The Meta-Modeling Language Calculus: Foundation Semantics for UML

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

[1] King's College London
anclark@dcs.kcl.ac.uk
[2] University of York
andye@cs.york.ac.uk
[3] University of Kent at Canterbury
s.j.h.k@ukc.ac.uk

Abstract. The Meta-Modeling Language (MML) is a sub-set of the Unified Modeling Language (UML) that is proposed as the core language used to bootstrap the UML 2.0 definition initiative. Since it is meta-circular, MML requires an external formal semantics in order to ground it. This paper defines the MML Calculus which is used to formally define MML and therefore provides a semantic basis for UML 2.0.

1 Introduction

The Unified Modeling Language [19] is a standardized graphical notation for expressing the structure and behaviour of object-oriented software systems. It is essentially a family of extensible modelling notations. The current UML definition lacks a number of desirable features that are currently being addressed through a co-ordinated effort to define a new version (UML 2.0). These features include enhancing the modularity and extensibility of UML and addressing the notion of UML semantics.

This paper describes the semantics of the MML Calculus which used as the basis for developing the MML meta-modelling language. MML is the basis of a modular semantics-rich method called MMF [7] [5] which is being proposed by the pUML group as a framework for the definition of UML 2.0. MML is a language mainly aimed at meta-modellers who are familiar with UML. This paper deals with foundational semantic issues that enable MML to be a generic meta-modelling language suitable for defining UML 2.0. Features which are outside the scope of this paper include: patterns for meta-modelling; details of inheritance mechanisms; details of class instantiation; details of package extension; details of invariant checking. These issues are dealt with in [5].

The rest of this paper is structured as follows Section 2 defines the MML Calculus syntax and semantics. Section 3 defines the MML language in terms of the MML Calculus. The MML language is textual, each new construct is introduced and translated to the MML Calculus. Section 4 concludes by reviewing MML and describing the future directions of this work.

2 The MML Calculus

The MML is based on a small language called the MML calculus. It is an imperative object-oriented calculus that captures the essential operational features of MML. The calculus is based on the ς-calculus of Cardelli and Abadi [6]. To define the MML Calculus the ς-calculus is extended with some basic data types (integer, string, boolean, set and sequence) and operations over values of these data types. The rest of this section is structured as follows. Section 2.1 defines the syntax of the MML Calculus. Section 2.2 defines the semantics of the MML Calculus. Section 2.3 defines how to encode functions in the calculus, these are required to define parameterized operations over models. Section 2.4 defines some builtin operations that are either convenient or which cannot be defined in the calculus.

2.1 Syntax

The core syntax and value domain of the MML Calculus is defined below:

$$
\begin{array}{lll}
e, a, b ::= & & \text{expression} \\
 & v, w & \text{variable} \\
 & n & \text{integer} \\
 & b & \text{boolean} \\
 & s & \text{string} \\
 & [v_i(w_i) = e_i]^{i \in [0,n]} & \text{object expression} \\
 & e.v & \text{field reference} \\
 & a.v := b & \text{attribute update} \\
 & a.v := (w)b & \text{method update} \\
 & \mathbf{Set}\{e_i\}^{i \in [0,n]} & \text{set expression} \\
 & \mathbf{Seq}\{\} & \text{empty sequence} \\
 & \mathbf{Seq}\{a|b\} & \text{pair expression}
\end{array}
\qquad
\begin{array}{lll}
x, y ::= & & \text{value} \\
 & n & \text{integer} \\
 & b & \text{boolean} \\
 & s & \text{string} \\
 & [v_i \mapsto \alpha_i]^{i \in [0,n]} & \text{object} \\
 & \mathbf{Set}\{x_i\}^{i \in [0,n]} & \text{set} \\
 & \mathbf{Seq}\{\} & \text{empty sequence} \\
 & \mathbf{Seq}\{x|y\} & \text{pair} \\
 & (v, \rho, e) & \text{field closure}
\end{array}
$$

The core syntax will be extended with extra features as the description of the MML Calculus proceeds. This document uses the following conventions to define syntax: terminals are given in bold; X^* represents 0 or more occurrences of X; $X^?$ represents an optional X; $X|Y$ represents X or Y parentheses can be used to group elements (bold parentheses are terminals); $X^{i \in [n,m]}$ is a sequence of elements where X contains free occurrences of i.

An object expression defines a collection of fields. Each field has a name, a self parameter and a body. Like the ς-calculus we conflate the notion of *attribute* and *method* into a single *field*. Because attribute update occurs sufficiently often we distinguish between method and attribute update.

Appendix A defines the substitution of expressions for free variables. Variables are bound in methods via the self argument. When a field is referenced in an object, the object is supplied as the value of the self argument.

2.2 Semantics

For convenience we conflate the syntactic and value domains for integers, booleans, strings and empty sequences. We also make use of different categories of *environment* which are partial functions. An environment ϵ is extended with an association between a key k and a value v to produce $\epsilon[k \mapsto v]$. Environments satisfy the following law:

$$(\epsilon[k_1 \mapsto v_1])[k_2 \mapsto v_2] = (\epsilon[k_2 \mapsto v_2])[k_1 \mapsto v_1] \text{ when } k_1 \neq k_2$$

An object is an environment that maps field names to addresses. A lexical environment maps variables to values. A heap h is an environment that maps addresses to field closures. A field closure (v, ρ, e) contains a self variable v, a body e and a lexical environment ρ mapping the free variables of e to values.

