Model Driven Language Engineering with Kermeta

Jean-Marc Jézéquel, Olivier Barais, Franck Fleurey

Model Driven Language Engineering with Kermeta

Jean-Marc Jézéquel, Olivier Barais, Franck Fleurey

INRIA & University of Rennes1
Campus Universitaire de Beaulieu
35042 Rennes CEDEX, France

Abstract. In many domains such as telecom, aerospace and automotive industries, engineers rely on Domain Specific Modeling Languages (DSML) to solve the complex issues of engineering safety critical software. Traditional Language Engineering starts with the grammar of a language to produce a variety of tools for processing programs expressed in this language. Recently however, many new languages tend to be first defined through metamodels, i.e. models describing their abstract syntax. Relying on well tooled standards such as E-MOF, this approach makes it possible to readily benefit from a set of tools such as reflexive editors, or XML serialization of models. This article aims at showing how Model Driven Engineering can easily complement these off-the-shelf tools to obtain a complete environment for such a language, including interpreter, compiler, pretty-printer and customizable editors. We illustrate the conceptual simplicity and elegance of this approach using the running example of the well known LOGO programming language, developed within the Kermeta environment.

1 Introduction

In many domains such as telecom, aerospace and automotive industries [21], engineers rely on Domain Specific Modeling Languages (DSML) to solve the complex issues of engineering safety critical software at the right level of abstraction. These DSMLs indeed define modeling constructs that are tailored to the specific needs of a particular domain. When such a new DSML is needed, it is now often first defined through meta-models, i.e. models describing its abstract syntax [14] when traditional language engineering would have started with the grammar of the language. Relying on well tooled standards such as E-MOF, the meta-modeling approach makes it possible to readily benefit from a set of tools such as reflexive editors, or XML serialization of models. More importantly, having such a tool supported *de facto* standard for defining models and meta-models paves the way towards a rich ecosystem of interoperable tools working seamlessly with these models and meta-models.

Combining this Model Driven approach with a traditional grammar based one has however produced mixed results in terms of the complexity of the overall approach. Several groups around the world are thus investigating the idea of a

new Language Engineering completely based on models [22], that we call Model Driven Language Engineering (MDLE).

In this paper we present one of these approaches, based on the Kernel Meta-Modeling environment Kermeta [16,7]. We start in Section 2 by giving a quick overview of executable meta-modeling, and then focusing on Kermeta, seen both as an aspect-oriented programming language as well as an integration platform for heterogeneous meta-modeling. We then recall in Section 3 how to model the abstract syntax of a language in E-MOF, allowing for a direct implementation of its meta-model in the Eclipse Modeling Framework (EMF). We then show how to weave both the static and dynamic semantics of the language into the meta-model using Kermeta to get an interpreter for the language. Then we address compilation, which is just a special case of model transformation to a platform specific model [17,3]. We illustrate the conceptual simplicity and elegance of this approach using the running example of the well known Logo programming language, for which a complete programming environment is concretely outlined in this article, from the Logo meta-model to simulation to code generation for the Lego Mindstorm platform and execution of a Logo program by a Mindstorm turtle.

2 Executable Meta-Modeling

2.1 Introduction

Modeling is not just about expressing a solution at a higher abstraction level than code. This limited view on modeling has been useful in the past (assembly languages abstracting away from machine code, 3GL abstracting over assembly languages, etc.) and it is still useful today to get e.g.; a holistic view on a large C++ program. But modeling goes well beyond that.

In engineering, one wants to break down a complex system into as many models as needed in order to address all the relevant concerns in such a way that they become understandable enough. These models may be expressed with a general purpose modeling language such as the UML [26], or with Domain Specific Modeling Languages (DSML) when it is more appropriate. Each of these models can be seen as the abstraction of an aspect of reality for handling a given concern. The provision of effective means for handling such concerns makes it possible to establish critical trade-offs early on in the software life cycle.

Models have been used for long as *descriptive* artifacts, which was already extremely useful. In many cases we want to go beyond that, i.e. we want to be able to perform computations on models, for example to simulate some behavior [16], or to generate code or tests out of them [19]. This requires that models are no longer informal, and that the language used to describe them has a well defined abstract syntax (called its meta-model) and semantics.

Relying on well tooled Eclipse standards such as E-MOF to describe these meta-models, we can readily benefit from a set of tools such as reflexive editors, or XML serialization of models, and also from a standard way of accessing models

from Java. The rest of this section introduces Kermeta, a Kernel Meta-Modeling language and environment, whose goal is to complement Eclipse off-the-shelf tools to obtain a complete environment for such DSMLs, including interpreters, compilers, pretty-printers and customizable editors.

2.2 Kermeta as a MOF extension

Kermeta is a Model Driven Engineering platform for building rich development environments around meta-models using an aspect-oriented paradigm [16,10]. Kermeta has been designed to easily extend meta-models with many different concerns (such as static semantics, dynamic semantics, model transformations, connection to concrete syntax, etc.) expressed in heterogeneous languages. A meta-language such as the Meta Object Facility (MOF) standard [18] indeed already supports an object-oriented definition of meta-models in terms of packages, classes, properties and operation signatures, as well as model-specific constructions such as containments and associations between classes. MOF does not include however concepts for the definition of constraints or operational semantics (operations in MOF do jot contain bodies). Kermeta can thus be seen as an extension of MOF with an imperative action language for specifying constraints and operation bodies at the meta-model level.

