# The Specification of a Reference Implementation for the Unified Modelling Language

Tony Clark*
Andy Evans†
Stuart Kent‡

March 3, 2000

## 1 Introduction

The Unified Modelling Language (UML) [1] is a language for modelling object systems based on a unification of Booch, Runbaugh and Jacobson's popular object-oriented modelling methods. It is rapidly emerging as a de facto standard for the modelling of such systems. The UML provides many standard diagrammatic modelling techniques representing static and dynamic system information.

Currently, UML version 1.3 is defined a collection of UML meta-models (a definition of UML in a subset of itself). Each meta-model describes the structure of part of the language and provides a collection of well-formedness constraints. The semantics of the language are given in informal text. The definition is unsatisfactory because it is partial, unstructured and introduces questions relating to the soundness of such a meta-circular language definition.

Under the auspices of the precise UML (pUML) group we have proposed a restructuring and semantic definition of the current version of UML (1.3) [5]. This work aims to provide a modular definition of the semantics that can support a wide variety of profiles. There are number of components to this definition: a kernel library, which provides a collection of modelling concepts essential to the building of UML profiles, an extension mechanism for constructing profiles as extensions of the kernel library or other profiles, and a constraint language for expressing invariant properties of UML models. It is intended that once completed, the kernel and associated domain specific profiles will provide a standard reference library for the UML.

In this paper, we consider the problem of building a reference implementation (RI) to support the proposed semantics structure. The purpose of the RI is to

*Department of Computer Science, Kings College, London
†Department of Computer Science, University of York, UK
‡Department of Computing, University of Kent, UK

enable the automated exploration of the semantics and to permit tool vendors to verify tool compliance. Using the RI, a vendor should be able to show that the abstract syntax of UML models processed by their tool and the semantics of those models comply with a ratified profile. The tool vendor may also show that the concrete syntax of the tool complies with a standard profile or use the features of the reference library to define a new concrete syntax and establish its semantic credentials.

As a pre-requisite to constructing the reference implementation (RI), we identify a number of key features that it must support. These include:

- A common model format: a meta-modelling sub-language which enables instances of profiles to be specified and dynamically created within the tool. We show how this can be reduced to a simpler semantic model called the meta-kernel which can be implemented more easily.

- An implementation of OCL: which encodes the semantic properties of OCL within the tool.

- An architecture for structuring and modelling profile semantics, and rules for translating meta-model descriptions of profiles into the common model format.

We will show that these features can be used to construct standard profiles in the form of packages. Thus, it will be possible to test whether a given model, expressed in the common model format, is consistent with a particular profile. Tool compliance can be established by translating snapshots of the tool states (or sequences of snapshots) to the common model format and then showing that the profile's constraints (expressed as OCL expressions) are satisfied.

The aim of this paper is to specify the RI. The specification defines the essential features of the RI and establish the criteria for satisfaction. The paper is structured as follows: section 2 analyses the current state of the UML 1.3 definition and identifies a list of requirements for the RI; section 4 describes a formal basis for the semantics to be supported by the RI; section 5 defines the meta-modelling sub-language used to define UML profiles; section 6 defines an example UML profile and shows how a tool might use the RI to check it; section 7 describes the RI in terms of how it will be used by the UML community; finally, section 8 outlines our future plans for implementing the RI.

# 2 Requirements of the RI

The purpose of the RI is to enable the automated exploration of the semantics of UML and to support automatic testing of conformance of CASE tools to the language definition. Examples of semantics-oriented tasks are: model simulation or (partial) execution; checking that different views on a model (class diagrams, invariants, state diagrams, sequence diagrams, etc.) are consistent with one another; checking that the behaviour of a superclass is preserved in a subclass;

and so on. In the following section we examine a number of features of UML 1.3. and argue that it is currently not suitable as a basis for constructing a reference implementation for UML. We then propose how these limitations can be addressed.

## 2.1 Limitations of UML 1.3

### 2.1.1 Semantic Foundation

To define a semantics requires (at least) an abstract syntax, a semantics domain and a relationship between the two to be defined (see [2] for more details). In the current UML semantics document, the abstract syntax is defined using a meta-model approach (class diagrams + OCL constraints), the semantics domain is English, and the relationship between the two is also expressed in English. Thus the semantics document is not a precise or formal description of the language.

Unfortunately, for a machine to process a language, that language must be defined precisely. If it is to perform semantics-oriented tasks, then its semantics must be defined precisely. Therefore, a pre-requisite for developing an RI for UML is the existence of a precise description of the UML semantics.

### 2.1.2 Multiple Modelling Languages

The current version of UML provides a large number of modelling facilities. Because of this, there is a danger of becoming overloaded with too many concepts, many of which are not widely used except in very specific circumstances. For example, the definition of class diagrams (static model elements) supports a wide variety of facilities for expressing constraints. In practice, these facilities are rarely used, or may be used inappropriately.

There has already been some attempt to architect the meta-model into packages so that pieces are brought in stage by stage. The limited power of the current UML package imports has meant that this is relatively coarse grained. It is not possible to define pieces of a class or in one package, then add more pieces in another package that imports it (and that adding might be by importing another package that supplies other pieces of that class). Thus, when a concept is introduced in a particular package (e.g. the concept of Operation in the core package), one is forced to introduce all the different facets of that concept, even if they are not relevant at that stage.

In practice, it is very important to be able to construct different semantic definitions for specific modelling domains. Some examples that have already been proposed for UML are: real-time, business and networking domains, among others. This has led to the notion of a UML profile[3]: a semantics definition which is specifically aimed at supporting a single modelling domain.

