Kermeta Compiler Documentation

Cyril Faucher
Didier Vojtisek

Abstract

This document aims to provide a development plan of the Kermeta compiler: Kermeta to Java.

Kermeta Compiler Documentation

Cyril Faucher
Didier Vojtisek

Published Build date: 23-January-2008
Published $Date: 2007/12/21 15:42:51 $

Table of Contents

CHAPTER

Preface

This document aims to provide a development plan of the Kermeta compiler: Kermeta to Java. The document is divided in three parts:

- An introduction dedicated to the architecture shows the process of the generation of the Java source code,
- a status board showing the state of the whole of tasks,
- a listing of the issues in compiling, giving the description of the issue and links to the files in the Kermeta CVS. These kmt files are pieces of KMT source code to explain and show the issues in different cases. A piece of a Java source code is as well provided to show the expected Java source code and to finish the generated Ecore file corresponding to the KMT file. All the resources are available at [cvs].

Important

Kermeta is an evolving software and despite that we put a lot of attention to this document, it may contain errors (more likely in the code samples). If you find any error or have some information that improves this document, please send it to us using the bug tracker in the forge: **http://gforge.inria.fr/tracker/?group_id=32** or using the developer mailing list (kermeta-developers@lists.gforge.inria.fr) Last check: v0.99.0

CHAPTER 1

Compilation Process

1.1. Step 1: Kermeta to Ecore

The compilation process is divided in 2 steps: the first one consists in the translation of a Kermeta model into a Ecore model, the second one consists in the generation of the Java source code from the previous Ecore model.

1.2. Step 1: Kermeta to Ecore

Currently this implementation is divided in 2 steps: Km 2 Ecore structure (written in Java), Km 2 Ecore behavior (written in Kermeta and Java).

"Km 2 Ecore structure" implements a visitor desing pattern and consists in the translation of each Kermeta concept to Ecore concept. This step is implemented in "fr.irisa.triskell.kermeta.io" plugin, EcoreExporter.

"Km 2 Ecore behavior" is written in Kermeta. This implementation is Aspect-Oriented, i.e.: we are using the aspect-oriented techniques in order to define how a concept is compiled.

Some examples:

```
aspect class ClassDefinition {
                method compile(context : Integer) : String is do
                        result := ""
                        // Set the traceability
                        self.ecoreProxy := CompileHelper.new.findEClassProxy(self)
                        // Compile the owned elements
                        self.eachOwnedElement{ o | o.compile(context) }
                end
        }
```

```
aspect class ClassDefinition {
        operation eachOwnedElement(func : MultiplicityElement -> Object) : Void is do
        self.ownedAttribute.each{ o |
                        func(o.asType(Property))
                }
                self.ownedOperation.each{ o |
                        func(o.asType(Operation))
                }
        end
}
```

```
aspect class Package {
        reference ecoreProxy : EPackage
        }

        aspect class ClassDefinition {
                reference ecoreProxy : EClass
        }

        aspect class Operation {
                reference ecoreProxy : EOperation
        }
```

This technique has further advantages: the separation of concerns and the reutilisability of several aspects, i.e.: some aspects are grouped by usage like: "containment discovering", "compile" or "traceability".

1.3. Step 2: Ecore to Java

Generation of a genmodel file from the Ecore file. This generation is automatic and the genmodel is modified to customize and fix some parameters for the source code generation. This step is implemented in ?org.kermeta.compiler" plugin.

Generation of the source code from the genmodel file thanks to the EMF generator abilities: Emfmodel interfaces + implementation. This step is implemented in "org.kermeta.compiler" plugin. During this step, a Simk is used too, this model contains informations about the generation of Runners, Launchers and other static methods. A Java source code generation step is made to take in account this model, and also to generate the corresponding Java classes.

CHAPTER 2

Listing of the issues in compiling

2.1. Status board of the issues

Identifier	**Note**	**Status**
Assignment		Impl_Java [in progress] Impl_Kermeta [in progress] Design [OK]
Attribute		Impl_Java [OK] Impl_Kermeta [NOK] Design [NOK]
Body		Impl_Java [OK] Impl_Kermeta [OK] Design [OK]
CallResult		Impl_Java [OK] Impl_Kermeta [OK] Design [OK]
CallSuperOperation		Impl_Java [NOK] Impl_Kermeta [NOK] Design [NOK]
ClassDefinition		Impl_Java [OK]

