MML and the Metamodel Architecture

José Álvarez, Universidad de Málaga, Spain
Andy Evans, University of York, UK
Paul Sammut, University of York, UK
contact: pauls@cs.york.ac.uk

January 16, 2001

Abstract: The Meta-Modelling Language is a static object-oriented modelling language whose focus is the declarative definition of languages. It aims to enable the UML metamodel to be precisely defined, and to enable UML to evolve into a family of languages. This paper argues that although MML takes a metamodelling approach to language definition, it cannot be described as strict metamodelling. This has significant implications on the nature of the metamodel architecture it supports, yet without contravening the OMG's requirements for the UML 2.0 infrastructure. In particular it supports a rich generic nested architecture as opposed to the linear architecture that strict metamodelling imposes. In this nested architecture, the transformation between the representation of any model at one metalevel and its representation in the metalevel below can be described by an information preserving one-to-one mapping. This mapping provides the basis for a powerful area of functionality that any potential metamodelling tool should look to exploit.

1. Introduction

The Unified Modeling Language (UML) has been rapidly accepted as a standard notation for modelling object-oriented software systems. However, the speed at which UML has developed has led to some issues, particularly regarding its customisability and the precision of its semantics [k99]. UML is therefore evolving, with an impending revision (UML 2.0 [uml2wg]) seeking to resolve these and other issues.

The customisability issue has arisen in part because UML has proven so popular. UML was originally designed as a general-purpose language that was not intended for use in specific domains [rjb99]. However as UML has become more widespread, there has been considerable demand for it to be applicable in specialised areas. As it currently stands, UML is a monolithic language with little provision for customisability. Any new features would need to be added to the single body of language, leading to an increasingly unwieldy language definition. There is also potential for conflicts between the requirements of different domains, which may not be resolvable within a monolithic language definition [Cook, cmwwe99]. An alternative approach (which the OMG has set as a mandatory requirement for UML 2.0 [omg00]) allows the language to naturally evolve into a family of distinct dialects called 'profiles' that build upon a core kernel language. Each profile would have its own semantics (compatible with the kernel language) specific to the requirements of its domain.

The semantics issue concerns the fact that the current specification for UML (version 1.3) [omg99] uses natural language to define its semantics. Thus there is no formal definition against which tools can be checked for conformance. This has resulted in a situation where few tools from different vendors are compatible [Warmer, cmwwe99]. Thus a formal precise semantics is essential for compliance checking and interoperability of tools.

Another key requirement for UML 2.0 is that it should be rigorously aligned with the OMG four-layer metamodel architecture [omg00]. In this architecture, a model at one layer is used to specify models in the layer below. In turn a model at one layer can be seen to be an 'instance' of some model in the layer above. The four layers are the meta-metamodel layer (M3), the metamodel layer (M2), the user model layer (M1) and the user

object layer (M0). The UML metamodel (the definition of UML) sits at the M2 layer, and as such it should be able to be described as an instance of some language meta-metamodel at the M3 level. Although the relationship between UML and MOF (the OMG's standard M3 language for metamodelling) loosely approximates this [omg99], the lack of a precise formal UML metamodel severely limits the potential value of such a relationship.

In line with these key requirements for UML 2.0, the Precise UML group [puml] have proposed 'rearchitecting UML as a family of languages using a precise object-oriented metamodelling approach' [bccek00]. The foundation of their proposal is the Meta-Modelling Facility, which comprises the Meta-Modelling Language (MML), which is an alternative M3 layer language for describing modelling languages such as UML, and a tool (MMT) that implements it. MML is described in detail in Section 2.

Whilst MML takes a metamodelling approach, it cannot be described as 'strict' metamodelling. This paper outlines the implications that this has on the four-layer metamodel architecture, and offers a modified form of this architecture. It also argues that when viewed in this modified architecture, any model can be represented at a number of different metalevels, and that a precise definition can be given to the transformation of a model between those metalevels. This transformational mapping should be a key feature of any metamodelling tool.

The paper is structured as follows: Section 2 gives an overview of MML; Section 3 describes the 'nested' metamodel architecture supported by MML; Section 4 discusses the mapping that describes the transformation between metalevels in this architecture; Section 5 demonstrates the nested architecture in relation to the MML metamodel itself; and Section 6 outlines conclusions and further work.

