CrossMark

Mashup of metalanguages and its implementation in the Kermeta language workbench

Jean-Marc Jézéquel · Benoit Combemale · Olivier Barais · Martin Monperrus · François Fouquet

Abstract With the growing use of domain-specific languages (DSL) in industry, DSL design and implementation goes far beyond an activity for a few experts only and becomes a challenging task for thousands of software engineers. DSL implementation indeed requires engineers to care for various concerns, from abstract syntax, static semantics, behavioral semantics, to extra-functional issues such as runtime performance. This paper presents an approach that uses one *metalanguage* per language implementation concern. We show that the usage and combination of those metalanguages is simple and intuitive enough to deserve the term *mashup*. We evaluate the approach by completely implementing the non-trivial fUML modeling language, a semantically sound and executable subset of the Unified Modeling Language (UML).

J.-M. Jézéquel · B. Combemale (✉) · O. Barais · F. Fouquet
IRISA, INRIA, University of Rennes 1, Rennes, France
e-mail: benoit.combemale@irisa.fr

J.-M. Jézéquel
e-mail: jean-marc.jezequel@irisa.fr

O. Barais
e-mail: olivier.barais@irisa.fr

F. Fouquet
e-mail: francois.fouquet@irisa.fr

M. Monperrus
LIFL, INRIA, University of Lille 1, Lille, France
e-mail: martin.monperrus@univ-lille1.fr

1 Introduction

Model-driven engineering (MDE) of software fosters the use of multiple domain-specific languages (DSL) during software design and development, up to several DSLs per engineering concern and application domain [1]. In the model-driven terminology, DSLs are generally defined using metamodels and consequently, DSL programs are generally referred to as models. For instance, a model-driven development process may use a DSL to simulate the system in proto-typing phases, another DSL to define its software architecture and yet another one to specify the set of valid inputs so as to allow test-case generation. The goal of using multiple DSLs is to improve various aspects of software, such as improving consistency with requirements, reducing development costs or reducing the number of bugs [1].

However, model-driven development is not silver bullet. One of its drawbacks is that industry, instead of relying on a small number of general-purpose languages, now needs many modeling or programming environments of production-level quality. In other terms, language design and implementation goes far beyond an activity for a few experts only and becomes a challenging task for thousands of software engineers and domain experts, which we call *DSL engineers* [2].

Domain-specific language (DSL) engineers who are responsible for designing and implementing a tool-supported DSL can of course use general-purpose programming languages such as Java. However, implementing a DSL runtime (whether compiler of interpreter) is complex. It requires orchestrating various concerns, as different as the definitions

🌀 Springer

of abstract and concrete syntaxes, static semantics (including the well-formedness rules), behavioral semantics, as well as extra-functional issues such as compile-time or runtime performances and memory footprint.

That is the reason for which researchers and innovators invent *language workbenches* [3–6]. A language workbench provides DSL engineers with languages, libraries or tools to ease the design and implementation of DSLs. Centaur [7] is an early contribution in this field, more recent approaches include Metacase's MetaEdit+ [8], Microsoft's DSL Tools [9], Clark et al.'s Xactium [10], Krahn et al.'s Monticore [11] Kats and Visser's Spoofax [12] and Jetbrain's MPS [13].

In this paper, we present the Kermeta language workbench. In a nutshell, the Kermeta workbench involves one different metalanguage per DSL implementation concern: one metalanguage for the abstract syntax (aligned with EMOF [14]); one for the static semantics (aligned with Object Constraint Language (OCL) [15]) and one for the behavioral semantics (Kermeta Language).[1] The Kermeta workbench uses an original modular compilation scheme to compose the three different metalanguages. We do not aim at presenting *how to* define a DSL in Kermeta. This is the goal of our tutorial paper published at GTTSE [16]. This paper is meant to describe the rationales and the evaluation of the main design choices of the workbench, as well as a thorough presentation of its novel compilation scheme.

Let us now say a few words about the composition of the meta-languages in the Kermeta workbench. The DSL concerns are organized into loosely coupled modules. They are all compiled into traits of the Scala programming language [17]. The composition semantics we use is inspired from the concept of aspects [18] and open classes [19]. In practice, this combination of metalanguages only uses two keywords ("aspect" and "require") and is simple and intuitive enough to deserve the term *mashup*. Implementing DSLs with the Kermeta workbench relieves DSL engineers from the burden of expressing and composing the abstract syntax together with the static and behavioral semantics.

The rest of the paper reads as follows:

– Section 2 describes fUML, a real-world modeling DSL defined by OMG. fUML is an executable subset of the Unified Modeling Language (UML). This DSL is used throughout the paper to illustrate the concepts as well as for evaluation.
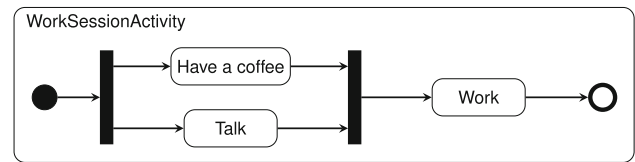

Fig. 1 Activity of the members of our team during our work sessions

– Section 3 presents the three metalanguages used in the Kermeta language workbench, the rationales and a qualitative evaluation of their conjunct usage to design DSLs. For a user manual on using them, we refer the reader to [16].
– Section 4 exposes the compilation scheme used to compose the three metalanguages aforementioned. In particular, the composition is described both formally and with respect to language operators of the Scala programming language. The evaluation of this section shows that the ease of design is not traded for efficiency. Indeed, the Kermeta version of fUML is as fast as the reference Java implementation.
– Section 5 compares the Kermeta workbench against the related work. The main strength of Kermeta Workbench is the use of standardized metalanguages which open doors to inter-interoperability with other tools and reusability of specifications.

Section 6 concludes and proposes directions of future work.

2 The case of fUML

We consider throughout the article the case of fUML (*Foundational Subset for Executable UML Models*) [20], an executable subset of the UML that can be used to define the structural and behavioral semantics of software systems. It is computationally complete by specifying a full behavioral semantics for activity diagrams. This means that this DSL enables implementors to execute well-formed fUML models (here *execute* means to actually *run a program*).

The main rationales of choosing fUML are that: (1) fUML is a real DSL with a non-trivial semantics, and (2) fUML has a Java reference implementation. We will compare our Kermeta design and implementation against this reference design and implementation in terms of modularity and performance.

2.1 The specification of fUML

fUML is an executable language for activity models. As an example, Fig. 1 shows an executable fUML model representing the activity of our team when we meet for work sessions. We are used to first having a coffee while talking together

---

[1] The concrete syntax is achieved thanks to a full compatibility with all EMF-based tools for concrete syntax, such as the de facto standards EMFText (see http://www.emftext.org) and XText (see http://www.eclipse.org/Xtext/).

about the latest news. When we finish to drink our coffee and to talk, we begin to work.

The fUML abstract syntax As illustrated by our example, the core concept of fUML is *Activity* that defines a particular behavior. An *Activity* is composed of different elements called *Activity Nodes* linked by *Activity Edges*. The main nodes which represent the executable units are the *Executable Nodes*. For instance, *Actions* are specific *Executable Nodes* that are associated with a specific executable semantics. Other elements define the activity execution flow, which can be either a control flow (*Control Nodes* linked by *Control Flow*) or a data flow (*Object Nodes* linked by *Object Flow*).

The example on Fig. 1 uses an illustrative set of elements of the abstract syntax of fUML. The start of the *Activity* is modeled using an *Initial Node*. A *Fork Node* splits the control flow in two parallel branches: one for the *Action* of having a coffee and the other for the *Action* of talking to each other. Then a *Join Node* connects the two parallel branches to the *Action* of working.

We refer the reader to the specification [20] for all the details about the comparison with UML2 and the whole description of the fUML abstract syntax.

The fUML static semantics In the specification, additional constraints are defined in order to precise the semantics and to clarify ambiguities. fUML uses the OCL in order to define those constraints. As an example, the additional constraint *fUML_is_class*, applied over the specific action *CreateObjectAction*, tells us that it can only be linked to an instance of *Class* (i.e., one can not create, say activities at runtime). In the specification, the OCL constraints are clearly separated from the rest of text.

