Preface

...

September 2001

Ákos Frohner
Workshop Chair
ECOOP 2001

Organization

...

Table of Contents

Adaptive Object-Models and Metamodeling Techniques

Nicolas Revault[1] and Joseph W. Yoder[2]

[1] University of Cergy-Pontoise (& Paris 6),
33 bd du Port 95011 Cergy-Pontoise Cedex, FRANCE
Nicolas.Revault@lip6.fr
[2] The Refactory, Inc.,
209 W. Iowa Urbana, IL 61801, USA
yoder@refactory.com

Abstract. This article reports on the presentations and discussions of the workshop on "Adaptive Object-Models and Metamodeling Techniques", held in conjonction with Ecoop'01 in Budapest on June 2001. After overviewing the themes of the workshop, its organization is briefly presented. This is followed by a summary of the works presented and a section dedicated to develop the results of the workshop. The main conclusions are about comparing and locating one towards another three techniques of interest: – Reflection at the Language Level, – Adaptive Object-Models and – Meta-Tool Approaches to meta-modeling. Moreover, a discussion on the needed levels of abstraction, and on their nature, is also developed in reference to the so-called "OMG four-layer architecture".

1 Overview

A system with an Adaptive Object-Model (AOM) has an explicit object model that it interprets at runtime. If you change the object model, the system changes its behavior. For example, a lot of workflow systems have an Adaptive Object-Model. Objects have states and respond to events by changing state. The Adaptive Object-Model defines the objects, their states, the events, and the conditions under which an object changes state.

There are various techniques that share common features with AOM's. Especially, those that capture business rules and build domain -or business- specific languages, namely – Grammar-Oriented Object Design (applied in the three major areas of configurable workflow, tier-to-tier mapping and object graph traversal) or – Meta-Tool Approaches[1], à la MetaEdit or à la MétaGen (applied in various fields of information system modeling: telecom, finance, medicine, etc.). There are other techniques which also describe ways to build systems that change behavior at runtime, namely – Reflection at the Language Level (mostly applied to programming language design). What is actually common to those various

[1] sometimes also refered to as "Meta-CASE Tool Approaches".

techniques is that they are leading to, or are driven by, meta-modeling principles and implementation using OO languages.

Adaptive Object-Models and other techniques such as Grammar-Oriented Object Design, Meta-Tool Approaches or Reflection at the Language Level, address at least one of the two following problems:

- Capturing (business) rules for user modeling and/or building (Domain/ Business) Specific Languages;
- Building systems that need to change requirements and reflect those requirements as quickly as possible, i.e. runtime or dynamic adaptability.

The workshop focused on identifying, cataloging and comparing these techniques one towards another. We looked at establishing some conclusions on the conditions of use of these techniques, on where they meet or overlap, and hopefully on some cross-fertilization ideas of benefit for each technique. What is generally common to these techniques is that they actually implement or use meta-modeling principles through OO languages capabilities.

Workshop position papers were presented that addressed one or more of the following:

- Examples of the different techniques;
- Concrete development reports with lessons learned;
- How can these techniques support one another;
- Prerequisites or needs for each technique;
- Pros and Cons of the different techniques;
- Comparison of the different techniques showing similarities and differences.

This paper is actually highlighting the submitted papers along with the result of our findings that address the above mentioned points.

2 Workshop organization

The workshop was first organized around the history of the workshop, the building of a common vocabulary and context for the workshop, along with the objectives of the workshop. We then looked at some details for two of the subfields of interest, followed by presentations from works in the subfields from submitted papers[2]. We then broke into some groups for discussions and synthesized our results to be presented at the end of this paper.

2.1 History of the workshop

From 1998 to 2000, AOM's have been discussed in various workshops [24, 23, 22, 25]. Metadata and Adaptive Object-Model workshops at the University of Illinois in 1998 and at OOPSLA '98 and '99 were devoted to "pattern mining" some of

[2] Available position papers from workshop participants [9, 12, 13, 21] can be found at `www.adaptiveobjectmodel.com/ECOOP2001`.

these ideas. The workshop at ECOOP '00 was oriented toward a broader analysis of the possible uses of the Adaptive Object-Model technology, based on the examination of a range of examples. The discussions their led to establish some "dimensions of abstraction in Adaptive Object-Models, Reflection and OMG's meta-modeling architecture". Further along after these discussions, we came out with a wider idea of comparison of the techniques of interest: meta-modeling appearing to be the core feature of each technique, it has naturally been tackled as the next stepping problem.

2.2 Context for the workshop

In order to fix a common vocabulary and settle the basis of the work for the workshop day, we recalled – 1. what we mean while using the "meta" prefix and what main idea is behind it for us, and – 2. the various techniques that deal with meta-modeling, defined in the workshop as the "subfields of interest". We also recalled the common problems we think these techniques intend to all address.

