

Mapping between Levels in the Metamodel Architecture

José Álvarez¹, Andy Evans², Paul Sammut²

¹ Dpto. de Lenguajes y Ciencias de la Computación, University of Málaga,
Málaga, 29071, Spain
alvarezp@lcc.uma.es

² Dept. of Computer Science, University of York,
Heslington, York, YO10 5DD, United Kingdom
andye@cs.york.ac.uk, pauls@cs.york.ac.uk

Abstract. The Meta-Modeling Language is a static object-oriented modeling 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 metamodeling approach to language definition, it cannot be described as strict metamodeling. This has significant implications for 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 metamodeling imposes. In this nested architecture, the transformation of any model between its representations at two adjacent metalevels can be described by an information preserving one-to-one mapping. This mapping, which can itself be defined in UML, provides the basis for a powerful area of functionality that any potential metamodeling tool should seek to exploit.

1. Introduction

The Unified Modeling Language (UML) has been rapidly accepted as a standard notation for modeling 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 [1]. UML is therefore evolving, with an impending revision (UML 2.0 [2]) 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 [3]. 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, 4]. An alternative approach (which the OMG has set as a mandatory requirement for UML 2.0 [5])

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) [6] 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, 4]. 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 [5]. 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 viewed as 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 metamodeling) loosely approximates this [6], 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 [7] have proposed ‘rearchitecting UML as a family of languages using a precise object-oriented metamodeling approach’ [8]. The foundation of their proposal is the Meta-Modeling Facility, which comprises the Meta-Modeling Language (MML), an alternative M3 layer language for describing modeling languages such as UML, and a tool (MMT) that implements it.

Whilst MML takes a metamodeling approach, it cannot be described as ‘strict’ metamodeling, and this has profound implications for the metamodel architecture. This paper (an expanded version of an earlier paper [9]) attempts to provide clarification of the exact nature of this architecture, in particular demonstrating that the linear hierarchical model is inappropriate; instead an alternative model for this architecture is offered. 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 metamodeling tool.

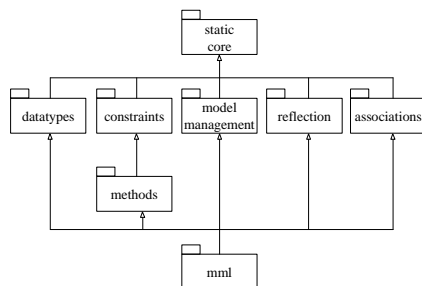
The paper is structured as follows: Section 2 gives an overview of MML; Section 3 describes the ‘nested’ metamodel architecture supported by MML; Section 4 defines the mapping that describes the transformation between metalevels in this architecture; Section 5 demonstrates the application of this mapping with a simple example; and Section 6 outlines conclusions and further work.

2. The Meta-Modeling Language

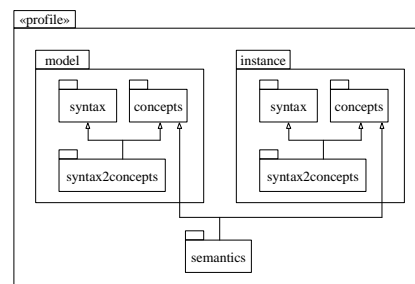
MML is a ‘static OO modeling language that aims to be small, meta-circular and as consistent as possible with UML 1.3’ [10]. The MML metamodel partitions the fundamental components of language definition into separate packages. It makes two key orthogonal distinctions [8]: between ‘model’ and ‘instance’, 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 MML metamodel defines the minimum number of concepts needed to define itself, and wherever possible uses the existing syntax and concepts of UML 1.3. This meta-circularity eliminates the need for another language to describe MML, thus closing the language definition loop. The clean separation of language components and the mappings between them is fundamental to the coherence and readability of the MML metamodel.

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 [11]. In the Catalysis approach, packages may be specialised in the same way as classes; all the contents of the parent package are inherited by the child package, where they can themselves be specialised. This allows the definition of language components to be developed incrementally over a number of packages [10].



