Using Smalltalk as a Reflective Executable Meta-Language

Appeared to MODELS 2006

Stéphane Ducasse[1 2], Tudor Girba[1]

[1] Software Composition Group, University of Bern,
www.iam.unibe.ch/∼scg
[2] Language and Software Evolution – LISTIC, Université de Savoie,
www.listic.univ-savoie.fr

Abstract. Object-oriented meta-languages such as MOF or EMOF are often used to specify domain specific languages. However, these meta-languages lack the ability to describe behavior or operational semantics. Several approaches have used a subset of Java mixed with OCL as executable meta-languages. In this paper, we report our experience of using Smalltalk as an executable meta-language. We validated this approach in incrementally building over the last decade, Moose, a meta-described reengineering environment. The reflective capabilities of Smalltalk support a uniform way of letting the developer focus on his tasks while at the same time allowing him to meta-describe his domain model. The advantage of our approach is that the developer uses *the same tools and environment* he uses for his regular tasks.

Keywords: meta behavior description, reflective language, Smalltalk

1 Introduction

Object-oriented meta-languages such as MOF [OMG97], EMOF [OMG04] or ECore [BSM+03] are often used to describe domain specific language meta-models. However, such object-oriented meta-languages only support the description of structural entities and their relationships. They do not have support for the definition of behavior, and, as such, they cannot be used to specify the operational semantics of meta-models [MFJ05].

Attempts such as the UML Virtual Machine [RFBL+01] failed similarly to capture the specification of operations at the meta level. Adaptive Object Models [RTJ05] used the Type-Object design pattern and workflow to describe at meta-level the structure and behavior of business models [YJ02]. Other approaches have used ECA rules to describe the behavior of the meta-level [DT98]. Recently, Xactium [CESW04] proposed a simple object-oriented model and imperative OCL to model state and behavior at the meta-level in an executable form. Xion [MMS+05] was an extension of OCL with imperative semantics to support the definition of action and behavior in web-modeling context. More recently, Kermeta was introduced as a meta-language that is based on a subset of Java and integrate OCL-like expressions [MFJ05].

In the late nineties we started to build a reengineering environment [DDL99,NDG05, DGLD05] and we faced the need to be able to describe not only the structure at the meta-level but also the behavior. After evaluating the different alternatives that were offered to us at that time, we decided to use Smalltalk. In this paper, we report on our experience of using Smalltalk as a meta-language to specify MOF structure *and* behavior in an uniform way.

In the next section we list the challenges we faced when building our reengineering environment emphasizing the need for an executable meta-description. In Section 3 we briefly describe Smalltalk and its reflective capabilities. Section 4 details our approach of integrating MOF in Smalltalk, and shows how we used the approach in the context of Moose. In Section 5 we evaluate the approach, and we conclude in Section 6.

2 The Need of Executable Meta-Language: the Moose Experience

Starting in 1996, our main research effort was concentrated on language design and reengineering object-oriented legacy systems [DD99b]. Since then we incrementally developed Moose, a reengineering environment [DL05, NDG05]. In this process, we felt the need to meta describe our environment to enable us to be more efficient building new tools for our reengineering research. Using meta-modeling was just a means to introduce more flexibility and extensibility in our tools and not a research topic on its own. Nowadays, Moose uses meta-descriptions to support automatic storage, browsing or annotations of models. The context had practical impact on our solution. We describe here the main constraints we faced so that the reader can assess our solution.

Not disrupting our developers. The goal of Moose is to enable other developers, mainly researchers or consultants, to develop new source code analysis, source code visualizations, metrics ...These researchers, while fluent in object-oriented programming, should not be dealing with the details of the meta-descriptions: the environment should let them express their ideas and require as less as possible for the meta-descriptions. Also, the developers that want to extend the environment should be able to do so without having to learn yet another language or formalisms.

The implications are the following ones. We do not want to use any generative techniques that would hamper developers to use their favorite environment. In particular, string manipulation and other such kind of low-level operations should not be used, because of breaking the object-oriented metaphor. The same environment should be used to program the base language and the meta one. In this way, the navigation, versioning tools, code refactorings, code browsers can be used at all levels. In particular, the developers should be able to use the same debugging tools and incremental hot recompilation (*e.g.*, editing and recompiling in the debugger) since this is one of the cornerstone of fast development in Smalltalk. Possibly the same paradigm should be used at the base and meta-level.

