practice one would probably wish to alter these packages to suit differing needs more easily.

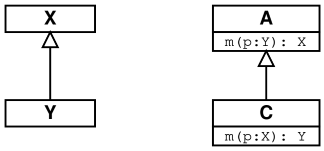
The abstract syntax and resolved syntax are defined in isolation in figures 2 and 3 respectively, and then merged together, using package extension, in figure 4 to form the basis of the semantic mapping. In order to model the varying



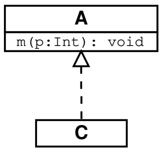
(a) Two methods with the same name



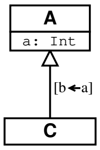
(b) Two methods with the same signature



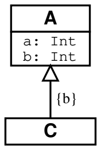
(c) Contravariant method overriding



(d) Conformance of a class



(e) Renaming



(f) Reuse of a specific element

Figure 1: Differing class to parent relationships

class-to-parent relationships, we add extra constraints to these packages when necessary.

It is important to note that in our model we intend that inheritance is 'copy down' i.e. we do not share attributes and methods between different classes. Although this is not necessarily always a perfect match with reality, the benefits gained through consistency outweigh the drawbacks in our context.

In order to distinguish between locally defined attributes and those inherited from parent classes, each element has an *is_local* boolean attribute.

All the class-to-parent relationships are modelled as relationships between a child class and a parent class; in the abstract syntax this is done via subclasses of the *SParentClassRel* abstract class and in the resolved syntax this is done via subclasses of the *ParentClassRel* abstract class. Concentrating on the resolved syntax, initially the only subclass of interest is the *SubclassRel* class which is intended to model the subclassing relationship; the other subclasses can be safely ignored until they are introduced and described in later sections (indeed, as we will see later, in an ideal world these classes would not be in this initial diagram). Each subclass of *ParentClassRel* has a boolean attribute which states whether any class which has a relation of this type with a parent class must conform to the parent class' type.

Finally we note one or two oddities in our forthcoming OCL definitions: we

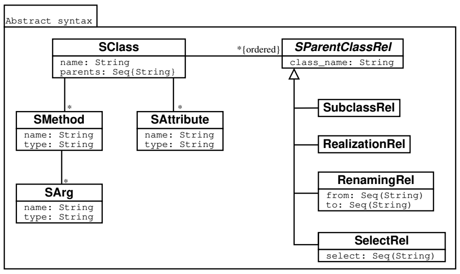


Figure 2: Abstract syntax

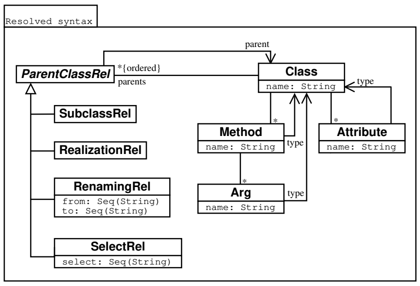


Figure 3: Resolved syntax

name invariants to enable us to refer to them easily; we prefix queries with the keyword `query` to more easily distinguish them from invariants.

1.3 Generic components of the modelling semantics packages

Several constraints and queries are shared by all our models of class-to-parent relationships, so we place them into the modelling semantics packages straight away.

1.3.1 Constraints on, and queries for the Resolved Syntax

The following two queries determine whether a class is a subtype or supertype of the current class. For simplicity, we define that a class is both a supertype and subtype of itself.

```
context Class:
  query is_subtype(a : Class) : boolean
    self = a or self.parents->select(b |
      b.is_subtype)->includes(a)

  query is_supertype(a : Class) : boolean
    self = a or a.parents->select(b |
      b.is_subtype)->includes(self)
```