The fUML behavioral semantics To support the execution of models, fUML introduces a dedicated *Execution Model*. The activity execution model has a structure largely parallel to the abstract syntax using the *Visitor* design pattern [21] (called *SemanticVisitor*). Note that although the semantics is explained using visitors, which are rather at the implementation level, it is left open by the fUML specification to implement the language using other means that visitors.

In addition, to capture the behavioral semantics, the interpretation needs to define how the execution of the activity proceeds over time. Thus, concepts are introduced in the execution model for which there is no explicit syntax. Such concepts support the behavior of an activity in terms of *tokens* that may be held by nodes and *offers* made between nodes for the movement of these tokens.

Based on the execution model, the specification denotationally describes the behavioral semantics of fUML using axioms in first-order logic.

2.2 The Java reference implementation of fUML

ModelDriven.org[2] is a consortium of government, commercial and university members which develops model-driven technologies. One of the consortium projects is to provide a reference implementation of fUML.[3] This implementation is written in Java and is capable of executing any valid fUML model which are specified in an XML-based format. In this reference implementation, all the concerns of the DSL described above are written in Java.

3 The specification and design of DSLs in Kermeta

Kermeta is a language workbench designed for specifying and designing DSLs. For this, it involves different metalanguages depending on the concern: abstract syntax (we will also use the term "metamodel" to refer to it[4]), static semantics and behavioral semantics.

The workbench integrates the OMG standards EMOF and OCL, respectively, for specifying the abstract syntax (cf. Sect. 3.1) and the static semantics (cf. Sect. 3.2). The choice of EMOF and OCL has been driven by the fact that they are de facto standards. This allows full interoperability with other tools. Indeed, during the years of development of Kermeta, we have reused EMOF models and OCL constraints from many different sources. The workbench also provides *Kermeta Language* to address the specification of the operational semantics (cf. Sect. 3.3).

The Kermeta Workbench also provides composition operators responsible for mashing-up these different concerns into a standalone execution engine (interpreter or compiler) of the DSL (cf. Sect. 3.4).

In this section, we illustrate all these features by presenting the implementation of fUML. Then we evaluate the use of the Kermeta workbench from the end user point of view (here, the DSL engineer) on the basis of this case study (cf. Sect. 3.5).

3.1 Concern #1: abstract syntax definition

First of all, to build a DSL in Kermeta, one defines its abstract syntax (i.e., the metamodel), which specifies the domain concepts and their relations. The abstract syntax is expressed in an object-oriented manner, using the OMG metalanguage

---

[2] See http://portal.modeldriven.org/.

[3] See http://portal.modeldriven.org/project/foundationalUML.

[4] This is one definition in the community. For some researchers, "metamodel" sometimes referred to abstract syntax plus static semantics.
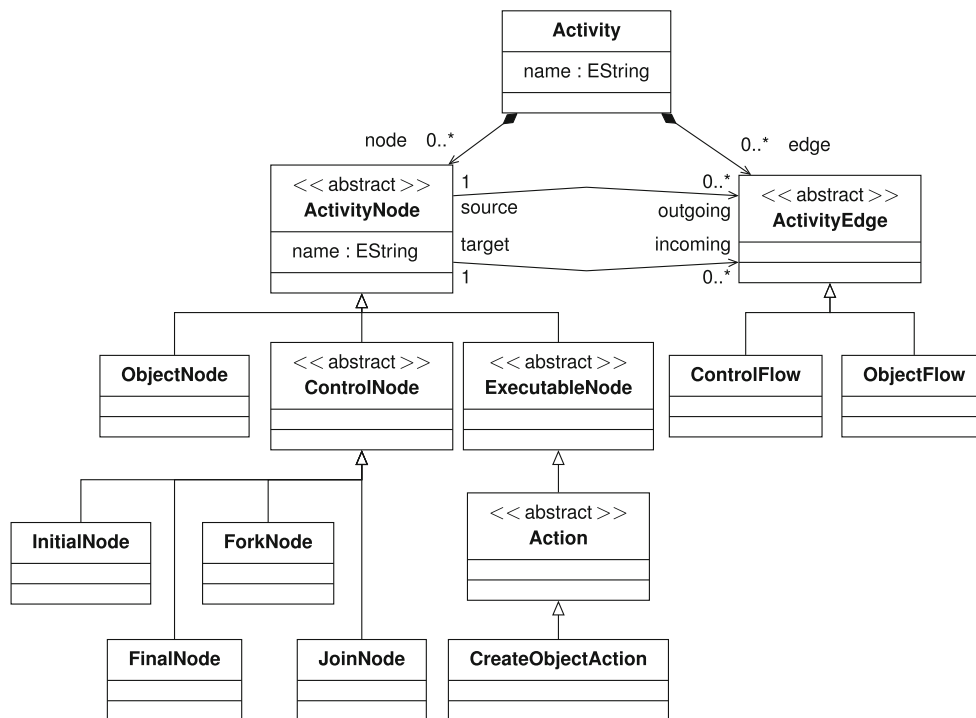
Fig. 2 Excerpt of the fUML metamodel

EMOF (Essential Meta Object Facility) [14]. EMOF provides the following language constructs for specifying a DSL metamodel: package, classes, properties, multiple inheritance and different kinds of associations between classes. The semantics of these core object-oriented constructs is close to a standard object model that is shared by various languages (e.g., Java, C#, Eiffel). *We chose EMOF for the abstract syntax because it is a* de facto *standard allowing interoperability with other tools*.

Figure 2 shows the excerpt of the fUML metamodel explained in the previous section and depicts those classes as a class diagram. In our Kermeta-based fUML design, we reuse the abstract syntax standardized by the OMG. In practice, OMG provides the fUML metamodel in terms of EMOF, and we automatically translate it into an Ecore-based metamodel (the format supported by the Kermeta workbench).

Since the abstract syntax is expressed as an object-oriented metamodel, a concrete fUML model (equivalent to a DSL progam) is composed of instances of the metamodel classes.

3.2 Concern #2: static semantics definition

The static semantics of a DSL is the union of the well-formed rules on top of the abstract syntax (as invariants of domain classes) and the axiomatic semantics (as pre- and postconditions on operations of metamodel classes). The static semantics is used to statically filter incorrect DSL programs before actually running them. It is also used to check parts of the correctness of a DSL program's execution either at design-time using model-checking or theorem proving, or at runtime using assertions, depending on the execution domain of the DSL. Kermeta uses the OMG OCL [15] to express the static semantics, directly woven into the metamodel using the Kermeta aspect keyword.

Listing 1 shows the previously introduced fUML invariant (specified with the keyword "inv") as expressed in the Kermeta workbench using OCL.

Listing 1 Weaving the Static Semantics of fUML into the Standard Metamodel

```
1  package fuml;
2  require "fuml.ecore"
3  aspect class CreateObjectAction {
4  // The given classifier must be a
       class.
5  inv fUML_is_class :
6    self.classifier.oclIsKindOf(Class
       )
7  }
```

In the Kermeta workbench, the abstract syntax and the static semantics are conceptually and physically (at the file level) defined in two different modules. Consequently, it is possible to define different semantic variants for the same domain, i.e., to have a single EMOF metamodel shared by different static semantics, e.g., to cope with language variants.

3.3 Concern #3: behavioral semantics definition

Essential metaobject facility (EMOF) does not include concepts for the definition of the behavioral semantics, and OCL is a side effect free language. To define the behavioral semantics of a DSL, we have created the Kermeta Language, an action language that is used to express the behavioral semantics of a DSL [22]. It can be used to define either a translational semantics or an operational semantics [23]. A translational semantics would result in a compiler while an operational semantics would result in an interpreter. In this paper, for sake of clarity, we will only present operational semantics. However, the language is exactly the same in both cases.

Using the Kermeta language, an operational semantics is expressed as methods of the classes of the abstract syntax [22]. Listing 2 is an excerpt of the operational semantics of fUML. Using the `aspect` keyword, a method "execute" is added to the metamodel class "Activity." The body of the method imperatively describes what is the effect of executing an activity. In this case, it consists of (i) creating an activity node activation group and activating all the activity nodes in the activity and (ii) copying the values on the tokens offered by the output parameter nodes to the corresponding output parameter.