Even if our solution is influenced by this specific context, we believe that it presents interesting results to enable executable meta-models in practical settings. It is the recent publications on executable meta-languages Xion [MMS+05], Kermeta [MFJ05], Xactium [CESW04] and our successful work in building our reengineering environment that convinced us that our approach is worth being reported to the modeling community.

2.1 A First Analysis

In this section we discuss the reasons why we need an executable meta-language.

Why meta-data description languages are not enough? As already mentioned by [MFJ05], MOF defines operations, but not their implementation counterparts, which have to be described in text. The following example is excerpted from the MOF 2.0 Core Specification. The definition of the isInstance operation of the EMOF class Type (section 12.2.3 page 34) is given as follow:

```
Operation isInstance(element : Element) : Boolean
  /*Returns true if the element is an instance of this type or a subclass of this type.
    Returns false if the element is null*/
```

The description is informal and cannot be executed. Meta-data description languages do not support the definition of a simple behavior such as the MOFType isInstance behavior. In Moose, the MOFType class defines the method isInstance as follow:

```
MOFType>>isInstance: element
    "Returns true if the element is an instance of this type or a subclass of this type.
    Returns false if the element is null"

  ^ element isNil
      ifTrue: [false]
      ifFalse: [element metaClass == self
                or: [element metaClass allSuperclasses includes: self]]
```

The caret sign ^ is a return statement, and ifTrue:ifFalse: is the Smalltalk if-then-else construct. In our approach, Smalltalk is used to specify the meta-model behavior: MOF meta-model entity behavior is plain executable code. We did not choose Action Semantics [MTAL98] as an executable meta language for the following reasons: Action Semantics did not exist when we started, it is defined for UML models, and it is too generic for our audience and constraints.

Customizable Executable Meta-Language. In the Moose environment any entity (*e.g.*, a program element), is described by an instance of MOFClass. The description includes the way the entity should be loaded from files, saved, how it should be navigated ... Moose also allows for a developer to describe precisely how to specify the resolution of undefined references. For example, the developer is free to define the logic for creating a stub[3] creation which can be complex and dependent of the domain. The code below shows that the class FamixClass which represents the class concept in a language independent way for our analysis is in fact described by a MOFClass instance named Class [DDL99]. What is important is that the end-user developer can specify specific domain actions at the meta-model level: here the optimize: method specification defines the way stub entities may be created when code models are extracted by code analyzers or model loaders.

[3] A stub is shell-entity that is creating to represent an entity that is not reified in our model: when an access to a variable that is not extracted from the source code, we create a stub variable.

```
FAMIXClass>>mofDescription
  ^ MOFClass new
      addSuperclass: self superclass mofDescription
      name: #Class;
      optimize: [:entity | (entity belongsTo isNamespace)
                           ifTrue: [entity belongsTo addClass: entity]];
      addAttribute: (MOFAttribute new
          name: #isAbstract;
          ...
          booleanType).
       ....
```

2.2 New language or not?

Defining a new language is always a challenging (and exciting) moment as we control the features that will influence our future expression possibilities. However, developping a new language also raises practical problems such as the language performance, memory consumption, the development of libraries or development tools and the cost in teaching new developers.

Our goal was *not* to define a new meta-language. We wanted to improve our reengineering environment by making it more flexible and extensible, while in the same time, we wanted to let our developers program in an environment in which they were comfortable and efficient. We favored the practical issues, and chose to use the same language (*i.e.*, Smalltalk) for both describing the meta-model and the meta-meta-model.

In this section we described our practical constraints, and how we came to the conclusion that Smalltalk is the solution for our problem. To let the reader better understand the detail, we briefly describe in the next section the key characteristics of the Smalltalk language and its meta-model. In the subsequent section we present the architecture we chose to integrate a MOF-based architecture inside the Smalltalk one.

3 Smalltalk in a Nutshell

While Smalltalk may seem to be an old language to a certain audience, its uniformity, simplicity and elegance make it still an innovative language. For example, Smalltalk iterators have influenced OCL statements (*e.g.*, select, collect). The recent introduction of built-in queriable declarative annotations make it a powerful language for meta descriptions since we can annotate methods and query such meta-descriptions from within the language.

The Smalltalk object model is a subset of the one of Java [GR83]. In Smalltalk everything is an object and objects communicate exclusively via message passing (method invocation). This is applied uniformly in the sense that message passing is preferred to new language constructs. For example, select: is a method defined in Collection, rather than being a language construct.