The semantics of the MML Calculus us given by a relation $h, \rho, e \Rightarrow x, h'$ which tells us the result x and final heap h' produced by performing expression e with respect to a starting heap h in the context of a lexical environment ρ. The relation is defined as follows:

$$h, \rho[v \mapsto x], v \Rightarrow x, h \tag{1}$$

$$h, \rho, k \Rightarrow k, h \tag{2}$$

$$h, \rho, [v_i(w_i) = e_i]^{i \in [1,n]} \Rightarrow [v_i = \alpha_i]^{i \in [1,n]}, h[\alpha_i \mapsto (w_i, \rho, e_i)]^{i \in [1,n]} \text{ fresh } \alpha_i \tag{3}$$

$$\frac{\begin{array}{c} h, \rho, e \Rightarrow x, h'[\alpha \mapsto (w, \rho', b)] \\ h'[\alpha \mapsto (w, \rho', b)], \rho'[w \mapsto x], b \Rightarrow y, h'' \end{array}}{h, \rho, e.v \Rightarrow y, h''} \quad x = [v_i = \alpha_i, v = \alpha]^{i \in [1,n]} \tag{4}$$

$$\frac{h, \rho, e \Rightarrow x, h'}{h, \rho, e.v := (w)b \Rightarrow x, h'[\alpha \mapsto (w, \rho, b)]} \quad x = [v_i = \alpha_i, v = \alpha]^{i \in [1,n]} \tag{5}$$

$$\frac{\begin{array}{c} h, \rho, e \Rightarrow x[v \mapsto f], h' \\ h', \rho, b \Rightarrow y, h'' \end{array}}{h, \rho, a.v := b \Rightarrow y, h''[\alpha \mapsto (w, [v \mapsto y], v)]} \tag{6}$$

$$\frac{h_i, \rho, e_i \Rightarrow x_i, h_{i+1} \;\; i \in [1,n]}{h_1, \rho, \mathrm{Set}\{e_i\}^{i \in [1,n]} \Rightarrow \mathrm{Set}\{x_i\}^{i \in [1,n]}, h_{n+1}} \tag{7}$$

$$\frac{\begin{array}{c} h, \rho, a \Rightarrow x, h_1 \\ h_1, \rho, b \Rightarrow y, h_2 \end{array}}{h, \rho, \mathrm{Seq}\{a|b\} \Rightarrow \mathrm{Seq}\{x|y\}, h_2} \tag{8}$$

Free variables are bound to values in the current lexical environment (1). Constant expressions k are integers, booleans, strings and the empty sequence. When evaluated, a constant expression produces itself as a value (2). An object expression denotes an object. Fresh heap addresses are used for the object's fields. Notice that the current lexical environment is captured by each field since it contains the bindings for all free variables in a field body (3). Field reference causes the field body to be evaluated with respect to its lexical environment (4). Method update (5) replaces a field with a delayed expression. Field update (6) replaces a field with the value of an expression. The component expressions of a set expression are all evaluated and then the set is created. Sets do not live in the heap and therefore set operations do not cause side-effects (7). A pair is created after evaluating its head and tail (8).

MML is implemented as a Java program called MMT (the meta-modelling tool). MMT runs a virtual machine that executes the MML Calculus. The machine is defined by transforming the relation $h, \rho, e \Rightarrow, x, h'$ into a transition function over machine states such that $([], \rho, [e], h, ()) \longmapsto^* ([x], \rho, [], h', ())$. A machine state has the form (s, ρ, c, h, d) where the new components are: a stack s for intermediate values; a control c that is used as an instruction stream; a dump d that is used to save and restore machine contexts during field reference. A prototype version of MMT is available at [27].

2.3 Functions

The kernel calculus is object-based and not function-based like the λ-calculus. However, functions can be easily embedded in the calculus thereby providing the best of both worlds. A function with an argument v and a body b is $\lambda v.e$:

$$\begin{array}{lll} e, a, b ::= & \ldots \text{as before} & \text{expression} \\ & \lambda v.e & \text{function expression} \\ & \textbf{let } v = a \textbf{ in } b \textbf{ end} & \text{local definition} \end{array}$$

A function is an object with structure: $[\mathrm{arg}(\mathrm{self}) = \mathrm{self}; \mathrm{val}(\mathrm{self}) = b[\mathrm{self.arg}/v]]$ An application expression has the form $a(b)$ where a is an expression denoting the operator and b is an expression denoting the operand. The following equivalence is used: $a(b) = (a.copy.arg := b).val$ The copying (see section 2.4) is required in case the function is recursive.

