Université Nice SOPHIA ANTIPOLIS

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

OPEN IMPLEMENTATION OF UML META-MODEL(S), MAKING META-MODELING AND META-PROGRAMMING MEET

Pascal RAPICAULT, Jean-Paul RIGAULT

Projet RAINBOW

Rapport de recherche
I3S/RR–2002-24–FR

Juillet 2002

LABORATOIRE I3S: Les Algorithmes / Euclide B – 2000 route des Lucioles – B.P. 121 –
06903 Sophia-Antipolis Cedex, France – Tél. (33) 492 942 701 – Télécopie : (33) 492 942 898
http://www.i3s.unice.fr/I3S/FR/

RÉSUMÉ :

Depuis son apparition, le méta-modèle UML s'avère être un mécanisme intéressant pour décrire la sémantique des modèles UML et de ses extensions. Cependant l'approche est purement déclarative. En particulier, le méta-modèle ne définit ni comportement ni opération. Ainsi l'implémentation d'outils UML et d'extensions ne peuvent pas directement tirer profit de celui-ci. Ce rapport propose un premier pas vers une implémentation ouverte du méta-modèle d'UML, rendant possible l'implémentation d'extensions sémantiquement sûres. Cette implémentation est basée sur MML (Meta-Modeling Language défini par le groupe pUML) et définit méta-opérations et méta-méthodes pour manipuler les éléments de modélisation et opérationnaliser les contraintes du méta-modèle. Puisque le méta-modèle UML se décrit en UML et que MML est réflexif, cette implémentation est aussi reflexive et constitue une sorte de méta-objet protocol (un MOP). Nous illustrons l'utilisation de notre MOP grâce à trois exemples qui couvrent tous les types d'extensions proposés par UML. Ces exemples sont : la factorisation d'attributs dans une nouvelle super-classe (ajout de comportement, amélioration des fonctionnalités d'un outil), définition d'un profil Java simple (spécialisation d'éléments de modélisation existants), ajout d'information sur les auteurs et les versions (introduction de nouveaux éléments de modélisation). En conclusion nous présentons les problèmes d'implémentations rencontrés lors de l'utilisation de techniques de méta-programmation dans le contexte de la méta-modélisation. De plus nous montrons l'importance du choix du langage d'implémentation dans la réalisation d'un méta-protocole. Enfin, nous discutons les bénéfices que pourraient tirer les outils d'une telle implémentation du méta-modèle, et le travail qui reste à faire pour créer un outil complètement opérationnel.

MOTS CLÉS :

UML, Méta-modélisation, Méta-programmation

ABSTRACT:

From the origin the UML meta-model has been a valuable mechanism to describe the semantics of UML models as well as to extend UML itself. However the approach is purely declarative. In particular the meta-model does not define any behavior nor operation. Thus the implementation of UML CASE tools cannot directly benefit from it, nor can the realization of extensions. This paper proposes a first attempt toward an Open Implementation of UML meta-models, making it possible to implement semantically sound extensions. The implementation relies on the MML (Meta-Modeling Language defined by the pUML group) and defines meta-operations and meta-methods to manipulate the modeling elements and to operationalize the meta-model constraints. Due to the fact that the UML meta-model is itself expressed in UML and that MML offers reflective capabilities, this implementation is naturally reflective and thus constitutes a sort of Meta-Object Protocol (a MOP). We illustrate the use of our MOP by implementing all possible kinds of extensions to UML with three examples: factorizing attributes into a new superclass (addition of behavior, improving tool features), defining a sketch of a Java profile (specialization of existing modeling elements), and adding version and author information (introduction of new modeling elements). In conclusion we present the implementation problems that we encountered when using meta-programming techniques in a meta-modeling context. We show the importance of the choice of the meta-protocol implementation language. Finally we discuss the benefit that CASE tools could draw from such an implementation of the meta-model, and the work that needs to be done to achieve a fully operational integration into CASE tools.

KEY WORDS :

UML, Meta-modeling, Meta-programming

Open Implementation of UML Meta-Model(s)

Making Meta-Modeling and Meta-Programming Meet

Pascal Rapicault[1,2] and Jean-Paul Rigault[1,3]

[1] RAINBOW and SPORTS Projects, I3S Laboratory, University of Nice Sophia Antipolis and CNRS (UMR 6070)
F-06902 Sophia Antipolis Cedex, France
[2] Object Technology International, Inc. (OTI)
[3] ORION Project, INRIA Sophia Antipolis, France