The action language of Kermeta is especially designed to process models. It is imperative and includes classical control structures such as blocks, conditional and loops. Since the MOF specifies object-oriented structures (classes, properties and operations), Kermeta implements traditional object-oriented mechanisms for multiple inheritance and behavior redefinition with a late binding semantics (to avoid multiple inheritance conflicts a simple behaviors selection mechanism is available in Kermeta). Like most modern object-oriented languages, Kermeta is statically typed, with generics and also provides reflection and an exception handling mechanism.

In addition to object-oriented structures, the MOF contains model-specific constructions such as containment and associations. These elements require a specific semantics of the action languages in order to maintain the integrity of associations and containment relations. For example, in Kermeta, the assignment of a property must handle the other end of the association if the property is part of an association and the object containers if the property is a composition.

Kermeta expressions are very similar to Object Constraint Language (OCL) expressions. In particular, Kermeta includes lexical closures similar to OCL iterators on collections such as *each, collect, select* or *detect*. The standard framework of Kermeta also includes all the operations defined in the OCL standard framework. This alignment between Kermeta and OCL allows OCL constraints to be directly imported and evaluated in Kermeta. Pre-conditions and post-conditions can be defined for operations and invariants can be defined for classes. The Kermeta virtual machine has a specific execution mode, which monitors these contracts and reports any violation.

2.3 Kermeta as an Aspect-Oriented Integration Platform

Since Kermeta is an extension of MOF, a MOF meta-model can conversely be seen as a valid Kermeta program that just declares packages, classes and so on but *does* nothing. Kermeta can then be used to *breath life* into this meta-model by incrementally introducing aspects for handling concerns of static semantics, dynamic semantics, or model transformations [17].

One of the key features of Kermeta is the static composition operator "*require*", which allows extending an existing meta-model with new elements such as properties, operations, constraints or classes. This operator allows defining these various aspects in separate units and integrating them automatically into the meta-model. The composition is done statically and the composed model is typed-checked to ensure the safe integration of all units. This mechanism makes it easy to reuse existing meta-models or to split meta-models into reusable pieces. It can be compared to the open class paradigm [4]. Consequently a meta-class that identifies a domain concept can be extended without editing the meta-model directly. Open classes in Kermeta are used to organize "cross-cutting" concerns separately from the meta-model to which they belong, a key feature of aspect-oriented programming [11]. With this mechanism, Kermeta can support the addition of new meta-class, new subclasses, new methods, new properties, new contracts to existing meta-model. The *require* mechanism also provides flexibility. For example, several operational semantics could be defined in separate units for a single meta-model and then alternatively composed depending on particular needs. This is the case for instance in the UML meta-model when several semantics variation points are defined.

Thank to this composition operator, Kermeta can remain a kernel platform to safely integrate all the concerns around a meta-model. As detailed in the previous paragraphs, meta-models can be expressed in MOF and constraints in OCL. Kermeta also allows importing Java classes in order to use services such as file input/output or network communications during a transformation or a simulation. These functionalities are not available in the Kermeta standard framework. Kermeta and its framework remain dedicated to model processing but provide an easy integration with other languages. This is very useful for instance to make models communicating with existing Java applications.

3 Building an integrated environement for the Logo Language

3.1 Meta-Modeling Logo

To illustrate the approach proposed in this paper, we use the example of the Logo language. This example was chosen because Logo is a simple yet real (i.e. Turing-complete) programming language, originally created for educational purposes. Its most popular application is *turtle graphics*: the program is used to direct a virtual turtle on a board and make it draw geometric figures when its pen

is down[1]. Figure 1 presents a sample Logo program which draws a square. In this paper we propose to build a complete Logo environment using model-driven engineering techniques.

```
1  # definition of the square procedure
2  TO square :size
3    REPEAT 4 [
4      FORWARD :size
5      RIGHT 90
6    ]
7  END
8
9  # clear screen
10 CLEAR
11
12 # draw a square
13 PENDOWN
14 square(50)
15 PENUP
```

Fig. 1. Logo square program

The first task in the model driven construction of a language is the definition of its abstract syntax. The abstract syntax captures the concepts of the language (these are primitive instructions, expressions, control structures, procedure definitions, etc.) and the relations among them (e.g. an expression is either a constant or a binary expression, that itself contains two expressions). In our approach the abstract syntax is defined using a meta-model.

Figure 2 presents the meta-model for the abstract syntax of the Logo language. The Logo meta-model includes:

- Primitive statements (*Forward*, *Back*, *Left*, *Right*, *PenUp* and *PenDown*). These statement allows moving and turning the Logo turtle and controlling its pen.
- Arithmetic Expressions (*Constant*, *BinaryExp* and its sub-classes). In our version of Logo, constants are integers and all operators only deal with integers.
- Procedures (*ProcDeclaration*, *ProcCall*, *Parameter* and *ParameterCall*) allow defining and calling functions with parameters (note that recursion is supported in Logo).
- Control Structures (*Block*, *If*, *Repeat* and *While*). Classical sequence, conditional and loops for an imperative language.

