

Building a Kermeta Compiler using Scala: an Experience Report.

François Fouquet Olivier Barais Jean-Marc Jézéquel

Université de Rennes 1, INRIA Triskell project

{ffouquet,obarais,jezequel}@irisa.fr

Abstract

This paper presents an experience report of building a Kermeta compiler using Scala as a target language. Kermeta is a domain specific language inspired by languages such as Eiffel or OCL for specifying the operational semantics of meta-models. This engineering work, initially motivated by performance issues of our Kermeta interpreter, is an excuse to study and discuss some paradigm mismatches between Scala and Kermeta. We particularly discuss the mapping on Scala of Kermeta concepts :open classes, multiple inheritance, design by contracts, model type, *etc.*

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords Scala, Kermeta, Composition operators, Model Driven Engineering, Aspects

1. Introduction

How difficult is it to write a compiler? Lots of papers, books [1] or blog entries ¹ try to answer this question or try to simplify the design and implementation of a compiler. This paper is not about discussing new compiling techniques, but to take a compiler case study to work on filling the gap between two high level languages: Kermeta [9, 15] and Scala [16] that both contain structural and functional paradigms. Both languages propose high-level composition operators: Scala for improving the possibilities of Object-Oriented Design and Kermeta for simplifying language integration in the Model Driven Domain. Kermeta is a domain specific language inspired by languages such as Eiffel [8, 12] or OCL. Some of Kermeta and Scala concepts can be mapped with a one-to-one mapping, others are more complex to align. This paper studies these later ones in order to evaluate how Scala can simulate other paradigms.

¹ http://tratt.net/laurie/tech_articles/articles/how_difficult_is_it_to_write_a_compiler

The reasons why we use Scala as a target language is that we have complex dynamic object composition problems to integrate Kermeta code with Java/Emf code. Scala features help us to solve them by allowing us to design the compiler with a high-level of abstraction manipulating high-level composition operators. This compiler uses Scala rich structural possibilities to finally target Java byte-code.

The contribution of this paper is not in the domains of compiler theory, but to describe a study on how to map object-oriented paradigms that exists in Kermeta into another high level language: Scala. It also discusses and presents how we can manage open-classes [4], multiple inheritance, design by contracts [13], model type paradigms [20], *etc.* using Scala. We also use this study to give some feedback on the Scala compiler and tools.

The reminder of this paper is organized as follows. Section 2 describes an overview of Kermeta features and motivations to use Scala as a compiler target language. This choice is based on issues observed in our first compiler versions. Section 3 presents some of the paradigms used by Kermeta and how we match them with Scala constructions. Section 4 gives some the feedback on Scala usage and compiler performance. Section 5 discusses related works and section 6 concludes by presenting a set of open research problems based on our experience.

2. Motivations to use Scala for building Kermeta compiler

2.1 Kermeta overview

2.1.1 EMF and MOF

The Eclipse Modelling Framework (EMF) is a framework and code generation facility for building Java applications based on simple model definitions using eMOF like models. Designed to make modelling practical and useful to the mainstream Java programmer, EMF unifies three important technologies: Java, XML, and UML. Software is focused on manipulating data that can be modelled, hence, models drive software development.

2.1.2 Kermeta features

Kermeta is an MDE platform designed to specify constraints and operational semantics of metamodels [15]. The MOF [17] and its Eclipse implementation EMF supports the definition of metamodels in terms of packages, classes, properties and operations but it does not include concepts for the definition of constraints or operational semantics. Kermeta

extends MOF and EMF with an imperative action language for specifying constraints and operation bodies at the meta-model level. A complete description of the way Kermeta is designed can be found in [15].

The action language of Kermeta is especially designed to process models. It is imperative and includes classical control structures such as blocks, conditional and loops. Since the MOF/EMF specifies object-oriented structures (classes, properties and operations), Kermeta implements traditional object-oriented mechanisms for multiple inheritance and behaviour redefinition with a late binding semantics (to avoid multiple inheritance conflicts a simple behaviour selection mechanism is available in Kermeta). Like most modern object-oriented languages, Kermeta also provides reflection and an exception handling mechanism. In addition to object-oriented structures, the MOF contains model-specific constructions such as containment and associations. These elements require a specific semantics of the action language in order to maintain the integrity of associations and containment relations. For example, in Kermeta, the assignment of a property must handle the other end of the association if the property is part of an association and the object containers if the property is a composition.

Kermeta expressions are very similar to Object Constraint Language (OCL) expressions. In particular, Kermeta includes lexical closures similar to OCL [18] or Scala iterators on collections such as `each`, `collect`, `select` or `detect`. The standard framework of Kermeta also includes all the operations defined in the OCL standard framework. This alignment between Kermeta and OCL allows OCL constraints to be directly imported and evaluated in Kermeta. Pre-conditions and post-conditions can be defined for operations and invariants can be defined for classes. The Kermeta virtual machine has a specific execution mode, which monitors these contracts and reports any violation. One of the key features of Kermeta is the static composition operator, which allows extending an existing metamodel with new elements such as properties, operations, constraints or classes. This operator allows defining these various aspects in separate units and integrating them automatically to the metamodel. The composition is done statically and the composed model is typed-checked to ensure the safe integration of all units. This mechanism makes it easy to reuse existing metamodels or to split metamodels into reusable pieces. It also provides flexibility. For example, several operational semantics can be defined in separate units for a single metamodel and then alternatively composed depending on a particular need. This is the case for instance in the UML metamodel when several semantics variation points are defined.

The last version of the Kermeta language integrates the notion of model typing [20], which corresponds to a simple extension to object-oriented typing in a model-oriented context. This feature is detailed in Section 3.7.

The purpose of Kermeta is to remain a core platform to safely integrate all the aspects around a metamodel. As detailed in the previous paragraphs, metamodels can be expressed in MOF/EMF and constraints in OCL. Kermeta also allows importing Java classes in order to use services such as

file input/output or network communications, which are not available in the Kermeta standard framework. Kermeta and its framework remain dedicated to model processing but provide an easy integration with other languages. This is very useful for instance to make models communicate with existing Java applications.

2.2 Main issues

Currently, Kermeta works mainly through an interpreter that loads Kermeta programs and support load and save of XMI models within Eclipse. The main issue with the interpreter is efficiency. Indeed, loading huge UML models for checking constraints or running simulation takes quite a long time (>30 minutes for loading huge UML Marte [5] models and checking 250 OCL [18] constraints that corresponds to the Marte static semantics). This performance issue is the main reason for building a compiler. The second motivation is to improve the integration with Java/EMF generated code. Indeed, even if people used EMF to build their meta-models, they often modify the generated code manually. Consequently, we have to translate our open class composition operator between Kermeta and EMF model into an open class mechanism at the code level with the Java/EMF generated code.