Abstract. From the origin the UML meta-model has been a valuable mechanism to describe the semantics of UML models as well as to extend UML itself. However the approach is purely declarative. In particular the meta-model does not define any behavior nor operation. Thus the implementation of UML CASE tools cannot directly benefit from it, nor can the realization of extensions.
This paper proposes a first attempt toward an Open Implementation of UML meta-models, making it possible to implement semantically sound extensions. The implementation relies on the MML (Meta-Modeling Language defined by the pUML group) and defines meta-operations and meta-methods to manipulate the modeling elements and to operationalize the meta-model constraints. Due to the fact that the UML meta-model is itself expressed in UML and that MML offers reflective capabilities, this implementation is naturally reflective and thus constitutes a sort of Meta-Object Protocol (a MOP).
We illustrate the use of our MOP by implementing all possible kinds of extensions to UML with three examples: factorizing attributes into a new superclass (addition of behavior, improving tool features), defining a sketch of a Java profile (specialization of existing modeling elements), and adding version and author information (introduction of new modeling elements).
In conclusion we present the implementation problems that we encountered when using meta-programming techniques in a meta-modeling context. We show the importance of the choice of the meta-protocol implementation language. Finally we discuss the benefit that CASE tools could draw from such an implementation of the meta-model, and the work that needs to be done to achieve a fully operational integration into CASE tools.

1 Introduction : Meta-Modeling in UML

1.1 The Role of a (Meta-)Model

Generally speaking a meta-model is a model of models. As such it should fulfill the traditional four aims of modeling, as expressed, for instance, in [3]:

1. visualize the system,
2. specify the structure and the behavior of the system,

3. help in building the system,
4. document the decisions made while building the system.

But here "the system" means an other (lower level) model.

The *Unified Modeling Language* (UML) comes with a meta-model [19], itself expressed in a combination of UML, natural language text, and OCL constraints. This UML meta-model (UML-MM for short) may help in visualizing the models (item 1), since it defines the graphical structure and the general organization of models. Documenting the decisions (item 4) would mean for instance supporting history mechanisms, versioning, journaling of models. This is not (currently) part of the UML-MM, and was not even intended to be.

The key items are in fact the middle ones (items 2 and 3). The UML-MM is well suited to express the structure of UML models. It describes the entities constituting the models together with the constraints they must respect and the rule to assemble them. By doing so it defines formally the notion of a *valid* UML model. The validity of a model may then be checked afterwards by CASE tools, even though the current tools are less than satisfactory in this matter.

However, the UML-MM does not (yet) try to describe the behavior of the UML models. To do so one would need to describe (within the meta-model) things like code generation, execution and simulation of model instances, etc. A step in this direction might be in the definition of some kind of "action language" for UML [5]. Another step is constituted by the so called "executable UML" models as implemented by commercial tools such as Rational Rose Real Time [20] or I-Logix Rhapsody [12, 7].

Does the UML-MM help in building UML models? Strictly speaking, this would mean that the UML-MM is one (the?) UML model for UML CASE tools. Few tools, if any, rely directly on the UML-MM. Some people are even afraid by a long history of meta-CASE tools failures! But are we in the same context with UML? Most of those meta-CASE tools had to be developed together with their meta-models, whereas in the case of UML, the meta-model definition preexists.

To be of some help for tool building, the meta-model should not only describe the structure of valid models, but also the valid operations and transformations that may be performed on models. Clearly this is not the current situation for UML-MM. This paper will show how adding behavior to the UML meta-model may help to use UML-MM to model UML CASE tools.

1.2 Extensibility of (Meta-)Models

The extensibility of meta-models is an important issue. UML currently provides two extension mechanisms:

- the *light weight* extensions, where one may introduce new stereotypes (sub-classes of existing meta-classes), new tagged values (adding meta-attributes), or new OCL constraints;
- the *heavy weight* extensions, where it is possible to define new meta-entities (*e.g.*, new meta-classes not necessarily deriving from existing ones) or to modify existing ones deeply (meta-attributes, meta-operations, meta-associations, etc.).

These extension mechanisms are becoming more and more important. Indeed, UML is now recognized not as a unique language, but as a *family* of different languages [6]. The emerging notion of an UML *profile*, either vertical (targeted to a particular language or implementation) or horizontal (dedicated to a particular domain of applications), requires precisely defined extension mechanisms. Thus, we must define the valid manipulations and operations on the meta-model itself. Since the UML-MM is given in UML, this calls for some kind of reflection mechanism on the meta-model, a Meta-Object Protocol (MOP) [14]. This will be detailed in section 2.1.

At this time most tools support extension mechanisms in a crude form, very often by wiring them into the tool implementation. If you want to define a new stereotype, most tools will allow you to define its label, as a character string. If you wish to associate an icon with your stereotype, you will have to modify some initialization file (`.ini`). And if you desire to associate some specific semantics your only chance may be to write some script in, say, Visual Basic, to access the API provided by the tool! None of these actions recurse to the UML meta-model.

If deeper modifications are required in the behavior of the tool, it is nearly a new implementation which has to be elaborated. As an example the *capsule* in UML RT are just presented as class stereotypes. However they do not seem to be implemented as simply in Rational Rose Real Time [20, 24] partially based on the ROOM method [23] which is an extensive re-implementation of Rational Rose, where *capsules* are first class citizen.