The main idea we have in mind while using the "meta" prefix is related to class-based object-oriented programming languages (OOPL). In these, we are used to differentiate between the "instance level" and the "class level". Briefly, the instance level is the one of runtime, where objects virtually exist in the computer memory, as instances of their class. The class level is the one of programming time, where classes are defined as "shapes" or "molds" for their future instances with references to other classes and actually planning (programming) the way objects will be created and will evolve at runtime. The operational link thus defined between instances and their class, is the one we call the "meta" link. This particularly makes sense when considering classes themselves as objects, instances of other classes (usual in some OOPL, e.g. Smalltalk): classes of the objects being classes are always called metaclasses.

The techniques dealing with meta-modeling we are interested in (as "subfields of interest") are those presented in the overview. Namely, Reflection at the Language Level, Grammar-Oriented Object Design, Adaptive Object-Models and Meta-Tool Approaches (à la MetaEdit or à la MétaGen), each one applied in some privileged areas. The main problems these techniques address are – Capturing (business) rules for user modeling and/or building (Domain/Business) Specific Languages and – Building systems that need to change requirements and reflect those requirements as quickly as possible, i.e. runtime or dynamic adaptability.

2.3 More on the objectives of the workshop

In addition to the general objectives presented in the overview section (identifying, cataloging and comparing techniques), some more specific objectives were given as part of the introduction to the workshop. These were mainly related to AOM's along with Meta-Tool Approaches.

A first idea was on the way of "going bottom up from" AOM's, in the sense that they allow to define operational domain (or business) specific languages.

One of the specific objectives here is to make clear, and hopefully systematic, the way(s) this kind of languages can be supported by tools.

Another idea was on the way of "going top down from" Meta-Tool Approaches. Indeed, these generally offer meta-modeling and generating environments, in the sense that they allow to build operational user-customized model editors. One objective on that subject is to explicit the way(s) to get the operationality for models: how code generators can be systematically specified and what are the most suitable forms for them?

In addition, another objective concerning interaction between the two sets of techniques was declared: "how to interact somewhere in the middle ?". Issues such as how the techniques might support one another while setting differences and overlapping (if any) were actually asked.

Finally, setting the potential answer as another objective, we wondered on how to integrate and locate the other techniques of interest, which have either some similar goals or some similar means.

3 Summary of Contributions

The workshop brought together researchers and practitioners from a variety of areas in order to explore the themes that underlie the various techniques of interest.

A good sign that a technical area is ripening is that a number of people independently discover or explore it. This would seem to be the case with AOM's and meta-modeling. In the main, participants focused on comparisons between the Adaptive Object-Model's approach and those of Reflection and meta-modeling through Meta-Tools. It emerged that indeed all three approaches share the same levels of abstraction.

The following is a more detailed view of the main experiences/positions that were presented for participation at the workshop.

3.1 Reflection at the Language Level

Reflection is one of the main techniques used to develop adaptable systems and, currently, different kinds of reflective systems exist[3]. Compile-time reflection systems provide the ability to customize their language but they are not adaptable at runtime. On the other hand, runtime reflection systems define meta-object protocols to customize the system semantics at runtime. However, these meta-object protocols restrict the way a system may be adapted before its execution, and they do not permit the customization of its language.

The system presented by Ortin et al. [12] implements a non-restrictive reflection mechanism over a virtual machine, in which every feature may be adapted at runtime. No meta-object protocol is used and, therefore, it is not needed to specify previously what may be reflected. With this reflective system, the programming language may be also customized at runtime.

[3] This work was presented by F. Ortin, on the basis of his workshop submission [12]

3.2 Adaptive Object-Model Architecture

Today, users themselves often seek to dynamically change their business rules without the writing of new code[4]. Customers require that systems are built that can adapt more easily to changing business needs, that can meet their unique requirements, and can scale to large and small installations.

On the other hand, the same technique is adequate for the slightly different purpose of producing a whole line of software products: of course, a line of products may be obtained by variously instantiating a unique abstract model, but also by adapting a given initial system to various requirements that appear simultaneously instead of evolving in time. Moreover, the diversification of a successful product may also be seen as a form of reengineering.

Black-box frameworks provided early solutions for the design of flexible implementation of business rules [17]. Recent research in the different types of architectures to meet such requirements from an object-oriented perspective has been catalogued as Adaptive Object-Models [2, 3, 1, 7, 23]. An Adaptive Object-Model is where the object representation of the domain under study has itself an explicit object model (however partial) that is interpreted at runtime. Such an object model can be changed with immediate (but controlled) effect on the system interpreting and running it. Note that Adaptive Object-Models usually requires a thorough analysis of the domain at hand, which may very well include a black-box framework as an initial stage.