The Kermeta language is imperative, statically typed and includes classical control structures such as blocks, conditionals, loops and exceptions. The Kermeta language also implements traditional object-oriented mechanisms for handling multiple inheritance and generics. The Kermeta language provides an execution semantics to all EMOF constructs that must have a semantics at runtime such as containment and associations. First, if a reference is part of a bidirectional association, the assignment operator semantics handles both ends of the association at the same time. Second, if a reference is part of a containment association, the assignment operator semantics unbinds existing references if any, so that one object is part of another one. Finally, for multiple inheritance, Kermeta borrows the semantics from the Eiffel programming language [24].

3.4 Composition operators for the mashup of metalanguages

As introduced above, mashing-up all DSL concerns in the Kermeta workbench is achieved through two keywords in the Kermeta language: `aspect` and `require`.

In Kermeta, all pieces of static and behavioral semantics are encapsulated in metamodel classes. For instance, in Listing 2, the behavioral semantics are expressed in the metamodel classes "Activity." The `aspect` keyword enables DSL engineers to relate the language concerns (abstract syntax, static semantics, behavioral semantics) together. It allows DSL engineers to reopen a previously created class to add some new pieces of information such as new methods, new properties or new constraints. It is inspired from open-classes [19] and will be further discussed in Sect. 4.

Listing 2 In Kermeta, the DSL operational semantics is expressed as methods of metamodel classes

```
1   aspect class Activity inherits
        Executable{
2   // the semantics of executing an
        activity
3   method execute(runnable : Runnable) :
        Void is do
4       // Creation of an activity node
            activation group
5       runnable.execute()
6       var group :
            ActivityNodeActivationGroup init
             ActivityNodeActivationGroup
7                   .new()
8       group.execution := runnable
9       runnable.group := group
10
11      // Activation of all the activity
            nodes in the activity
12      runnable.group.activate(self.node,
13          self.edge)
14      var outputNodeActivations :
15          OrderedSet<ActivityParameterNode
                >
16          init runnable.group.
17      fumlGetOutputParameterNodeActivations
            ()
18
19      // Copy the values on the tokens
            offered by output parameter
            nodes to the corresponding
            output parameters
20      outputNodeActivations.each {
            outputNodeActivation |
21          var parameterValue :
                ParameterValue init
                ParameterValue.new()
22          parameterValue.parameter := (
                outputNodeActivation.asType(
                ActivityParameterNode)).
                parameter
23          var tokens : Set<Token> init
                outputNodeActivation.
24                  fumlGetTokens()
25          tokens.each { token |
26              var val : Value init (token.
                    asType(ObjectToken)).val
27              if (val != void) then
28                  parameterValue.values.
                        add(val)
29              end
30          }
31          runnable.fumlSetParameterValue(
                parameterValue)
32      }
33  end
34  }
```

The keyword `require` enables the mashup itself. A DSL implementation *requires* an abstract syntax, a static semantics and a behavioral semantics. Listing 3 shows how the final fUML mashup looks like in Kermeta. Three `require`

are used to specify the three concerns. The require mechanism also provides some flexibility with respect to static and behavioral semantics. For example, several behavioral semantics could be defined in different modules (all on top on the same metamodel) and then chosen depending on particular needs (e.g., simulation, compilation). It is also convenient to support semantic variants of a given language. For instance, Kermeta can be used to specify several implementations of UML semantics variation points.

Listing 3 Mashup of the fUML Concerns

```
1  package fuml;
2  require "fuml.ecore" // abstract
       syntax
3  require "fuml.ocl"   // static
       semantics
4  require "fuml.kmt"   // operational
       semantics
5  class Main {
6    operation Main(): Void is do
7      // Execute a fUML model
8    end
9  }
```

3.5 Evaluation of the mashup-based design From the end user viewpoint

The Kermeta-based design of fUML follows the approach described in this section. It means that all fUML concerns (abstract syntax, statics semantics and behavioral semantics) are separated in different units and that the fUML runtime environment is the result of the mashup. In other terms, the Kermeta-based design clearly separates the three concerns of DSL implementation, and thus loyally reflects the structure of the specification.

For us, the driving motivation of having three different modules for implementing a DSL is (1) allowing that each part of a DSL is done by different stakeholders (e.g., the abstract syntax by the standardization body and the compiler by the tool vendors), possibly in parallel; (2) supporting different variation points (either syntactic or semantic). The main con is that there is one DSL engineer, responsible for the metalanguage integration, who must understand the different modules.

Jezequel et al. [16] presented a reference manual on how to design and implement a DSL with the Kermeta language workbench. Beyond *how to*, we present in this section the advantages of our approach from the viewpoint of the DSL engineer (the end user in our context).

3.5.1 Are concerns designed in different modules?

Our mashup approach provides two dimensions of modularity:

- modularity of domain concepts (metamodel concepts)
- modularity of language engineering concerns (parsing, static semantics, interpretation, compilation, etc.)

The first one of course does not bring anything new with respect to a Java-based approach (e.g., the fUML reference implementation): This is just the usual class-based modularity found in OO languages, including MOF. Still it is very helpful when one wants to slightly change an existing concept, or even add or remove one into the meta-model.

So the originality of our approach lies in the modularity of language engineering concerns. It is accepted that the design of a modeling language deals at least with [25] the abstract syntax (called a metamodel in the MDE terminology), the set of constraints on the abstract syntax (called static semantics) and the behavioral semantics. Using our approach, all these concerns are implemented in different modules:

- the abstract syntax is defined as an EMOF metamodel [14]. Technically speaking, it is completely defined in an Ecore file (e.g., `fuml-metamodel.ecore`). Please refer to [26] for more details about this file format. This module is standalone and does not depend from others modules.
- the static semantics is defined in a module dedicated to invariants, pre- and postconditions. When using OCL, this means creating a file, say `fuml.ocl`. This module imports the language metamodel (the Ecore file aforementioned) and has no other dependencies.
- the behavioral semantics (also known as execution semantics) is defined in a dedicated module using the Kermeta language, say `fuml.kmt`.

There is an exact one-to-one mapping between the abstract concerns of language design and the concrete design modules. The design of the modules are clearly layered so that there are no more spurious design dependencies than logical dependencies.

This architecture also enables DSL engineers to get rid of certain heavyweight design patterns, such as the design pattern *Visitor* which is often used to inject the semantics. Not using such patterns has two advantages. First, the DSL design is easier to understand and maintain. Second, at runtime, our DSL architecture requires less communication between objects (delegates and proxy calls), which contributes to a better efficiency (this point is further discussed in Sect. 4).

3.5.2 Are concerns designed using appropriate metalanguages?

The research on aspect-oriented software development has shown [27] that not all languages are equal to implement aspects. Certain concerns are well suited to be implemented

```
/****** In Java *********/                              /****** In Kermeta *********/
class Pin extends ObjectNode {
  // simulates multiple inheritance using delegation     aspect class Pin inherits ObjectNode, MutiplicityElement {
  public multiplicityElement = new MultiplicityElement();  ...
}                                                       }
// elsewhere
class InputPinActivation extends PinActivation {        aspect class InputPinActivation {
  public boolean isReady() {
    // The DSL engineer has to know                       operation isReady() : Boolean is do
    // all low-level implementation choices, here the      // "lower" from MultiplicityElement on node
    // use of delegation to simulation multiple inheritance  var minimum : Integer init self.node.lower
    int minimum = this.node.multiplicityElement.lower;     end
  }
}                                                       }
```

Fig. 3 Comparison of the Java implementation against the Kermeta implementation. The programmer is relieved from the accidental complexity due to the code generation

using DSL; they are sometimes called domain-specific aspect languages.

Using our approach, all concerns are designed using *meta*-DSLs. The abstract syntax is designed using EMOF, the static semantics is designed using OCL and the operational semantics is designed using the Kermeta language. Let us now review the advantages of certain particular *meta*-DSLs for the related concerns.

Metamodeling with EMOF We argue that EMOF is really appropriate to design DSL metamodels:

– it is based on object-oriented modeling. Thus, any engineer who is fluent with object-oriented thinking is able to intuitively design a language metamodel with EMOF.
– it is the result of years of discussion between DSL experts: it contains a lot of constructs that are known to be useful for metamodeling (e.g., association containment and multiple inheritance).
– the tool support for EMOF is good: there are several vendors and mature tools; it is possible to express EMOF metamodels using different textual and graphical editors.

