Weaving executability into object-oriented meta-languages

Pierre-Alain Muller, Franck Fleurey, Jean-Marc Jézéquel

Weaving Executability into Object-Oriented Meta-Languages

Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel

IRISA / INRIA Rennes
Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 RENNES Cedex - France
{pa.muller, franck.fleurey, jean-marc.jezequel}@irisa.fr

Abstract. Nowadays, object-oriented meta-languages such as MOF (Meta-Object Facility) are increasingly used to specify domain-specific languages in the model-driven engineering community. However, these meta-languages focus on structural specifications and have no built-in support for specifications of operational semantics. In this paper we explore the idea of using aspect-oriented modeling to add precise action specifications with static type checking and genericity at the meta level, and examine related issues and possible solutions. We believe that such a combination would bring significant benefits to the community, such as the specification, simulation and testing of operational semantics of metamodels. We present requirements for such statically-typed meta-languages and rationales for the aforementioned benefits.

1 Introduction

In the model-driven engineering community, meta-languages such as MOF [11, 12], EMOF [12] or Ecore [3] are increasingly used to describe the metamodels of domain-specific languages. These meta-languages focus on structural specifications but have no built-in support for the definition of actions. Therefore, they cannot be used to precisely specify the operational semantics of metamodels. When such a precise semantics is needed, one has to resort to external languages, either imperative such as Java, or declarative such as OCL [14].

In this paper we explore the idea of using aspect-oriented modeling to build an executable meta-language by composing action metamodels with existing meta-languages. We examine how imperative control structures and iterators can be incorporated, while ensuring static typing capabilities and preserving compatibility with existing tools supporting MOF-like meta-languages (e.g.; Eclipse/EMF).

The intent of this paper is to describe why an executable meta-language would be valuable, to examine how such language could be composed using aspect-oriented modeling and then to expose how it may be promoted to become an executable meta-language.

This paper is organized as follows: Section 2 presents our motivations for adding action specifications in metamodels and justifies our proposal. Section 3 examines

Preprint of the article accepted for publication at MODELS 2005: The 8th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences) Jamaica, 2-7 October 2005

how aspect-oriented modeling can be used to extend existing meta-data languages with action specifications and presents the metamodel of KerMeta (our experimental language). Section 4 shows how KerMeta is used to define a simple finite-state machine language. Section 5 examines some related works and finally the conclusion opens some general perspectives about aspect-oriented meta-modeling.

2 Motivations

In this section we present the rationales of our work. We explain why we believe that an action specification capability at the meta-level would be useful, and we examine some alternatives.

Why do we need to extend meta-languages, aren't meta-data languages powerful enough? MOF is an example of an object-oriented meta-data language, which provides support for metamodel modeling via object-oriented constructs such as classes, operations, attributes and relations. MOF defines operations, but not their implementation counterparts, which have to be described in text. The following example is excerpted from the MOF 2.0 Core Specification. The definition of the *isInstance* operation of the EMOF class *Type* (section 12.2.3 page 34) is given as follows:

Operation ***isInstance(element : Element) : Boolean***

"*Returns true if the element is an instance of this type or a subclass of this type. Returns false if the element is null*".

Such specification of the operational semantics of an operation is not easily amenable to automatic execution. Indeed, we need a real programming language, which goes beyond a meta-data description language. According to N. Wirth [20], a program is made of data structures + algorithms; so we propose to see executable metamodels as meta-data + actions. We see a lot of value in this capability, for instance for model transformations which involve strong algorithmic facets, e.g. the synthesis of state-machines from hierarchical message sequence charts [21].

The following specification given in our experimental language KerMeta is an example of executable specification which could be used in place of the previous textual description.

```
operation isInstance(element : Element) : Boolean is do
   // false if the element is null
   if element == void then result := false
   else
      // true if the element is an instance of this type
      // or a subclass of this type
      result := element.getMetaClass == self or
          element.getMetaClass.allSuperClasses.contains(self)
   end
end
```

Fig. 1. Executable specification of the *isInstance* operation of the EMOF *Type* class

Preprint of the article accepted for publication at MODELS 2005: The 8th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences) Jamaica, 2-7 October 2005

Why not use an existing programming language like Java? 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…).

There is no easy way to simultaneously restrict and extend such existing languages. Extension is difficult because of the typing mismatch between MOF and Java. Restriction is even more difficult, and we would have to stay with the existing language design choices for things such as single- or multiple-inheritance of classes, early- or late-binding, and schemes of method redefinition.

Why not use the OCL language? OCL (the Object Constraint Language) has been designed to express side-effect free constraints on UML models, and is also very well adapted for expressing queries and navigation expressions.

Provided that we would restrict OCL to work only on the subset of UML which could be aligned onto the MOF, we could use OCL to specify operations in terms of pre- and post-conditions. Nevertheless, and this was already pointed out by S. Mellor and al. [8], there is often a need to include some level of algorithmic specifications to ensure efficient implementation. This could be done by extending OCL with constructions such as assignment or object creation to support an imperative style for writing complex algorithms.