Objects have states and respond to events by changing state. The Adaptive Object-Model defines the object model, i.e. the objects, their states, the events, and the conditions under which an object changes state, in a way that allows for dynamic modification. If you change the object model, the system changes its behavior. For example, such a feature makes it easy to integrate a workflow mechanism, which proves useful in many systems [10, 19].

Adaptive Object-Models successfully confront the need for change by casting information like business rules as data rather than code. In this way, it is subject to change at runtime. Using objects to model such data and coupling an interpretation mechanism to that structure, we obtain a domain-specific language, which allows users themselves to change the system following the evolution of their business.

Metadata[5] is then often used in adaptive object-models to describe the object model itself. When runtime descriptions of these objects are available, users can

[4] This work was presented by J. Yoder, on the basis of his workshop submission [21]

[5] MetaData can be described by saying that if something is going to vary in a predictable way, store the description of the variation in a database so that it is easy to change. In other words, if something is going to change a lot, make it easy to change. The problem is that it can be hard to figure out what changes, and even if you know what changes then it can be hard to figure out how to describe the change in your database. Code is powerful, and it can be hard to make your data as powerful as your code without making it as complicated as your code. But when you are able to figure out how to do it right, metadata can be incredibly powerful, and can decrease your maintenance burden by an order of magnitude, or two. [R. Johnson]

directly manipulate these objects. Since the system can interpret the metadata to build and manipulate these runtime descriptions, it is easy to add new objects to the adaptive object-model, and make them immediately available to users. This approach has been validated by several successful industrial projects (see submissions to [24] and [25] along with [19, 14]).

Adaptive Object-Model architectures are usually made up of several patterns: *TypeObject* [7] is used to separate an Entity from an EntityType, Entities have Attributes, which are implemented with the *Property* pattern [3], and the *TypeObject* pattern is used a second time to separate Attributes from AttributeTypes. As is common in Entity-Relationship modeling, an Adaptive Object-Model usually separates attributes from relationships.

The *Strategy* pattern [4] is often used to define the behavior of an EntityType. These strategies can evolve to a more powerful rule-based language that gets interpreted at runtime for representing changing behavior. Finally, there needs to be support for reading and interpreting the data representing the business rules that are stored in the database and there is usually an interface for non-programmers to define the new types of objects, attributes and behaviors needed for the specified domain.

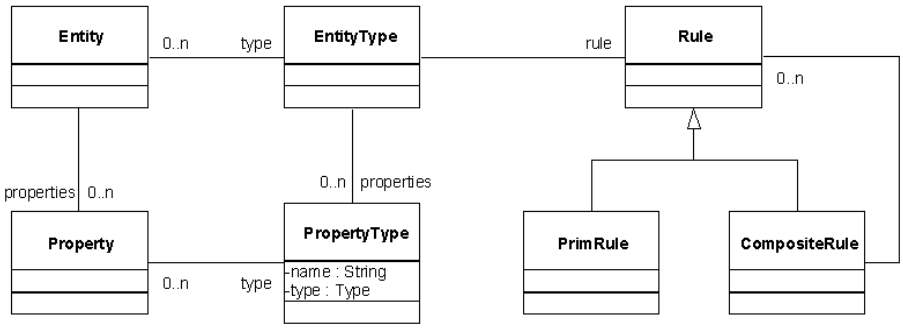

Fig. 1. Type Square

Fig. 1 is a UML diagram of applying the *TypeObject* pattern twice with the *Property* pattern and then adding *Strategies/RuleObjects* for representing the behavior. We call this resulting architecture the *TypeSquare* pattern and it is often seen in adaptable systems with knowledge levels as described in this paper.

Metadata for describing the business rules and objects model is interpreted in two places. The first is where the objects are constructed otherwise known as instantiating the object-model. The second is during the runtime interpretation of the business rules.

The information for describing the types of entities, properties, relationships, and behaviors are stored in a database for runtime manipulation, thus allowing

for the business model to be updated and immediately reflected in applications interpreting the data.

Regardless of how the data is stored, it is necessary for the data to be interpreted to build up the adaptive object-model that represents the real business model. If an object-oriented database is used, the types of objects and relationships can be built up by simply instantiating the *TypeObjects*, *Properties*, and *RuleObjects*. Otherwise, the metadata is read from the database for building these objects, which are built using the *Interpreter* and *Builder* pattern.

The second place where the *Interpreter* pattern is applied is for the actual behaviors associated with the business entities described in the system. Eventually after new types of objects are created with their respective attributes, some meaningful operations will be applied to these objects. If these are simple *Strategies*, some metadata might describe the method that needs to be invoked along with the appropriate *Strategy*. These *Strategies* can be plugged into the appropriate object during the instantiation of the types.

As more powerful business rules are needed, *Strategies* can evolve to become more complex such that they are built up and interpreted at runtime. These can be either primitive rules or the combination of business rules through application of the *Composite* pattern. If the business rules are workflow in nature, you can use the Micro-Workflow architecture as described by Manolescu [10]. Micro-Workflow describes classes that represent workflow structure as a combination of rules such as repetition, conditional, sequential, forking, and primitive rules. These rules can be built up at runtime to represent a particular workflow process.