The **let** expression introduces local definitions. Each definition consists of a variable v and a value a. The **let** expression has a body b. The scope of the variable is the body of the **let**. A **let** expression is defined in terms of a function: $\textbf{let } v = a \textbf{ in } b \textbf{ end} = (\lambda v.b)(a)$

2.4 Builtin Operations

The definition of MML relies on a number of builtin methods and operators. Extra rules (like δ-rules for the λ-calculus) are added to the calculus in order to define these features. Most of the rules involve functions, such as + or 'and' that do not depend on or change the current lexical environment or heap. The syntax is defined as follows:

$$\begin{array}{lll} e, a, b ::= & \ldots \text{as before} & \text{expression} \\ & \oplus {e_i}^{i \in [1,n]} & \text{operator expression} \end{array}$$

The semantics for each operator $\oplus$ is given by a rule $\mathrm{Op}(\oplus)$:

$$\frac{h_i, \rho, e_i \Rightarrow x_i, h_{i+1} \;\; i \in [1, n]}{h_1, \rho, \oplus(e_i)^{i \in [1,n]} \Rightarrow \oplus(x_i)^{i \in [1,n]}, h_{n+1}} \mathrm{Op}(\oplus_n) \tag{9}$$

The following binary operations are builtin: Op(+); Op(−); Op(∗); Op(/); Op(and); Op(or); Op(xor); Op(implies) When an object is copied, fresh addresses (α_i') are allocated for its fields. Note that the copy is *shallow*, i.e. the values associated with the fields are not copied:

$$\frac{h, \rho, e \Rightarrow [v_i \mapsto \alpha_i]^{i \in [1,n]}, h[\alpha_i \mapsto (w_i, \rho_i, e_i)]^{i \in [1,n]}}{h, \rho, e.\text{copy} \Rightarrow [v_i \mapsto \alpha_i']^{i \in [1,n]}, h[\alpha_i \mapsto (w_i, \rho + i, e_i)]^{i \in [1,n]}}$$

Copying for atomic values and sets has no effect. This is expressed by a generic rule Copy(x) and a collection of rules for all integers n Copy(n), for all booleans b Copy(b), for all strings s Copy(s) and for all collections c Copy(c):

$$\frac{h, \rho, e \Rightarrow x, h'}{h, \rho, e.\text{copy} \Rightarrow x, h'} \text{ Copy}(x)$$

Operations on sets are based on two operations: non-deterministic selection and adjoining an element to a set. Set extension is an operation that is constructed from set adjoin. Basic set operations are Op(select) and Op(adjoin): $\text{select}(\text{Set}\{x_i\}^{i \in [1,n]}) = x_j$ for some $j \in [1,n]$
$\text{adjoin}(x, \text{Set}\{x_i\}^{i \in [0,n]}) = \text{Set}\{x, x_i\}^{i \in [0,n]}$
The head and tail of sequences are accessed using the following operations Op(head) and Op(tail): $\text{head}(\text{Seq}\{h|t\}) = h$ and $\text{tail}(\text{Seq}\{h|t\}) = t$. Equality Op(=) is defined as a builtin operation that returns true when atomic values are the same, objects have the same fields, when sets have the same elements and when sequences are either both empty or have equal heads and tails. Otherwise equality returns false. More sophisticated notions of equality can be constructed as methods for classes of MML object.

Boolean values support conditional commands by defining a function that evaluates a consequent c or alternative a: true = $[\text{if}(\_) = \lambda c.\lambda a.c.val]$ and false = $[\text{if}(\_) = \lambda c.\lambda a.a.val]$ The calculus is extended with a conditional expression:

$$\begin{array}{lll} e, a, b ::= & \ldots \text{as before} & \text{expression} \\ & \textbf{if } e \textbf{ then } a \textbf{ else } b \textbf{ end} & \text{conditional expression} \end{array}$$

The conditional expression is defined as follows: **if** e **then** a **else** b **end** = $e.\text{if}([\text{val}(\_) = a])([\text{val}(\_) = b])$

3 The Meta-Modelling Language

The meta-modelling language is given a semantics by a translation to the MML Calculus. MML is meta-circular, in the sense that it represents all the types and operations in order to describe its own operation. This section defines how the following MML features are represented in the MML Calculus: objects (section 3.1); methods (section 3.2); types (section 3.3); classes (section 3.4); method invocation (section 3.5); OCL (section 3.6); packages (section 3.7). The meta-circular MML model is shown in figure 1.

3.1 Objects and Atomic Expressions

MML supports integer; boolean; string; set and sequence expressions. All of the builtin operators defined in section 2.4 are supported by MML. The semantics of the following MML expressions is given by a direct translation to the MML Calculus:

$m, l ::=$		MML expression
	v, w	variable
	n	integer
	b	boolean
	s	string
	$\oplus {m_i}^{i \in [1,n]}$	operator expression
	$m.v := l$	field update
	$\mathbf{Set}\{m_i\}^{i \in [0,n]}$	set expression
	$\mathbf{Seq}\{\}$	empty sequence expression
	$\mathbf{Seq}\{l \vert m\}$	pair expression
	$\mathbf{let}\ v = a\ \mathbf{in}\ b\ \mathbf{end}$	local definition
	$\mathbf{if}\ e\ \mathbf{then}\ a\ \mathbf{else}\ b\ \mathbf{endif}$	conditional expression

All data values in MML have a field named 'of' whose value is a *classifier* for the data value. A classifier contains type information for a collection of its instances. MML knows directly about a number of builtin classifiers. Access to the builtin classifiers is defined by the generic rule Of:

$$\frac{h, \rho, m \Rightarrow x, h'}{h, \rho, m.\mathrm{of} \Rightarrow y, h'}\ \mathrm{Of}(x, y)$$

which is used to construct rules for every integer n $\mathrm{Of}(n, \mathrm{Integer})$, for every boolean b $\mathrm{Of}(b, \mathrm{Boolean})$, for every string s $\mathrm{Of}(s, \mathrm{String})$, for every set s Of(s, SetOfInstance) and for every sequence s $\mathrm{Of}(s, \mathrm{SeqOfInstance})$. The classifiers Boolean, String, SetOfInstance and SeqOfInstance are MML objects that define the appropriate data types.

