Meta-Environment and Executable Meta-Language using Smalltalk: an Experience Report

Stéphane Ducasse, Tudor Gîrba, Adrian Kuhn, Lukas Renggli

SPECIAL SECTION PAPER

Meta-environment and executable meta-language using smalltalk: an experience report

Stéphane Ducasse · Tudor Girba · Adrian Kuhn · Lukas Renggli

Abstract Object-oriented modelling languages such as EMOF are often used to specify domain specific meta-models. However, these modelling 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 experience report we show how we use Smalltalk as an executable meta-language in the context of the MOOSE reengineering environment. We present how we implemented EMOF and its behavioral aspects. Over the last decade we validated this approach through incrementally building a meta-described reengineering environment. Such an approach bridges the gap between a code-oriented view and a meta-model driven one. It avoids the creation of yet another language and reuses the infrastructure and run-time of the underlying implementation language. It offers an uniform way of letting developers focus on their tasks while at the same time allowing them to meta-describe their domain model. The advantage of our approach is that developers use *the same tools and environment* they use for their regular tasks. Still the approach is not Smalltalk specific but can be applied to language offering an introspective API such as Ruby, Python, CLOS, Java and C#.

S. Ducasse (✉) · T. Girba · A. Kuhn · L. Renggli
Software Composition Group, University of Bern,
Bern, Switzerland
e-mail: stephane.ducasse@free.fr
URL: http://www.iam.unibe.ch/~scg

S. Ducasse
Language and Software Evolution Group, ADAM Team,
INRIA-Lille Nord Europe, LIFL/USTL1, CNRS UMR,
8022 Lille, France
e-mail: stephane.ducasse@inria.fr

1 Introduction

Object-oriented modelling languages such as MOF, EMOF [23] or ECore [7] are often used to describe domain specific meta-models. However, these modelling 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 [32].

Attempts such as the UML Virtual Machine [38] failed similarly to capture the specification of operations at the meta-level. Adaptive Object Models [40] used the Type-Object design pattern and workflow to describe at meta-level the structure and behavior of business models [42]. Other approaches have used ECA rules to describe the behavior of the meta-level [17]. Recently, Xactium [8] proposed a simple object-oriented model and imperative OCL to model state and behavior at the meta-level in an executable form. Xion [34] 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 an executable EMOF compliant meta-language with OCL-like expressions [19,32].

In the late nineties we started to build MOOSE, a reengineering environment [10,12,15,36], 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. For the meta-level we first extended Smalltalk with a simple entity relationship meta-meta-model. Then we

migrated to MOF and finally EMOF. In this paper, we report on our experience of using Smalltalk as a meta-language to specify EMOF structure *and* behavior in an uniform way. Such an approach avoids to have to design and implement yet another language and it allows the developer to only use one single language to program at different meta-layers.

This paper is an extension of our previous paper in which we show how we integrated in Smalltalk a self-described EMOF, and how we used Smalltalk as an executable meta-language [14]. The novelties are:

Complete meta-environment. In the previous paper, we focused on meta-descriptions for one application alone. In this paper, we present FAME, a complete meta-environment, that extends the complete Smalltalk environment. Not only domain classes, but also system classes from the standard libraries may be decorated with descriptions and can thus be serialized or browsed using the same infrastructure and UI. In particular, while EMOF is self-described on paper, it is now self-described in our implementation.

Complete development-environment. As we reuse Smalltalk to implement the behavioral semantics of the meta-model, the same tools (code browser, debugger, refactoring tools, etc.) can be used to maintain both code and meta-model. Also, we have extended the UI of the object inspector to add the meta-description as an additional view.

Improved implementation. We have enhanced our MOF implementation (called FAME) from MOF 1.4 to EMOF 2.0, the main difference between the two standards being that associations are modeled as opposing attributes rather than as first-class elements. Thus, the new implementation fits better to Smalltalk, which also does not have associations as first-class objects.

In the next section we list the challenges we faced when building our reengineering environment emphasizing the need for an executable meta-description. In Sect. 3 we briefly describe Smalltalk and its reflective capabilities (an overview of the Smalltalk syntax can be found in Appendix A). Section 4 details our approach of integrating MOF in Smalltalk and we describe our implementation called FAME. The following sections present some of the infrastructure that we built on the meta-description, and how we used the approach in the context of the MOOSE reengineering environment and the FAMIX meta-model [12,18]. In Sect. 7 we evaluate the approach and present future work, and eventually conclude in Sect. 8 with a summary.

2 The need of executable meta-language

As "meta" is an overloaded term, we start with a number of definitions (for further details on meta and reflexive

architecture we invite the reader to read Maes's work [29, 30]): *"A meta-system is a computational system that has as its domain another computational system called its object-domain. [...] A meta-system has a representation of its object-system in its data. Its program specifies* meta-computation *about the object-system and is therefore called a meta-program. A programming environment has a meta-level architecture if it has an architecture which supports meta-computation."* Muller et al. proposed to see executable meta-models as meta-data + actions [32]. We define an executable meta-language as a meta-program that can be executed. Such meta-language can be self-described.

Starting in 1996, our main research effort was concentrated on reengineering object-oriented legacy systems [9,11]. Since then we incrementally developed MOOSE, a modular reengineering environment [16,36]. 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. We started using an entity-relationship meta-meta-model, then moved to MOF and recently to EMOF. Nowadays, MOOSE uses meta-descriptions to support various activities such as model persistency and automatic UI generation. This context had practical impact on our solution. We describe here the main challenges that we faced so that the reader can assess our solution. They basically can be summarized by two questions:

– How can we get the best of the two worlds? We would like to continue with our code-oriented development, and at the same time benefit from using a meta-modeling approach. It should be possible to edit code and meta-models, exactly the same way using the same set of standard development tools.

– How to make the meta-model executable? The need for imperative semantics at the meta-model level is well documented [8,32]. We want to avoid generated code, and we want to use the same programming language to express the meta-semantics in the same language as the rest of the code.

Even if our solution is developed in the specific context of Smalltalk, we believe that it presents interesting results and shows how executable meta-models can be used in practical settings. It is the recent publications on executable meta-languages Xion [34], Kermeta [32], Xactium [8] and our successful work in building our reengineering environment that convinced us that our experience is worth being reported to the modeling community. We believe that our approach is applicable to other languages and in particular to languages

offering introspective capabilities like CLOS [3,13], Ruby, Python, and Java. In fact, a subset of FAME has recently been ported to both Python and Java[1].

The goal of MOOSE, our reengineering environment, is to enable other developers, mainly researchers or consultants, to develop new analysis techniques, visualizations, metrics, etc. These researchers, while fluent in object-oriented programming, should not be hampered with the details of meta-modeling: the environment should let them express their ideas with as less additional programming effort as possible. Developers that want to extend the environment should be able to do so without having to learn a new language or new tools. They should be able to use their "native" programming language and their favorite development tools.

The implications are the following ones. We do not want to use generative techniques that would yield forbidden code, i.e., code that the developers must not edit in their IDE. String manipulation and other such kind of low-level operations are to be avoided, 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, debuggers, code refactorings, code browsers can be used at all levels. The same argumentation applies to the semantics of the meta-model.