1a. Profile Packages



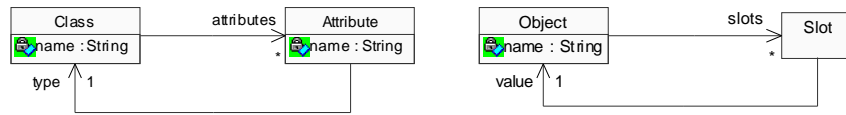
1b. Profile Framework

Fig. 1. The MML Architecture

The architecture of the MML metamodel is founded on the separation of language components and the notion of package specialisation described above. The metamodel is split into a number of key profile packages, each of which describe a fundamental aspect of modeling languages, and which are combined through an inheritance hierarchy to give the complete MML definition (Fig. 1a adapted from [8]).

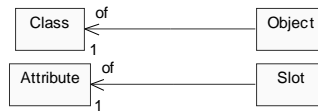
In turn, each of the profile packages shares the same framework of sub-packages as depicted in Fig. 1b (adapted from [8]).

These sub-packages represent the basic language components and the appropriate mappings between them, and contain modeling constructs such as classes and associations. A simplified form of the *static core* profile is depicted in Fig. 2, by way of illustration. The *model.concepts* packages describe 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). Constraints are applied to the constructs where appropriate through the use of OCL (the Object Constraint Language).



2a. *model.concepts* Package

2b. *instance.concepts* Package



2c. *semantics* Package

Fig. 2. Simplified 'Static Core' Profile

A key idea to grasp for the discussion that follows is that MML serves two distinct functions: it is both a metamodeling 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.

3. Strict Metamodeling and the Four-Layer Architecture

This section argues that MML does not in fact fit into a *strict* metamodeling architecture, and that instead of being a problem, this actually makes MML more powerful. It also argues that this deviation from strict metamodeling does not contravene the mandatory requirements for UML 2.0 [5].

In a strict metamodeling architecture, every element of a model must be an instance of *exactly one* element of a model in the *immediate* next metalevel up [12]. 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. 3 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 metamodeling 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.

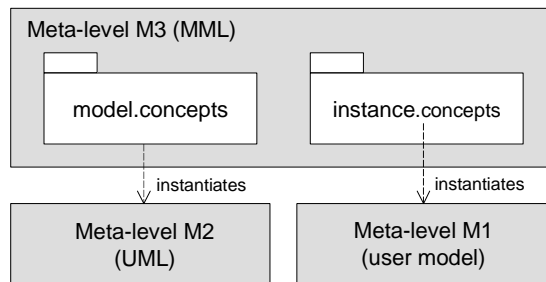
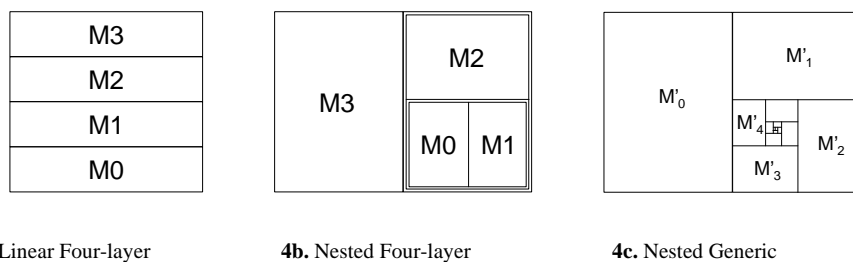


Fig. 3. Instantiation of the MML Metamodel

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. 4a but the nested structure shown in Fig. 4b. If Fig. 4a 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). In the nested architecture however, 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 achieved is the subject of subsequent sections.



4a. Linear Four-layer

4b. Nested Four-layer

4c. Nested Generic

Fig. 4. Metamodel Architectures

Theoretically, the nested structure of the metamodel architecture (according to MML) is not confined to four levels; it could be applied at any number of levels, as depicted in Fig. 4c. In effect, the four-level metamodel architecture is a specialisation of this generic nested 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. 4c 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 [5] 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 element is intended to 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

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, a class can always be thought of as an instance of the metaclass *Class*). These two representations of the same entity are related by a one-to-one information preserving mapping, arbitrarily referred to as ‘G’ (Fig. 5a). In effect the *G* mapping represents the crucial notion of *meta*-instantiation. In fact an entire model can be represented by another model at a metalevel below it through the application of this same *G* mapping (Fig. 5b).