Why not use the Action Semantics? Like OCL, the Action Semantics is defined for UML models. The Action Semantics defines a minimal set of concepts useful to describe actions. Action languages are free to provide more sophisticated constructs, as long as these constructs can be translated into the basics concepts defined by the Action Semantics.

Executable meta-level languages and the UML Action Semantics are defined at different levels of abstraction. The Action Semantics defines fine-grained general purpose actions. Executable meta-languages define specific actions dedicated to metamodel specifications; e.g. the specification of the Action Semantics itself.

3 Using aspect-oriented modeling to extend a meta-language

In this section, we examine how aspect weaving can be used to extend existing meta-languages. Fig. 2 shows how we use aspect-oriented modeling [5, 17] to compose the primary structural aspect (e.g. EMOF) with a behavioral aspect (which conforms to EMOF as well). The resulting metamodel is then further promoted to the M3 level, and can be substituted to EMOF. The composition process is designed to ensure that existing metamodels (such as UML), already defined in terms of EMOF, remain fully compatible with the new executable meta-language.

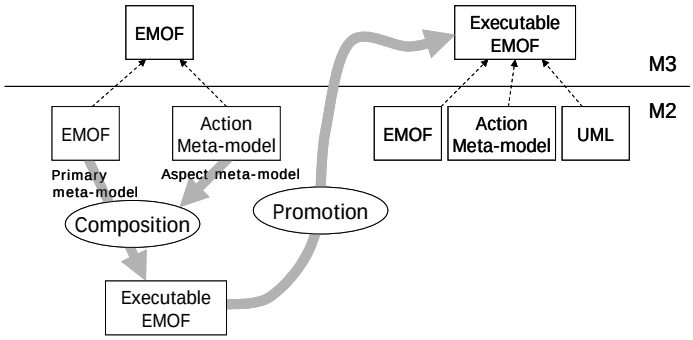


Fig. 2. Composing an action metamodel into the EMOF metamodel

The following paragraphs investigate how a meta-data language and a statically typed action language can be woven into a consistent executable meta-language. Section 3.1 motivates the choice of EMOF, section 3.2 lists the language constructs for precise action specification, section 3.3 presents how actions can be attached to EMOF operations and discusses the use of classical OO behavioral extension mechanisms. Sections 3.4 and 3.5 present several issues related to EMOF type system and some possible extensions to allow static typing of the action specifications. The discussion is illustrated by examples in KerMeta, our prototype language presented in section 3.6.

3.1 Choosing a meta-data language

The process described in this paper is directly applicable to any object-oriented meta-data language, such as the OMG MOFs (MOF 1.4, CMOF and EMOF) or Eclipse ECore. For our prototype language, we have chosen EMOF as the structural base for the executable meta-language. This decision is motivated by two main reasons: first because EMOF is standardized by the OMG and second as it is well-supported by tools such as Eclipse/EMF.

EMOF provides concepts for structuring data (package, classes, properties…), mechanisms for extending data structures (inheritance) and a type system (based on classes, primitive types and enumerations).

The following paragraphs will examine issues and solutions to integrate action specifications with those concepts and mechanisms.

3.2 Adding imperative control structures

Adding imperative control structures into EMOF requires identifying how behavior can be attached to the structural constructions of the meta-language. As EMOF is already object-oriented, we have found that it is convenient to use class operations to hold this behavior specification.

Our work takes places in a software engineering context which includes relations with industrial partners who have very strong reliability concerns. In this context,

static typing is a recognized best practice as it allows many kinds of early verifications, such as model-driven editing, testing and simulation.

Therefore, we have chosen to complement EMOF with imperative and statically-typed action specifications, which include the following control structures:

- Conditionals, loops and blocks.
- Local variable declarations.
- Call expressions for reading variables and properties or calling operations.
- Assignment expressions for assigning variables and properties.
- Literals expressions for primitive types,.
- Exception handling mechanism: exceptions can be raised and caught by rescue blocks.
- A limited form of lambda expressions which correspond to the implementation of OCL-like iterators such as *collect*, *select* or *reject* on collections.

Fig. 4 shows how behavior is attached to operations, using the property *body* of class *Operation* (to define the behavior of operations) and the properties *getterbody* and *setterbody* of class *Property* (to specify derived properties).

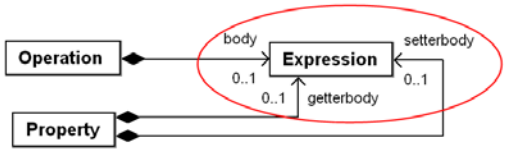


Fig. 3. Using operation bodies as join points between structure and behavior

However, this simple join point between data and behavior is not sufficient to define a full-fledged object-oriented executable meta-language, and a special attention has to be paid to ensure compatibility between extension mechanisms and type systems. The next subsections examine these points.

3.3 Adding late binding

Experience with the OO paradigm has demonstrated that operation redefinition is a useful and powerful mechanism to define the behavior of objects and support variability. However, EMOF does not provide semantics neither for operation call nor for operation inheritance and redefinition.