The most common semantics of a meta-model are constraints, e.g., on derived attributes or as pre- and postconditions on operations. These constraints are usually expressed in OCL, a purely declarative language that lacks imperative power.

As a consequence, several approaches added imperative features to OCL-extensions in their own object-oriented kernels [8,32,34]. However, using a different language involving different tools to define meta-program is exactly what we want to avoid, rather the same environment should be used at all levels. In particular, the developers should be able to use the same debugging tools and incremental hot recompilation, i.e., editing and recompiling in the debugger, since this is one of the cornerstone of productive Smalltalk development. Possibly the same paradigm should be used at the base and meta-level.

Therefore, we needed to extend our implementation of EMOF to allow for executable semantics written in Smalltalk at the meta-level as well. For example, we would like to add the semantics of a derived property using Smalltalk code similar to the following constraint.

context Person :: numberOfAllChildren
post: result = self.children–>size() + self.children–>sum
    (numberOfAllChildren)

In the next section we present Smalltalk. Readers fimilar with the language may skip to Sect. 4 that explains how
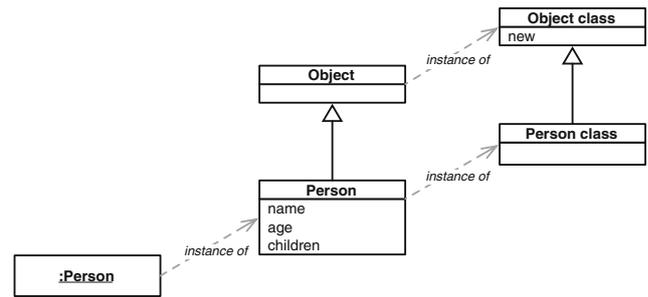
Fig. 1 The class Person is an instance of the metaclass Person class

we implemented FAME, our running meta-environment in Smalltalk that fulfills the requirements formulated in this section.

3 Smalltalk in a nutshell

Smalltalk is a pure object-oriented language. Smalltalk programs consists of objects, classes, methods and messages [21]. The support for 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.

In this section we describe the Smalltalk object model [21] and we point out some similarities of the Smalltalk language with OCL, the object constraint language.

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 favored over other language constructs, even over control-flow constructs. For example, ifTrue:ifFalse is a method defined in Boolean, rather than being a language construct as in other languages.

Objects are instances of classes. All instance variables are private to the object and all methods are public. There is single inheritance between classes, classes are objects too. A class is an 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 Fig. 1, the class Person is an instance of the metaclass Person class.

The complete Smalltalk system (library, compiler, environment) is written in itself, therefore can be queried and manipulated within itself allowing powerful introspective *and* reflective facilities [39]. In particular, the Smalltalk meta-level can be queried.

Query 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 Person. For comparison, equivalent OCL statements are shown beside the Smalltalk code.

| Smalltalk	OCL
Person selectors	Person.ownedOperation
Answer the method names defined locally.	
Person allSelectors size	Person.operation->size()
Answer the number of methods locally and inherited by Person.	
Person allInstances	Person.allInstances()
Answer all the instances of the class Person in the system.	

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

With a sequenceable collection col containing the numbers from 1 to 4 we can evaluate the following expressions (an overview of the Smalltalk syntax can be found in Appendix A):

Smalltalk	OCL
col := #(1 2 3 4)	**let** col=Sequence{1, 2, 3, 4} **in**
col collect: [:each | each even]	col->collect(even)
Answer the collection #(false true false true).	
col select: [:each | each even]	col->select(even)
Answer the collection #(2 4).	
col inject: 0 into: [:each :next | each + next]	col->iterate(v, a = 0 | v + a)
Answer the sum of all the numbers, 1 + 2 + 3 + 4 = 10.	

Declarative meta descriptions. The Smalltalk language provides declarative annotations called Pragmas. Pragmas are pure annotations that can be attached to method definitions and do no influence the behavior of the method. These annotations can be queried directly from the language which makes them useful as a declarative registration mechanism.

The following example shows how an application can define a method and several menu items that will invoke this method in the very same method body. In our example, the method openFileBrowser is defined in class VisualLauncher and it consists of a line that opens the FileBrowser application. The two annotations between < > are used to declare that such a method can be invoked from the menu bar using the browse menu item and from the tool bar by clicking on the icon, see the Fig. 2.
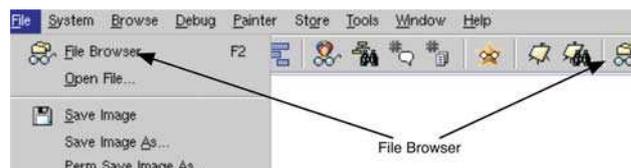
Fig. 2 The File Browser can be invoked both from the menu and from the toolbar due to the two Pragmas