1.3 Current situation and evolution of the UML Meta-Model

The current UML-MM is too big to be really tractable and understandable. Many errors and inconsistencies have been discovered. Moreover the lack of integration of the Object Constraint Language into UML does not help in expressing precisely the semantics of the meta-model.

These flaws have been recognized and solutions are called upon in the RFP for UML 2.0 [18]. In particular the precise UML group (pUML [1]) has made an important contribution by refactoring the UML-MM and defining a Meta-Modeling Language (MML) to describe it [4]. The model can now be smaller, it relies on strong semantics, and it makes it possible to describe behavior through pseudo-code.

The inventors of MML recognize the importance of tool support. They provide an implementation of MML (MMT) and encourage other people to implement tools using MML as a kernel. The automation provided by these tools would then benefit from the precise semantics of MML as well as from a guarantee of model validity.

This is precisely why we use MML to describe our behavioral extensions to the UML meta-model (see section 2.2).

1.4 Outline of the paper

This paper proposes a Meta-Object Protocol for a subset of the meta-model of UML. This MOP relies on MML. It makes it possible to introduce behavior in UML-MM. A first tentative implementation and a few simple examples of application are given.

The paper is organized as follows. The next section presents in more details the motivation for a MOP for UML, describes the MOP itself, and our first implementation in Java. Section 3 gives three examples of the application of the MOP, dealing with various kinds of extension. In section 4 we evaluate the contribution of such a MOP and we discuss the work remaining to be done to reach an usable tool.

2 A MOP for UML

2.1 Motivation (lack of behavior)

Our main motivation for introducing a MOP in UML is related to tools [9]. As mentioned by Gregor Kiczales [13], tools are usually "black boxes". Each time a user tries to write an application, to make an extension, or to design a plugin, she faces with communication problems with the tool.

The case of plugin or extension is typical. Usually tools provide an input interface allowing to install plugins. However they do seldom provide an output interface. Thus, one can add an element from outside the tool but no warning is given when a new element is added within the tool. Even if the tool may provide some notification, this may not be enough. For instance, if one wants to modify some internal behavior of the tool or simply to track its inner state, one is often obliged to recreate ad hoc structures outside the tool, duplicating the more or less equivalent existing ones.

In the previously mentioned paper, Kiczales advocated that an Open Implementation is a nice and elegant solution to such problems. In the case of UML, since the meta-model has to be extensible, an easy and semantically sound way of adding behavior is needed. A standard way to provide such a feature would be a Meta Object Protocol.

It is difficult to base such a Meta Object Protocol directly onto the UML meta-model as it is currently defined. The UML-MM is big and too intricate. The behavior of the modeling entities is expressed both informally (simple text describing the semantics of operations) and more formally (multiplicity of meta-associations, OCL constraints).

Indeed there exists some predefined semi-automatic generation procedures and mappings for structures and methods, like the so-called MOF/IDL mapping [17]. However, this mapping to an implementation interface is not detailed enough (it consists mainly of accessors) and the constraints of the meta-model are not isolated but they are buried into the code of the accessors, making it difficult to change or refine them.

Before constructing a tool, a full analysis of these constraints is required. This is a very difficult task with the UML-MM; it is much easier with MML.

2.2 A tentative implementation of a MOP for UML

In order to prove the necessity of a MOP for UML, we prototyped an open-implementation of UML-MM (in Java). Since the OMG's one is too monolithic, we decided to base our implementation onto the MML [4, 10]. MML presents the same facilities as Meta-UML but it has been refactored and packaged in smaller sets. It is thus easier to manipulate: it clearly separates the concepts from the syntax, and the model elements from their instances. The semantic relationship between model and instance is defined in a semantic

package importing all necessary packages. The notion of a package is slightly different from the one known in programming languages (such as Java) since, once imported, an element can be enhanced (new fields, methods...). Here packages are used as a unit of layering and to separate concerns [8, 4], and so offer more flexibility to define elements step by step. Figures 1 and 2 show the concepts of model and instance for classes and attributes, and figure 3 shows the semantics binding them (the `semantics` package imports the concept and instance package, and add associations between elements of the two packages). To simplify the diagrams, we did not show class `Package`, a simple derivative of class `Classifier`.