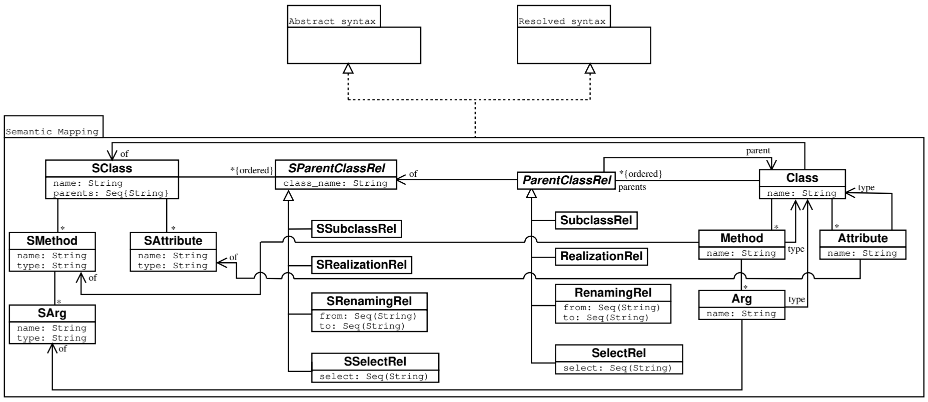


Figure 4: Semantic mapping

The following query checks whether a given methods signature matches the current methods signature i.e. the signature is invariant. It does this by: comparing the names of the methods; ensuring the types signatures invariantly match; and ensuring that both have the same number of arguments and that, for each argument in sequence, their signatures match invariantly.

```
context Method
  query invariant_signature(a : Method) : boolean
    self.name = a.name and self.type = a.type and
    self.args->size = a.args->size and
    Sequence{1 .. self.args->size}->forAll(i |
      self.args->at(i).invariant_signature(a.args->at(i)))

context Attribute
  query invariant_signature(a : Method) : boolean
    self.name = a.name and self.type = a.type

context Arg
  query invariant_signature(a : Arg) : boolean
    self.name = a.name and self.type = a.type
```

We need to ensure that there are no cycles in the class-to-parent relationship:

```
context Class:
  query all_parents() : Sequence(Class)
    self.parents->union(self.parents->iterate(a p = {} |
      p->union(a.all_parents())))

  inv no_circular_class_to_parent_relationships
    not self.all_parents()->exists(a | a = self)
```

We also define the simple queries on *SubclassRel*. In a subclass relationship, the subclass can potentially inherit all methods and attributes from its superclass and must conform to all methods and attributes from its superclass (we do not make use of the conformance methods in this section but define them for completeness).

```
context SubclassRel:
  query inheritable_methods() : Set(Method)
    self.parent.methods

  query inheritable_attributes() : Set(Attribute)
    self.parent.attributes

  query conformable_methods() : Set(Method)
    self.parent.methods

  query conformable_attributes() : Set(Attribute)
    self.parent.attributes
```

1.3.2 Constraints on, and queries for, the Semantic Mapping

There are many constraints needed in the semantic mapping to ensure that everything in the abstract syntax has a correct counterpart in the resolved syntax. Most of these constraints are very trivial so, in order to avoid cluttering the paper up with such trivialities, we give two simple examples of these constraints and do not explicitly define the rest.

```
context Class:
  inv must_contain_local_attributes
    self.attributes->select(is_local = true)->size >=
    self.sclass.attributes->size

  inv must_contain_local_methods
    self.methods->select(is_local = true)->size >=
    self.sclass.methods->size
```

2 Modelling UML generalization and realization

2.1 Generalization

As noted in section 1, the UML has a very Simula-esque view of the world: a generalizable element (the superclass) can be specialized by another (the subclass) i.e. there is said to be a 'generalization relationship' between the two. Although in the UML definition, many differing types of element can be generalized with elements of the same type, we restrict ourselves to considering classes. The UML definition specifies that the subclass must inherit from the superclass and in so doing specifies inheritance between the two elements thus ([OMG01], page 126):

> If a generalizable element has one or more parents, then it ... contains the union of the features from its own segment descriptor and the segment descriptors of all of its ancestors. For a classifier, no attribute, operation, or signal with the same signature may be declared in more than one of the segments (in other words, they may

not be redefined). A method may be declared in more than one segment. A method declared in any segment supersedes and replaces a method with the same signature declared in any ancestor. If two or more methods nevertheless remain, then they conflict and the model is ill formed. The constraints on the full descriptor are the union of the constraints on the segment itself and all of its ancestors. If any of them are inconsistent, then the model is ill formed.

In order to define this with a modelling semantics (see [Öve98] for an operational semantics), we need to add several constraints to figures 3 and 4.