Objects are instances of classes. All instance variables are private to the object[4] and all methods are public. There is single inheritance between classes, classes are objects too. A class is instance of a metaclass which has this class as its sole instance. Class methods are simply methods of the metaclasses and follow all the previous rules. For example, in the figure below, the class Workstation is an instance of the metaclass Workstation class.

The complete system is written in itself, therefore can be queried and manipulated within itself allowing powerful introspective *and* reflective facilities [Riv96].

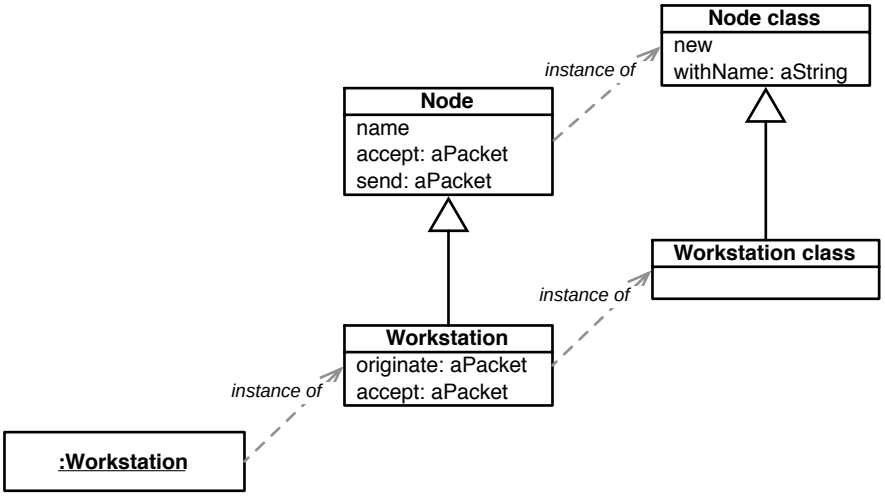

Fig. 1. The class Workstation is instance of the metaclass Workstation class

Query meta-language. Because of its reflective capabilities, Smalltalk can be easily used as a query meta-language on it own structure. For example, the following expressions query the methods defined locally, all the methods, and all the instances of the class Set.

```
Set selectors
    returns the method names defined locally
Set allSelectors size
    returns the number of methods locally and inherited by Set
Set allInstances
    returns all the instances of the class Set in the system
```

[4] Contrary to Java and C++ where private is class-based *i.e.*, two objects of the same classes can directly access their private fields.

OCL like iterators. Smalltalk offers high level iterators such as collect:, select:, reject:, includes:, do:, do:separatedBy:, occurencesOf:, and more interestingly the definition of new iterators is open and simple. The iterators are passed closures to be evaluated. For example, [:each |each even] is equivalent with *(lambda (each) (even each))*, or with (each|each->even()) in OCL.

```
#(1 2 3 4) collect: [:each | each even]
   returns: #(false true false true)
#(1 2 3 4) select: [:each | each even]
   returns: #(2 4)

| string |
string := ''
#(1 2 3)
        do: [ :each | string := string, each printString]
        separatedBy: [string := string, '-'].
string.
    returns the string '1-2-3'
```

Declarative built-in meta descriptions. Since several years, several Smalltalk implementations introduced built-in declarative annotations, called Pragmas. Pragmas are pure annotations without any behavior influence, attached to the method definitions. These annotations can be queried from the language which makes them useful as declarative registration mechanisms.

The following example shows how an application can define *at the same time* a method and several menu items that will invoke such a method. In our example, the method openFileBrowser is defined in class VisualLauncher and it consists of the last line that open the FileBrowser application. Then two annotations between < > are used to declare in this specific case that such a method can be invoked from the menu bar using the browse menu item and from the Launcher tool bar by clicking on the icon (see the Figure 2).

```
VisualLauncher>>openFileBrowser
  <menuItem: 'File Browser'  icon: #fileBrowser menu: #(#toolBar)>
  <menuItem: 'File Browser'  icon: #fileBrowser shortcut: #F2 menu: #(#menuBar file)>

  FileBrowser open
```

An annotation is defined within a method body and in addition it should first be *declared* so that the compiler can verify that the correctness of the annotations. Below we give the query example that returns a collection with the annotations named menuItem:icon:menu: defined in the system. An annotation knows the relevant meta-information about its use such as the method and class in which it is declared.

```
Pragmas allNamed: #menuItem:icon:menu:
```