Ideally, an RI should be flexible enough to support profiles. Furthermore, it needs to provide extension mechanisms for constructing profiles from pre-existing ones, thus enabling profile reuse. For example, it should be possible to have a core or kernel profile (introducing common UML semantic concepts:

classes, associations, operations, etc.), then for each profile to import from the core, adding further concepts and placing restrictions on the use of imported concepts. One way in which this can be achieved is by providing a

#### 2.1.3 Constraint Language

In addition to its many diagrammatical notations, the UML semantics definition currently provides a textual language, the Object Constraint Language (OCL), which is used to describe constraint on UML models. These constraints include: invariants, pre- and post-conditions and guards. An OCL constraint is applied in the context of a specific class instance using basic predicate logic and set theory operators (for a detailed description of the language see [4]).

Clearly, in order that the RI can be used to explore the effect of constraints on the logical properties of a UML model, the RI needs to support a constraint language. However, the current definition of OCL suffers from a number of limitations which make it difficult to support at present:

- It does not have a precise semantics, i.e. there is no means of taking an OCL expression and rigorously evaluating whether or not a particular model instance satisfies the implied constraint.
- The language uses a rather non-intuitive syntax, which is still not entirely resolved.
- The language supports a very rich set of expressions. This richness mitigates against providing OCL with a precise semantics.

Thus, some means must be found to provide a simpler, more precise definition of OCL, before it can be considered suitable for implementation in an RI.

### 2.2 Summary of Requirements

In order to support the development of an RI, the following are required:

- A precise semantics definition of UML.
- A modular architecture for the semantics, which can support the incremental definition of profiles.
- A semantic definition of a simple (OCL) like constraint language

Finally, for a the RI to support the above semantics, a mechanical means must be found for incorporating their definition in a tool, and for calculating whether or not constraints are satisfied by instances of a particular UML model.

## 3 A Semantic Architecture for UML

This section briefly summarises recent work done to provide a semantics architecture for UML, which supports the precise definition of UML profiles. This work was presented as a response to the OMG's request for information (RFI) regarding the next major release of UML (version 2.0) by the precise UML group [5].

The semantics architecture presented in [5] is based upon the use of metamodelling to provide a precise denotational description of UML concepts. The definition is structured into packages, based on a kernel library of language definition tools and components. A profile is a definition of a language that may specialise and/or extend other profiles, and incorporate components from the kernel library. Figure 1, shows the general architecture.

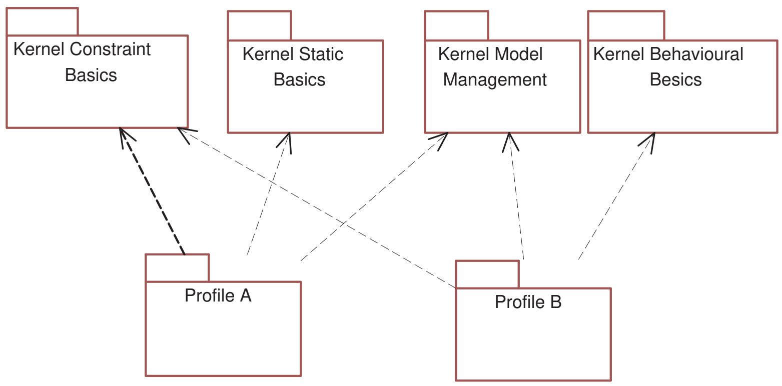


Figure 1: Profile architecture

The kernel library consists of a number of basic packages containing fundamental UML concepts. These include:

Static basics - generalised constructs for modelling the static properties of systems.

Constraint basics - constructs relating to the expression of constraints.

Dynamic basics - constructs for modelling the behaviour of systems.

Model management basics - general mechanisms for extending and specialising the components of the language.

As shown, profiles are extensions of these basic packages. An extension mechanism, similar to that proposed in the Catalysis method [6] is used to copy elements from one package into another, whilst also permitting extension of their features.

Each profile is organised into abstract syntax, semantics domain and a satisfaction/denotation relationship between the two (see Figure 2). Both abstract syntax and semantics domain may have many concrete representations.

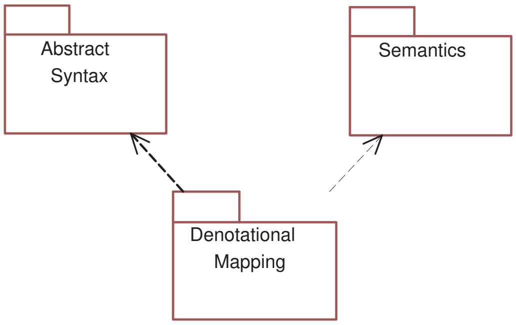


Figure 2: Profile semantics

## 3.1 Meta-modelling sub-language

An essential component of the proposed architecture is a *meta-modelling sub-language*. This is used to characterise all aspects of a profile and the kernel library. It provides all the facilities necessary to write profiles, including: simple class diagrams, a simple constraint language, packages (to represent models), an enhanced version of package imports, and a notion of package realisation.

Like any other profile, the meta-modelling sub-language imports a number of concepts from the kernel library (see figure 3).

