Kermeta language

Reference manual

Zoé Drey
Cyril Faucher
Franck Fleurey
Vincent Mahé
Didier Vojtisek

Abstract

This manual presents the various aspects of the Kermeta language. This is the reference manual for anybody who want to use Kermeta.

Published Build date: 3-November-2010
$LastChangedDate:: 2010-10-25 10:46:57#$

Preface. viii

Chapter 1. Introduction to the language 1

1.1. Presentation 1

1.2. An imperative syntax 2

1.2.1. First Program 2

1.2.2. Classic features 2

1.2.3. Corresponding sections in the Reference chapter 3

1.3. An object-oriented language 3

1.3.1. Corresponding sections in the Reference chapter 4

1.4. A model-oriented language 4

1.4.1. Associations : toward a first concrete example of a Kermeta model 5

1.4.2. Loading an existing model 6

1.4.3. Navigation in a model 6

1.4.4. Model type 7

1.4.5. Kermeta model reflexively available 7

1.4.6. Corresponding sections in the Reference chapter 7

1.5. Aspect-oriented language 7

1.5.1. Corresponding section in the Reference chapter 8

1.6. Some "advanced" features 8

1.6.1. Functions in Kermeta 8

1.6.2. Other advanced features 9

1.6.2.1. High level modeling 9

1.6.2.2. Other technical features 9

Chapter 2. Reference 10

2.1. Comments 10

2.1.1. Simple and multi-line text comments 10

2.1.2. Linked comments 10

2.2. Escaping reserved keywords 11

2.3. Operators 11

2.4. Statements : block, condition, loop 13

2.4.1. Block statement 13

2.4.2. Conditional statement 13

2.4.3. Loop Statement 14

2.5. File dependency : structuring code 14

2.5.1. Require 14

2.5.2. Accepted require content 15
2.5.3. Supported protocols 15
2.5.4. Organizing code 16
2.6. Using Variables 16
2.7. Basic types 17
2.7.1. Primitive types 17
2.7.2. Enumeration types 18
2.7.3. Local datatype using "alias" 18
2.8. Classes and methods 19
2.8.1. Classes 19
2.8.2. Defining operations 19
2.8.2.1. Result 20
2.8.2.2. Operations as main entry point. 21
2.8.3. Initializing classes 21
2.8.4. Rationale 21
2.8.4.1. No constructor 21
2.8.4.2. No operation overloading 22
2.9. Inheritance 22
2.9.1. Using simple inheritance 22
2.9.2. Cast 23
2.9.2.1. Casting using conditional assignment 23
2.9.2.2. Casting using asType 24
2.9.2.3. Useful functions when casting 24
2.9.3. Using multiple inheritance 24
2.9.4. Overriding behavior with methods 25
2.9.5. Some limitations 26
2.9.5.1. Operation redefinition 26
2.9.5.2. Operation specialization 27
2.9.5.3. Operation overloading 27
2.9.5.4. Conflicts related to multiple inheritance 28
2.10. Genericity 29
2.10.1. Generic classes 29
2.10.2. Generic operations 29
2.10.3. Type usable with genericity 30
2.11. Exception handling 30
2.12. Loading and saving models 31
2.12.1. Prepare a model 32
2.12.2. Load a model from an EMF Resource 32
2.12.3. Save a model into an EMF Resource 33
2.13. Packages 33

2.13.1. Definition 34
2.13.2. Use of packages 34
2.13.3. Syntaxic sugars 35
2.14. Collections 35
2.14.1. Definition and initialization 35
2.14.2. Some existing useful functions 37
2.14.2.1. Function each 37
2.14.2.2. Function forAll 38
2.14.2.3. Function select 38
2.14.2.4. Function reject 38
2.14.2.5. Function collect 38
2.14.2.6. Function detect 38
2.14.2.7. Function exists 39
2.15. Class properties 39
2.15.1. Attributes (attribute), references (reference) 39
2.15.2. properties modifiers 40
2.15.3. Association using opposite properties 40
2.15.4. Derived properties (property) 42
2.15.5. How to access and control the properties in Kermeta 43
2.15.6. Assignment behavior for attribute (and reference) 45
2.16. Objects comparison 46
2.16.1. equals method 46
2.16.2. Comparison for Primitive Types 47
2.16.3. Collection comparison 48
2.17. Lambda Expressions and functions 48
2.17.1. Syntax 49
2.17.2. Some existing useful functions 49
2.17.3. Defining new functions in a class 50
2.17.4. Defining lambda expression variables 51
2.18. Dynamic evaluation of Kermeta expressions 52
2.19. Design by contract (pre, post, inv contraints) 53
2.19.1. Writing a contract 53
2.19.1.1. pre – post conditions syntax 53
2.19.1.2. Invariant constraint syntax 53
2.19.2. Checking your constraints 54
2.19.2.1. Checking pre – post condition 54
2.19.2.2. Checking invariant 54
2.20. Weaving Kermeta code 55
2.20.1. Textual syntax for merging 55
2.20.2. Example 1: Simple Class merge 56
2.20.3. Example 2: merge with feature redefinition 57

2.20.4. Example 3: merge with operation overload 58
2.20.5. Aspect and inheritance 58
2.20.6. Aspect and abstract 58
2.20.7. Aspect without common base 59

2.21. Model type 59

2.21.1. Definition of a model type 59
2.21.2. Using Model types variables 60
2.21.3. Model type serialisation 60
2.21.4. Model type conformance 60
2.21.5. Using Model types in generic classes/operations 61
2.21.6. A more complex example of conformance : FSM variants 61

2.22. Using existing java code in Kermeta 61

2.22.1. Using extern to call java code 62
2.22.2. Requiring jar file to call java code 63

2.23. Cloning objects 64

Chapter 3. Kermeta Metamodel 66

3.1. Architecture 66

3.2. Structure package 67

3.2.1. NamedElement view 68
3.2.2. Type system view 68

3.3. Behavior package 69

3.3.1. Control Structures 70
3.3.2. Variables 70
3.3.3. Call Expressions 71
3.3.3.1. CallSuperOperation 71
3.3.3.2. CallVariable 72
3.3.3.3. CallResult 72
3.3.3.4. CallFeature and SelfExpression 72
3.3.4. Assignment 73
3.3.5. Literals 73
3.3.6. Lambda Expression 74

3.4. Viewing Kermeta metamodel 75

Chapter 4. Kermeta framework 76

A. Language keywords. 78
B. Known Kermeta tags. 81
C. Kermeta / Ecore mapping. 83

List of Figures

1.1. Kermeta positioning ... 1

1.2. A concrete example : a library ... 5
2.1. A simple family tree model ... 23
2.2. : multiple inheritance ... 25

2.3. cs.ecore sample metamodel ...31

2.4. : The Kermeta collections ... 36
2.5. Attributes and references ... 41

3.1. EMOF extension and Kermeta promotion ...67

3.2. Structure package ...67

3.3. NamedElement class diagram ... 68
3.4. Kermeta type system class diagram (the big picture) ... 69

3.5. Behavior package ...70

3.6. Control structure ... 70
3.7. Use of variables ... 71
3.8. use of exceptions ... 71
3.9. Kermeta assignment expression ... 73
3.10. Kermeta Literal Expression ... 74
3.11. Kermeta lambda expressions ... 75

CHAPTER
Preface

Kermeta is a Domain Specific Language dedicated to metamodel engineering. It fills the gap let by MOF which defines only the structure of meta-models, by adding a way to specify static semantic (similar to OCL) and dynamic semantic (using operational semantic in the operation of the metamodel). Kermeta uses the object-oriented paradigm like Java or Eiffel.

This document gives beginners an introduction to the Kermeta language, then it offers a reference of all the aspects of the language, including the textual syntax, the metamodeling features and some more advanced features. Two other chapters present the Kermeta Metamodel and the Kermeta framework.

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: v1.4.1

Tip

The most update version of this document is available on line from http://www.kermeta.org .

CHAPTER 1

Introduction to the language

This chapter aims to help you to have a quick overview of most of the features of Kermeta. Then, it gives the pointers to the detailed sections in the reference chapter. It also gives some small examples that should help you to understand the basis of the concepts without having to jump to the corresponding detailed section.

1.1. Presentation

Kermeta is a metamodeling language which allows describing both the structure and the behavior of models. It has been designed to be compliant with the OMG metamodeling language EMOF (part of the MOF 2.0 specification) and Ecore (from Eclipse). It provides an action language for specifying the behavior of models.

Kermeta is intended to be used as the core language of a model oriented platform. It has been designed to be a common basis to implement Metadata languages, action languages, constraint languages or transformation language.

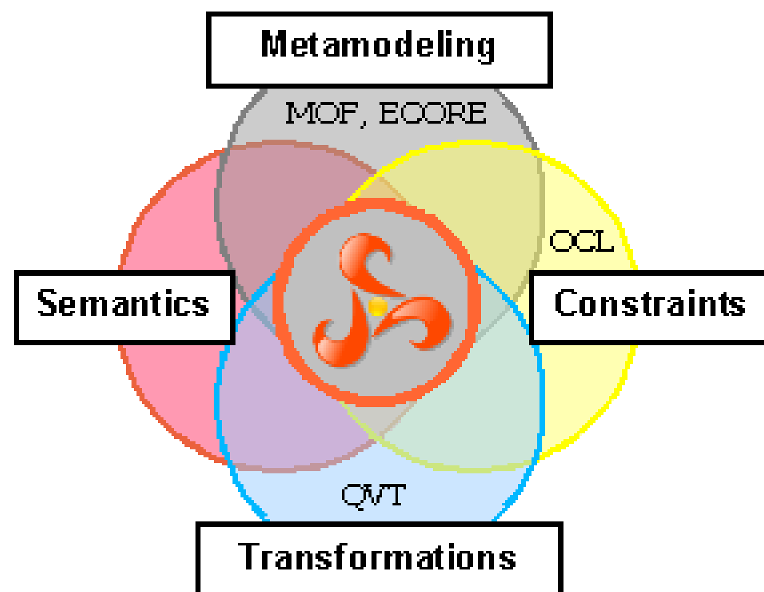


Figure 1.1. Kermeta positioning

In a nutshell, Kermeta is :

- MOF compliant (EMOF compliant to be precise)
- Model oriented
- Imperative
- Object-Oriented
- Aspect-Oriented
- Statically Typed (100% typesafe)

In addition to these characteristics, it includes some typically model-oriented concepts like associations, multiplicities or object containment management.

This chapter presents the main features of the Kermeta language. Section 2 presents the general syntax of the language, sections 3 & 4 give details about the object-oriented and model-oriented features of the language and finally section 4 provides information about some extra concepts in Kermeta (including aspect orientation).

With its workbench, it's goal is to provide a support for all Language Driven Engineering activities. It will be typically used to build tools useful to build software. This includes (but is not restricted to): model checkers, simulators, model transformations (any kind of transformations including model weavers or compilers).

1.2. An imperative syntax

Kermeta is an imperative language for modeling, with a basic syntax inspired from Eiffel. Code is statically type checked, and execution is made by an interpreter (a compiler is on the way, for exhausted performances).

1.2.1. First Program

Even if it is not very useful in our context, since it doesn't show the really interresting structures of the language, here is the traditional " Hello world " example you can find in every programming book.

```
@mainClass "helloworld::HelloworldExample"
@mainOperation "sayHello"
package helloworld;

require kermeta
using kermeta::standard

class HelloworldExample
{
   operation sayHello() is
   do
      stdio.writeln("Hello world, ...")
   end
}
```

1.2.2. Classic features

Kermeta language includes usual statements like blocks and loops, comments, etc

```
do
  // a loop for getting a text from an user
  var s : kermeta::standard::String
  from var found : kermeta::standard::Boolean init false
  until found
  loop
    s := stdio.read("Enter a text:\n --> ")
    if s.size > 0 then
      found := true
    else
      stdio.writeln("ERROR - Empty text!")
    end
  end
  stdio.writeln("\n You entered: " + s)
end
```

1.2.3. Corresponding sections in the Reference chapter

All these "classic" imperative features and their syntaxes are described in Chapter 2, *Reference*. More precisely in

- Section 2.1, “Comments”,
- Section 2.4, “Statements : block, condition, loop”,
- Section 2.6, “Using Variables”,
- Section 2.3, “Operators”,
- Section 2.2, “Escaping reserved keywords”

1.3. An object-oriented language

Users of modern programming languages, like Java, would feel easy with object-oriented features in Kermeta: classes, inheritance, exceptions, and even genericity.

```
require "truc"
// persons who write documents
class Writer {
  attribute name : kermeta::standard::String
}

// generic concept for every document
abstract class Document {
  reference author : Writer
  attribute text : kermeta::standard::String
}

// a "Document" from the real world
class Book inherits Document {}

// a specialized "Book"
class ChildBook inherits Book {
  attribute minimalAge : kermeta::standard::Integer
```

```
}
```

Such classes can be used for verifications:

```
// a specialized Exception
class AgeException inherits kermeta::exceptions::Exception {}

abstract class Reader {
   operation read(book : ChildBook) : Void is abstract
}

class Child inherits Reader {
   attribute age :  kermeta::standard::Integer
   operation initialize(age :  kermeta::standard::Integer) : Child is
   do
      self.age := age
      result := self // return self so we can chain this call directly after a new
   end
   // an action which triggers an Exception
   operation read(book : ChildBook) : Void is
   do
      if age < book.minimalAge then
         raise AgeException.new
      end
   end
}
```

1.3.1. Corresponding sections in the Reference chapter

You can get more informations about Kermeta object-oriented features in Chapter 2, *Reference*. More precisely in

- Section 2.8, “Classes and methods”
- Section 2.9, “Inheritance”
- Section 2.10, “Genericity”
- Section 2.11, “Exception handling”
- Section 2.13, “Packages”
- Section 2.15, “Class properties”
- Section 2.16, “Objects comparison”

1.4. A model-oriented language

As explained in the Preface and in Architecture, Kermeta extends the MOF. It provides useful means to manipulate models. The support of model introduces the main difference with "more" traditional programming languages.

1.4.1. Associations : toward a first concrete example of a Kermeta model

Association is one of the key concepts when using and defining models. It is obviously part of Kermeta.

MOF defines the concept of "Property" which generalizes the notions of attributes, and associations (composite or not) that you can find in UML. Kermeta syntax also distinguishes these two notions as introduced in Section 2.15, “Class properties”.

As a reminder, the `attribute` keyword defines a link with containment (a composite association) whereas the `reference` keyword just defines an association. As you can see, property declarations are very close to variable declarations introduced in Section 2.6, “Using Variables”). Each reference may be explicitly linked to another reference (it is the `opposite` concept in MOF terminology – see also section Section 2.15.1, “Attributes (attribute), references (reference) ”).

```
class Library
{
   attribute books : set Book[0..*]
}

class Book
{
   attribute title : String
   attribute subtitle : String

   reference authors : oset Author[1..*]#works
}


class Author
{
   attribute name : String
   attribute lastName : String

   reference works : set Book[0..*]#authors
}
```

If we represent our Kermeta model in a graphical syntax we obtain the following class diagram (Figure 1.2, “A concrete example : a library”).

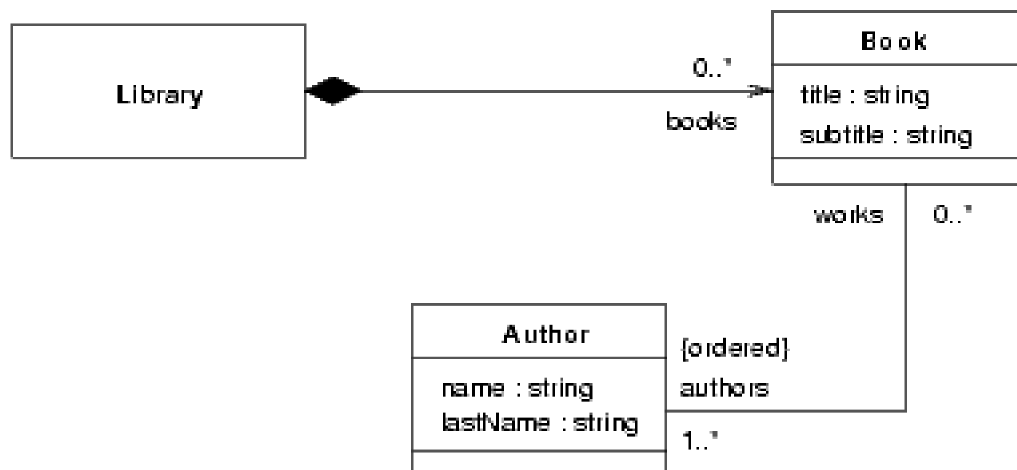


Figure 1.2. A concrete example : a library

1.4.2. Loading an existing model

Using Eclipse Modeling Framework (EMF), Kermeta can load and save models done with other EMF tools.

```
/* Initialize the EMF repository */
var repository : EMFRepository init EMFRepository.new

/* Create an EMF Resource, given model and metamodel URIs as String */
var resource : Resource init repository.createResource(myModelURI, itsMetamodelURI)

/* Load the resource */
resource.load

// get elements from the resource
// in this sample, you know that your root element is always a Library,
// so you can directly get the first one
var aLibrary : Libray
aLibrary ?= resource.one // note the conditional assignment using the ?=, if not a Library you'll get Void
```

In the same way, you can serialize a model, or load, change and save an existing model.

> **Caution**
>
> Your model URI MUST be of the form "platform:/resource/myProject/myModel" or "platform:/plugin/myProject/myModel".
>
> Your metamodel URI MUST be of the form "platform:/resource/myProject/myModel" or "platform:/plugin/myProject/myModel" or an URI registered in the EMF registry.

> **Caution**
>
> Be aware that you CANNOT load kermeta text files (*.kmt). Only xmi files are allowed to be loaded. Parsing and obtaining a model from a textual syntax is not part of Kermeta. This is the role of other tools (like sintaks). Technically, it is possible to create some Kermeta operation that will hide this step, however, this is not the goal of this manual to explain this procedure.

1.4.3. Navigation in a model

Actually, navigating in a model is as simple as using objects in an object-oriented program. However, several features have been added in order to ease this activity.

For example, thanks to the lambda expressions, the collections of the language are easily manipulated using lexical closure (select, collect, each, etc). This applies to all the collections of the language, the one you may define directly but also the one which are used when an Attribute or Reference has a multiplicity greater than 1.

Example (based on the library sample of Section 1.4.1, “Associations : toward a first concrete example of a Kermeta model”):

```
var smithBooks : Set<Book> init Set<Book>.new
smithBooks.addAll(
    lib.books.select{aLibraryBook |
        aLibraryBook.authors.exists{aBookAuthor | aBookAuthor.lastName == "Smith"}})
```

In the example above, lib is an instance of Library. It searchs in the books, select the books where the author last name is "Smith".

1.4.4. Model type

In order to improve reuse of existing code between metamodel variants, the language introduces the notion of ModelType. It is based on the notion of conformance between two metamodels. This allows to write behavior that is valid for a given metamodel and that will also work for any conformant metamodel.

TODO write a small illustrative example of a simple printer based on a ModelType : a subset of class diagram of UML

1.4.5. Kermeta model reflexively available

Kermeta has been developed, using MDE principles so it also provides its own metamodel (reflectiveley available). Details of Kermeta metamodel is available in Chapter 3, *Kermeta Metamodel*

1.4.6. Corresponding sections in the Reference chapter

You can get more informations about all Kermeta model-oriented features in the Chapter 2, *Reference*. More precisely in

- Section 2.12, “Loading and saving models”
- Chapter 3, *Kermeta Metamodel*
- Section 2.21, “Model type”

1.5. Aspect-oriented language

Since Kermeta is an extension of MOF, a MOF meta-model can conversely be seen as a valid Kermeta program that just declares packages, classes and so on but does nothing. Kermeta can then be used to breath life into this meta-model by incrementally introducing aspects for handling concerns of static semantics, dynamic semantics, or model transformations.