The requirements for the compiler are the following;

- Integration with generated legacy Java code (EMF). Kermeta compiler output must be composable with generated Java/EMF code, even when it modified by hand.
- Mismatch paradigm between Kermeta and Java/Scala. Kermeta proposes closures, semantics for associations, model typing, *etc.* All these concepts should be specified using Scala and Java composition operators.
- Modular compilation. For improving compiler performance, it has to support modular compilation.
- Model lazy loading. To avoid creating too many objects, the integration with generated Java/EMF code must permit the lazy loading mechanism provided by EMF.
- OSGi Integration [21]. As EMF code is often used in the context of Eclipse plugins, the generated code should work in an OSGi runtime.
- Model Type compilation. The compiler must support the model type paradigm [20].

2.3 Compiler V1 issue

A first version of Kermeta compiler was written two years ago. Its main item was to leverage EMF tools and template engine to generate Java Code. Aspect and open-classes are flatten at compile time using the model as the unique source for body definition. Drawbacks of this architecture included performance issues and the impossibility of reusing legacy EMF generated code. We must also face to a lot of process exceptions used by EMF structure generator. So in a rational approach we had to reuse the EMF provided library instead of regenerate it from the model. The consequence of this choice is that the program composition must now be performed at the byte-code level.

3. Paradigm

In this section we present several features of Kermeta and how we map them to Scala. We detail the solution adopted to implement them using compositions of Scala constructions. Each feature is illustrated as far as possible with a sample of generated code. This section addresses EMF Integration, open-class aspect mechanism, Multiple-inheritance model, design by contracts, parametric class and model typing. Some features like model type or Parametric class are only partially solved and can open the discussion on different generative strategies.

3.1 Seamless integration with Java EMF

EMF is built around the Eclipse environment and thus around Java and OSGi [21]. EMF offers a serialisation in XMI format which does not use the Java reflexivity. Runtime structure of EMF models is then provided in a Java library. A first step integration of Scala with EMF is just to interoperate with a Java library. For that, Scala readily offers all the needed mechanisms.

Figure 1 details the Java elements generated by EMF GenModel from a simple model. For each model class, EMF generates a Java interface, a Java Concrete class and a common factory for instances creation.

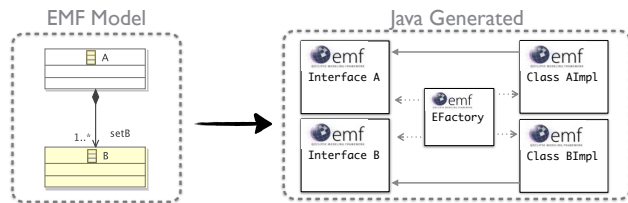


Figure 1. Details of generated Java elements for a simple EMF metamodel

The next step in EMF integration is the overload of the EMF Factory. In short, EMF implements the design pattern Abstract Factory [7] to create new instances. We simply extend and overload the generated EMF concrete factory and we inject this new dependency. Each EMF element has a dependency to its Factory through a singleton pattern: we set the static property of the singleton pattern at runtime.

The final step is the integration of closures to make it possible to seamlessly use closures on Java/Scala/EMF collections. For that implementation we reuse the Scala source code provided in the last version (2.8) to make implicit conversion from Java to Scala collections. In fact we have implemented a similar trait to make implicit conversion from EMF to Scala collection and included it into every generated trait.

3.2 Closure and lambda expression

To implement and statically type-check OCL-like iterators, Kermeta includes some limited functional features by implementing lambda expressions. This is typically used on Collections which provide functions like: each, select, forAll, detect, etc.

As an example, the following code builds a collection of names of operations that start with "test".

Listing 1. Closure in Kermeta

```
1 var names : Collection<String>
2 names := self.getMetaClass.classDefinition.
   ownedOperation
3   .select{ op | op.name.indexOf("
   test") == 0}
4   .collect{ op | op.name }
```

A user can also define his own closures to obtain a time function on Integer, as illustrated in listing.

Listing 2. Closure definition example in Kermeta

```
1 operation times(body : <Integer->Object>) :
   Void is do
2   from var i : Integer init 0
3   until i == self
4   loop
5     body(i)
6     i := i + 1
7   end
8 end
```

This allows Kermeta programmers to write code such as:

Listing 3. Closure usage in Kermeta

```
1 var res : Integer
2 10.times { i | stdio.writeln(i.toString + "
   : Hello") } // Say 10 times Hello
```

The mapping onto Scala is direct. For example, the *times* function declared for Integer can be translated into a lambda declaration in Scala.

Listing 4. Generated Closure definition example in Scala

```
1 class RichInteger(value: Int) extends
   RichNumeric[Int] {
2   def times(body : Int => Unit):Unit = { for
   (i <- 0 until value){body(i)} }
3 }
```

3.3 Open Class Aspects

This section describes how we build an Open Class mechanism using traits and mixins in Scala. After a brief reminder of open-classes, we explain its Kermeta usage before detailing how we compile it into Scala to obtain a modular aspect composition at runtime. This solution works even in a dynamic environment like OSGi.

3.3.1 Open Class Mechanism Overview

The concept of Open Class allows designers to make static introduction in a previously created class by reopening it. Open Class allows him to directly add new methods or new attributes in a class without creating distinct subclasses or editing existing code. Unlike the "Visitor" design pattern, Open classes do not require advance planning and preserve the ability to add new subclasses modularly and safely. Open Class can be then compared to inter type declaration in AspectJ.

3.3.2 Kermeta Open Class usage

Open Classes are a central feature of Kermeta, because they permit to reuse and enrich types declared in models. Indeed, in most cases, models are built with domain analysis and specified using a modelling editor. Behaviour code is later integrated through aspects, which are statically introduced at relevant places to get executable models. Note also that, Open classes must be transient for models manipulated by Kermeta. Consequently, if a designer adds an attribute using open class mechanism in Kermeta, the value should not be persistent when the model is stored in an XMI file.