2. The Meta-Modelling Language

The focus of MML is the declarative definition of languages [bccek00]. MML is a 'static OO modelling language that aims to be small, meta-circular and as consistent as possible with UML 1.3' [ceks00]. This section describes the approach, architecture and components of MML used to fulfil these aims. A detailed description of MML, which is outside the scope of this paper, can be found in [bccek00].

MML partitions the fundamental components of language definition into separate packages. It makes two key orthogonal distinctions [bccek00]: between 'model' and 'instance', and between 'syntax' and 'concepts'. Models describe valid expressions of a language (such as a UML 'class'), whereas instances represent situations that models denote (such as a UML 'object'). Concepts are the logical elements of a language (such as a class or object), and (concrete) syntax refers to the representation of those concepts often in a textual or graphical form (for example the elements of a class diagram or XML code). MML also defines appropriate mappings between these various language components, in particular a semantic mapping between the model and instance concepts, and mappings between the syntax and concepts of both the model and instance components. The clean separation of language components and the mappings between them is fundamental to the coherence and readability of the MML metamodel.

Since MML is itself a language, the above separation of language components is applied to its own metamodel. In fact a key feature and test for the MML metamodel is that it must be expressive enough to describe itself [ceks00]. This meta-circularity also eliminates the need for another language to describe MML, thus closing the language definition loop. MML defines the minimum number of concepts needed to define itself, and wherever possible uses the existing syntax and concepts of UML 1.3.

The key extensibility mechanism in MML that provides the means of realising UML as a family of languages is the notion of package specialisation based on that of Catalysis [dw98]. In the Catalysis approach, package specialisation is achieved through inheritance (just as classes can be specialised through inheritance in UML). This allows the definition of language components to be developed incrementally over a number of packages [ceks00]. Package inheritance is illustrated in Fig. 1 below. *PackageR* inherits the properties and contents of both *PackageP* and *PackageQ*, which includes the two definitions of *MyClass*. Thus, even though *PackageR* has no explicit contents, it implicitly contains a single class *MyClass* that combines the properties of both *PackageP.MyClass* and *PackageQ.MyClass*.

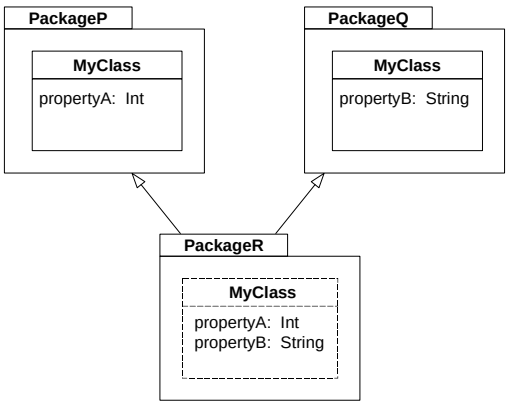
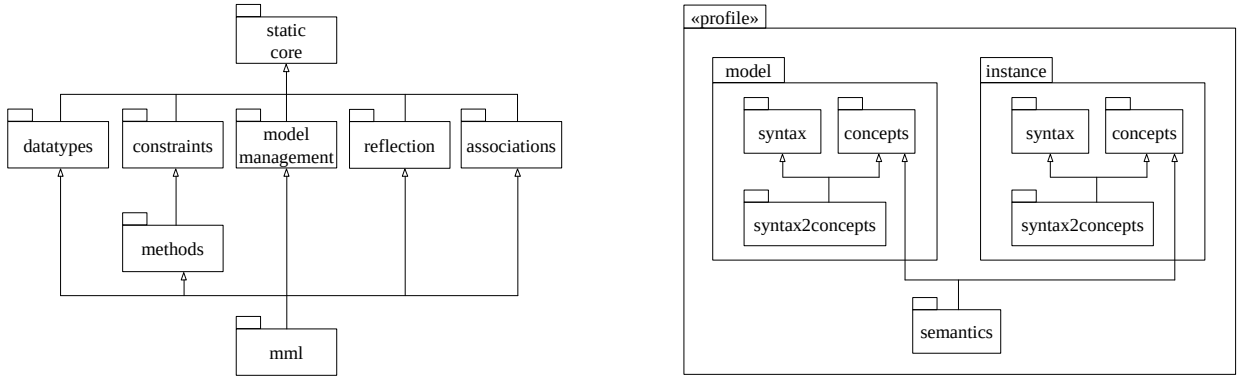


Fig. 1 Catalysis Package Inheritance Mechanism

The architecture of the MML metamodel is founded on the separation of language components and the notion of package inheritance described above. The metamodel is split into a number of key profile packages, each of which describe a fundamental aspect of modelling languages, and which are combined through an inheritance hierarchy to give the complete MML definition (Fig. 2(a) adapted from [bccek00]). In turn, each of the profile packages shares the same framework of subpackages as depicted in Fig. 2(b) (adapted from [bccek00]).