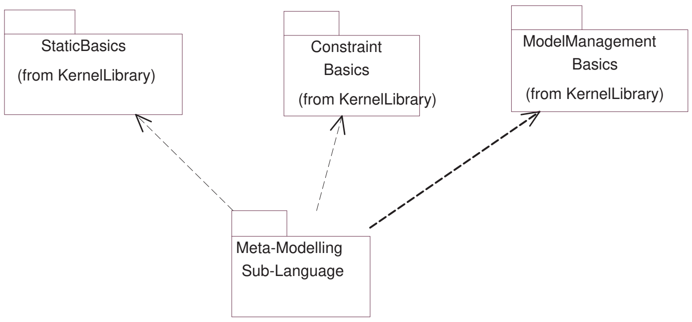


Figure 3: The meta-modelling sub-language package

As an example, figure 4 shows some of the classes that might belong to the abstract syntax of the sub-language. These deal with two fundamental static modelling concepts: classifiers and attributes. Note that there will be many other classes defined in the sub-language, including packages, associations, generalization and constraints. However, these are omitted for brevity.

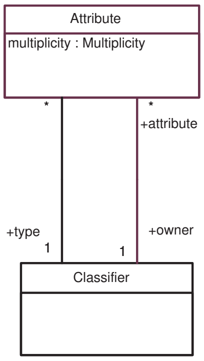


Figure 4: Abstract syntax fragment of meta-modelling sub-language

A number of OCL constraints are required in order to ensure that the concepts in the abstract syntax are well-formed. In this example it is required that attributes belonging to a classifier have unique names:

```
c : Classifier
c.attributes -> forall(a1, a2 | a1.name = a2.name implies a1 = a2)
```

The semantics domain of the meta-modelling sub-language is described by the class diagram shown in figure 5. This represents the values that denote the meaning of the constructs in the abstract syntax package. For example, a classifier is denoted by a collection of objects, each of which conform to the various features of the classifier (e.g. attributes). There are no well-formedness constraints required in this package.

Finally, both the abstract syntax and semantic packages will be imported by the mapping package, as shown in figure 6.

Associations between the classifiers in the abstract syntax package and those in the semantics domain describe the denotational relationship between the various language constructs. A number of additional OCL constraints are required. The first constraint requires that values of an attribute are objects of the type of the attribute:

```
a : Attribute
a.instances -> forall(al | al.value.classifier = a.type)
```

Secondly, it is required that each object has attribute slots corresponding to the attributes of its classifier:

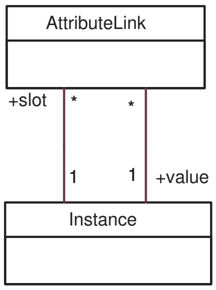


Figure 5: Semantic domain of meta-modelling sub-language

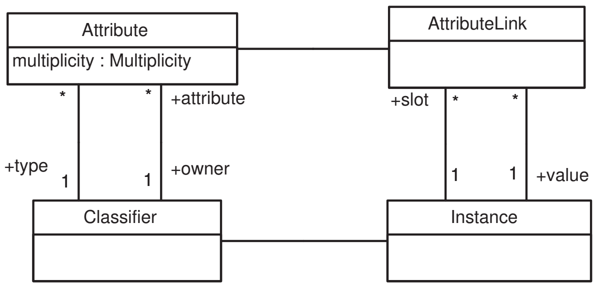


Figure 6: Mapping between abstract syntax and semantic domain

```
o :  Object
o.slots.attribute -> includesAll(o.classifier.attributes)
```

This example, though short, illustrates the general approach used to construct a precise description of the meta-modelling sub-language. It is important to note that if a precise description of the meta-modelling sub-language can be given, then any profiles built using the language will be reducible to well-defined expressions in this language.

As an example, consider the object diagram (snapshot) shown in figure 7, which describes a fragment of a profile, written using the meta-modelling sub-language. This profile contains two classifiers, Operation and Parameter. Operation has an attribute, parameter, whose type is the class Parameter. Their instances are represented by instances of Instance and AttributeLink classifiers.

As we will discuss in the next section, the meta-modelling sub-language is of central importance to the implementation of the RI for this ability to represent the elements of any profile using a common set of concepts.

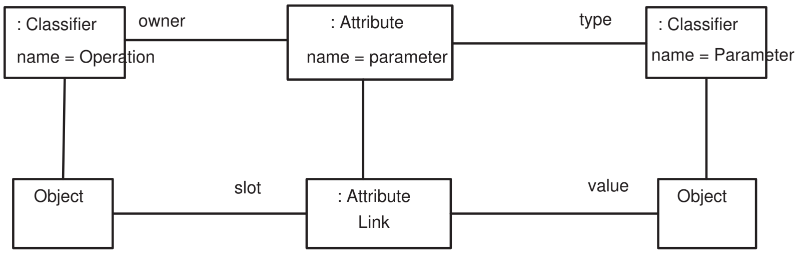


Figure 7: Example of profile snapshot

# 4 The Meta-Kernel Calculus

As discussed in section 2, the purpose of the RI is to enable the implementation and manipulation of different UML profiles. This goal can be elegantly achieved if the RI *implements the meta-modelling sub-language*. If the semantics of the meta-modelling profile are built into the RI, the RI can assist with exploring the properties of the profile being defined by looking at particular examples of expressions of that profile, their mappings into the semantics domain, and so on.

Whilst it would be possible to directly implement the meta-modelling sub-language described above, there are a number of disadvantages to this approach. Firstly, the full meta-modelling sub-language will be quite large (although not as large as the current UML semantics). Significant work will therefore be required to implement the language. Secondly, the implementation will be inflexible: if the meta-modelling sub-language is changed, modifications would be required of the RI. For example, if one wanted to change the mechanism for dealing with profile extension, a major rewrite would be necessary. Finally, any attempt at giving an external definition to the language (for example using set theory and predicate logic) would also be lengthy. Each concept in the sub-language definition would have to be mapped to an expression in an appropriate mathematical language.