The issue of choosing semantics for operation overriding has been widely studied for the design of OO languages [1]. However, OO languages have not adopted a unique solution to this problem. For the sake of simplicity, in the current version of KerMeta we have chosen to implement invariant overriding (i.e. no specialization) and to exclude overloading.

EMOF provides multiple inheritance of classes. In the current version of KerMeta, we have chosen to include a minimal selection mechanism that allows the user to explicitly select the inherited method to override when several implementations of an operation are inherited. In the future, we plan to include a more general mechanism such as *traits* proposed by Schärli et al [15]. In any case, we believe that the conflict resolution mechanism should be explicitly stated by the programmer.

The following picture shows the extensions that we have made to the *Operation* class to support operation inheritance and redefinition (the red ellipses show what has been added).

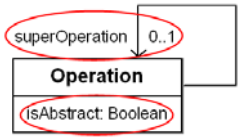


Fig. 4. Extensions to support operation inheritance and redefinition

3.4 Adding genericity

As stated in section 2, an executable meta-language should include convenient model navigation capabilities. This section discusses the static typing of such navigation expressions.

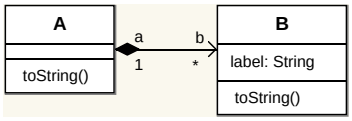


Fig. 5. A very simple metamodel

Fig. 5 presents a small metamodel which defines two classes *A* and *B* linked through an association. Let *myA* be an instance of class *A*; we need to write expressions such as:

myA.b.first.label

The type of *myA* is *A*, and according to EMOF the type of *myA.b* is *Set* whereas it should be *Set* of *B*. The type checking problem raised by this expression then concerns the type of *myA.b.first* that must be determined statically in order to check that it contains an attribute called *label* (*first* is supposed here to be an operation on sets which returns the first element of a set). Java, for instance, would ask the programmer to explicitly specify the expected type with static casts: *((B)myA.b.first).label*. More generally, the problem is that the type of what is returned by operations on collections depends on the contents of the collection at runtime. The following paragraphs detail two options for implementing static typing in this context.

Option 1: Specific language constructions. Collections (sets, ordered sets, bags and sequences) are introduced in the language, as in Xion [10] or MTL [19]. This can be viewed as an extension of the definition of arrays in OO languages such as Java 2.

This option requires syntactic and semantic language constructions specific to collections and the definition of corresponding policies in the type-checker. For this reason we did not chose this option for KerMeta.

Option 2: Parameterized classes support. Parameterized classes (generics) are included in the language. This way, typed collections can be defined like any other class. The concept of generics is now widely adopted in OO languages, including the

recent versions of Java and C#, and it offers an elegant solution to static typing. However, the introduction of parameterized classes imposes some changes in the EMOF type system.

In EMOF a type can be a primitive type, an enumeration or a class. Adding type parameters on classes makes the link between classes and types more complex. For instance, a class *Collection<G>* defined with a type parameter *G* does not define a type *Collection* but engenders a family of types by binding *G* to actual types. In other words the type of a variable cannot simply be a *Collection* but must be a collection of something.

Fig. 6 shows the principle of weaving genericity into EMOF. Box A shows the original EMOF classes. Class *Class* inherits from class *Type* and contains a set of *Features*, i.e. properties and operations. Box B displays how genericity can be modeled. On one hand a parameterized class is modeled by a class *ClassDefinition* which contains a set of *TypeVariables* and a set of *Features*. On the other hand *ParametrizedClass* inherits from *Type*, references a *ClassDefinition* and is composed of a binding between the type variable of the class definition and actual types.

A careful weaving must be performed to obtain a model both compatible with EMOF and which supports genericity. In our context, "compatible with EMOF" means that EMOF features should be available in the woven model. As an example, since EMOF provides a class named *Class* that has two properties *isAbstract* and *ownedFeatures*, the resulting model should provide a similar structure.