[1] A complete history of the Logo language and many code samples can be found on wikipedia (http://en.wikipedia.org/wiki/Logo_(programming_language))

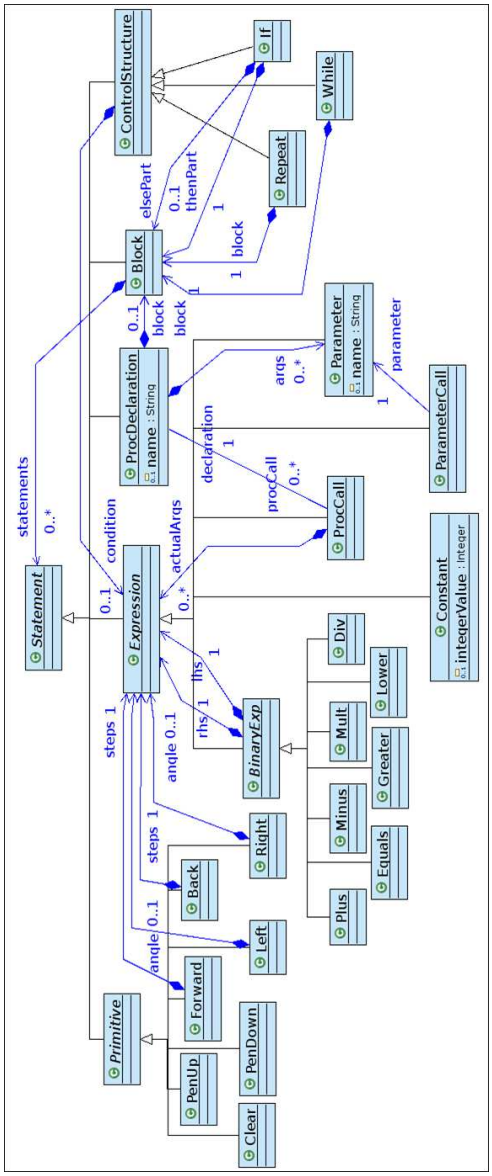

Fig. 2. Logo Abstract Syntax.

In practice the Logo meta-model can be defined within the Eclipse Modeling Framework (EMF). EMF is a meta-modeling environment built on top of the Eclipse platform and based on the Essential-MOF standard. Within Eclipse several graphical editors can be used to define such meta-models. Once the meta-model is defined, the EMF automatically provides editors and serialization capabilities for the meta-model. The editor allows creating instances of the classes of the meta-model and saving these instances models in the XMI standard format.

As soon as the meta-model of Fig. 2 has been defined, it is possible to instantiate it using the generated editor in order to write Logo programs. Figure 3 presents a screen-shot of the generated editor with the square program presented previously. The program was defined in the tree editor and the right part of the figure shows how the logo program was serialized.

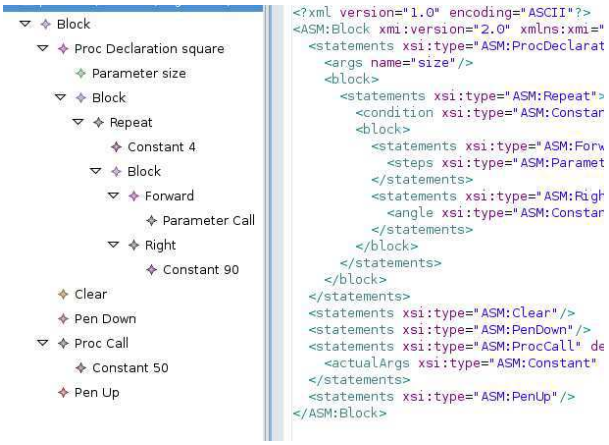

Fig. 3. Logo square program in the generated editor and serialized in XMI.

3.2 Weaving static semantics

The Object Constraint Language A meta-model can be seen as the definition of the set of allowed configurations for a set of objects representing a domain. All structures are represented as classes, relations and structural properties. In MDLE, a meta-model defines a set of valid programs. However, some constraints (*formulas* to the logician, *Boolean expressions* to the programmer) cannot directly be expressed using EMOF. For example there is no easy way to express that formal parameter names should be unique in a given procedure declaration, or that in a valid Logo program the number of actual arguments in a procedure call should be the same as the number of formal arguments in the declaration. This kind of constraints forms part of what is often called the static semantics of the language.

In Model-Driven Engineering, the Object Constraint Language (OCL) [20] is often used to provide a simple first order logic for the expression of the static semantics of a meta-model. OCL is a declarative language initially developed at IBM for describing constraints on UML models. It is a simple text language that provides constraints and object query expressions on any Meta-Object Facility model or meta-model that cannot easily be expressed by diagrammatic notation. OCL language statements are constructed using the following elements:

1. a context that defines the limited situation in which the statement is valid
2. a property that represents some characteristics of the context (e.g., if the context is a class, a property might be an attribute)
3. an operation (e.g., arithmetic, set-oriented) that manipulates or qualifies a property, and
4. keywords (e.g., if, then, else, and, or, not, implies) that are used to specify conditional expressions.

Expressing the Logo Static Semantics in OCL The Logo meta-model defined on figure 2 only defines the structure of a Logo program. To define the sub-set of programs which are valid with respect to Logo semantics a set of constraints has to be attached to the abstract syntax. Figure 4 presents the OCL listing of two constraints attached to the Logo meta-model. The first one is an invariant for class *ProcCall* that ensures that any call to a procedure has the same number of actual arguments as the number of formal arguments in the procedure declaration. The second invariant is attached to class *ProcDeclaration* and ensures that the names of the formal parameters of the procedure are unique.