(a) Profile packages **(b) Profile framework**

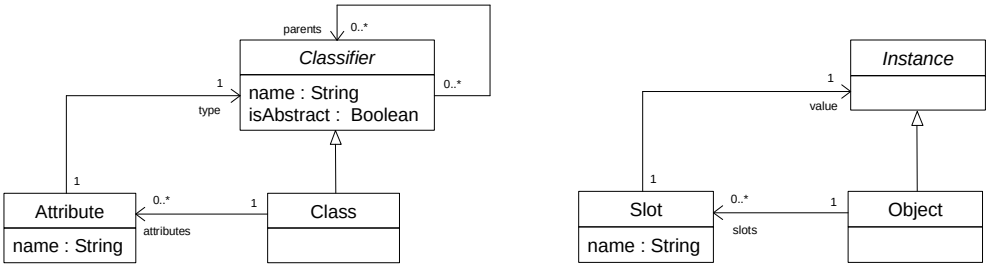
Fig. 2 The MML Architecture

These subpackages represent the basic language components and the appropriate mappings between them, and contain modelling constructs such as classes and associations (by way of illustration, a simplified form of the *static core* profile is depicted in Fig. 3). Well-formedness rules are realised by applying OCL constraints to those constructs, for example, circular inheritance is prohibited with the constraint:

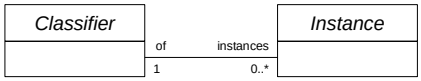
```
context Classifier inv:
   not self.allParents -> includes(self)
```

The *model.concepts* packages thus contain constructs that denote valid language metamodels (such as the UML metamodel), and the *instance.concepts* packages describe constructs that denote valid instances of those metamodels (such as UML models). In order to be valid, those metamodels must adhere to the same architecture depicted in Fig. 2(b).

A key idea to grasp for the discussion that follows in Section 3 is that MML serves two distinct functions: it is both a metamodelling language (such that the UML metamodel instantiates the MML metamodel) and a kernel language (it defines a subset of elements needed in the UML metamodel). Ultimately then, the aim is for the UML 2.0 metamodel to be both a specialisation and instance of the MML metamodel.



(a) *model.concepts* package **(b) *instance.concepts* package**



(c) *semantics* package

Fig. 3 Simplified *staticCore* Profile

3. Strict Metamodelling and the Four-Layer Architecture

This section argues that MML does not in fact fit into a *strict* metamodelling architecture, and that instead of being a problem, this actually makes MML more powerful. This section finishes by arguing that the deviation from strict metamodelling that this paper outlines does not contravene the mandatory requirements for UML 2.0 [omg00].

In a strict metamodelling architecture, every element of a model must be an instance of *exactly one* element of a model in the *immediate* next metalevel up [ak00]. As MML stands, it satisfies the 'exactly one' criterion, but every element does not instantiate an element from the immediate next metalevel up.

This is illustrated in Fig. 4 below. The concepts in the *model.concepts* packages (*e.g.* Class) and the concepts in the *instance.concepts* packages (*e.g.* Object) are all in the same metalevel (M3) since they are all part of MML. In line with the strict metamodelling mandate, instantiations of elements from the *model.concepts* packages belong to the metalevel below (M2) – these will be the elements that form the UML metamodel. However, instantiations of elements from the *instance.concepts* packages belong to the metalevel below that (M1), since these will form models that must satisfy the languages defined at the M2 level.