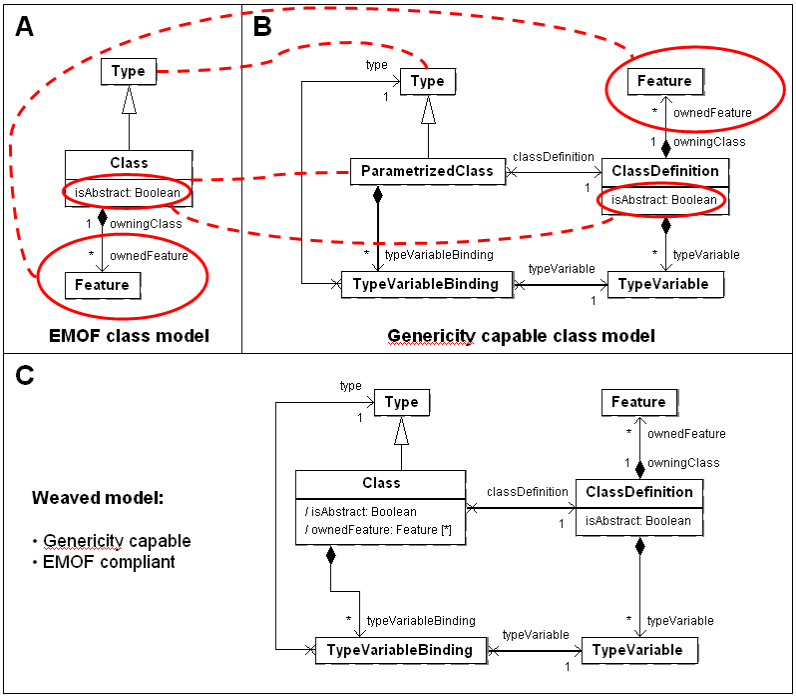


Fig. 6. Adding genericity to EMOF

Box C presents the result of the composition process. Classes such as *Type* or *Feature* play the role of join points between A and B models. EMOF class *Class* has been merged with classes *ClassDefinition* and *ParametrizedClass*. The name *Class* has been kept instead of *ParametrizedClass* to ensure EMOF compliance. Finally, some derived properties such as *isAbstract* and *ownedFeature* have been added to be fully-compatible with EMOF.

We have chosen this solution for the KerMeta language not only because it provides a general answer to the typing problem of navigation expressions but also, as presented in the next section, because it helps designing an elegant solution to the typing problem of OCL-like iterators.

3.5 Adding function objects (typed functions)

Iterators (such as *collect*, *select*, *reject* and *foreach*) are some of the most convenient constructions of OCL because they simplify a lot collection processing. The following expression (based on the metamodel described in Fig. 5) illustrates the kind of expressions that we want to type statically; the operation *toUpper* is supposed to be defined on String.

myA.b.collect{ o | o.label }.first.toUpper .

Statically type checking such an expression raises two issues. First, in order to type-check the expression *o.label*, the type of variable *o* must be deduced from the type of the elements of the collection *myA.b*. Second, the type of the elements contained in the collection *myA.b.collect{ o | o.label }* depends on the type of the expression *o.label* in the body of *collect*.

In the following paragraphs, we examine two options to address these issues of typing.

Option 1: Specific language constructions. The problem can be addressed by defining iterators directly as constructions of the language. This is what is done in languages such as Xion, MTL, Java or C#. Iterators are implemented through specific statements such as *foreach* in C# and Java. This solution requires the set of iterators to be fixed during language design.

Each iterator needs to be specifically added to the language and specific rules for the type checker must be designed. Furthermore, this solution does not allow the programmer to easily define custom iterators or similar constructions on her classes. We thus rather propose a more general solution involving parameterized operations and an extension of the type system.

Option 2: Function types and generic operations support. The idea here is to extend the language to allow the definition of OCL-like iterators as regular operations in class *Collection*. In practice, this is implemented in dynamic OO languages such as Ruby [18] and in functional languages. This can be implemented in a statically typed OO language by extending the type system to support function types and by using generic operations.

The *collect* iterator for instance applies an expression to each element of a collection and collects the results. The operation *collect* has one parameter: an expression which is applied to each element of the collection. It returns a collection which contains the results of the application of the expression passed as parameter on each element of the collection. This example pin-points two major typing issues:

- The type of the parameter is a function. The language should then support function types.
- The return type depends on the type of the parameter. The language should support generic operations.

Using these concepts, iterators can be defined as regular operations. Fig. 7 presents the definition of the *collect* iterator in the class *Collection* of our prototype language.

```
abstract class Collection<G> {
[...]
   operation collect<T> (collector : <G -> T>) : Sequence<T> is do
      result := Sequence<T>.new
      from var it : Iterator<G> init iterator
      until it.isOff
      loop
         result.add( collector(it.next) )
      end
   end
[...]
}
```

Fig. 7. Using an operation to implement the collect iterator

Operation *collect* has a parameter *collector* which represents the expression to be applied to each element of the collection. The type of the parameter is a function type *<G -> T>*; the actual parameter should be a function applicable on the elements of the collection. *T* is a type parameter of the operation; it allows the expression of the link between the type of the parameters and the return type. Here the return type is a *Sequence* of *T*.

Fig. 8 shows how function types and generic operations can be added into EMOF, consistently with the adjustments made previously to support genericity.

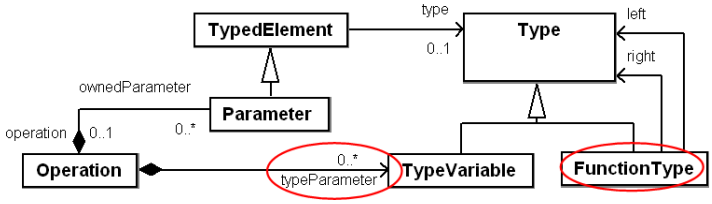


Fig. 8. Adding function objects and generic operations to EMOF

The class *TypeVariable* defines the generic formal type parameters of the operation, which must be used as the type of the formal parameters and return type of the operation. Actual types are bound to the generic formal type parameters for each call of the operation depending on the type of the actual parameters. In KerMeta, support for function types is provided by a sub-class *FunctionType* of EMOF class *Type*.