Weaving the Logo Static Semantics into its Meta-Model In the Kermeta environment, the OCL constraints are woven directly into the meta-model and can be checked for any model which conforms to the Logo meta-model. In practice, once the designer has created a meta-model with the E-core formalism in the Eclipse Modeling Framework (called e.g. `ASMLogo.ecore`), she can import it into Kermeta (see line 2 in Figure 5) using the *require* instruction as described in Section 2. Suppose now that the Logo static semantics (of Fig. 4) is located

```
1  package kmLogo::ASM
2
3  context ProcCall
4  inv same_number_of_formals_and_actuals :
5    actualArgs->size() = declaration.args->size()
6
7  context ProcDeclaration
8  inv unique_names_for_formal_arguments :
9    args->forAll ( a1, a2 | a1.name = a2.name implies a1 =
       a2 )
10
11 endpackage
```

Fig. 4. OCL constraint on the Logo meta-model

in a file called `StaticSemantics.ocl`. Then the same *require* instruction can be used in Kermeta to import the Logo static semantics and weave it into the Logo meta-model (see line 3 in Figure 5).

```
1  package kmLogo;
2  require "ASMLogo.ecore"
3  require "StaticSemantics.ocl"
4  [...]
5  class Main {
6    operation Main(): Void is do
7    // Load a Logo program and check constraints on it
8    // then run it
9    end
10 end
```

Fig. 5. Weaving Static Semantics into the Logo Meta-Model

The integration of OCL into Kermeta relies onto two features:

- First, as presented in Section 3, Kermeta already supports a native constraint system made of invariants, pre and post conditions which makes it possible to work within a Design-by-Contracts methodology.
- Secondly, the support for the OCL concrete syntax is implemented with a model transformation from the AST of OCL to the AST of Kermeta. This transformation has been written in Kermeta. The result model of this transformation is automatically associated to the meta-model implicated, using the built-in static introduction of Kermeta.

Kermeta allows the user to choose when his constraints should be checked. That can be done class by class or at the entire model level with primitive *checkInvariant* on class or *checkAllInvariants* on the root element of the meta-model. The operation constraints (*pre*, *post*) are optionally checked depending on the type of "Run" chosen from the Eclipse menu: normal run or run with constraint checking.

So, at this stage the meta-model allows defining Logo programs with a model editor provided by the EMF and this model can be validated with respect to Logo static semantics within the Kermeta environment. For instance if we modify the Logo program of Fig. 1 by calling `square(50,10)` instead of `square(50)`, and if we load it into Kermeta, then by calling *checkAllInvariants* we get the expected error message that

```
Invariant same_number_of_formals_and_actuals
  has been violated for: square(50,10)
```

One point of interest is that this implementation extends the expressiveness of OCL. OCL already offers the possibility to call operations or derived properties declared in the meta-model. Kermeta allows the designer to specify the operational semantic of these methods or these properties. Then, using the OCL implementation in Kermeta, it is possible to express any expression based on the first-order logic and extend it with the imperative operations provided by Kermeta. Designer must of course still guarantee that these operations are free from side-effects on the abstract state of the models.

3.3 Weaving dynamic semantics to get an interpreter

The next step in the construction of a Logo environment is to define Logo operational semantics and to build an interpreter. In our approach this is done in two steps. The first one is to define the runtime model to support the execution of Logo programs, i.e. the Logo *virtual machine*. The second one is to define a mapping between the abstract syntax of Logo and this virtual machine. This is going to be implemented as a set of *eval* functions woven into the relevant constructs of the Logo meta-model.

Logo runtime model As discussed earlier, the most popular runtime model for Logo is a turtle which can draw segments with a pen. As for the language abstract syntax, the structure of the runtime model can be defined by a meta-model. The advantage of this approach is that the state of the running program is then also a model. Like for any model, all the tools available in a framework like EMF can then readily be used in order to observe, serialize, load or edit the state of the runtime.

Figure 6 presents a diagram of the Logo virtual machine meta-model. The meta-model only defines three classes: *Turtle*, *Point* and *Segment*. The state of the running program is modeled as a single instance of class *Turtle* which has a position (which is a *Point*), a heading (which is given in degrees) and

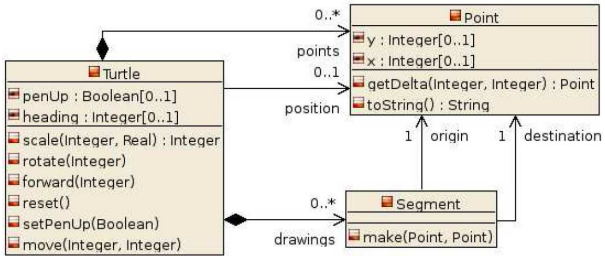

Fig. 6. Logo runtime meta-model.

a Boolean to represent whether the pen is up or down. The Turtle stores the segments which were drawn during the execution of the program. In practice the meta-model was defined without operation using EMF tools. The operations, implemented in Kermeta, have been later woven into the meta-model to provide an object-oriented definition of the Logo virtual machine. Figure 7 presents an excerpt of the Kermeta listing. It adds three operations to the class *Turtle* of the meta-model.

Operational semantics We are now going to define the operational semantics for each constructs of the abstract syntax. The idea is again to weave operations implemented in Kermeta directly into the meta-model in such a way that each type of statement would contain an *eval* operation performing the appropriate actions on the underlying runtime model. To do that, a context is provided as a parameter of the *eval* operation. This context contains an instance of the Turtle class of the runtime meta-model and a stack to handle procedure calls. Figure 8 presents how the operation *eval* are woven into the abstract syntax of Logo. An abstract operation *eval* is defined on class *Statement* and implemented in every sub-class to handle the execution of all constructions.