3.3.3 Java compiler problems and limitations

A first version of the compiler was built using the EMF template mechanism to generate Java code. In this generation, the template mechanism injects Kermeta behaviour into EMF structure. This solution permits to flatten aspects at compile-time considering the model as the unique data source. The main limitation of this mechanism is that we cannot reuse legacy EMF code. In particular if it was previously generated by EMF and/or hand-written. A weird as it looks from MDE perspective, this scenario arises frequently, even into the EMF ECore² model library which compiled version cannot be generated from its model due to a specific post generation process. We must thus compile Open Classes in a different way for being able to add behaviour aspects to legacy structural code more dynamically, in an OSGi environment.

3.3.4 Compiling Open Class

Several languages are available to generate JVM compatible byte-code. Each approach provides a different way to re-open classes. Let's consider how we could add a "addedMethod" to the String class in several languages:

- Groovy has a reflexivity layer which permits to dynamically extend a meta-class.

Listing 5. Groovy open class mechanism

```
1 String.metaClass.addedMethod = { ->
    return delegate.toString() }
```

- JRuby is a Java wrapper for the Ruby language. Ruby classes are never closed, in fact we can extend them at any time.

Listing 6. JRuby open class mechanism

```
1 class String
2   def addedMethod
3     self.toString
4   end
5 end
```

- MultiJava is an extension to the Java programming language that adds an open class mechanism. But it seems that development is stopped since 2006.
- Scala provide a Trait feature with a Mixin operator. One goal of this paper is to provide an open class mechanism using a composition of them.

² ecore is the language used to define EMF metamodel

3.3.5 Compiler V2 Target solution

In this section we describe our solution for Kermeta compiler using Scala as a target language. In addition behaviour Open class aspects must be applied at model load-time for being compatible with a lazy loading strategy which is really useful for huge model and OSGi environment.

First step is the use of the design pattern Abstract Factory implemented by EMF for overriding the creation of instances. In the generated factory, we use Scala mixins to compose a structural class with behaviour trait.

As a second step, we process structural classes definitions in an homogeneous way.

If a class is defined in a model, EMF generates an interface and an a concrete implementation. Then we mix it with the behaviour trait. In case of class definition coming from Kermeta, we create an interface trait to make for lack of EMF interface and we mix it with the default EMF object class implementation *EObjectImpl* and the behaviour trait.

For being totally transparent when calling base and aspect methods, we also generate a global conversion named *ImplicitConversion* trait which declares an implicit conversion method for converting base to aspect and aspect to base as illustrated in Listing 8. With this mechanism, we can use any object transparently with aspects without the need for explicit cast.

Generated aspects can then access base classes using the "self" or "this" pointer which is usable after class linearisation and implicit conversions (see Listing 8). Another possible solution for converting aspect to base, is to use the self-type solution offered by Scala, but the drawback is that it is not working outside the aspect. This notation has a limited scope. In fact, we could use both solutions, self-typing for most cases and implicit conversion for outside call.

As a summary, for each meta-class defined in a Ecore meta-model, we compile it into a dynamically mixed object which uses:

- A structural legacy EMF interface or a generated structural trait if the class is a pure Kermeta class (class A & B comes from EMF and class C is a pure Kermeta class in Listing 8 and in Figure 2).
- A structural legacy EMF implementation class which extends *EObjectImpl*,
- A behaviour Scala aspect trait,
- An optional contract Scala aspect trait,
- Two inputs in a global Implicit Conversion trait (aspect to base and base to aspect),
- One input in a generated EMF Factory for creating new "dynamic" anonymous object using ordered linearisation.

With this mechanism, we have a systematic solution for each step:

- At compile time, double implicit conversion permits to type check the code.
- At runtime, self pointer is linearised and so can access the right byte-code.

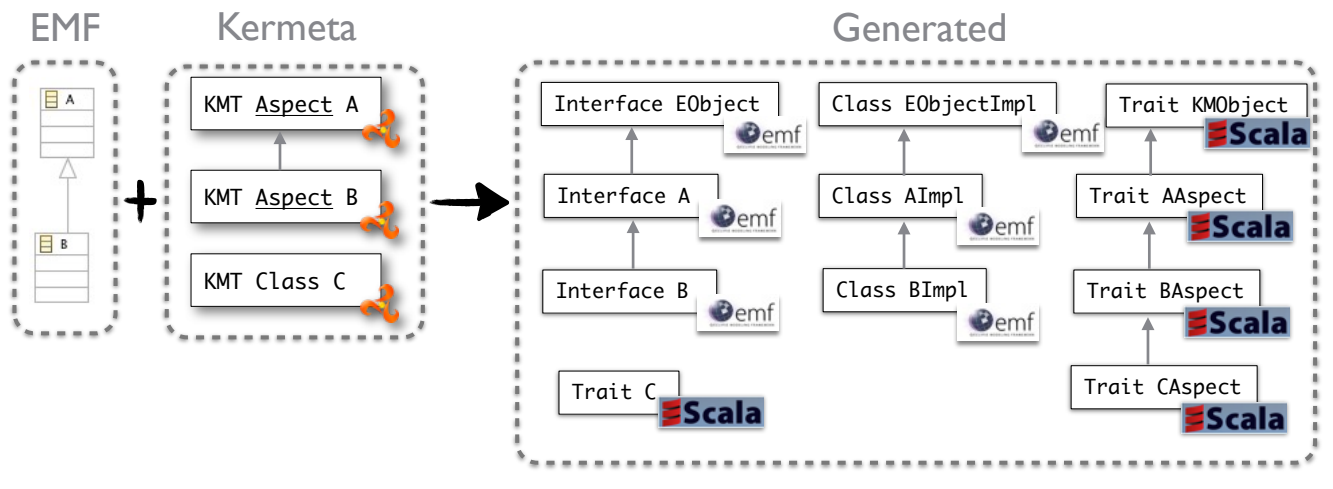


Figure 2. Open Class in Scala

We limit the overhead introduced by implicit conversion by limiting the use of this method to explicitly casts and exclude object creation.

3.3.6 Compilation example

We can illustrate our base mechanism with a simple example. In Listing 7, we write a little Kermeta model which uses an ECore model: A.ecore. This ECore model only describes a A class with a name attribute. Then a Kermeta program adds a method to A using the open class aspect, and also declares a pure Kermeta class: B.

Listing 7. Compilation example

```

1 require kermeta
2 require platform:/resource/A.ecore
3 class B
4 {
5   operation newOp() : Void is do
6     stdio.println("test")
7   end
8 }
9 aspect class A {
10  operation newOp() : Void is do
11    stdio.println(self.name+"test")
12  end
13 }

```

From A.ecore, EMF GenModel generates: an interface named A and a class named B. Kermeta Compiler does not regenerate them and simply imports them. EMF GenModel also generates a base EMF factory implementation. In addition, our Kermeta Scala Compiler generates aspect traits, implicit conversion traits and a new factory that extends the generated EMF Factory to load class with mixins.