5a. Mapping between Model Elements

5b. Mapping between Models

Fig. 5. Mapping between Metalevels

Thus for a model X:

$$X(M'_n).G = X(M'_{n+1}) ; \quad (1)$$

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

$$X(M'_n).G.G = X(M'_{n+2}) ; \quad (2)$$

and in order to translate to a metalevel up:

$$X(M'_n).G^{-1} = X(M'_{n-1}) . \quad (3)$$

It can be seen that model information does not have two representations, but an infinite hierarchy of representations, corresponding to the nested generic metamodel architecture of Fig. 4c. It will be shown that the further down a model is in the metalevel hierarchy, the larger and more complex it is. However, as with the metamodel architecture, only the first few metalevel representations hold any real practical value.

How might this mapping be modeled? One approach would be to include a method in every model element that would create the appropriate element to represent it at the next metalevel below; for example *Class* would have a method *G()* that would create an instance of *Object* with a slot '*of = Class*'. This is an intuitive way of viewing the transformation, but would result in a single confusing model that contained both representations simultaneously. Instead what we need is a way of separating these two representations. This can be achieved by modeling the *G* mapping as a package that specialises both the metamodel for elements that can appear in a model at M'_n and the metamodel for elements of its equivalent model at M'_{n-1} (Fig. 6). This mapping package has visibility (through specialisation) of both metamodels, so it can define how one model is a valid *G* transformation of another model by means of appropriate associations between elements of the metamodels, and constraints on those associations. This model of mapping, which is also used in the *semantics* packages of the MML metamodel [8] and will provide the basis of the OMG's Model Driven Architecture [13], is based on the Catalysis [11] model of refinement.

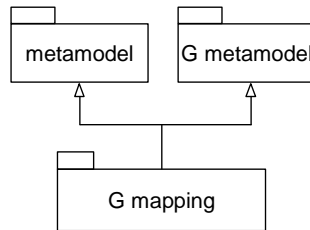


Fig. 6. The *G* Mapping Package

The two metamodels in Fig. 6 are in fact identical (one is a copy of the other), but they must be shown as separate packages as the mapping needs to refer to elements from two metamodels. For illustrative purposes, this paper assumes the metamodel (*i.e.* the M'_0 layer in the metamodel architecture) to be that of Fig. 2. Thus the only valid elements are classes, attributes, objects and slots. There are obviously severe limitations to the expressibility of this metamodel, but fewer concepts aid clarity. The following points must be considered regarding this metamodel:

- the name attribute on an Object instance is optional, so unnamed objects are valid;

- datatypes (such as String) are not modeled explicitly for the sake of simplicity, but do feature in the associated models;
- slots do not have a name; instead they are simply 'of' a named attribute. Thus in Fig. 5a, the syntax '*name* = Dog' actually represents an unnamed slot 'of' the attribute *name*, whose value is 'Dog'. This relationship between slots and attributes is introduced as it is closer to the relationship between objects and classes, and thus provides consistency between different elements as the *G* mapping is applied.

The associations of the *G* mapping are defined in Fig. 7; in this class diagram, the *Object* metaclass in *G metamodel* has been renamed as *GObject* to distinguish it from *Object* in *metamodel*. Note how every element in *metamodel* maps to *Object* in *G metamodel*. Translating to natural language, the *G* mapping maps classes, attributes, objects and slots to instances of *Class*, *Attribute*, *Object* and *Slot*.