For simple cases such as the *PenDown* instruction, the mapping to the virtual machine is straightforward: it only boils down to calling the relevant VM instruction, i.e. `context.turtle.setPenUp(false)` (see line 36 of Fig. 8).

For more complex cases such as the *Plus* instruction, there are two possible choices. The first one, illustrated on lines 9–13 of Fig. 8, makes the assumption that the semantics of the Logo *Plus* can be directly mapped to the semantics of "+" in Kermeta. The interest of this first solution is that it provides a quick and straightforward way of defining the semantics of that kind of operators. If however the semantics we want for the Logo *Plus* is not the one that is built-in Kermeta for whatever reason (e.g. we want it to handle 8-bits integers only), we can define the wanted *Plus* operation semantics in the Logo Virtual Machine (still using Kermeta of course) and change the *eval* method of lines 9–13 so that it first calls *eval* on the left hand side, push the result on the VM stack, then

```
1  package kmLogo;
2
3  require "VMLogo.ecore"
4  [...]
5  package VM {
6    aspect class Turtle {
7      operation setPenUp(b : Boolean) is do
8        penUp := b
9      end
10     operation rotate(angle : Integer)  is do
11       heading := (heading + angle).mod(360)
12     end
13     operation forward(steps : Integer)  is do
14       var radian : Real init math.toRadians(heading.
             toReal)
15       move(scale(steps,math.sin(radian)), scale(steps,
             math.cos(radian)))
16     end
17     [...]
18   }
19 }
```

Fig. 7. Runtime model operations in Kermeta

calls *eval* on the right hand side, again push the result on the VM stack, and finally call the *Plus* operation on the VM.

Getting an Interpreter Once the operational semantics for Logo has been defined as described above, getting an interpreter is pretty straightforward: we first have to import each relevant aspect to be woven into the Logo meta-model (using *require* instructions, see lines 2–5 in Fig. 9). We then need to load the Logo program into Kermeta (see lines 9–12 in Fig. 9), instantiate a *Context* (that contains the Logo VM) and then call *eval(Context)* on the root element of the Logo program.

Loading the *Square* program of Fig. 1 and executing it this way will change the state of the model of the Logo VM: during the execution, four new *Segments* will be added to the *Turtle*, and its position and heading will change. Obviously, we would like to see this execution graphically on the screen. The solution is quite easy: we just need to put an Observer on the Logo VM to graphically display the resulting figure in a Java widget. The Observer is implemented in Kermeta and calls relevant Java methods to notify the screen that something has changed.

```
package kmLogo;
require "ASMLogo.ecore"
require "LogoVMSemantics.kmt"
[...]
package ASM {
  aspect class Statement {
    operation eval(context : Context) : Integer is
        abstract
  }
  aspect class Plus {
    method eval(context : Context) : Integer is do
      result := lhs.eval(context) + rhs.eval(context)
    end
  }
  aspect class Greater {
    method eval(context : Context) : Integer is do
      result := if lhs.eval(context) > rhs.eval(context)
          then 1 else 0 end
    end
  }
  aspect class If {
    method eval(context : Context) : Integer is do
      if condition.eval(context) != 0 then
        result := thenPart.eval(context)
      else
        result := elsePart.eval(context)
      end
    end
  }
  aspect class Forward {
    method eval(context : Context) : Integer is do
      context.turtle.forward(steps.eval(context))
      result := void
    end
  }
  aspect class PenDown {
    method eval(context : Context) : Integer is do
      context.turtle.setPenUp(false)
      result := void
    end
  }
  [...]
}
```

Fig. 8. Logo operational semantics

```
1  package kmLogo;
2  require "ASMLogo.ecore"
3  require "StaticSemantics.ocl"
4  require "LogoVMSemantics.kmt"
5  require "OperationalSemantics.kmt"
6  [...]
7  class Main {
8    operation Main(): Void is do
9      var rep : EMFRepository init EMFRepository.new
10     var logoProgram : ASMLogo::Block
11     // load logoProgram from its XMI file
12     logoProgram ?= rep.getResource("Square.xmi").one
13     // Create a new Context containing the Logo VM
14     var context : LogoVMSemantics::Context init
           LogoVMSemantics::Context.new
15     // now execute the logoProgram
16     logoProgram.eval(context)
17   end
18 end
```

Fig. 9. Getting an Interpreter

3.4 Compilation as a kind of Model Transformation

In this section we are going to outline how to build a compiler for our Logo language. The idea is to map a Logo program to the API offered by the Lego Mindstroms so that our Logo programs can actually be used to drive small robots mimicking Logo turtles. These *Robot-Turtles* are built with Lego Mindstroms and feature two motors for controlling wheels and a third one for controlling a pen (see Fig. 10).

A simple programming language for Lego Mindstroms is NXC (standing for Not eXactly C). So building a Logo compiler for Lego Mindstroms boils down to write a translator from Logo to NXC. The problem is thus much related to the Model Driven Architecture (MDA) context as promoted by the OMG, where a Logo program would play the role of a Platform Independent Model (PIM) while the NXC program would play the role of a Platform Specific Model (PSM). With this interpretation, the compilation we need is simply a kind of model transformation.