One approach which overcomes all these problems is to provide a semantic definition for a meta-modelling sub-language that supports a reduced set of language concepts. This idea is similar to a reduced instruction set, where large instructions are broken down into a number of very small, general purpose instructions built from a restricted operation set. In the case of the UML semantics, the aim is to define rules for translating each concept or expression in the meta-modelling sub-language into a restricted set of concepts in a simpler semantic model. The advantage of this approach is its great flexibility, simpler semantic definition, and generality. In particular, the approach makes the task of grounding the semantics and its implementation much easier due to the reduced number of language constructs.

$$
\begin{array}{lll}
e & ::= & \text{expressions} \\
 & a & \text{constants} \\
| & v & \text{variables} \\
| & \{e, \ldots, e\} & \text{sets} \\
| & [v_i = e_i{}^{i \in 1,\ldots,n}] & \text{objects} \\
| & e.v & \text{field reference} \\
| & ee & \text{operation invocation} \\
| & \lambda v.e & \text{method} \\
| & e \rightarrow \text{iterate}(v\ v = e|e) & \text{iteration} \\
| & \text{if } e \text{ then } e \text{ else } e & \text{conditionals} \\
| & e \rightarrow \text{including}(e) & \text{set extension}
\end{array}
$$

Figure 8: Meta-Kernel Calculus Syntax

The simple semantic language adopted here is based on a very small set of concepts: objects and slots (relations between objects). This is because any concept can be viewed as a collection of related objects. For example, classifiers, attributes, packages, instances and even profiles themselves can all be thought of as objects. So, an object representing a classifier would be related to other objects representing its attributes, and so on. Because this language is suitable for describing concepts at all levels of abstraction, we have called it the *meta-kernel language*.

In the following section a definition of the semantics of the meta-kernel is given. This is given using set theory and predicate logic (as opposed to a meta-model). A meta-model definition would eventually reach a point where features of the language were defined in terms of circular definitions, thus leading to infinite regress.

## 4.1 Meta-Kernel Calculus Syntax

The Meta-Kernel Calculus syntax is defined in figure 8. Expressions in the calculus denote values from the following categories: atomic constants; objects; sets of values.

Atomic constants include integers, booleans and strings. Objects are simple record structures consisting of field names and field values. All the field names in an object must be distinct. Sets are builtin to the calculus, but bags and sequences are not. We claim that sets are the underlying representation for UML collections and that all other types of collection can be expressed in terms of objects and sets.

A core OCL expression syntax is builtin to the calculus. The core abstracts the details of OCL operations such as conjunction and disjunction. All operations are represented uniformly as operators that are applied to operands. All operators are defined to take a single operand. Multiple arguments are simply packaged up into a single object. OCL operators such as `and` are defined as

$$
\begin{aligned}
&a\delta = a \\
&v\delta = \delta(v) \\
&\{e_i^{i\in 1,\ldots,n}\}\delta = \{e_i^{i\in 1,\ldots,n}\delta\} \\
&[v_1 = e_i{}^{i=1,\ldots,n}]\delta = [v_i = (e_i(\delta\backslash\{v_i^{i\in 1,\ldots,n}\}))^{i\in 1,\ldots,n}] \\
&(e.v)\delta = (e\delta).v \\
&e_1 e_2\delta = (e_1\delta)(e_2\delta) \\
&(\lambda v.e)\delta = \lambda v.(e(\delta\backslash\{v\})) \\
&(e_1 \to \mathrm{iterate}(v_1\ v_2 = e_2|e_3))\delta = (e_1\delta) \to \mathrm{iterate}(v_1\ v_2 = (e_2\delta)|e(\delta\backslash\{v_1, v_2\})) \\
&(\text{if } e_1 \text{ then } e_2 \text{ else } e_3)\delta = \text{if } e_1\delta \text{ then } e_2\delta \text{ else } e_3\delta \\
&(e_1 \to \mathrm{including}(e_2))\delta = (e_1\delta) \to \mathrm{including}(e_2\delta)
\end{aligned}
$$

Figure 9: Substitution into Meta-Kernel Expressions

builtin operators (see below).

The core constructs of OCL are therefore: constants, variables, field reference, operation invocation, iteration, conditionals and set extension. We claim that all other OCL constructs can be defined as either builtin operators of the calculus or syntactic sugar (see below). OCL expressions denote objects of an appropriate OCL expression type and therefore do not require a new value domain. The class of OCL expressions are those calculus expressions that denote boolean constants.

Figure 9 defines the substitution of values for variables in Meta-Kernel Calculus expressions. A substitution $\delta$ is a sequence of variables and expressions $[e_1/v_1, \ldots, e_n/v_n]$. A substitution is applied to an expression $e\delta$; all free occurrences of each variable in $\delta$ are simultaneously replaced with the corresponding expression. Variables in a set $V$ are removed from a substitution by $\delta\backslash V$.

## 4.2 Meta-Kernel Calculus Semantics

The semantics of the Meta-Kernel Calculus is defined by a congruence relation (reflexive, transitive, associative and equality of all sub-expressions implies equality of composite expressions) on expressions in figure 10. If $e_1 = e_2$ then the two expressions denote the same value. The rest of this section describes key features of the semantics.

An object is a collection of definitions. The definitions are mutually recursive and shadow any definitions for variables in scope with the same name. The variable 'self' may be used in each of the slot value expressions to refer to the object. The semantics of field selection (axiom REF) shows that the fields and 'self' are substituted into the field value when it is extracted from an object.