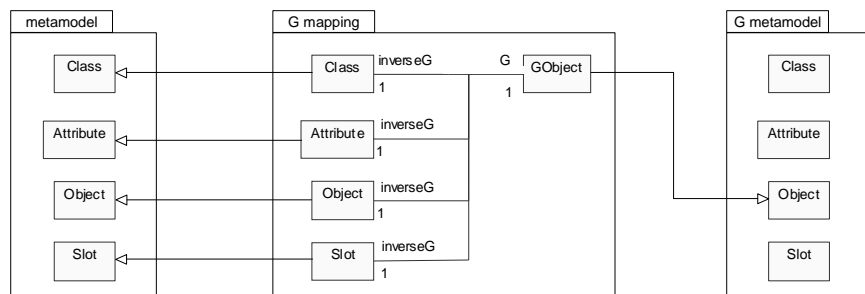


Fig. 7. The *G* Mapping Class Diagram

The OCL constraints on each of the metaclasses are given in Table 1. There is considerable commonality in these constraints and these could potentially be abstracted into a generic pattern. It should be noted that these constraints only cover one direction of the *G* mapping transformation (*i.e.* what models are valid *G* transformations of other models), but constraints could be added to cover the reverse direction (*i.e.* what models are valid inverse *G* transformations of other models). One limitation that arises through the use of constraints (or methods) is that constraints only talk about *instances* of their associated model element, not model elements themselves. Because of this, there is no way of translating down from or up to level M'_0 (the true model level) – thus Equations 1 to 3 above only hold for $n > 0$.

Table 1. The *G* Mapping Constraints

<p>context Class inv: G.of.name = "Class" G.slots -> exists (s s.of.name = "name" and s.value = self.name) self.attributes -> forAll(a G.slots -> exists (s s.of.name = "attributes" and s.value = a.G</p>	<p>context Attribute inv: G.of.name = "Attribute" G.slots -> exists (s s.of.name = "name" and s.value = self.name) G.slots -> exists (s s.of.name = "type" and s.value = self.type.G</p>
<p>context Object inv: G.of.name = "Object" G.slots -> exists (s s.of.name = "of" and s.value = self.of.G G.slots -> exists (s s.of.name = "name" and s.value = self.name) self.slots -> forAll(s G.slots -> exists (sl sl.of.name = "slots" and sl.value = s.G</p>	<p>context Slot inv: G.of.name = "Slot" G.slots -> exists (s s.of.name = "of" and s.value = self.of.G G.slots -> exists (s s.of.name = "value" and s.value = self.value.G</p>

5. Application of the *G* Mapping

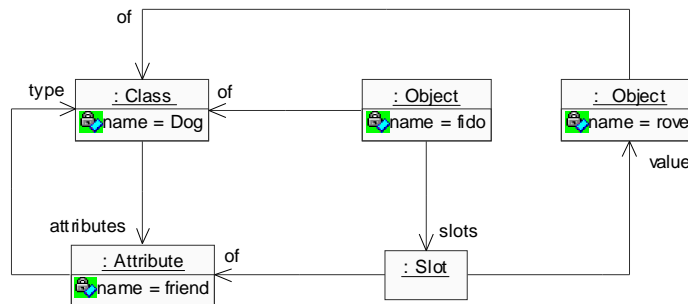
The application of the *G* mapping can be demonstrated by introducing a simple example. First however, a subtle but important point must be made. When a class such as *Dog* in Fig. 8 is drawn in a class diagram, it must conform to some metamodel. It is the metamodel therefore that resides at the top level (M'_0), so only the metamodel can contain true classes – every other level must be represented by instances. Therefore the class box for *Dog* does not in fact represent a true class, but an instance of the metaclass *Class*. It is no coincidence that this difference between the apparent representation of an element and its ‘true’ representation is modeled by the *G* mapping. It is important therefore to realise that every class diagram and object diagram has the *G* mapping implicit in its syntax.