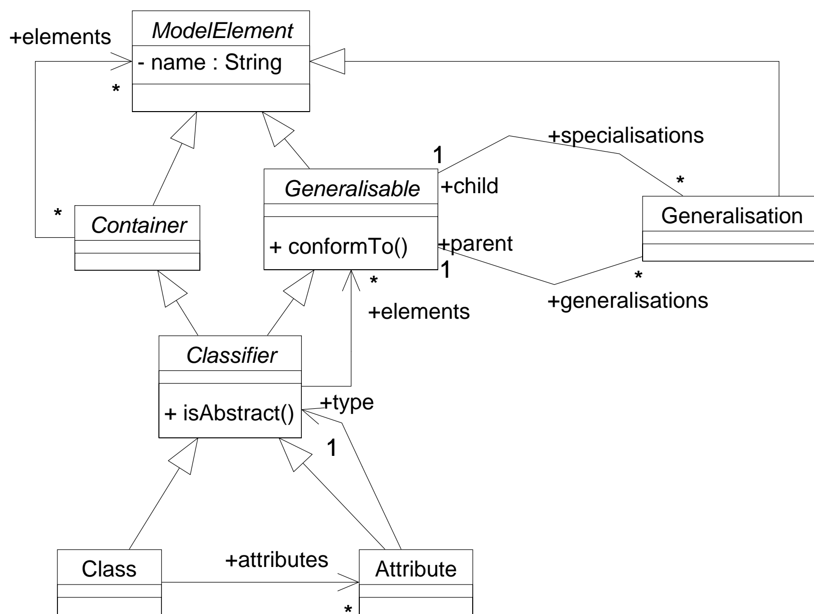


Fig. 1. MML package defining the core model elements.

We now shortly present our simple Open Implementation of the classes (figure 4). After flattening the hierarchy, the first step in this implementation, consists in creating a Java class for each UML-MM meta-class (meta-classes from figure 3 do not have an associated Java class, since they are imported and extended in package `semantics`). For every attribute a set of accessors is created allowing to add, remove, or access attributes. From our personal experience in using Meta-UML implementations [2, 16] to build extensions, we gathered that we often lacked a fine grained protocol. Conse-

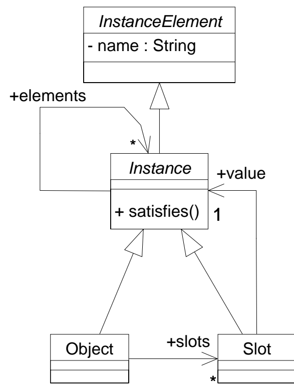


Fig. 2. MML package defining the core instance elements.

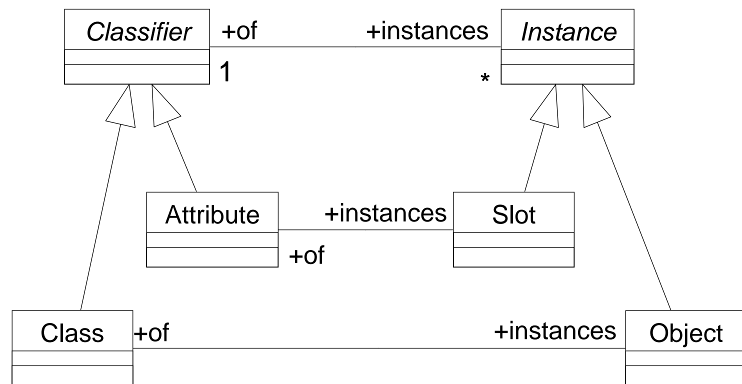


Fig. 3. MML package defining the semantics binding the core elements of model and instance packages.

quently, to achieve this fine granularity we designed three kinds of operations for every attribute:

– basic operations: setting values, adding elements in collections, etc.
– constraint operations: implementing a constraint in one or several methods
– regular operations: mixing constraint operations and basic operations to provide operations enforcing the model semantics.