Identifier	**Note**	**Status**
		Impl_Kermeta [NOK] Design [NOK]
Collection		Impl_Java [in progress] Impl_Kermeta [in progress] Design [in progress]
Conditional		Impl_Java [OK] Impl_Kermeta [NOK] Design [OK]
Containment		Impl_Java [OK] Impl_Kermeta [NOK] Design [NOK]
Exception		Impl_Java [in progress] Impl_Kermeta [NOK] Design [in progress]
FunctionType		Impl_Java [in progress] Impl_Kermeta [NOK] Impl_Simk [OK] Impl_Jet [OK] Design [OK]
Generics		Impl_Java [OK] Impl_Kermeta [NOK] Design [NOK]
Inheritance		Impl_Java [OK] Impl_Kermeta [NOK] Design [NOK]
Initialization		Impl_Java [OK] Impl_Kermeta [NOK]

Identifier	Note	Status
		Design [OK]
InstanceOf		Impl_Java [NOK] Impl_Kermeta [NOK] Design [NOK]
Invariant		Impl_Java [NOK] Impl_Kermeta [NOK] Design [NOK]
JavaStaticCall		Impl_Java [OK] Impl_Kermeta [NOK] Design [OK]
LambdaExpression		Impl_Java [in progress] Impl_Kermeta [NOK] Impl_Simk [OK] Impl_Jet [OK] Design [OK]
Loop		Impl_Java [OK] Impl_Kermeta [NOK] Design [OK]
ModelElementReflexivity		Impl_Java [NOK] Impl_Kermeta [NOK] Design [NOK]
ModelingUnit		Impl_Java [OK] Impl_Kermeta [NOK] Design [NOK]
MultipleInheritance		Impl_Java [NOK] Impl_Kermeta [NOK] Design [NOK]

Identifier	**Note**	**Status**
MultiplePackagesRoot		Impl_Java [NOK] Impl_Kermeta [NOK] Design [NOK]
Multiplicity		Impl_Java [OK] Impl_Kermeta [NOK] Design [OK]
New		Impl_Java [OK] Impl_Kermeta [OK] Design [OK]
OperationStructure		Impl_Java [OK] Impl_Kermeta [NOK] Design [NOK]
Opposite		Impl_Java [OK] Impl_Kermeta [NOK] Design [OK]
Package		Impl_Java [OK] Impl_Kermeta [NOK] Design [NOK]
Postcondition		Impl_Java [NOK] Impl_Kermeta [NOK] Design [in progress]
Precondition		Impl_Java [NOK] Impl_Kermeta [NOK] Design [in progress]
PrimitiveType		Impl_Java [in progress] Impl_Kermeta [NOK]

Identifier	**Note**	**Status**
		Design [in progress]
PropertyBehavior		Impl_Java [NOK] Impl_Kermeta [NOK] Impl_Jet [OK] Design [OK]
PropertyStructure		Impl_Java [OK] Impl_Kermeta [NOK] Design [NOK]
Reference		Impl_Java [OK] Impl_Kermeta [NOK] Design [NOK]
Runner		Impl_Java [OK] Impl_Kermeta [NOK] Impl_Simk [OK] Impl_Jet [OK] Design [OK]
SubPackage		Impl_Java [OK] Impl_Kermeta [NOK] Design [NOK]
Tag		Impl_Java [OK] Impl_Kermeta [NOK] Design [NOK]
Traceability		Impl_Java [NOK] Impl_Kermeta [NOK] Design [in progress]
UriManagement		Impl_Java [NOK] Impl_Kermeta [NOK]

Identifier	**Note**	**Status**
		Design [in progress]

CHAPTER 3

Detailed Description of the Use Cases

3.1. Assignment

Design:

If the left and right expressions have the same type and "attr" is a property from a Class Definition "myCD" and the multiplicity is [0..1] or [1..1]

```
//In kmt:
attr := theValueOfAttr

//In Java:
myCD.setAttr(theValueOfAttr);
```

If the left and right expressions have the same type and "attr" is a property from a Class Definition "myCD" and the multiplicity is [0..*] or [1..*]

```
// In kmt:
attr := theValueOfAttr

// In Java:
myCD.getAttr.add(theValueOfAttr);
```

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

Assignment of a variable has type: String.

test002 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

Assignment of a variable has type: Integer.

test003 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

Assignment of a variable has type of a feature from another ClassDefinition.

test004 [UnitTest]:

Input resources: [kermeta]

Output resources: [eclipse_launcher] [ecore] [java]

Assignment of a feature has type of a feature from another ClassDefinition with further cases: different multiplicities.

Important note: the semantic of the assignment is delegated to EMF, i.e.: the assignment semantic is defined by the Java source code that EMF generate. It would be interesting to add a survey of the behaviour of EMF in order to track semantic evolution from EMF.

3.2. Attribute

3.3. Body

In kmt the body of an operation begins by "do" and is closed by "end".

In Java: "{" to begin and "}" to close an operation body.

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

Design:

Empty body of the operation.