Hereafter follows a sample code snippet which details the generated elements. Important elements there are the way we inject ImplicitConversion and the way we handle aspect mixin instance declaration.

Listing 8. Example of converting open classes mechanism using Traits and Mixin

```

1 trait B extends kermeta.framework.Object

```

```

2 trait BAspect extends kermeta.framework.
   ObjectAspect with ImplicitConversion{
3   def newOp() = { println("test") }
4 }
5 trait AAspect extends kermeta.framework.
   ObjectAspect with ImplicitConversion{
6   def newOp() = { println(this.name+"test")
7   }
8 }
9 object RichFactory extends FactoryImpl with
   ImplicitConversion{
10  override def createA : A = {new AImpl
11    with A with AAspect}
12  def createB : B = {new EObjectImpl with B
13    with BAspect}
14 }
15 trait ImplicitConversion {
16  implicit def rich(v : A) = v.asInstanceOf
17    [AAspect]
18  implicit def rich(v : AAspect) = v.
19    asInstanceOf[A]
20  implicit def rich(v : B) = v.asInstanceOf
21    [BAspect]
22  implicit def rich(v : BAspect) = v.
23    asInstanceOf[B]
24 }

```

3.4 Multiple inheritance

This subsection describes our solution to implement the Kermeta multiple inheritance model using Scala Traits and Mixins. This solution can be extrapolated to any Eiffel inspired multiple inheritance model.

3.4.1 Multiple inheritance in Kermeta

The Kermeta language implements multiple inheritance, but forbids two methods from having same name: in case of a name clash the programmer is forced to specify an explicit renaming. With Kermeta, developers must choose only one of super methods as the target of polymorphic dispatches.

3.4.2 Scala Trait to implement multiple inheritance

We have already write that each class of a metamodel (or from a Kermeta program) is compiled into 4 layers:EMF

Java interface, Java class implementation, Traits and Mixin. This way, a metamodel multiple-inheritance generates:

- Structural Object Interface multiple-inheritance level actually offered by EMF or Scala Trait,
- Structural Object implementation Multi-inheritance level actually offered by EMF or Scala Trait,
- Behaviour multiple-inheritance aspect level offered by Scala Trait,
- Contract multiple-inheritance aspect level offered by Scala Trait.

For choosing in which branch we want to call a given method, we use the *super[superClassName]* notation. It allows to call the correct method.

Contract level is an exception, we have chosen to flatten inherited contracts at compile-time to perform optimisations.

3.4.3 Diamond inheritance compilation example

We can illustrate our solution with the well know problem of diamond inheritance as illustrated in Figure 3. Since in this case, we are not using an ecore input metamodel, we build the model structure from scratch in pure Kermeta. In the result each *view* in extending the *EObjectImpl* base implementation, each structure trait in extending *KermetaObject*.

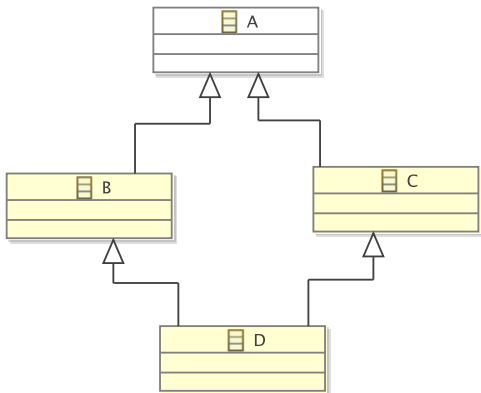


Figure 3. Example of diamond inheritance

In Kermeta we declare a *getName* and *getForName* method as follow for each element of the diamond. For resolving conflicts for D class we use the following Kermeta program to choose a branch for *getName* super method and another branch for *getForName* super method as illustrated in the code snippet of Listing 9.

Listing 9. Example of solving Diamond Problem in Kermeta

```

1 class A{
2   operation getName():String is do
3     result := "A"
4   end
5   operation getForName():String is do
6     result := "A"
7   end
8 }
9 class C inherits A{

```

```

10 method getName():String is do
11   result := super() + "C"
12 end
13 method getForName():String is do
14   result := super() + "C"
15 end
16 }
17 class B inherits A{
18 method getName():String is do
19   result := super() + "B"
20 end
21 method getForName():String is do
22   result := super() + "B"
23 end
24 }
25 class D inherits C,B{
26 method getName():String from B is do
27   result := super() + "D"
28 end
29 method getForName():String from C is do
30   result := super() + "D"
31 end
32 }

```

This Kermeta model can now be compiled into several Scala traits organised as shown in Figure 4:

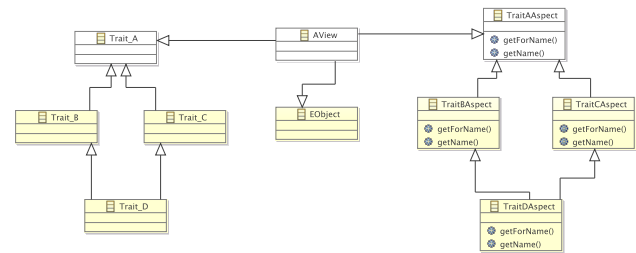


Figure 4. Diamond Inheritance Solution in Scala

Listing 10 details the generated Scala code for the *DAspect* trait using *super* to choose super methods that have to be invoked. Other generated Scala code is quite similar with the precedent Open Classes case study. The main point is the usage of *super* call for choosing the *super* trait to solve the ambiguity.

Listing 10. Diamond Inheritance Solution in Scala

```

1 trait DAspect extends CAspect with BAspect
2   with ImplicitConversion {
3   override def getForName() = super[
4     CAspect].getForName()+"D"
5   override def getName() = super[BAspect
6     ].getName()+"D"
7 }

```

3.5 Design by Contrat

Kermeta is a specific language for meta-model manipulation. This domain leverages design by contract with a dedicated language such as OCL [18]. This section describes the support of Kermeta for OCL notation and the solution to compile it to Scala in an elegant way using Closure, and Trait, to support a modular compilation.