We can implement this model transformation either using Model-to-Model Transformations or Model-to-Text Transformations:

Model-to-Text Transformations are very useful for generating code, XML, HTML, or other documentation in a straightforward way, when the only thing that is needed is actually a syntactic level transcoding (e.g. Pretty-Printing). Then we can resort on either:

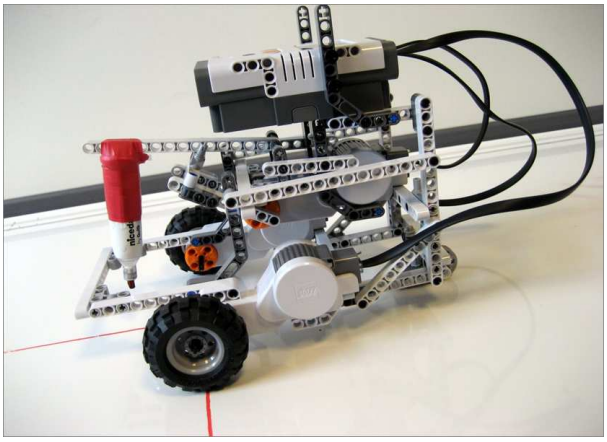

Fig. 10. A Lego mindstorm robot-turtles.

- Visitor-Based Approaches, where some visitor mechanisms are used to traverse the internal representation of a model and directly write code to a text stream.
- Template-Based Approaches, based on the target text containing slices of meta-code to access information from the source and to perform text selection and iterative expansion. The structure of a template resembles closely the text to be generated. Textual templates are independent of the target language and simplify the generation of any textual artifacts.

Model-to-Model Transformations would be used to handle more complex, semantic driven transformations.

For example if complex, multi-pass transformations would have been needed to translate Logo to NXC, it could have been interesting to have an explicit meta-model of NXC, properly implement the transformation between the Logo meta-model and the NXC one, and finally call a pretty-printer to output the NXC source code.

In our case however the translation is quite simple, so we can for example directly implement a visitor-based approach. In practice, we are once again going to use the aspect weaving mechanism of Kermeta simplify the introduction of the Visitor pattern. Instead of using the pair of methods *accept* and *visit*, where each *accept* method located in classes of the Logo meta-model would call back the relevant *visit* method of the visitor, we can directly weave a *compile()* method into each of these Logo meta-model classes (see Fig. 11).

Integrating this compilation aspect into our development environment for Logo is done as usual, i.e. by *requiring* it into the main Kermeta program (see Fig. 12).

```
package kmLogo;

require "ASMLogo.ecore"
[...]
package ASMLogo {
  aspect class PenUp {
    compile (ctx: Context) {
    [...]
    }
  }

  aspect class Clear {
    compile (ctx: Context) {
    [...]
    }
  }
[...]
}
```

Fig. 11. The Logo Compilation Aspect in Kermeta

```
package kmLogo;
require "ASMLogo.ecore"
require "StaticSemantics.ocl"
require "Compiler.kmt"
[...]
class Main {
  operation Main(): Void is do
    var rep : EMFRepository init EMFRepository.new
    var logoProgram : ASMLogo::Block
    // load logoProgram from its XMI file
    logoProgram ?= rep.getResource("Square.xmi").one
    // Create a new Context for storing global data
        during the compilation
    var context : Context init Context.new
    // now compile the logoProgram to NXC
    logoProgram.compile(context)
  end
end
```

Fig. 12. Getting a Compiler

3.5 Model to Model Transformation

For the Logo compiler described above to work properly, we have to assume though that all Logo function declarations are performed at the outermost block level, because NXC does not support nested function declarations. Since nothing in our Logo meta-model prevents the Logo user to declare nested functions, we need to either add an OCL constraint to the Well-Formedness Rules of the language, or we need to do some pre-processing before the actual compilation step. For the sake of illustrating Kermeta capabilities with respect to Model to Model Transformations, we are going to choose the later solution.

We thus need a new aspect in our development environment, that we call the *local-to-global* aspect (See Listing 13) by reference to an example taken from the TXL [5] tutorial. We are using a very simple OO design that declares an empty method *local2global* (taking as parameter the root block of a given Logo program) in the topmost class of the Logo meta-model hierarchy, *Statement*. We are then going to redefine it in relevant meta-model classes, such as *ProcDeclaration* where we have to move the current declaration to the root block and recursively call *local2global* on its block (containing the function body). Then in the class *Block*, the *local2global* method only has to iterate through each instruction and recursively call itself.

```
1  package kmLogo;
2
3  require "ASMLogo.ecore"
4  [...]
5  package ASMLogo {
6  aspect class Statement
7    method local2global(rootBlock: Block) is do
8    end
9  end
10 aspect class ProcDeclaration
11   method local2global(rootBlock: Block) is do
12     rootBlock.add(self)
13     block.local2global(rootBlock)
14   end
15 end
16 aspect class Block
17   method local2global(rootBlock: Block) is do
18     statements.each(i| i.local2global(rootBlock))
19    end
20 end
21 }
```

Fig. 13. The Logo *local-to-global* Aspect in Kermeta

Note that if we also allow *ProcDeclaration* inside control structure such as *Repeat* or *If*, we would also need to add a *local2global* method in these classes to visit their block (*thenPart* and *elsePart* in the case of the *If* statement).