3.6 The result: KerMeta

Fig. 9 shows an excerpt of the KerMeta metamodel, which results from the weaving of EMOF with our action specification metamodel. As detailed previously, it includes constructions that permit the static typing of KerMeta expressions in addition to EMOF constructions. The KerMeta metamodel can be divided in two parts,: structural and behavioral. The structural part is fully-compatible with EMOF. The behavioral part corresponds to class *Expression* and its sub-classes and is used for the specification of the operational semantics of metamodels.

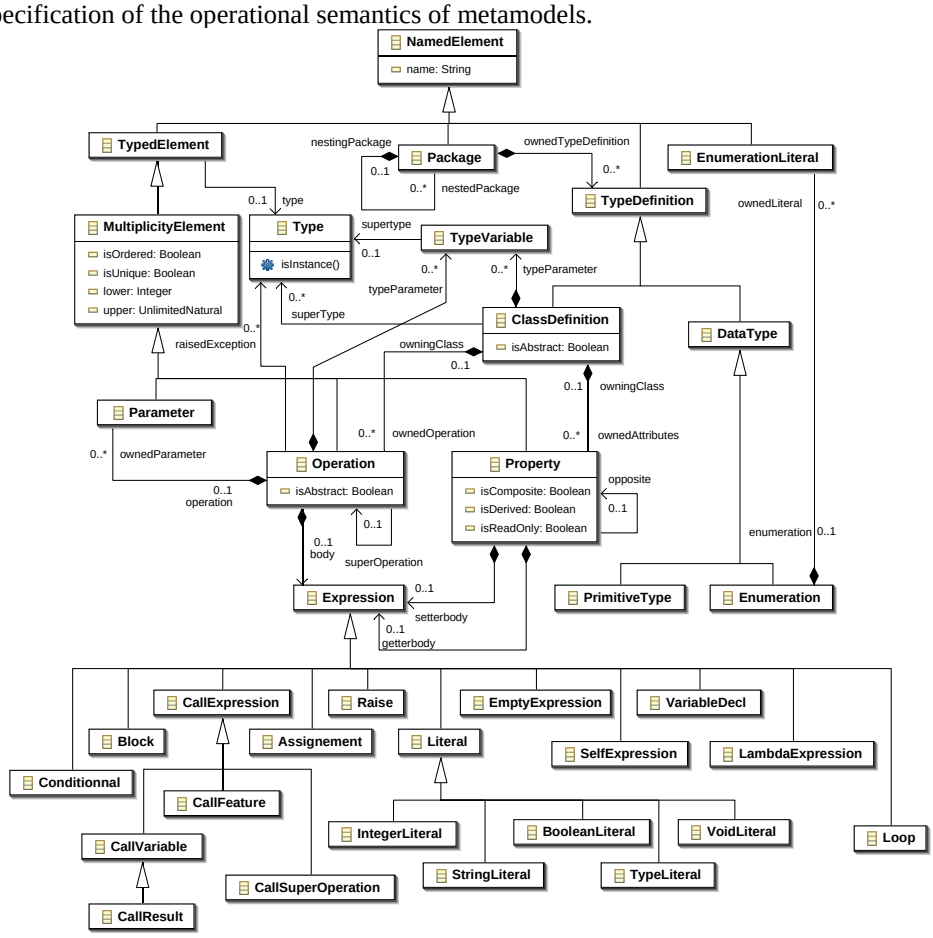


Fig. 9. Excerpt of the Kermeta metamodel

The next section presents an example of using KerMeta to define a simple state-machine language.

Preprint of the article accepted for publication at MODELS 2005: The 8th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences) Jamaica, 2-7 October 2005

4 A simple example: a language to define Finite-State Machines

This section illustrates the benefits of an executable meta-language. The example is based on the definition of simple finite state-machines such as the one presented Fig. 10. Each state is labeled by a string, and each transition reads a character and writes another one. Section 4.1 shows how both the structure and the semantics of such a language can be defined and section 4.2 presents some simple example of how it can be used.

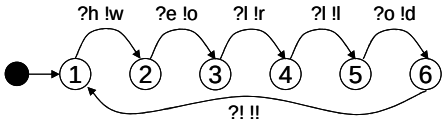


Fig. 10. A simple state machine to translate "*hello*" into "*world*"

4.1 Definition of the FSM language

Fig. 11 presents a metamodel for this simple FSM (Finite-State Machines). An FSM is composed of states, it refers to an initial state and it can refer to a current state. A state has a name, it contains outgoing transitions and it refers to incoming transitions. A transition contains an input character and an output character and it refers both to a source and to a target state.