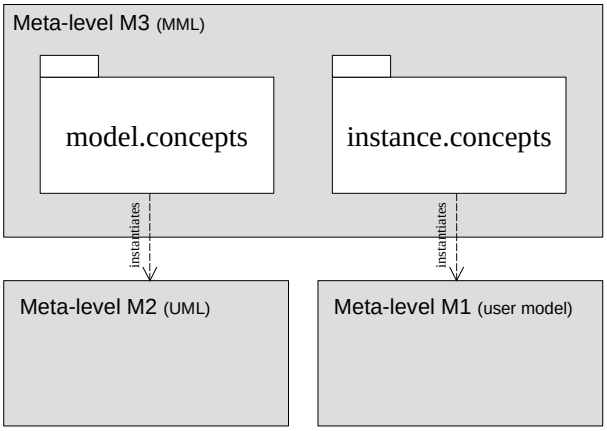


Fig. 4 Instantiation of the MML Metamodel

In fact, since UML extends MML (through specialisation), this pattern is repeated for the relationship between M2, M1 and M0. So the Four-Layer Metamodel Architecture does not in fact resemble the linear hierarchy shown in Fig. 5(a) but the nested structure shown in Fig. 5(b). If Fig. 5(a) was the true representation, any metalevel could only describe elements from the metalevel immediately below it; for example MML (level M3)

could describe elements from the UML metamodel (M2) but not elements from user models (M1). The nested architecture however means a model can describe elements from *every* metalevel below it. This is a very powerful feature since it means that if a tool implements the MML metamodel, then it can not only be used to define languages such as UML, but user models and objects as well. How this is done is the subject of subsequent sections.

(a) Linear **(b) Nested**

Fig. 5 Representations of the Four-Level Metamodel Architecture

Theoretically, the nested structure of the metamodel architecture (according to MML) is not confined to four levels – it could of course be applied at any number of levels, as depicted in Fig. 6. However, this only works if MML (or a specialisation of it) is represented at every level except the bottom two. There would be little value in this, but it is suggested here just to illustrate the fact that MML supports a very general architecture. In effect, the four-level metamodel architecture is a specialisation of this generic nested architecture.

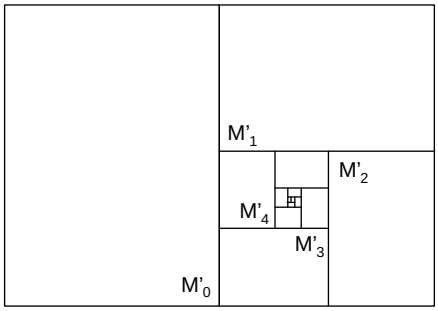


Fig. 6 Potential n-level Metamodel Architecture

In fact, the fundamental metalevel of this architecture is not the bottom level as suggested by the OMG metalevel 'M0', but the top level, since only the top level is represented by classifiers (*e.g.* classes and packages) – all other metalevels are represented by instances. An alternative notation for the levels of the metamodel architecture (M'_1 *etc.*) is introduced in Fig. 6 to emphasise this, and also to distinguish these levels from the four OMG metalevels.

This nested metamodel architecture might be seen as being at odds with the requirements set out by the OMG for the UML 2.0. However, the UML 2.0 Infrastructure RFP [omg00] states:

> Proposals shall specify the UML metamodel in a manner that is strictly aligned with the MOF meta-metamodel by conformance to a 4-layer metamodel architecture pattern. Stated otherwise every UML metamodel element must be an instance of exactly one MOF meta-metamodel element.

It should be noted that this does not specify a *strict* metamodel architecture, and every UML metamodel would be an instance of exactly one MML meta-metamodel element (where an MML meta-metamodel element is equivalent to a MOF meta-metamodel element). This paper therefore argues that the nested metamodel architecture outlined in this section fulfils the infrastructure requirements.

4. Mapping Between Metalevels

A key aspect of the nested metamodel architecture described above is that any model can be represented by another model at a metalevel below it without losing any information. The relationship between these two models can be described by a one-to-one information preserving mapping (arbitrarily referred to as '*G*'), as depicted in Fig. 7. Now as described in Section 2, MML makes a fundamental distinction between *model* concepts and *instance* concepts. However, any model element can in fact be thought of as an instance element; for example, the class *Dog* is a model element (as is any class), but is also an instance of the metaclass *Class*. Thus there are two representations of *Dog* (as a class and as an object), which are related by the mapping *G*. This section illustrates how the crucial notion of *meta*-instantiation can be modelled in the nested metamodel architecture by repeated application of the *G* mapping to a simple model.