3.5.1 Kermeta usage and integration with OCL

Kermeta contract model is built around the Eiffel Design by Contract approach [13] which has been developed by

Bertrand Meyer for about twenty years. In short, this approach uses class invariants and method pre and post conditions to specify contracts; and contracts are inherited in subclass, there is just an exception for class invariant evaluation between the Kermeta and the Eiffel model. In Kermeta, programmers must explicitly ask for invariant evaluation by calling the method `checkInvariants`, whereas in the Eiffel model, invariants are checked before and after a method call.

For supporting the OCL notation, in a pre-compilation step, the Kermeta merger converts OCL notation into Kermeta contracts and merges them. We obtain a merged Kermeta model which contains the contracts, to compile in a separated step.

3.5.2 Contracts as transversal aspects

Considering design by contract as a transversal aspect and using our previously presented dynamic aspects weaving, we can now compile contracts with modularity. In fact we just have to compile then into a separate aspect trait. With this mechanism we do not have to recompile all structural and behaviour codes. This compilation saved time is important in case of huge metamodels(e.g UML with profiles).

Although Scala does not provide explicit support for Design by Contract. Using *require* and *ensure* functions, easy to built it with Closure and Function Types. For invariants, we simply define a class `Constraint` that hosts a field named `body` whose type is `() => Boolean`. Each Kermeta Class is now generated with a new method returning the flat set of class invariants.

Listing 11. Function `getInvariants` in Scala

```
1 def getInvariants : immutable.List(
  Constraint) = List(()=>{if (cond) return
  result})
```

We choose to flatten invariant inheritance at compile-time to perform optimisations. So the new `getInvariants` method returns the optimised immutable list of `Constraints`. The evaluation method of these invariants is named `CheckInvariant`. It is offered by Kermeta Framework through the inheritance of `KermetaObject`.

For *Pre* and *Post* conditions, we generate something similar around contracted methods using interceptor Traits as we would do using `AspectJ`. Again we choose to flatten the set of pre and post conditions to perform at compile-time optimisations instead of using contract inheritance. When flattening inherited the pre and post conditions we add `&&` and `||` boolean operators to respect covariance and contra-variance semantics of contract inheritance [13].

3.5.3 Using a composition of contract aspect with multiple-inheritance

We illustrate the composition of contract aspects in a simple case study. Considering model containing *Restaurant* and *Hotel Restaurant* classes with a capacity property and invariants based on them. Listing 12 is the Kermeta code that declares model. It also directly declares contracts using Kermeta Keywords *inv* and *pre post*.

Listing 12. DbC Case Study in Kermeta

```
1 class RESTAURANT{
2   var restaurant_capacity : Integer init
   100
3   var restaurant_busy : Integer init 0
4   inv RestaurantNeverFull is do
       restaurant_busy < restaurant_capacity
       end
5
6   operation book():Void is
7     pre atLeastOneEmpty is restaurant_busy <
       restaurant_capacity
8     post notEmpty is restaurant_busy > 0
9     do
10      restaurant_busy := restaurant_busy + 1
11    end
12 }
13 class HOTEL_RESTAURANT inherits HOTEL{
14   var hotel_capacity : Integer init 100
15   var hotel_busy : Integer init 0
16   inv HotelNeverFull is do hotel_busy <
       hotel_capacity end
17
18   operation book():Void is
19     pre atLeastOneEmpty is hotel_busy <
       hotel_capacity
20     post notEmpty is hotel_busy > 0
21     do
22       if (hotel_busy < hotel_capacity) then
23         hotel_busy := hotel_busy + 1
24       if (restaurant_busy <
25         restaurant_capacity) then
26         restaurant_busy :=
27         restaurant_capacity + 1
28     end
29 }
30 }
```

The code snippet 13 illustrates the result of the compilation to Scala with invariants and pre and post conditions flattened inheritance.

Listing 13. DbC Case Study in generated Scala

```
1 case class Constraint(body : ()=>Boolean) {
2   def check = body()}
3 trait RESTAURANTAspect extends ObjectAspect
4   with ImplicitConversion {
5     var restaurant_capacity : Int = 100
6     var restaurant_busy : Int = 0
7     def book = { restaurant_busy =
8       restaurant_busy + 1 }
9   }
10 trait HOTEL_RESTAURANTAspect extends
11   RESTAURANTAspect with
12   ImplicitConversion {
13     var hotel_capacity : Int = 100
14     var hotel_busy : Int = 0
15     override def book = {
16       if (hotel_busy < hotel_capacity) then
17         hotel_busy := hotel_busy + 1
18       if (restaurant_busy <
19         restaurant_capacity) then super.
20         book
21     }
22 }
23 trait RESTAURANTContract extends RESTAURANT
24   with RESTAURANTAspect with
25   ImplicitConversion {
26     override def getInvariants() : List(
27     Constraint) = List(Constraint(()=>{
28     restaurant_busy < restaurant_capacity
29     }))
30     override def book = {
```

```

18     val precondition = List(Constraint(()
19         =>{restaurant_busy <
20         restaurant_capacity}))
21     val postCondition = List(Constraint(()
22         =>{restaurant_busy > 0}))
23     if(!preCondition.forAll.check) throw
24         new ConstraintViolatedException
25     super.book
26     if(!postCondition.forAll.check) throw
27         new ConstraintViolatedException
28 }
29 }
30 trait HOTEL_RESTAURANTContract extends
31     HOTEL with HOTELAspect with
32     ImplicitConversion {
33     override def getInvariants() : List(
34         Constraint) = List(Constraint(()=>{
35         restaurant_busy < restaurant_capacity
36         }),Constraint(()=>{hotel_busy <
37         hotel_capacity}))
38     override def book = {
39         val precondition = List(Constraint(()
40             =>{hotel_busy < hotel_capacity}))
41         val heritedPreCondition = List(
42             Constraint(()=>{restaurant_busy <
43             restaurant_capacity}))
44         val postCondition = List(Constraint(()
45             =>{hotel_busy > 0}))
46         val heritedPostCondition = List(
47             Constraint(()=>{restaurant_busy >
48             0}))
49         if(!(preCondition.forAll.check||
50             heritedPreCondition.forAll.check))
51             throw new
52             ConstraintViolatedException
53         super.book
54         if(!(postCondition.forAll.check&&
55             heritedPostCondition.forAll.check)
56             ) throw new
57             ConstraintViolatedException
58     }
59 }

```

This solution might not be the most elegant one but it permits to perform optimisations at compile time. It has no drawback on execution time. Solution would have been is to use multiple-inheritance to inherit contracts too. But the drawback of this solution is the problem for pre and post condition inheritance semantics. We can declare a stronger precondition than the inherited one when we override a method. So here we declare the *HOTELRESTAURANT* Contract book method pre-condition as a conditional separated by `||` operator. Conversely post condition must be stronger and has a `&&` operator.