One of the key features of Kermeta is the static composition operator `require` allows extending an existing meta-model with new elements such as properties, operations, constraints or classes. This operator allows defining these various aspects in separate units and integrating them automatically into the meta-model. 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 meta-models or to split meta-models into reusable pieces. It can be compared to the open class paradigm. Consequently a meta-class that identifies a domain concept can be extended without editing the meta-model directly. Open classes in Kermeta are used to organize `cross-cutting` concerns separately from the meta-model to which they belong, a key feature of aspect-oriented programming. With this mechanism, Kermeta can support the addition of new meta-class, new subclasses, new methods, new properties, new contracts to existing meta-model. The require mechanism also provides flexibility. For example, several operational semantics could be defined in separate units for a single meta-model and then alternatively composed depending on particular needs. This is the case for instance in the UML meta-model when several semantics variation points are defined.

Thank to this composition operator, Kermeta can remain a kernel platform to safely integrate all the concerns around a meta-model. This feature has proved to be extreemly useful in the context of metamodel engineering.

1.5.1. Corresponding section in the Reference chapter

You'll find more details in the Reference chapter, more precisely in: Section 2.20, “Weaving Kermeta code”

1.6. Some "advanced" features

Kermeta implements several "less common" or advanced features that helps in many situations.

Typically, lambda expressions/functions is a well known concept, very useful in almost all kermeta code. Even, if you will probably not write your own function, you'll certainly use the existing one defined on collections.

1.6.1. Functions in Kermeta

In order to implement and statically type check OCL-like iterators, Kermeta includes some limited functional features by implementing lambda expressions.

This is typically used on `Collection` which provides functions like : `each`, `select`, `forAll`, `detect`, ...

Example 1: the following code will build a collection of names of the operations that start with "test".

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

Example 2: Providing a time function on Integer

```
operation times(body : <Integer->Object>) : Void is do
  from var i : Integer init 0
  until i == self
  loop
    body(i)
    i := i + 1
  end
end
```

this allows to write code like :

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

See sections "Lambda Expressions and functions" and Lambda Expression for detailed informations.

1.6.2. Other advanced features

1.6.2.1. High level modeling

Kermeta most recent versions embed advanced concepts like Lambda Expressions and functions, Dynamic evaluation of Kermeta expressions, Design by contract (pre, post, inv contraints) or Weaving Kermeta code.

1.6.2.2. Other technical features

As Kermeta language is implemented upon Eclipse and Java, you can call Java code inside Kermeta code.(see Section 2.22, “Using existing java code in Kermeta”)

There is some special behavior regarding object comparison or cloning.

CHAPTER 2

Reference

The reference sections are ordered in four parts: the imperative syntax (1-6), the object-oriented (7-10) and the model-oriented (11-15) features, the advanced concepts (16-21).

2.1. Comments

> **Warning**
>
> Comments are a little bit particular in the Kermeta language because they are not all ignored by the interpreter. There are two kinds of comments: the first (syntax : // foo , /* bar * **/) is only associated to the text editor view in which the user edits his Kermeta file, and the second one(syntax : /** foo */ **or** @tag_name "tag value") is translated into a MOF tag linked to structures described in the Kermeta model. The first one, is a text decoration, the second one is part of the Kermeta model.

2.1.1. Simple and multi-line text comments

Like in many languages, Kermeta provides two ways to add comments in your models:

- simple line comments, i.e text line beginning with //

```
// This is a single line comment
```

- multi-line comments, i.e text between /* */. Be careful with this notation : the opening syntax must not have juxtaposed stars ("/**"), otherwise it will be considered as a linked comment (see Section 2.1.2, “Linked comments”), i.e a comment that is part of the Kermeta program as a model.

```
/* This a multi line comment
all these lines are comments
this line too. */
```

2.1.2. Linked comments

Kermeta provides a way to define named and unnamed annotations, that have to be defined just above any model element among `Package`, `ClassDefinition`, `Property`, `Operation`. Such annotations correspond to MOF tags, and are linked to the elements which immediately follows.

To define a named annotation, you have to use a special symbol "@", whereas an anonymous annotation has to be written between /** and */

Example 1: you can define an annotation to describe the role of a property

```
@usage "count the number of ..."
reference myCounter : Integer
```

Example 2: you can document your classes and operation using /** ... */

```
/** * This is a documentation tag for the class "myClass" */
class MyClass {
   /** This is a documentation tag for myOperation */
   operation myOperation() is do
      // Unlinked comment
   end
   @desc "This is a named annotation for thisOperation"
   operation thisOperation() is do
      /* This is an unlinked comment */
   end
}
```

2.2. Escaping reserved keywords

Kermeta textual syntax uses several keywords like `class`, `attribute`, `reference`, `result`, etc. (Please see Appendix A, *Language keywords* at the end of this document to get the complete list of Kermeta keywords.)

This doesn't mean you cannot use these words for your own model. Moreover, this is only a textual syntax limitation.

So Kermeta syntax allows you to use the word you want, you simply have to prefix it with a tilda ~.

This example is valid even if we use "class" and "attribute" which are keywords in the language:

```
class ~class {
   attribute ~attribute : kermeta::standard::String
   attribute ~class : kermeta::standard::String
}
```

2.3. Operators

Priority	Operator	Operand types	Semantic
1	+	Numeric	Add two numeric values
		String	Concatenate two strings

Priority	Operator	Operand types	Semantic
1	-	Numeric	Subtract two numerous values
2	*	Numeric	Multiply two numeric values
2	/	Numeric	Divide the first operand by the second

Notice that most of these arithmetic operators are only defined for numeric primitive types like Integer. Except the + operator which is a concatenation operator for String, they are not applicable on String, Boolean nor collections

Priority	Operators	Operand Types	Semantics
3	==	All	True, if op1 value's is the same that op2 value's
3	!=	All	True if op1 value's is different of op2 value's
3	<	Numeric	True if op1value's is strictly smaller than op2 value's
3	<=	Numeric	True if op1 value's is smaller or equals than op2 value's
3	>	Numeric	True if op1 value's is strictly greater than op2 value's
3	>=	Numeric	True if op1 value's is greater or equals than op2 value's

Priority	Operators	Operands Types	Semantics
4	and	Boolean	True if op1 and op2 are evaluated to true
4	or	Boolean	True if one of the operators is evaluated to true
4	not	Boolean	True if op is false.

Note

The `==` and `!=` can be applied to any Object, in this case, the comparison is based on the `equals` operation defined for the object and thus can be redefined to compare only the values of the objects. To be sure to compare the identity of objects even if they redefine the operation `equals`, you need to compare their `oid` (ie. their identifiers).

See Section 2.16, “Objects comparison” for more details about object comparison.

2.4. Statements : block, condition, loop

2.4.1. Block statement

Kermeta provides a block notion to manage scope of variable. Instruction of the block have to be inserted between `do` and `end` keywords. Theses two keywords may be omitted for the conditional and for the loop structures.

A variable could only be accessed in the block where it was defined and in its sub blocks:

```
do
   var v1 : Integer init 3
   var v2 : Integer init 2
   do
      var v3 : Integer
      v3 := v1 + v2

      var v2 : Integer // error : v2 is already declared in the upper block
   end
   var v4 : Integer init v3 // error : v3 is unknown here
end
```

2.4.2. Conditional statement

Kermeta's conditional statement is composed of at least two elements : a boolean expression and a block that is executed if the boolean is evaluated to `true`. You can add a third element, with the `else` keyword, that is executed if the boolean expression is evaluated to `false`.

Example 1: if..then..else block

```
var v1 : Integer init 2
var v2 : String init "blah"

if v1 > 5 then v1 := v1-5

if v1 == 2 then
   v2 := v1
   v1 := v2 + v1
else
   v1 := 0
end
```

The `if` statement is an expression (see Chapter 3, *Kermeta Metamodel*). As any expression in Kermeta, it can return a value. The return type of the `if` statement must be a super type of the values "returned" by both `then` and `else` blocks (otherwise the type checker will send an error). The values considered as the result of the evaluation (the "returned" values) of the `if` statement are the last evaluated statement inside `then` or `else` block

Example 2: conditional is an expression

```
var s : String
   s := if false then "a" else "b" end
```

Example 3: a more complex conditional

```
var x : String
   var y : Integer init 5
   x := if y < 3 then
       stdio.writeln("hello")
       "a"
       else
       "b"
       "c"
       end // The String "c" will be the value of x
```

2.4.3. Loop Statement

Here is a sample of a typical loop in Kermeta.

```
var v1 : Integer init 3
var v2 : Integer init 6

from var i : Integer init 0
until i == 10
loop
   i := i + 1
end
```

> **Note**
>
> Unlike Java, there is no exit, break or continue function in Kermeta.

See Section 2.14, “Collections” for functions offering iterator-like scanning.

2.5. File dependency : structuring code

Kermeta code can be quite large, and invloves many classes in different metamodels. The main mechanism used to organise your code is the `require` statement.

2.5.1. Require

When you need to refer explicitly another entity defined in another file, you have to use the `require` primitive. It allows loading definitions of an other Kermeta file when file is processed. In the following example, we define a class C which inherits from the A class previously defined.

```
// file : MyPackage-part1.kmt
package subPackage1;

class A
{
  // ...
}
```

```
// file : MyPackage-part2.kmt
```

```
package MyPackage;
require "MyPackage-part1.kmt"

class C inherits subPackage1::A
{
    // ...
}
```

> **Note**
>
> In most case, the order of the declaration in not important. The only exception is a very rare situation related to aspects (see Section 2.20, “Weaving Kermeta code”).

> **Note**
>
> You can create cyclic dependencies of files, the environment will deal with that.

2.5.2. Accepted require content

You can `require` different kind of entity

Obvioulsy, you can `require` Kermeta textual syntax, ie. ***.kmt files**

You can `require` Kermeta models, ie. ***.km files**.

You can `require` Ecore models, ie. ***.ecore files**. These models can then be used as if they were written in Kermeta. In order to add behavior to the classes defined in .ecore files you may : use the weaving to dynamically weave behavior to the ecore, or roundtrip using the button action in the workbench (ecore->kmt->ecore) to statically add the behavior (as EAnnotations) into the ecore file.

> **Warning**
>
> A special attention must be put when requiring resources. It is not allowed to get several versions of the same class that comes from several required files. Kermeta cannot know which version it must use, so you'll get an error.
>
> A typical error, is to require both an ecore file and the registered version of it. They represent two distinct resources in memory.
>
> A more clever error, is when the two versions are hidden inside an erroneous ecore file which uses several ways to access some metaclass and actually is not consistent. (ex: you find references to both http://www.eclipse.org/emf/2002/Ecore and platform:/plugin/org.eclipse.emf.ecore/model/Ecore.ecore in the ecore file or one of the dependent files)

In order to use the seamless java import (still prototype in v0.4.2), you can require a **jar file**. It will automatically convert the java into a Kermeta representation in order to be able to call java code as if it was written in Kermeta. see Section 2.22, “Using existing java code in Kermeta” for more details.

2.5.3. Supported protocols

Kermeta reuse some of the protocols provided by Eclipse.

Relative path. You can use a path relative to the current file using the normal syntax (./ , ../, etc)

Eclipse workspace relative path. In Eclipse, you can use a path relative to the user's workspace. In that case the path looks like `platform:/resource/`*`your_project_name`*`/`*`path_to_the_file`*

Eclipse plugin relative path. In Eclipse, you can use a path that search in the installed plugins. In that case the path looks like `platform:/plugin/`*`plugin_name`*`/`*`path_to_the_file`*

Eclipse workspace or plugin relative path. When deploying Kermeta code, it is sometime interesting to search in both the user's workspace and in the plugins. In that case the path looks like `platform:/lookup/`*`plugin_or_project_name`*`/`*`path_to_the_file`*

> **Note**
>
> This platform:/lookup/ is available in Kermeta only, and only in the require statement.
>
> Be careful, using the current version (Kermeta 1.3.1) since a deployed plugin using this protocol may be modified by the user's workspace content!

Registered Ecore. A variant of requiring an ecore file is to require a **registered EPackage**. When Eclipse deploys an ecore model plugin, it also registers the EPackage using a unique identifier (nsuri) in order to retreive it quickly. In Kermeta you can also use this nsuri into the require statement. This approach is useful because you can be sure that you require the very same model as Eclipse use to load and save models of that sort. For example, instead of requiring ecore.ecore you may use `require "http://www.eclipse.org/emf/2002/Ecore"`. This also works for all other registered metamodels (Your own registered metamodel, UML metamodel, etc). Kermeta user interface provides a view that display all registered EPackages and their nsuri.

2.5.4. Organizing code

Kermeta lets you organise your code the way you prefer. There is no predefined organisation of your code, but the language and the workbench proposes various mechanisms and tools to structure it according to your needs and the style you wish for your code. Typically if you use Kermeta internal weaver (see Section 2.20, “Weaving Kermeta code”), manually transform your ecore into Kermeta, eventually weaving the ecore using the merge button available on the worbench.

For example, you can put all your classes into a single file (like in Ecore, every classes is in the .ecore file) or you may create a file for each of them and then use the `require` to indicates that they must know each other. You can also use an organisation like java, with a file per class and a directory per package.

2.6. Using Variables

Whenever you need a data locally to a block because it doesn't goes into a class attribute or reference, you can use a local variable.

A variable is defined by a name and by a type. If needed, the name of the variable can be escaped using the tilda (~)

Declaring a variable:

```
var foo : String // This is a variable declaration
```

In the following example, we define 3 variables of type integer. The first is initialized with the "14" literal

value, the second is initialized with an expression using v1. For the last variable, we use a multiplicity notation to specify an ordered set of integer (see Section 2.14, “Collections” and Section 2.15, “Class properties” for more information on sets).

```
do
   var v1 : Integer init 14
   var v2 : Integer init 145 * v1

   var tab : Integer[0..*] init kermeta::standard::OrderedSet<Integer>.new
   v1 := v2/v1
end
```

Be careful to the multiplicity, when you create a variable with multiplicity, you have to initialize it with a collection. Then to use its content, you need to use the collection operation like add, addAll, remove, select, etc. If you use the assignment `:=` it will replace your collection.

2.7. Basic types

2.7.1. Primitive types

Kermeta implements a few primitive types. By primitive types, we mean types that are basic to any language, i.e integer, and that have a literal value. See below table.

Name	Description	Literal Example
Integer	Represents integer numeric value like 10 or 12. (« int » data type in Java)	101, 12, 14, -45, -123
String	Represents a string of like « helloworld » (« String » data type in Java)	"helloworld", "this is a string !!!"
Boolean	Represents a true/false value. (« boolean » data type in Java)	true, false

```
// Simple datatypes examples
var myVar1 : Integer init 10
var myVar2 : Integer
var myVar4 : String init "a new string value"
var myVar5 : boolean
```

Kermeta also supports some other primitive types like Float and Character, but they currently don't have a surface syntax for their literals. The way to get them is to use the convertion methods available on String (for both of them) or Integer (for Float).

For example:

```
var c : Character init "f".elementAt(0)
var r : Real init "4.5".toReal
```

2.7.2. Enumeration types

You can define enumerations using the following syntax. Note that each enumeration literal must end with a ";".

```
enumeration Size { small; normal; big; huge; }
```

You can manipulate enumerated variables and literals with classical operators such as in the following example.

```
var mySize : Size
if ( mySize == Size.small ) then
   stdio.writeln("This is small !")
end
```

> **Note**
>
> Enumeration is a concept of the same level as Class, they must both be defined in a package.

2.7.3. Local datatype using "alias"

In Kermeta, you can define your own datatype based on existing types without a hard dependency like inheritance.

This is done using the alias syntax.

Ex:

```
alias MyString : kermeta::standard::String;
```

In this sample, MyString has all the behavior of kermeta::standard::String but is declared locally. This means that you don't have any dependency to the framework, even to the String in the framework.

Obviously you can reuse existing type names :

```
alias String : kermeta::standard::String;
```

This will create a new type String in your package with the behavior of String in the framework.

The definition of an alias is different from the use of "using" statement (as defined in Section 2.13.3, “Syntaxic sugars”), when you write

using kermeta::standard

you simply defined a syntactical shortcut allowing you to access any definition in this package from within this file.

Wheras defining an alias allows you to access this new definition from another package if needed.

> **Tip**

> It is interesting to redefine your own datatype for all the standard type you use in your metamodel, so when you convert the file into ecore in order to have serialisation, you won't have any dependency to framework.ecore (which is the ecore version of the framework where Kermeta standard type) This allow a lazy coupling of the type definitions.

2.8. Classes and methods

2.8.1. Classes

As introduced in the "Hello world" example (see Section 1.2.1, “First Program”), Kermeta is an object-oriented language. Kermeta provides all MOF concepts like properties, attributes, package. In addition, it provides a body to operations and derived properties.

Classes and abstract classes can be defined in a Java-like way. Class definition must be placed into brackets as it is in the following example.

```
// an empty simple class
class myFirstClass
{

}

// a simple abstract class
abstract class MyAbstractClass
{

}
```

Additionally, for better code robustness, and more user-friendly programming, classes can use the genericity mechanisms (see Section 2.10, “Genericity” for more information).

```
// This is a parametric class
class A<G> {
}

// This is the type variable binding : G is binded with Integer
var a : A<Integer>
a := A<Integer>.new
```

There are some limitations in regards to Java. For example, you cannot define nested classes in Kermeta. Kermeta offers the same structural concepts than MOF language.

2.8.2. Defining operations

Kermeta provides a way to add operational (action) semantics to your metamodels. For that, you can define operations with their body, as in a classical programming language. You can also specify abstract operations (they have no body). Kermeta requires that every class that contains an abstract operation must be declared as an abstract class.

In the following example, we define some operations, based on a visitor pattern implementation

```
class Inventory
```

```
{
   operation visitLibrary(l : Library) is
   do
      writeln("Inventory : ");
      l.books.each(b : Book | b.accept(self))
      writeln("----")
   end

   operation visitBook(b : Book) is
   do
      stdio.write("Book : ", b.title, " by ")
      b.authors.each(a : Author | a.accept(self))
   end

   operation visitAuthor(a : Author) is
   do
      stdio.write(a.name, " ", a.lastName)
   end
}

class Library {
   // ...
   operation accept(visitor : Inventory) is
   do
      visitor.visitLibrary(self)
   end
}

class Book
{
   // ...
   operation accept(visitor : Inventory) is
   do
      visitor.visitBook(self)
   end
}

class Author
{
   // ...
   operation accept(visitor : Inventory) is
   do
      visitor.visitAuthor(self)
   end
}
```

In this small example we define an `Inventory` class which can go over the library structure and print books informations. For that, we apply the visitor GoF pattern's on the library structure defining an `accept` method in every library structures.

2.8.2.1. Result

The special variable `result` is used to store the value that will be returned by the operation.

```
operation getName() : String is
   do
      result := self.name
   end
```

> **Note**
>
> This is different of the `return` in java, since it doesn't end the block. Other instructions can be used

> after the `result` assignment.

2.8.2.2. Operations as main entry point.

When you run a Kermeta program you'll start from an operation. The main operation that you want to run can have any number of parameters, whose types must be only and only String. For operations that are **not** intended to be run, you can put any type of parameters.

Technically, if you use Kermeta inside Eclipse, when you will ask it to run your operation, the interpreter will implicitely instanciate the class containing this operation, and then will call this operation.