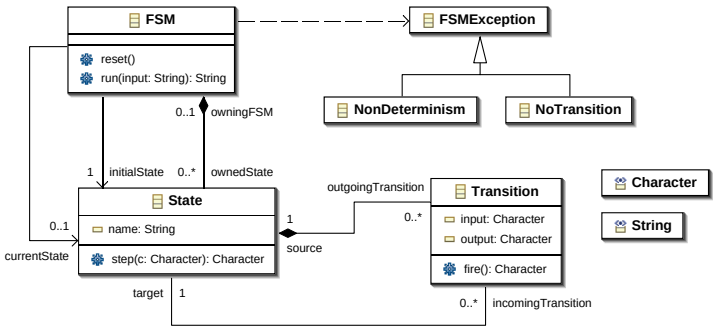
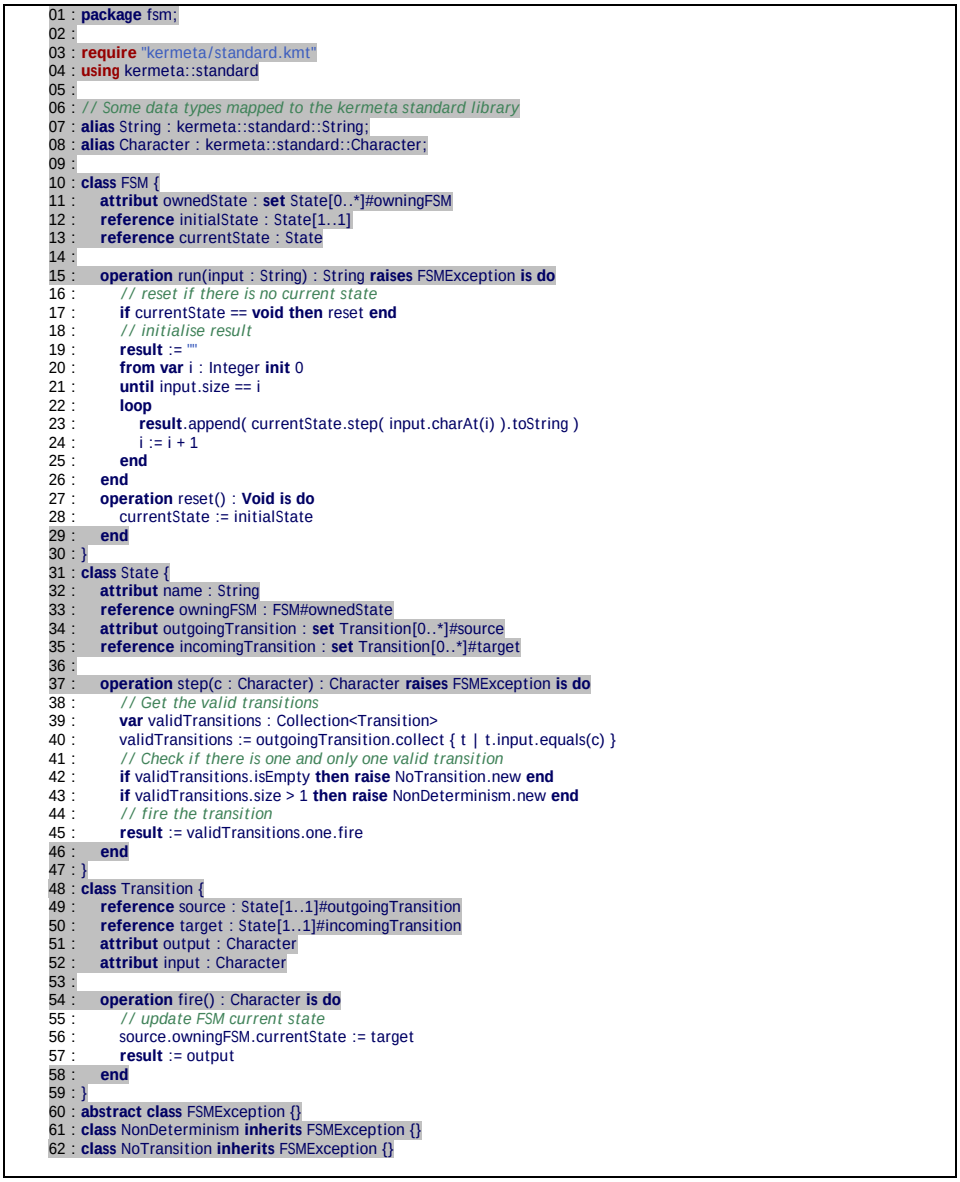