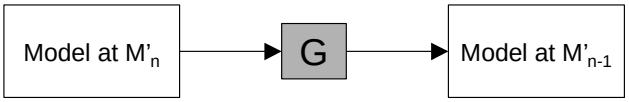
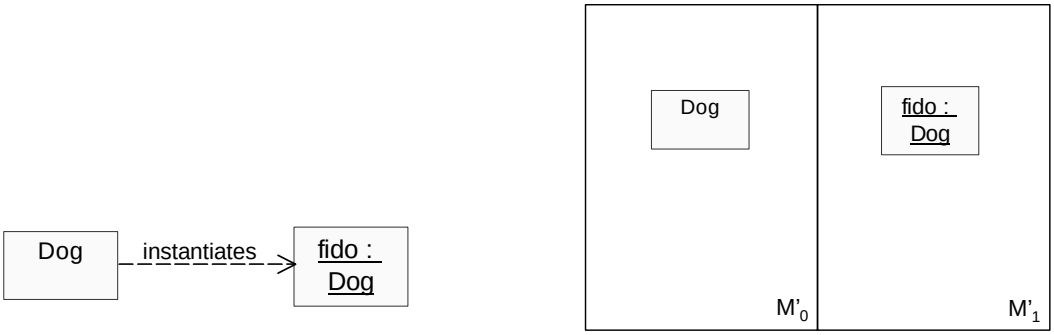


Fig. 7 Mapping Between Metalevels

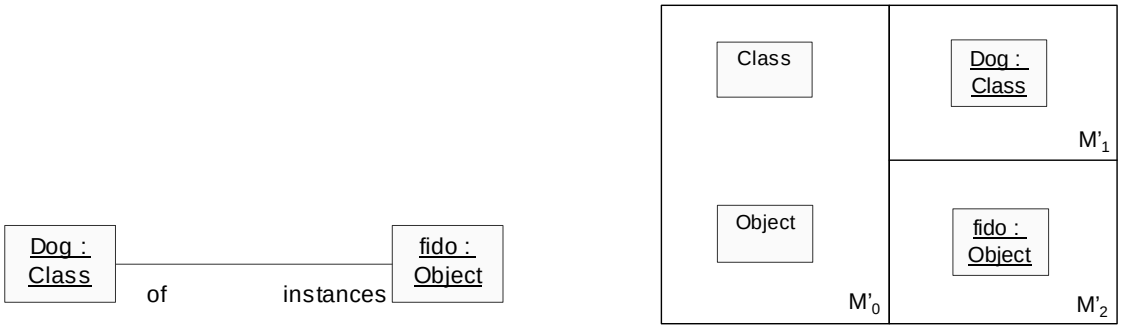
In the model depicted in Fig. 8(a) below, there is a single class *Dog*, and a single object *fido* which instantiates *Dog*. Because *Dog* is a class, it must belong to metalevel M´$_0$ (see Fig. 3), and because *fido* instantiates it, it belongs to M´$_1$. Fig. 8(b) emphasises the position of these elements within the metamodel architecture.



(a) The model **(b) Position in Metamodel Architecture**

Fig. 8 *Dog* and *fido* at Metalevels M´$_0$ and M´$_1$

This is the natural intuitive way to think of these elements, yet it is not the only way. Saying that *Dog* is a class and that *fido* is an object, is equivalent to saying that *Dog* instantiates the metaclass *Class* and that *fido* instantiates the metaclass *Object*, the metaclasses being defined in the UML metamodel. Thus the same information can be represented as shown in Fig. 9(a). *Dog* now belongs to the M´$_1$ level and fido to the M´$_2$ level (see Fig. 9(c)).



(a) The model **(b) Position in Metamodel Architecture**

Fig. 9 *Dog* and *fido* at Metalevels M´$_1$ and M´$_2$

There is a key relationship between the models depicted in Fig. 8(a) and Fig. 9(a). Both describe the same information, but at different metalevels. If a model is described to be at metalevel M'_n if it contains any elements at metalevel M'_n and below (hence the model in Fig. 5(a) is at metalevel M'_1 because it contains elements at levels M'_1 and M'_2), then there is a one-to-one information preserving mapping (*G*) between the representation of a model at one metalevel and its representation in the metalevel below it (see Fig. 7).