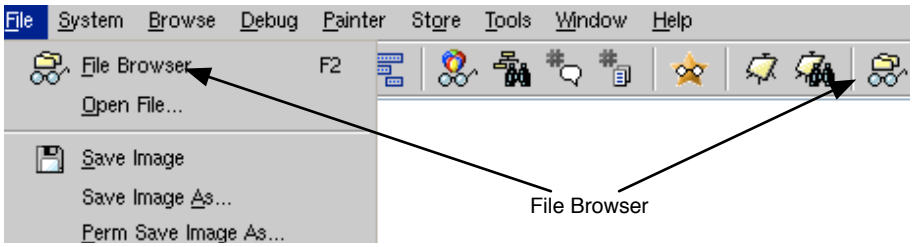

Fig. 2. The File Browser can be invoked both from the menu and from the toolbar due to the two Pragmas.

Class Extension Mechanism. Contrary to Java or C++, in Smalltalk as well as in Objective-C, we can package a method in a different package than the one the class belongs to. For example, in the example above, the class VisualLauncher is defined in one package, while the openFileBrowser is defined in another package named Tools-File Browser. As a result, this method is available on the class VisualLauncher, and consequently appears in the menu, *only* when the Tools-File Browser package is loaded.

This mechanism, called class extension, lets the developer add methods to classes that did not provide the expected behavior. Inheritance is not a solution to the problem that class extension solves since clients may still refer to the original class. In our example, extending the VisualLauncher via subclassing would not work since the menu can be extended by different clients, and we still want to open the VisualLauncher to see what tools are available [BDN05]. C# recently introduced static class extensions to improve the extensibility of the applications written with this language.

4 Integrating MOF in Smalltalk and Moose

Smalltalk being a reflective language (*i.e.*, supporting both introspection and intercession [BGW93]), it already includes a causally connected meta-description of its own run-time and structure. To introduce a fourth layer [5], we used the architecture shown in Figure 3. In the example, the Java class Point is represented as an instance of the FAMIXClass [DD99a]. The FAMIXClass is described by the instances of the class MOFAttribute and MOFClass.

Such an architecture is not new and can be seen as a validation of the nowadays well-known distinction between two conceptually different kinds of instance-of relationships: (i) a traditional and implementation driven one where an instance is an instance of its type, and (ii) a representation one where an instance is described by another entity [BG01]. Atkinson and Kühne named these two forms: form vs. contents or linguistic and logical [AK05] [AK01][6].

[5] We started in early 1997 with an entity relationship meta-meta-model then since 2003 we replaced it by a MOF-based one.

[6] In 1997, the distinction between the implementation and the representation was not clear nor described in the literature. Hence, our architecture was not influenced by existing readings, and therefore it acts as a confirmation of the related work.

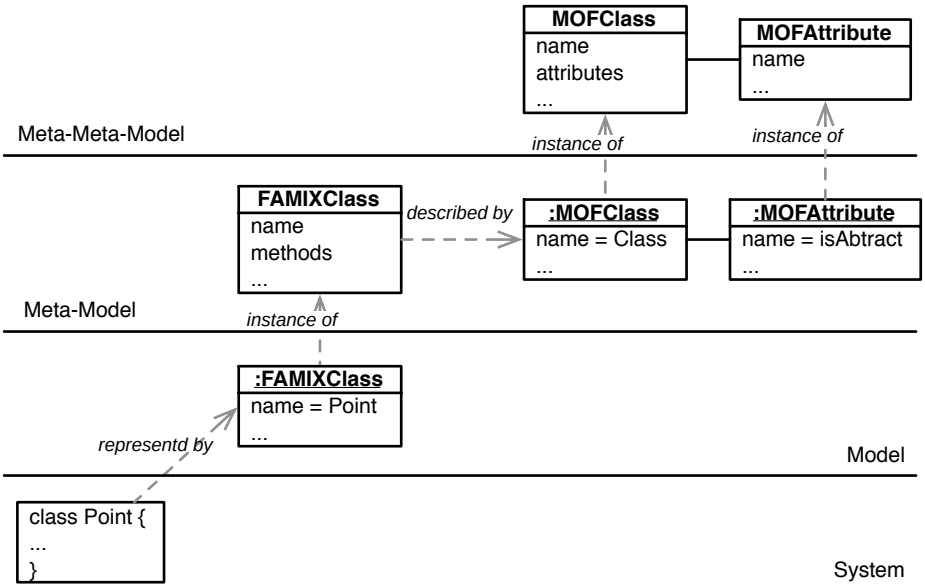

Fig. 3. Mapping Meta-Descriptions to Smalltalk.