Adaptive Object-Models are usually built from applying one or more of the above patterns in conjunction with other design patterns such as *Composite*, *Interpreter*, and *Builder* [4]. *Composite* is used for either building dynamic tree structure types or rules. For example, if your entities need to be composed in a dynamic tree like structure, the *Composite* pattern is applied. *Builders* and *Interpreters* are commonly used for building the structures from the meta-model or interpreting the results.

But, these are just patterns; they are not a framework for building Adaptive Object-Models. Every Adaptive Object-Model is similar to a framework of a sort but there is currently no generic framework for building them. A generic framework for building the *TypeObjects*, *Properties*, and their respective relationships could probably be built, but these are fairly easy to define and the hard work is generally associated with rules described by the business language. This is something that is usually very domain-specific and varies quite a bit.

3.3 Meta-modeling through Meta-Tool Approaches

For about the last 10 years, meta-modeling environments have been developed, sometimes originally for specific application areas (see e.g. [6, 18, 16, 5, 9])[6]. These environments generally share two main features: – they allow explicit meta-modeling, where meta-models and models are fully reified at instance level, and

[6] This work was presented by N. Revault, as an introduction to that subfield of interest

– they allow to derive (full or partial) application code from the models being specified in their editors.

In order to illustrate this kind of specification tools, a particular tool has been presented at the workshop: the MétaGen system [16]. This meta-tool has for specificity to address the problem of code generation by model transformation, this process being itself expressed in the form of a (first-order logic) rule-based system. And like other meta-tools, it allows model edition and modeling language (meta-model) prototyping, by supporting dynamically model/meta-model articulation.

For being concrete, a simple but illustrative example has been used to show the various components of the tool. The example is about a reduced dataflow modeling language, where simple *operators* (standard arithmetic ones) might be used in connection through *flows* to *constants* or *variables*, for building models of equations. Two models expressed in the language were given as examples of equations: one with just a simple operation and two variables as input and one as output ($a * b = c$, Fig. 2), the second with several operations, for representing the way an amount of installments can be computed from an initial capital, a rate and a number of installments (not shown here). Both of the models for the corresponding equations were in turn automatically operationalized as a Smalltalk application, with the necessary GUI built for giving arbitrary values to the relative input variables.

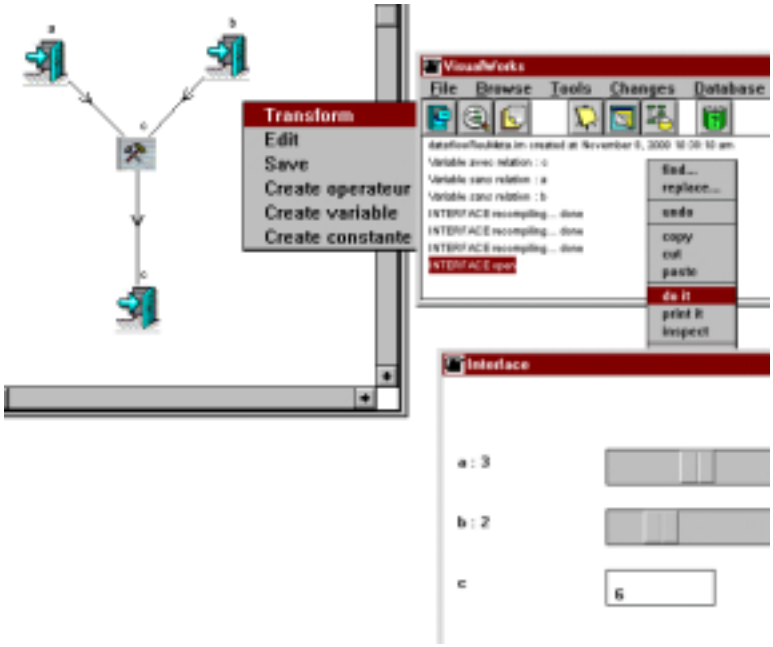

Fig. 2. Example of a model operationalization, input and output

Each of the dataflow models has been used for explaining how the transformation process operates for finally generating the application code. Actually, the initial model is used as input of a first step in the transformation: it is transformed into another model, representing the same "core" equation, but with some more information, added in order to express the way to edit the equation elements, e.g. standard sliders for input variables, or simple text fields for output variables. The new model is expressed in another modeling language, "Appli", used for specifying the application more in details. It is actually an intermediate that is in turn used as input of the second step in the generation: it is straight fully interpreted for generating the Smalltalk code for implementing the equation and allowing to edit its parameters.[7]

On the practical aspect, the transformation operates on objects reifying the models. These objects are instances of classes that are called "model classes". The model classes are in fact the implementation of a meta-model for the modeling language in use, the dataflow language or Appli in our example. The meta-models are themselves reified as objects in the environment (Fig. 3). They are specified as instances of constructs of the meta-modeling language of the tool: Pir3 in MétaGen, which roughly allows to define classes and set associations between them [15].