Thus for a model X:

$$X(M'_n).G = X(M'_{n-1}) \qquad \textbf{Equation (1)}$$

The mapping between a model and its representation two metalevels below is given as:

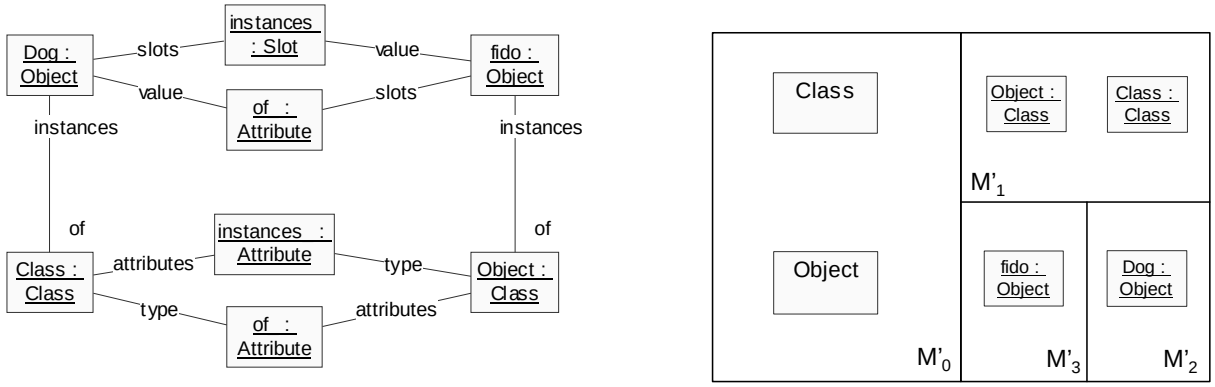
$$X(M'_n).G.G = X(M'_{n-2}) \qquad \textbf{Equation (2)}$$

And in order to translate to a metalevel up:

$$X(M'_n).G^{-1} = X(M'_{n+1}) \qquad \textbf{Equation (3)}$$

Note however that any model that contains classifiers (*i.e.* elements at level M'_0) cannot be mapped to higher metalevel.

The model in Fig. 8(a) can therefore be represented not just at the metalevel immediately below it (Fig. 9(a)), but also at the metalevel below that, as depicted in Fig. 10(a). It should be noted that *Dog* was represented first as a class, then as a *Class* object, and finally as an *Object* object. The UML metaclasses *Class* and *Object* are now represented at the M'_1 level, and the MML metaclasses *Class* and *Object* are represented at the M'_0 level.



(a) The model **(b) Position in Metamodel Architecture**

Fig. 10 *Dog* and *fido* at Metalevels M'_2 and M'_3

It can be seen that as the representation of a model descends the metalevels, the model representation increases in size and complexity, but nevertheless it can theoretically be represented at any metalevel down. In reality this should not be an issue since a user model is unlikely to be modeled at a level below M´2.

5. MML and the Metamodel Architecture

The previous section introduced the basic mechanism of mapping from one metalevel to another from the viewpoint of a very simple model. This section expands those ideas by applying them to a rather more complex model – the MML metamodel itself.

The UML metamodel specialises the MML metamodel, so there is a subset of the UML metamodel that is the MML metamodel. It is no coincidence that a special case arises when the mapping *G* is applied to the MML metamodel: the result is an instantiation of the MML metamodel. Fig. 11 depicts a small subset of the UML metamodel represented at the M'_1 level, as well as an example M'_2 user model and M'_3 user object. This could be thought of as a model created in a tool that implements the MML metamodel (at the M'_0 level).

There are some interesting points to note regarding reflection and instantiation when looking at a model such as this. In the model depicted in Fig. 11, *fido* can be thought of as an instance of both *Dog* and *Object*, but if (as MML suggests) an object can only be 'of' a single class, how can these two notions of instantiation be modelled? The crucial idea is that whenever a query is made on a model (such as 'what is *fido*.of?'), that query has an associated metalevel as well as the model elements themselves. This is in contrast with current thinking that queries can only be made about a model from the top metalevel (M'_0).

If this query is applied at the M'_0 level (*i.e.* from the point of view of the MML metamodel), the answer is returned as *Object*. However, the '*fido*.of = *Dog*' relationship is also modelled, but at the M'_1 level (the UML metamodel viewpoint). Thus if the mapping G is applied to the query '*fido*.of' itself, the result would be the *Dog* object. The *G* mapping can similarly be applied to any operation, whether side effect free or not.

This suggests that an explicit reflection facility does not strictly need to be included in MML (there is currently a reflection profile); all the mechanisms are already in place for such relationships in the current MML metamodel. For example the MML metamodel currently has a constraint that checks that a class is an instance of the metaclass *Class* (Section 4.7.3 in [bccek00]), but this is redundant since in the *staticCore.model.concepts* class diagram, *Class* is defined as a class. These and similar constraints are retained when they are mapped between metalevels, since *G* is an information preserving mapping.