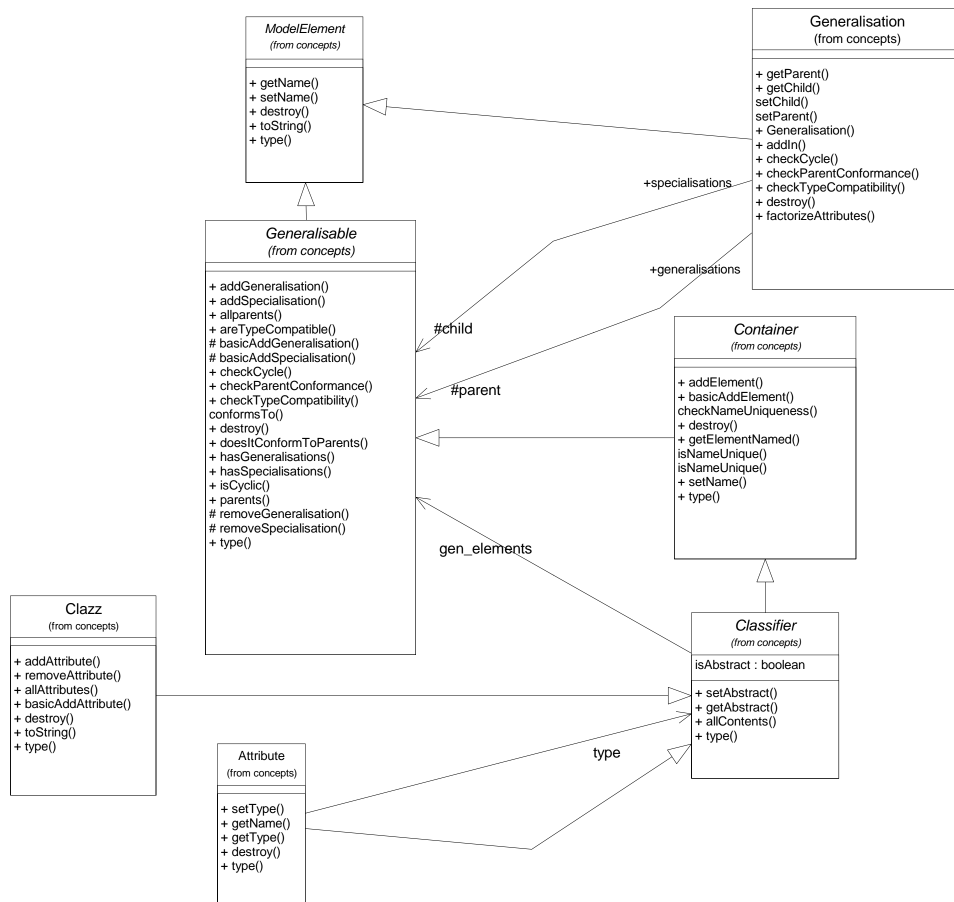


Fig. 4. The MOP.

It should be noticed that we did not try to implement the full MOF mapping even though it could have appeared reasonable (after all, the UML-MM is a MOF instance). However the MOP mapping generates many classes (in fact many interfaces) for each

meta-class and its accessors are too general for our purpose. As an example of MOF mapping, one may refer to the implementation of NsUML [16] [1].

One of our goals is to make constraints operational and to keep them isolated so that they may be adapted easily: they constitute the main way to change the UML-MM semantics. Thus the implementation of a constraint is split into several methods. The first one is just a regular predicate (for example: `areTypeCompatible`). This method does not throw any exception and thus can be used anywhere. Now when we want that the violation of this constraint raises an exception, we use a second method, `checkTypeCompatibility` that raises `TypeIncompatibleException` if the predicate `areTypeCompatible` is false. Note that we define exactly one exception for each constraint.

The creation of model elements is done through factory methods, thus adding an indirection for each element creation. These methods are located in the meta-class embedding the elements being created. For example, meta-class `Clazz` contains a method to create instances of attribute (`addAttribute`).

2.3 Why do you call this a MOP?

At a first glance, this implementation just looks like a framework with an API manipulating classes, operations, attributes, etc. Depending of your background you may call this a MOP or not. What makes this implementation a true MOP is the presence of a reflection package in MML (see figure 5): all model elements are objects, and thus it is possible to describe meta-level architectures in MML. "MML model can be viewed as an instance of the MML meta-model, or as an instance of itself" [4]. From this, it follows that the protocol offered here to manipulate the elements is a meta-protocol.

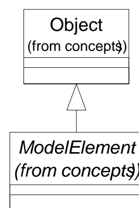
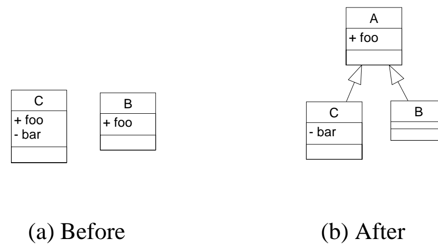


Fig. 5. Reflection in MML.

In order to show the applicability of our protocol, we describe three applications using it.

[1] It is not an exact MOF mapping, though; the resulting protocol suffers from the same problems.



(a) Before (b) After

Fig. 6. Factorizing attributes.

3 Application

This section shows through progressive examples the use and interest of the meta-protocol. The first application will show how to simply extend the behavior, in adding an attribute factorizing feature. Then, the definition of a Java profile will illustrate how to specialize meta-elements and change their behavior by changing constraints. The third application will add version and author information to modeling elements.

3.1 Factorizing attributes

Factorizing attributes is the fundamental process of gathering into a superclass those elements that are shared by sub-classes. Even if this operation is not described in the UML meta-model –and it is on purpose–, it is probably a useful operation that most tool users would like to have. Here we automate this feature and figure 6 shows its effect.

This operation could be implemented without extensive meta-model extensions, using an observer watching for class and attribute addition. However, observers usually have a warning role only, and are not supposed to change the model. Of course bypassing the latter rule is possible, but this implementation would still require to implement an observer pattern within the UML-MM and, sometimes, to keep an external and redundant representation of the model.

Our implementation just modifies the behavior of the method to add an attribute to a class and that of the `Generalization` meta-class which models the inheritance relationship. When an attribute is added, it is moved into the right place in the hierarchy. Since one cannot know before the addition of the inheritance relationship that two classes will be sisters, the `Generalization` constructor has been extended to call the `factorizeAttributes` method, which then call the `factorizeAttributes` method on the new superclass. The factorizing mechanism is defined in class `Clazz`.