Example: 3 different kinds of "runnable" operations

```
class A{
   operation main0() is do
      // do something
   end
   operation main1( arg1 : String) is do
      // do something with 1rst argument
   end
   operation main3( arg1 : String, arg2 : String) is do
      // do something with 1st and 2nd arguments
   end
}
// If you ask to launch main0, Kermeta interpereter will create an instance of A and will run main0 on it.
```

2.8.3. Initializing classes

Kermeta doesn't use constructors.

However in some situation, you may need to provide some initialization operation. The best approach is simply to declare an operation which will return self after having done the initialization work.

```
class A
{
      attribute name : String
      operation initialize(name : String) : A is do
            self.name := name
      end
}
```

```
// now you can use it easily in one line
      var aA : A init A.new.initialize("hello")
```

2.8.4. Rationale

Here are some explanation about some design choice of Kermeta classes.

2.8.4.1. No constructor

This is because of the compatibility with EMOF and Ecore. In those metalanguages, the user must always be able to create object (except if they are declared abstract), it cannot rely on a action language (since they don't define one). In addition, we want that all Meta tool be able to create the object the same way, so, as Ecore doesn't provide constructor with its editors, then neither does Kermeta.

However, you can easily add an `init` or `myInitialize` operation that will take as many parameters as you want. Even better, you can use a Builder or a Factory design pattern.

```
aspect class ClassFromMyEcore {

    operation init(myParam : String) : ClassFromMyEcore is do
        // assign myParam to some attribute
        self.name := myParam
        // return self for easier use
        result := self
    end
    // now you can create and initialize this class in one line
    // var foo : ClassFromMyEcore init ClassFromMyEcore.new.init("foo")
}
```

2.8.4.2. No operation overloading

In order to simplify multiple inheritance management, Kermeta allows only one operation or property with the same name in a given class.

2.9. Inheritance

As Kermeta is object-oriented, it supports inheritance. Like its basis EMOF, it support both, simple and multiple inheritance.

2.9.1. Using simple inheritance

```
abstract class Person
{
  attribute name : string
  attribute lastName : string

  reference father : Male#children
  reference mother : Female#children
}

class Male inherits Person
{
  reference children : oset Person[0..*]#father
  reference wife : Female[0..1]#husband
}

class Female inherits Person
{
  reference children : oset Person[0..*]#mother
  reference husband : Male[0..1]#wife
}
```

In this example, we define a simple model which represent simples family trees. Here, persons are defined by their name and last name. Each person have a father and mother and respectively might have children. The figure representing this example is Figure 2.1, “A simple family tree model”

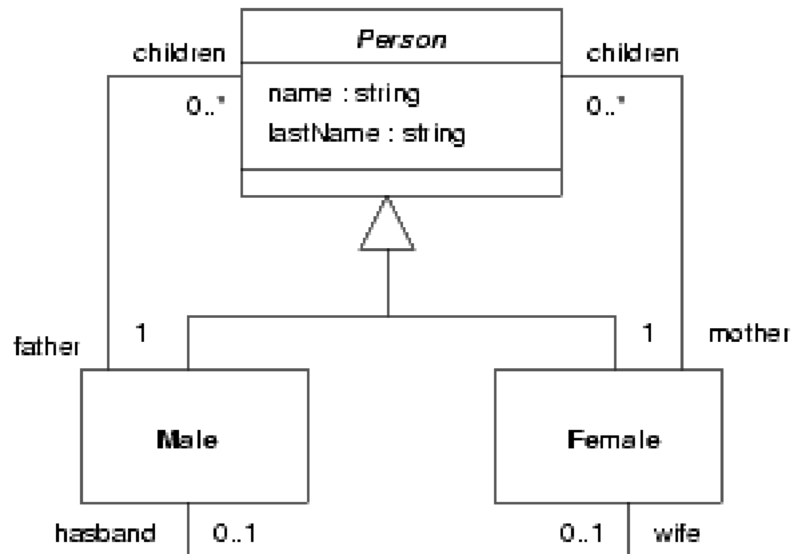


Figure 2.1. A simple family tree model

2.9.2. Cast

As Kermeta is strongly typed, you cannot assign or pass something of the wrong type.

For example :

```
class A {
}
class SubA inherits A {
}
class AnotherSubA inherits A {
}

// ...
    var aA    : A init SubA.new
    var aSubA : SubA
    aSubA := aA  // doesn't work because not type safe
```

In this situation you must use one of the following methods : conditional assignment or asType

2.9.2.1. Casting using conditional assignment

The conditional assignment `?=` allows to assign only if the passed object is of the correct type. If not of the correct type, the assigned value will simply be `Void`.

```
    aSubA ?= aA  // works
```

```
var aAnotherSubA : AnotherSubA
aAnotherSubA ?= aA  // works too, but has been assigned Void and not the value
```

This still propose some control on the types and won't accept all kind of cast, for example, you cannot conditionaly assign if you don't have a common supertype.

```
class B {
}
// ...
    var aB : B
    aB ?= aA  // doesn't work because not type safe
```

2.9.2.2. Casting using asType

You can also use the operation `asType` to cast your value. This one use a syntax which is shorter in some situation, since you don't have to create an intermediate variable before passing the casted value to an operation. However, the drawback is that, if it fails to cast, then it will raise an exception.

```
class C {
    operation needASubA(sa : SubA) : Void is do
        // do something
    end
}
    // ... typical code using ?=
    var aC : C init C.new
    aC.needASubA(aA.asType(SubA))
```

2.9.2.3. Useful functions when casting

In complement you can use the `isVoid` operation (available on every object) to test the result of a conditional cast. Or you may use the `isKindOf` or the `isInstanceOf` operations to test if the calling object has the same type of the given class before the assignment/asType. The `isKindOf` operation returns `true` if the calling object has exactly the same type than the given class. The `isInstanceOf` operation returns `true` if the calling object has the exact type or one of its super type than the given class.

2.9.3. Using multiple inheritance

```
class Parent
{
    reference children : oset Child[0..*]#parent
}

class Child
{
    reference parent : Parent#children
```

```
}

class Male { }
```

```
class GrandFather inherits Parent, Male
{
    boolean healthy : Boolean
}
```

The above example defines a class "GrandFather" that inherits at the same time the class "Parent", and the class "Male". Its graphical representation is shown in below figure.

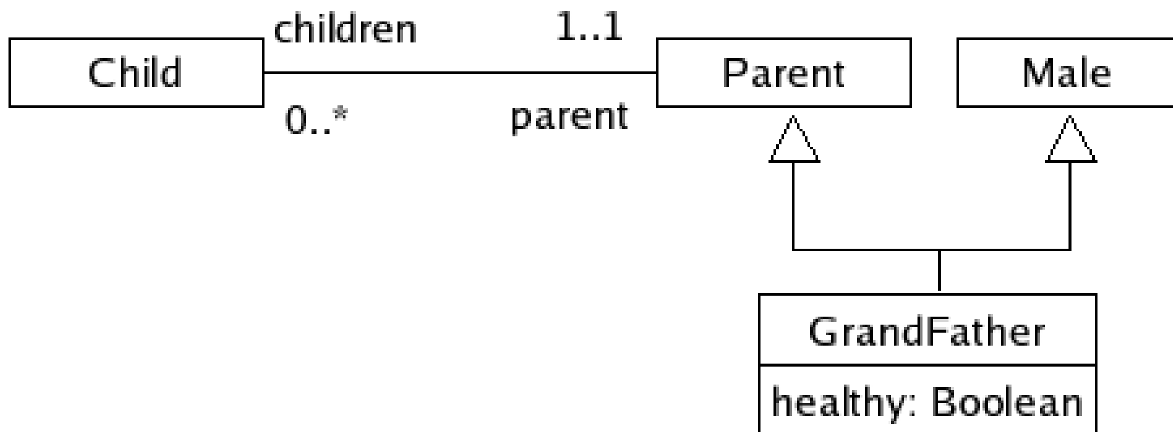


Figure 2.2. : multiple inheritance

2.9.4. Overriding behavior with methods

In the following sample, when an operation is declared for the first time (in the parent class Person), it uses the `operation` keyword. Then, whenever you override it, you will have to use the `method` keyword.

> **Note**
>
> In the sample,Person and adopt are abstract, but this has no influence on the operation-method keywords rule, it would have been the same even if Person was not abstract or if adopt had a behavior in this class.

```
abstract class Person
{
   attribute name : String
   reference father : Male#children
   reference mother : Female#children
   operation adopt(child : Person) is abstract}
```

```
class Male inherits Person
{
   reference wife : Female#husband
   reference children : oset Person[0..*]#father    method adopt(child : Person) is    do        children.add(child)
      if not wife.children.contains(child) then          child.father := self          wife.adopt(child)
      end    end}
```

```
class Female inherits Person
{
   reference husband : Male#wife
   reference children : oset Person[0..*]#mother    method adopt(child : Person) is    do        children.add(child)
      if not husband.children.contains(child) then            child.mother := self
         husband.adopt(child)
      end    end}
```

2.9.5. Some limitations

A MOF class can have operations but MOF does not provide any way to describe the behavior of these operations. Furthermore MOF does not provide any semantics neither for operation call nor for operation inheritance and redefinition. This section investigates how, while weaving actions into MOF, MOF semantics can be extended to support behavior definition and extension mechanisms provided by the action language. This implies answering several questions concerning redefinition and dispatch.

2.9.5.1. Operation redefinition

MOF does not specify the notion of overriding an operation because from a structural point of view it does not make any sense. To stick to MOF structure one can argue that redefinition should be forbidden in an executable MOF. This is the simplest solution as it also solves the problem of the dynamic dispatch since a simple static binding policy can be used.

However, operation redefinition is one of the key features of Object-Oriented (OO) languages. The OO paradigm has demonstrated that operation redefinition is a useful and powerful mechanism to define the behavior of objects and allow for variability. This would be very convenient to properly model dynamic semantic variation points existing in e.g. UML state-charts. For this reason we believe that an important feature of an executable MOF is to provide a precise behavior redefinition mechanism. The choice of the operation overriding mechanism must take into account the usual problem of redefinition such as method specialization and conflicting redefinitions related to multiple inheritance.

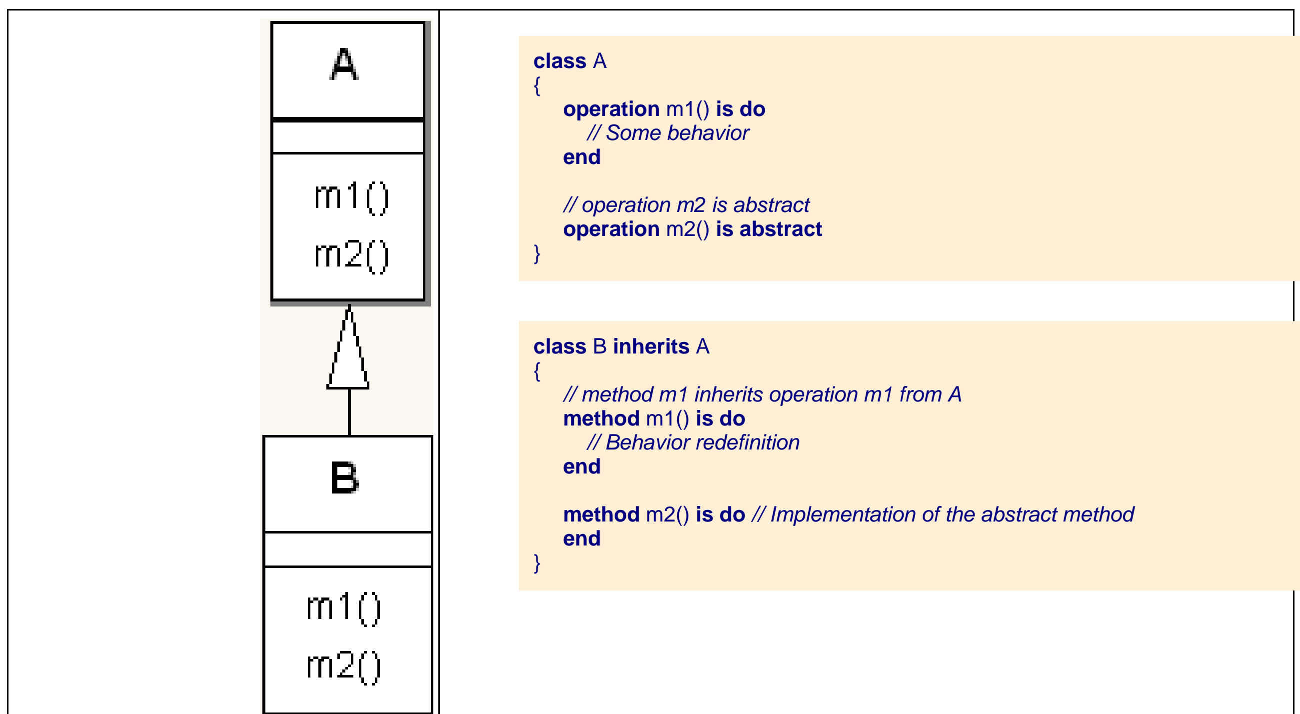


Table 2.1. Operation redefinition in Kermeta

> **Note**
>
> Notice in that sample that method redefinition uses the `method` keyword instead of `operation`.

2.9.5.2. Operation specialization

The issue of choosing semantics for operation overriding has been widely studied for the design of OO languages (cf. M. Abadi and L. Cardelli, A theory of objects, Springer). However, OO languages have not adopted a unique solution to this problem. In this context, any language that defines an operation overriding mechanism should define precisely the solution it implements.

The simplest approach to overriding is to require that an overriding method has exactly the same signature as the overridden method. That is that both the type of the parameters and the return type of the operation should be *invariant* among the implementations of an operation. For the sake of simplicity this is the solution we have chosen for the current version of Kermeta.

However, this condition can be relaxed to allow method *specialization,* i.e. specialization on the types of parameters or/and return type of the operation. On one hand, the return type of the overriding method can be a sub-type of the return type of the overridden method. Method specialization is said to be *covariant* for the return types. On the other hand, the types of parameters of the overriding method might be super types of the parameters of the overridden methods. Method specialization is thus *contravariant* for the parameters.

In practice languages can allow method specialization only on the return type (this is the case of Java 1.5) or both on parameters and return type (this is the case of Eiffel). Among these solutions, we may choose a less restrictive policy then strict invariance for future versions of Kermeta in order to improve the static type checking of Kermeta programs.

2.9.5.3. Operation overloading

Overloading is not allowed in Kermeta. This mechanism allows multiple operations taking different types of parameters to be defined with the same name. For each call, depending on the type of the actual parameters, the compiler or interpreter automatically calls the right one. This provides a convenient way for writing operations whose behaviors differ depending on the static type of the parameters. Overloading is extensively used is some functional languages such as Haskell and has been implemented in OO languages such as Java or C#. However it causes numerous problems in an OO context due to inheritance and even multiple inheritance in our case [REF?]. It is not implemented in some OO languages such as Eiffel for this reason, and that is why we chose to exclude overloading from Kermeta.

```
class A
{
   method m(i : Integer) is do
      // [...]
   end
   method m(s : String) is do // this is not allowed in Kermeta !!!
      // [...]
   end
}
```

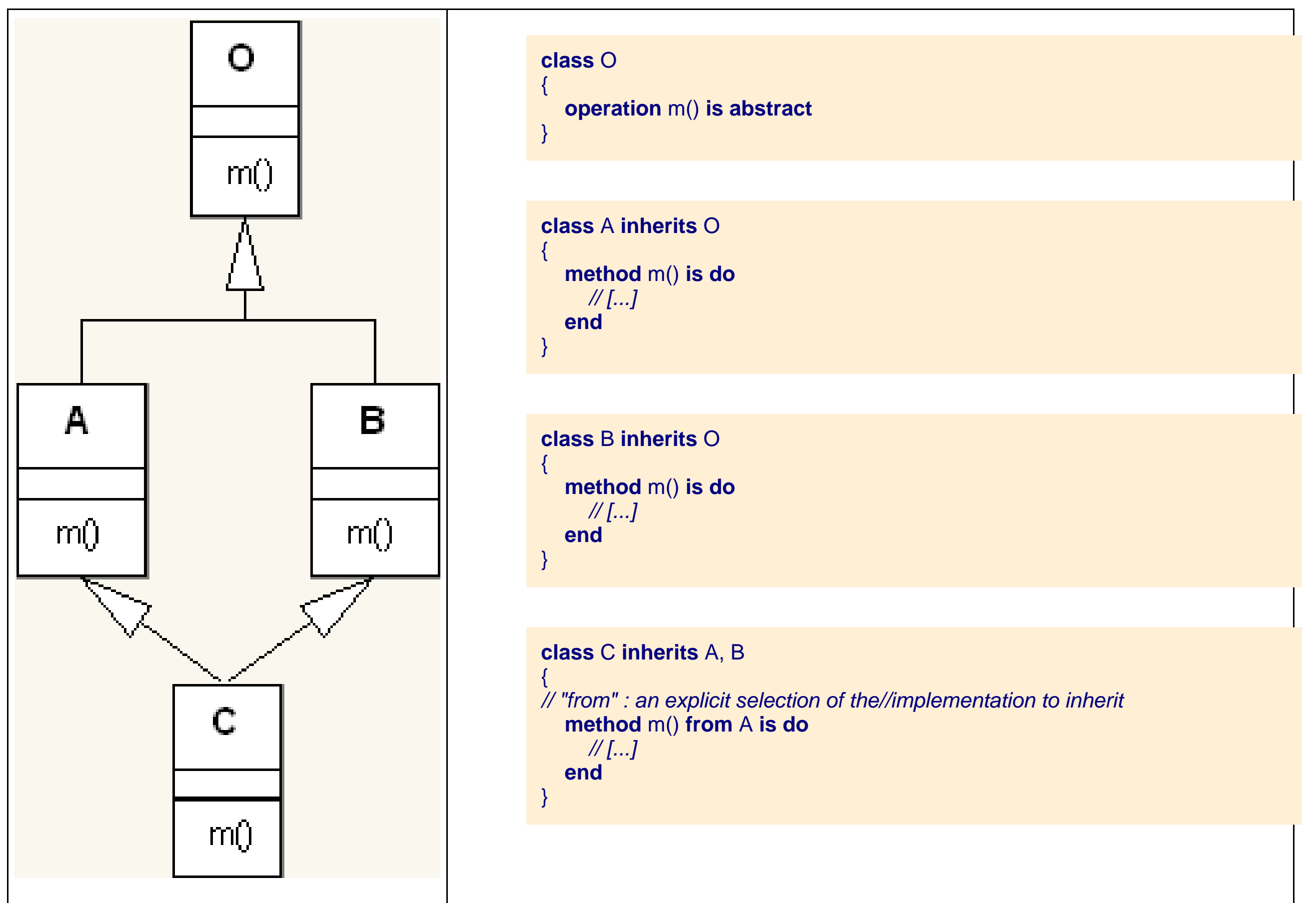
2.9.5.4. Conflicts related to multiple inheritance

This is also a classical problem that has been solved in several OO languages. There are mainly two kinds of conflicts when a class inherits features from several super-classes:

- Several features with the same name might be inherited from different super classes causing a name clash.
- Several implementations of a single operation could be inherited from different super classes.

There are two kinds of solutions to resolve these conflicts. The first one is to have an implicit resolution mechanism which chooses the method to inherit according to an arbitrary policy. The second one is to include in the language constructions that allow the programmer to explicitly resolve conflicts. In Eiffel, for instance, the programmer can rename features in order to avoid name clashes and can select the method to inherit if several redefinition of an operation are inherited from parent classes.