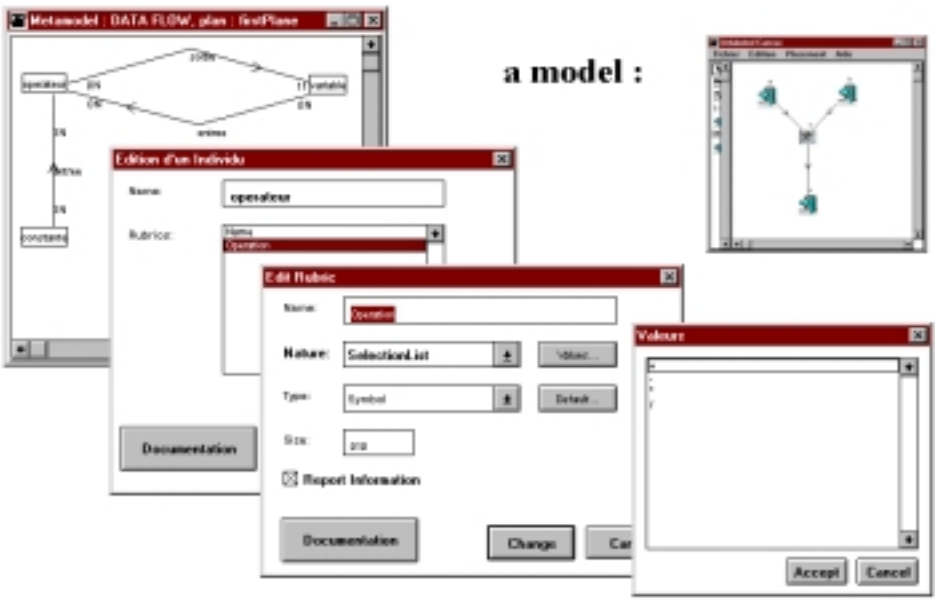

Fig. 3. Example of meta-model specification

[7] Note that there might be several possible interpretations here, e.g. for generating code towards another support language: it was shown for Java for instance.

What is particular to the MétaGen tool is that the transformation process is preferably expressed with first-order logic (forward-chaining) rules. The rules are expressed on the basis of the meta-models' elements: in our example, one of the rules states for instance that a variable of a dataflow model without any input flow relation is to be treated such that the intermediate model must include a slider connected to it, and thus that such a GUI component must be used for graphically representing it in the final application... Of course, it is the execution of the various rules that operates the automatic model transformation.

Finally, in order to illustrate adaptable model edition and actually dynamic modeling language prototyping, an evolution of the representation for variable intervals was presented: min, max and step values, initially treated within a single blank-separated string, were restructured as a record or "struct" data type. Naturally, it would be necessary to update the transformation rules for taking that evolution in account in order to make the generation process consistent.

As a summary, Fig. 4 shows a schema for the whole implementation of the example. It is actually an archetypal schema for a lot of the projects developed with the MétaGen tool.

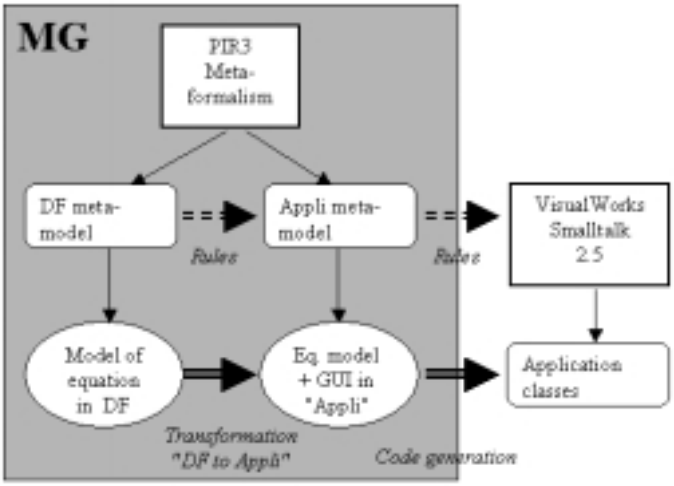

Fig. 4. Schema of the dataflow example implementation in MétaGen (MG)

3.4 Composable Meta-modeling Environment

Domain-Specific Design Environments (DSDE) capture specifications and automatically generate or configure the target applications in particular engineering fields[8]. Well known examples include Matlab/Simulink for signal processing and

[8] This work was presented by A. Ledeczi, on the basis of his workshop submission [9]

LabView for instrumentation, among others. The success of DSDEs in a wide variety of applications in diverse fields is well documented. Unfortunately, the development of a typical DSDE is very expensive. To solve this problem, Ledeczi et al. advocate the idea of a Configurable Domain-Specific Development Environment (CDSDE) that is configurable to a wide range of domains. The Generic Modeling Environment (GME 2000) is a configurable toolkit for creating domain-specific design environments. The configuration is accomplished through metamodels specifying the modeling paradigm (modeling language) of the application domain. GME 2000 follows the standard four-layer metamodeling architecture applied in the specification of CDIF and UML. It is thus representing another particular Meta-Tool Approach.