MML object expressions require that the object's classifier be supplied. Each MML field is a MML calculus field and may optionally supply the self parameter w_i. If the self parameter is not supplied then it defaults to 'self':

$l, m ::=$ ... as before		MML expression
	$@m\ v_i(w_i) = m_i\ \mathbf{end}^{i \in [0,n]}$	object expression

where m denotes the object's classifier. The expression is translated to an MML calculus expression by inserting the 'of' field with a dummy self parameter:

$$[\mathrm{of} = (\_)m; v_i(\mathrm{self}) = m_i]^{i \in [0,n]}$$

3.2 Methods

MML methods are parameterized OCL expressions whose values are computed when a message is sent to an object. The methods are based on the representation of functions given in section 2.3; the representation is extended so that functions are classified as instances of the class Function and to allow functions with multiple arguments. Methods extend functions with an extra implicitly defined argument for 'self'. A function applies arguments using the following method:

Function::apply(args : Seq(Instance)):Instance
  if args = Seq{} **then** self.val
  else (self.copy.arg := args.head).val.apply(args.tail)
  endif

The following syntax is used to denote MML functions:

$$
\begin{array}{lll}
m, l ::= & \ldots \text{as before} & \text{MML expression} \\
 & \textbf{fun}\ (v^*)\ m\ \textbf{end} & \text{function} \\
 & \textbf{meth}\ (v^*)\ m\ \textbf{end} & \text{method}
\end{array}
$$

Unary functions are translated to MML objects, multary functions are curried and methods insert an extra argument named 'self':

$$
\begin{array}{l}
\textbf{fun}\ (v)\ m\ \textbf{end} = \text{@Function arg(self)} = \text{self}; \text{val}(f) = m[f.\text{arg}/v]\ \textbf{end} \\
\textbf{fun}\ (v_1, \ldots, v_n)\ m\ \textbf{end} = \textbf{fun}\ (v_1)\ \ldots\ \textbf{fun}\ (v_n)\ m\ \textbf{end}\ \ldots\ \textbf{end} \\
\textbf{meth}\ (v_1, \ldots, v_n)\ m\ \textbf{end} = \textbf{fun}\ (\text{self}, v_1, \ldots, v_n)\ m\ \textbf{end}
\end{array}
$$

Methods are defined by classes and invoked by sending an MML object a message. Message delivery involves looking the method up via the object's classifier and then invoking the method with respect to the object and the arguments. This is explained in section 3.5:

$$
\begin{array}{lll}
l, m ::= & \ldots \text{as before} & \text{MML expression} \\
 & m.v(l_i^{i \in [0,n]}) & \text{send expression}
\end{array}
$$

3.3 Types and Type Expressions

All MML data values have classifiers; given a value x, the classifier of x is x.of. A classifier may be user defined (such as Animal or Factory) or may be defined as part of the MML language (such as String and Boolean). MML makes a distinction between *data types* that classify non-object values and *classes* that classify object values. This section describes the basic infrastructure of data types and their denotation.

All classifiers define a collection of methods and invariants for their instances. Each classifier must specify a default value to be used when a new slot using the classifier as a type is created; this is initially specified as Instance and is redefined in concrete sub-classes of Classifier. There are two main sub-classes of Classifier: Class and DataType. Instances of DataType classify non-object data values. DataType redefines the default method to return the value of the default attribute.

3.4 Class Expressions

MML classes are defined using *class expressions*. A class expression is sugar for object expressions and simply serves to capture the common pattern of class definition. Since MML is meta-circular, all classes are objects, all meta-classes are objects and so on. Class definition syntax is defined as follows:

$l, m ::=$	... as before $\mid c \mid k$	MML expressions
$c ::=$	**class** $v\mu^?\pi^?(\alpha\|\nu\|\sigma)^*\iota^?$ **end**	class definition
$\mu ::=$	**metaclass** m	metaclass
$\pi ::=$	**extends** $m(,m)^*$	parents
$\alpha ::=$	$v : \tau$	attribute
$\nu ::=$	$v(\delta^?(,\delta)^*)m$	method
$\sigma ::=$	$v = m$	slot
$\delta ::=$	$v : \tau$	declaration
$\tau ::=$		type
	v	type name
	$\mathbf{Set}(\tau)$	set type
	$\mathbf{Seq}(\tau)$	seq type
$\iota ::=$	**inv** $(sm)^*$	invariant

A class expression consists of the name of the class followed by a number of clauses. A class is an object with its own classifier. The classifier is expressed in a class expression in the optional *metaclass* clause; if it is omitted then it defaults to Class. A class has multiple parents from which it inherits various definitions. The parents of a class are expressed in the optional *parents* clause; if it is omitted then the class will have the single parent Object. A class contains a number of definitions for attributes, methods and slots in the *definitions* clause. The attributes define the slots contained in instances of the class and the methods define the behaviour of the instances. The slot definitions allow extra information to be added to the class being defined where there is no syntax support. Extra slots are required when using a non-standard meta-class.