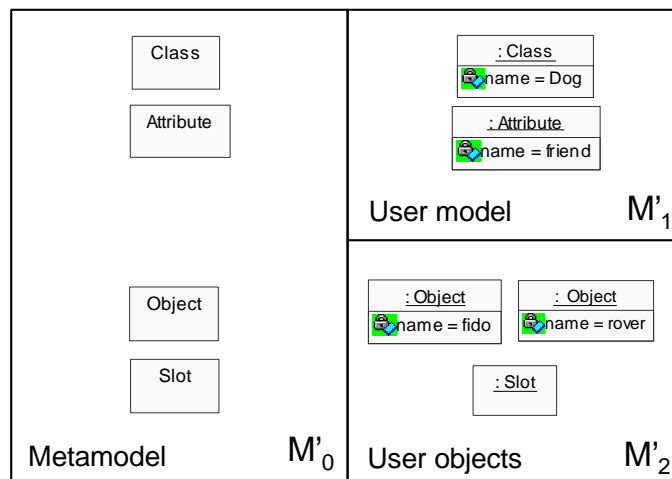
Fig. 8. User Model and User Objects

To illustrate this, in Fig. 8 we have a class *Dog*, and two instantiations of *Dog*. However, as described above, if the metamodel of Fig. 2 resides at the M'_0 level, then the true representation of the *Dog* model and instance is that of Fig. 9a. It is helpful

to see how the elements in this view of the model fit in the metamodel architecture; this is shown in Fig. 9b. This is the view of a system that a modeling tool takes – it implements the metamodel, and translates user models and objects down to the M'_1 and M'_2 level respectively.



9a. Model



9b. Position in the Metamodel Architecture

Fig. 9. User Model and User Objects at Metalevels M'_1 and M'_2

A metamodeling tool might view a system quite differently however. It would implement a meta-metamodel (such as the MOF or MML metamodel), and take a metamodel (such as the UML metamodel) as an instance at the M'_1 level. User models and user objects can still be included in the system, but they must be translated a further metalevel down. If the basic Fig. 2 metamodel is assumed to be both the meta-metamodel and the metamodel, then it too must be translated a metalevel down (Fig. 10), so that it can be represented at the M'_1 level.

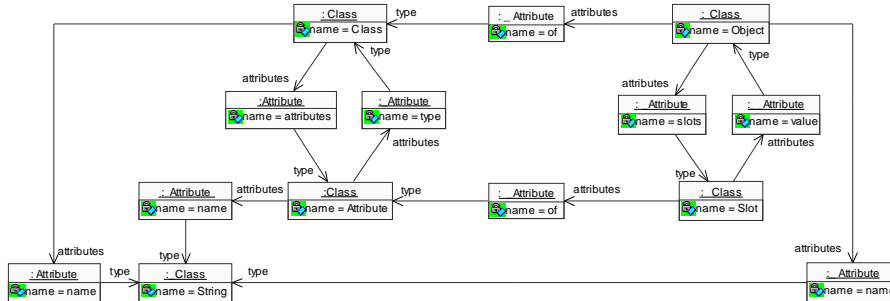
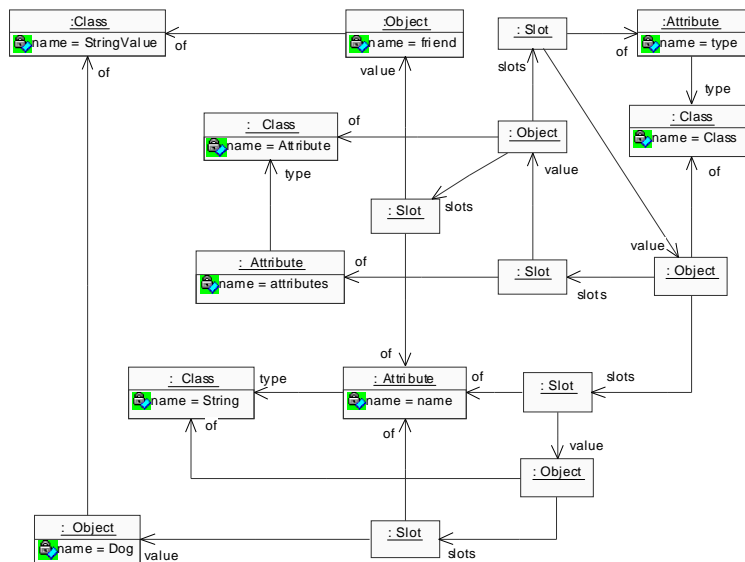
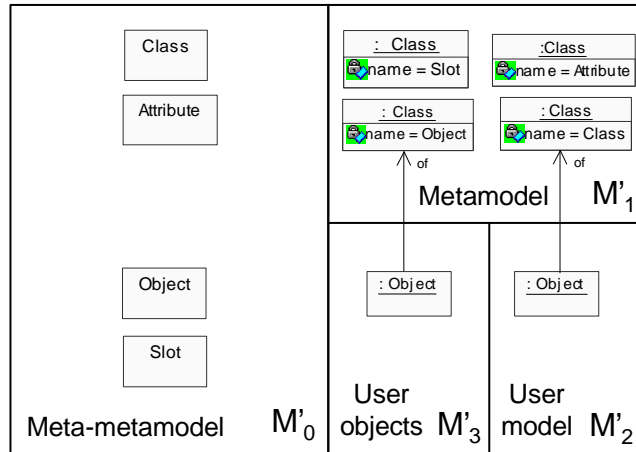