The metamodeling language in GME 2000 is based on the UML class diagram notation including OCL constraints. Just as the reusability of domain models from application to application is essential, the reusability of metamodels from domain to domain is also important. Ideally, a library of metamodels of important sub-domains should be made available to the metamodeler, who can extend and compose them together to specify domain languages. These sub-domains might include different variations of signal-flow, finite state machines, data type specifications, fault propagation graphs, petri-nets, etc. The extension and composition mechanisms must not modify the original metamodels, just as subclasses do not modify base classes in OO programming. Then changes in the metamodel libraries, reflecting a better understanding of the given domain, for example, can propagate automatically to the metamodels that utilize them. Furthermore, by precisely specifying the extension and composition rules, models specified in the original domain language can be automatically translated to comply with the new, extended and composed, modeling language.

To support metamodel composition, some new UML operators are necessary. The equivalence operator is used to represent the union of two UML class objects. The two classes cease to be separate entities, but form a single class instead. Thus, the union includes all attributes, compositions and associations of each individual class. Equivalence can be thought of as defining the "join points" or "composition points" of two or more source metamodels.

New operators were also introduced to provide finer control over inheritance. When the new class needs to be able to play the role of the base class, but its internals need not be inherited, interface inheritance is used. In this case, all associations and those compositions where the base class plays the role of the contained object are inherited. On the other hand, when only the internals of a class are needed by a subclass, implementation inheritance is used. In this case, all the attributes and those compositions where the base class plays the role of the container are inherited. Notice that the union of these two new inheritance operators is the "regular" UML inheritance.

It is important to observe that the use the equivalence and new inheritance operators are just a notational convenience, and in no way change the underlying semantics of UML. In fact, every diagram using the new operators has an equivalent "pure" UML representation, and as such, each composed metamodel could

be represented without the new operators. However, such metamodels would either need to modify the original metamodels or require the manual repetition of information in them due to the lack of fine control over inheritance. These metamodels would also be significantly more cluttered, making the diagrams more difficult to read and understand.

4 Workshop results

Following the workshop, we came to several conclusions on various themes. Hereafter, we present the main ones:

- on comparing techniques of Reflection at the Language Level and AOM techniques, the former being seen as a more transversal case of the latter (or the later as a more application specific case);
- on paralleling the relationship between AOM's definition/utilization and meta-model/model of Meta-Tool Approaches to the relationship between interpreted and compiled expressions in programming languages; and finally
- on discussing about abstraction levels, in reference to the standard OMG 4-layers architecture.

Considering the ins and outs of activities using Reflection at the Language Level (RaLL) on the one hand (e.g. [12] and also its references), and those of AOM based developments on the other hand (see references of [21]), some commonalities appear between the two sets of works. In particular, in both kinds of works a main concern is put on the property of (dynamic) adaptability, where a developed system is supposed to automatically adjust (at runtime) to new specifications of its users. Moreover, for achieving the above property, both sets of works lead to define and develop some kind of an interpreter for runtime adaptation: the one specified by a Meta-Object Protocol (MOP) in the first case [8], and a more application specific one in the second case (e.g. for the developed health system for the Illinois Department of Public Health [20]).

For these reasons, and because of the differences we explain just below, we might see RaLL techniques as some more general or transversal cases of AOM techniques, or *a contrario*, AOM techniques as more specific cases of RaLL techniques.

The main differences between the techniques are concerning their scope wideness, their application domains, and also their applicability. Indeed, RaLL techniques are definitely more powerful in scope that AOM techniques, in the sense they are mainly developed for general programming languages design, whereas AOM's are more restricted for them being developed for particular application domains. Another difference is about complexity and applicability: RaLL techniques are actually complex and not easy to manage and apply in business oriented developments, precisely for their wideness of scope, whereas AOM's, as offset by their restriction to particular application domains, reduce complexity and increase applicability in business.

For observing both the techniques for AOM based developments and for projects where Meta-Tool Approaches have been used, we came also to parallel their relationship to that which exists between interpreted expressions in programming languages and compiled expressions.