Once again this *local2global* concern is implemented in a modular way in Kermeta, and can easily be added or removed from the Logo programming environment without any impact on the rest of the software. Further, new instructions could be added to Logo (i.e. by extending its meta-model with new classes) without much impact on the *local2global* concern as long as these instructions do not contain any block structure. This loose coupling is a good illustration of Kermeta advanced modularity features, allowing both easier parallel development and maintenance of a DSML environment.

4 Discussion

4.1 Separation of Concerns for language engineering

From an architectural point of view, Kermeta allows the language designer to keep his concerns separated. Designers of meta-models will typically work with several artifacts: the structure is expressed in the *Ecore* meta-model, the operational semantics is defined in a Kermeta resource, and finally the static semantics is brought in an OCL file. Consequently, as illustrated in Figure 9, a designer can create a meta-model defined with the Ecore formalism of the Eclipse Modeling Framework. He can define the static semantics with OCL constraints. Finally with Kermeta, he can define the operational semantics as well as some useful derived features of the meta-models that are called in the OCL specifications. The weaving of all those model fragments is performed automatically in Kermeta, using the *require* instruction as a concrete syntax for this static introduction. Consequently, in the context of the class Main, the meta-model contains the data-structure, the static semantics and the operational semantics.

4.2 Concrete Syntax issues

Meta-Modeling is a natural approach in the field of language engineering for defining abstract syntaxes. Defining concrete and graphical syntaxes with meta-models is still a challenge. Concrete syntaxes are traditionally expressed with rules, conforming to EBNF-like grammars, which can be processed by compiler compilers to generate parsers. Unfortunately, these generated parsers produce concrete syntax trees, leaving a gap with the abstract syntax defined by meta-models, and further ad hoc hand-coding is required. We have proposed in [15] a new kind of specification for concrete syntaxes, which takes advantage of meta-models to generate fully operational tools (such as parsers or text generators). The principle is to map abstract syntaxes to concrete syntaxes via bidirectional mapping-models with support for both model-to-text, and text-to-model transformations. Other tools emerge for solving this issue of defining the concrete syntax from a mapping with the abstract syntax like the new Textual Modeling

Framework[2]. To get usable graphical editors for your domain specific language (DSL), several projects provides a generative approach to create component and runtime infrastructure for building graphical editors as GMF or TopCaseD. We have used Sintaks and TopCaseD to respectively build the Logo concrete syntax, the Logo graphical syntax and their associated editors.

4.3 Evolution issues

Thanks to the separation of concerns, constraints and behavior aspects are independent and may be designed in separate resources. Then they can be developed and modified separately. The only consideration during their implementation is that they depend on the structure defined in the Ecore meta-model. Modification to this structure can have consequences that have to be considered in the behavior aspects and in the constraints (if a method signature is changed for example). Here, Kermeta's type system is useful as a way of detecting such incompatible changes at compile time.

5 Related works

There is a long tradition of basing language tools on grammar formalisms, for example higher order attribute grammars [25]. JastAdd [8] is an example of combining this tradition with object-orientation and simple aspect-orientation (static introductions) to get better modularity mecanisms. With a similar support for object-orientation and static introductions, Kermeta can then be seen as a symetric of JastAdd in the DSML world.

Kermeta cannot really be compared to source transformation systems and languages such as DMS [1], Rascal [12], Stratego [2], or TXL [5] that provide powerful general purpose set of capabilities for addressing a wide range of software analysis problems. Kermeta indeed concentrates on one given aspect of DSML design: adding executability to their meta-models in such a way that any other tool can still be used for advanced analysis or transformation purposes. Still, as illustrated in this paper, Kermeta can also be used to program simple, algorithmic and object-oriented transformations for DSML (e.g.; the *local-to-global* transformation.

In the world of Modeling, Model Integrated Computing (MIC) [23] is probably the most well known environment centered on the development of DSML. The MIC comprises the following steps:

- Design a Domain Specific Modeling Language (DSML): this step allows engineers to create a language that is tailored to the specific needs of the application domain. One has also to create the tools that can interpret instances of this language (i.e. models of the application),
- this language is then used to construct domain models of the application,

[2] http://www.eclipse.org/modeling/tmf/

– the transformation tool interprets domain models to build executable models for a specific target platform,

This approach is currently supported by a modeling environment including a tool for designing DSMLs (GME) [6] and a model transformation engine based on graph transformations (GREAT). MIC is a standalone environment for Windows of a great power but also of a great complexity. Kermeta brings in a much more lightweight approach, leveraging the rich ecosystem of Eclipse, and providing the user with advanced composition mechanisms based on the notion of aspect to modularly build his DSML environment within Eclipse.

Another approach builds on the same idea: multi-paradigm modeling. It consists in integrating different modeling languages in order to provide an accurate description of complex systems and simulate them. The approach is supported by the ATOM3 graph transformation engine [24].

Microsoft Software Factories [9] propose the factory metaphor in which development can be industrialized by using a well organized chain of software development tools enabling the creation of products by adapting and assembling standardized parts. A software factory may use a combination of DSMLs to model systems at various levels of abstraction and provide transformation between them. A software factory schema is a directed graph where the nodes are representing particular aspects (called viewpoints) of the system to be modeled and edges represent transformations between them. In particular, viewpoints provide the definition of the DSMLs to be used to create model of the viewpoints, development processes supporting model creation, model refactoring patterns, constraints imposed by other viewpoints (that help ensuring consistency between viewpoints) and finally any artifact assisting the developer in the implementation of models based on viewpoints. Transformations between viewpoints are supported mostly in an hybrid or imperative way through templates, recipes and wizards that are integrated as extensions to Visual Studio. Compared to Software Factories, Kermeta provides an integration platform that makes it much easier to develop independantly and later combine the various aspects of a development environment for a given DSML. Further Kermeta follows OMG standards (MOF, OCL, etc.) and is smootly integrated in the Eclipse platform, that provides an alternative open source IDE to Visual Studio and Software Factories.