An implementation of the *G* mapping would be a powerful feature for any tool. One use concerns the multiple representations of a model within a modelling tool. There is no reason why the metalevel most appropriate for visualising a model is the metalevel most appropriate for storing it (just as numbers in different bases are appropriate for humans and computers). The *Dog* and *fido* model from Section 4 is in its most comprehensible form in Fig. 8(a), but it is probably preferable to represent everything as an object within the tool as in Fig. 9(a). A model could thus be stored at one metalevel but visualised at another. This is in fact what both Rational Rose and MMT (the tool that implements MML) do already – model elements that are entered in a tool as a classes are actually represented as objects. Implicit in these tools is the functionality to not only map concepts to (concrete) syntax, but also to map between one metalevel and another.

Another potential use of *G* would be the wholesale transformation of metamodels. Once the metamodel for a language such as UML or a profile has been proven to be MML compliant, it can be translated up from the M'_1 to the M'_0 level by applying G^{-1}. In its translated form, user models would sit at the M'_1 level and user objects would sit at the M'_2 level (so *fido*.of would now return *Dog* rather than *Object*). The metamodel would now be in a form suitable for conformance checking of any tools that would claim to implement it.

Thus the mapping *G* has three key uses:

- to translate queries and manipulations on models to any metalevel.
- to translate representations of models 'on the fly' to a metalevel most appropriate for visualising them (*e.g.* a *Dog* object is visualised as a *Dog* class).
- to translate a language metamodel from the M'_1 level to the M'_0 level resulting in a usable specification for the language that can be used for conformance checking of tools.

This mapping is so fundamental to MML that it should be brought out explicitly. It should not be part of MML, rather it should be a part of the Meta-Modeling Facility, something that defines some core functionality for any tool that is MML compliant.