Figure 2 defines the translation of an MML class definition to an MML object expression. We will consider each feature category in turn; where appropriate we will draw attention to the scope of names available when each category is evaluated.

There are $|p|$ parent expressions. The scope of the new class name v does not include the parents since it is illegal to create cycles in the inheritance structure. There are $|a|$ attributes. The scope of v includes the attribute definitions since a class can contain an attribute whose values are instances of the class. There are $|m|$ methods, Each method has m_i arguments. The body of the method is a method function. The scope of v contains the method definitions since the methods of a class can refer to that class. Note that the arguments of the method functions are defined in an inner scope so they may shadow the class name v. There are $|\iota|$ constraints defined for the invariant of a class. The scope of v includes the invariant. There are $|s|$ slots. The scope of v includes the slot values. The slots should correspond to attributes of the meta-class m that are not explicitly introduced by the class definition transformation.

3.5 Method Invocation

Method invocation in MML occurs when a send expression is performed. The expression has the following syntax:

$$\begin{array}{rll} l, m ::= & \ldots \text{as before} & \text{MML expression} \\ & m.v({l_i}^{i\in[0,n]}) & \text{send expression} \end{array} \tag{10}$$

where m is the target, v is the method name and l_i are arguments. A method with the name v is found by searching through the methods defined by the classifier of m.

Message delivery occurs by invoking the message delivery service. This is implemented directly in the calculus, but it is convenient to think of it as a method defined by the classifier of the target[1]. The method is defined as follows:

```
Classifier::send(target:Instance,message:String,args:Seq(Instance)):Instance
  let methods = self.allMethods()→select(m | m.name = message)
  in if methods = Seq{} then methods.head.apply(Seq{target | args})
     else self.error("no method for " + message)
     endif end
```

3.6 The Object Constraint Language

The Object Constraint Language (OCL) is an expression language used to express invariants, pre- and post- conditions, and guards on state transitions. OCL is fully integrated within MML by using the builtin operations defined by the Op rule 9 and by reducing collection operations to a very small number of primitives and then using these to implement methods in MML classifiers. OCL thereby becomes a convenient syntax for invoking a collection of predefined operations and methods whose implementation can be understood in terms of a very small number of primitive operations.

In this section we extend MML with the rest of OCL expression syntax. Figure 1 shows the root of the OCL expression hierarchy Exp. Each new syntactic category involves a new method definition for the appropriate classifier and a translation from the OCL expression to the appropriate method call. There are three groups of definitions: generic collections; sets; sequences. Bags are currently not implemented in MML but should follow the same implementation pattern as sets and sequences.

Collections Many OCL operations work on all types of collection. Typically these operations are implemented in terms of lower-level operations that are

[1] If this approach was implemented efficiently then it would provide a mechanism for controlling message delivery at the meta-level

specific to the particular type of collection:

$$\begin{array}{ll} l, m ::= \ldots \text{as before} & \text{MML expression} \\ \quad m \to v(m(,m)^*)^? & \text{collection operation} \\ \quad m \to v(w|l) & \text{quantified operation} \\ \quad m \to \text{iterate}(vw = m|m) & \text{iteration} \end{array}$$

Collection operation names are: size, includes (an element), count (ocurrences of an element), includesAll (elements of another collection), isEmpty, notEmpty, sum (the elements). Quantified operations are exists and forAll. The class CollectionOfInstance is the root of the collection type hierarchy. Its implementation is an example of how MML supports multiple meta-levels. Its meta-class Collection is a sub-class of DataType and therefore is the class of all collection types.

There are two different categories of expression: those that introduce new variables (exists, forAll and iterate) and those that do not. Expressions that do not introduce new variables are translated to send expressions in MML. For example $m \to \text{includes}(l)$ becomes $m.\text{includes}(l)$, with the following implementation:

CollectionOfInstance::includes(o : Instance) : Boolean
 self$\to$ count(o) $>$ 0
CollectionOfInstance::count(o : Instance) : Integer
 self$\to$ iterate $(v_1 v_2 = 0|$ **if** v_1.equals(o) **then** $v_2 + 1$ **else** v_2 **endif**)

Expressions that introduce variables are translated to send expressions that bind the variables using one or more functions. Functions are objects in MML and may be passed as arguments to methods. The following table shows the translations:

$$\begin{array}{rl} m \to \text{exists}(v|l) & m.\text{exists}(\textbf{fun}(v)\ l\ \textbf{end}) \\ m \to \text{forAll}(v|l) & m.\text{forAll}(\textbf{fun}(v)\ l\ \textbf{end}) \\ m_1 \to \text{iterate}(v_1 v_2 = m_2|m_3) & m_1.\text{iterate}(\textbf{fun}(v_1, v_2)\ m_3\ \textbf{end}, m_2) \end{array}$$

The quantified operations are defined in terms of iterate. Iterate is abstract at the collection level and is implemented by each concrete type of collection.