Fig. 11. Simple FSM metamodel

Fig. 12 presents the FSM metamodel in the textual syntax of KerMeta. The code highlighted in gray is derived from the class diagram represented in Fig. 11 and conforms to the structural part of KerMeta (and thus also to EMOF). The remaining part of the code represents the specification of the behavior of the operations (which could not be expressed in vanilla EMOF). The method *run* of the FSM reads its input string character by character. If the current state contains an outgoing transition which matches the read character then this transition is fired. If no transition or several transitions match the read character then an exception is raised. When a transition is fired the current state of the FSM becomes the target state of the transition.



```
01 : package fsm;
02 :
03 : require "kermeta/standard.kmt"
04 : using kermeta::standard
05 :
06 : // Some data types mapped to the kermeta standard library
07 : alias String : kermeta::standard::String;
08 : alias Character : kermeta::standard::Character;
09 :
10 : class FSM {
11 :     attribut ownedState : set State[0..*]#owningFSM
12 :     reference initialState : State[1..1]
13 :     reference currentState : State
14 :
15 :     operation run(input : String) : String raises FSMException is do
16 :         // reset if there is no current state
17 :         if currentState == void then reset end
18 :         // initialise result
19 :         result := ""
20 :         from var i : Integer init 0
21 :         until input.size == i
22 :         loop
23 :             result.append( currentState.step( input.charAt(i) ).toString )
24 :             i := i + 1
25 :         end
26 :     end
27 :     operation reset() : Void is do
28 :         currentState := initialState
29 :     end
30 : }
31 : class State {
32 :     attribut name : String
33 :     reference owningFSM : FSM#ownedState
34 :     attribut outgoingTransition : set Transition[0..*]#source
35 :     reference incomingTransition : set Transition[0..*]#target
36 :
37 :     operation step(c : Character) : Character raises FSMException is do
38 :         // Get the valid transitions
39 :         var validTransitions : Collection<Transition>
40 :         validTransitions := outgoingTransition.collect { t | t.input.equals(c) }
41 :         // Check if there is one and only one valid transition
42 :         if validTransitions.isEmpty then raise NoTransition.new end
43 :         if validTransitions.size > 1 then raise NonDeterminism.new end
44 :         // fire the transition
45 :         result := validTransitions.one.fire
46 :     end
47 : }
48 : class Transition {
49 :     reference source : State[1..1]#outgoingTransition
50 :     reference target : State[1..1]#incomingTransition
51 :     attribut output : Character
52 :     attribut input : Character
53 :
54 :     operation fire() : Character is do
55 :         // update FSM current state
56 :         source.owningFSM.currentState := target
57 :         result := output
58 :     end
59 : }
60 : abstract class FSMException {}
61 : class NonDeterminism inherits FSMException {}
62 : class NoTransition inherits FSMException {}
```

Fig. 12. Simple FSM in KerMeta concrete syntax

4.2 Benefits of defining the FSM language with an executable meta-language

The benefits of using KerMeta to describe a language such as the FSM language are two-fold. First, KerMeta is, by construction, compatible with EMOF which allows reusing existing EMOF tools to manipulate state machines. Fig. 13 (A) presents a screenshot of the model editor generated by EMF from the state machine metamodel. Second, as KerMeta is executable, the state machines can be instantly simulated. Fig. 13 (B) presents a screenshot of the Kermeta interactive interpreter. The *require* statement is used to get access to a metamodel. The operation *system.load* loads a model from its representation in XMI (generated by the EMF generated editor).

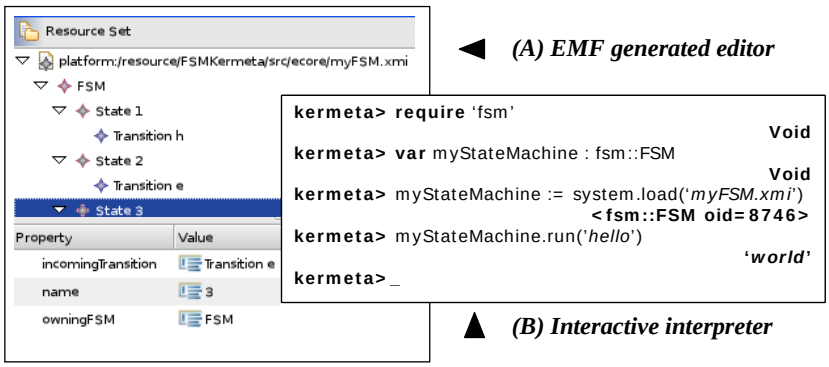


Fig. 13. KerMeta editor and interpreter

5 Related works

Our work is related to many other works, and can be considered as some kind of synthesis of these works, in the specific context of model-driven engineering applied to language definition. The sections below include the major areas of related works.

Grammars, graphs and generic environment generators. Much of the concepts behind our work take their roots in the seminal work conducted in the late sixties on grammars and graphs and in the early eighties in the field of generic environment generators (such as Centaur [2]) that, when given the formal specification of a programming language (syntax and semantics), produce a language-specific environment. The generic environment generators sub-category has recently received significant industrial interest; this includes approaches such as Xactium [4], or Software Factories [7]. Among these efforts, it is Xactium which comes closer to our work. The major differences include the fact that we use aspect composition to generate the meta-meta level while preserving a clean separation of concerns, and that we have a fully static type system.

Model-driven environments. There are several ongoing efforts to promote the convergence of open-source model-driven environments, for instance: Modelware (http://www.modelware-ist.org/), TopCaseD (http://www.laas.fr/SPIP/spip-topcased/) and TopModL [9], and our work takes place in this context.

Generative programming and domain-specific languages. Generative programming aims at modeling and implementing system families in such a way that a given system can be automatically generated from a specification written in a domain-specific language. This includes multi-purpose model-aware languages such as Xion [10] or MTL [19], or model transformation languages such as QVT [13].

We share the vision of generative programming, and we use models to generate fully executable code which can be compiled. The Xion and MTL languages have had a direct impact on our work.