4.1 Describing Smalltalk Classes with MOF

Because Smalltalk classes are objects we can attach the MOF description to the class objects. One possibility of providing the descriptions are like in the code below.

```
FAMIXClass class>>mofDescription
  ^ MOFClass new
        superClass: self superclass mofDescription;
        name: #Class;
        ...
        addAttribute: (MOFAttribute new
                        name: #isAbstract;
                        loadMethod: #setAbstract:;
                        saveMethod: #getAbstract;
                        booleanType)
```

In this example, we show an excerpt of the mofDescription method attached to the FAMIXClass class. The method returns a MOFClass with the name Class. Attached to the MOFClass are several attributes. For example, isAbstract is an MOFAttribute. Particular to our implementation is that we did not use MOF, but an extension of MOF. The reason for it, is that we needed to attach *executability* to the descriptions as we show in the previous section. For example, for the isAbstract attribute we added information of which methods should be used to read or store the attribute in an instance of FAMIXClass.

As shown by the previous example, the method mofDescription is a class method of FAMIXClass. In Smalltalk, the class and the instance methods are clearly separated, both in the language and in the IDE user interface. Usually, the regular programmer spends most of the time programming on the instance side. Hence, having the mofDescription on the class side is rather distant from the actual focus of the programmer. That is why, we provided another way to express meta-descriptions using Pragmas. Below we give an example of how we use the Pragmas to attach the numberOfMethods metric as a MOFAttribute to the description of FAMIXClass. The developer only has to write the regular method in the model class and how he defines a property. This illustrates how the base code is annotated with a meta-description and also how the meta-description behavior can be specified by the end-user programmer. Note that we call numberOfMethods a property, and not an attribute, as the reverse engineer thinks in terms of entities and properties, rather then classes and attributes.

```
FAMIXClass>>numberOfMethods
  <property: #NOM longName: 'Number of methods'>
  ^self methods size
```

We fill our MOF repository by querying the existing annotations. The below code shows how we compute the MOF descriptions for all the entities defined as subclass of AbstractEntity. The method traverses all the subclasses and for each of it, it initializes the description and then it queries all the defined Pragmas and transforms them into MOF annotations.

```
AbstractEntity class>>initializeAllMofDescriptions
  self withAllSubclasses do: [:each | each registerMofPackage].
  self withAllSubclasses do: [:each |
            each initializeMofDescription.
            each attachPragmasToMOFDescription]
```

4.2 Building Meta-Aware Tools

Research in reverse engineering is about creating new ways of representing software. As the representation is dictated by the meta-model, we needed the meta-model to be extensible. This is not a problem per se, but in the same time we needed to be able to browse the results and also interact with other tools via external formats. As a consequence we built several generic tools that would cope with the extensions.

To be able to communicate with third parties tools we provided generic import/export. We started with supporting the CDIF format and later we also implemented the support for XMI [TDD00]. The generic engine depends only on the meta-description of the meta-model. That is, the only thing the programmer has to do is to build his meta-model, and describe the storable attributes. Based on this, the objects in the model can be serialized in either CDIF or XMI.

The act of analyzing can be decomposed in several generic atomic actions: (i) introspection - given an entity, what are its attributes, (ii) selection - given a collection of entities, which are the entities that obey a certain rule, (iii) navigation - given an entity,

what are the nearby objects, and (iv) presentation - given a collection of entities, what is the order of the entities. In the same time, an important factor in reverse engineering research is the exposure to the data. That is why we implemented generic tools to address the four points above while being independent on the type of data. Again, we accomplished this by making the tools dependent only on the meta-descriptions [DGLD05].