CollectionOfInstance::exists(f : Function) : Boolean
 self$\to$iterate(e a = false | a or f.apply(Seq{e}))
CollectionOfInstance::forAll(f : Function) : Boolean
 self$\to$iterate(e a = true | a and f.apply(Seq{e}))

There is nothing special about the collection method definitions. MML is open to extension just like any other object-oriented model. The combination of polymorphism, dynamic binding and first class functions provides a very powerful abstraction and extension mechanism.

Sets Set expression syntax is the same as that for collections. Set operations are: union; intersection; $-$; including (adjoin); symmetricDifference; asSequence. Parametric operations are: select (positive filter); reject (negative filter); collect (map). In each case the parametric operations are translated to send expressions with a function argument.

The class SetOfInstance is a sub-class of CollectionOfInstance and implements set operations. Many of the methods are implemented in terms of other SetOfInstance methods. The only set operations that rely on primitive operations are including that uses adjoin and iterate that uses select. These primitives along with the empty set can be viewed as being the essence of sets in MML. The following shows the definition of iterate:

SetOfInstance::iterate(v:Instance,f:Function):Instance
 if self = Set{} **then** v
 else **let** e = self.select
  in (self→excluding(e)).iterate(f.apply(Seq{e,v}),f) **end**
 endif

Sequences OCL sequence expressions are implemented in MML and the MML Calculus using the same pattern of features as defined in the previous section. OCL sequences ultimately depend upon two primitive operations: head and tail. These are builtin to the OCL Calculus as defined in section 2.4.

3.7 Package Expressions

A package in MML is a container of classes and packages similar to those defined by the Catalysis approach [12]. Each class and package has a name and navigation expressions can be used to extract package contents. A package is itself a classifier; it is used to classify collections of objects. It is beyond the scope of this paper to deal with this feature of MML in detail; as a classifier, a package must have parents, methods and invariants. A package is created using a *package expression*:

l, m ::=	... as before $\mid p$	MML expressions
p ::=	**package** $v\mu^{?}\pi^{?}(c\mid k\mid p)^{*}(\nu\mid\sigma)^{*}\iota^{?}$ **end**	package definition

A package definition expression is translated to an object expression in the MML Calculus. This translation is the same as that shown in figure 2 except that a package has an extra field called contents whose value is a set of classifiers. Each classifier defined by the package is also a field of the package; this allows us to navigate to package elements using names. The name of the package is scoped over each of the contents; this allows the contents to be mutually recursive.

MML consists of a collection of packages. It is beyond the scope of this paper to describe the package structure of MML. Consider a package called concepts that defines the core modelling concepts in MML. Figure 3 shows how such a package is expressed in MML and given a semantics by a translation to the MML Calculus; the details of class translation are omitted, but follow the definition given in figure 2.

4 Conclusion

This paper has defined the MML Calculus which is used as the basis for a meta-circular modelling language called ML. The aim of MML is to provide a sound basis for UML 2.0. The need for a precise semantics for UML and OCL is shown by the increasing number of papers describing sophisticated tools and techniques. The following is a small selection: [1], [16], [23]. It is essential that the UML 2.0 initiative provide a standardized formal semantics for a UML core. The work described in this paper is part of an ongoing initiative by the pUML Group (`http://www.puml.org`) [7], [5], [8], [9], [27], [13]. The approach separates syntax and semantics so that the syntax of UML may change (for example graphical OCL [15], [18]) while the semantic domain remains the same.

We have taken a translational approach to the semantics of MML. This provides a semantics in terms of a much smaller calculus thereby achieving a parsimony of concepts. Such an approach has a distinguished history leading back to Landin and Iswim. We would hope that this approach will lead to proof systems and refinement calculi for MML. The approach should be contrasted with other meta-modelling methods and tools including: BOOM [20] and BON [21]. The key novel feature of MML is that is aims to be exclusively based on UML and OCL; this allows UML to be both an extension and an instance of MML. UML is based on the Meta-Object Facility (MOF) which is a relatively small meta-model. MOF is not based on a precise semantics; it is hoped that work on MML will influence the future development of MOF.

It is somewhat difficult to validate the semantics since all definitions of UML are very imprecise; most parts of UML 1.3 are defined as syntax only. We can report that the implementation of MML based on the calculus in MMT has been used to develop MML programs of about 3000 lines of code; based on this empirical evidence we can tentatively claim that MML behaves as expected.

The semantics described in this paper is imperative and operational. In that sense it is richer than UML whose informal semantics is *declarative*. However, the MML Calculus has a strictly declarative sub-language which captures the same semantics as other model-based approaches to UML semantics such as [24] and those that use Z.

The use of meta-circular language definitions has a distinguished history, for example CLOS [2], ObjVLisp [3] [4] [11] and Smalltalk [14] use meta-classes. The theory of meta-classes is discussed in [17]. The use of meta-models to describe UML and OCL is the accepted technique [22].

This is ongoing work. Core MML is nearing completion and a tool, MMT, that supports definition, use and checking of MML is under development. We intend to consolidate MML in the near future, for example by adding types [10], [25]; then, MML will be used to define a number of languages that contribute to the UML family.