QVT is different as it addresses mappings between models. QVT works on structures, by specifying how one structure is mapped into another one; for instance translating a UML class diagram into a RDBMS schema. QVT is not suitable for the definition of the behavior of metamodels.

Meta-CASE systems. Meta-CASE systems, such as MetaEdit [16], Dome [6] or EMF [3], provide customized software engineering environments, separately from the main software components. The major difference with meta-CASE systems is that we remove the M3 level by the promotion process.

6 Conclusion

In this paper we have been discussing the rationales and benefits of weaving executability into meta-data languages. We have presented the benefits of defining the operational semantics of metamodels (using a statically-typed meta-language) for testing and simulation purposes.

We have explained how it is possible to design new executable meta-languages, using aspects to weave metamodels of existing meta-data languages with metamodels for precise action specifications. We have examined general issues related to meta-languages extensions, and we have presented several options to address these issues.

We have presented how to generate the metamodel of KerMeta, our experimental meta-language, and illustrated the benefits of static-typing and genericity, via an example featuring finite-state machines.

The same kind of process could be used to build tools such as parsers, code generators, interpreters or editors for specific languages. In fact each kind of tool requires a specific aspect to be composed with the metamodel of the language. For instance, a concrete syntax could be woven into the model of the language to allow the generation of a parser. Generally speaking, the idea is to define the aspect metamodel of the information which has to be composed in order to automatically obtain tools to work with domain-specific languages.

Preprint of the article accepted for publication at MODELS 2005: The 8th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences) Jamaica, 2-7 October 2005

References

1. Abadi, M. and Cardelli, L. *A theory of objects*. New York: Springer, 1996.
2. Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B. and Pascual, V. Centaur: the system. *Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on practical software development environments*, *13* (5). 14 - 24.
3. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R. and Grose, T. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.
4. Clark, T., Evans, A., Sammut, P. and Willans, J. Applied Metamodelling: A Foundation for Language Driven Development, http://albini.xactium.com, 2004.
5. Clarke, S. Extending standard UML with model composition semantics. *Science of Computer Programming*, *44* (1). 71-100.
6. Engstrom, E. and Krueger, J. Building and rapidly evolving domain-specific tools with DOME. *Proceedings of IEEE International Symposium on Computer-Aided Control System Design (CACSD 2000)*. 83-88.
7. Greenfield, J., Short, K., Cook, S., Kent, S. and Crupi, J. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.
8. Mellor, S., Tockey, S., Arthaud, R. and Leblanc, P. Action Language for UML: Proposal for a Precise Execution Semantics. *Proceedings of UML 98 (LNCS1618)*. 307-318.
9. Muller, P.-A., Dumoulin, C., Fondement, F. and Hassenforder, M. The TopModL Initiative. in *UML Satellite Activities*, Springer, 2005, 242-245.
10. Muller, P.-A., Studer, P., Fondement, F. and Bezivin, J. Platform independent Web Application Modeling and Development with Netsilon. *Accepted for publication in Journal on Software and Systems Modelling (SoSym)*. http://www.sciences.univ-nantes.fr/lina/atl/www/papers/netsilon_sosym.pdf.
11. OMG. Meta Object Facility (MOF) Specification 1.4, Object Management Group, http://www.omg.org/cgi-bin/doc?formal/2002-04-03, 2002.
12. OMG. MOF 2.0 Core Final Adopted Specification, Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/03-10-04, 2004.
13. OMG. Revised submission for MOF 2.0 Query/View/Transformation, Object Management Group (QVT-Merge Group), http://www.omg.org/cgi-bin/apps/doc?ad/2005-03-02, 2005.
14. OMG. UML 2.0 Object Constraint Language (OCL) Final Adopted specification, Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/2003-10-14, 2003.
15. Scharli, N., Ducasse, S., Nierstrasz, O. and Black, A. Traits: Composable units of behavior. *Proceedings of ECOOP 2003*.
16. Smolander, K., Lyytinen, K., Tahvanainen, V.-P. and Marttiin, P. MetaEdit: a flexible graphical environment for methodology modelling. *Proceedings of the third international conference on Advanced information systems engineering*.
17. Straw, G., Georg, G., Song, E., Ghosh, S., France, R.B. and Bieman, J.M. Model Composition Directives. *Proceedings of the 7th International Conference Unified Modelling Language: Modelling Languages and Applications*. 84-97.
18. Thomas, D., Fowler, C. and Hunt, A. *Programming Ruby - The Pragmatic Programmer's Guide, Second Edition*, 2004.
19. Vojtisek, D. and Jézéquel, J.-M. MTL and Umlaut NG: Engine and Framework for Model Transformation. *ERCIM News*, *58*.
20. Wirth, N. *Algorithms + data structures = programs*. Prentice-Hall, Englewood Cliffs, 1976.
21. Ziadi, T., Hélouët, L. and Jézéquel, J.-M. Revisiting statechart synthesis with an algebraic approach. *Proceedings of the 26th International Conference on Software Engineering (ICSE 04)*.