Fig. 10. Metamodel at Metalevel M'1

The corresponding view of the user model and user objects is now two metalevels removed from that of Fig. 8. The complete model is too large to include here, so only a small segment (representing the *Dog* class only) is shown in Fig. 11a; the position of the various elements of the system is again shown in Fig. 11b. It can thus be seen how the complexity and size of a model increases rapidly the lower it is represented in the metalevel hierarchy. However, the gain is the potential for a metamodeling tool where the metamodel can be changed as required, so long as it conforms to the meta-model (the only fixed model in the system), which can also be used for user modeling. A user would never have to understand a model such as that of Fig. 11a; their model would be translated on-the-fly to the representation appropriate for them (Fig. 8).



11a. Model



11b. Position in the Metamodel Architecture

Fig. 11. User Model and User Objects at Metalevels M'_2 and M'_3

In the model discussed in this section, *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 modeled? 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). In the Fig. 11 view of the system, if the '*fido.of*' query is applied at the M'_0 level, the answer is returned as *Object*. However, the '*fido.of = Dog*' relationship is also modeled, but at the M'_1 level. 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.

6. Conclusion

The Meta-Modeling Language (MML) is a static object-oriented modeling language whose focus is the declarative definition of other languages such as UML. MML takes a metamodeling approach to defining languages, in line with the OMG's requirement that UML 2.0 must be aligned with their Four-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 would have the following potential uses in a metamodelling tool:

- to enable metamodeling tools to be used as modeling tools;
- to translate representations of models ‘on the fly’ to a metalevel most appropriate for visualising them (*e.g.* a *Class* object is visualised as a class);
- to translate queries and manipulations on models to any metalevel;

This mapping is so fundamental that it should be brought out explicitly. It should not be part of MML, rather it should be a separate part of the Meta-Modeling Facility, something that defines some core functionality for any tool that is MML compliant. It is planned to continue this work by implementing a simple metamodeling tool that is able to translate models using the G mapping.

References

1. Kobryn C.: UML 2001 : A Standardization Odyssey. Communications of the ACM, 1999 [42, 10]
2. UML 2.0 Working Group web site:
<http://www.celigent.com/omg/adptf/wgs/uml2wg.htm>
3. Rumbaugh J., Jacobson I., Booch G.: The Unified Modeling Language Reference Manual. Addison-Wesley, 1999
4. Cook S., Mellor S., Warmer J., Wills A., Evans A. (moderator): Advanced Methods and Tools for a Precise UML. Available at [7]
5. Request for Proposal: UML 2.0 Infrastructure RFP. Available at [2], 2000
6. OMG Unified Modeling Language Specification. Available at [15], 1999
7. Precise UML group web site: <http://www.puml.org/>
8. 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 [7], 2000
9. Alvarez J., Evans A., Sammut P.: MML and the Metamodel Architecture. Available at [7], 2001
10. Clark T., Evans A., Kent S., Sammut P.: The MMF Approach to Engineering Object-Oriented Design Languages. Available at [7]
11. D’Souza D., Wills A.: Objects, Components and Frameworks with UML: The Catalysis Approach. Addison-Wesley, 1998
12. Atkinson C., Kuhne T.: Strict Profiles: Why and How. In [14], 2000
13. Soley R. & OMG: Model Driven Architecture White Paper. Available at [15], 2001
14. 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
15. OMG web site: <http://www.puml.org/>