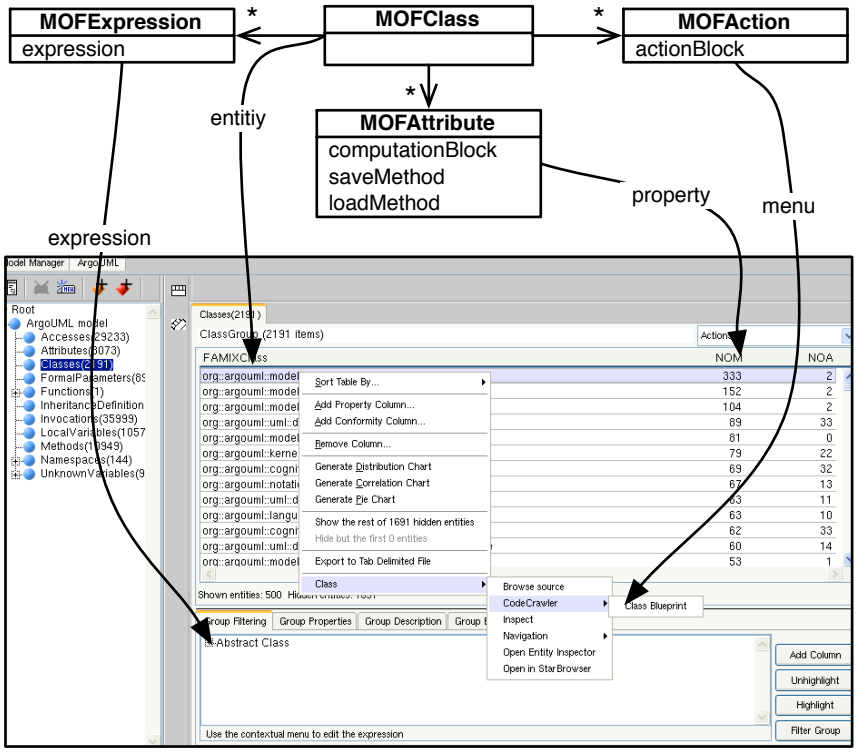

Fig. 4. We extended MOF with new entities and new methods to hook in the execution. The Moose Browser is a generic tool based on the meta-descriptions.

Because of the extension possibilities, Moose enabled several directions of research in reverse engineering. As a result, several techniques have been implemented to deal with the diversity of data, techniques which are orthogonal to the type of data. As a consequence, we have implemented a mechanism for integrating these techniques. Our solution was to extend MOF with other types of annotations. One such an annotation is the MOFAction that a tool can perform on an entity. Based on this annotation we can build a menu, and different tools can register themselves to the context they can handle.

Figure 4 shows the different extensions we performed on MOF as well as one application in building a generic browser. We added the information about loading and saving an attribute, and we added the possibility of hooking in a computation block that

would be executed if the attribute is not already computed for a given entity. We also added two new classes for Action and Expression. The Action represents a particular action that can be triggered on a certain type of entity, while the Expression is a boolean query that shows whether an entity obeys the rule or not.

Figure 4 also shows how the generic browser of Moose uses the meta descriptions. The mapping between the different parts of the browser and the meta-descriptions are denoted with arrows that also show how the meta-descriptions are seen by the user. For example, by selecting an entity we can trigger its menu which is composed of actions. In the figure, we selected a FAMIXClass and in its menu we have a CodeCrawler submenu. One visualization defined in CodeCrawler is the Class Blueprint, and it can be applied on any class through the contextual menu [DL05]. The code below shows the method that CodeCrawler uses to extend the FAMIXClass to spawn the Class Blueprint. Note that the below method is packaged in CodeCrawler, and not in Moose where FAMIXClass is defined. Like this, we can trigger the menu action *only* when CodeCrawler is loaded.

```
FAMIXClass>>openClassBlueprint

    <action: 'Class Blueprint' category: 'CodeCrawler'>
    CodeCrawler openClassBlueprintOn: self
```

CodeCrawler is a generic visualization tool based on a graph model [LD05]. The main technique implemented by CodeCrawler is called polymetric views which maps on the nodes different measurements. As Moose provides the description of the measurements computable on the entities, CodeCrawler offers an interactive tool for the user to set the mapping between the measurements and the visualization properties.

4.3 Using Meta-Descriptions For Generating Meta-Models

While our approach is not MOF-compliant, it still holds the good property that we can query and manipulate the meta-description and run-time of the model and programs themselves. Indeed the fact that Smalltalk is reflective makes it possible to query the run-time or structural representation of the language itself and to modify it in a causally connected way [Riv96]. While we favored a code centric approach, we also believe in generative ones when appropriate. Moose supports the generation of meta-described meta-models from MOF description: from a MOF description, the system can generate classes representing new models and their associated descriptions. However, while the generation of initializers, accessors and other structural navigation facilities is trivial (and ressemble to the work on the UML virtual machine [RFBL$^+$01]), the behavior is expressed as plain Smalltalk methods.