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

  FileBrowser open
```

Below we give the expression that returns a collection of all 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. The system uses this expression to collect all the menu items for the launcher application and therefore remains extensible as the menu and toolbar structure is not hardcoded into the class.

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

Class Extension Mechanism. In Smalltalk as in CLOS or Objective-C 2.0, a method can be part of a different package than the package its class definition belongs to. In the example above, the class VisualLauncher is defined in the package Tools-Misc, while the method openFileBrowser is defined in Tools-File Browser. As a result, this method is available on the class VisualLauncher, and consequently only appears in the menu, when the Tools-File Browser package is loaded.

This mechanism, called class extension, lets the developer extend existing classes with new behavior. Inheritance is not a solution to the problem as clients should still refer to the original class. In our example, extending the VisualLauncher via subclassing would not work as the menu could have been extended by different clients, and we still want to refer to VisualLauncher to open and see all the tools that are available [4].

Please note that while class extensions are extensively used in Smalltalk, the mechanism can also be found in other languages. For example in Java methods can be added to other classes using AspectJ [27].

Class extensions and meta annotations are crucial to extend an *existing* system with new behavior and meta-data. The class extensions allow one to integrate new code tightly in the adequate and responsible class. Method annotations add meta information to the code, so that the existing system knows about the new functionality and, for example, can provide entry points through the user interface.

4 Integrating a self-described EMOF in smalltalk

In this section we describe how we integrate FAME, an EMOF compliant and self-described meta-environment, in Smalltalk. As motivated in Sect. 2, the rationale is to describe existing source code entities using meta-descriptions that comply to EMOF We want to describe the domain classes in a generic way, such that certain parts of our application (such as user interface and serialization) can be written as generic interpreters using these classes' descriptions rather than requiring custom code for each single domain class. In particular, we present how we extended EMOF with executability.

Smalltalk is a reflective language (i.e., supporting both introspection and intercession [6]), it already includes a causally connected meta-description[2] of its own run-time and structure in a similar fashion as CLOS and its Meta-Object protocol [26]. To bridge the two worlds (EMOF and Smalltalk) we used the architecture shown in Fig. 3. In the example, the class Person is described by an instance of the EMOF.Class class, and instances of the class Person are meta-described by that EMOF.Class description.

Such an architecture 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 [5]. Atkinson and Kühne [1,2] named these two kinds: form vs. contents or linguistic vs. ontological. However, back in 1997, when development of MOOSE started, the distinction between instance-of and described-by relationships was neither clear nor described in the literature. Hence, our architecture acts as a confirmation of the related work as it was not influenced by existing readings.

We implemented the EMOF core of MOF 2.0 as meta-description framework, and which is the main focus of this paper, executable behavior by reusing Smalltalk's language and object system. In our implementation EMOF is self-described and its behavior is implemented using Smalltalk.

4.1 The triangle between instance, class and description

In FAME each object may take part in an *instance-of* relationship with its Smalltalk class, and an equivalent *meta-described-by* relationship with its EMOF class. To avoid confusion between the object models of Smalltalk and EMOF,
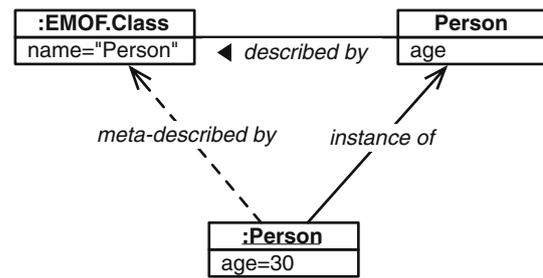
Fig. 3 Relation between an instance of Person and both its meta-description and its Smalltalk class

in this paper, we refer to Smalltalk classes as class and to EMOF instances as meta-description objects.

The semantics of these two views is as follows: the *instance-of* represents the implementation view of an object, including all implemented methods, whereas *meta-described-by* represent the domain view of the object, including all properties relevant for the application's domain. The relation between methods and properties is neither surjective nor injective, on the one hand, there may be implementation specific methods that do not access application properties, on the other hand, there may be properties whose value is not accessed by methods but rather stored in a dictionary for example.

As illustrated on Fig. 3, there is a triangular relation between object, class and description:

- *instance-of* relates objects to classes,
- *meta-described-by* relates objects to meta-description objects, and
- *described-by* related classes with meta-description objects.

All associations are navigable both ways: the *instance-of* association is managed natively by Smalltalk, whereas *meta-described-by* and *described-by* are managed by FAME:

```
"Native Smalltalk methods"
Object >> class.
Class >> allInstances.

"Fame methods"
Object >> metaDescription.
EMOF.Class >> allModelElements.

Class >> asMetaDescription.
EMOF.Class >> asSmalltalkClass.
```

For most uses of FAME in the context of MOOSE we preserve a one-to-one mapping between each Smalltalk class and its corresponding meta-description on FAME's meta-model layer (except for the implementation of EMOF itself, whose classes are mapped to FAME's meta-meta-model layer

---

instead). However, FAME can manage both multiple descriptions for a Smalltalk class and descriptions that do not have Smalltalk representation.

On Fig. 3, the Smalltalk object :Person is an instance of the Smalltalk class Person and meta-described by an instance of EMOF.Class named "Person". Further, the class specifies an instance variable "age" which is accessible in the instance using the accessors age() and age(int) (or age and age: respectively in Smalltalk lingo), and described in the meta-description by a contained instance of EMOF.Property. We show below how we create the meta-description of Person:

```
person := EMOF.Class new.
person name: #Person.
person ownedAttributes
  add: (EMOF.Property new name: #name; type: String primitive);
  add: (EMOF.Property new name: #age; type: Integer primitive);
  add: (EMOF.Property new name: #children; type: person;
     lower: 0; upper: Unlimited positive).
```

Please note that this description of Person is manually defined manually since this is an example. In the running meta-environment however, all meta-descriptions declared using method annotations and assembled dynamically, as described in the next section.

The *meta-described-by* relation of FAME's meta-meta layer, i.e., how its EMOF implementation describes itself, deserves a deeper discussion. The diagram in Fig. 4 offers an illustration of this relationship. Each meta-description is an instance of the Smalltalk class EMOF.Class, which is, in turn, meta-described using an instance of itself. This means that on FAME's meta-meta layer, the triangle is rather a pair of objects with all three associations of the triangle linking between the same two participants. Not only is the M3 layer described using descriptions of itself, but also its implementation is described using instances of itself.

As a consequence, source code and meta-model and meta-meta-model are all written in the same language and treated in the same fashion. Note that this bootstrap is not new as it is present in the core of the CLOS and Smalltalk metaclass core.

4.2 Declaration of meta-descriptions in source code

The declaration of meta-descriptions can either be done in a separate location (e.g., external XML files) or embedded in the source code using method annotations. In our experience the latter is better, as storing meta-description in a separate location is not optimal. Information that belongs together is stored in two places, which leads to duplication and thus the danger of getting out of sync. In particular, since external files may not be taken into account by refactorings and

integrated version control system. Hence, we chose to embed the declaration in the actual source code using annotations.

At the source code level, the relation between accessors and property descriptions is maintained with method annotation, as shown below:

```
Person >> age
  <property: #age type: Integer>
  ^age

Person >> age: anInteger
  <property: #age type: Integer>
  age := anInteger
```

For each method annotation, FAME creates an EMOF. Property instance and for each class annotation an EMOF.Class instance.

The relation between classes and meta-descriptions is not as straightforward as we would like, as VisualWorks does not feature class annotations. Hence, as a workaround, each Smalltalk class has a fix-named method containing a method annotation, which FAME relates to the class based on naming convention. The same scheme is used to meta-describe properties without an implementing method, e.g., derived attributes.

This solution prevents us to maintain several meta-descriptions related to one class, but it is good enough for our purpose, as in the context of Moose we mainly use a one-to-one mapping between classes and meta-descriptions. However, FAME can handle several descriptions to be attached to the same class. In this case, the programmer can either define a new fix-named prefix to be used for another meta-description, or he needs to maintain the mapping programmatically.

4.3 Extending the triangle with executability

Both the class and the meta-description describe a different view of an object's properties and operations. The class describes (and implements) attributes and methods in terms of Smalltalk. The meta-description describes properties and operations in terms of EMOF, but does not provide implementation neither for the operations behavior nor for the properties setters and getters.

When executing ordinary Smalltalk code, the Smalltalk run-time is aware of the *instance-of* relationship only and executes code accordingly. The question is how the meta-description layer can access the domain it describes. For example, how can we obtain, given a Person object, the actual value of the age EMOF property? In particular, it may happen that there is not a direct mapping between the meta-description level and the underlying domain classes. For example, in the case of a derived property it is frequent that
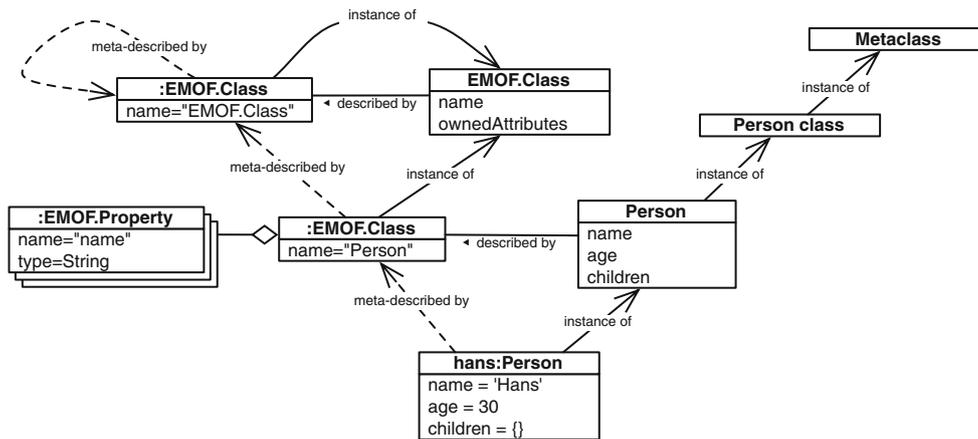
Fig. 4 The complete architecture bridging the Smalltalk and the EMOF meta-model
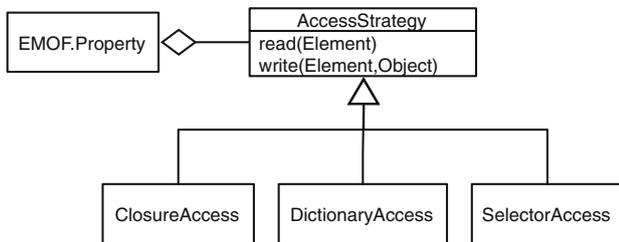

Fig. 5 Extension of EMOF.Property with binding to behavior in Smalltalk

the equivalent Smalltalk method is absent from the domain class. Therefore, we have to find a way to store and manage behavior in terms of EMOF. Bridging these two different levels is one aspect of bringing executability to the meta-level, the other being that EMOF classes are implemented within an implementation language, Smalltalk in our case.

Figure 5 illustrates how we extend EMOF.Property with an access strategy to achieve executability at the meta-level. An access strategy instance knows how to invoke the setter and getter methods associated with an EMOF attribute. In the same way, we add an additional default value strategy to the Property and also extend EMOF.Operation with an invocation strategy. There are two ways to bind behavior to a property (i) first we can bind closures (called blocks in Smalltalk) to the property, which are then performed on each access, (ii) we can bind method names to the property, which are then used to call an existing Smalltalk method using reflection. The third strategy shown on the figure, the DictionaryStrategy, provides glue code to store properties in a general purpose dictionary.

Using the ClosureAccess it is possible to define the semantics of a derived attribute as follows:

```
(Person asMetaDescription at: #numberOfAllChildren)
    strategy: (ClosureAccess new
      getBlock: [ :element |
        element children size + element children sum: [ :child |
          child numberOfAllChildren ]]).
```

The difference between this example and the OCL constraint in Sect. 2, is that the above implementation is actually executed each time the derived property is accessed at runtime, whereas a constraint is checked during modeling time to verify that a model conforms to its meta-model.

Note that the closure access strategy is useful to bridge model when there is not a one to one mapping between the level and that extra computation should be performed.

5 Building meta-aware infrastructural tools

In this section, we present the tools that form the important parts of the infrastructure of our environment: model import, meta-model generation, and access and navigation between objects at runtime. Whereas, in the following section we show more precisely the navigation in the context of the MOOSE reengineering environment and the FAMIX code meta-model.

5.1 Importing and exporting models

Import and export is an important benefit of using meta-descriptions. Having meta-described objects, we can automate the serialization framework to become a mere interpreter of the meta-description.

- To store a set of elements, for each element, the type and all non-derived properties are serialized to a stream.
- To load a set of element, the stream is read and for each entry an object of the according type is created and its non-derived attributes read from the stream and set in the newly created instance.

Special care has to be taken to resolve cyclic dependencies between objects.

To allow for different file formats, we defined MSE, an intermediate format that facilitate the serialization of