In the current version of Kermeta, we have chosen to include a minimal selection mechanism that allows the user to explicitly select the inherited method to override if several implementations of an operation are inherited. This mechanism does not allow resolving some name clashes and thus reject some ambiguous programs. For the future version of Kermeta we plan to include a more general mechanism such as *traits* proposed by Schärli et al. In any case we believe the conflict resolution mechanism should be explicit for the programmer.




```
class O
{
   operation m() is abstract
}

class A inherits O
{
   method m() is do
      // [...]
   end
}

class B inherits O
{
   method m() is do
      // [...]
   end
}

class C inherits A, B
{
// "from" : an explicit selection of the//implementation to inherit
   method m() from A is do
      // [...]
   end
}
```

Table 2.2. Explicit selection of super operation in Kermeta

2.10. 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 had to be made to the EMOF type system (Please refer to paper Weaving Executability into Object-Oriented Meta-Languages by P.A. Muller et al., presented at the Models05 conference).

As a result to these modifications genericity support has been added into Kermeta. Like Eiffel and Java 5, Kermeta supports generic classes and generic operations. This section gives on overview of these concepts in Kermeta.

2.10.1. Generic classes

In Kermeta classes 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.

```
// A class with a type variable G that can be bound with any type
```

```
class Queue<G>
{
    reference elements : oset G[*]

    operation enqueue(e : G) : Void is do
        elements.add(e)
    end

    operation dequeue() : G is do
        result := elements.first
        elements.removeAt(0)
    end
}
```

```
// A class with a type variable C that can be bound with any sub-type of Comparable
```

```
class SortedQueue<C : Comparable> inherits Queue<C>
{
    method enqueue(e : C) : Void is do
        var i : Integer
        from i := 0
        until i == elements.size or e > elements.elementAt(i)
        loop
            i := i + 1
        end
        elements.addAt(i, e)
    end
}
```

2.10.2. 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.

```
class Utils {
   operation max<T : Comparable>(a : T, b : T) : T is do
      result := if a > b then a else b end
   end
}
```

> **Note**
>
> Notice in that sample that even the "if" is an expression that can return a value that is assigned here to the special variable "result".

2.10.3. Type usable with genericity

Actually, all types can be used as a parameter of a generic class or generic operation. More, the ModelType, is really useful when combined with generics. see Section 2.21, “Model type”

2.11. Exception handling

Kermeta provides also an exception mechanism. You can define a "rescue" block to manage errors occurring during the execution of another block. Exception mechanism is very close to the Java Exception mechanism.

Example 1: a simple exception raising

```
do
   var excep : Exception

   excep := Exception.new
   stdio.writeln("Throwing an exception ! ")
   raise excep
end
```

Any block can then rescue exceptions.

Example 2: rescue block

```
var v1 : Integer init 2
var v2 : Integer init 3

do
   var v3 : Integer
   v3 := v1 + v2
rescue (myConstraintError : kermeta::exceptions::ConstraintViolatedInv)
   // something with myConstraintError
   // ...
rescue (myError : Exception)
   // something with myError
   // ...
end
```

> **Tip**
>
> do not hesitate to create "do .. end" block into another block if you want to check for an exception only in a part of the code. This also works if you want to rescue code from within a rescue code.

2.12. Loading and saving models

This section explains how to load and save (deserialize and serialize) an EMF model in Kermeta. For this purpose, we will use the following small example :

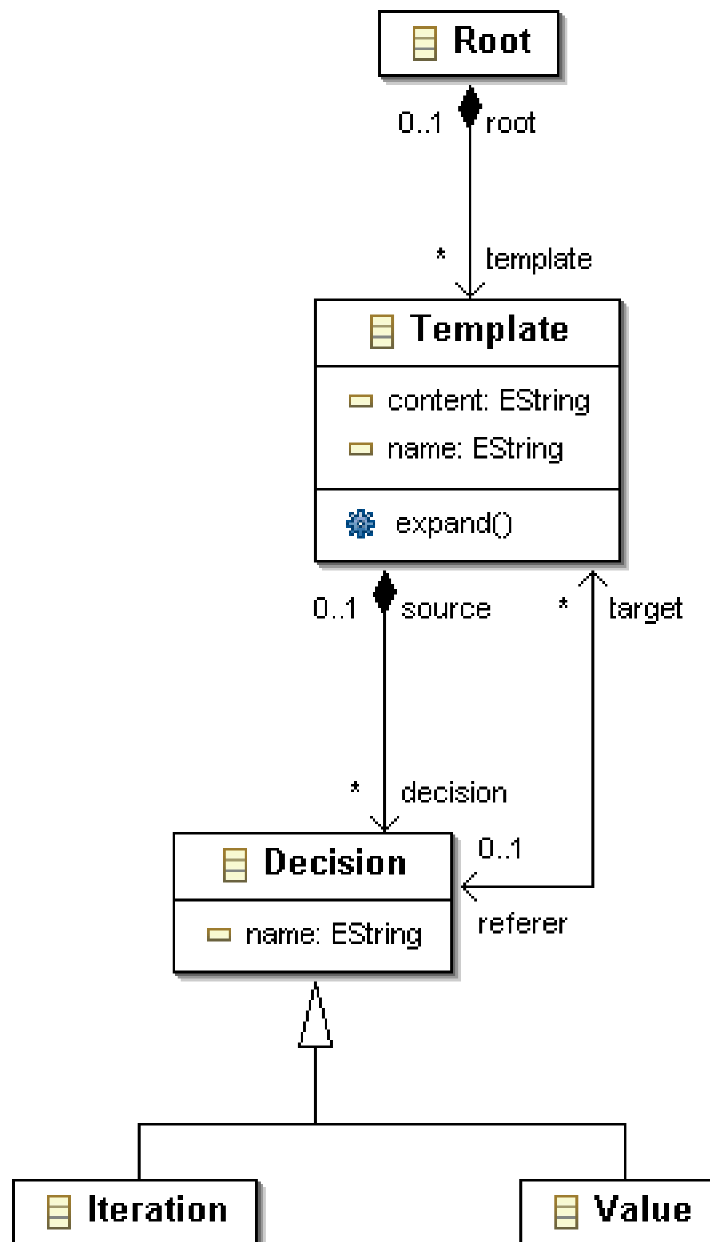


Figure 2.3. cs.ecore sample metamodel

> **Note**
>
> Loading and saving model has it own tutorial. It provides more information in a step by step approach. See the EMF tutorial at http://www.kermeta.org/documents/emfTutorial/

2.12.1. Prepare a model

The user will refer to Eclipse documentation for the creation of an EMF model from its ECore meta-model. We suggest to use the Wizard samples to create, at the first hand, an Ecore meta-model, and then, at the second hand, instances of this Ecore meta-model, using the generated EMF reflexive editors.

Once you created the ECore meta-model, please check that you correctly filled the property "`Ns URI`" of the root package of the Ecore meta-model, otherwise the resource load will fail. This NsURI must equal the real path of your metamodel. (You can modify this property through the Properties View of your meta-model)

2.12.2. Load a model from an EMF Resource

In the current version of the EMF resource loader, you have to prepare your EMF Resource following these rules :

- At the top of the source code where you will access your model, don't forget to add `require "your_metamodel.ecore"` so you can tell Kermeta that you will create/load/manipulate instances of these metaclasses.

  Alternatively, you can use the Kermeta version of your metamodel using `require "your_metamodel.kmt"` or `require "your_metamodel.km"`.

  In this case, be careful that your kmt required metamodel is strictly equivalent to the ecore version of your metamodel that is used in createResource method.

- Then, create a repository and the resource that will contain the model instance that you want to load. In the following code example, `uri` stands for the uri (as relative or absolute path[1]) of the model instance, and `mm_uri` is the uri of the meta-model of the model instance.

```
@mainClass "root::TestCSLoading"
@mainOperation "main"

package root;

require kermeta
require "cs.ecore"

using kermeta::standard
using kermeta::persistence

class TestCSLoading
{
    operation initialize(uri : String, mm_uri : String) : Set<Object> is do
        /* Initialize the EMF repository */
        var repository     : EMFRepository init EMFRepository.new
        /* Create an EMF Resource */
```

[1]in this c

```
    var resource : Resource init repository.createResource(uri, mm_uri)
    /* Load the resource */
    resource.load
    /* Get the loaded __root__ instances (a Set<Object>) */
    result := resource  // a resource is a collection of objects contained
  end
```

- Once you loaded your EMF resource, you can get its attribute `instances`, that contains all the objects that you created through your EMF generated reflexive editor. Now you can "visit" your instances, provided you "visit" them according to their types. In the simplest way, you can make very basic tests to display your instances content, as in the following example, which visit the objects of resource instances which types are `cs::Template` and `cs::Decision`.

```
operation main() is do
  var instances : Set<Object> init self.initialize("./test.cs", "./cs.ecore")
  instances.each{ o |
    if (o == void) then stdio.writeln("Void object!")
    else
      stdio.writeln("--------------------------------")
      stdio.writeln("Objet : " + o.getMetaClass.typeDefinition.qualifiedName
        + " ( " + o.getMetaClass.typeDefinition.ownedAttribute.size.toString+ "attr.)" )
    end
    var template : cs::Template // Print instances which type is cs::Template
    if (cs::Template.isInstance(o))
    then
      template ?= o
      stdio.writeln(" name : " + template.name)
      stdio.writeln(" decision : " + template.decision.toString)
      stdio.writeln(" content : " + template.content) stdio.writeln(" referer : " + template.referer.toString)
    end
    // Print instances which type is cs::Decision
    if (cs::Decision.isInstance(o))
    then
      decision ?= o
      stdio.writeln(" name : " + decision.name)
    end
  }
}
```

If your resource is dependent of other resources and that EMF succed to load it, the Repository that was used to load your resource will automatically load all these dependent resources.

2.12.3. Save a model into an EMF Resource

To save a model, simply add the model elements in a Resource then call the save operation. All model elements contained by these added elements will also be saved in the Resource.

You can split your model in several files by using several resources, but you need to make sure that they all belong to the same Repository. Otherwise, you'll get at best a Dangling exception, or worse create inconsistent files.

2.13. Packages

2.13.1. Definition

Kermeta provides package to structure models. So you can define packages and sub-package as deep as you want. There are two main ways to do this, as shown in examples 2 and 3 (below). If you want to define some classes in a package you may define them in a specific file and start this file with a package naming directive (followed by a semi-colon) like in the following example[2].

Example 1: 1 file.kmt == 1 package naming directive

```
// My file
package MyNewPackage;

class A {
// ...
}
```

```
class B {
// ...
}
```

Here, classes A and B are defined in a package called "MyNewPackage". All classes defined in this file are under the scope of this package.

You can also define explicitly sub-packages using braces (see the following example):

Example 2: Defining subpackages using braces

```
// file : MyPackage-part1.kmt
package MyPackage;

package subPackage1
{
   class A
   {
// ...
   }
}

package subPackage2
{
   class B
   {
// ...
   }
}
```

In this example, we define a main package called "MyPackage" which contains 2 sub-packages "subPackage1" and "subPackage2". The first one contains the A class and the second one the B class.

2.13.2. Use of packages

TODO : example of use inside code (without the "using" primitive)

[2]Each Kermeta file(.kmt) must declare a package directive.

2.13.3. Syntaxic sugars

If you want, you can declare the same package in several files. You can also directly define subpackages in the package naming directive of your file (see example 1 above). In the following example, we define a new sub-package of "MyPackage" called "subPackage3" directly in the file. All features defined in this file will belong to this sub-package.

Example 3: Defining subpackage using `::` syntactic sugar

```
// file : subPackage3.kmt
package MyPackage::subPackage3;

class C
{
// ...
}
```

If a given file, when you want to use a class definition that is not in the scope of the current package inside which you are working, you have to provide the full path of this class definition to be able to use it. However, you can also define shortcuts to make your code clearer: the `using` primitive provides such a shortcut. Taking the previous example, it becomes:

```
// file : MyPackage-part2.kmt
package MyPackage;

require "MyPackage-part1.kmt"

using subPackage1 // <- "shortcut"!

class C inherits A
{
// ...
}
```

The using statement is a syntactical shorcut, it has no conterpart in Kermeta metamodel.

> **Note**
>
> The using statement is different from the definition of a local datatype with an alias. (See Local datatype using "alias" subsection)

2.14. Collections

Collections are widely used in Kermeta not only because of their usual usage for collecting data in variables but also because they are used to represent class property when the multiplicity is greater than 1.

Collections is one of the concept where the genericity is the most visible since it greatly helps to write more robust code.

2.14.1. Definition and initialization

Kermeta defines some collection data types handling sets of values. The different available collection types

are the result of a combination of the constraints **unique** and **ordered**.

- Unique specifies a collection type that cannot contain doublet
- Ordered specifies a collection type where the position of an object inside the collection can be modified.

Name	Description	Constraints	
		Unique	Ordered
set	Represents an unordered collection of objects with no doublet (Set)	True	False
oset	Represents an ordered collection of objects with no doublet (OrderedSet)	True	True
seq	Represents an ordered collection of objects (Sequence)	False	True
bag	Represents a basic collection of objects (Bag)	False	False

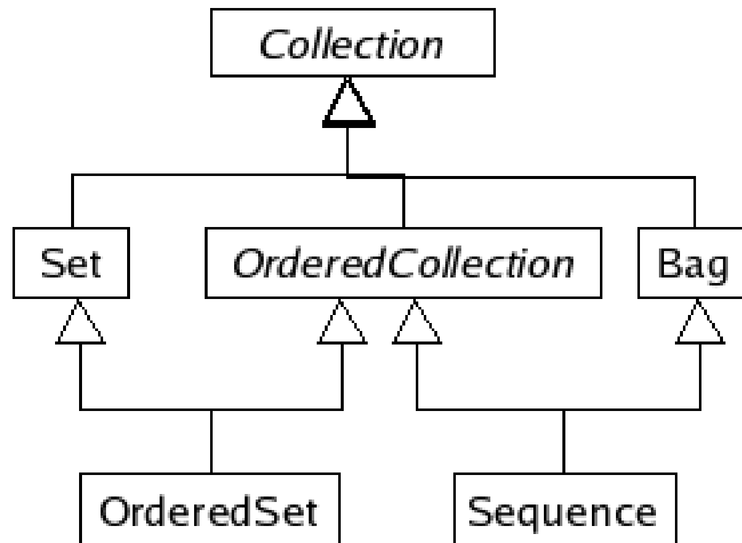


Figure 2.4. : The Kermeta collections

Another way to define set of objects would have been to use arrays. In fact, Kermeta does not define explicitly the concept of array, but it provides a multiplicity concept which can be used instead. Multiplicities are a way to define a lower and upper bound on a collection. Syntactically, lower and upper bounds are defined between brackets and are separated by two dots. Bounds can be an integer literal value or a star to specify there's no upper bound.

Example 1: how to declare collections

```
using kermeta::standard        // don't need to specify it all the time
    // This is the simpliest and recommanded way of declaring a collection variable
    var myColA : Set<Integer> // this is equivalent to saying set Integer[0..*]
    var myColB : OrderedSet<Integer> // this is equivalent to saying oset Integer[0..*]
```

```
    // Collection with multiplicities
  var myCol1 : set Integer[4..6] // At least 4 elements and never more than 6
    var myCol3 : seq String[2..*] // At least two strings in the sequence
    var myCol4 : set String[1..*] // An non empty set
    var myCol5 : String[1..*] // If you don't specify any keyword, it is an ordered set
```

There is currently no way to define a collection by extension like you can do in C or Java. You must initialize your collection either by calling `new` (Kermeta constructor operation) on your collection type, or initialize by copy.

Example 2: initialize collections

```
// Example of declaration of variables as Collections. All those syntaxes are valid
var myCol1 : set Integer[0..*]    init kermeta::standard::Set<Integer>.new
// Fill in myCol1
myCol1.add(10)
myCol1.add(50)

var myCol2 : oset String[0..*]    init kermeta::standard::OrderedSet<String>.new
var myCol3 : bag Boolean[0..*]    init kermeta::standard::Bag<Boolean>.new
var myCol4 : seq Integer[0..*]    init kermeta::standard::Sequence<Integer>.new
// if no keyword specified, and multiplicity is set, it is an OrderedSet
```

```
var myCol4 : String[0..*] init kermeta::standard::OrderedSet<String>.new
```

```
var myCol1a : seq Integer[0..*]    init myCol1
var myCol2a : oset String[0..*]    init myCol2
var myCol3a : kermeta::standard::Bag<Boolean> init myCol3
var myCol3a : kermeta::standard::Sequence<Integer> init myCol4
```

> **Note**
>
> Conclusion : in most cases, you don't need to use this special syntax, you can simply use the generic collection names (Set<Something>, OrderedSet<Something>, etc.) available in Kermeta framework. Moreover, lower and upper bounds aren't checked yet by Kermeta type checker and interpreter.

2.14.2. Some existing useful functions

The collections in Kermeta implement several functions based on lambda expressions (see Section 2.17, “Lambda Expressions and functions”). These ones are very useful for model navigation.

2.14.2.1. Function *each*

TODO

```
aCollection.each { e |
        /* do something with each element e of this collection */
}
```

2.14.2.2. Function *forAll*

TODO

```
aBoolean := aCollection.forAll { e | /* put here a condition */
      } // return true if the condition is true for all elements in the collection.
```

2.14.2.3. Function *select*

TODO

```
aCollection2 := aCollection.select { e |
         /* put here a condition that returns true for elements that must be included in the resulting Collection */
      }
```

2.14.2.4. Function *reject*

TODO

```
aCollection2 := aCollection.reject { e |
         /* put here a condition that returns true for elements that must be exclude in the resulting Collection */
      }
```

2.14.2.5. Function *collect*

TODO

```
// return a new collection which size is the same as in the original
                  // collection, and which element type is the type of the result of the expression.
                  aCollection2 := aCollection.collect { e |
                      /* put here an expression, for example e.name */
                  }
```

2.14.2.6. Function *detect*

TODO

```
anObject := aCollection.detect { e | /* a condition */} // returns an element (usually the first) that fulfills the condition.
```

2.14.2.7. Function *exists*

Function *exists* returns *true* if at least one element fulfills the last boolean expression defined into the block. For instance:

```
aBoolean := aCollection.exists { e | /* a condition */} // returns true if at least one element fulfills the condition.
```

If several boolean condition are defined into the block of function *exists*, the last condition is used for the test. For instance, the following function exists returns true at the first iteration:

```
aBoolean := aCollection.exists { e | false true }
```

Instructions can be defined into the block of function *exists*. For instance, the following code displays the current object at each iteration:

```
aBoolean := aCollection.exists { e |
                        stdio.writeln(e.toString)
                        /* a condition */
                  }
```

If no boolean expression is defined, value *Void* is returned, as illstrated in the following example. Such a use of function *exists* is not adviced since it corresponds to the use of function *each*.

```
aBoolean := aCollection.exists { e | stdio.writeln(e.toString) }/* The value of aBoolean is Void */
```

2.15. Class properties

A property of a (meta)class can be expressed in three ways : as a reference, an attribute, or a derived property. In Kermeta, each kind of these properties has a specific behavior and a dedicated syntax, which is **attribute** (for attribute), **reference** (for reference), **property** (for derived property)

References and attributes can have **opposite** properties. This last concept is explained in the following subsection.

Unlike UML, there is no concept of visibility in Kermeta, so every property is visible

2.15.1. Attributes *(attribute)*, references *(reference)*

We introduce here the 2 first cases, which are relationships between two concrete entities.

- *attribute*: an attribute defines a *composition* (e.g the black diamond) between two entities. The diamond-ed association end is navigable by definition

```
class A { attribute x : set X[0..*] }
class X {}
```

> **Note**
>
> Composition notion also relates to containment notion. So, there are some restriction about valid model. For example, in an association, only one end can be an attribute, otherwise this means that we have an association with a diamond on both end and we cannot say who is the container of the other.