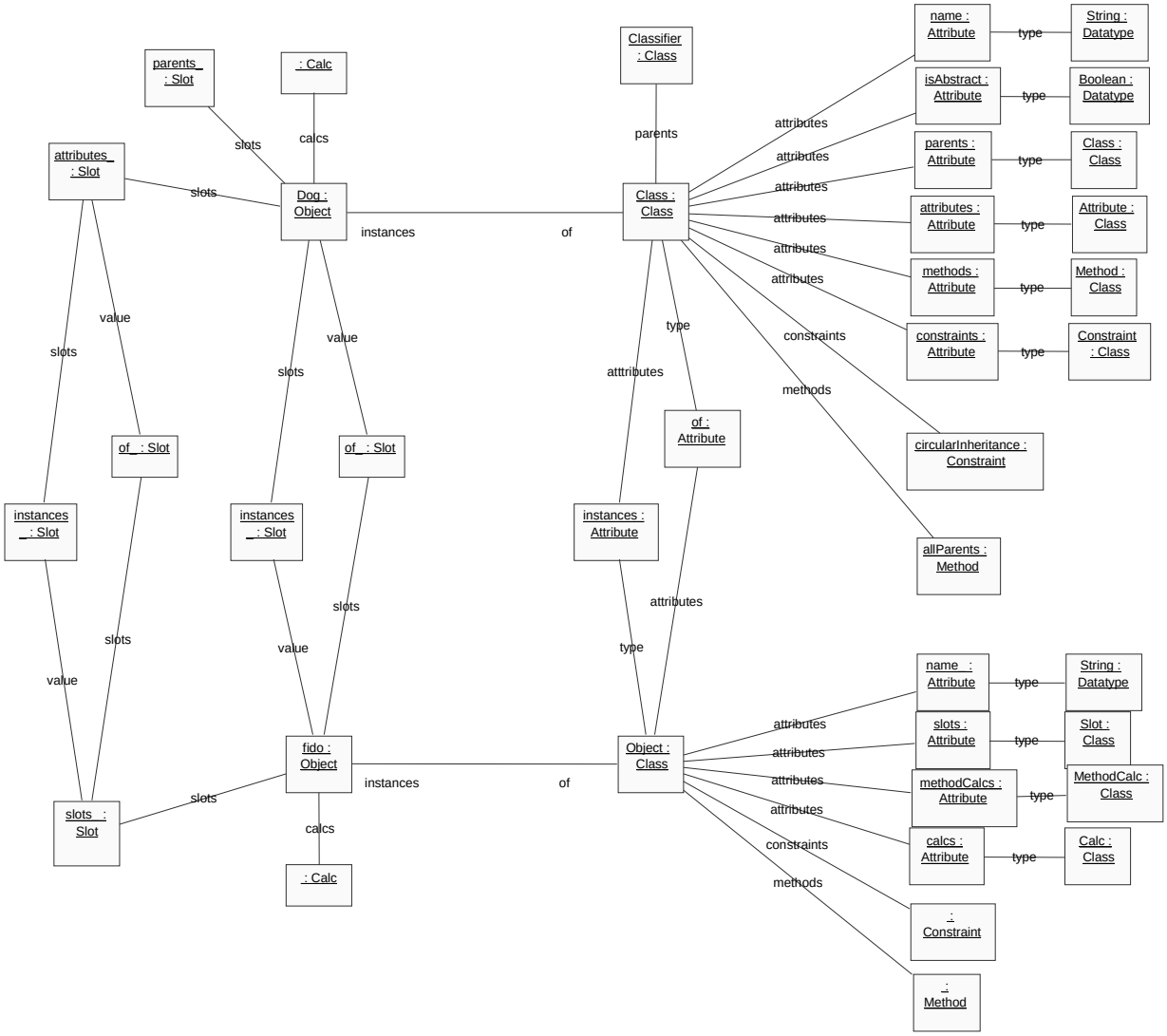


Fig. 11 An Instance of the MML Metamodel

6. Conclusion

The Meta-Modelling Language (MML) is a static object-oriented modelling language whose focus is the declarative definition of other languages such as UML. Its development arose out of the needs to rearchitect UML as a family of languages, and to provide a precise unambiguous semantics for UML; these needs are to be realised in the next version (2.0) of UML. MML takes a metamodelling approach to defining languages, in line with the OMG's requirement that UML 2.0 must be aligned with their 4-layer metamodel architecture. However, the architecture supported by MML cannot be described as a *strict* metamodel architecture, since not all elements instantiate elements from the *immediate* metalevel up. Instead, MML supports a powerful nested metamodel architecture, which this paper argues does not contravene the mandatory requirements for UML 2.0.

A key aspect of this architecture is that a model can be represented at any metalevel. The transformation between the representation of any model at one metalevel and its representation at the metalevel below it can be described by an information preserving one-to-one mapping (*G*). This mapping can also be applied to queries and manipulations of models, which enables reflection type information to be determined without the need of an explicit reflection facility in MML. In fact many of the constraints defined in the *reflection* package in MML are redundant.

The *G* mapping could also be used 'on the fly' by a tool to convert its internal representation of a model to a metalevel more appropriate for visualisation, and to convert a language metamodel to the topmost metalevel to produce a usable specification for that language or profile.

It is proposed that this *G* mapping should not be part of the MML metamodel, but defined in a separate part of the Meta-Modeling Facility. The next stage of work is to precisely define the *G* mapping, and to verify its one-to-one information preserving nature.

References

[ak00] Atkinson C., Kühne T
Strict Profiles: Why and How
In [eks00], 2000

[bccek00] Brodsky S., Clark A., Cook S., Evans A., Kent S.
A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach
Available at [puml], 2000

[ceks00] Clark T., Evans A., Kent S., Sammut P.
The MMF Approach to Engineering Object-Oriented Design Languages
Available at [puml]

[cmwwe99] Cook S., Mellor S., Warmer J., Wills A., Evans A. (moderator)
Advanced Methods and Tools for a Precise UML
Available at [puml]

[dw98] D'Souza D., Wills A.
Objects, Components and Frameworks with UML: The Catalysis Approach
Addison-Wesley, 1998

[eks00] Evans A., Kent S., Selic B.
Proceedings of <<UML>> 2000 – The Unified Modeling Language, Advancing the Standard: 3rd International Conference (LNCS 1939)
Springer-Verlag, 2000

[k99] Kobryn C.
UML 2001 : A Standardization Odyssey
Communications of the ACM, 1999 [42, 10]

[omg] **OMG web site**
http://www.omg.org/

[omg99] **OMG Unified Modeling Language Specification**
Available at [omg], 1999

[omg00] **Request for Proposal: UML 2.0 Infrastructure RFP**
Available at [uml2wg], 2000

[puml] **Precise UML group web site**
http://www.puml.org/

[rjb99] Rumbaugh J., Jacobson I., Booch G.
The Unified Modeling Language Reference Manual
Addison-Wesley, 1999

[uml2wg] **UML 2.0 Working Group web site**
http://www.celigent.com/omg/adptf/wgs/uml2wg.htm