3.6 Genericity

One of the core characteristics of Kermeta is to be statically typed. In order to allow static typing of OCL-like expressions, a few modifications have been made to the EMOF type system (Please refer to paper [15]). As a result to these modifications genericity support has been added into Kermeta. Like Eiffel, Java (since version 5) or Scala, Kermeta supports generic classes and generic operations. This section gives an overview of this concept in Kermeta. Since it is less powerful than Scala genericity, it is directly mapped onto Scala genericity.

3.6.1 Generic classes

A Kermeta class can have a set of type parameters. These type variables can be used in the implementation of the class as any other type. By default a type variable can take as value any type; but a type variable can also be constrained by a type: in that case, this type variable can only be substituted by a sub-type of this type. The following code demonstrates how to create generic classes.

Listing 14. Generic classes in Kermeta

```

1 // A class with a type variable G that can
2 // be bound with any type
3 class Queue<G>
4 {
5     reference elements : oset G[*]
6     operation enqueue(e : G) : Void is do
7         elements.add(e)
8     end
9     operation dequeue() : G is do
10        result := elements.first
11        elements.removeAt(0)
12    end
13 }
14 // A class with a type variable C that can
15 // be bound with any sub-type of
16 // Comparable
17 class SortedQueue<C : Comparable> inherits
18     Queue<C>
19 {
20     method enqueue(e : C) : Void is do
21         var i : Integer
22         from i := 0
23         until i == elements.size or e >
24             elements.elementAt(i)
25         loop
26             i := i + 1
27         end
28         elements.addAt(i, e)
29     end
30 }

```

3.6.2 Generic operations

Kermeta operations can contain type parameters. Like type variables for classes these type parameters can be constrained by a super type. However, unlike for classes, for which the bindings to these type parameters are explicit, for operations the actual type to bind to the variable is statically inferred for each call according to the type of the actual parameters.

Listing 15. Generic operations in Kermeta

```

1 class Utils {
2     operation max<T : Comparable>(a : T, b
3         : T) : T is do
4         result := if a > b then a else b
5     end
6 }

```

3.6.3 Mapping on Scala

Scala has a built-in support for generic classes which is really close to the Kermeta mechanism and more expressive

than the Kermeta one (Bounds are less expressive in Kermeta). Scala classes can be parametrized by an explicit type which can be used everywhere into class definition. Here is the generated Scala version of the Listing 14

Listing 16. Generic classes in Scala

```

1  trait QueueAspect[G] {
2    var elements : List[G] = Nil
3    def enqueue(e : G) = elements.add(e)
4    def dequeue : G = {
5      var result : G = null.asInstanceOf[G]
6      {
7        result = elements.first
8        elements.removeAt(0)
9      }
10     result
11   }
12 }

```

Scala methods can be parametrized too, and as in Kermeta the type can be explicit or inferred by parameter type. This parameter type can be constrained with a bound type. Here is the Scala version of the Listing 15

Listing 17. Generic operations in Scala

```

1  trait Utils {
2    def max[T <: Comparable](a : T, b : T){
3      var result : T = if(a > b) { a } else {
4        b }
5      result
6    }
7  }

```

3.6.4 Open issue

There is just one exception for this generic class one-to-one mapping. Kermeta permits to call some methods on a type and especially the "new" method to create new instance. It seems that Scala type variable does not offer this mechanism, we manage this special case by using the Scala reflexivity layer to create instance using type name. This solution is not ideal and we plan to study a mechanism similar to companion class for generic types.

3.7 Model Typing

3.7.1 Model Typing

Model typing can be related to structural typing found in several languages including Scala. Indeed, a model typing is a strategy for typing models as collections of interconnected objects while preserving type conformance, used as a criterion of substitutability.

The notion of model type conformance (or substitutability) has been adapted and extended to model types based on Bruce's notion of type group matching [2]. The matching relation, denoted $<\#$, between two metamodels defines a function of the set of classes they contain according to the following definition:

Metamodel M' matches another metamodel M (denoted $M' <\# M$) iff for each class C in M , there is one and only one corresponding class or subclass C' in M' such that every property p and operation op in $M.C$ matches in $M'.C'$ respectively with a property p'

and an operation op' with parameters of the same type as in $M.C$.

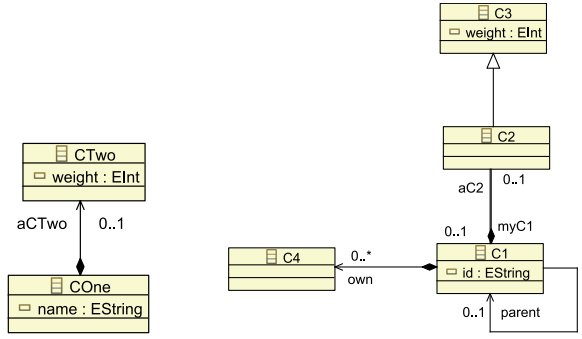


Figure 5. Metamodel M . **Figure 6.** Metamodel M' .

This definition is adapted from [20] and improved here by relaxing two strong constraints. First, the constraint related to the name-dependent conformance on properties and operations was relaxed by enabling their renaming. The second constraint related to the strict structural conformance was relaxed by extending the matching to subclasses.

Let's illustrate model typing with two metamodels M and M' given in Figures 5 and 6. These two metamodels have model elements that have different names and the metamodel M' has additional elements compared to the metamodel M .

$C1 <\# COne$ because for each property $COne.p$ of type D (namely, $COne.name$ and $COne.aCTwo$), there is a matching property $C1.q$ of type D' (namely, $C1.id$ and $C1.aC2$), such that $D' <\# D$.

Thus, $C1 <\# COne$ requires $D' <\# D$, which is true because:

- $COne.name$ and $C1.id$ are both of type *String*.
- $COne.aCTwo$ is of type $CTwo$ and $C1.aC2$ is of type $C2$, so $C1 <\# COne$ requires $C2 <\# CTwo$ or that a subclass of $C2$ matches $CTwo$. Only $C3 <\# CTwo$ is true because $CTwo.element$ and $C3.elem$ are both of type *String*.