Implementation:

If the body is empty, the generation is not processed.

test002 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

Design:

The body of the operation is not empty

Implementation:

If the body is not empty, the generation is processed and "do" is replaced by "{" and "end" by "}".

3.4. CallResult

In Kermeta syntax the type of the CallResult variable: "result" can be used anywhere, in Java the "return" statement must be used as a root expression in the operation body. Also, we have to add systematically a first declaration of a variable which has the type of the return type with an initialization: String result=null; and as well as the "return" statement at the end of the operation: return result;.

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

The result type of the method is String, then the result variable (having String as type) and the return statement will be added.

test002 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

The result type of the method is VoidType, then the return is not required.

test003 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

The result type of the method is a class A, then the result variable (having A as type) and the return statement will be added.

3.5. CallSuperOperation

3.6. ClassDefinition

3.7. Collection

Design:

Helper:

In order to access to the specific Kermeta?s methods on the Collections, we could use Java static methods. These methods will be generated from the framework in external Java classes, but not integrated into the Ecore model, because the EMF generation does not support static methods.

3.8. Conditional

Design:

```
// In kmt:
if(...) then
...
else
...
end

//In Java:
if(...) {
} else {
}
```

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

An example of a Conditional with a simple "if".

test002 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

An example of a Conditional with an "if" with an "else".

3.9. Containment

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

3.10. Exception

3.11. FunctionType

Design:

A Function Type is called inside another method.

The function type is prettyprinted in a specific static method and called by the method that uses it.

In the context of a lambda expression:

The two methods are generated in a same new Java Class in a "helper" package.

```
/** Kermeta source code **/
a.ref.each{ r |
        stdio.writeln(r.toString)
}

/** Java source code **/
// Method that uses the function type
  public static void each(EList String list) {
    for(String it : list) {
      func(it);
    }
  }

// The FunctionType prettyprinted in the same Java class
  private static void func(String r) {
    System.out.println(r.toString());
  }
```

The call of the method from the original method:

```
Main_3_Helper.each(a.getRef());
```

3.12. Generics

3.13. Inheritance

Simple inheritance support.

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

Simple inheritance sample between 2 Kermeta ClassDefinition

test002 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

Simple inheritance sample between a new ClassDefinition and an EClass from the Ecore metamodel

test003 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

Simple inheritance sample between a new ClassDefinition and an EClass from the UML metamodel

3.14. Initialization

Design:

- Initialization of a variable:

```
// In kmt:
var aPrinter : Printer init Printer.new

// In Java:
comptest2.Printer aPrinter = comptest2.Comptest2Factory.eINSTANCE.createPrinter();
```

- Initialization of a String:

```
// In kmt:
var text1 : String init "Hello"

// In Java:
String text1 = "Hello ";
```

```
// In kmt:
var text2 : String
text2 := "World"

// In Java:
String text2 = null;
text2 = "World";
```

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

Direct initialization of a variable (String).

test002 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

Indirect initialization of a variable (String).

test003 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

Direct initialization of a variable (Integer).

test004 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

Indirect initialization of a variable (Integer).

test005 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

Direct initialization of a feature (A).

test006 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

Indirect initialization of a feature (A).

3.15. InstanceOf

TODO

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

TODO

3.16. Invariant

3.17. JavaStaticCall

Design:

```
//In kmt:
extern fr::irisa::triskell::kermeta::runtime::io::SimpleFileIO.writeTextFile(filename, text)

//In Java:
fr.irisa.triskell.kermeta.runtime.io.SimpleFileIO.writeTextFile(filename, text);
```

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

An example of a JavaStaticCall.

3.18. LambdaExpression

A Function Type is called inside another method.

The function type is prettyprinted in a specific static method and called by the method that uses it.

In the context of a lambda expression:

The two methods are generated in a same new Java Class in a "helper" package.

```
/** Kermeta source code **/
a.ref.each{ r |
  stdio.writeln(r.toString)
}

/** Java source code **/
// Method that uses the function type
public static void each(EList String list) {
  for(String it : list) {
    func(it);
  }
}

// The FunctionType prettyprinted in the same Java class
private static void func(String r) {
  System.out.println(r.toString());
}
```

```
// The call of the method from the original method:
Main_3_Helper.each(a.getRef());
```

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [eclipse_launcher] [ecore] [java]

3.19. Loop

Design:

```
// In kmt:
var j : Integer init 5
from var i : Integer init 0
until i == self
loop
        i := i + 1
end

// In Java:
int j = 5;
int i = 0;
while( i != j ) {
        i = i + 1;
}
```

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

An example of a loop.