Method invocation is defined by the axiom APP. A method occurring in an object must be referenced before it can be invoked. When a method is extracted from an object using REF, the values of the fields and the value of 'self' will be subtituted into the method body. In this way methods can access

$$[v_i = e_i{}^{i\in 1\ldots,n}].v_j = (e_j[\mathrm{self}.v_i/v_i{}^{i\in 1,\ldots,n}])[[v_i = e_i{}^{i\in 1,\ldots,n}]/\mathrm{self}]\ j \in 1,\ldots,n \text{ REF}$$

$$(\lambda v.e_1)e_2 = e_1[e_2/v] \text{ APP}$$

$$\frac{e'[e_i/v_1, e'_i/v_2] = e'_{i+1}\ i \in 1,\ldots,n}{\{e_i^{i\in 1,\ldots,n}\} \rightarrow \mathrm{iterate}(v_1\ v_2 = e'_1|e') = e'_{n+1}} \text{ ITER}$$

$$(\text{if true then } e_1 \text{ else } e_2) = e_1 \text{ IFTRUE}$$

$$(\text{if false then } e_1 \text{ else } e_2) = e_2 \text{ IFFALSE}$$

$$\{e_i^{i\in 1,\ldots,n}\} \rightarrow \mathrm{including}(e) = \{e, e_i^{i\in 1,\ldots,n}\} \text{ SETINC}$$

Figure 10: Meta-Kernel Expression Equivalence

all components of the object.

Execution of an iterate expression is defined by rule ITER. The result of iterate is the value accumulated in $v_2$. This value will be atomic, an object or a collection of values. We claim that all other OCL values can be modelled using these basic value types.

Using iteration expressions and a notion of set extension (axiom SETINC) defined in figure 10 we claim that all OCL set operations can be defined as sugar. Set extension simply adds a value to an existing set.

## 4.3 Calculus Extensions

The Meta-Kernel Calculus provides a sub-set of OCL and support for very simple objects. The calculus must be extended with extra features in order to support the whole of OCL. We will extend the calculus in the following ways:

- Syntactic sugaring does not change the underlying semantics of a language; new types of expression are defined by giving a translation to the basic calculus.
- Builtin methods extend the syntax of the calculus with new operators that can be applied to operands. Each builtin method must have an associated rule like APP that defines what happens when it is applied.

### 4.3.1 Iteration Expressions

The 'iterate' expression is builtin to the Meta-Kernel Calculus. All other types of iteration expression are *sugar* and can be expressed using basic 'iterate'. Figure 11 gives a suitable translation.

$$
\begin{aligned}
&[\![e_1 \to \mathrm{forAll}(v_1 \mid e_2)]\!] = \\
&\quad [\![e_1]\!] \to \mathrm{iterate}(v_1 v_2 = \mathrm{true} \mid \mathrm{if}\ [\![e_2]\!]\ \mathrm{then}\ v_2\ \mathrm{else\ false}) \\
&[\![e_1 \to \mathrm{exists}(v_1 \mid e_2)]\!] = \\
&\quad [\![e_1]\!] \to \mathrm{iterate}(v_1 v_2 = \mathrm{false} \mid \mathrm{if}\ [\![e_2]\!]\ \mathrm{then\ true\ else}\ v_2) \\
&[\![e_1 \to \mathrm{select}(v_1 \mid e_2)]\!] = \\
&\quad [\![e_1]\!] \to \mathrm{iterate}(v_1 v_2 = [\![e_1]\!] \mid \mathrm{if}\ [\![e_2]\!]\ \mathrm{then}\ v_2\ \mathrm{else}\ v_2.\mathrm{excluding}(v_1)) \\
&[\![e_1 \to \mathrm{reject}(v_1 \mid e_2)]\!] = \\
&\quad [\![e_1]\!] \to \mathrm{iterate}(v_1 v_2 = [\![e_1]\!] \mid \mathrm{if}\ [\![e_2]\!]\ \mathrm{then}\ v_2.\mathrm{excluding}(v_1)\ \mathrm{else}\ v_2) \\
&[\![e_1 \to \mathrm{collect}(v_1 \mid e_2)]\!] = \\
&\quad [\![e_1]\!] \to \mathrm{iterate}(v_1 v_2 = \{\} \mid v_2.\mathrm{including}([\![e_2]\!]))
\end{aligned}
$$

Figure 11: Translation of Iteration Expressions

$$
\begin{aligned}
&[\![S \to \mathrm{size}]\!] = [\![S]\!] \to \mathrm{iterate}(x\ i = 0 \mid i + 1) \\
&[\![S \to \mathrm{includes}(o)]\!] = [\![S]\!] \to \mathrm{exists}(x \mid x = [\![o]\!]) \\
&[\![S_1 \to \mathrm{union}(S_2)]\!] = [\![S_2]\!] \to \mathrm{iterate}(x\ s = [\![S_1]\!] \mid s \to \mathrm{including}(x)) \\
&[\![S_1 \to \mathrm{intersection}(S_2)]\!] = [\![S_2]\!] \to \mathrm{collect}(x \mid [\![S_2]\!] \to \mathrm{includes}(x))
\end{aligned}
$$

Figure 12: Translation of Set Operations

#### 4.3.2 Set Expressions

The Meta-Kernel Calculus provides a builtin operator 'including' that is used to construct new sets from existing ones. All other set operations are sugar and can be constructed using 'including' and 'iterate'. Figure 12 defines some of the translations to show how this is achieved. Note that bags and sequences are viewed as being higher-level structures constructed using basic sets and objects.