Static semantics with OCL Object Constraint Language (OCL) is well suited for defining a DSL static semantics for the same reasons as those presented in the previous paragraph (maturity and tool support). Furthermore, since OCL is side effect free, it is impossible for DSL engineers to accidentally introduce some pieces of behavioral semantics in the static semantics definition. In other words, the DSL design itself participates to ensuring the separation of concerns. On the contrary, using Java/AspectJ or another general-purpose programming language for expressing the static semantics would open the door to concern tangling.

Operational semantics with the Kermeta language As already stated, Kermeta is a workbench as well as a language. As a language (*Kermeta Language*), it has been specifically designed to express the operational semantics of languages.

Let us now review a couple of examples that show the power of the Kermeta language with this respect.

Manipulating collections of objects The operational semantics of DSLs often deals with manipulating collections of objects. For instance, a DSL for state machines would at some point traverse all transitions starting from a given state. Kermeta includes useful functions based on lambda expressions to manipulate collections: e.g., *collect*, *select* or *reject* (similar functions can be found in OCL and in some general-purpose languages such as Smalltalk). Those constructions give a natural way to navigate through models compared to iteration over Java collections.

Manipulating metamodel concepts within the operational semantics When implementing a behavioral semantics, one often manipulates concepts of the metamodel. If the language used to implement operational semantics does not natively support metamodeling concepts, the operational semantics is bloated with workarounds to approximate the metamodeling concepts. For example, Fig. 3 represents an excerpt of the fUML implementation where the *Pin* class inherits from both *ObjectNode* and *MutiplicityElement*, and *isReady* is a method of *Pin* that expresses a piece of operational semantics. Let us compare how to handle two excerpts of this method in Java and Kermeta.

This listing emphasizes one important characteristics of our approach. The Kermeta language enables to directly manipulate the concepts of the language without having to use special wrapper methods. For instance, on the right-hand side listing, `lower` directly refers to the field *lower* of the metamodel class *MutiplicityElement*. On the contrary, in Java, the DSL engineer always has to master the simulation of the semantics of multiple inheritance, association, containment, etc. In other words, our approach lowers the representational gap between the code of the operational semantics and the metamodel concepts (i.e., of the abstract syntax). Also, writing the operational semantics with the Kermeta language enables to avoid the bloating (code generation, annotations, etc.) due to embedding the powerful semantics of the metamodeling language (EMOF) into a programming language that does not support it by default (e.g., Java).

4 The compilation of the mashup of metalanguages in Kermeta

This section presents how we compile a mashup of three meta-languages into a single DSL. This poses a number of challenges, described in Sect. 4.1 that we solve by generating Scala code as exposed in Sect. 4.2. The resulting code satisfies the composition semantics expressed at the metalanguage level (cf. Sect. 4.3).

4.1 Challenges

We have to choose a single appropriate target language for compiling the mashup. As input, we have three different metalanguages (Ecore, OCL, Kermeta). For sake of simplicity, the target language must not be too far from the concepts and the abstraction level of our metalanguages. While many modern programming language can be appropriate (e.g., Java, C#, Python), we have identified the following challenges that should be tackled in our context:

Challenge #1: Expressing the composition semantics The target language must not only be appropriate for supporting the compilation of each metalanguage, it must also support the expression of our composition semantics (*à la* open-class) used in the Kermeta language.

Challenge #2: Integration with legacy code We described in the introduction that DSL engineers have strong incentives to design their language in a way that is interoperable with other language engineering tools. For instance, there is today a strong ecosystem of language engineering tools around the EMF platform. The core of EMF is a generator that outputs Java code responsible for handling the metamodeling semantics and marshaling (read and save DSL programs from disk). The second design challenge of our compilation chain is to plug our mashup compiler onto the untouched code generated from EMF.

Challenge #3: Efficiency The approach aims to improve the design of DSLs in providing support of separation of concerns; one of the risk is to introduce a performance overhead. The last challenge is to produce executables as efficient as an ad hoc implementation.

4.2 Using Scala as a target language

Our mashup compiler generates Scala code from the three language concerns. Indeed, Scala is a solution to the three aforementioned challenges: (1) it has a low gap with OCL and Kermeta. In particular, OCL and Kermeta closures are straightforwardly compiled to Scala closures; (2) Scala's mixin composition semantics is a nice building block for
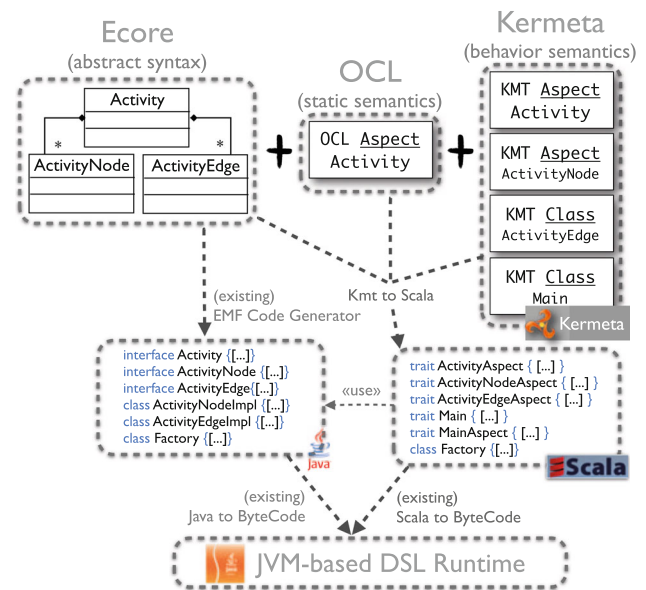

Fig. 4 Java and Scala elements generated from a simple Ecore and Kermeta metamodel

defining our open-class composition semantics at the metalanguage level; (3) it is able to seamlessly use the Java code generated from the EMF compiler; and (4) Scala is known to be efficiently compiled into bytecode and there has been significant work on Scala performance [28].

4.3 Mashup compilation scheme

The main input of the mashup is a set of Ecore classes, the static semantics defined in OCL and the behavioral semantics defined in Kermeta (see Fig. 4). To build the mashup at the bytecode level, the mashup compiler must plug new generated code into the generated, untouched code from EMF.

4.3.1 Overview

The compilation scheme relies on six main generated artifacts as follows. Most explanations refer to Listing 4 which shows an excerpt of the generated Scala code.

1. EMF generates one Java class ($A_{Java_C}^{Emf}$) and one Java Interface ($A_{Java_I}^{Emf}$) per type of the abstract syntax (that we now call, respectively, metaclass and metainterface).
2. EMF generates one factory class ($Factory_{Java}^{Emf}$) per Ecore package.
3. The Kermeta compiler generates one trait per class for the mashup of the static semantics and the behavioral semantics ($A_{Scala_T}^{Kmt}$) (e.g., `ActivityAspect`, see line 9).
4. The Kermeta compiler generates one Scala class for composing the EMF class and the aforementioned traits

(*RichA*$_{Scala_C}$) (e.g., class RichActivity, see line 23. Note that ActivityImpl is generated by EMF).

5. The Kermeta compiler generates one Scala single-ton object that extensively uses Scala implicits (e.g., ImplicitConversion, see lines 37–53).

6. The Kermeta compiler generates a factory Scala object (*RichFactory*(*RichA*$_{Scala_C}$)$_{Scala}$) that overrides the default EMF factory (e.g., RichFactory, see lines 32–36).

4.3.2 Compilation scheme of the open-class mechanism

Let us now summarize the semantics of the open-class mechanism provided by Kermeta as implemented in the compiler, and relying on existing the following object-oriented composition operators Java *extends* (•), Java *implements* (◦), Scala *extends* (⊙) and Scala *with* (◊). Since the abstract syntax is compiled using the EMF generator, the main goal of this specification is to show how Kermeta permits a fine grained integration with the Java code generated by EMF.

To compose two classes *A* and *B* (*A* ⊕ *B*) that share the same signature using Kermeta open-class operator, three cases must be considered.

Case 1 A_{Kmt} and B_{Kmt}, i.e., the two different classes are defined using the Kermeta language. For this case, the composition is done at the metamodel level. Thus, there is no integration issue at the code level. This case can be transformed to the composition of a Kermeta *C*, that results from the composition of *A* and *B* at the model level.

$$A_{Kmt}^{base} \oplus B_{Kmt}^{aspect} = C_{Kmt}^{base} \tag{1}$$

Case 2 A_{Ecore} and B_{Ecore}, i.e., the two different classes are defined using Ecore and composed using the Kermeta language. For this case, the composition is forbidden. Kermeta does not allow mixing of two Ecore classes having the same signature.