In the Eclipse environment, several languages have been developed on top of OCL for model navigation and modification. For instance the Epsilon Object Language (EOL) [13] is a meta-model independent language that can be used both as a standalone generic model management language or as infrastructure on which task-specific languages can be built. The EOL is not object-oriented (in the sense that it does not define classes itself), even if it is able to manage objects of types defined externally in EMF meta-models in the spirit of JavaScript. In contrast to the EOL, Kermeta is an object-oriented (and aspect-oriented) extension to the EMF, providing full static typing accross the languages it integrates: E-Core, OCL and Kermeta.

6 Conclusion

This article presented the Kermeta platform for building Eclipse based, integrated environments for DSML. Based on an aspect oriented paradigm [16,10] Kermeta has been designed to easily extend meta-models with many different concerns, each expressed in its most appropriate language: MOF for abstract syntax, OCL for static semantics, Kermeta itself for dynamic semantics and model transformations [17], Java for simulation GUI, etc.

Technically, since Kermeta is an extension of MOF, a MOF meta-model can be seen as a valid Kermeta program that just declares packages, classes and so on but *does* nothing. Kermeta can then be used to *breath life* into this meta-model, i.e. transform it into a full blown development environment by introducing nicely modularized aspects for handling concerns of static semantics, dynamic semantics, or model transformations, each coming with Eclipse editor support.

Kermeta is already used in many real life projects: more details are available on www.kermeta.org.

References

1. Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Dms. In *ICSE*, pages 625–634, 2004.
2. Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, 2008.
3. Franck Chauvel and Jean-Marc Jézéquel. Code generation from UML models with semantic variation points. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages –, Montego Bay, Jamaica, octobre 2005. Springer.
4. Curtis Clifton and Gary T. Leavens. Multijava: Modular open classes and symmetric multiple dispatch for java. In *In OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, 2000.
5. James R. Cordy. The txl source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006.
6. James Davis. Gme: the generic modeling environment. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 82–83, New York, NY, USA, 2003. ACM Press.
7. Zoé Drey, Cyril Faucher, Franck Fleurey, and Didier Vojtisek. *Kermeta language reference manual*, 2006.
8. Torbjörn Ekman and Görel Hedin. The jastadd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.
9. Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, August 2004.
10. Jean-Marc Jézéquel. Model driven design and aspect weaving. *Journal of Software and Systems Modeling (SoSyM)*, 7(2):209–218, may 2008.
11. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean M. Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

12. Paul Klint, Jurgen J. Vinju, and Tijs van der Storm. Language design for meta-programming in the software composition domain. In *Software Composition*, pages 1–4, 2009.
13. Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The epsilon object language (eol). In *ECMDA-FA*, pages 128–142, 2006.
14. P.-A. Muller. *From MDD Concepts to Experiments and Illustrations*, chapter On Metamodels and Language Engineering. ISTE, ISBN 1905209592, 2006.
15. Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model-driven analysis and synthesis of concrete syntax. In *Proceedings of the MoDELS/UML 2006*, Genova, Italy, octobre 2006.
16. 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, octobre 2005. Springer.
17. Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, octobre 2005.
18. Object Management Group (OMG). Meta Object Facility (MOF) specification. OMG Document ad/97-08-14, septembre 1997.
19. Simon Pickin, Claude Jard, Thierry Jéron, Jean-Marc Jézéquel, and Yves Le Traon. Test synthesis from UML models of distributed software. *IEEE Transactions on Software Engineering*, 33(4):252–268, avril 2007.
20. Mark Richters and Martin Gogolla. OCL: Syntax, semantics, and tools. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 42–68. Springer, 2002.
21. Sébastien Saudrais, Olivier Barais, and Noël Plouzeau. Integration of time issues into component-based applications. In *The 10th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE'07)*, Medford (Boston area), Massachusetts, USA, juillet 2007. Springer Lecture Notes in Computer Science (LNCS).
22. Jim Steel and Jean-Marc Jézéquel. On model typing. *Journal of Software and Systems Modeling (SoSyM)*, 6(4):401–414, décembre 2007.
23. Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *IEEE Computer*, 30(4):110–111, 1997.
24. Hans Vangheluwe, Ximeng Sun, and Eric Bodden. Domain-specific modelling with atom3. In *ICSOFT 2007, Proceedings of the Second International Conference on Software and Data Technologies, Volume PL/DPS/KE/MUSE, Barcelona, Spain, July 22-25, 2007*, pages 298–304, 2007.
25. Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher-order attribute grammars. In *PLDI*, pages 131–145, 1989.
26. Tewfik Ziadi, LoÃŕc Hélouët, and Jean-Marc Jézéquel. Towards a UML profile for software product lines. In *Proceedings of the Fifth International Workshop on Product Familly Engineering (PFE-5)*, volume 3014 of *LNCS*, pages 129–139. Springer Verlag, 2003.