- *reference*: a reference defines an association between two entities.

```
class A { reference x : set X[0..*] }
class X {}
```

2.15.2. properties modifiers

Attributes, references and properties underlying collections may eventually be specialized. By default, they are represented by an OrderedSet. If you wish to be more precise and for exemple allow to have several items with the same value in your collection, you can use the collection modifiers as defined in Section 2.14, “Collections”

For example :

```
class A {
	attribute x1 : seq String[0..*]  // allows duplicates, ordered
	attribute x2 : set String[0..*]  // doesn't allow duplicates, not ordered
	attribute x3 : bag String[0..*]  // allows duplicates, not ordered
	attribute x4 : oset String[0..*] // (default if no modifier) doesn't allow duplicates, ordered
}
```

2.15.3. Association using opposite properties

The opposite [property] of a property defines an association (bidirectional link) between two entities. An opposite is expressed by a sharp #. In the following example, `a` is the opposite property of the entity of type B, and `b` is mutually the opposite property of the entity of type A.

> **Caution**
>
> A property whose type is a DataType (i.e String, Boolean, Integer) cannot have an opposite!

Example 1: definition of an attribute, a reference, and an opposite property

This means that a can be accessed from b. Subsequently, if you modify the property b of an instance of A, it will mutually update its opposite property, i.e the property a of the instance of B contained in property b. You can make a test with the following example of code.

Example 2: navigability of opposite properties

```
var a1 : A init A.new
a1.name := "a1"
var b1 : B init C.new
a1.b := b1
```

```
stdio.writeln("b1 opposite : " + b1.a.name) // This prints "a1"!
```

The following paragraph shows a set of examples of attributes and references.

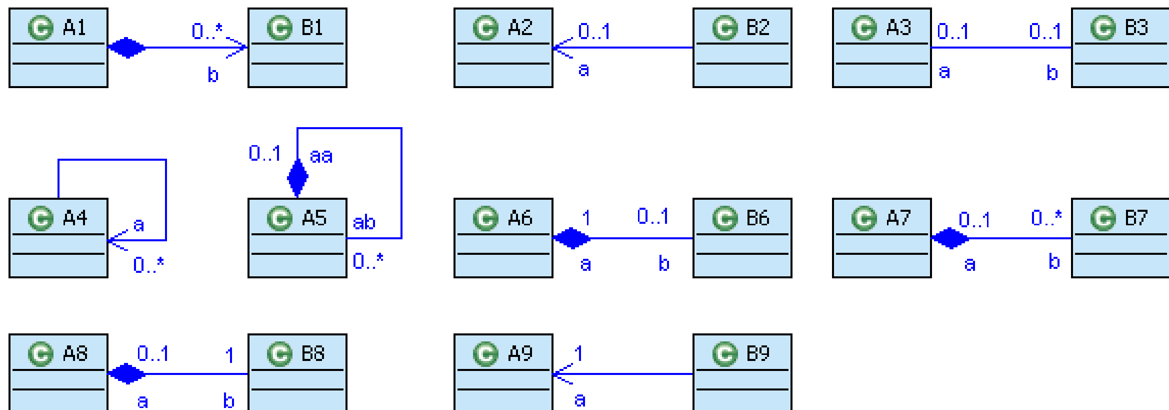


Figure 2.5. Attributes and references

Example 3: a set of attributes and references, with multiplicities and opposites.

TODO : add an example of code using those classes !! See https://gforge.inria.fr/plugins/scmcvs/cvsweb.php/integration_projects/other/org.openembedd.formations/Kermeta/Basics/docs/FR_formation_Kermeta_1er_niveau.odp?cvsroot=openembedd for such examples (p. 21).

```
package root;
```

```
class A1 {
    attribute b : B1[0..*]
}
class B1 {}
```

```
class A2 {}
class B2 {
    reference a : A2
}
```

```
class A3 {
    reference b : B3#a
}
class B3 {
    reference a : A3#b
}
```

```
class A4 {
    reference a : A4[0..*]
```

```
}
```

```
class A5 {
    attribute ab : A5[0..*]#aa
    reference aa : A5#ab
}
```

```
class A6 {
    attribute b : B6#a
}
class B6 {
    reference a : A6[1..1]#b
}
```

```
class A7 {
    attribute b : B7[0..*]#a
}
class B7 {
    reference a : A7#b
}
```

```
class A8 {
    attribute b : B8[1..1]#a
}
class B8 {
    reference a : A8#b
}
```

```
class A9 {}
class B9 {
    reference a : A9[1..1]
}
```

> **Note**
>
> For every case where the upper bound of a property is upper to 1, the type of the property is `OrderedSet` . The reader will refer to Section 2.14, "Collections" (except the bag type) to have the available types for a [m..n](n>1) multiplicity property.

2.15.4. Derived properties *(property)*

In a class definition, a derived property is a property that is derived or calculated, i.e it contains a body, like operations. Usually, such properties are calculated from other properties available from its owning class definition. In practice, you can define the code that you want inside a derived property.

In other words it does not reference to a concrete entity: it is calculated, through the accessor operations getter and setter.

The special parameter `value` is used in the setter to get the value passed when calling the setter.

Let's take the following class definitions :

```
// readonly property : it has no setter
class A
{
   attribute period : Real
   property readonly frequency : Real // property : keyword for derived property
      getter is do
         result := 1/period
      end
}
// modifiable property :
class B
{
   attribute period : Real
   property frequency : Real
      getter is do
         result result := 1/period
      end
      setter is do
         period := 1/value
      end
}

// a typical use would be (with aB an instance of class B)
var freq : Real
fred := aB.frequency // to use the getter
aB.frequency := freq + 1 // to use the setter, the period is also updated in the process
```

> **Note**
>
> To understand to role of the `value` keyword, you can imagine that in this sample the setter syntax is a shortcut syntax of : `setter( value : Real) is do ...` (even if actually this syntax isn't supported).

> **Warning**
>
> Since derived properties aims to behave like attributes or references, if the multiplicity is greater than 1, it doesn't make sense to define a setter. This is because it means reassigning the internal collection which is in fact calculated.

Properties are accessed or modified as classical properties. See next subsection for examples.

2.15.5. How to access and control the properties in Kermeta

Example 1: let's take the example with A6 and B6 :

```
class A6 {
    attribute b : B6[0..*]#a
}

class B6 {
   reference a : A6#b
}
```

Access	Kermeta expression
Get the attribute of an	`var a6 : A6 init A6.new` `var b6 : OrderedSet<B6>`

Access	Kermeta expression

instance	`// get the b attribute`
	`// Note that as the attribute as a multiplicity >1 it is an OrderedSet`
	`b6 := a6.b`

Add an element to a property with multiplicity [m..n], n>1	`var a6 : A6 init A6.new`
	`var b6 : B6 init B6.new`
	`// add b6 to the attribute b of A.`
	`// Note : you don't have to initialize b! done through A6.new`
	`a6.b.add(b6)`

Remove an element from a property	`// OrderedSet owns a method that removes an element given its`
	`// index in the set. For unordered sets, use "remove" method`
	`a6.b.removeAt(0)`
	`// Also valid : a6.b.remove(b6)`

Get the opposite of a property	`var a6 : A6 init A6.new`
	`var b6 : B6 init B6.new`
	`a6.b.add(b6)`
	`// this assertion is true. Moreover, any instance in a6.b will`
	`// have a6 as their opposite since b is a collection`
	`assert(b6.a == a6)`

Get the container of a property	`var a6 : A6 init A6.new`
	`var b6 : B6 init B6.new`
	`// add ab6 to the attribute "b"`
	`a6.b.add(b6)`
	
	`var a6c : A6 init b6.container()`
	`// this assertion is true`
	`assert(a6c.equals(a6))`

It is not different with references that have a [m..1] (m=0 or m=1) multiplicity:

Example 2: the A5 B5 case

```
class A5 {
   attribute b : B5#a // no multiplicity means [0..1]
}
class B5 {
   reference a : A5[1..1]#b
}
```

Access	Kermeta expression

Get the attribute of an instance	`var a5 : A5 init A5.new`
	`var b5 : B5`
	`// get the b attribute`
	`b5 := a5.b`

<table>
<tr><th>Access</th><th>Kermeta expression</th></tr>
<tr><td>Set a property (with multiplicity [m..1], m#1)</td><td><pre>var a5 : A5 init A5.new
var b5 : B5 init B5.new
// set b5 to the attribute b of A.
a5.b := b5</pre></td></tr>
<tr><td>Unset a property</td><td><pre>a5.b := void</pre></td></tr>
<tr><td>Get the opposite of a property</td><td><pre>var a5 : A5 init A5.new
var b5 : B5 init B5.new
a5.b := b5
// this assertion is true.
assert(b5.a == a5)</pre></td></tr>
<tr><td>Get the container of a property</td><td><pre>var a5 : A5 init A5.new
var b5 : B5 init B5.new
// add b5 to the attribute "b"
a5.b := b5</pre><pre>var a5c : A5 init b5.container()
// this assertion is true
assert(a5c.equals(a5))</pre></td></tr>
</table>

Note

Be careful with attributes or references ref with multiplicity greater than 1, they are automatically initialized with a reflective collection (ie. a collection that is aware of an eventual opposite). So, you cannot assign them using := If you wish to change completely its content with the one of another collection, you must use the `clear` and `addAll` operations.

Note

For the same reason,`reference refA : OrderedSet<String>` is different from `reference refB : String[0..*]`. If both can be navigated the same way (using add, each or any operation available on Collection), `refA` needs to be created before use with `refA := OrderedSet<String>.new`. `refA` is a reference to a colection whereas `refB` is a reference to String with a multiplicity greater than 1 (which is internally represented as a ReflectiveCollection<String>)

2.15.6. Assignment behavior for attribute (and reference)

Attribute and reference have one main behavior difference.

Attribute has a notion of containment that reference hasn't.

This has some implication on the behavior of the assignment because an attribute cannot be owned by more than one object at a time.

There is an exception for attributes which type is a primitive type (String, Integer, Boolean, Real, Char) : since those types inherit from `ValueType`, they are not concerned by the composition, opposite concepts. In this case, the assignment doesn't impact any value but the assigned one.

Example 1: Assignment behavior for attribute

```
class A { attribute c1 : C }
class B { attribute c2 : C }
class C { }
aA.c1 := C.new
aB.c2 := aA.c1     // now aA.c1 == void !!!
```

Example 2: Assignment behavior for reference

```
class A { reference c1 : C }
class B { reference c2 : C }
class C { }

aA.c1 := C.new
aB.c2 := aA.c1     // aB.c2 == aA.c1 and aA.c1 keeps its value !!!
```

Example 3: Assignment behavior for attribute which type is String

```
class A { reference c1 : String }class B { reference c2 : String }aA.c1 := "Robert"aB.c2 := aA.c1 // aB.c2 == aA.c1 == "Robert"
```

> **Note**
>
> The assignment into a `variable` or a `reference` is not a problem because it doesn't change the owner of the assigned object.

2.16. Objects comparison

Kermeta users could be confused about `==` operator and `equals` method. Unlike, Java they do the same and redefining the equals to compare the content also affects the `==`.

To compare the identity of two objects you must compare the identifier of each object : using their oid

2.16.1. equals method

The *equals* method behaves the same way as in java. This means that you can overwrite it, if you want to compare the contents of two objects of the same class.

```
@mainClass "root::Main"
@mainOperation "main"

package root;

require kermeta
using kermeta::standard
using kermeta::kunit
class A {
```

```
  reference x : Integer

  method equals(compared : Object) : Boolean is do
    var castedCompared : A
    castedCompared ?= compared
    result := x.equals (castedCompared.x)
  end

}


class Main inherits kermeta::kunit::TestCase{

  operation main() : Void is do

    var a1 : A init A.new
    var a2 : A init A.new

    a1.x := 10
    a2.x := 20

    assert( not a1.equals(a2) ) // objects are different with all methods
    assert( not a1 == a2 )
    assert ( a1.oid == a2.oid )

    a2.x := 10

    assert ( a1.equals(a2) )        // objects becomes equals
    assert ( a1 == a2  )        // also with ==
    assert ( a1.oid == a2.oid ) // but they physically are still different

  end

}
```

2.16.2. Comparison for Primitive Types

In Kermeta language, there is a notion of primitive type. Here is the list of primitive types :

- Boolean
- Integer
- String
- Character
- Real

We do not want two Integer objects with the same value to be different. What we want is to use == operator to compare values of primitive types. These classes simply redefine the equals. Then we can write this code :

```
var i1 : Integer init 10
var i2 : Integer init 10
assert (i1.equals(i2))

var i3 : Integer init 10
var i4 : Integer init 10
assert (i3 == i4)
```

2.16.3. Collection comparison

The method `equals` exists for the collections. The generic behavior is the following : for two collections a and b, if all elements of a are contained in b and vice-versa, then it returns true, false otherwise. This behavior is little bit different for ordered collection which takes into account the order of elements. Have a look to the following pieces of code :

```
operation test_try1() : Void is do
   var os : OrderedSet<Integer> init OrderedSet<Integer>.new
   os.add(1) os.add(2) os.add(3) os.add(4) os.add(5) os.add(6)

   var os2 : OrderedSet<Integer> init OrderedSet<Integer>.new
   os2.add(1) os2.add(2) os2.add(3) os2.add(4) os2.add(5) os2.add(6)

   assert (os.equals(os2) )
end

operation test_try2() : Void is do
   var os : OrderedSet<Integer> init OrderedSet<Integer>.new
   os.add(1) os.add(2) os.add(3) os.add(4) os.add(5) os.add(6)

   var os2 : OrderedSet<Integer> init OrderedSet<Integer>.new
   os2.add(1) os2.add(2) os2.add(3) os2.add(4) os2.add(5)

   assert ( not os.equals(os2) )
end

/**
 * An ordered set takes care about the order.
 */
operation test_try3() : Void is do
   var os : OrderedSet<Integer> init OrderedSet<Integer>.new
   os.add(1) os.add(2) os.add(3) os.add(4) os.add(5) os.add(6)

   var os2 : OrderedSet<Integer> init OrderedSet<Integer>.new
   os2.add(4) os2.add(6) os2.add(3) os2.add(1) os2.add(5) os2.add(2)

   assert ( not os.equals(os2) )
end

/**
 * A set does not care about the order.
 */
operation test_try4() : Void is do
   var os : Set<Integer> init Set<Integer>.new
   os.add(1) os.add(2) os.add(3) os.add(4) os.add(5) os.add(6)

   var os2 : Set<Integer> init Set<Integer>.new
   os2.add(4) os2.add(6) os2.add(3) os2.add(1) os2.add(5) os2.add(2)

   assert ( os.equals(os2) )
end
```

2.17. Lambda Expressions and functions

Kermeta proposes an implementation of lambda expressions, that is mainly useful for implementing OCL-like functions.

Note

> Limitations of lambda expressions usage:An operation with a function as parameter : if you define an operation which has a function as a parameter (ex: `operation op( func : <Integer->Object>)`), then you cannot put more parameters : only one parameter is allowed in such a case.

2.17.1. Syntax

Here is a definition of function syntax (this is not a formal representation)

- Signature of an operation defined as a function:

```
// With one parameter
operation FUNCTION_NAME ( LAMBDA_NAME : <TYPE->RETURN_TYPE> )
// With many parameters
operation FUNCTION_NAME ( LAMBDA_NAME : <[TYPE_1, TYPE_2, ...]->RETURN_TYPE> )
```

- Call of a function:

```
anInstance.FUNCTION_NAME ( TYPE | SOME_CODE )
anInstance.FUNCTION_NAME ( TYPE_1, TYPE_2 | SOME_CODE )
```

- Declaring and defining an "anonymous" function inside an operation body

```
var f : <TYPE -> RETURN_TYPE>
f := function { VAR_NAME : TYPE | SOME_CODE_WITH_RETURN_TYPE_RESULT }

var f : <[ TYPE_1, TYPE_2,... ] -> RETURN_TYPE>
f := function { VAR_NAME_1 : TYPE_1, VAR_NAME_2 : TYPE_2,... | SOME_CODE_WITH_RETURN_TYPE_RESULT }
```

In the following sections, you will find many examples of declarations, definitions, and uses of functions.

2.17.2. Some existing useful functions

The collections in Kermeta implement several functions based on lambda expression. These ones are very useful for model navigation.

Example 1: closure definition for collection iterator-like in Kermeta

```
aCollection.each { e |
   /* do something with each element e of this collection */
}
```

See Section 2.14, “Collections” for other existing functions on collections.

Example 2: another useful function that is defined on Integer : the function times

```
10.times { i | stdio.writeln(i.toString) } // prints 0 to 9
```

Notice that you can also write some complex code in the function, using internal variables, etc. In such a

case, the last statement evaluation will be the returned result of the lambda expression (provided it is declared to return something)

Example 3: complex code in a lambda expression

```
aCollection.each { e |
    stdio.writeln("I am complex code!"
    stdio.writeln("Element : " + e.toString)
    var i : Integer init 5
    i := i + 132458
}
```

Example 4: postponing parameter evaluation in the andThen operation

The operation andThen and orElse of Boolean use the ability of functions to postpone the parameter evaluation. This allows to implement the expected behavior without adding new construct to the language.

```
// this kind of code allows to avoid a cast exception on the second test
if cl.isInstanceOf(NamedElement).andThen{f| cl.asType(NamedElement).name == "foo"} then
        // do something...
end
```

> **Note**
>
> The parameter `f` is not used in this function. This is due to the current surface syntax of Kermeta which need at least one parameter. In this case, it's value is void.

2.17.3. Defining new functions in a class

You can also define your own functions, by declaring an operation, with a parameter as a function using the syntax described in Section 2.17.1, “Syntax”.

Example: definition of functions for collections

```
abstract class Collection<G>
{
    /** * runs func on each element of the collection */
    operation each(func : <G -> Object>) : Void is do
        from var it : Iterator<G> init iterator
        until it.isOff
        loop
            func(it.next)
        end
    end
```

```
    /** * checks that the condition is true on all the element of the collection * returns true if the collection is empty */
    operation forAll(func : <G -> Boolean>) : Boolean is do
        var test : Boolean init true
        from var it : Iterator<G> init iterator
        until it.isOff
        loop
           test := test and func(it.next)
        end
        result := test
    end
}
```

2.17.4. Defining lambda expression variables

You can also define lambda expression as variable. This can be useful if you don't want to (or can't) modify the class.

- A basic lambda expression

With one Integer argument and returning an Integer.

```
var aLambdaExp : <Integer->Integer>
var aLambdaResult : Integer
aLambdaExp := function { i : Integer | i.plus(4) }
// aLambdaResult equals 7
aLambdaResult := aLambdaExp(3)
```

- A lambda expression with several parameters

```
var aLambdaExp : <[Integer, Integer]->Integer>
var aLambdaResult : Integer
aLambdaExp := function { i : Integer, j : Integer | i * j }
// aLambdaResult equals 12
aLambdaResult := aLambdaExp(3, 4)
```

- A lambda expression on a collection

```
var sequence : Sequence<Integer> init Sequence
var init_set : Set<Integer> init Set<Integer>.new
init_set.add(32)
init_set.add(23)
init_set.add(41)

// This sequence equals : [320, 230, 41]
sequence := init_set.collect { element | element*10}
```

The code within the function can be as complex as you want, using internal variables, etc.

```
var factoExp : <Integer->Integer>
factoExp := function { n : Integer |
    var fact : Integer init 1
    from var x : Integer init 1
    until x > n
    loop
        fact := fact * x
        x:=x+1
    end
    fact // return fact as the result of the function
    //shorter alternative ;-)... if n<=1 then 1 else factoExp(n -1) * n end

}
var cnkExp : <[Integer, Integer]->Integer>
cnkExp := function { n : Integer, k : Integer |
    factoExp(n) / (factoExp(k) * factoExp(n-k))
}
```

2.18. Dynamic evaluation of Kermeta expressions

Kermeta allow you to evaluate dynamically a Kermeta Expression with a specific context.