This example shows how adding behavior to the UML-MM as well as defining an Open Implementation with a fine grained protocol make tool extensions easy and avoid redundant information, painful to manage.

3.2 Sketch of a Java profile in UML

The Java profile proposed here is far from being complete. We only consider Java classes, Java interfaces and their inheritance model, and we just address two issues: forbidding multiple inheritance between Java classes and forbidding a Java interface to inherit from a Java class. For the latter issue we used the notion of type compatibility defined by an MML constraint.

Three UML modeling elements are sub-classed: `Clazz`, `Generalization`, `Package`. The sub-classing results in four classes: `JavaClass`, `JavaInterface`, `JavaGeneralization` and `JavaPackage` (figure 7)

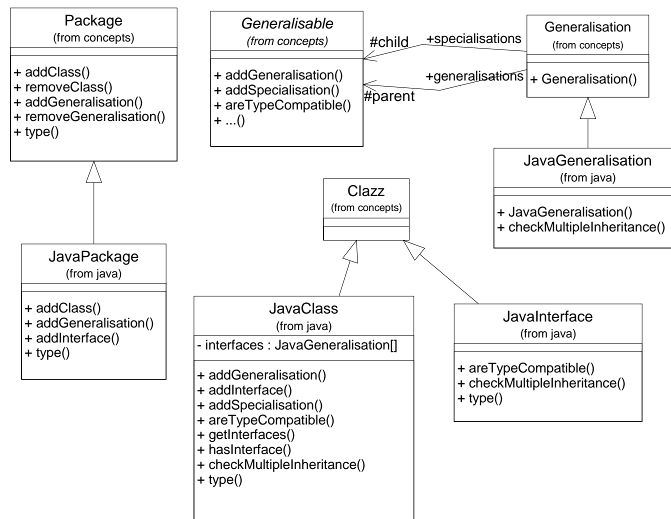


Fig. 7. A Java profile.

Class `JavaPackage` overrides the `addClass` method in order to create an instance of `JavaClass` instead of `Clazz`. It also defines a method for adding interfaces, and overrides `addGeneralization`. The `JavaGeneralization` meta-class is added to reflect the difference with `Generalization`: `JavaGeneralisation` needs to check that a class does not inherit from two classes, whereas the default `Generalization` allows multiple inheritance.

To prevent a Java interface to inherit from a Java class, we define two meta-classes, `JavaClass` and `JavaInterface` and we override the `isTypeCompatible` method defined in `Clazz`.

These three meta-classes can be used only if some (embedding) element is able to build them: this is the role of `JavaPackage`.

Note that in figure 7, we define an attribute of `JavaClass` to represent the interfaces implemented by the current class and we override the `addGeneralisation` method to fill this attribute.

This sort of extension, where one sub-classes existing meta-classes, is generally considered as *light weight* and represented as *stereotypes*. However, the use of stereotypes in this context may cause implementation problems that will be discussed in section 4.1.

3.3 Versions and authors

In this last example, we will add a notion of version and author to some modeling elements. The version is a simple attribute (a character string) whereas the author is represented by a new meta-class containing information related to the user's access rights.

We wish the author information to be added to all classes; thus, it is sufficient to add it to meta-class `Clazz` (we suppose that a person writing a class is also the one writing the methods, so we do not duplicate this information), and also to meta-class `Package`. By contrast, we wish the version information to be added only to selected meta-classes. This implies that this information cannot be added into meta-class `ModelElement`, the root of all meta-classes. The behavior of every method adding or editing an element needs to be refined to keep track of new versions.

Our current implementation is done in Java which is not a truly reflective language. Thus we had to edit the code of meta-classes in order to add the feature. A truly reflective language would have made simpler to implement the model as an instance of itself. Consequently, we could have done the addition of version and author information by simply calling the `addAttribute` method on the meta-classes to change.

The MOP we defined allows to extend the basic UML-MM in a simple way thanks to the operationalization of constraints that can be reused or customized, and to the fine grained protocol. However we encountered problems during the implementation, and we shall discuss them in the next section.

4 Discussion

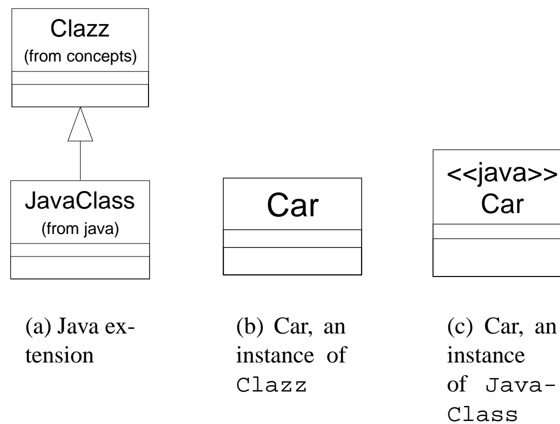
The MOP described here permits to access the operational semantics of the system, and especially to change it by modifying or overloading constraints, creators... However, the goal was not to provide a fully functional case tool but to explore the problems and difficulties caused by the implementation of a meta-model offering extension capabilities. It was also a feasibility study about providing a tool open to plugins.