As such this domain generation can be seen as a simple model transformation. Tools such as VAN [GÛ5] which enables the definition of temporal, history analyses, are based on the transformation of models: starting with the structural model we can build the historical meta-model [GD06]. In this case too, we describe the transformation itself as Smalltalk code: we can query the models entities and manipulate them to generate new entities [Pol05].

5 Evaluation

Our approach takes the best of the object-oriented programming and meta-modeling worlds and uses it in a practical setup. One the one hand, we continue to use only one paradigm and environment. This helps our developers to develop their own applications or to extend our environment. They do not have to learn a new language and they stay within their known environment. On the other hand, we provide a meta-described extensible environment in which meta-interpreters can deliver their power. Using Smalltalk as a meta-modeling language provided us with several advantages:

- Executability – We obtained a meta-model that is executable and that can be extended using the Smalltalk language constructs (declarative annotations, class extensions).
- Good performance – Because we use a professional Smalltalk environment, we can focus on our main activities and we do not have to worry about performance that building our own language would have implied.
- Tools support – We can use the same toolsets (debugger, version management, refactorings) to develop both our domain and our meta-domain.
- Extensibility – Using class extensions we can package our meta-model extensions with the domain entities they describe. But we can also package new tools orthogonally to the base domain and even meta-model. For example, we can package all the navigation facilities independently of the rest even if the code is attach conceptually to the core entities.

However, our approach is not completely MOF compliant since the MOF does not describe execution. It does not follow a traditional MDA decomposition. As such, model transformation of behavior may be more difficult than if we would have been using a model to describe the behavior as suggested by Action Semantics [MTAL98], or a dedicated language such as Kermeta [MFJ05]. However since Smalltalk also offers a reflective API, we developed some simple meta-model transformations using Smalltalk.

The common objection against using a programming language as an executable meta language can be summarized by saying that languages provide too much or too few. Muller et al. said: “Existing programming languages already provide a precise operational semantic for action specifications. Unfortunately, these languages provide both too much (*e.g.*, interfaces), and too few (they lack concepts available in MOF, such as associations, enumerations, opposite properties, multiplicities, derived properties...).” [MFJ05]

However, like other mainstream object-oriented programming language, Smalltalk does not support associations, derived entities, opposite properties directly in the language, and because of that the developer may be facing implementation decisions instead of meta-modeling ones. From the language point of view, the Smalltalk meta-model is minimalist. We believe that given our constraint of use a programming language to describe both our base domain and the meta-description, the choice of Smalltalk was adequate and offered a good and practical solution to our problems.

6 Conclusion and Future Works

To make our reengineering environment more flexible and extensible, we introduced a meta-description and used this meta-description to build extensible reengineering tools. We used Smalltalk as an executable meta-language, and we simplified our code and its logic by factoring knowledge at the meta-level. Our developers could focus on their tasks without having to learn new languages and new tools that would not be casually connected with the objects they manipulate.

We show how a four layer architecture can be introduced in a reflective language, validating the distinction between instantiation and representation links in meta-modeling tools architectures [BG01, AK05]. We believe that our approach can be applied in other mainstream programming languages, and we can imagine doing the same using EMF. Still to gain the maximum from this approach we believe that being able to annotate methods, to query these annotations, and to package methods independently from the classes they belong are important factors.

Our solution influenced our reengineering environment in several ways:

- The decision to use Smalltalk as a meta-language makes it possible to use all the tools provided by the development environment: browser, debugger, versioning, testing, refactoring, etc. Moreover it eases the entry level as developers do not need to learn another language.
- Having first class meta-description as ordinary objects also helps manipulating the meta-model, and building flexible tools based on it. For example, we can develop meta-interpreters as simple methods or objects.
- Having a meta-description greatly enhances the possibilities to refactor and change existing code, since a change to the meta-model only needs to be performed at one single place, without requiring to change the generic tools (*e.g.*, import/export).

By letting the end-user programmer naturally annotate his base code with meta-descriptions, we narrow the gap between what are traditionally seen as complex and separated tasks. We coined this approach literate meta-programming [Knu92].