```
(Moose.Model
  (entity
    (FAMIX.Namespace (id: 1)
      (name 'Sandbox')
    )
    (FAMIX.Class (id: 2)
      (name 'Foo')
      (belongsTo (idref: 1))
    )
    (FAMIX.Method
      (name 'zork')
      (signature 'zork()')
      (belongsTo (idref: 2))
    )
    (FAMIX.Attribute
      (name 'bar')
      (belongsTo (idref: 2))
    )
  )
)
```
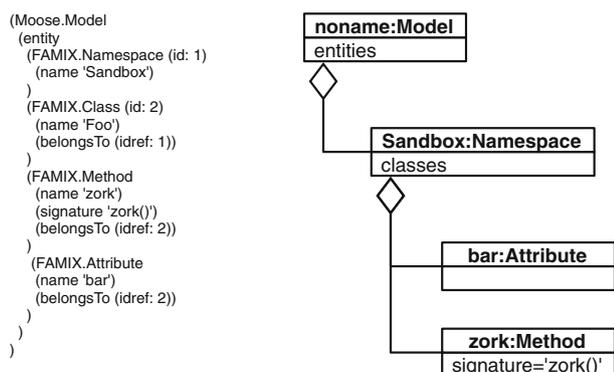

Fig. 6 A sample MSE intermediate and the corresponding object structure

elements at any meta-layer. MSE is based on literal arrays and a LISP-like syntax. Please refer to Fig. 6 for an MSE example.

FAME is an open environment in the sense that it accepts different kinds of source code information coming from external plugins. We use MSE as intermediate format for all available import and export plugins. Initially, we supported CDIF and XMI files, whereas currently KM3 [24] and ECORE files are supported, as well as savefiles in the MSE format itself.

5.2 Importing other meta-models

FAME is based on existing domain classes annotated with the declarations of their corresponding meta-model. However, when importing a third-party meta-model such as EMF, these domain classes may be missing. As we do not want to implement missing classes by hand, first we import the meta-model (which can be expressed using EMF) and create description instances that conform to the EMOF meta-meta-model, then use these imported descriptions to generate the missing classes and annotate the generated classes with declarations. This way we get to the same point as if the classes would have been there in the first place: we get existing domain classes annotated with the declarations of their corresponding meta-model.

FAME 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 resemble to the work on the UML virtual machine [38]), more sophisticate behavior can not be infered from model data alone.

5.3 Inspector

Smalltalk applications are developed at runtime, they do not require to be restarted when a method is compiled or a class

is defined. Using FAME this is no different: developers create, modify and remove meta-descriptions while the meta-described application is running. Thus we extend Smalltalk to let programmers interact with meta-descriptions at runtime. Programmers can query property values and operation results at runtime using the Object Inspector.

The *Object Inspector* is a cornerstone of any Smalltalk IDE, it presents the internal state of an object and offers to navigate the object structure. An inspector window is divided into two panes, see Fig. 7, the left pane enumerates all attributes of the inspected object, the right pane shows the value of the attribute selected on the left. When double clicking on an attribute, the inspector "dives" into the attribute, i.e., the value of the attribute becomes the new focus of the inspector. Furthermore, as Smalltalk is a living system it is possible to change the internal state of an object by typing an expression into the right pane and evaluating it.

However, relying on the built-in reflection of Smalltalk, the inspector is restricted to present the Smalltalk view of objects. We extended the inspector to also provide the EMOF view of objects, showing all meta-described properties on the left and using the access strategies to read and write values presented in the right pane. Thanks to the inspector's extension mechanism and the reflective nature of meta-description, we are able to extend the IDE with convenient access on the meta-layers.

6 MOOSE: a dedicated meta-described environment

Now we briefly present how the meta-model infrastructure is used in the particular context of our reengineering environment. 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 these extensions. To represent source code of different languages we defined a specific code-oriented meta-model named FAMIX [18,12]

To be able to communicate with third parties tools we provided generic import/export. We started with supporting the CDIF format, later we replaced it with XMI [41] and eventually moved to MSE (our own compact format). 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 code his domain classes and describe storable elements. Based on this, the objects in the model can be serialized in either XMI or MSE.

The act of analyzing can be decomposed in several generic atomic actions: (i) introspection—given an element, what are
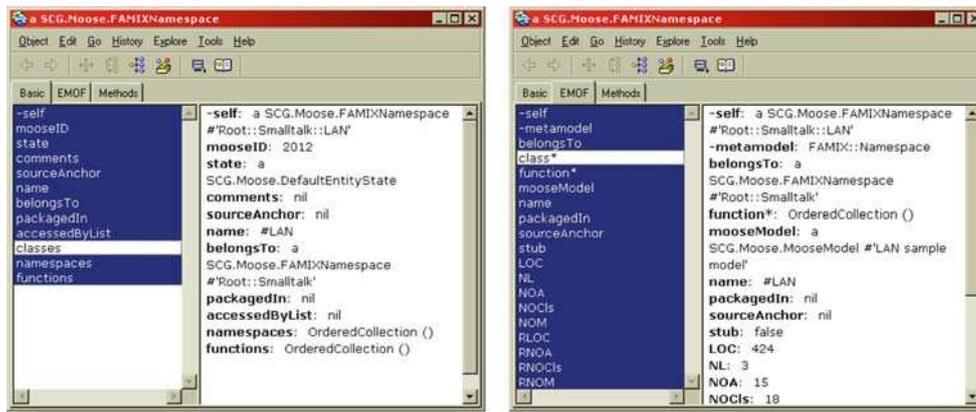
Fig. 7 Two screenshots of the same inspector window: on the left showing Smalltalk's view of an object, on the right showing the EMOF view. Note that the EMOF view includes more attributes, as some of these attributes are derived ones

its properties? (ii) selection—given a collection of elements, which are the elements matching a certain rule? (iii) navigation—given an element, what are its related elements? and (iv) presentation—given a collection of elements, what is the order of the elements? Also, 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 of the type of data. Again, we accomplished this by making the tools dependent only on the meta-descriptions [15].

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 types of data. As a consequence, we have implemented a mechanism for integrating these techniques. Our solution was to extend EMOF.Property and EMOF.Operation with a UI-displayable string. Using this annotation we can build a menu, and different tools can register themselves to the context they can handle.

Figure 8 shows the different extensions we performed on EMOF as well as one application in building a generic browser. An operation may represents a particular *Action* that can be triggered on a certain type of element. It extends EMOF.Operation with the title of the action and a category that is used to construct submenus. Also an operation may represent a boolean *Expression*, which is how search filter rules are called in MOOSE, and properties may represent a *Metric* which is a description for a measurement. Both the invocation of the actions and the computation of the metrics and of the expressions is done via the mechanisms specified in the EMOF standard.

Figure 8 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
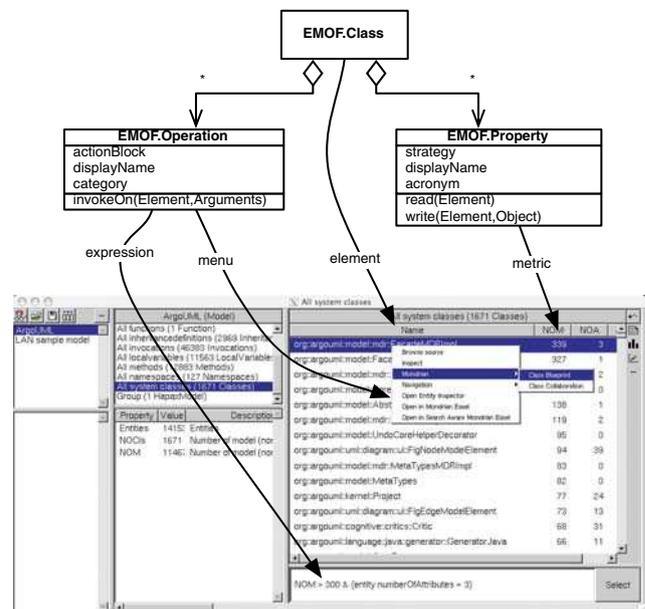
Fig. 8 We extended EMOF with new entities and new methods to hook in the execution. The MOOSE Browser is a generic tool based on the meta-descriptions

seen by the user. For example, by selecting an element we can trigger its menu which is composed of actions. In the figure, we selected a FAMIXClass and in its menu we have a Mondrian submenu. Mondrian is a visualization engine for scripting visualizations base on a graph model [33], and one visualization defined in Mondrian is the Class Blueprint [16], and it can be applied on any class through the contextual menu. The code below shows the method that Mondrian extends the FAMIXClass to spawn the Class Blueprint. Note that the method below is packaged in the Mondrian package, using class extension, and not in MOOSE where the class FAMIXClass is defined. Like this, we can trigger the menu action *only* when Mondrian is loaded.
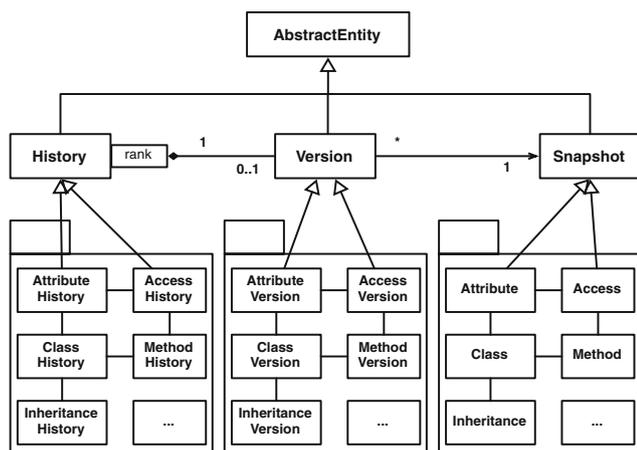
Fig. 9 An excerpt from the HISMO meta-model