Example 1: my first dynamic expression

```
var de : DynamicExpression init DynamicExpression.new
de.initializeDefaults
de.formalParameters.put("a", String)
de.parse("stdio.writeln(a)")
var params : Hashtable<String, Object> init Hashtable<String, Object>.new
params.put("a", "hello world!")
de.execute(void, params)
```

If you want to dynamically evaluate more than one statement, you will have to surround your set of statements with "do.. end" block:

Example 2: yet another example

```
// let's get previous example and modify it // [...] (carriage return is not necessary inside the block)
de.parse("do stdio.writeln(a) stdio.writeln("another stdio writeln ... ") end")
// [...]
```

Example 3: another more complex sample of dynamic expressions :

```
package testDynamicExpression;

using kermeta::interpreter
using kermeta::utils
using kermeta::standard

class TestMain
{

   operation TestToto() is do
      stdio.writeln("J'ai essayé de lancer testtoto!")
   end

   operation testDynExp() is do
      var dynExpr : DynamicExpression init DynamicExpression.new
      dynExpr.initializeDefaults()

      self.getMetaClass.ownedOperation.select{op| not( op.name.indexOf("Test")==-1)
and op.name.indexOf("All")==-1}

         .collect{op|op.name}.each{opName|

               stdio.writeln("execution de "+opName)
               dynExpr.initializeDefaults
               dynExpr.parse("testDynamicExpression ::TestMain.new." +opName)
               dynExpr.execute(void,Hashtable<String,Object>.new)
         }
   end
}
```

> **Caution**

> You cannot use "self" inside a dynamic expression

2.19. Design by contract (pre, post, inv contraints)

In Kermeta, a contract is specified by "pre" and "post" conditions and by the "invariant" constraint too.

2.19.1. Writing a contract

2.19.1.1. pre – post conditions syntax

A pre or a post condition is a boolean expression that may be a simple equality checking or a lambda expression. The Kermeta interpreter evaluates the body content like a boolean result

The "pre condition" are declared just before the "is" of the operation body

```
operation opName(c : String) : String

   // Declaration of the pre-condition
   pre notVoidInput is do
      c != void and c != ""
   end

   // Declaration of the post-condition
   post notVoidOutput is
      result != void and result != ""

is
   do
      // operation body
   end
```

> **Tip**
>
> If the body contains only one expression, the block declaration "do ... end" is not mandatory. If your block contains several instructions, the latest one will be evaluated as a boolean expression.

2.19.1.2. Invariant constraint syntax

An invariant constraint is declared anywhere in a ClassDefinition block.

An invariant declaration is a boolean expression that may be a simple equality checking or a lambda expression. The Kermeta interpreter evaluates the body content like a boolean result.

A very simple example :

```
class className {

   ...

   // Declaration of the invariant : deterministicTransition
   inv nameOfTheInvariant is do
```

```
      self.name != ""
   end

   ...
}
```

> **Tip**
>
> If the body contains only one instruction, the block declaration "do ... end" is not mandatory.
>
> ```
> // Declaration of the invariant : deterministicTransition
> inv nameOfTheInvariant is self.name != ""
> ```

A lambda expression can be used into an invariant declaration:

```
// Declaration of the invariant : deterministicTransition
inv deterministicTransition is do
   self.outgoingTransition.forAll{tr1 |
      self.outgoingTransition.forAll{ tr2 |
         ( tr2.input==tr1.input ) == (tr1==tr2)
      }
   }
end
```

2.19.2. Checking your constraints

2.19.2.1. Checking pre – post condition

The activation of the checking of the pre - post conditions depends of the run configuration, see the Kermeta UI user guide for more information.

If the boolean statement is evaluated to "false" then the pre or post condition is violated and an exception `ConstraintViolatedPre` or `ConstraintViolatedPost` is raised.

2.19.2.2. Checking invariant

In order to check the well-formedness rules of a model element, there are two methods in Kermeta. The first-one : `checkInvariants`, consists to check only the current model element and the second-one : `checkAllInvariants`, checks recursively the element being a containment link with the checked element.

```
theModelElement.checkInvariants
```

The checkAllInvariants operation is a recursive method which checks all the elements having a containment relation by transitivity with the checked element.

checkAllInvariants is used especially to check the validity of a model fragment or a complete model.

```
theModelElement.checkAllInvariants
```

If the boolean statement is evaluated to "`false`" then the invariant constraint is violated and an exception `ConstraintViolatedInv` is raised. This exception can be handled by a `rescue` call.

```
// Call the invariant verification
do
   theModelElement.checkInvariants
   rescue (err : ConstraintViolatedInv)
      stdio.writeln(err.toString)
      stdio.write(err.message)
end
```

2.20. Weaving Kermeta code

Since version 0.4.1, it is possible to write Kermeta code using a simplified Aspect Oriented approach.

Technically, you can declare classes as "aspects" that will contribute features (attributes, references, properties, operations, constraints) to an existing classes. In such situation, the definition of two classes that have the same qualified name will be merged into a single class in the interpreter memory.

This is a great help when you want to separate the concerns into several files. For example, you may have one file the strictly conforms to the structural part of your metamodel, one file containing the constraints for a given purpose and another file containing the operation and special extension to the metamodel for an interpreter.

Obviously, the merge will be successful only if there is no conflict between all the declared features.

2.20.1. Textual syntax for merging

The merge is driven by the qualified name of the element to merge. Two classes will be merged if they have exactly the same qualified name (packages names + class name)

In order to keep the compatibility with previous behaviour, the merge is allowed only if you add some keyword or tag:

aspect
: This keyword is placed on a class, it indicates that this class is an aspect of another one. This allows to complement a class with the features of the aspect class.

@overloadable "true"
: This tag is placed on an operation, it indicates that the body of the operation can be overloaded by another definition of the same operation in an aspect class. This is useful in some situation were the code expect to be overloaded.

> **Note**
>
> When using overloadable tag, for a given set of definition, the operation that is not tagged overloadable will overload all other definitions. If all the declaration declare to be overloadable, then the declaration order is important. The last declared will be used by the interpreter.

> **Note**

> Since version 1.0.0, there is a dedicated keyword *aspect*, with previous version this feature was possible using the tag *@aspect "true"* on classes.

2.20.2. Example 1: Simple Class merge

In this sample, writing:

```
// aspect1.kmt
package pack;
require kermeta
// this is the first aspect of class A
aspect class A {
	attribute decorations : Decoration[0..*]
}
// this aspect also declares a new class, visible only when requiring this aspect
class Decoration{
	attribute val : kermeta::standard::String
}
```

```
// base.kmt
package pack;
require kermeta
// this is the base class
class A {
	attribute name : kermeta::standard::String

	 // operation in this context have access only to the base's features
	operation getName() : kermeta::standard::String is do
		result := name
	end
}
```

```
// aspect2.kmt
package pack;
require kermeta
require "base.kmt"
require "aspect1.kmt"

// is this context we have access to all the features of class A : from base, aspect1 and aspect2
aspect class A {
	attribute id : kermeta::standard::Integer
	operation getFullSpecification() : kermeta::standard::String is do
		result := name
		result := "(" + id.toString + ")"
		decorations.each{ decoration | result := result + "<" + decoration.val+ ">" }
	end
}
```

> **Note**
>
> Please note that the visibility of the features is respected. In base.kmt, class A doesn't know about attributes id or decorators.

> **Tip**
>
> In the previous example we aspectize a *.kmt, we could choose to aspectize the corresponding

> *.ecore file.
>
> In fact, the both formats are available: *.kmt or *.ecore.

from the point of view of aspect2.kmt (ie. any file that requires aspect2.kmt) is equivalent to write :

```
class A {
	attribute name : kermeta::standard::String
	attribute decorations : Decoration[0..*]
	attribute id : kermeta::standard::Integer

	 // operation in this context have access only to the base's features
	operation getName() : kermeta::standard::String is do
		result := name
	end
	operation getFullSpecification() : kermeta::standard::String is do
		result := name
		result := "(" + id.toString + ")"
		decorations.each{ decoration | result := result + "<" + decoration.val+ ">" }
	end
}
class Decoration{
	attribute val : kermeta::standard::String
}
```

> **Note**
>
> When there is no conflict like in this sample, the order used to declare the features doesn't changes the final behavior of the code.

2.20.3. Example 2: merge with feature redefinition

This sample shows that if the signature are strictly equivalent, then you can redefine the same structural feature (attribute, reference) in several aspect class.

In addition, if an operation is declared abstract, it can be an aspect of another one which is concrete. This is useful in order to add pre or post conditions to a given operation.

Currently, (v1.0.0) the derived properties cannot be redefined in several aspect classes.

```
aspect class A {
	//the feature is equivalent to the one define in the base class, so this is legal
	attribute name : kermeta::standard::String

	// the operation has the same signature and is abstract, so this is legal
	operation getQualifiedName() : kermeta::standard::String
		post notVoid is not result == Void
		is abstract
}
```

```
package pack;
require kermeta
class A {

	attribute name : kermeta::standard::String
	operation getQualifiedName() : kermeta::standard::String is do
		result := name
	end
}
```

2.20.4. Example 3: merge with operation overload

This sample shows the use of the overloadable tag

```
package pack;
require kermeta
aspect class A {
	@overloadable "true"
	operation getQualifiedName() : kermeta::standard::String
		post notVoid is not result == Void
		is abstract
}
```

```
package pack;
require kermeta
aspect class A {
	@overloadable "true"
	operation getQualifiedName() : kermeta::standard::String is do
		raise kermeta::exceptions::NotImplementedException.new
	end
}
```

```
class A {
	attribute name : kermeta::standard::String
	operation getQualifiedName() : kermeta::standard::String is do
		result := name
	end
}
```

In this sample, the first declaration of the operation is adds a post condition to the operation. As it is abstract, it isn't in conflict with the other declarations.

The second declaration, has a body. It implements a kind of default behavior for the operation that will be used if no other body is declared for this operation. this is useful when converting ecore models into Kermeta for example.

The last definition is the one that will be used in this context.

2.20.5. Aspect and inheritance

If you define some inheritance in one of your aspects, all the aspects that require this aspect will have access to it. You don't need to redefine it. However, if it helps to clarify the code to write it again, it will behave the same way.

2.20.6. Aspect and abstract

In Kermeta textual syntax, the keyword aspect must be placed before the keyword abstract.

Example :

```
package pack;
require kermeta
aspect abstract class A {
	// add something
```

```
}
```

2.20.7. Aspect without common base

A special case of the use of AOP in Kermeta is having a ClassDefinition A defined in 2 separately modeling units (*.kmt) with the keyword aspect. But the two modeling units does not require another kmt or ecore file containing the base definition of A. Now if there is a new Modeling Unit that requires the two previous one the available definition of A is the composition of the 2 defined aspects.

2.21. Model type

Warning

In current version of Kermeta(v1.1.0), ModelType is still a prototype. Any help, feedback, contribution is welcome to help us to finalize it.

Kermeta is clearly a model oriented language, one of its first concept is a model element, because they are the manipulated objects. However, we often need to organise them and manipulate them as a set : a model. A common way to do that is to simply use a set, and a resource (used for load and save) is just a set. This approach is not type safe enough.

So, Kermeta implements a clear notion of model type, in order to be able to typecheck this kind of model. A model type is simply the set of the type of all allowed model elements.

In addition, it introduce a mechanism of conformance between modeltype that helps to reuse code from one metamodel to another.

2.21.1. Definition of a model type

A model type is simply a set of metaclasses. Expressed in Kermeta, this is a collection of Kermeta classes.

```
package aMetamodel;
// let's define some normal classes
class C1 {
	attribute name : String
	attribute aC2 : C2#aC1
}
class C2 {
	reference aC1 : C1#aC2
}

// then defining a modeltype is just listing the classes of this modeltype
// here, the model type is composed of two metaclasses, C1 and C2
modeltype MainMT { aMetamodel::C1, aMetamodel::C2}
```

Tip

You can use all classic way to define classes, in kmt, but you can also require an ecore file that will

> contain your class definition (see require section ???)

2.21.2. Using Model types variables

Now that you have defined a ModelType, you can use it to declare model variables.

```
var aMainMT : MainMT init MainMT.new
```

But, your model is still empty. As its contents is a (constrained) Set you simply have to fill it with your model element using typechecked operations like `add`, or using filtered operation like `addCompatible` and `addAllCompatible` (with this one you can pass any collection, only compatible objects will be added to the model).

```
// create a model element
   var newC1 : aMetamodel::C1 init aMetamodel::C1.new()
   newC1.name := "hello"
   // then add it to the model
   aMainMT.add(newC1)
```

2.21.3. Model type serialisation

As a model type is based on "normal" class definitions, loading and saving models mostly relies on the normal way to load resource (see Section 2.12, “Loading and saving models”)

A simple approach for loading is to load with the classical approach, then to use the operations `addAllCompatible` with the content of the resource or `resourceToModel` with the resource. That will fill the model with compatible objects.

```
// load a resource with model elements
var res : EMFRepository init EMFRepository.new
var resource : Resource init res.createResource(modelFileName, ecoreMetamodelFileName)
resource.load()

// then add them to the model
aMainMT.addAllCompatible(resource.contents)
// alternative
// aMainMT.resourceToModel(resource)
```

TODO sample of saving (how to put root objects of the model into a resource)

2.21.4. Model type conformance

TODO explain how Modeltype helps to reuse existing code from a metamodel to another

Model types use a notion of conformance that allows to determine if a metamodel is conformant to another. The goal is to be able to write code that will be valid not only for models of a given metamodel, but will be

valid for models of conformant metamodels.

In order to be flexible, the conformity between metamodel is based on the properties offered by the metaclasses. It ignores the name of the metaclasses.

As a sample, let's define a mini metamodel and its corresponding model type :

Note

We could also define a modeltype directly on top of the previous metamodel (and restrain it to some of its metaclasses), but the sample would have been less general and wouldn't have illustrated the fact that the name of metaclasses are ignored.

```
modeltype MiniMT_2 { aMetamodel::C1 }
```

Warning

The current implementation works only when there is no ambiguity, if there is ambiguity between two metamodels, then they are considered as not conformant. A future version, may eventually introduce a binding mechanism that would allows to remove those ambiguity. (See Jim Steel Phd thesis for more details ???)

2.21.5. Using Model types in generic classes/operations

TODO

2.21.6. A more complex example of conformance : FSM variants

TODO reuse JM slides and the conformity table

2.22. Using existing java code in Kermeta

If you have existing code that you want to run in a Kermeta program, you can use one of those two mechanisms : extern call or seamless require.

The extern call is currently the more robust approach as it is used internally by Kermeta to implement some part of its framework. It also helps to clearly specify the border between java and Kermeta.

The seamless java require is more straitghforward to use (no wrapper to write) but is still in prototype (v0.4.2) and still have several limitations.

2.22.1. Using extern to call java code

The extern allows you to call a java static method from Kermeta. But, to do that, you will have firstly to create a Java wrapper, that will be able to manipulate correctly the Java objects, and secondly to add this wrapper in your java global classpath.

Then, from this method you can access all your java libraries. One task of the static method will be to convert the basic types like Integer or String.

You'll need to refer to the Javadoc of the interpreter in order to know how to access the internal RuntimeObject of Kermeta.

Example 1: sample of Kermeta code using extern (io.kmt):

```
/** * An implementation of a StdIO class in Kermeta using existing Java: standard * input/output */
class StdIO
{
 /** * write the object to standard output */
 operation write(object : Object) : Void is do
   result ?= extern fr::irisa::triskell::kermeta::runtime::basetypes::StdIO.write(object)
 end

 /** * writeln the object to standard output */
 operation writeln(object : Object) : Void is do
   result ?= extern fr::irisa::triskell::kermeta::runtime::basetypes::StdIO.writeln(object)
 end

 /** * read an object from standard input */
 operation read(prompt : String) : String is do
   result ?= extern fr::irisa::triskell::kermeta::runtime::basetypes::StdIO.read(prompt)
 end

}
```

Example 2: sample of Java code ("wrapper") called by the Kermeta extern:

```
/** Implementation of input and output methods */
  public class StdIO{
    // Implementation of method write called as :
    // extern fr::irisa::triskell::kermeta::runtime::basetypes::Io.write(output)
    public static RuntimeObject write(RuntimeObject output) {
      output.getFactory().getKermetaIOStream().print(output.getData().get("StringValue"));
    return output.getFactory().getMemory().voidINSTANCE;
    }
    // Implementation of method writeln called as : // extern fr::irisa::triskell::kermeta::runtime::basetypes::Io.writeln(output)
    public static RuntimeObject writeln(RuntimeObject output) {
      write(output);
      output.getFactory().getKermetaIOStream().print("\n");
    return output.getFactory().getMemory().voidINSTANCE;
    }
    // Implementation of method writeln called as : // extern fr::irisa::triskell::kermeta::runtime::basetypes::Io.read(output)
    public static RuntimeObject read(RuntimeObject prompt) {
      java.lang.String input = null;
      // We also have our own String wrapper
    if (String.getValue(prompt).length()>0)
      prompt.getFactory().getKermetaIOStream().print(String.getValue(prompt));
    // FIXME : dirty cast.. read returns a String or could return smthg else?
    input = (java.lang.String)prompt.getFactory().getKermetaIOStream().read( String.getValue(prompt));
    RuntimeObject result = String.create(input, prompt.getFactory());
    return result;
  }
```

Tip

This method is used to implement Kermeta framework. You'll find much more code samples of extern call in its sources.

2.22.2. Requiring jar file to call java code

Warning

This feature is still a prototype and still have many limitations. The first version is already available but any help is welcome to help us to improve it.

The basic principle, is to simply require your jar file.

```
require "yourjar.jar"
```

Then Kermeta automatically retreive the class definition it contains to be used from Kermeta code. However, as java and Kermeta have different language constraints, some adaptation are automatically made.

If there is several operations with the same name (as this is legal in Java but not in Kermeta) they are renamed. (Please use the outline to find the new name of the operation you want)

Java contructors are generated as "initialize" operation.

When creating a new java object from Kermeta, a call to new is not enough, you also need to call one of the "initialize" operation in order to correctly create it.

In order to get the standard library direclty from your running java, you can `require java_rt_jar` but as this library is really big and as you probably don't need all java from Kermeta ;-) then you must use the includeFilter and excludeFilter.