Indeed, both techniques essentially share a common objective: defining a business-specific language for "operational business modeling" (see details for each technique in the relative sections of this document). On the other hand, the means used by each technique for that purpose are quite different: – in the case of AOM's, the language is in some sense "pulled up" from the implementation of an interpreter and in that way driven by the (operational) semantics; – in the case of Meta-Tool Approaches, the language is specified by a meta-model and in the some sense "pushed down" from it whereas its semantics is fixed by a third-party artifact, a model transformer and/or code generator, which can be viewed as an actual model compiler.

Finally, we can find the same kind of differences between AOM's (a) and Meta-Tool Approaches (b) than between interpreted expressions (a') and compiled expressions (b') in programming languages:

- the specifications (models or expressions) and their usage are more interactive in case of (a) and (a'), whereas there is a more static usage in case of (b) and (b');
- some direct executions might be obtained for (a) and (a') whereas they are delayed by definition for (b) and (b').

Of course, this parallel is to be treated with caution because of the scale of specification which is much larger in the case of the techniques we are interested in - (a) and (b) - than in the case of (a') and (b'), and also because of their modeling orientation and business concern.

As we also had some points on the various levels of abstraction needed for each technique, here is a summary of what appeared to some of us. First, we made the assessment that on a -precise- operational point of view (while using class-instance based OOPL), there are only two levels of abstractions: roughly, the two levels cover the class level and the instance level, as introduced in the context section of this document (while talking about the "meta" prefix). Programming is what occurs at the first level, which is the definition level, where classes are specified, and runtime is about the second level, the execution level, where instances are actually "activated" (at least as the current context for execution). Then, especially while dealing with meta-programming or meta-modeling, we need to settle a conceptual framework where more levels are available in order to isolate things well.

That's is why the OMG's 4-layers architecture is certainly justified and why we might like to conform to it [11]. However, we must keep in mind that when implementing such a conceptual framework or part of it, the various conceptual levels of abstractions are finally all projected onto the two operational levels of classes and instances.

Finally, we think it might also be useful to have in mind that on top of the standard 4-layer architecture, whatever its implementation is, each layer is

actually aggregating a dual representation: indeed, on the one hand a model (or expression) at one level is being composed of instances of the classes of the level above, and on the other hand, it is also defining classes for the level below. For example, a meta-model (resp. a model) is built of instances of a meta-modeling language (resp. a modeling language), whereas it describes itself classes of a modeling language (resp. an object model or class diagram).

There are an infinite number of modeling languages. A metamodeling language is just another modeling language. A metamodel specifies one specific modeling language. If that specific modeling language is the metamodeling language, then its metamodel is called the meta-metamodel. In that sense, it is often the case that the fourth layer is not needed; the third layer (the metamodeling layer) is the top layer and it can describe itself.

5 Conclusions

A dominant theme was that the need to confront change is forcing system architects to find ways to allow their systems and users to more effectively keep pace with these changes. A way to do this is to cast information like business rules as data rather than code, so that it is subject to change at runtime. When such data are reflected as objects, these objects can, in time, come to constitute a domain-specific language, which allows users themselves to change the system as business needs dictate.

A major accomplishment of this workshop was to finally get this disparate group together, and to establish this dialog. However, we've only begun the task of fleshing out these architectural issues, uncovering the patterns, and of better defining our vocabulary. It was noted that principles of Reflection and Meta-Modeling could be used to better support the goals of AOM's and vice-versa. We are looking forward to reconvening the members of this community to continue this work. The focus of the next meeting will be around on seeing where these communities meet and how they can support one another for achieving the goal of building dynamically adaptable systems.

References

1. Francis Anderson. A Collection of History Patterns. In *Collected papers from the PLoP '98 and EuroPLoP '98 Conference, Technical Report #wucs-98-25.* Dept. of Computer Science, Washington University, 1998.
2. Brian Foote and Joseph Yoder. Architecture, Evolution, and Metamorphosis. In J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors, *Pattern Languages of Program Design 2.* Addison-Wesley, Reading, MA., 1996.
3. Brian Foote and Joseph Yoder. Metadata and Active Object-Models. In *Collected papers from the PLoP '98 and EuroPLoP '98 Conference, Technical Report #wucs-98-25.* Dept. of Computer Science, Washington University, 1998.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