```
FAMIXClass>>viewClassBlueprint
   <action: 'Class Blueprint' category: 'Mondrian'>
   | view |
   view := ViewRenderer new.
   view open
```

The described mechanisms in MOOSE allowed for definition and manipulation of several meta-models. Some of these are:

- FAMIX is a language-independent meta-model for representing procedural and object-oriented code [18].
- HISMO is a generic meta-model for analyzing software evolution based on representing history as first class entity (see Fig. 9) [20].
- DYNAMIX offers support for dynamic analysis by extending the FAMIX meta-model with the notions of trace and features [22].

7 Evaluation and discussion

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 preserve the one-to-one correspondence to the meta-descriptions while still be able to extend the entities with new capabilities given by orthogonal tools. For example, we can package the visualization specific methods and their meta-descriptions together with the visualization engine, rather than have these methods in the base code. In this way, we obtain loose tool inter-depencies rather than a hard-coded one.

As the meta-description are retained at runtime of the application, programmers can benefit from generic UI and IO components. Instead of generating custom code for each meta-described object's UI, we rather have one generic component that creates on-the-fly a dynamic UI based on the meta-descriptions of any object. The same goes for serialization.

7.1 Language features necessary to develop an executable meta-environment

When developing FAME we identified the following language features as most useful for implementing an executable meta-evironment:

- **Reflection**: The ability to introspect on objects, including the possibility to create instances by class name and to perform methods that are not known at compile time. This is a must have to achieve executability.
  For example the serializer, during import it must be able to create new instances by name and to call accessors based on property descriptions. The same for the runtime-generated UI, it must read and store the value of properties based on their description.
- **Annotations**: Declarative annotations are necessary to ensure the extensibility of the system. This is nice to have to guarantee that meta-descriptions stay in sync when applying code manipulation tools (e.g., refactorings). However, already in our environment, we have to live with a workaround for classes, as Smalltalk does not support class annotations. Relying on naming conventions could be a way when annotations are not available, but is less flexible (Note that JavaBeans relies on naming

convention rather than annotations to guarantee the same issue).

- **Closures**: Closure or the ability to define code on the fly whose execution can be delayed is important in particular to be able to specify mapping between the levels. Closures are used to express how domain objects are accessed and how such information is composed to define semantics at the EMOF level.
- **Class extensions**: Class extensions allow the domain code to be extended with behavior and annotations by third parties. This is a nice feature to achieve package merging. Using class extensions we get the merging semantics of EMOF for free: the plugins of Moose, for example, use class extensions to merge additions into the core packages of the FAMIX metamodel.

7.2 Code-generation versus runtime meta-interpretation

As we show in Sect. 6, the MOOSE UI is completely driven by the meta-descriptions of the entities MOOSE presents to the user. In particular, the meta-descriptions are used to control the navigation, menus and the information displayed. Furthermore, meta-descriptions are used for generic import/ export.

Most model-driven approaches rely on code generation. The meta-model is used to generate not only domain classes, but also to generate custom UI code and data access objects (DAO) or serialization code. An important feature of our system is that we prefer to avoid code generation (e.g., for custom UI and serialization). Instead we rely on generic implementations that interpret the meta-descriptions at runtime.

If developers use a traditional modeling tool, they can obtain a skeleton of program automatically generated by the tool from a given model. Using our approach developers do not use a modeling tool, but they focus on writing the Smalltalk code and then add the meta-description to benefit from the generic mechanisms. The meta-descriptions are also added as code, which means that both the code and the meta-description are developed using the Smalltalk tools.

Note that EMF which favors code generation also associates meta-description with the generated classes. However, such descriptions which are called metaclasses in EMF jargon, do not seem to be used by the run-time environment once the code is generated.

While for most activities we rely on runtime interpretation, when do use code generation in the first stage of implementing a foreign meta-model (as described in Sect. 5.2).

7.3 Discussion about causal connection

With our approach we bridge two worlds, and as we discuss above, this satisfies our goals. However, we want to stress

that our current implementation is not reflective in the sense that descriptions are not causally connected [30] to the other two parts of the meta-triangle, object and class. This has an implication: the correspondence between the base domain and the meta-model has to be manually done by attaching meta-descriptions to the domain classes. This limit can be circumvented when we deal with foreign meta-models that we load in our environment (as presented in Sect. 5.2) because we can generate code that is based on the meta-descriptions.

However, causal connection can be done as follows: The reflective features of Smalltalk allow to a running program to change, add and remove parts of itself at runtime. Using this, an executive meta-environment can change the code of the running program whenever the metamodel changes and thus establish full causality. An implementation of the meta-environment can even rely on the predefined refactorings of the RefactoryBrowser [37] to do so. This is different from purely introspective reflection, as for example given in Java, where the structure of running programs must not be changed and thus causality is difficult to establish.

Still, even given that, when describing EMOF in itself we cannot ensure the causal connection. Let us illustrate this point. When we describe the class EMOF.Class we declare that it has the property ownedAttributes by defining the class method metamodelOwnedAttributes which represents the property.

```
EMOF.Class class>>metamodelOwnedAttributes
    ^EMOF.Property new
        name: #ownedAttribute;
        type: EMOF.Property description;
        oppositeName: #class;
        upper: Unlimited positive;
        isOrdered: true
```

Smalltalk EMOF.Class class has an instance variable ownedAttributes that represents the attributes. Methods then can access and describe the behavior of the EMOF.Class. For example, the following code shows an accessor and an iterator over all attributes:

```
EMOF.Class>>ownedAttributes
    "ownedAttribute : Property 0..* -- The attributes owned by a
    class. These do not include the inherited attributes.
    Attributes are represented by instances of
    Property. [UML Infrastructure 2.0, p93]"
    ^ownedAttributes ifNil: [ ownedAttributes := Set new ]