References

1. Bottoni P., Koch M., Parisi-Presicce F., Taentzer G. (2000) Consistency Checking and Visualization of OCL Constraints. In Evans A., Kent S., Selic B. (eds) UML 2000, LNCS 1939, 278 – 293, Springer-Verlag.
2. Bobrow D. (1989) Common Lisp Object System Specification. Lisp and Symbolic Computation, 1(3/4).
3. Briot J-P, Cointe P. (1986) The ObjVLisp Model: Definition of a Uniform Reflexive and Extensible Object-oriented Language. In Proceedings of ECAI 1986.
4. Briot J-P., Cointe P. (1987) A Uniform Model for Object-oriented Languages Using the Class Abstraction. In proceedings of IJCAI 1987.
5. Brodsky S., Clark A., Cook S., Evans A., Kent S. (2000) A feasibility Study in Rearchitecting UML as a Family of Languages Using a Precise OO Meta-Modeling Approach. Available at `http://www.puml.org/mmt.zip`.
6. Cardelli L., Abadi M. (1996) A Theory of Objects. Springer-Verlag.
7. Clark A., Evans A., France R., Kent S., Rumpe B. (1999) Response to IML 2.0 Request for Information. Available at `http://www.puml.org/papers/RFIReponse.PDF`.
8. Clark A., Evans A., Kent S. (2000) The Specification of a Reference Implementation for UML. Special Issue of L'Objet.
9. Clark A., Evans A., Kent S. (2000) Profiles for Language Definition. Presented at the ECOOP pUML Workshop, Nice.
10. Clark A. (1999) Type-checking OCL Constraints. In France R. & Rumpe B. (eds) UML '99 LNCS 1723, Springer-Verlag.
11. Cointe, P. (1987) Metaclasses are First Class: the ObjVLisp Model. In Proceedings of OOPSLA 1987.
12. D'Souza D., Wills A. C. (1998) Object Components and Frameworks with UML – The Catalysis Approach. Addison-Wesley.
13. Evans A., Kent S. (1999) Core meta-modelling semantics of UML – The pUML approach. In France R. & Rumpe B. (eds) UML '99 LNCS 1723, 140 – 155, Springer-Verlag.
14. Goldberg A., Robson D. (1983) Smalltalk-80: The Language and Its Implementation. Addison Wesley.
15. Howse J., Molina F., Kent S., Taylor J. (1999) Reasoning with Spider Diagrams. IEEE Symposium on Visual Languages '99, 138 – 145. IEEE CS Press.
16. Hussmann H., Demuth B., Finger F. (2000) Modular Architecture for a Toolset Supporting OCL In Evans A., Kent S., Selic B. (eds) UML 2000, LNCS 1939 LNCS, 278 – 293 , Springer-Verlag.
17. Jagannathan S. (1994) Metalevel Building Blocks for Modular Systems. ACM TOPLAS 16(3).
18. Kent S. (1997) Constraint Diagrams: Visualizing Invariants in Object-Oriented Models. In Proceedings of OOPSLA '97, 327 – 341.
19. Object Management Group (1999) OMG Unified Modeling Language Specification, version 1.3. Available at `http://www.omg.org/uml`.
20. Overgaard G. (2000) Formal Specification of Object-Oriented Meta-Modelling. FASE 2000, LNCS 1783.
21. Paige & Ostroff (2001) Metamodelling and Conformance Checking with PVS. To be presented at FASE 2001.
22. Richters M., Gogolla M. (1999) A metamodel for OCL. In France R. & Rumpe B. (eds) UML '99 LNCS 1723, 156 – 171, Springer-Verlag.

23. Richters M., Gogolla M. (2000) Validating UML Models and OCL Constraints. In Evans A., Kent S., Selic B. (eds) UML 2000 LNCS 1939, 265 – 277, Springer-Verlag.
24. Richters M., Gogolla M. (2000) A Semantics for OCL pre and post conditions. Presented at the OCL Workshop, UML 2000.
25. Schurr A. (2000) New Type Checking Rules for OCL (Collection) Expressions. Presented at the OCL Workshop at UML 2000. Available from `http://www.scm.brad.ac.uk/research/OCL2000`.
26. Warmer J., Kleppe A. (1999) The Object Constraint Language: Precise Modeling with UML. Addison-Wesley.
27. The MMT Tool. Available at `http://www.puml.org/mmt.zip`.

A Substitution for Free Variables

$v[e/w] = \begin{cases} e \text{ when } v = w \\ v \text{ otherwise} \end{cases}$	variable
$k[e/v] = k$	atomic expression
$[v_i(w_i) = e_i]^{i\in[1,n]}[e/v] = [(v_i(w_i) = e_i)[e/v]]^{i\in[1,n]}$	object expressions
$(v(w) = a)[b/w'] = \begin{cases} v(w) = a & \text{when } w' = w \\ v(w) = (a[b/w']) & \text{otherwise} \end{cases}$	method
$a.v[b/w] = (a[b/w]).v$	field reference
$(a.v := b)[e/w] = (a[e/w]).v := (b[e/w])$	field update
$(a.v := (w)b)[e/w'] = \begin{cases} (a[e/w']).v := (w)b & \text{when } w' = w \\ (a[e/w']).v := (w)(b[e/w']) & \text{otherwise} \end{cases}$	method update
$\text{Set}\{e_i\}^{i\in[1,n]}[e/v] = \text{Set}\{e_i[e/v]\}^{i\in[1,n]}$	set expression
$\text{Seq}\{a\|b\}[e/v] = \text{Set}\{a[e/v]\|b[e/v]\}$	pair