We must ensure that all inheritable attributes in all parents are not overridden in the local class:

```
context Class:
  inv no_overriding_attributes
    self.parents->forAll(a |
      a.inheritable_attributes()->forAll(b |
        not self.attributes->exists(c |
          c.is_local and c.invariant_signature(b))))
```

There can be no attributes with the same signatures in different parents:

```
  inv attributes_in_at_most_one_parent
    self.parents->forAll(a, b | a <> b implies
      a.inheritable_attributes()->forAll(c |
        not b.inheritable_attributes()->exists(d |
          c.invariant_signature(d))))
```

Then we need to ensure that inheritable attributes that exist in the parent classes also exist in the subclass and are flagged as non-local (this last condition is technically redundant in conjunction with *no_overriding_attributes*, but we choose to retain it):

```
context Class:
  inv inherit_invariant_attributes
    self.parents->forAll(a |
      a.inheritable_attributes()->forAll(b |
        self.attributes->exists(c | c.is_local = false
          and b.invariant_signature(c))))
```

Although we need to allow the overriding of methods, the UML definition does not allow a particular method signature to occur in more than one parent class unless that method is overridden in the local class.

```
context Class:
  inv methods_in_at_most_one_parent_if_not_overridden_locally
    self.parents->forAll(a, b |
      a <> b implies a.inheritable_methods()->forAll(c |
        not self.attributes->exists(d |
          d.is_local and d.invariant_signature()) implies
        not b.inheritable_methods()->exists(d |
          c.invariant_signature(d))))
```

Next we specify the inheritance of all methods from the parent classes where the subclass does not define a method with the same signature (i.e. we only force the copy down of methods which are not overridden locally):

```
context Class:
  inv inherit_invariant_methods
    self.parents->forAll(a | a.inheritable_methods()->forAll(b |
        not self.methods->exists(c | c.is_local = true
          and b.invariant_signature(c)) implies
        self.methods->exists(c | c.is_local = false and
          b.invariant_signature(c))))
```

As the UML definition has no notion of renaming, we can add to the earlier constraints *must_contain_local_attributes* and *must_contain_local_methods* by ensuring that attributes in the resolved syntax have the same name as in the abstract syntax:

```
context SemanticMapping::Class:
  inv resolved_attribute_names_match_abstract_syntax
    self.sclass.attributes->forAll(a |
      self.attributes->exists(b |
        a.name = b.name and a.type = b.type.name))

  inv resolved_method_names_match_abstract_syntax
    self.sclass.methods->forAll(a |
      self.methods->exists(b |
        a.name = b.name and a.type = b.type.name and
        a.args->forAll(c | b.args->exists(d |
            c.name = d.name and c.type = d.type.name))))
```

2.1.1 Analysis of UML generalization

Now that we have a formal definition of UML generalization, we feel it is clear that it is unable to cope with even the simple real world variations outlined in section 1.1. For example, UML generalization cannot cope with C++'s qualified naming mechanism as it breaks the *methods_in_at_most_one_parent* constraint; it cannot cope with the left-to-right depth-first inheritance model of Python which is inconsistent with the *inherit_invariant_methods* constraint. Of course, the more exotic class-to-parent relations of section 1.1 are far beyond the scope of this single mechanism.

2.1.2 The <<implementation>> stereotype

The UML does allow for one slight variation on generalization, if the generalization arrow is sterotyped with <<implementation>>. The definition of this sterotype is given as follows [OMG01]:

> Specifies that the child inherits the implementation of the parent (its attributes, operations, and methods) but does not make public the supplier's interfaces nor guarantee to support them, thereby violating substitutability. This is private inheritance and is usually used only for programming implementation purposes.

Initially, it may seem that this sterotype would address at least some of the complaints against the UML that we levelled against it in section 2.1.1. Particularly, it may seem to address the problem of 'selecting' items from a parent class as shown in figure 1(f). Unfortunately, this is not the case: whilst this stereotype does allow a child class to not be a subtype of its parent class, it still forces the child class to inherit all the contents of the parent.

2.2 Realization

Realization is defined in UML thus ([OMG01]):

> [Realization is a] relationship between a specification model element or elements ... and a model element or elements that implement it .... The implementation model element is required to support all of the operations ... that the specification model element declares. The implementation model element must make or inherit its own declarations of the operations ....