A useful set operation is *transitive closure*. If the value of a field $v$ in an object is a set $S$ then it is useful to be able to transitively follow the field $v$ from the elements of $S$. This is defined as syntactic sugar as follows:

$$[\![e.v^*]\!] = [\![e]\!].v \to \mathrm{union}([\![e]\!].v \to \mathrm{iterate}(o\ v = \{\} \mid v \to \mathrm{union}(o.v^*)))$$

#### 4.3.3 Builtin Methods

Builtin methods are used to extend the calculus with useful operations. The extensions take the form of new cases for the definition of the equivalence relationship on calculus expressions given in figure 10.

The calculus is extended with boolean operators `and`, `or` and `not`. Operator has a collection of equivalence rules that define what it means to apply the operator to boolean operands, for example:

$$\mathrm{true\ and\ true} = \mathrm{true}$$

A useful builtin method permits OCL expressions to be viewed as objects. The rule is as follows:

$$e_1.\mathrm{satisfiedBy}([v_i = e_i{}^{i\in 1,\ldots,n}]) = (e_1[\mathrm{self}.v_i/v_i{}^{i\in 1,\ldots,n}])[[v_i = e_i{}^{i\in 1,\ldots,n}]/\mathrm{self}]$$

An OCL expression is satisfied by an object when the result of substituting the object into the expression is equal to 'true'.

#### 4.3.4 Recursive Local Definitions

It is useful to be able to define local method definitions. This is achieved using the following syntactic sugar:

$$\mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 = [v = e_1, v' = e_2].v'$$

where $v'$ is a fresh variable.

## 5 The Meta-Modelling Sub-Language

The Meta-Modelling Sub Language (MMSL) is a meta-circular representation of the essential features necessary to define UML profiles. The language is written in the Meta-Kernel Calculus and consists of a collection of classifiers.

A Classifier in the MMSL is an object that has instances and an OCL invariant that is true for each of the instances. Classification is the process by which the RI determines whether the OCL invariant of a given classifier holds for a given object.

The RI must be able to accept new profiles that define extensions to core UML concepts at the meta-level. Each new profile must be checked against some unform definition of *profile*. This is classification at the meta-level. Although many users of the RI will not require such meta-linguistic machinery, we believe that there are advantages in conceptual parsimony and that everything in the RI must be an object classified by something.

Figure 13 shows the main MMSL classifiers and some important relationships between them. All objects in the RI are classified by a most specific classifier. An unbroken arrow from classifier $C_1$ to classifier $C_2$ defines that $C_2$ is the most specific classifier for $C_1$ *viewed as an object*. A broken line shows inheritance relationships between classifiers *viewed as classes*.

The essential features of the MMSL is given in figure 14. The MMSL includes syntactic sugar for class definitions, package definitions and associations. The translations for these sugared constructs simply produces the appropriate object definitions by adding the appropriate slot information.

Each classifier has slot named 'invariant' that contains the OCL expression that must be satisfied by each of its instances. The invariants are very important because they provide the semantics of the RI. The rest of this section gives an overview of the MMSL invariants.

'Kernel.Classifier' is the basic definition of a classifier. All other classifiers in the RI are specializations. Every classifier has at least an invariant and a set

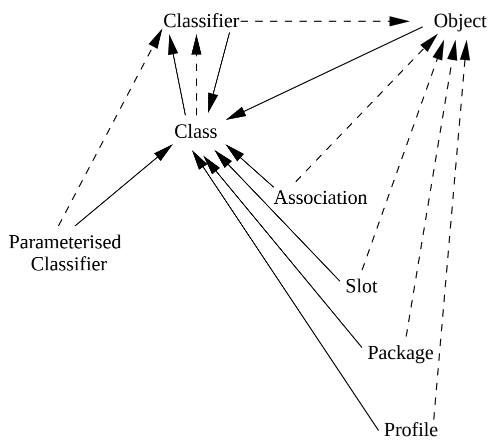


Figure 13: Meta-Modelling Sub-Language Classifiers

of instances. The invariant of 'Kernel.Classifier' requires that the invariant is satisfied by each instance:

$$\text{instances} \rightarrow \text{forall}(i \mid \text{invariant.satisfiedBy}(i))$$

'Kernel.Class' extends 'Kernel.Classifier' with features normally associated with *inheritance*. Every class defines a collection of associations (that turn into instance slots). Every class has a collection of generalizations. A class inherits all associations from its generalizations and all the generalization invariants must be true for the classes instances. The following shows part of the invariant for 'Kernel.Class':

$$\begin{array}{l} \text{self.specializations}^* \rightarrow \text{forall}(s \mid \\ \quad s.\text{instances} \rightarrow \text{forall}(i \mid \\ \quad\quad \text{invariant} \rightarrow \text{satisfiedBy}(i))) \end{array}$$

'Kernel.Object' defines the essential features of an object. Everything in the RI is an object and therefore will have a classifier, an identity label and a collection of slots.

'Kernel.Package' defines the basic package structure that is used to build the RI. Each package is a collection of definitions (expressed as slots). All definitions in the MMSL involve a name and a value. A definition cannot occur outside the context of an object or a package. A package may import the definitions from another package: imports $\rightarrow$ forall($i$ | definitions $\rightarrow$ includesAll($i$.definitions))

# 6 Profiles

A profile is a packaged language definition. The package includes a description of the syntax and semantics of the language defined using the MMSL. A profile is a classifier and therefore, given an instance of the profile expressed as a RI

```
Kernel.Classifier =
  class Classifier extends Kernel.Class
    associations = {
      invariant : OCL,
      instances : Value
    }
  end

Kernel.Object =
  class Object
    associations = {
      classifier : Kernel.Classifier,
      identity : {Integer},
      slots : {Slot}
    }
  end

Kernel.Class =
  class Class extends Kernel.Classifier
    associations = {
      generalizations : {Kernel.Classifier},
      specializations : {Kernel.Classifier},
      name : String,
      package : Kernel.Package,
      associations : {Kernel.Association}
    }
  end

Kernel.Association =
  class Association extends Kernel.Object
    associations = {
      name : String,
      type : Kernel.Classifier
    }
  end

Kernel.Slot =
  class Slot extends Kernel.Object
    associations = {
      name : String,
      value : Value
    }
  end

Kernel.Package =
  class Package extends Kernel.Object
    associations = {
      name : String,
      imports : {Kernel.Package},
      definitions : {Kernel.Slot}
    }
  end
```

Figure 14: The Meta-Modelling Sub-Language

object, we can determine whether or not the object is a correct phrase of the language using RI classification.