Case 3 A_{Ecore} and B_{Kmt} (or A_{Kmt} and B_{Ecore}, respectively), i.e., a class defined using Ecore is composed with a class defined using the Kermeta language. This case requires a fine grained integration at the code level between the legacy EMF code and the Kermeta aspect code. In this case $A \oplus B = A_{Ecore} \oplus B_{Kmt}$.

From A_{Ecore}, we obtain the following building blocks:

$$(i) \quad A_{Java_C}^{Emf} \bullet super(A)_{Java_C}^{Emf} \circ A_{Java_I}^{Emf} \tag{2}$$

$$(ii) \quad Factory(A)_{Java}^{Emf}$$

where *super* is a function which returns all the parent classes of a given class.

From B_{Kmt}, we obtain the following building blocks:

$$B_{Scala_T}^{Kmt} \odot super(B)_{Scala_T}^{Kmt} \tag{3}$$

Finally, the composition provides:

$$(i) \quad RichA_{Scala_C} \bullet A_{Java_C}^{Emf} \odot B_{Scala_T}^{Kmt}$$

$$(ii) \quad RichFactory(RichA_{Scala_C})_{Scala} \bullet Factory(A)_{Java}^{Emf}$$

$$(iii) \quad ImplicitConversion\ A2RichA\&B_{Scala_T}^{Kmt}2RichA \tag{4}$$

The role of the ImplicitConversion object is discussed in Sect. 4.4.4.

4.4 Specific issues

Let us now describe in details some points of interest of our compiler.

4.4.1 Integration with EMF

Our generated code integrates with EMF by extending EMF classes in the object-oriented sense. For instance, in Listing 4, the class RichActivity extends the class ActivityImpl which is generated by EMF. Also, we leverage the design pattern "abstract factory" implemented by EMF for overriding the creation of instances. The generated factory RichFactory overrides the EMF factory FactoryImpl and returns objects that are enriched with static and behavioral semantics. As a result, all EMF-compatible programs are interoperable with our DSL runtime.

4.4.2 Compiling the static semantics

The static semantics is a set of invariants and pre-/post-conditions (well-formed rules). For each metaclass, we generate a Scala trait that contains the corresponding invariants and pre- and postconditions.

To preserve substitutability, the static semantics respects the Liskov substitution principle [29]. Liskov's principle imposes some standard requirements on signatures that have been adopted in newer object-oriented programming languages. In addition to the traditional covariance and contravariance rules, there is a number of behavioral conditions leading to some restrictions on how contracts can interact with inheritance.

– Rule 1: Invariants of the supertype must be preserved in a subtype.
– Rule 2: Preconditions cannot be strengthened in a subtype, postconditions cannot be weakened in a subtype [24].

Scala provides building blocks to help support Design by Contract[5] [30]. To satisfy Rule 1, we choose to flatten the invariant inheritance at compile-time. Consequently each Scala trait representing a metaclass contains its own invariants and all the invariants of its super classes. Pre- and poscondition are also flattened statically. Consequently, we do a logical disjunction between the preconditions of an operation and the preconditions of its super operation, we do a logical conjunction between the postconditions of an operation and the postconditions of its super operation.

4.4.3 Expressing the composition

The semantics of composition that we have introduced at the meta-language level (keyword "aspect") is implemented using Scala traits as follows:

```
class Activity extends ActivityImpl
    with ActivityAspect
```

Indeed, Scala's class linearization mechanism[6] mixes in the two language implementation concerns. `ActivityImpl` comes from EMF and contains the abstract syntax definition of Activity; `ActivityAspect` is generated from the OCL specification of the static semantics and the Kermeta behavioral semantics. For type-checking, the trait uses an implicit object behind the scene that we present in Sect. 4.4.4. Let us now explain how we deal with Kermeta multiple inheritance.

Since EMOF offers multiple inheritance, Kermeta has to support it. When two methods with the same signature are present, the programmer must specify an explicit renaming. With this mechanism, developers can explicitly resolve the ambiguity by choosing one of super methods. In Scala, you can invoke the supertype using the super keyword, or you can directly reference any of the traits in the declaration of the class, possibly skipping some of the overridden methods of the traits between, by qualifying the super keyword with a trait type (*super[superClassName]*). The mashup compiler uses this mechanism for choosing in which branch we want to call the super method.

4.4.4 The need for Scala's implicits

To support the openclass mechanism, at the JVM level, a *ActivityNode* class would have to be extended with the sta-

tic semantics and the behavioral semantics integrated with Kermeta which is impossible without modifying the bytecode of the class. Since only object typed as e.g., `RichActivityNode` or `ActivityNodeAspect` can invoke the method `fire`, an object typed as a `ActivityNode` can not see the added method. In order for base instances to transparently be seen as enriched objects, the mashup compiler generates a global conversion object named `ImplicitConversion`. This object is imported in each class. It declares an implicit conversion method for converting base to rich class and aspect to rich class as illustrated in Listing 4. This conversion mechanism is based on the Scala view operator.

A view from type S to type T is defined by an implicit value which has function type $S \rightarrow T$, or by a method convertible to a value of that type. Views are applied in two situations: (i) If an expression e is of type T, and T does not conform to the expression's expected type *pt*. (ii) In a selection *e.m* with e of type T, if the selector m does not denote a member of T. In the first case, a view v is searched which is applicable to e and whose result type conforms to *pt*. In the second case, a view v is searched which is applicable to e and whose result contains a member named m.

With this mechanism, Java code that uses EMF generated types can view new methods and attributes as soon as it imports the `ImplicitConversion` object. We also reuse the view mechanism to provide an implicit conversion between EMF Collections and Scala Collections.

4.5 Compilation example

We can illustrate our composition mechanism by the simple Petri net example described in the previous section.

For the class *Activity*, *ActivityNode* and *ActivityEdge*, EMF generates three Java interfaces named *Activity*, *ActivityNode* and *ActivityEdge* and three Java classes named *ActivityImpl*, *ActivityNodeImpl* and *ActivityEdgeImpl*. The Kermeta compiler does not regenerate them and simply imports them. EMF also generates a factory implementation. In addition, our Kermeta Compiler generates Scala aspects traits *ActivityAspect*, *ActivityNodeAspect* and *ActivityEdgeAspect*, an implicit conversion object *ImplicitConversion*, and new composed class *RichActivity*, *RichActivityNode* and *RichActivityEdge* and a new factory *RichFactory* that extends generated EMF Factory to create objects extended with mixins. Note that *k2.core.Object* is the root class of the Kermeta language, it contains methods for reflection, and all classes inherit from this class.

Listing 4 shows an excerpt of code snippet illustrating the generated elements.

---

[5] https://wiki.scala-lang.org/display/SYGN/Design-by-contract.

[6] In Scala, the classes and traits inheritance hierarchy forms a directed acyclic graph (DAG). The term linearization refers to the algorithm used to "flatten" this graph for the purposes of resolving method lookup priorities, constructor invocation order, binding of super, etc. The linearization defines the order in which method lookup occurs. We refer the interested reader to [17] for an exhaustive explanation of the Scala's class linearization mechanism.