In order to model this new type of class-to-parent relationship, we now bring in to play the *SConformanceRel* and *ConformanceRel* classes of figures 2 and 3 respectively. In the UML definition, a conformance relationship states that a child class must provide a definition for each method signature in the parent class:

```
context ConformanceRel:
  query inheritable_methods() : Set(Method)
    Set{}

  query inheritable_attributes() : Set(Attribute)
    Set{}

  query conformable_methods() : Set(Method)
    self.parent.methods

  query conformable_attributes() : Set(Attribute)
    Set{}
```

We then need to add a constraint to figure 3 to ensure that all subclasses in a conformance relationship respect that relationship:

```
context ResolvedSyntax::Class
  inv method_conformance
    self.parents->forAll(a |
      a.conformable_methods()->forAll(b |
        self.methods->exists(c |
          c.invariant_signature(b))))
```

2.2.1 Analysis of UML realization

UML realization tackles a subject with much less real-world variation – conformance – than UML generalization and therefore is less obviously open to criticism than generalization. However, two issues mean that a single definition of realization can never be any more of a success than generalization.

Firstly, the realization definition arbitrarily dictates that only methods must be conformed to in the relationship: one may wish that other elements such as attributes are also involved in the conformance relationship. Although we do not make use of it, our *conformable_attributes* method is the obvious twin of the *conformable_methods* method and, alongside a suitable mirroring of the *method_conformance* constraint, can be utilized in exactly the same way.

The second issue is far deeper and is one to which we propose no solution for in this paper; it is deserving of a far more thorough treatment than we can afford here. Although we have avoided talking of it in these terms, realization is really intended to be 'type conformance': that is, the notion of class A conforming to class B. Unfortunately this is yet another instance of the unfortunate conflation of classes and types in object-orientated programming languages: we should really be talking of class A conforming to the type of class B. It is possible to disambiguate between classes and types in all circumstances where the two are conflated (see [AC96] whose *InstanceTypeOf* function mechanically creates a type from a class) and indeed in some circumstances the two have to be considered separately when classes and types diverge from each other: see section 3.3 for more details.

3 Creating differing class-to-parent relationships

In section 2.1 we presented a formal semantic definition of UML generalization and were able to justify our earlier examples where we asserted that it was unable to cope with variations in real-world programming languages. Thus, having shown that the generalization mechanism is insufficient, we would like to be able to present a scheme which is sufficient to cope with the examples we presented. However, we showed in section 1.1 that there is more to life than subclassing and we would therefore also like to be able to cope with some of the other more exotic class-to-parent relationships which exist.

Starting first with subclassing, it is our contention that no one mechanism could cope with the variations on a theme that exist in the real world: it is not just that some programming languages have more or less features in this area than others, but that some languages have conflicting features (e.g. invariant versus covariant method overriding). We therefore need a mechanism which allows us to create differing versions of the subclassing arrow: different modelling languages can then create a subclassing arrow which suits their needs precisely. In a wider sense we also need to be able to create different arrows (and, as with subclassing, different types of these different arrows) to deal with other class-to-parent relationships such as renaming.

Our solution to this problem is simple. In order to cope with differing types of the same arrow, we isolate a number of characteristics of the varying class-to-parent relationships as OCL constraints, and allow them to be combined in differing ways. To deal with different types of arrows, the various subclasses of *SParentClassRel* come in to play: although we defined them up front in the interests of brevity, these subclasses can clearly be introduced independently of each other.

From a practical point of view, our differing constraints can be put into

	UML	C++	Python	Java	Eiffel
Methods in at most one parent if not overridden locally	•	◦	◦	◦	◦
Attributes in at most one parent	•	◦	◦	◦	◦
No overriding attributes	•	◦	◦	◦	•
Multiple parents	•	•	•	◦	•
Invariant method args	•	•	◦	•	◦
Covariant method args	◦	◦	◦	◦	•
Contravariant method args	◦	◦	◦	◦	◦
Conformance	•	◦	◦	•	◦
Renaming	◦	◦	◦	◦	•
Left to right depth first subclassing	◦	◦	•	◦	◦

Table 1: How the varying constraints can be combined

packages and via package extension can be merged as appropriate into the abstract syntax and resolved syntax packages. A summary of the constraints and how they can be used in differing combinations to get a representative (though note, not necessarily complete) chunk of the semantics of some well known languages is show in table 1 (a full circle in a column meaning that a particular programming language needs that feature, a hollow circle that it does not).