Sample using java.io from java Standard library

```
require kermeta

require java_rt_jar includeFilter ("java::io")   // the filter ensure we don't get all java

using java::io
using kermeta::kunit
class testRequireJava inherits kermeta::kunit::TestCase
{
    operation main() : Void is do
        var tr : TestRunner init TestRunner.new
        tr.run(testRequireJava)
        tr.printTestResult
    end

    operation testmain() : Void is do

        // create and initialize a File with the String
        var f : File init File.new.initialize_String("c:/temp/test.txt")
        var f2 : File
        // create and initialize a FileWriter with the File
        var fwriter : FileWriter init FileWriter.new.initialize_File(f)
        if (f.exists) then
            stdio.writeln(f.toString + " already exists")
        else
            stdio.writeln(f.createNewFile.toString)
        end
```

```
            fwriter.write_String("Hello world")
            fwriter.close
            stdio.writeln("file written")
            stdio.writeln(fwriter.toString)

            stdio.writeln(f.getPath)
            stdio.writeln(f.separator)
            f2 := f
            stdio.writeln(f2.createNewFile.toString)
            stdio.writeln((f2.equals(f)).toString)
            assert( f2.equals(f))

            var fwriter2 : FileWriter
            fwriter2 :=  fwriter
            stdio.writeln((fwriter2.equals(fwriter)).toString)
            assert( fwriter2.equals(fwriter))

            stdio.writeln(f.getPath)
            stdio.writeln(f.toString)
            stdio.writeln("End")
        end
}
```

`includeFilter` and `excludeFilter` accept a comma separated list of qualified name. `includeFilter` adds only elements whose qualified name start with one of the list. `excludeFilter` removes elements whose qualified name start with one of the list. If you use a combinaison of `includeFilter` and `excludeFilter`, then the `includeFilter` is applied before the `excludeFilter` (that'll remove element from the included one.

Currently known limitations: no support for java5 generics (they are ignored), requiring very big jar like the full java library end up with out of memory error (you need to use the includefilter and excludeFilter), some bugs with some primitives types (double/float)

2.23. Cloning objects

As we saw in previous sections, class properties can be defined as attribute or reference. An attribute cannot be shared between two or more objects whereas a reference can be. Let's consider a class "`Family`" with a property "`father`" (of type "`Person`") defined as an attribute. In the following example, we defined two objects of type `Family` and we want to define the `father` attribute of the second with the `father` of the first. To do that, we need to clone the object Person which represents the father of "`family1`" because, as said in Section 2.15.6, “Assignment behavior for attribute (and reference)”, it could not be shared between the two objects, by definition of attribute (in "technical" words, an object cannot be contained by 2 containers).

```
class Person
{
    attribute name : String
}
```

```
class Family
{
   attribute father : Person
}

class Main
{
   operation run() is
   do var family1 : Family init Family.new
```

```
      var p1 : Person init Person.new
      p1.name := "Robert"
      family1.father := p1
 var family2 : Family init Family.new

      // ERROR 1 : this assigns p1 to family2.father, which
      // is already owned by family1.father, so it unsets family1.father
      // family2.father := p1

      // ERROR 2 : this assigns family1.father's value to family2.father,
      // so it unsets family1.father
      // family2.father := family1.father

      // This is correct! family2.father keeps its value
      family2.father := Person.clone(p1)
   end
}
```

The "`clone`" method creates a copy of the object that it receives as input. If it is a complex object, a deep clone is performed for each attribute of its meta-class and a shallow clone is performed for each reference.

> **Caution**
>
> Reminder : be very careful with the use of the assignment operator on object. Most of the time, you need to use the "clone" feature. Using assignment on attributes break the previous link between objects. So, In the previous example, p1 has no more name after the assignment !There is one exception to this behavior : when the type of attributes are DataType, i.e, in Kermeta, String, Integer, Boolean, the assignment behaves as if those entities were defined as references.

CHAPTER 3

Kermeta Metamodel

As Kermeta is designed to be used in a model driven environment, its structure is given as a model. This section presents the metamodel of Kermeta which corresponds to the abstract syntax of Kermeta.

This metamodel may be useful for many use cases. For example, you can use it to manipulate your Kermeta code for analysis or even generate some Kermeta code. This may be useful to understand how Kermeta works too.

> **Note**
>
> All the code samples in this section are for illustration of the given concepts.

The goal of the Kermeta language is to provide an action language for MOF models. The idea is to start from MOF, which provides the structure of the language, and to add an action model. Using the MOF to define the structure of the Kermeta language has an important impact on the language. In fact, as MOF concepts are Object-Oriented concepts, Kermeta includes most of the classical Object-Oriented mechanisms. Yet, MOF only defines structures, and the operational semantic corresponding to MOF concepts has to be defined in Kermeta. For instance MOF does not provide a semantic for behavior inheritance (concepts like method redefinition, abstract method, etc does not have any sense in the MOF).

3.1. Architecture

Kermeta has been designed to be fully compatible with the OMG standard meta-data language EMOF. The metamodel of Kermeta is divided into two packages :

- *structure* which corresponds to EMOF
- *behavior* which corresponds to the actions. This section gives an overview of these two packages and their relationships.

Thanks to this reuse, Kermeta is fully compatible with EMOF. This is useful in the promotion of Kermeta as a metalanguage.

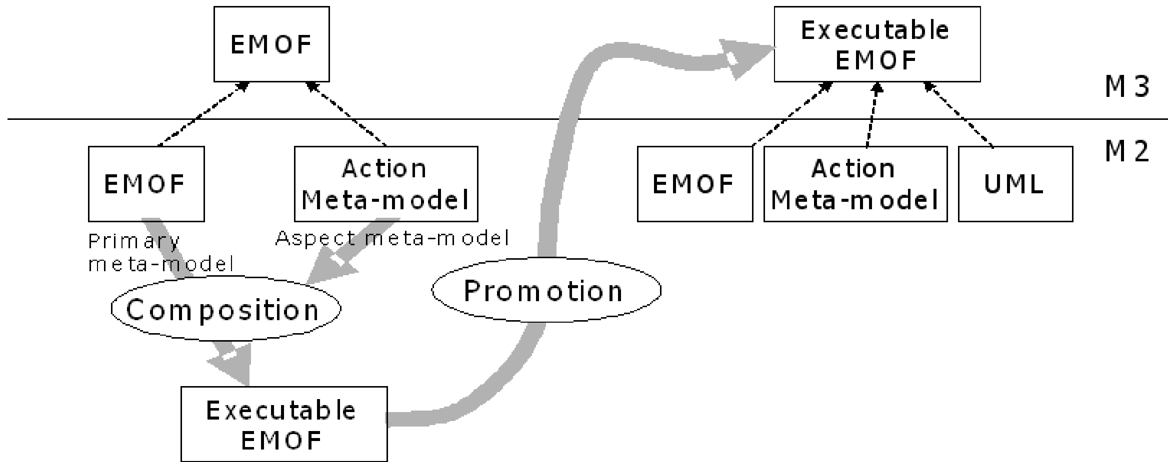


Figure 3.1. EMOF extension and Kermeta promotion

Note

This weaving of behavior into MOF has been explained in Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. -- Weaving executability into object-oriented meta-languages. -- In S. Kent L. Briand, editor, Proceedings of MODELS/UML'2005, volume 3713 of LNCS, pages 264--278, Montego Bay, Jamaica, October 2005. Springer.

The link between structure and behavior is made through the property `body` of class `Operation` which allows to define the behavior of an operation using a Kermeta expression.

A more detailed description of the architecture of Kermeta is presented in next sections.

3.2. Structure package

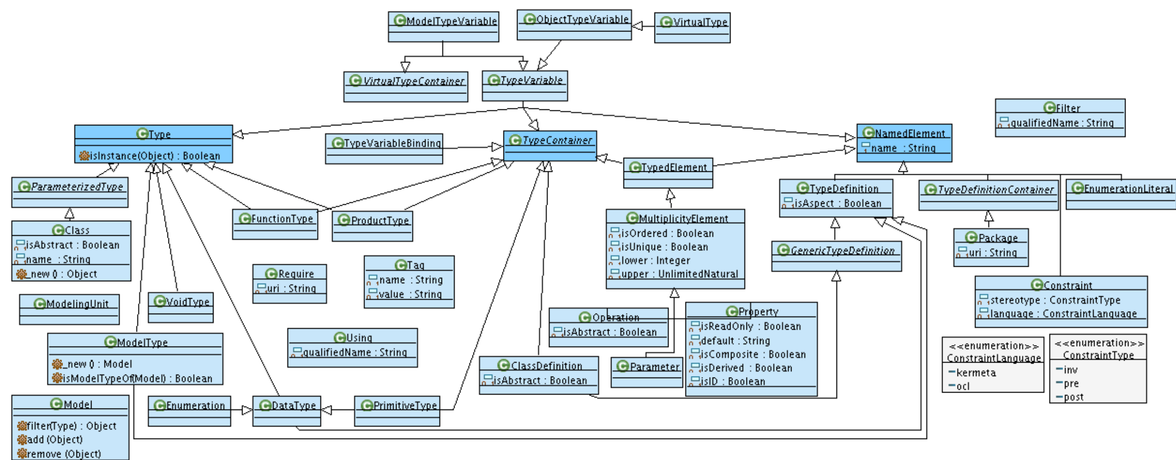


Figure 3.2. Structure package

This figure presents the main classes of the structure package. To design this package, we started from EMOF and completed it to build the Kermeta language. The choice of EMOF is motivated by two main reasons : firstly it is standardized by the OMG and secondly is is well-supported by tools such as Eclipse/EMF.

As MOF is not initially designed to be executable, several concepts have to be completed and slightly modified to build an executable language. The first and most important modification is to add the ability to define the behavior of operations. To achieve this we define an action language in the package `behavior` of Kermeta. The class hierarchy of the package behavior is presented on Figure 3.5, “Behavior package”. In practice, Kermeta expressions have been designed by adding model modification capabilities (like assignment of properties for instance) to OCL expressions.

This represents the static part of the metamodel.

As a reminder, the structure of Kermeta is derived from EMOF from the OMG. During the build process, we merge and connect it with the behavior part.

So all the meaning of those metaclasses are very close to those described in the OMG standard specification.

3.2.1. NamedElement view

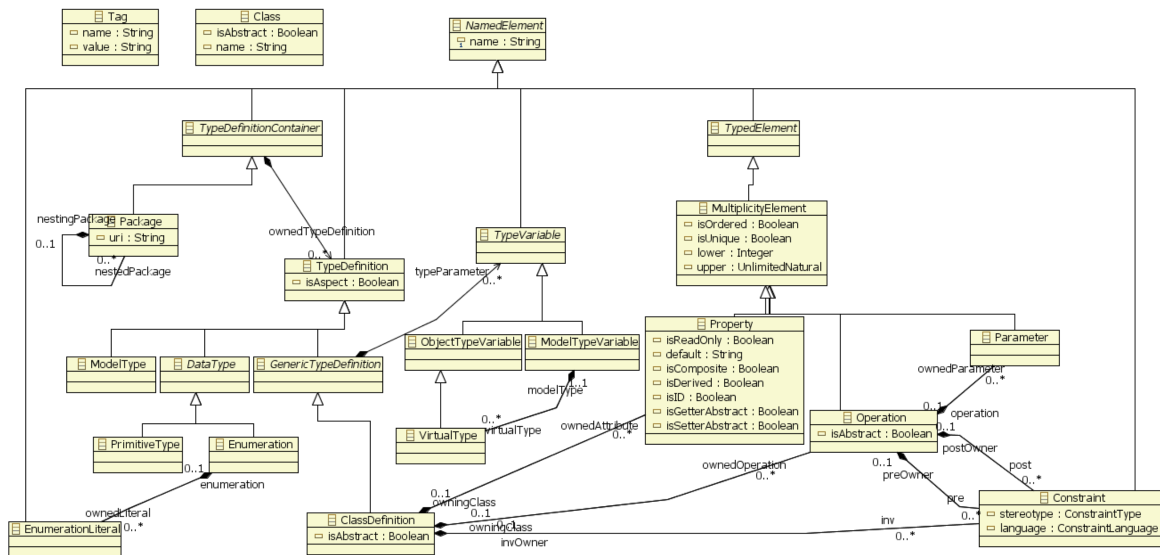


Figure 3.3. NamedElement class diagram

This figure presents the element in the structure package that have a name. Kermeta relies on in various situation (For example in the typechecking process). The containment hierarchy is also presented since it is used in the identification process.

Class is a bit special, since its name is a derived property.

3.2.2. Type system view

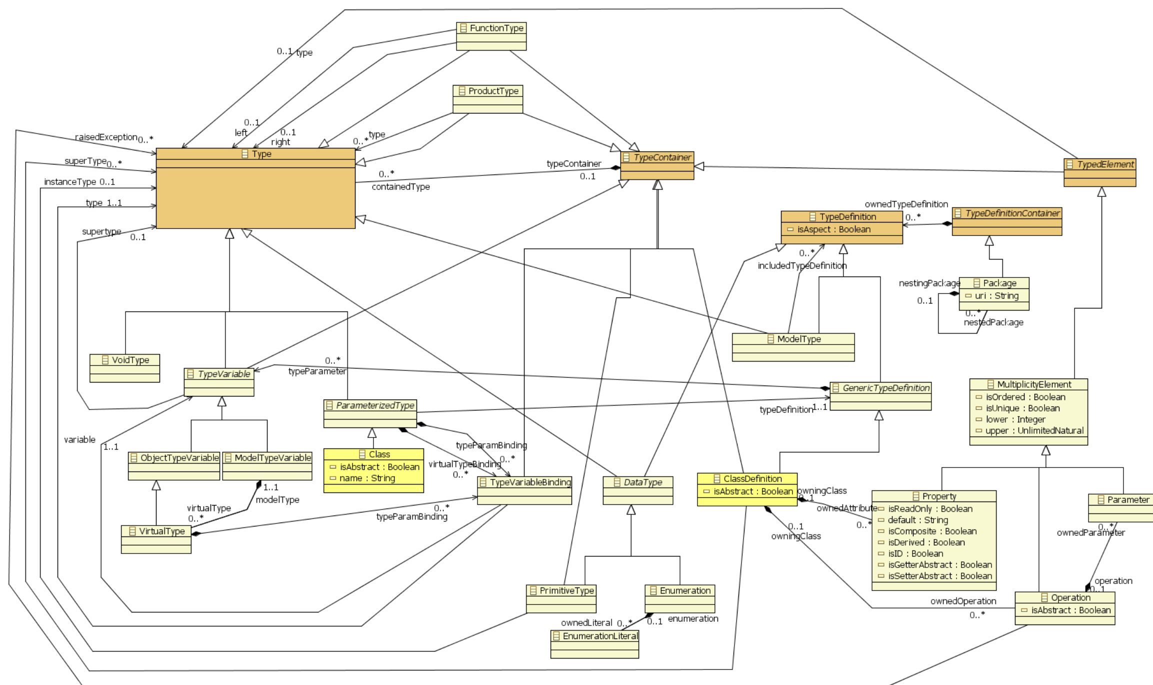


Figure 3.4. Kermeta type system class diagram (the big picture)

This figure presents the global view of all the metaclasses involved in Kermeta type system.

Basically, you can notice the split between Type and TypeDefinion needed in order to handle generics.

The containment is also represented here.

Please note that there is both `Type` and `TypeDefinition`. This is needed because of the support of the generics. For example in

```
var mycoll : Collection<String>
```

mycoll is a VariableDecl which point to a `Class` whose typeDefinition is the ClassDefinition `Collection` and has a TypeVariableBinding that lead to String.

TODO add an object diagram or a figure to illustrate.

TODO : if time : provide a set of small diagrams that focus of some elements of the type system.

3.3. Behavior package

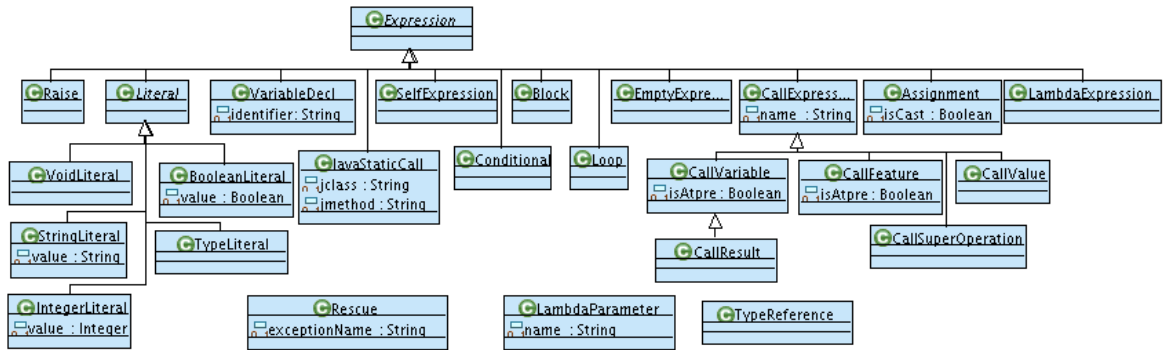


Figure 3.5. Behavior package

3.3.1. Control Structures

Kermeta provides basic control structures : block, conditional branch, loop, and exception handling. Here there an excerpt of the Meta-model describing control structures. Each basic control structures derives from the Expression concept.

