A Demonstration for Building Modular and Efficient DSLs: The Kermeta v2 Experience

Olivier Barais, Benoit Baudry, Arnaud Blouin, Benoit Combemale, Jean-Marc Jézéquel, Didier Vojtisek

A Demonstration for Building Modular and Efficient DSLs

The Kermeta *v2* Experience

Olivier Barais[1], Benoit Baudry[2], Arnaud Blouin[3], Benoit Combemale[1], Jean-Marc Jézéquel[1] and Didier Vojtisek[2]

[1] Université de Rennes 1, France
firstname.lastname@irisa.fr
[2] Inria Rennes Bretagne Atlantique, France
firstname.lastname@inria.fr
[3] INSA Rennes, France
firstname.lastname@irisa.fr

Abstract

This demonstration presents the new version (*v2*) of the Kermeta workbench that uses one domain-specific meta-language per language implementation concern. We show that the usage and combination of those meta-languages is simple and intuitive enough to deserve the term *mashup* and implemented as an original modular compilation scheme in the new version of Kermeta. This demonstration illustrates the use of the new version of Kermeta by presenting its use to design and implement two DSLs: Kompren, a DSL for designing and implementing model slicers; KCVL, the *Commun Variability Language* dedicated to variability management in software design models.

1 Introduction

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 demonstration presents the new version (*v2*) of the Kermeta workbench that uses one domain-specific meta-language per language implementation concern. We show that the usage and combination of those meta-languages is simple and intuitive enough to deserve the term *mashup* and implemented as an original modular compilation scheme in the new version of Kermeta. We also demonstrate that such a modular DSL design is not traded for efficiency at runtime.

This demonstration illustrates the use of the new version of Kermeta by presenting its use to design and implement two DSLs: Kompren, a DSL for designing and implementing model slicers; KCVL, the *Commun Variability Language* dedicated to variability management in software design models.

2 DSLs Design and Implementation with Kermeta v2

2.1 DSL Design in Kermeta

Kermeta is a language workbench designed for specifying and designing domain-specific languages (DSL). For this, it involves different *meta-languages* depending on the language imple-

mentation concern: abstract syntax (we will also use the term "metamodel" to refer to it[1]), static semantics, behavioral semantics. The workbench integrates the OMG *de facto* standards EMOF and OCL, respectively for specifying the abstract syntax (cf. Subsection 2.1.1) and the static semantics (cf. Subsection 2.1.2). The workbench also provides *Kermeta Language* to address the specification of the behavioral semantics (cf. Subsection 2.1.3).

The Kermeta Workbench also provides composition operators responsible for mashing-up these different concerns into a standalone runtime (*e.g.,* interpreter or compiler) of the DSL (cf. Subsection 2.1.4).

2.1.1 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 meta-language EMOF (Essential Meta Object Facility) [7]. 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).

2.1.2 Concern #2: Static Semantics Definition

The static semantics of a DSL is the union of the well-formedness rules on top of the abstract syntax (as invariants of domain classes) and the axiomatic semantics (as pre- and post conditions 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 run-time using assertions, depending on the execution domain of the DSL. Kermeta uses the OMG Object Constraint Language (OCL) [6] to express the static semantics, directly woven into the metamodel using the Kermeta `aspect` keyword.

2.1.3 Concern #3: Behavioral Semantics Definition

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 [5]. It can be used to define either a translational semantics or an operational semantics [2]. A translational semantics would result in a compiler while an operational semantics would result in an interpreter.

Using the Kermeta language, a behavioral semantics is expressed as methods of the classes of the abstract syntax [5]. 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 run-time such as containment and associations. First, for bidirectional associations the assignment operator semantics handles both ends of the association. Second, for compositions the assignment operator semantics unbinds existing references.

[1] This is one definition in the community. For some researchers, "metamodel" sometimes referred to abstract syntax plus static semantics.

Finally, for multiple inheritance Kermeta borrows the semantics from the Eiffel programming language [4].