3.1 Python style left-to-right depth-first subclassing

In figure 1(a), we showed that Python's left-to-right depth-first inheritance mechanism would retrieve the method m from class A for class C. Our constraint to force this style of inheritance – which is mutually exclusive with our other constraints for inheritance – is as follows:

```
context Class:
  inv inherit_methods_left_to_right_depth_first
    Sequence{1 .. self.parents->size}->forAll(a |
      self.parents->at(a).inheritable_methods()->forAll(b |
        (not self.methods->exists(c |
          c.is_local = true and
          b.invariant_signature(c)) implies
          not Sequence{1 .. self.parents->size}->forAll(d |
          d < a and
          self.parents->at(d).inheritable_methods()->forAll(e |
            e.invariant_signature(b)))
        ) implies self.methods->exists(c |
          c.is_local = false and b.invariant_signature(c))))
```

Essentially, we work left to right through the classes' parent classes and check that for each method, if there is not a locally defined method then the parent classes method must exist in the local class unless a parent class from earlier in the sequence of parents also defines a method with the same signature.

3.2 Subclassing with contravariant method overriding

Many programming languages, such as Simula and C++, force methods in subclasses to have the same type as in the subclasses they inherit from. It is possible, however, to safely allow an overriding method to have a different type than that which it overrides providing that its argument types are parent classes of its parents argument types and it returns a subtype (i.e. the argument types vary contravariantly, the return type covariantly. We discuss this in more detail in section 3.3).

In order to model this, we first need to add a query to the *Method* class that checks whether its signature varies contravariantly compared to a given methods a. Note that any method with an invariant signature is also said to have a contravariant signature but not vice versa in our definition.

```
context Method:
  query contravariant_signature(a : Method) : boolean
    self.name = a.name and
    self.type.is_sub_type(a.type) and
    self.args->size = .args->size and
    Sequence{1 .. self.args->size}->forAll(i |
      self.args->at(i).is_super_type(a.args->at(i)))
```

We then need to add a constraint which forces the inheritance of all methods which are not overridden by a local contravariant method.

```
context Class:
 inv inherit_contravariant_methods
    self.parents->forAll(a | a.methods->forAll(b |
        not self.methods->exists(c | c.is_local = true and
          b.contravariant_signature(c)) implies
        self.methods->exists(c | c.is_local = false and
          b.invariant_signature(c))))
```

3.3 Subclassing with covariant method overriding

Some programming languages – most notably Eiffel – go further than simply allowing contravariant argument types, and allow methods to vary their argument types covariantly. Unsurprisingly, the necessary query and constraint look very similar to those in section 3.2.

```
context Method:
  query covariant_signature(a : Method) : boolean
    self.name = a.name and
    self.type.is_sub_type(a.type) and
    Sequence{1 .. self.args->size}->forAll(i |
      self.args->at(i).is_sub_type(a.args->at(i)))

context Class:
 inv inherit_covariant_methods
    self.parents->forAll(a |
      a.methods->forAll(b | not self.methods->exists(c |
```

```
      c.is_local = true and  b.covariant_signature(c))
    implies self.methods->exists(c |
      c.is_local = false and b.invariant_signature(c))))
```

This flexibility comes with a price: in situations where a method argument is of type *Self* (in Eiffel this is the *Like Current* type) one can create subclasses which are not necessarily subtypes of the superclass (see [CHC90] for the classical argument of this fact, and [Cas95, AC96, Bru96, Tai96] for more gentle introductions to the subject). Thus it is no longer true that our *SubclassRel* class should have the *is_subtype* relation set to true. Covariant method overriding opens up a whole new can of worms for which the UML can provide no solution whatsoever, and that our simple framework cannot handle satisfactorily.

3.4 Renaming

To accommodate the renaming of figure 1(e) in our framework is very simple. We now make use of the *Renaming* class. This class contains two sequences of strings *from* and *to*, which are of equal length and record renamings. We define the necessary methods thus:

```
context RenamingRel:
  query inheritable_methods() : Set(Method)
    self.parent.methods->collect(a |
      a.rename(self.from, self.to))

  query inheritable_attributes() : Set(Attribute)
    self.parent.attributes->collect(a |
      a.rename(self.from, self.to))

  query conformable_methods() : Set(Method)
    self.parent.methods->collect(a |
      a.rename(self.from, self.to))

  query conformable_attributes() : Set(Attribute)
    self.parent.attributes->collect(a |
      a.rename(self.from, self.to))
```

We do not provide a definition of the rename query here – the intention is that it should return a renamed copy of the modelling element in question. It is not possible in the current OCL definition to do this (though we note that it is rumoured that future OCL versions will allow object creation); it is possible via several extra associations and constraints in the semantic mapping to precalculate the renamings but in the interests of brevity we do not give those here as they add nothing to our understanding of the core problem.