EMOF.Class>>allAttributesDo: aBlock
    self ownedAttributes do: aBlock.
    self superClasses do: [ :superClass | superClass
    allAttributesDo: aBlock ]
```

We see here that the correspondence between the meta-description and the implementation that they describe has to be maintained manually (even if some behavior can be

automated by querying and manipulating the meta-descriptions). This is normal for two reasons. First, it is difficult to generate the Smalltalk code for the EMOF kernel from the EMOF descriptions, since in order to do so, instances of the EMOF classes are needed. Second, even if we would use another representation to represent temporarily the EMOF descriptions and generate the code and their respective EMOF descriptions from this other representation, we could not generate all the behavior but only the generic behavior that is linked with structural code navigation and querying.

7.4 Tradeoffs

Our approach is not strictly EMOF compliant as the MOF standard does not describe execution. (Note that the same situation arises with EMF since ECore meta-models are by nature not compliant to MOF.) 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 [35], or a dedicated language such as Kermeta [32]. However, since Smalltalk also offers a reflective API, we developed some simple meta-model transformations based on the Refactoring Browser.

An alternative would have been to extend the language or the Smalltalk meta-model itself, but from a practical point of view, extending a language is expensive. We want to reuse existing tools that work at the level of the current Smalltalk meta-model. For example, we want to store the meta-level code in our default storage database, which uses a fixed schema for example. Such limitations are to be found in any language's development environment.

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. [32] 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 …*)."*

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 to 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.

7.5 Related work and applicability

While we reported our experience using Smalltalk as an executable meta-language, our approach can be applied to other environments or languages with different levels of integration in the host language. Before highlighting some of the key features that will help reproducing our approach we discuss some related work.

Our approach is close to EMF [7]: both approaches have an implementation of a MOF kernel in a generalist language. The main difference between our approach and the one used in EMF is that we manipulate EMOF entities at run-time via meta-interpreters while EMF traditional use is geared towards code generation.

In our approach, EMOF entities are attached with the domain code they describe. Generating code can be more powerful when there is not a simple one to one mapping between the described entities and the entities that are generated. Interestingly the EMF code generation attaches the MOF entities to the generated class: The EObjectImpl class defines the method eclass() which returns the meta-entity (called in EMF jargon the metaclass of the class and which corresponds to our meta-description). However to the best of our knowledge, the run-time environment executing the generated code does not make use of the so called metaclasses.

MDR (Metadata Repository) implements the MOF, XMI and JMI standards for the Java NetBeans platform [31]. MDR uses an event notification mechanism to integrate the meta-model into the NetBeans IDE, similar to our approach using annotations. MDR supports multiple inheritance at the model level by generating and combining interfaces default behavior. Contrary to other MOF implementations, MDR uses automatic run-time code generation. All JMI methods that do not require custom implementation are implemented during the runtime by dynamically generating bytecode. This has the advantage that code generation is delayed until it is actually used, but also it also freezes the meta-model as soon as the code is generated. Our system allows one to easily change and adapt the meta-model at run-time, even after it has been instantiated.

The alternative to get executable metamodels are Xactium [8] and Kermeta [32]. These two executable meta-languages define a new language whose core is close to the EMOF one. Kermeta which is an implementation of EMOF in addition to the traditional class, method, inheritance model supports multiple inheritance and relationships. The pros of such an approach is a minimal and dedicated language to specify meta-level operations. The cons is that all the tools have to be redeveloped or adapted. In addition, using our approach we get a minimal model and run-time semantics of Smalltalk plus all the available tools.

Johnson et al. propose the type object design pattern [25] to separate the object from its type. In the meta-triangle Fig. 3

we presented two different applications (and a derived one) of the type object: (1) the *instance of* relationship represents the classic paradigm of object oriented programming with instances and classes, whereas (2) the *described by* relationship decouples the object from its class by supplementing it with an exchangeable type.

8 Conclusion

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.

We show how a 3-layer architecture can be introduced in a reflective language, validating the distinction between instantiation and representation links in meta-modeling tools architectures [2,5]. We believe that our approach can be applied in other mainstream programming languages. 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 reuse the tools provided by the development environment: browser, debugger, versioning, 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 coin this approach "literate meta-programming" [28].

As future work we want to extend Fame towards a scoped environment where multiple meta-models may co-exist at the same time in the same environment, thus offering alternative descriptions on the same data.

Table 1 Syntactical elements of Smalltalk

.	a period separates statements.
;	a cascade is used to send multiple messages to the same receiver.
:=	assignment.
^	returns statement; exit the method where it appears.
'a string'	single quotes delimit strings.
[statements]	square brackets delimit lexical closures (AKA block).
[:x :y | 2 * y * x]	closures can take arguments.
| tmpVar1 tmpVar2 |	temporary variables are declared using pipes.
"commented"	comments.
$a	the character a.
#symb	a symbol which is a unique string.
#(a b)	an array containing a and b.
{ 1+2 . 4}	a dynamic array containing two elements: 3 and 4.
true, false	booleans.
nil	the undefined value.
self	the current receiver of a message. Same semantics as this in Java.
super	to invoke overridden method. Same semantics as in Java.
thisContext	to access execution stack.

Appendix A: Syntax

Here we give the complete Smalltalk syntax. Table 1 shows the different syntactical elements of Smalltalk. There are three kinds of messages: unary, binary and keyword messages.

Unary. Messages that do not take any arguments. Example: aCollection removeAll.

Binary. Messages that take one argument and whose name is among (+ * @ , ...)

1 + 3. "Hello ", "world !"

Keyword-based. Messages that can take multiple parameters. Keywords are composed of token ended by a semi column. Each argument are written after each of the token.

```
aCollection replaceFrom: 1 to: 6 with: anOtherCollection.
"Equivalent in Java to:" aCollection.replaceFromToWith(1, 6,
anOtherCollection);
```

Message sending priority. Messages have different priority: unary are executed first, then binary, and finally keyword-based. For example:

5 factorial + 5 gcd: 5   should be read as: ((5 factorial) + 5) gcd: 5

Thus mathematical operation order is not preserved. So 3 + 4 * 3 is equal to 21 and not 15. You should write 3 + (4 * 3) if you want to give priority to the multiplication.

Cascades. You also can send several messages to the same object. To do so, you use the ; construct.

```
aCollection
    add: anObject;
    add: anOtherObject;
    add: aThirdObject.
```

Closure application. A closure is executed using messages value, value:, ...