Listing 4 Excerpt of the generated Scala code from Figure 4

```scala
import kermeta.implicitconversion._


trait Main
trait MainAspect extends k2.core.
    Object {
    def newOp() = { ... }
}

trait ActivityAspect extends k2.core.
    Object, ExecutableAspect{
    def execute() = { ... }
}

trait ActivityNodeAspect extends
        k2.core.Object{
    def run() = { ... }
}

trait ActivityEdgeAspect extends
        k2.core.Object{
    def run() = { ... }
}

class RichActivity extends
    ActivityImpl with ActivityAspect
class RichActivityNode extends
    ActivityNodeImpl
        with ActivityNodeAspect
class RichActivityEdge extends
    ActivityEdgeImpl
            with ActivityEdgeAspect
class RichMain extends EObjectImpl
            with Main
            with MainAspectAspect

object RichFactory extends FactoryImpl
        {
    override def createActivity :
        Activity = {new RichActivity }
    def createMain : Main = {new
        RichMain}

}
object ImplicitConversion {
    implicit def rich(v : Activity) =
            v.asInstanceOf[
                RichActivity]
    implicit def rich(v :
        ActivityAspect) = v.
        asInstanceOf[RichActivity]
    implicit def rich(v : ActivityNode
        ) =
            v.asInstanceOf[
                RichActivityNode]
    implicit def rich(v :
        ActivityNodeAspect) =
            v.asInstanceOf[
                RichActivityNode]
    implicit def rich(v : ActivityEdge
        ) =
            v.asInstanceOf[
                RichActivityEdge]
    implicit def rich(v :
        ActivityEdgeAspect) =
            v.asInstanceOf[
                RichActivityEdge]
    implicit def rich(v : Main) =
            v.asInstanceOf[RichMain]
    implicit def rich(v : MainAspect)
        =
            v.asInstanceOf[RichMain]
}
```

4.6 Evaluation of the mashup compiler

We evaluate the mashup compiler regarding the challenges presented in the Sect. 4.1. For the first challenge, Sect. 4.3 showed how we can express the mashup composition semantics using the Scala composition operator and an architecture style that uses the Scala implicit conversion operator and the abstract factory pattern. This section focuses mainly on the composition process automation for this challenge. Then, we provide a quantitative evaluation for challenges 2 and 3. We show how the Kermeta workbench supports the integration with EMF legacy code. We discuss the performance of the resulting Scala code comparing to the Java version of the fUML case study.

4.6.1 Is the mashup of DSLs fully automated?

Kermeta fully automates the mashup of the different concerns of a DSL implementation. The language designer does not need to manipulate other constructs than `require` and `aspect` (see Sect. 3). Then, executing the mashup simply means writing: `dslprogram.execute(args)` (where `dslprogram` is an instance of the metamodel).

Furthermore, Kermeta supports compiling the mashup to a standalone Java jar file. In this case, the DSL runtime requires only a standard Java Virtual Machine and can be executed on the command line. Moreover, as a jar file, it can also be used as a library for IDE plugins (such as Eclipse plugins). This feature can be tested using the last version of the Kermeta workbench.[7]

4.6.2 Is the mashup compatible with all components of EMF ecosystem?

Our approach to implement DSLs as mashups of metalanguages is based on EMF/ECore metamodels. Thus, the mashup can be seamlessly integrated with all EMF-based technologies. Kermeta-based DSLs can readily benefit of all the generic and generative technologies which exist in the EMF ecosystem. In particular, it is compatible with textual or graphical syntax mappers and editor generators.

The tooling for manipulating the textual syntax of DSLs (tokenizers, parsers, editors) can be generated with textual syntax mappers. The generated tools translate some text (DSL programs) to instances of the abstract syntax. Hence, textual syntax mappers are logically and technically independent of the static and the behavioral semantics, and there is no need for DSL designers to provide anticipated hooks or pieces of semantics for this concern. Note that there are already very good textual syntax mappers to provide parser and editors from text to instances of an Ecore metamodel

---

[7] See https://gforge.inria.fr/forum/forum.php?forum_id=11075.

(in the context of EMF/Ecore, for instance, EmfText[8] and Xtext[9]).

Along the same line, Kermeta-based DSLs can benefit from the graphical editor generator technologies existing in EMF. Again, DSL designers do not have to anticipate the possible use of a graphical syntax when they design the DSL, and this hampers by no means the possibility of later generating a graphical editor for the DSL under consideration. For instance, in the context of fUML, an editor based on UML activity diagrams can be generated to specify and modify fUML models.

4.6.3 Quantitative evaluation of the execution time

In the previous section, we have shown that the design and the implementation of a DSL can be elegant in terms of separation of concerns and conciseness. However, this may come to the price of a high performance overhead. In this section, we provide quantitative insights into the execution performance of DSL implemented using Kermeta.

More specifically, we compare a compiled version of the Kermeta-based fUML implementation with the Java-based reference fUML implementation. The compiled version was created using the Kermeta compiler. The compiler generates JVM-ready classes out of the mashup of the abstract syntax, the static semantics and the behavioral semantics.

Experimental protocol **Subjects**

In the following, we list the fUML programs that we use to measure the performance. We also report on their size in terms of number of model elements. The three first activities were obtained from the reference fUML implementation while the last example was created by the *Logo2fUML* transformation using as input the *Hilbert* model defined in the *Kermeta's KmLogo Language Example*. As output, this transformation returns a well-formed fUML model of the corresponding activity.

TestSimpleActivities is one *fUML* test model contained in the reference implementation. This model uses different control nodes of fUML (*ForkNode*, *MergeNode*, *DecisionNode* and *JoinNode*). Different external activities are called by the *TestSimpleActivities* executable model using a sequence of *CallBehaviorAction*.

TestIntegerFunctions contains other activities that the previous ones and demonstrates the utilization of some primitive functions contained in the fUML library (it uses a set of *CallBehaviorAction* to call these primitives functions). Size of the model: 158 model elements.

Contacts is another example included in the reference implementation. This example is based on a simple operation

that counts how many objects are inside a list and returns this number for the user.

The activity starts creating an object with a *CreateObjectAction*. This object contains a list in which contacts can be inserted. Then, it counts how many objects are in the database. Afterward, an object is created using the *CreateObjectAction* (*"create contact1"*) and added to the list using the *CreateLinkAction*. Finally, the model counts again how many objects are in the database and verifies that there is one object in the database. Size of the model: 121 model elements.

Hilbert is a program example based on the *Hilbert curve* problem [31]. The main algorithm computes a continuous fractal space-filling curve, using a limited series of activity recursions. Size of the model: 723 model elements.

Measurement The main metric of the experiment is the execution speed. The measurement is subject to the following characteristics:

– we measure only the execution time and not the loading time of models. The rationale is that the reference fUML implementation and our implementation use different loading libraries (the reference implementation uses a specialized loading library while our implementation uses the EMF loading facilities).
– we present the average execution time based on 30 different runs. This allows to mitigate the measurement variations due to our multitask operating system.
– The measurement is done on a computer with an *Intel Core 2 duo 3 GHz* processor and 4 Gb memory, running Windows XP.

We also discuss the compilation time that needs to stay reasonable to be useful in practice.

Results Let us now present the execution time of the Java-based fUML implementation and the compiled version of our Kermeta-based fUML implementation. Figure 5 describes this comparison, presenting the mean in milliseconds of the execution time of each model. To obtain the following result, we run the Java-based fUML interpreter and the Kermeta-based one for each test model. We instrument the code to get the time before the loading of model, after the load of model and after the end of the execution of the fUML program. For sake of artifact evaluation and replication, fUML models and the Kermeta compiled version of the overall fUML metamodel are available online.[10]

A first finding is that our approach does not introduce a runtime performance overhead. Interestingly, all executable models run faster using the Kermeta-based implementation.

---

8 See http://www.emftext.org/index.php/EMFText.

9 See http://www.eclipse.org/Xtext/.
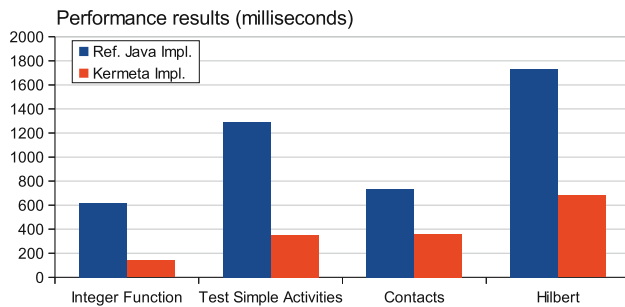
10 http://goo.gl/4RR9k.

Fig. 5 Execution time using the reference fUML implementation and the Kermeta-based implementation

The ratio is not always the same. It can be explained as they use different types of actions that not all equally optimized in the Kermeta version and in the Java Version.Note that the metrics do not take into account the loading time of models. However, we also measured the loading time of models and our Kermeta-based implementation is also faster than the one of the reference implementation. In other words, the Kermeta implementation is faster with respect to both loading and execution time.