3.20. ModelElementReflexivity

The ModelElement reflexivity is provided by EMF on models and metamodels. When a km model is translated into in XMI, all the full reflexivity is able. The limit of the reflexivity is given by the input metamodel.

3.21. ModelingUnit

3.22. MultipleInheritance

3.23. MultiplePackagesRoot

3.24. Multiplicity

Design:

Implemented by the structural part (?model? plugin) of the EMF generated source code.

Some issues for multiplicity have taken into account during the assignment translation.

3.25. New

Design:

The creation of Object is performed by the new method, in EMF-Java: a factory is used and it can be called like:

For MyClassDefinition owns by the package: comp_new_test001

```
// In Kermeta:
A.new

// In EMF-Java:
Comp_new_test001Factory.eINSTANCE.createA();
```

test001 [UnitTest]:

Input resources: [kermeta]

Output resources:

TODO

3.26. OperationStructure

3.27. Opposite

Design:

Implemented by the structural part (?model? plugin) of the EMF generated source code.

3.28. Package

Design:

The granularity of the EMF regeneration is the EPackage. If we want to generate incrementally the Java source code, also we regenerate the whole of the EPackage.

3.29. Postcondition

3.30. Precondition

3.31. PrimitiveType

Design:

Helper:

In order to access to the specific Kermeta?s methods on the Primitive Types, we could use Java static methods. These methods will be generated from the framework in external Java classes, but not integrated into the Ecore model, because the EMF generation does not support static methods.

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

TODO

3.32. PropertyBehavior

Design:

Behavior of a derived property: getter/ setter.

This ability is not supported by the EMF mechanism based on the use of EAnnotations containing the Java source code corresponding to the expected behavior.

Thus, we are using the "overloadable" property of the JET templates. In a JET template, that is able to define some pieces of source code that will be overloadable. The expected source code is also added via a file with a specific name and location (define in the JET template)

During the compilation, we are using the JET templates from EMF (plugin: org.eclipse.emf.codegen.ecore). Also, in your case, the derived property, we have written new implementations for the getter and setter. This new implementation uses a EAnnotation containing the Java source code implementing the getter and setter. Finally, the mechanism to get and put the code is very close to this one used by EMF to compile the EOperation behavior

The specific source added are located in the plugin: org.kermeta.compiler.generator.emftemplates

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

This sample shows an example of the compilation a derived property with a getter and a setter.

3.33. PropertyStructure

Design:

Derived property signature.

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [ecore] [java]

This sample shows the structural compilation of derived property in different cases: property typed by a PrimitiveType or by a ClassDefinition. The second variation is the "readonly" property.

3.34. Reference

3.35. Runner

Design:

In order to facilitate the accessibility for the users of the generated source code, some runners are generated. A Runner is a Java Class containing a main method which is able to launch the execution of a method. Not all the methods should infer a runner, only the methods which have none parameter or some parameters having String as type. The parameterized Method and method owning by a parameterized Class should not infer a Runner too. The runners are generated in a specific Java Package for each source code package. Another file is generated that is a ?Eclipse Java Application Launch? file: *.launch (generated in the root of the project). This one is able to configure automatically the "Run ?" assist of Eclipse to launch quickly an execution. The parameters of this file are provided by the tags: mainClass and mainOperation given by the ModelingUnit.

Warning

The simk model should be populated by the pass written in Kermeta and not in Java.

test001 [UnitTest]:

Input resources: [kermeta]

Output resources: [eclipse_launcher] [ecore] [java]

This sample shows the different cases for the generation of Runners: none parameters, one parameter having String as type, one parameter with different type, multiple parameters having String as type, multiple parameters with different types.

3.36. SubPackage

3.37. Tag

3.38. Traceability

A traceability mechanism could be added between the merged Kermeta model file (*.km) and the generated Java.

3.39. UriManagement

Problem:

To load the user model (conforms to a metamodel in ecore), we are using the uri-registry mechanism from EMF.

In the context where an user has its own generated source code, then the both uri: this one generated by the user and this one generated by the compiler are the same. If the two plugins are deployed then some conflicts will appear.

A primary approach is to ignore if the user has generated its own source code. Also we consider the uri generated by the compiler as the reference.

Several solutions could be considered:

using the plugin extension: factory_override

From Eclipse: (http://dev.eclipse.org/newslists/news.eclipse.tools.emf/msg22892.html)

This extension point is used to register an overriding Ecore factory implementation against a namespace URI (Uniform Resource Identifier) in EMF's global package registry, EPackage.Registry.INSTANCE. When the corresponding Ecore package is initialized, the package registry is consulted for the registered factory implementation to override the default that would otherwise be used.

uri - A URI that uniquely identifies an Ecore package for which this factory in an override.

class - A fully qualified Java classname of an overriding factory implementation.