```
[:x :y | 2 * y * x ] value: 3 value: 100
returns 600
```

References

1. Atkinson, C., Kuehne, T.: The essence of multilevel metamodeling. In: Proceedings of the UML Conference. LNCS, vol. 2185, pp. 19–33 (2001)
2. Atkinson, C., Kuehne, T.: Concepts for comparing modeling tool architecture. In: Proceedings of the UML Conference. LNCS, vol. 3713, pp. 19–33 (2005)
3. Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S., Kiczales, G., Moon, D.A.: Common lisp object system specification, x3j13. Technical Report 88-003, (ANSI COMMON LISP) (1988)
4. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: Controlling the scope of change in Java. In: Proceedings of 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), pp. 177–189. ACM Press, New York (2005)
5. Bézivin, J., Gerbé, O.: Towards a precise definition of the OMG/MDA framework. In: Proceedings Automated Software Engineering (ASE 2001), pp. 273–282. IEEE Computer Society, Los Alamitos CA (2001)
6. Bobrow, D.G., Gabriel, R.P., White, J.L. : CLOS in context—the CLOS in context—the shape of the design. In: Paepcke, A. (ed.) Object-Oriented Programming: The CLOS Perspective, pp. 29–61. MIT Press, Cambridge (1993)
7. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Addison Wesley Professional, Reading (2003)
8. Clark, T., Evans, A., Sammut, P., Willans, J.: Applied Metamodelling: A Foundation for Language Driven Development (2004)
9. Ducasse, S., Demeyer, S. (eds.): The FAMOOS Object-Oriented Reengineering Handbook. University of Bern, Switzerland (1999)
10. Demeyer, S., Ducasse, S., Lanza, M.: A hybrid reverse engineering platform combining metrics and program visualization. In: Balmas, F., Blaha, M., Rugaber, S. (eds.) Proceedings of 6th Working Conference on Reverse Engineering (WCRE'99). IEEE Computer Society (1999)
11. Demeyer, S., Ducasse, S., Nierstrasz, O.: Object-Oriented Reengineering Patterns. Morgan Kaufmann, Los Altos (2002)
12. Demeyer, S., Ducasse, S., Tichelaar, S.: Why unified is not universal. UML shortcomings for coping with round-trip engineering. In: Rumpe, B. (ed.) Proceedings UML'99 (The Second International Conference on the Unified Modeling Language). LNCS, vol. 1723, pp. 630–644, Kaiserslautern, Germany. Springer, Heidelberg (1999)
13. DeMichiel, L.G., Gabriel, R.P.: The common lisp object system: an overview. In: Bézivin, J., Hullot, J.-M., Cointe, P., Lieberman, H. (eds.) Proceedings ECOOP '87 of LNCS, vol 276, pp. 151–170. Springer, Paris (1987)
14. Ducasse, S., Gîrba, T.: Using Smalltalk as a reflective executable meta-language. In: International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006). LNCS, vol. 4199, pp. 604–618. Springer, Berlin (2006)
15. Ducasse, S., Gîrba, T., Lanza, M., Demeyer, S.: Moose: a collaborative and extensible reengineering environment. In: Tools for Software Maintenance and Reengineering, RCOST/Software Technology Series, pp. 55–71. Franco Angeli, Milano (2005)
16. Ducasse, S., Lanza, M.: The class blueprint: visually the class blueprint: visually supporting the understanding of classes. Trans. Softw. Eng. (TSE) **31**(1), 75–90 (2005)
17. Devos, M., Tilman, M.: Incremental development of a repository-based framework supporting organizational inquiry and learning. In: OOPSLA'98 Practioner's Report (1998)
18. Demeyer, S., Tichelaar, S., Ducasse, S.: FAMIX 2.1—The FAMOOS Information Exchange Model. Technical Report. University of Bern, Switzerland (2001)
19. Fleurey, F.: Langage et méthode pour une ingénierie des modèles fiable. Ph.D. Thesis, Thèse de doctorat, Université de Rennes 1 (2006)
20. Gîrba, T., Ducasse, S.: Modeling history to analyze software evolution. J. Softw. Maintenance Res. Practice (JSME) **18**, 207–236 (2006)
21. Goldberg, A., Robson, D.: Smalltalk 80: the Language and its Implementation. Addison Wesley, Reading (1983)
22. Greevy, O.: Enriching Reverse Engineering with Feature Analysis. Ph.D. Thesis, University of Berne (2007)
23. Object Management Group. Meta object facility (MOF) 2.0 core final adopted specification. Technical Report, Object Management Group (2004)
24. Jouault, F., Bézivin, J.: KM3: a DSL for metamodel specification. In: IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems. LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006)
25. Johnson, R., Wolf, B.: Type object. In: Martin, R.C., Riehle, D., Buschmann, F. (eds.) Pattern Languages of Program Design, vol. 3, chap. 4. Addison Wesley, Reading (1998). ISBN:0-201-31011-2
26. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press, Cambridge (1991)
27. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of Aspect J. In: Proceeding ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
28. Knuth, D.E.: Literate Programming. Center for the Study of Language and Information, Stanford (1992)
29. Maes, P.: Computational reflection. Ph.D. Thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels (1987)
30. Maes, P.: Concepts and experiments in computational reflection. In: Proceedings OOPSLA'87, ACM SIGPLAN Notices, vol. 22, pp. 147–155 (1987)

31. Matula, M.: Netbeans metadata repository. Technical Report, NetBeans (2003)
32. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving executability into object-oriented meta-languages. In: Kent, S., Briand, L. (eds.) Proceedings of MODELS/UML'2005. LNCS, vol. 3713, pp. 264–278, Montego Bay, Jamaica. Springer, Heidelberg (2005)
33. Meyer, M., Gîrba, T., Lungu, M.: Mondrian: An agile visualization framework. In: ACM Symposium on Software Visualization (SoftVis 2006), pp. 135–144. ACM Press, New York (2006)
34. Muller, P.-A., Studer, P., Fondement, F., Bézivin, J.: Independent web application modeling and development with netsilon. Softw. Syst. Model. **4**(4), 424–442 (2005)
35. Mellor, S.J., Tockey, S., Arthaud, R., LeBlanc, P.: Software-platform-independent, precise action specifications for UML. In: Bézivin, J., Muller, P.-A. (eds.) The Unified Modeling Language, UML'98—Beyond the Notation. First International Workshop, Mulhouse, France. LNCS, vol. 1618, pp. 281–286 (1998)
36. Nierstrasz, O., Ducasse, S., Gîrba, T.: The story of Moose: an agile reengineering environment. In: Proceedings of the European Software Engineering Conference (ESEC/FSE 2005), pp. 1–10. ACM Press, New York (2005). Invited paper
37. Roberts, D., Brant, J., Johnson, R.E.: A refactoring tool for Smalltalk. Theory Pract. Object Syst. (TAPOS) **3**(4), 253–263 (1997)
38. Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N.: The architecture of a uml virtual machine. In: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01), pp. 327–341 (2001)
39. Rivard, F.: Pour un lien d'instanciation dynamique dans les langages à classes. In: JFLA96. INRIA—collection didactique (1996)
40. Riehle, D., Tilman, M., Johnson, R.: Dynamic object model. In: Pattern Languages of Program Design, vol. 5. Addison-Wesley, Reading (2005)
41. Tichelaar, S., Ducasse, S., Demeyer, S.: FAMIX: Exchange experiences with CDIF and XMI. In: Proceedings of the ICSE 2000 Workshop on Standard Exchange Format (WoSEF 2000) (2000)
42. Yoder, J.W., Johnson, R.: The adaptive object model architectural style. In: Proceeding of the working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02) (2002)

Author's Biography

Tudor Girba attained the Ph.D. degree in 2005, and since then he is working as senior researcher at the Software Composition Group, University of Berne, Switzerland. His interests lie in the area of software engineering with focus on software understanding. He is one of the main architects and developers of the Moose reengineering, and he participated in the development of several other reverse engineering tools and models. He authored more than 40 technical papers and he participated in more than 10 program committees of international conferences and workshops. He is the president of the Moose Association and is member in the Executive Board of CHOOSE (the Swiss Object-Oriented Software Engineering society). He offers consulting services in the area of software engineering, reengineering and quality assessment. He also blogs on the topic of presentation and modeling.

Adrian Kuhn is a doctoral candidate in computer science at University of Bern. He is the main architect and developer of Fame, the meta-modeling framework presented in this publication. His research interests include programming language design, software evolution, and tool building. He received his MSc in computer science from the University of Bern. Contact him at E-mail.: akuhn@iam.unibe.ch.

Stéphane Ducasse After spending 10 years at the Software Composition Group of the University of Bern, Stèphane Ducasse is senior researcher (Directeur de recherche) at INRIA. His fields of interests are: dynamic languages, reflective systems, reengineering of object-oriented applications, program visualization, modeling, maintenance. He is involved in the development of Squeak an open-source Smalltalk and he is the president of the European Smalltalk User Group. He can be reached at E-mail.: stephane.ducasse@inria.fr.

Lukas Renggli is a doctoral candidate in computer science at the University of Bern. His research interests include programming languages, software design, domain specific languages, meta-programming, and web application development. He received his MSc in computer science from the University of Bern. Contact him at Software Composition Group, Institut für Informatik, Neubrückstrasse 10, 3012 Bern, Switzerland, E-mail.: renggli@iam.unibe.ch.