2.1.4 Composition Operators for the Mashup of Meta-Languages

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. The `aspect` keyword enables DSL engineers to relate the language concerns (abstract syntax, static semantics, behavioral semantics) together: an existing class is reopened to be augmented with new methods, new properties, *etc.*

The keyword `require` enables the mashup itself. A DSL implementation *requires* an abstract syntax, a static semantics, and a behavioral semantics. Listing 1 shows how the final DSL mashup looks like in Kermeta. Three `require` are used to specify the three concerns. This mechanism is convenient to support semantic variations of the same language. For instance, Kermeta can be used to specify several implementations of UML semantics variation points.

Listing 1: Mashup of the DSL Concerns

```
1  package mydsl;
2  require "mydsl.ecore" // abstract syntax
3  require "mydsl.ocl"  // static semantics
4  require "mydsl.kmt"  // operational semantics
5  class Main {
6    operation Main(): Void is do
7      // Manipulate a model according to the DSL runtime
8    end
9  }
```

2.2 DSL Implementation in Kermeta

This section presents how we compile a mashup of three meta-languages into a single domain specific language (DSL) runtime environment. This poses a number of challenges, described in Subsection 2.2.1 that we solve as exposed in Subsection 2.2.2. The resulting code satisfies the composition semantics expressed at the meta-language level (cf. Subsection 2.2.3).

2.2.1 Challenges

We have to choose a single appropriate target language for compiling the mashup. As input, we have three different meta-languages (Ecore, OCL, Kermeta). For sake of simplicity, the target language must not be too far from the concepts and the abstraction level of our meta-languages. 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 meta-language, 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.

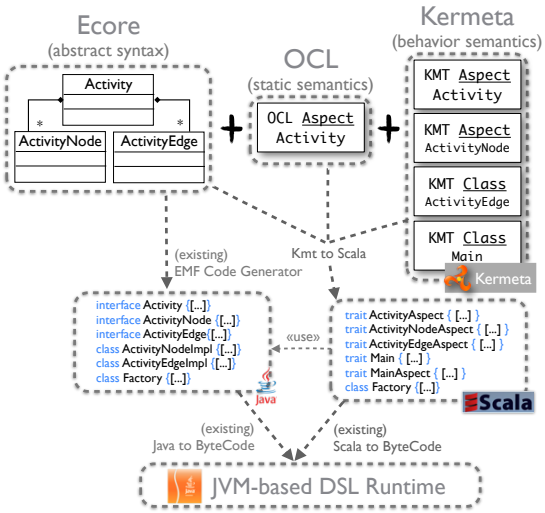

Figure 1: Java and Scala Elements Generated from Ecore, OCL and Kermeta

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.

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

2.2.3 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 Figure 1). To build the mashup at the bytecode level, the mashup compiler must plug new generated code into the generated, untouched code from EMF.

Our experiments, whose some of them are presented in the next sections, show that although our approach uses and composes different meta-languages to implement a DSL, the advantages of having a clean design and a concise implementation are not traded for run-time performance overhead. These benefits to both the design-time and run-time 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.

3 Kompren

Kompren is a DSML to model model slicers for a particular domain [1]. Model slicing is a model comprehension technique inspired by program slicing. The process of model slicing involves extracting a subset of model elements which represent a model slice. The model slice may vary depending on the intended purpose. For example, when seeking to understand a large class diagram, extracting from a class diagram a subset composed of all the dependencies of a particular class of interest may help.

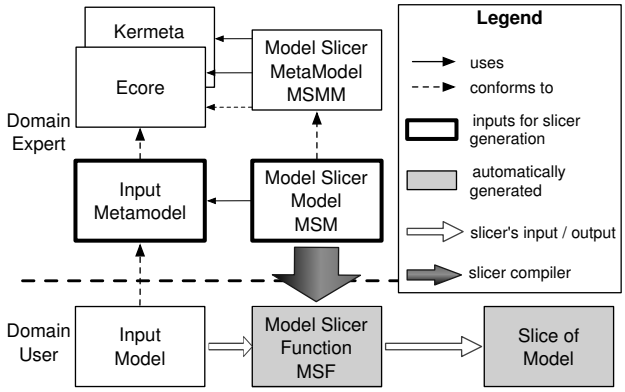

Figure 2: Overview for Modeling Model Slicers with Kompren, from [1]

In this demonstration, we explain how we tackled the three challenges previously introduced by leveraging Kermeta v2 for building the Kompren toolchain depicted in Figure 2. First, we show how we defined in a modular way the abstract syntax (MSMM), the static semantics, and the behavioral semantics (slicer compiler) of Kompren. The behavioral semantics of Kompren is illustrated by the slicer compiler transforming model slicer models into executable model slicer functions (Kermeta v2 programs) using Kermeta v2. Then, we explain how the challenges #2 (integration with legacy code) and #3 (efficiency) have been tamed through a model slicer function that has been integrated into a Java Swing application: a metamodel visualizer providing filtering features thanks to a model slicer function.