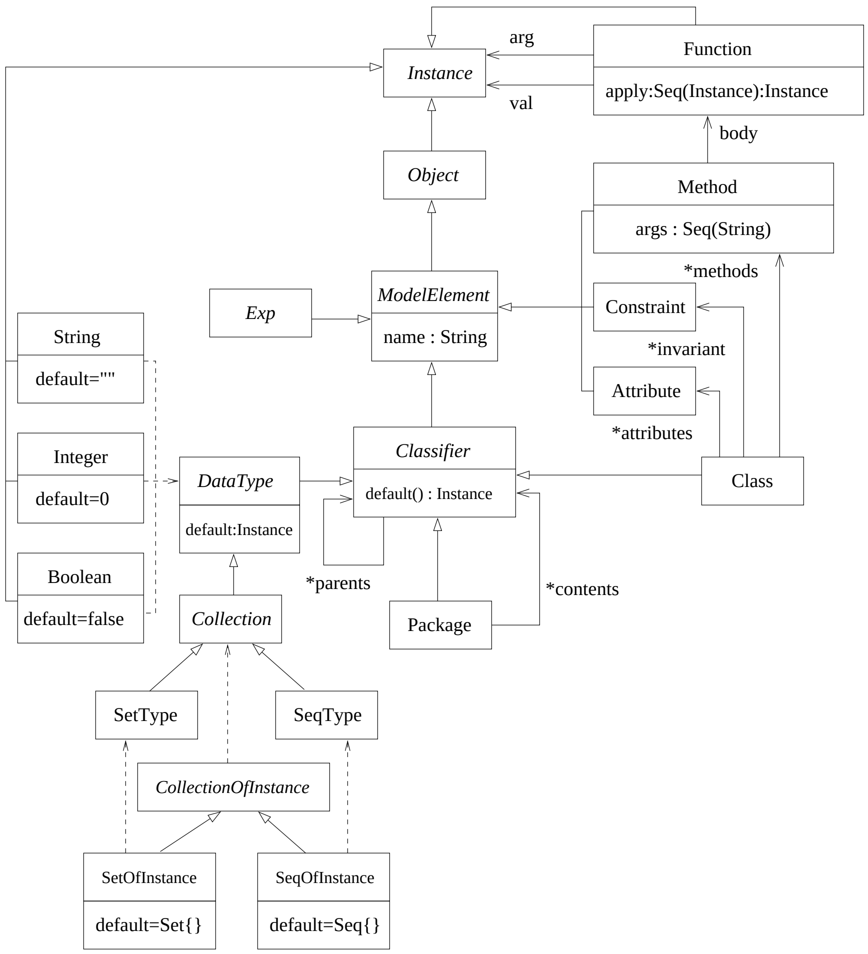

Fig. 1. The MML Language Model

class v
  metaclass m
  extends ${m_i^p}^{i \in [1,|p|]}$
  ${v_i^a : \tau_i^a}^{i \in [0,|a|]}$
  ${v^{m_i}({v_j^{m_i} : \tau_j^{m_i}}^{j \in [0,|m_i|]}) m_i^m}^{i \in 0,|m|}$
  ${v_i^s = m_i^s}^{i \in [0,|s|]}$
  inv ${s_i m_i^\iota}^{i \in [0,|\iota|]}$
end

@m
  name = "v";
  parents = Set$\{m_i^p\}^{i \in [1,|p|]}$
  attributes(v) = Set{
    @Attribute
      name = "v_i^a";
      type = τ_i^a
    end$\}^{i \in [0,|a|]}$;
  methods(v) = Set{
    @Method
      name = "v_i^m";
      args = Seq$\{$"$v_j^{m_i}$"$\}^{j \in [0,|m_i|]}$;
      body = **meth** $(v_j^{m_i})^{j \in [0,|m_i|]}$ m_i^m **end**
    end$\}^{i \in [0,|m|]}$;
  invariant(v) = Set{
    @Constraint
      name = s_i;
      body = **meth** () m_i^ι **end**
    end$\}^{i \in [0,|\iota|]}$;
  ${v_i^s(v) = m_i^s}^{i \in [0,|s|]}$
end

Fig. 2. Translation of Class Definition to Object Expression

package concepts
  class Classifier
  ...
  end
  class Package
    extends Classifier
  ...
  end
  class Class
    extends Classifier
  ...
  end
end

[of(concepts) = concepts.Package;
 name(_) = "concepts";
 contents(concepts) = Set{
   concepts.Classifier,
   concepts.Package,
   concepts.Class};
 Classifier(concepts =
   [of(_) = concepts.Class;
    name(_) = "Classifier"; ...];
 Package(concepts =
   [of(_) = concepts.Class;
    name(_) = "Package";
    parents(_) = Set{concepts.Classifier}; ...];
 Class(concepts =
   [of(_) = concepts.Class;
    name(_) = "Class";
    parents(_) = Set{concepts.Classifier}; ...];

Fig. 3. Package Translation