A further complication when renaming is that of preserving the subsumption property (i.e. where an object of a subclass A of B may be viewed as an object of type B). When renaming is not an issue, this scheme is most simply achieved by working with the names of model elements. There are two main ways of achieving the desired effect in our model: by creating a link between the element in the subclass and the element in the superclass; by using the *from* and *to* attributes in the *RenamingRel* to calculate element access.

3.5 Select

To model the inheritance (or, as we prefer to call it to avoid confusion, the 'selection') of specific elements from one class into another as in figure 1(f), we introduce the *Select* class. It very simply allows a class to inherit specific named elements (the names being contained in the *select* attribute) from another class. As this is in no way a subtyping relationship, there are no methods or attributes for the class doing the selection to conform to, so we can make use of the default behaviour of the conformance methods.

```
context SelectRel
  query inheritable_methods() : Set(Method)
    self.parent.all_methods()->select(a |
      self.select->exists(b | b.name = a.name))

  query inheritable_attributes() : Set(Attribute)
    self.parent.all_methods()->select(a |
      self.select->exists(b | b.name = a.name))
```

4 Conclusions

By showing different real-world class-to-parent relationships, defining a modelling semantics for UML generalization and UML realization we have been able to show that these concepts are insufficient to model many programming languages in wide-spread use today. We then developed a flexible mechanism which allows the swift and precise creation of variant class-to-parent relationships, as well as completely new types of class-to-parent relationships, when creating new modelling languages.

Nevertheless there is still a great deal of work to be usefully done. Most notably, in sections 2.2.1 and 3.3 we touched upon the problems of conflating the notions of class and type in UML. The separation of these concepts would facilitate relationships between not just classes and classes, but classes and types and would allow the modelling of classes of problem completely out of reach of the current definition. With respect to the work presented in this paper, we believe there are a number of areas to explore further: a mechanism for specifying the valid combination of our various constraints etc. which would prevent contradictory statements being combined; whether operational semantics can be utilized in a similar manner; the exploration of more exotic types of class-to-parent relationships e.g. those involving bound polymorphism; the applicability of our results to other containment mechanisms such as packages.

Acknowledgments: The authors would like to thank Biju Appukuttan for his input. This work was funded by a grant from Tata Consultancy Services.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[Bru96] Kim Bruce. Typing in object-oriented languages: Achieving expressibility and safety. Technical report, Williams College, 1996.

[Cas95] Giuseppe Castagna. Covariance versus contravariance: Conflict without a cause. In *ACM Transactions on Programming Languages and Systems*, pages 431–447. May 1995.

[CEK02] Tony Clark, Andy Evans, and Stuart Kent. Engineering modelling languages: A precise meta-modelling approach. In Ralf-Detlef Kutsche and Herbert Weber, editors, *FASE 2002*, volume 2306 of *Lecture Notes in Computer Science*, 2002.

[CHC90] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *Seventeenth Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[DN66] Ole-Johan Dahl and Kristen Nygaard. An algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall International, 1992.

[OMG97] Object constraint language specification, 1997. `ftp://ftp.omg.org/pub/docs/ad/97-08-08.pdf`.

[OMG01] OMG. OMG unified modeling language specification 1.4, 2001. `http://www.omg.org/cgi-bin/doc?formal/01-09-67`.

[Öve98] Gunnar Övergaard. A formal approach to relationships in the Unified Modeling Language. In *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*, 1998.

[SO99] David Stoutamire and Stephen Omohundro. Sather - language specification, 1999. `http://www.gnu.org/software/sather/Doc/specification.html`.

[Str87] Bjarne Stroustrup. Multiple inheritance for C++. In *European Unix Users Group Conference '87*, pages 189–207, 1987.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[Tai96] Antero Taivalsaari. On the notion of inheritance. In *ACM Computing Surveys*, volume 28, pages 438–479. September 1996.

[vR01] Guido van Rossum. Python 2.2 reference manual, 2001. `http://www.python.org/doc/2.2/ref/ref.html`.