Thus, matching between classes may depend on the matching of their related dependent classes. As a consequence, the dependencies involved when evaluating model type matching are heavily cyclical [19]. The interested reader can find in [19] the details of matching rules used for model types.

However, model typing with the mechanisms of renaming and inheritance is not sufficient for matching metamodels that are structurally different. We can overcome this limitation of the model typing using aspect weaving [14].

3.7.2 Compiling model type

Model type compilation is probably the most difficult part of our Kermeta Compiler. The first version of the Kermeta compiler does not support this feature. But we have now

some new elements to implement it in compiler V2 using Scala traits and implicits. The main problem for the model type compilation is to add enough information at compile time to make dynamic adaptation at runtime. Another problem is that a model is not declared as a subtype of another. This relationship is computed. Currently, the Kermeta type checker provides a trace of the binding it has found between the model type and a meta-model when it performs the type check. We build a solution using a generation of wrapper and implicit conversion to adapt a model object graph to respect another one using various mapping strategies. We could use several solutions to compile model types using Scala. Some discussions are still opened on the best way to implement model type as wrappers on enclosed classes or considering models as object. We provide in the next subsection the main ideas of the current solution.

3.7.3 Model Type compilation example

This subsection details a solution based on the example presented as an introduction to Model Type. This section simulates model types at runtime using Scala implicit conversion and Trait. In this solution, we consider the model type at runtime as a set of wrappers to adapt from the subtype (meta-model) to the super type (model type) implicitly. A model type is defined by a set of interfaces which defines the scope of visibility for a generic usage of it. Let's detail it with code snippet of the Scala version of previously presented Meta-Model *M* definition as a Model Type (see Figure 5).

Listing 18. Model Type Definition

```

1 package mt
2 trait C0ne {
3   var name : String
4   def aCTwo : List[CTwo]
5 }
6 trait CTwo { var weight : String }
```

We can now detail the relevant elements of metamodel *M'* (see Figure 6). The generated Scala code for this metamodel is presented in Listing 19

Listing 19. Model Type Subtype definition

```

1 package mm
2 trait C1 {
3   var id : String
4   def aC2 : java.util.List[C2]
5 }
6 trait C3 { var weight : String }
7 trait C2 extends C3
```

Model Type named *MM* can not directly match as a subtype of Model Type named *MT*. In Kermeta, we have to declare an adaptation as explain in [14] and compile it into specialized implicit conversion and rich type as illustrated in Listing 20. As we can not currently find a good solution using Scala generic bounds, the generic list resulting from association ends with multiplicity in *MT* has to be converted programatically as shown in lines 13-14 in Listing 20. This solution has to be improved in the next version of the compiler.

Listing 20. Model Type Subtype definition

```

1 package mtbinding
2 import mt._
3 import mm._
4 import mtbinding.ImplicitConversion._
5 import scala.collection.JavaConversions._
6 object ImplicitConversion {
7   implicit def richC1(x : C1) = new RichC1(x)
8   implicit def richC2(x : C2) = new RichC2(x)
9 }
10 class RichC1(var self : C1) extends C0ne
11   with scala.Proxy {
12   def aCTwo : java.util.List[CTwo] = {
13     var res : java.util.List[CTwo] = new
14     java.util.ArrayList[CTwo]()
15     self.aC2.foreach(att => res.add(att))
16     res
17   }
18 }
19 class RichC2(var self : CTwo) extends CTwo
20   with scala.Proxy
```

Every method that works on models which metamodel *MT* is reusable with any model conforming to metamodel *MM*. At runtime any model built around the structure *C1,C2,C3* is now usable with the generic method which use *C0ne,CTwo*. Then, the following *genericMMFunction* can be used on an instance of *C1*.

Listing 21. Model Type usage

```

1 import mtbinding.ImplicitConversion._
2 def genericMMFunction(c : C0ne){ c.aC2.
3   foreach(c2 => println(c2.weight)) }
```

4. Discussion

This section gives some feedback on our usage of Scala and discusses compiler performance.

4.1 About performance

Our version of the Kermeta compiler is still in an incubator phase so detailed benchmarks haven't been performed yet. However, we have already collocated some data. Indeed, with the previous Java version of the compiler, most test models took several minutes to be processed. On the current Scala version, the average compile time (Kermeta to Scala compiler written in Scala) is between 70 and 150 ms on a Core 2 duo, 3 GHz processor. This reduced compile time obtained by the use of parallel treatments and modular compilation (no regeneration of EMF structure for each compilation).

4.2 Scala Tools and integration with Enterprise development

One of the main problems for a compiler is the testing phase. Even if the designer defines rewriting rules with a formal specification, an important test effort has to be done to obtain a valid solution. It was really important then in our development to adopt an adapted environment for a large scale software. Scala provides Maven tools that really answer that needs. The interaction with continuous Integration Environment is built-in. In this work, we can couple Scala with

Maven and Hudson to test the output of the compiler. The oracle compares the output of the Kermeta test case programs as run by the Kermeta Interpreter and the execution output of the generated Scala code.

4.3 Flatten contract @Compile-Time or use inheritance model

As we said, the flattening of the contracts at compile-time can be replaced by an inheritance of methods that delivers contract. Main differences here are the memory usage compared to the CPU usage at runtime required to check contracts. The flattening of the contracts takes more memory because it forces to duplicate some contracts declaration in each inheritance step. The experience feedback we got on several huge meta-model with lot of OCL constraints, we choose to flatten and consume less CPU at runtime. However, a better strategy could be found, using both methods depending on the use-cases.

4.4 EMF dependency

We are working to implement the Kermeta reflexivity layer with the usage of Scala reflexivity layer instead of using the EMF reflexivity layer. Two reasons justify this choice: First, we have performance issues using the EMF reflexivity layer, the Scala reflexivity offers better performances, it also provides a better flexibility to flatten some attributes at compile-time in order to perform some optimisations. Besides, we can remove a part of our dependency to EMF structure. As a result, Kermeta Scala Compiler usage of EMF is limited to serialisation. If for any reasons we have to change the language used to define the meta-model structure, for example using POJO objects or XML Schemas, we would just have to write or generate the load/save strategy.

4.5 Implicit conversion collision

In some cases, we observe a collision problem on the implicit conversion generated. That is, for encapsulation reasons, we use several different interfaces for objects, we can then make several conversions for one object. By default in Scala implicit definitions have no order³. For solving this problem we need a strict order. Here the solution is to use one implicit conversion for a type and make an ordered object pattern matching on subtype to choose which conversion must be applied. Once again, Scala features permit to build a paradigm with another.