## 6.1 Profile Classifier

Figure 15 defines a profile classifier contained in the MMSL. A profile contains packages defining the syntax and semantics of the language. 'Profile' is a meta-class since it specializes 'Kernel.Class'. The instances of 'Profile' are therefore classifiers called *profiles*. Since it is a classifier, a profile has an invariant that is used to define conditions that must hold between the syntactic and semantic components of instances of the profile.

```
Kernel.Profile =
    class Profile extends Kernel.Class
        associations = {
            syntax : Kernel.Package,
            semantics : Kernel.Package
        }
    end
```

Figure 15: MMSL Profile

### 6.2 An Example Profile

Figure 16 defines a simple profile for UML class diagrams as defined by the meta-model in figure 6. The profile, has four main components: the syntax package; the semantics package; the associations; and, the invariant.

The package 'syntax' conforms to the meta-model in figure 4. A tool using the 'ClassDiagram' package to check its syntactic representation of class diagrams would translate its internal representation of classes and associations to instances of 'ClassDiagram.syntax.Classifier' and 'ClassDiagram.syntax.Attribute' respectively.

The package 'semantics' conforms to the meta-model in figure 5. The semantics is used in two ways: a tool can check its own semantic model against that provided by a profile; and, the semantics can be used to prove properties about a UML model fragment. In the case of class diagrams, a tool may represent instances of classes in which case the objects are translated to instances of 'ClassDiagram.semantics.Instance' and 'ClassDiagram.semantics.AttributeLink'.

The class diagram profile defines two associations 'classes' and 'instances'. A tool will check its internal representation against a profile by supplying an instance of the profile. Instances of 'ClassDiagram' consist of a collection of classes (the syntactic part) and a collection of instances (the semantic part).

### 6.3 Using Profiles

Consider a UML CASE tool that supports drawing class diagrams and object diagrams. The tool vendors wish to determine whether or not their representation of classes and instances is correct with respect to the UML standard. Suppose also that the standard for UML class diagrams is defined by the RI profile 'ClassDiagram' defined in figure 16.

The RI will accept as input tool model components translated to instances of known classifiers. The tool can then be used to check whether the instances are classified by standard profiles. The RI makes no distinction between metaclasses, classes or objects and therefore the RI can be used to check meta-models as easily as models or snapshots of models.

Suppose the tool wishes to check a particular class diagram containing two

```
class ClassDiagram extends Kernel.Object
    classifier = Kernel.Profile,
    package syntax =
        class Classifier extends Kernel.Object
            associations = {name : String, attributes : {Attribute}},
        end,
        class Attribute extends Kernel.Object
            associations = {name : String, type : Classifier}
        end
    end,
    package semantics
        class Instance extends Kernel.Object
            associations = {slots : {attributeLink}},
        end,
        class AttributeLink extends kernel.Object
            associations = {value : Instance, name : String}
        end
    end,
    associations = {classes : {syntax.Classifier}, instances : {semantics.Instance}},
    invariant =
        let classifies(c, o) =
            o.slots → forall(s |
                c.attributes → exists(a |
                    s.name = a.name and
                    classifies(a.type, s.value)))
        in
            instances → forall(i |
                classes → exists(c |
                    classifies(c, i)))
end
```

Figure 16: A Simple Profile for Class Diagrams

classes 'Operation' and 'Parameter' (see example in section 3.1). There is a single association from 'Operation' to 'Parameter' named 'param'. In addition the tool has a representation of a Operation 'o' and its parameter 'p'. This is translated to the package shown in figure 17.

# 7 The Reference Implementation

The RI is an implementation of the MKC, the MMSL and a collection of profiles defined using the MMSL. It is intended that the RI serve as a shared resource for the UML community and that it be used to collectively define and experiment with the semantics for UML.

In addition, the RI provides an API that allows users to connect and use the tool to build profiles and check candidate models against profiles. The API is

**package** Example **import** ClassDiagram.syntax, ClassDiagram.semantics
    Operation = [classifier = Classifier, name = "Operation", attributes = {param}],
    paramAtt = [classifier = Attribute, name = "param", type = Parameter],
    Parameter = [classifier = Classifier, name = "Parameter", attributes = {}],
    o = [classifier = Instance, slots = {paramLink}],
    p = [classifier = Instance, slots = {}],
    paramLink = [classifier = AttributeLink, name = "param", value = p],
    profileInstance = [classes = {Operation, Parameter}, instances = {o, p}]
**end**

Figure 17: Example Class Diagram

planned to include an XML interface, a textual language interface (the language used in this paper) and a programming interface in Java.

The RI will be implemented in Java and will support distributed use via the Internet. Users will be able to connect to the RI and use the facilities as part of an applet. The RI will act as a server and maintain the profiles in a database.