To understand this performance improvement, we can note that several factors have an influence on this performance improvement. Among this factors, three of them can justify that the Java implementation is slower than the Scala/Kermeta one:

1. The current Java reference implementation has not been designed for performance. Some piece of the code (*e.g.,* to break some loops) have been designed more efficiently in the Kermeta implementation.
2. It has been shown that Scala is more efficient than Java. In [28], Robert Hundt shows why Scala can run faster than Java in some cases. In particular, he explains that compared to Scala, Java also creates a temporary object for *foreach* loops on loop entry. He found that this was one of the main causes of the high GC overhead for the Java version and changed the Java *forall* loops to explicit while loops. This over-use of the Garbage Collector during the loop is also a main reason why Scala goes faster during the loop that the standard Java implementation.
3. Unlike the Java compiler, the Scala compiler already performs several optimizations on its own: method inlining, escape analysis (for closure elimination) and tail call optimization. Since we compile using the *-optimize* option, all these optimizations are obtained in the current bytecode obtained from the Kermeta code.

To sum up, this experiment shows that although our approach uses and composes different metalanguages to implement a DSL, the advantages of having a clean design and a concise implementation are not traded for runtime

performance overhead. These benefits to both the design-time and runtime are possible thanks to the definition of an advanced compiler considering high-level concepts as input and producing efficient code as output. Our approach also largely benefits from the large body of work on the efficiency of Scala.

A reasonable compilation time allows easy prototyping and efficient debugging. Thus, we have measured the execution time of the Kermeta-to-Scala compilation step, and the Scala-to-bytecode compilation step on different use cases. The compilation steps for fUML (375 classes) take 1.5 s to load the Kermeta model (the AST of the mashup), 1.8 s to produce the Scala code and 61 s to produce the bytecode. This compilation time can be a problem for designers. Indeed, it can become painful if designers have to wait several seconds after each modification in the semantics to run a simulation. In investigating in which steps the compiler spends time, we can see that most of the time is consumed by the Scala compiler to type check the fUML output Scala code (20 s) and to optimize the bytecode (inlined in 20 s). Since Scala is a popular language; it now provides an incremental compiler. We could improve our tool chain to use the compiler incremental feature provided by SBT[11], Scala maven plugin[12] or FSC.[13]

5 Related work

Much work has been done on modeling languages definition, included the definition of the abstract syntax, and the static and behavioral semantics [23]. Nevertheless, none of them provides the possibility to combine different metalanguages as a core language design principle. For this purpose, Kermeta provides a composition operator (i.e., the `require` keyword) that allows us to compose the various concerns of a language, each described with a dedicated formalism. Also, the sake of composing metalanguages contains specific issues that are not well handled by generic software composition engines [32,33] (e.g., the composition of the contracts of the static semantics).

The mechanism provided by Kermeta with the `aspect` keyword can be compared to the open-class mechanism [19]. Unlike the design pattern *Visitor* [21], open-classes do not require advance planning and preserve the ability to add new properties and methods modularly and safely. Open-Class mechanism is useful for Kermeta, because it permits to reuse and enrich classes declared in metamodels. Indeed, in most cases, metamodels are built with domain analysis and specified using a graphical modeling editor. Static semantics or behavioral semantics is later integrated through aspects,

---

[11] See http://code.google.com/p/simple-build-tool/.

[12] See http://scala-tools.org/mvnsites/maven-scala-plugin/.

[13] See http://www.scala-lang.org/docu/files/tools/fsc.html.

which are statically introduced at relevant places. The composition is done statically, and the composed metamodel is typed-checked to ensure the safe integration of all units. Aspectized classes in Kermeta organize cross-cutting concerns separately from the metamodel.

Note that existing commercial frameworks perform composition of different languages through compilation. For example, all languages in the .NET framework are interoperable given their compilation to the Common Intermediate Language (CLI). In .NET, certain intra-language composition capabilities are similar to the Kermeta workbench (C# offers a kind of open-class). However, there is no inter-language class composition as the Kermeta workbench supports.

The modular compilation scheme we propose uses Scala as a target language. Other languages could be used to compile the features provided by Kermeta. For example, Groovy offers a reflexivity layer to dynamically extend a metaclass but clearly raises performance issues [34]. AspectJ could be also used to extend the generated classes in the compilation scheme, but Scala lets the compiler minimizes the representational gap thanks to features such as mixins, implicit and function type.

Several authors explored the problem of modular language design (e.g., [35–38]). For example, LISA [36] and Silver [37] integrate specific formal grammar-based language specifications supporting the automatic generation of various language-based tools (e.g., compiler or analysis tools). One practical obstacle to their adoption is a perceived difficulty to adopt specific grammar-based language specifications, and the relative gap with the development environments daily used by software engineers. JastAdd [38] combines traditional use of higher order attribute grammars with object-orientation and simple aspect-orientation (static introduction) to get better modularity mechanism. With a similar support for object-orientation and static introduction, Kermeta can be seen as a symmetric of JastAdd in the DSML world.Rebernak et al. [39] and Krahn et al. [11] contributed to the field in the context of model-driven DSLs. While there approaches was designed with the goal of easy composition of DSLs, they also advocate modularity of DSL compilers and interpreters. In the Kermeta workbench, we left for now DSL composition as future work, but we go further on DSL modularity: we marry modularity and low representational gap, the latter being obtained thanks to appropriate metalanguages (EMOF, OCL and the Kermeta action language).

A language workbench is a software package for designing software languages [40,41]. For instance, it may encompass parser generators, specialized editors and metalanguages for expressing the semantics and others [3–6]. Early language workbenches include Centaur [7], ASF+SDF [42], TXL [43] and Generic Model Environment (GME) [44]. Among more recent proposals, we can cite Metacase's MetaEdit+ [8],

Microsoft's DSL Tools [9], Clark et al.'s Xactium [10], Krahn et al's Monticore [11], Kats and Visser's Spoofax [12] and Jetbrain's MPS [13]. The important difference of Kermeta is that it is 100 % compatible with all EMF-based tools (at the code level, not only at the abstract syntax level provided by Ecore), hence designing a DSL with Kermeta easily allows reusing the rich ecosystem of Eclipse/EMF. This issue was previously addressed in the context of the Smalltalk ecosystem [45]. Kermeta brings in a much more lightweight approach using one dedicated meta-language per language design concern, and providing the user with advanced composition mechanisms to combine the concerns in a fully automated way.

6 Conclusion and future work

This paper presented an approach to design and implement DSLs that is based on using and composing several metalanguages. The key ideas behind the approach are that each language implementation concern (e.g., abstract syntax, static semantics and behavioral semantics) can easily be described in an appropriate language and that the mashup of language modules can be automatically compiled. As a result, the task of designing and implementing DSLs can be easily modularized into loosely coupled activities, with the exception of the abstract syntax on which every other component depends. Furthermore, we have also shown how the Kermeta language workbench produces DSL runtimes that are al least as fast as ad hoc implementations, and still fully compatible with the EMF ecosystem.

Future work will explore (1) the use of the Kermeta language workbench for generating DSL runtimes capable of running in constrained environments (e.g., mobile and embedded devices), (2) the use of the provided DSL modularity to support the design and implementation of language families thanks to an explicit variability modeling and (3) another level of composition: the composition of different DSLs (e.g., the GEMOC Initiative[14]), to support a coordinated use of multiple DSLs across the development of heterogeneous aspects of a system.

References

1. Schmidt, D.C.: Model-driven engineering. IEEE Comput. **39**(2), 25–31 (2006)
2. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: International Conference on Software Engineering (ICSE), pp. 471–480. ACM (2011)
3. Ghosh, D.: DSLs in Action. Manning Publications Co., Greenwich (2010)

---

[14] cf. http://gemoc.org.

4. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional, Reading (2010)

5. Mernik, M.: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. IGI Global (2013)

6. Markus, V., Sebastian, B., Christian, D., Birgit, E., Mats, H., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering—Designing, Implementing and Using Domain-Specific Languages. dslbook.org, (2013)

7. Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: Centaur: the system. ACM SIGPLAN Notices **24**(2), 14–24 (1989)

8. Tolvanen, J.-P., Rossi, M.: MetaEdit+: defining and using domain-specific modeling languages and code generators. In: Companion of the 18th Annual ACM SIGPLAN Conference OOPSLA, pp. 92–93. ACM (2003)

9. Cook, S., Jones, G., Kent, S., Wills, A.C.: Domain-Specific Development with Visual Studio Dsl Tools. Addison-Wesley Professional, Reading (2007)

10. Clark, T., Sammut, P., Willans, J.: Applied Metamodelling—A Foundation for Language Driven Development, 2nd edn. (2008)

11. Krahn, H., Rumpe, B., Volkel, S.: Monticore: modular development of textual domain specific languages. In: Objects, Components, Models and Patterns, vol. 11 of LNBI, pp. 297–315. Springer, Berlin (2008)

12. Kats, L.C.L., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and ides. ACM SIGPLAN Notices **45**, 444–463 (2010)

13. Voelter, M., Solomatov, K.: Language modularization and composition with projectional language workbenches illustrated with MPS. In: van den Brand, M., Malloy, B., Staab, S. (eds.) Third International Conference on Software Language Engineering (SLE 2010). Lecture Notes in Computer Science. Springer, Berlin (2010)

14. Object Management Group, Inc.: Meta Object Facility (MOF) 2.0 Core Specification (2006)

15. Object Management Group, Inc.: UML Object Constraint Language (OCL) 2.0 Specification (2003)

16. Jézéquel, J.-M., Barais, O., Fleurey, F.: Model driven language engineering with kermeta. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'09), vol. 6491 of Lecture Notes in Computer Science, pp. 201–221. Springer, Berlin (2011)

17. Odersky, M., Altherr, P., Cremet, V., Dragos, I., Dubochet, G., Emir, B.., McDirmid, S., Micheloud, S., Mihaylov, N., Michel, S., Spoon, L., Stenman, E., Zenger, M.: An Overview of the Scala Programming Language, 2nd edn. Technical report (2006)

18. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: European Conference on Object-Oriented Programming (ECOOP 1997), pp. 220–242 (1997)

19. Clifton, C., Leavens, G.T.: Multijava: Modular open classes and symmetric multiple dispatch for Java. In: International Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA), pp. 130–145 (2000)

20. Object Management Group, Inc.: Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0 (2011)

21. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Reading (1995)

22. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving executability into object-oriented meta-languages. In: International Conference on Model Driven Engineering Languages and Systems (MoDELS), vol. 3713 of Lecture Notes in Computer Science, pp. 264–278. Springer, Berlin (2005)

23. Combemale, B., Crégut, X., Garoche, P.-L., Thirioux, X.: Essay on semantics definition in MDE. An instrumented approach for model verification. J. Softw. **4**(9), 943–958 (2009)

24. Meyer, B.: Eiffel: The Language. Prentice-Hall, Inc., Englewood Cliffs (1992)

25. Harel, D., Rumpe, B.: Meaningful modeling: what's the semantics of "semantics"? Computer **37**(10), 64–72 (2004)

26. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Reading (2008)

27. Shonle, M., Lieberherr, K., Shah, A.: XAspects: an extensible system for domain-specific aspect languages. In: Companion of the 18th Annual ACM SIGPLAN Conference OOPSLA, pp. 28–37. ACM (2003)

28. Hundt, R.: Loop Recognition in C++/Java/Go/Scala

29. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**, 1811–1841 (1994)

30. Odersky, M.: Contracts for scala. In: First international conference on Runtime verification (RV'10), pp. 51–57. Springer, Berlin (2010)

31. Hilbert, D.: Ueber die stetige Abbildung einer Line auf ein Flächenstück. Mathematische Annalen, 38 (1891)

32. Harrison, W.H., Ossher, H., Tarr, P.L.: General composition of software artifacts. In: Software Composition (SC 2006), pp. 194–210 (2006)

33. Apel, S., Kastner, C., Lengauer, C.: Featurehouse: Language-independent, automated software composition. In: International Conference on Software Engineering (ICSE), pp. 221–231 (2009).

34. Dinkelaker, T., Eichberg, M., Mezini, M.: An architecture for composing embedded domain-specific languages. In: International Conference on Aspect-Oriented Software Development (AOSD), pp. 49–60. ACM (2010)

35. Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: 11th International Conference on Compiler Construction (CC 2002), pp. 128–142. Springer, (2002)

36. Henriques, P.R., Pereira, M.J.V., Mernik, M., Lenic, M., Gray, J., Hui, W.: Automatic generation of language-based tools using the LISA system. Softw. IEE Proc. **152**(2), 54–69 (2005)

37. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. Sci. Comput. Program. **75**(1–2), 39–54 (2010)

38. Ekman, T., Hedin, G.: The jastadd system—modular extensible compiler construction. Sci. Comput. Program. **69**, 14–26 (2007)

39. Rebernak, D., Mernik, M., Wu, H., Gray, J.: Domain-specific aspect languages for modularising crosscutting concerns in grammars. IET Softw. **3**(3), 184–200 (2009)

40. Fowler, M.: Language workbenches: The killer-app for domain specific languages. Accessed online from: http://www.martinfowler.com/articles/languageWorkbench.html (2005)

41. Voelter, M.: From programming to modeling-and back again. Softw. IEEE **28**(6), 20–25 (2011)

42. Klint, P.: A meta-environment for generating programming environments. ACM Trans. Softw. Eng. Methodol. (TOSEM) **2**(2), 176–201 (1993)

43. Cordy, J.R., Halpern, C.D., Promislow, E.: Txl: a rapid prototyping system for programming language dialects. In: International Conference on Computer Languages (ICCL), pp. 280–285 (1988)

44. Sztipanovits, J., Karsai, G.: Model-integrated computing. IEEE. Comput. **30**(4), 110–111 (1997)

45. Renggli, L., Gîrba, T., Nierstrasz, O.: Embedding languages without breaking tools. In: European Conference on Object-Oriented Programming (ECOOP 2010), vol. 6183 of Lecture Notes in Computer Science, pp. 380–404. Springer, Berlin (2010)

Author Biographies

Jean-Marc Jézéquel is a Professor at the University of Rennes and Director of IRISA, the largest public research lab in Informatics in France. His interests include model-driven software engineering for software product lines, and specifically component-based, dynamically adaptable systems with quality of service constraints, including reliability, performance and timeliness. He is the author of several books published by Addison-Wesley and of more than 200 publications in international journals and conferences. He was a member of the steering committees of the AOSD and MODELS conference series. He also served on the editorial boards of IEEE Computer, IEEE Transactions on Software Engineering, the Journal on Software and Systems, on the Journal on Software and System Modeling and the Journal of Object Technology. He received an engineering degree in Telecommunications from ENSTB in 1986 and a Ph.D. degree in Computer Science from the University of Rennes, France, in 1989.

Benoit Combemale received his Ph.D. in computer science from the University of Toulouse, France in 2008. He first worked at Inria before joining the University of Rennes 1 in 2009. He is now associate professor in software engineering at the University of Rennes 1 and evolve within both the IRISA and Inria Labs. His research interests include model-driven engineering (MDE), software language engineering (SLE) and Validation & Verification (V&V). He also teach object-oriented programming and modeling, MDE and V&V in different universities and engineering school. He is a member of the IEEE and the IEEE Computer Society.

Olivier Barais is an associate professor at the University of Rennes 1, where he is a member of the Triskell INRIA research team. He received an engineering degree from the Ecole des Mines de Douai, France in 2002 and a Ph.D. in computer science from the University of Lille 1, France in 2005, in the Jacquard INRIA research team. His research interests include component-based software design, model-driven engineering and aspect-oriented modeling. Olivier Barais has co-authored 8 journal articles and 35 international conference papers, in conferences and journals such as Sosym, IEEE Computer, ICSE, MoDELS, SPLC and CBSE.

Martin Monperrus has been an assistant professor at the University of Lille since 2011. He was previously with the Darmstadt University of Technology as a research associate. He received a Ph.D. from the University of Rennes in 2008 and a Master's degree from the Compiègne University of Technology in 2004. His current and past research activities span the fields of recommendation systems for software engineering, empirical software engineering, model-driven software development and machine learning.

François Fouquet received his Ph.D. in the Irisa/Inria laboratory at the University of Rennes 1 in 2013. His main topics of interests are the modeling tools and languages and in particular their application for distributed adaptive systems. Main contributor of the Kevoree project and its dedicated tools for modeling and code generation, Francois has also actively participate to the development of the Kermeta compiler used in this paper.