We do not claim to propose *the* ultimate protocol for building UML tools. This work just constitutes a step in an experimentation process to build more adaptative UML modeling environments.

Indeed the protocol described here went through several phases of refinement and refactoring while we were building the extensions described section 3.

4.1 MM implementation and meta programming: common problems

As in all software activity, the most difficult part has been to define the protocol manipulating the elements, since in the given specification (a meta-model partly represented



(a) Java extension

(b) Car, an instance of `Clazz`

(c) Car, an instance of `JavaClass`

Fig. 8. Applying stereotypes.

in figures 1, 2, 3) very few clues were given to implement the system. There were only a few constraints and methods describing some operations that could not have been defined in other ways (for example, the method `conformsTo`). This problem of tweaking a MOP is known as been difficult and “developing a protocol that is simultaneously powerful, easy to use, and efficient involves a number of issues” [14] that are more complicated that they seem to be.

Implementing stereotypes The stereotype mechanism consists in extending UML in a very light way (sub-classing existing meta-elements). In the UML diagrams, it is depicted by character strings between <<French guillemets>> that decorate the stereotyped element (see figure 8). This textual annotation can be added and removed at wish, then changing the semantics of the element. When a stereotype is applied, the user expects to take advantage of the behavior defined in the new meta-class the stereotype corresponds to. However, those semantic changes are currently not taken into account by tools, and they are problematic to implement. The figure 8 shows the class `Clazz` and its Java specialization. When an ordinary class (an instance of `Clazz`) is stereotyped to become a Java class, one should be able to add an interface to it and to prevent it from multiple inheritance. To do so, the ordinary class has to change its (meta-)type to become an instance of `JavaClass`. Changing an object type can either be implemented with programming techniques, with meta-programming techniques, or using some languages specific features.

Using a non reflective programming language, a simple way is to delete the element and recreate it as an instance of the new class. This technique is slow and complicates the code. It also causes problems for embedded elements: some might require to be rebuilt, some might not. For example, applying the stereotype Java to a package requires all the classes to become Java classes or Java interfaces.

An other technique to implement stereotypes is to use the decorator design pattern. The decorator pattern "attaches additional responsibilities to an object dynamically"

and "provides a flexible alternative to sub-classing for extending functionality" [11]. However, it requires a special design of the hierarchy where decorators need to sub-class the decorated element. The decorator pattern is troublesome when the decoration needs to change dynamically since a reference to a decorated object is not identical to a reference to the original object.

An other possibility is to consider the stereotype as a meta-object changing the behavior of the base object. This can be seen as similar to the decorator pattern, but it is much more versatile. For example the class `JavaClass` could be considered as a meta-object dynamically adapting the behavior of the base class `Clazz`. This solution works fine if the meta-object only contains behavior refinement of the base element. However, it does not allow to define new behavior (like the `addInterface` method in `JavaClass`) because it cannot dynamically add new operations to the base element.

On the one hand this meta-object solution, and in the other the logical need to support multiple stereotypes for one element (for example a class could be both a Java class and a Real Time class (a *capsule*)) relate to the problem of meta-classes composition [15].

In a programming language, the primary feature required to implement stereotypes is the ability to change the class of an instance. This feature exists in Smalltalk under two different possibilities: `changeClassToThatOf` and `become`. They respectively allow to change the class of an instance and to switch two instance references. Each of those facilities are both problematic. `changeClassToThatOf` has restriction regarding the number of fields contained in the original and the destination classes: they must to be equal and the order of fields needs to be the same. `become` requires the new object to be recreated before being interchanged with the first one. Those problems, identified by Fred Rivard, are solved in NeoClasstalk [22].

It could also be noticed that the wide classes [25] should be of a good use since they allow to change the class of an instance to any class in its inheritance hierarchy.

4.2 What remains to be done? Is it worth the effort?

Towards an usable tool This first implementation in a language offering no support for intercession rose the problem that implementing the UML meta-model, or even a subset of it, is not an easy operation. To build a real tool, the first recommendation could be to base the implementation on a language facilitating meta-programming. For example, Smalltalk is a good candidate [21], since it allows dynamic modification of class (see section 4.1). Indeed using another language is possible (just look at existing tools!), but high capabilities of reflection facilitate the implementation.

Once the language has been chosen, comes the problem of the protocol definition. Even for the few classes implemented here, the definition of the protocol and especially the organization of the code for managing constraints or for adding elements was difficult. So, according to the big number of classes and constraints contained in the original UML meta-model, a fully consistent Open Implementation appears as an overwhelming task. This is the reason why we chose the MML the small size of which makes more likely to go through the implementation. In these conditions we wonder if the effort of defining a minimal protocol to manipulate UML-MM, should not be done by an organization like the OMG when defining the meta-model. This would make sense since

many OMG members working on the UML specification are supposed to deliver tools implementing these specifications.