Listing 22 is a code snippet that clarifies the problem, *KermetaObject* is a subtype of *Object* and there is a conflict on implicit conversions.

Listing 22. Implicit conversion collision

```
1 trait ImplicitConversion {
2   implicit def rich(o : Object) = ...
3   implicit def rich(o : KermetaObject) =
4     ...
}
```

³One reviewer told us that in Scala 2.8, implicits can be prioritised, where the ordering is determined by the same rules as static method overloading, we have to study this solution

Then, we detail in Listing 23 the solution with Scala Pattern matching. Guard conditions are optional but they allow us to improve the precision.

Listing 23. Implicit conversion collision solution

```
1 trait ImplicitConversion {
2   implicit def rich(o : Object) = o match {
3     case KermetaObject = ...
4     case Object if (guardCond) = ...
5   }
6 }
```

4.6 Implicit conversion problems in sub scope closure, generic with bound or super notation

We have observed some problems using implicit conversion systematically. In fact when using super type notation, implicit conversions are not applied in the current 2.8 compiler⁴ and the 2.7 compiler as illustrated in Listing 24.

Listing 24. Implicit conversion problem

```
1 super[superTrait].someMethodUsingImplicit()
```

Another use case that has similar problems is the use of closure that cannot access to implicit conversion. Listing 25 is a code snippet that details generic bound problem with implicit.

Listing 25. Another implicit conversion problem

```
1 trait test{
2   def genTest[A <: Object](a : A){
3     a.useSomeMethodByImplicit
4   }
5 }
```

5. Related Work

The languages presented in section 3.3 (Ruby, Groovy, MultiJava) provide some mechanisms to implement open class paradigm. AspectJ [10, 11] static introduction can also be an alternative for implementing open classes and simplify EMF introduction. This framework allows to introduce new attributes or new methods at compiled-time using a cross-cutting instruction. This mechanism permits to modify produced byte-code. It would be another solution as a target language for the Kermeta compiler. However, we would still have to reimplement closures as an alternative Scala provides some interesting composition operators, that are promising for compiling and playing with the model type paradigm.

Another solution to compile the open-class paradigm and simplify the Java/EMF integration is to use a byte code modifier library. There are some mature tools to manipulate Java byte-code. One of them is ASM Java byte-code engineering library [3]. This library allows programmers to dynamically modify existing classes or dynamically generate classes. By its low-level character, this solution is expressive and permits to realize every use cases. By the way it is used by many high level frameworks such as JPA implementations or AspectJ. The main drawback is that it requires to manipulate

⁴January 2010 version

byte-code directly and so requires good knowledge of the in Java Byte Code structure.

6. Conclusion

In the context of Model Driven Engineering, modularity and composition aspects of languages is still a research challenge. Kermeta and related works like Kompose⁵ [6] or Model typing define high-level model composition operators for design time. We can find several solutions to precisely define the semantics of these operators. Defining them using a mapping to Scala composition mechanisms is for us an elegant solution because it allows us to reason at a high-level of abstraction using expressive object composition operators provided by Scala. It is also an efficient solution because it provides a technical solution to translate composition operators used at design time into an executable code without dealing with technical integration with Java byte code.

To conclude this experience report, Scala is really a good candidate for many reasons. First this language implements interesting paradigms such as traits, implicit, mixin or multiple-inheritance support using traits. Sometimes, we need to mix them to build complex mapping between some Kermeta features and Scala features, but it is always possible to find a solution. Secondly, Scala leverage a long Object-Oriented and functional programming experience. Consequently, lots of Kermeta features that are not detailed in this paper can be mapped directly to Scala features (Genericity, Reflexivity layer, Exceptions, *etc.*). Lastly, from our experience on this project, the Scala compiler is really a mature solution, generated byte-code is really effective and predictable even with usage of dynamic solutions like OSGi platforms and even if you combine high-level feature like implicit, trait, mixin and closure. Consequently, it was a real pleasure to define the semantics of Kermeta features using high-level Object Oriented composition operators without managing technical Java byte code integration problems.

References

- [1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2 edition, August 2006. ISBN 0321486811.
- [2] K. B. Bruce and J. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. *Electronic Notes in Theoretical Computer Science*, 20:50–75, 1999.
- [3] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.
- [4] C. Clifton and G. T. Leavens. Multijava: Modular open classes and symmetric multiple dispatch for java. In *In OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, 2000.
- [5] S. Demathieu, F. Thomas, C. André, S. Gérard, and F. Terrier. First experiments using the uml profile for marte. In *ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 50–57, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3132-8. doi: <http://dx.doi.org/10.1109/ISORC.2008.36>.
- [6] F. Fleurey, B. Baudry, R. France, and S. Ghosh. A generic approach for automatic model composition. In *Aspect Oriented Modeling (AOM) Workshop*, Nashville, USA, octobre 2007.
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
- [8] J.-M. Jézéquel. *Object-oriented software engineering with Eiffel*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996. ISBN 0-201-63381-7.
- [9] Kermeta. The kermeta project home page. <http://www.kermeta.org>.
- [10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.
- [11] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003. ISBN 1930110936.
- [12] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN 0-13-247925-7.
- [13] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992. ISSN 0018-9162. doi: <http://dx.doi.org/10.1109/2.161279>.
- [14] N. Moha, V. Mah, O. Barais, and J.-M. Jézéquel. Generic Model Refactorings. In *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, Denver, Colorado, USA, Oct 2009.
- [15] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *MODEL-S/UML*, volume 3713, pages 264–278, Montego Bay, Jamaica, octobre 2005. Springer.
- [16] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc, 1st edition, November 2008. ISBN 0981531601.
- [17] O. M. G. (OMG). Meta Object Facility (MOF) specification. OMG Document ad/97-08-14, septembre 1997.
- [18] O. M. G. OMG. *UML 1.5 Object Constraint Language Specification*, mars 2003. URL <http://www.omg.org/cgi-bin/doc?formal/03-03-13>. Version 1.5.
- [19] J. Steel. *Typage de modèles*. PhD thesis, Université de Rennes 1, avril 2007.
- [20] J. Steel and J.-M. Jézéquel. On model typing. *Journal of Software and Systems Modeling (SoSyM)*, 6(4):401–414, December 2007.
- [21] The OSGi Alliance. OSGi Service Platform Core Specification, Release 4.1, mai 2007. <http://www.osgi.org/Specifications/>.

⁵ www.kermeta.org/mdk/kompose