In the future, we plan to describe the Smalltalk meta-entities with MOF to get a fully MOF-compliant Smalltalk. For example, the class CompiledMethod could be described to represent the fact that a method can be abstract and that it has parameters. We would then have a completely executable MOF meta-language.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Recast: Evolution of Object-Oriented Applications (SNF 2000–061655.00/1)” and the French National Research Agency (ANR) for the project “Cook: Rearchitecting object-oriented applications”(2005-2008).

References

[AK01] Colin Atkinson and Thomas Kuehne. The essence of multilevel metamodeling. In *Proceedings of the UML Conference*, number 2185 in LNCS, pages 19–33, 2001.

[AK05] Colin Atkinson and Thomas Kuehne. Concepts for comparing modeling tool architecture. In *Proceedings of the UML Conference*, number 3713 in LNCS, pages 19–33, 2005.

[BDN05] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 177–189, New York, NY, USA, 2005. ACM Press.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proceedings Automated Software Engineering (ASE 2001)*, pages 273–282, Los Alamitos CA, 2001. IEEE Computer Society.

[BGW93] D.G. Bobrow, R.P. Gabriel, and J.L. White. Clos in context — the shape of the design. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS perspective*, pages 29–61. MIT Press, 1993.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.

[CESW04] Tony Clark, Andy Evans, Paul Sammut, and James Willans. Applied metamodelling: A foundation for language driven development, 2004.

[DD99a] Serge Demeyer and Stéphane Ducasse. Metrics, do they really help? In Jacques Malenfant, editor, *Proceedings LMO '99 (Languages et Modèles à Objets)*, pages 69–82. HERMES Science Publications, Paris, 1999.

[DD99b] Stéphane Ducasse and Serge Demeyer, editors. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Bern, October 1999.

[DDL99] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In Francoise Balmas, Mike Blaha, and Spencer Rugaber, editors, *Proceedings WCRE '99 (6th Working Conference on Reverse Engineering)*. IEEE, October 1999.

[DGLD05] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[DL05] Stéphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31(1):75–90, January 2005.

[DT98] Martine Devos and Michel Tilman. Incremental development of a repository-based framework supporting organizational inquiry and learning. In *OOPSLA'98 Practioner's Report*, 1998.

[GÔ5] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, November 2005.

[GD06] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *International Journal on Software Maintenance: Research and Practice (JSME)*, 18:207–236, 2006.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[Knu92] Donald E. Knuth. *Literate Programming*. Stanford, California: Center for the Study of Language and Information, 1992.

[LD05] Michele Lanza and Stéphane Ducasse. Codecrawler–an extensible and language independent 2d and 3d software visualization tool. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 74–94. Franco Angeli, Milano, 2005.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.

[MSFB05] Pierre-Alain Muller, Philippe Studer, Frédérick Fondement, and Jean Bézivin. Independent web application modeling and development with netsilon. *Software and System Modeling*, 4(4):424–442, November 2005.

[MTAL98] Stephen J. Mellor, Steve Tockey, Rodolphe Arthaud, and Philippe LeBlanc. Software-platform-independent, precise action specifications for UML. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, number 1618 in LNCS, pages 281–286, 1998.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[OMG97] Object Management Group. Meta object facility (MOF) specification. Technical Report ad/97-08-14, Object Management Group, September 1997.

[OMG04] Object Management Group. Meta object facility (MOF) 2.0 core final adopted specification. Technical report, Object Management Group, 2004.

[Pol05] Damien Pollet. *Une architecture pour les transformations de modèles et la restructuration de modèles UML*. PhD thesis, Université de Rennes 1, June 2005.

[RFBL+01] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, , and Nosa Omorogbe. The architecture of a uml virtual machine. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01)*, pages 327–341, 2001.

[Riv96] Fred Rivard. Pour un lien d'instanciation dynamique dans les langages à classes. In *JFLA96*. INRIA — collection didactique, January 1996.

[RTJ05] Dirk Riehle, Michel Tilman, and Ralph Johnson. Dynamic object model. In *Pattern Languages of Program Design 5*. Addison-Wesley, 2005.

[TDD00] Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. FAMIX: Exchange experiences with CDIF and XMI. In *Proceedings of the ICSE 2000 Workshop on Standard Exchange Format (WoSEF 2000)*, June 2000.

[YJ02] Joseph W. Yoder and Ralph Johnson. The adaptive object model architectural style. In *Proceeding of The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02)*, August 2002.