The scope of the RI does not extend beyond checking the conformance of UML models in a standard format. However, it does provide the basis for a standard representation for a suite of tools that extend the capabilities of the RI. Tools that are able to interface with the RI will be able to make use of the semantic definitions encoded in profiles to perform sophisticated manipulation of UML models. We would expect the RI to be the basis for tools for: editing; model checking; proof; code generation; reverse engineering.

# 8 Conclusion

This paper has investigated the requirements of a reference implementation (RI) for UML. It has concluded that the current status of the UML definition is presently unsuitable as a foundation for an RI due to its large size, lack of precise semantics and inflexibility. A meta-modelling sub-language was then proposed which overcomes all these limitations. This language can be used to describe the semantics of any UML profile using a relatively small number of concepts. A further reduction in the size of the language was achieved by showing how this language can be translated into a meta-kernel language consisting of slots and links and a simple constraint language.

The meta-kernel language will form the basis for the reference implementation. We have shown that its simple object/links semantics model can be readily implemented in the RI and that it can be used to check the validity of snapshots against profiles. Work will now continue to further develop the tool. This will include work to develop its user interface and its analysis capabilities. For example, the ability to 'model check' profiles would be extremely useful. In the medium to long term, we believe that the semantics approach taken in this paper, and the preliminary work done on its implementation, are a major step towards our broader goals of developing the UML as a precise, adaptable, and scalable modelling language. By providing developers with such tools and techniques, it is hoped that the development of new profiles (i.e. new modelling language) will increasingly become a precise and verifiable process.

## 8.1 Related work and issues

*How does the proposed semantics compare with those of other modelling languages?*: The denotational approach has been used in the definition of many modelling languages and notations. For example, formal specification notations such as Z [7] and CSP [8] use this approach. The novelty of our approach is that we have concentrated on building a meta- semantics model that has the flexibility to be applied in the definition of any UML-like language. Languages such as those above do not have this flexibility, and therefore are not capable of adapting to changes in language requirements. Many other have looked at formalising OO methods and UML [9, 10]. However, we believe that ours is the first comprehensive semantics that can deal with UML profiles and OCL. Recently, Alloy [11] has been proposed by Daniel Jackson as a precise language for object modelling. Alloy also has an associated tool, Alcoa, which permits model checking of alloy expressions. However, the language does not conform to the UML standard, and has a single, unchangeable semantics. Others have provided preliminary work on OCL meta-modelling semantics, including [12, 13]. However, these are not yet complete.

*Isn't the meta-modelling sub-language similar to the MOF?* We have proposed a precise definition of a sub-language of UML for meta-modelling. This is similar to the MOF meta-meta model [14], but defined in a more declarative, logical style. It is also, itself, more suited to a declarative style of meta-modelling as required for UML. We have given clear guidelines for determining what should go in this language, and have based the language on those guidelines. In particular, the language is intended to be small (but not too small), must at least be good enough to describe itself (completely and with semantics), and is targeted at defining the language in a declarative fashion. It also has constructs to support a fine-grained language architecture and development of profiles. In [5] we have also described a richer, more precise and better delineated architecture for language definition than the 4-level meta-model architecture (at least as it is described in the semantics chapter of UML 1.3). We propose that this is used as the reference architecture for defining UML.

*Is the idea of a meta-modelling tool new?*: No meta-case tools have been available for over a decade. Commercial meta-CASE tools such as Toolbuilder and Meta-Edit provide meta-repositories. Such repositories could in theory be propagated with UML profiles. However, they currently don't provide facilities for the semantic analysis, nor do they support OCL.

# References


[1] OMG Unified Modeling Language Specification (1.3), Available from http://www.omg.org, 1999.

[2] A.S.Evans and S.Kent: Meta-modelling semantics of UML: the pUML approach. 2nd International Conference on the Unified Modeling Language. Editors: B.Rumpe and R.B.France, Colorado, LNCS 1723, 1999.

[3] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, A. Wills: Defining UML Family Members Using Prefaces.In: Technology of Object-Oriented Languages and Systems, TOOLS'99 Pacific. Ch. Mingins, B. Meyer (eds.) IEEE Computer Society

[4] Jos. Warmer, A. Kleppe: The Object Constraint Language: Precise Modeling with UML, Addison-Wesley, 1998.

[5] A. Clark, A. Evans, R. France, S. Kent, B. Rumpe: Response to UML 2.0 Request for Information, Available at: http://www.cs.york.ac.uk, 1999.

[6] D.F.D'Souza and A.C.Wills: Objects, Components and Frameworks with UML, Addison-Wesley, 1999.

[7] J.M. Spivey: The Z Notation, Prentice Hall, 1992.

[8] C.A.R Hoare: Communicating Sequential Processes, Prentice Hall, 1985.

[9] R. France, J-M. Bruel, M. Larrondo-Petrie and M. Shroff: Exploring the Semantics of UML Type Structures with Z, Proc. 2nd IFIP Conf. Formal Methods for Open Object-Based Distributed Systems (FMOODS'97), 1997.

[10] A. Evans and T. Clark: Foundations of the Unified Modeling Language, Proc. of the 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, UK, 23-24 September 1997, 1997.

[11] D. Jackson: Alloy: A Lightweight Object Modelling Notation, Available at: http://sdg.lcs.mit.edu/alcoa, 1999.

[12] M.Richters and M. Gogolla, A Meta-model for OCL. 2nd International Conference on the Unified Modeling Language. Editors: B.Rumpe and R.B.France, Colorado, LNCS 1723, 1999.

[13] M. Richters and M. Gogolla: On Formalizing the UML Object Constraint Language OCL, Proc. 17th Int. Conf. Conceptual Modeling ER'98, 1998

[14] The Meta-Object Facility Specification. Available at: http://www.omg.org, 1999.