4 KCVL

To be adopted in industrial cases, the Software Product Line paradigm must be adapted to the specific organizational context and culture. In particular, SPL approaches must be suitable for companies that use a model-based software and system development process, which allows them to build reliable and consistent system. In this trend, the Common Variability Language (CVL) is a domain-independent language for specifying and resolving variability over any instance of any MOF-compliant metamodel. CVL is the result of the work to standardize a language for variability modeling in the OMG.

Also inspired by FODA, CVL mainly defined three models. Firstly it has a Variability Abstraction Model (*VAM*), which is the part of CVL in charge of expressing the variability in terms of a tree-based structure. Besides the VAM, CVL also contains a second model: Variability Realization Model (VRM). This model makes possible to specify the changes in the third model (the base model) implied by the *VSpec* resolutions. These changes are expressed as Variation Points (VPs) in the VRM. VPs can mainly express three different types of semantics, which are

following described. **Existence**. This type of VP expresses whether an object (*ObjectExistence* VP) or a link (*LinkExistence* VP) exists or not in the materialized model. **Substitution**. This type of VP expresses a substitution of a model object by another (*ObjectSubstitution* VP) or of a fragment of the model by another (*FragmentSubstitution*). **Value Assignment**. This type of VP expresses that a given value is assigned to a given slot in a base model element (*SlotAssignment*VP) or a given link is assigned to an object (*LinkAssignment* VP).

The Common Variability Language (CVL) provides a well-structured mechanism to express variability and to relate this variability to any MOF-compliant model. This characteristic allows users to define the materialization of a given CVL resolution/configuration. Using variation points, it is possible to express and manipulate the links between the variability abstraction model and the base model. However, the meaning of a given variation point can vary according to the semantics of each domain. For example, a variation point that excludes an element in the base model can lead to further operations, like excluding other elements which were associated to the deleted element, or even to reassign references to another model element. Therefore, it is necessary to address this semantic variability in order to align the materialization semantics to the base model semantics. In this demo, we demonstrate how we manage this issue in KCVL (the CVL implementation in Kermeta). We show in particular the benefits of K2 to implement and customize the semantics of the CVL's VP, according to the semantics of the base model domain. By default, CVL proposes a set of VP with a well-defined semantics and keeps one as an extension point to implement its own semantics: *Opaque Variation Point* (OVP). The OVP is a black box that can define an arbitrary behavior (in our case Kermeta or Groovy behavior) to execute during derivation. The use of OVPs can be seen as a mechanism to propose a particular semantic for the derivation engine. Besides the OVP, KCVL proposes two other mechanisms to customize the semantics of CVL. The first one is the static introduction of a new semantic following the pattern used in Kermeta to define the DSL behavioral semantics and the second one is using the strategy pattern.

5 Future Work

The mashup compiler presented throughout this demonstration will used as the basis of our future work on the usage and combination of meta-languages. We are integrating the concept of model typing into the Kermeta workbench to improve the mashup expressiveness. Furthermore, we are investigating the generation of LLVM[2] code instead of Scala code to improve the efficiency of model executions and to support various platform devices with a limited memory and CPU.

References

[1] Arnaud Blouin, Benoit Combemale, Benoit Baudry, and Olivier Beaudoux. Kompren: Modeling and generating model slicers. *Software and Systems Modeling (SoSyM)*, pages 1–17, 2012.

[2] Benoit Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification. *Journal of Software*, 2009.

[3] Robert Hundt. Loop Recognition in C++/Java/Go/Scala.

[4] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992.

[5] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *MoDELS*, volume 3713 of *LNCS*, pages 264–278. Springer, 2005.

[6] Object Management Group, Inc. *UML Object Constraint Language (OCL) 2.0 Specification*, 2003.

[7] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Core Specification*, 2006.

[2] `http://llvm.org/`