5. W. E. Grosso, H. Eriksson, R. W. Fergerson, J. H. Gennari, S. W. Tu, and M. A. Musen. Knowledge Modeling at the Millennium (The Design and Evolution of Protege-2000). Internal report SMI-1999-0801, Stanford Medical Informatics, 1999.
6. P. Jeulin, M. Khlat, and L. Wilhem. GRAPHTALK, GQL et GKNOWLEDGE: Des techniques d'Intelligence Artificielle au service d'un environnement de Génie Logiciel. Technical report, Rank Rerox France, 1989.
7. R. E. Johnson and B. Woolf. Type Object. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, Reading, MA., 1998.
8. Kiczales, Riviers, and Bborow. *The art of the MOP*. MIT Press, Cambridge, MA, 1991.
9. Akos Ledeczi, Peter Volgyesi, and Gabor Karsai. Metamodel Composition in the Generic Modeling Environment. Comm. at workshop on Adaptive Object-Models and Metamodeling Techniques, Ecoop'01, Budapest, Hungary, 2001.
10. D. Manolescu. *Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development*. PhD thesis, Computer Science Technical Report UIUCDCS-R-2000-2186, University of Illinois at Urbana-Champaign, Urbana, IL, October 2000.
11. OMG. Meta-Object Facility (MOF) Specification v.1.3. TC Document ad/99-09-05, OMG, 1999.
12. Francisco Ortín-Soler and Juan Manuel Cueva-Lovelle. Building a Completely Adaptable Reflective System. Comm. at workshop on Adaptive Object-Models and Metamodeling Techniques, Ecoop'01, Budapest, Hungary, June (18) 2001.
13. John D. Poole. Model-Driven Architecture: Vision, Standards And Emerging Technologies. Comm. at workshop on Adaptive Object-Models and Metamodeling Techniques, Ecoop'01, Budapest, Hungary, 2001.
14. Reza Razavi. Active Object-Models et Lignes de Produits. In *OCM'2000*, Nantes, France, May 2000. www-poleia.lip6.fr/∼razavi.
15. N. Revault, X. Blanc, and J-F. Perrot. On Meta-Modeling Formalisms and Rule-Based Model Transforms. Comm. at workshop Iwme'00, Ecoop'00, Sophia Antipolis & Cannes, France, 2000.
16. N. Revault, H.A. Sahraoui, G. Blain, and J.-F. Perrot. A Metamodeling technique: The MétaGen system. In *Tools 16: Tools Europe'95*, pages 127–139, Versailles, France, 1995. Prentice Hall. Also RR LAFORIA95/01.
17. D. Roberts and R. Johnson. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, Reading, MA., 1997.
18. K. Smolander, P. Marttiin, K. Lyytinen, and V. Tahvanainen. MetaEdit - a flexible graphical environment for methodology modelling. In *Caise'91*, pages 168–193, Trondheim, Norway, 1991. Springer Verlag, Berlin.
19. M. Tilman and M. Devos. A Reflective and Repository Based Framework. In *Implementing Application Frameworks*, pages 29–64. Wiley, 1999.
20. J. W. Yoder, F. Balaguer, and R. E. Johnson. Architecture and Design of Adaptive Object-Models. In *Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01)*. ACM Press, 2001.
21. J. W. Yoder, F. Balaguer, and R. E. Johnson. The Architectural Style of Adaptive Object-Models. Comm. at workshop on Adaptive Object-Models and Metamodeling Techniques, Ecoop'01, Budapest, Hungary, June (18) 2001.

22. J. W. Yoder, B. Foote, D. Riehle, M. Fowler, and M. Tilman. Metadata and Active Object-Models. Workshop report, www.adaptiveobjectmodel.com/OOPSLA99, OOPSLA'99, 1999.
23. J. W. Yoder, B. Foote, D. Riehle, and M. Tilman. Metadata and Active Object-Models Workshop. In *OOPSLA '98 Addendum*. ACM Press, 1998.
24. J. W. Yoder and R. E. Johnson. MetaData Pattern Mining. Workshop report, www.joeyoder.com/Research/metadata/UoI98MetadataWkshop.html, University of Illinois at Urbana-Champaign, Urbana, IL, May 1998.
25. J. W. Yoder and R. Razavi. Metadata and Active Object-Model pattern mining. In *Ecoop'00 workshop reader*. Springer-Verlag, 2000.

Workshop participants

Our workshop gathered researchers from Adaptive Object-Modeling with others from Meta-Tool Approaches and Reflection. Table 1 below shows the list of all participants to our workshop:

surname	**first**	**company**	**email**
Alvarez	Dario	University of Oviedo	darioa@pinon.ccu.uniovi.es
Gabor	Andras	University of Debrecen	gabora@dragon.klte.hu
Gerhardt	Frank		fg@acm.org
Jonas	Richard	University of Debrecen	jonasr@math.klte.hu
Kollar	Layar	University of Debrecen	kollarl@math.klte.hu
Ledeczi	Akos	Vanderbilt University	akos@isis.vanderbilt.edu
Madacas	Bodnar	IQSoft	bodnari@iqsoft.hu
Oliver	Ian	Nokia Research Center	ian.oliver@nokia.com
Ortín	Francisco	University of Oviedo	ortin@pinon.ccu.uniovi.es
Revault	Nicolas	Univ. Cergy-Pontoise - LIP6	Nicolas.Revault@lip6.fr
Vereb	Kriantian	University of Debrecen	sparrow@math.klte.hu
Yoder	Joseph	The Refactory	yoder@refactory.com

Table 1. Workshop participants

Author Index