Is it worth the effort? The effort is worth it if striving for building more open tools is worth it. For instance, every implementation of Smalltalk offers a set of basic features and extended ones. With UML tools, it could be the same.

It also worth the effort if the UML, as it is defined now, wants to become a programming language. If the description of the UML as a programming language does not change from what it is now, we can see it has similarities with Smalltalk, in that it is bootstrapped, it is extensible, it offers capabilities to change its core, and it needs to be able to change the class of an instance (applying a stereotype). The definition of an operational semantics (in complement of the denotational one provided by pUML) is required for the precise definition of the operations of the MOP.

A tool fully supporting UML will keep on increasing the success of UML. Thanks (partly) to UML, the notion of meta-modeling is now becoming familiar to more programmers. Having tools fully supporting it will certainly improve global understanding of average users and may prompt them to consider meta-programming techniques.

5 Conclusion

In this paper we have demonstrated how defining a meta-object protocol to manipulate the UML meta-model could be a valuable help for building open tools, ready to accommodate plugins, extensions, and profiles.

We have also shown that the implementation of a MOP in a modeling context relies on meta-programming. The meta-programming problems are well-known and apply also to meta-modeling (*e.g.*, MOP composition).

There is no ideal and Almighty MOP. At best one must strive to facilitate the most frequently encountered manipulations. This is even more difficult as the meta-model claims to be open or extensible, as it is precisely the case for UML.

However, hand-crafted implementations are bad, since a major change in the meta-model may induce extensive recrafting of the code. And if meta-models are defined, there are to be implemented. And to be used. And tools should rely on them. And extensions should be defined within them and implemented through them.

Although further work is obviously needed to obtain a complete environment, the simple experiment which has been exposed in this paper is a step toward the operationalization of meta-modeling as a basis for an automatic and semantically sound extension facility and tool specification.

References

[1] The precise UML group. http://www.cs.york.ac.uk/puml/.

[2] ArgoUML project. http://argouml.tigris.org/, 2001.

[3] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling language User Guide*. Addison-Wesley, 1999.

[4] Tony Clark, Andy Evans, Stuart Kent, Steve Brodsky, and Steve Cook. A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach. Technical report, pUML, 2000.

[5] Action Semanctics Consortium. Action Semantics for the UML. http://www.umlactionsemantics.org/.

[6] Steve Cook. The UML Family: Profiles, Prefaces and Packages. In *UML 2000*, pages 255–264. Springer-Verlag, 2000.

[7] Bruce Powell Douglas. *Doing Hard Time: Developping Real-Time Systems with UML, Objects, Frameworks and Patterns*. Addison-Wesley, 1999.

[8] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, components and frameworks with UML*. Addison-Wesley, 1998.

[9] Nathan Dykman, Martin Griss, and Robert Kessler. Nine Suggestions for Improving UML Extensibility. In *UML '99*, pages 236–248. Springer-Verlag, 1999.

[10] Andy Evans and Stuart Kent. Core Meta-Modelling Semantics of UML: the pUML Approach. In *UML '99*, pages 140–155. Springer-Verlag, 1999.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[12] I-Logix. Rhapsody. http://www.ilogix.com.

[13] Gregor Kiczales. Beyond the black box: open implementation. *IEEE Software*, 13(1):8–11, January 1996.

[14] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[15] Philippe Mulet, Jacques Malenfant, and Pierre Cointe. Towards a methodology for explicit composition of metaobjects. In *OOPSLA '95*, pages 316–330, 1995.

[16] NovoSoft. NsUML, 1999. http://sourceforge.net/projects/nsuml.

[17] OMG. OMG Meta Object Facility Specification, version 1.3. Technical report, OMG, April 2000. ftp://ftp.omg.org/pub/docs/formal/00-04-03.pdf.

[18] OMG. OMG UML 2.0 Infrastructure RFP. Technical report, OMG, 2000. http://cgi.omg.org/cgi-bin/doc?ad/2000-09-01.

[19] OMG. OMG Unified Modeling Language Specification, Version 1.3. Technical report, OMG, March 2000.

[20] Rational. Rational Rose Real Time. http://www.rational.com/products/rosert/, 2001.

[21] Fred Rivard. Smalltalk: a Reflective Language. In *Reflection 96*, 1996.

[22] Fred Rivard. *Object Behavioral Evolution Within Class Based Reflective Languages*. Thèse de doctorat, Université de Nantes, Ecole des mines de Nantes, 1997.

[23] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time Object-Oriented modeling*. Wiley and Sons, 1994.

[24] Bran Selic and James Rumbaugh. Using UML for Modeling Complex Real Time Systems. http://www.rational.com.

[25] Manuel Serrano. Wide Classes. In *ECOOP '99*, pages 391–415. Springer-Verlag, 1999.