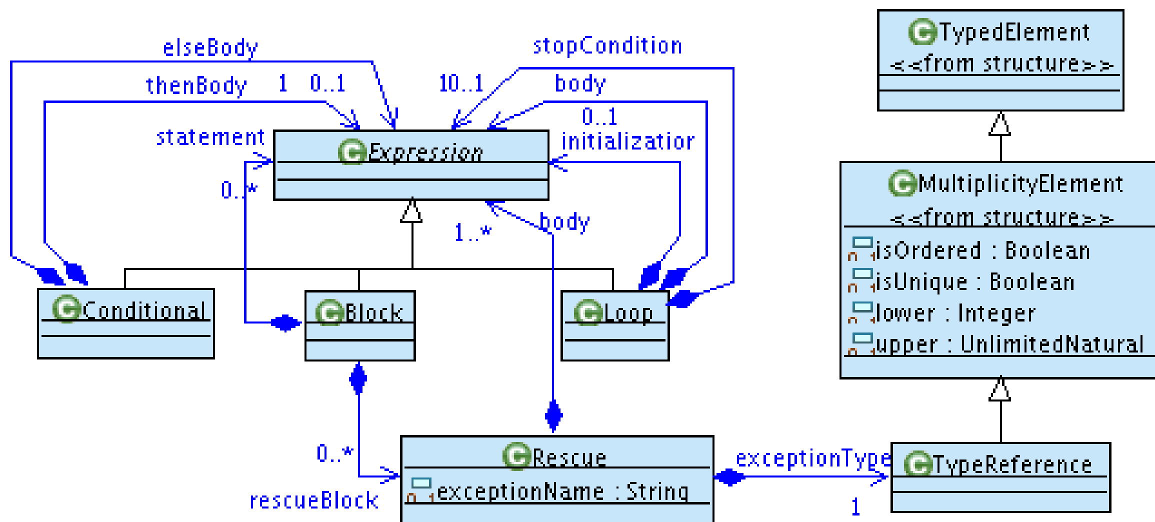


Figure 3.6. Control structure

3.3.2. Variables

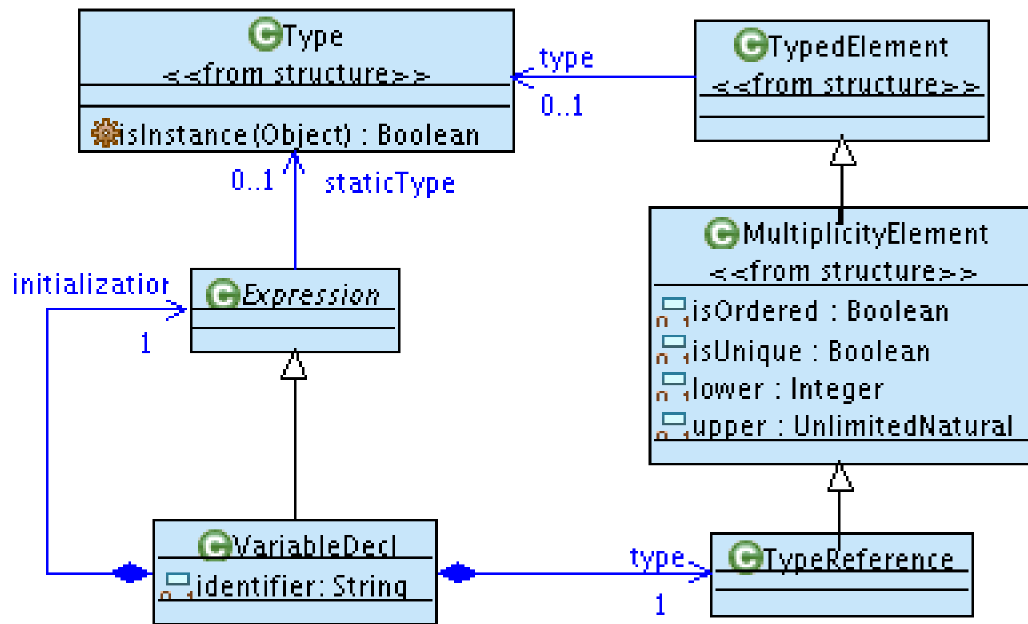


Figure 3.7. Use of variables

3.3.3. Call Expressions

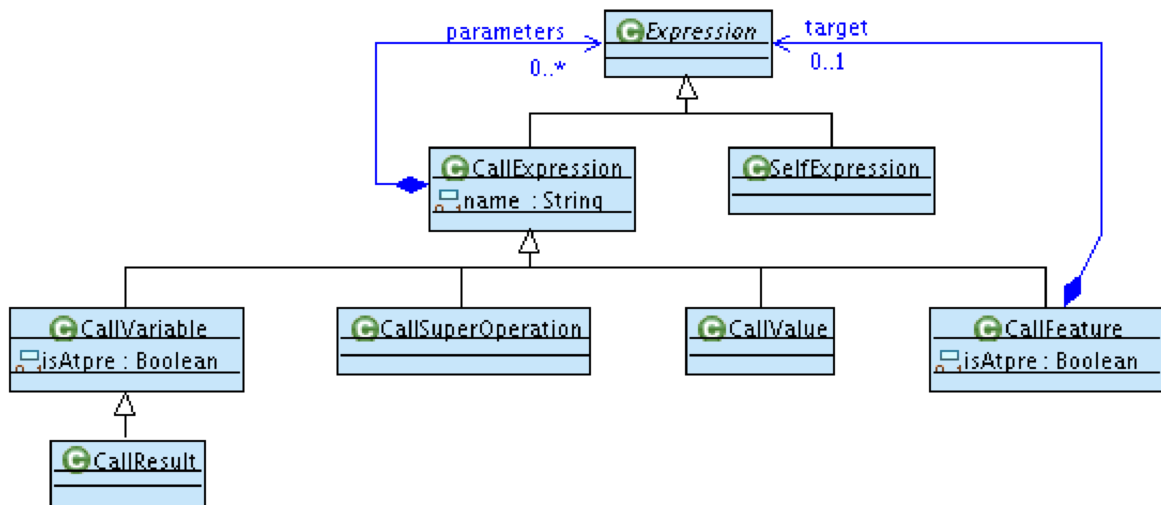


Figure 3.8. use of exceptions

3.3.3.1. CallSuperOperation

In the following example, the type of *super(element)* is *CallSuperOperation*:

```
class ParentClass {
   operation op(element : Integer) : Integer is do
      result := element + 1
   end

}

class ChildClass {
   method op(element : Integer) : Integer is do
      result := super(element)
   end
}
```

3.3.3.2. CallVariable

The type of *callvar*, below, is CallVariable:

```

var myvar : Integer
var callvar : Integer init 4
//
myvar := callvar
```

A special case, when calling a lambda expression : the type of *lf* in the assignment of res, is CallVariable.

```

var lf : <Integer->Integer>
var res : Integerlf := function  { i : Integer | i.plus(1) }
// The type of lf, below, is CallVariable
res := lf(4)

```

3.3.3.3. CallResult

The type of *result* is *CallResult*

```
operation op() : Integer is do
   result := 61
end
```

3.3.3.4. CallFeature and SelfExpression

- The type of *self* is a SelfExpression!
- The type of *attr* in the body of the operation myoperation is *CallFeature* (a callfeature on *self*), and so is the type of *myoperation(4)* (a callfeature on **a**).

```
class A {
   attribute attr : Integer
   operation myoperation(param : Integer) : Integer is do
```

```
        result := self.attr + param
    end
}
class B {
    operation anotheroperation() : Integer is do
        var a : A
        result := a.myoperation(4)
    end
}
```

3.3.4. Assignment

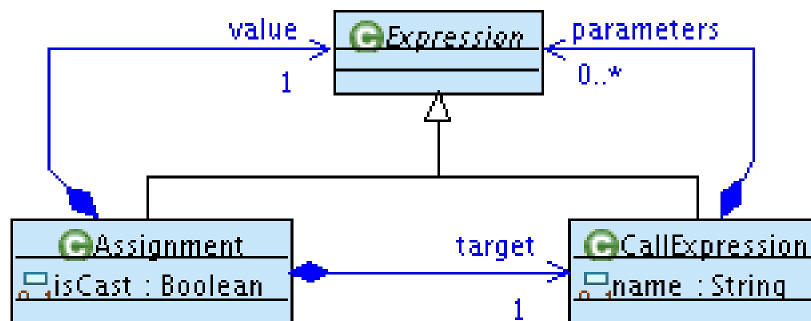


Figure 3.9. Kermeta assignment expression

In the following example, *thetarget* is of type *CallExpression* and *thevalue* is of type *Expression*.

```
var num : Numeric
var thetarget : Integer
var thevalue : Integer
// assignment : thetarget->target, thevalue->value
thetarget := thevalue
// casting : a is casted into the type of num which is Numeric.
num ?= a
```

3.3.5. Literals

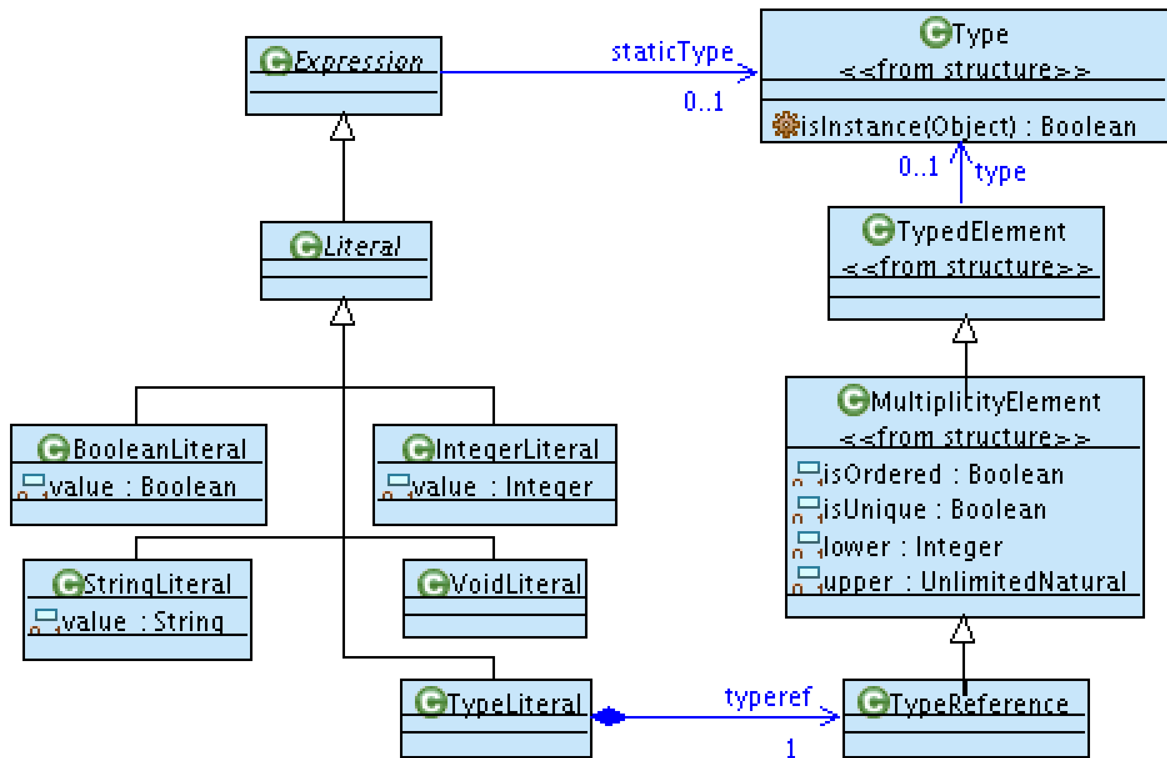


Figure 3.10. Kermeta Literal Expression

```
var i : Integer
i := 5    // 5 is a IntegerLiteral
var s : String
s := "I am a string" // "I am a string" is a StringLiteral
```

3.3.6. Lambda Expression

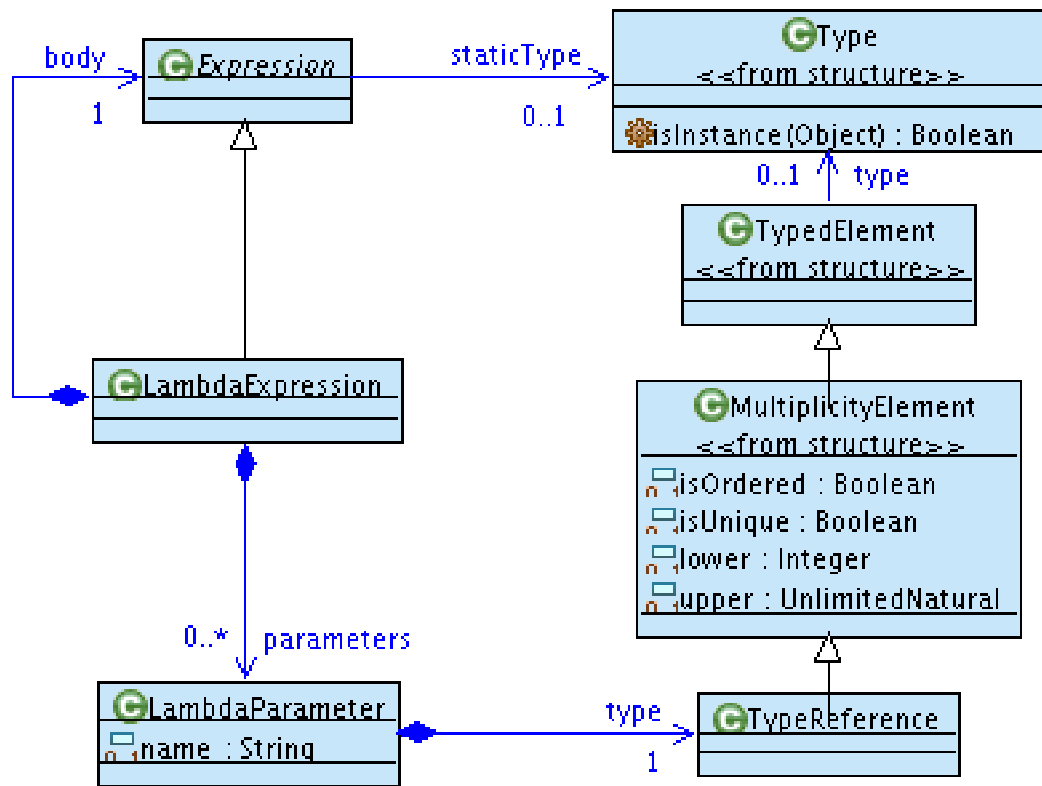


Figure 3.11. Kermeta lambda expressions

3.4. Viewing Kermeta metamodel

In Kermeta, you can view its metamodel by several ways.

First, the ecore file kermeta_java.ecore is available in the lib folder of Kermeta main plugin.It is used when saving a Kermeta program in XMI ("Compile to XMI" function on *.kmt files). You can then load Kermeta program as a model, typically to transform it.

> **Warning**
>
> You should not try to execute operations on Kermeta models you've just dynamically created unless you froze it. This is a feature which has not been completly tested.

Another typical way to access to a Kermeta model and Kermeta metamodel is to use the reflection. All objects inherits from Object that defines the `getMetaClass` operation. This use is used in one of the samples of Section 2.18, “Dynamic evaluation of Kermeta expressions” (when it selects an operation to be executed).

At last, the parse method on dynamic expression presented in section Dynamic evaluation of Kermeta expressions can give you some way to access a Kermeta model as it parses a Kermeta text and provides the corresponding model.

CHAPTER 4

Kermeta framework

Kermeta is bundled with a framework that provides the base functionalities for metamodel engineering.

> **Tip**
>
> When you write `require kermeta` at the head of your file, you are importing the file framework.km located in the Kermeta plugin.

Currently, it provides the following packages :

Packages in Kermeta framework

`kermeta::interpreter`	This package defines some classes related to Kermeta interpreter and also to Kermeta surface syntax. Some uses of this package are illustrated in Section 2.18, "Dynamic evaluation of Kermeta expressions".
`kermeta::persistence`	This package defines the notion required to serialize and deserialize models. This is illustrated in Section 2.12, "Loading and saving models" and in the EMF tutorial document (http://www.kermeta.org/documents/emfTutorial/)
`kermeta::kunit`	This is a basic support for unitary tests (like Junit but for Kermeta).
`kermeta::language::structure`	It contains the classes of Kermeta structure. Note that it slightly differs from the kermeta_java.ecore due to an inheritance to the classes in Kermeta::reflexion. However, it proposes the very same functionalities.
`kermeta::language::behavior`	It contains the classes for the behavior of Kermeta.
`kermeta::exceptions`	It defines various exceptions that you can use in Kermeta. Inheriting from kermeta::exceptions::Exception allows to give more information to the end user because it will also provide a stack trace (which would not be available otherwise).
`kermeta::io`	This package provides basic support for input/output with Kermeta. It is voluntary minimal because that not the main role of Kermeta to provide such primitives. In the future, this package may even disappear and be replaced by the ability to directly call Java libraries.
`kermeta::reflection`	This package contains all the abstract classes needed for the reflexivity of Kermeta. The concrete implementation are in kermeta::standard or kermeta::structure.

`kermeta::utils` This package defines various tool classes that cannot be part of the core of the language, but are important enough to be part of the framework.

`kermeta::standard` This package defines all the basic objects needed for a typical Kermeta application. For example, it defines data types, collections, etc.

> **Note**
>
> For more details about the content of these packages, please look at the generated documentation available on line: http://www.kermeta.org/docs/KermetaFramework/framework.km.html

CHAPTER A

Language keywords

The following keywords are used by Kermeta textual syntax.

> **Note**
>
> If you want to name some class or property using those names, you'll need to escape them using the ~ (tilda).Ex: `class ~class {}`
>
> (See Section 2.2, “Escaping reserved keywords”)

Keyword	Usage
@pre	Represents self before the call to this operation in the scope of a post condition. See Section 2.19.1.1, “pre – post conditions syntax”
abstract	Modifier for class or operation
alias	Definition of a primitive type
and	Boolean expression
attribute	Definition of an attribute
bag	See Section 2.14, “Collections”
class	Definition of a class
do	Beginning of a block
else	Else part of a conditional instruction
end	End of a block
enumeration	Definition of an enumeration
extern	Call of a java static operation
false	Boolean literal
from	Loop instruction

Keyword	Usage
function	Declares a local function
getter	Declaration of a property getter
if	Conditional instruction
inherits	Declartion of the super classes of the class
init	Initialization of a variable
inv	Declaration of an invariant. See Section 2.19.1.2, “Invariant constraint syntax”
is	part of the declaration of an operation or method. See Section 2.8.2, “Defining operations”
loop	Loop instruction
method	Redefinition of an operation. See Section 2.8.2, “Defining operations”
modeltype	Definition of a modeltype.
not	Boolean expression
operation	Declaration of an operation. See Section 2.8.2, “Defining operations”
or	Boolean expression
oset	See Section 2.14, “Collections”
package	Declaration of package
post	Declaration of a postcondition. See Section 2.19.1.1, “pre – post conditions syntax”
pre	Declaration of a precondition. See Section 2.19.1.1, “pre – post conditions syntax”
property	Declaration of a derived property
raise	Throw an exception
raises	Declares the exception that an operation can throw
readonly	Modifier for properties
reference	Declaration of a reference

Keyword	Usage
require	Declaration of a required file
rescue	Catch an excetion
result	Special variable used for the return value of an operation. See Section 2.8.2, “Defining operations”
setter	Declaration of property setter
self	Special varaible representing this instances
set	See Section 2.14, “Collections”
seq	See Section 2.14, “Collections”
super	Call to the super operation
then	Then part of a conditional instruction
true	Boolean literal
until	Loop condition
using	Shortcut used to avoid to write the full qualified name
value	Special variable used in getter and setter to represent the actual value of the property
var	Declaration of a variable
void	Void literal

CHAPTER B

Known Kermeta tags

This annex collect the use of known tags. They are generally used as an extension on Kermeta in these typically uses :

- supporting Ecore compatibility,
- preparing a future language improvement,
- supporting a specific need that isn't part of the language core.

> **Note**
>
> TODO : finish this by looking into kermeta2ecore transformation

Kown tags :

`aspect`

Possible values. "true", case insensitive, any other value is considered as "false".

Context. Class

Use description. Old way to indicate that a Class must be reopened as an aspect of another Class and then merging its features. We now prefer to use the dedicated keyword.

Use sample.

```
@aspect "true"
class MyClas {
// ...
}
```

`ecoreUri`

Possible values. An nsURI.

Context. Package

Use description. Indicates the nsUri for the given package when it is transformed in Ecore.

Use sample.

```
@ecoreUri "http://myPackage/1.0/"
package myPackage;
```

`mainClass`

Possible values. Qualified name of a class in the Kermeta program.

Context. ModelingUnit

Use description. Indicates a preferred class from which the run configuration should try to start the Kermeta program. This helps to create a default run configuration. Must be used with the mainOperation tag.

Use sample.

```
@mainClass "myPackage::MyMainClass"
@mainOperation "main"

package myPackage;

require kermeta
class MyMainClass
{
	operation main() : Void is do
		// TODO: implement 'main' operation
	end
}
```

`mainOperation`

Possible values. name of an operation in the main class.

Context. ModelingUnit

Use description. Indicates a preferred operation from which the run configuration should try to start the Kermeta program. This helps to create a default run configuration. Must be used with the mainClass tag. The operation must : either have no parameter, or have string parameters.

Use sample.

```
@mainClass "myPackage::MyMainClass"
@mainOperation "main"

package myPackage;

require kermeta
class MyMainClass
{
	operation main() : Void is do
		// TODO: implement 'main' operation
	end
}
```

CHAPTER C

Kermeta / Ecore mapping

Kermeta natively integrates Ecore defintions, however it uses an internal mapping in order to import ecore model as Kermeta models. In the other way round a Kermeta model can be translated into a pure ecore model. (for example in the compiler)

This annex indicates how Kermeta models are translated into Ecore models and vice versa.

> **Note**
>
> TODO : finish this by looking into kermeta2ecore transformation and the compiler specifications.

Kermeta concept	Ecore concept	Notes
Class	EClass	
Constraint (invariant)	EAnnotation(source="kermeta.inv") + DetailEntry	
Constraint (precondition|postcondition)	EAnnotation(source="kermeta.pre"|"kermeta.post") + DetailEntry	
Constraint (postcondition)	EAnnotation(source="") + DetailEntry	
Operation	EOperation	
Tag (general case)	EAnnotation(source="kermeta") + DetailEntry	Most Tags are translated into an EAnnotation with source="kermeta" and one DetailEntry with key=*NameOfTheTag* and value=*ValueOfTheTag*. If this is a /** */ comment, the name of the tag is "documentation".
Tag (applied on Constraint)	EAnnotation(source="kermeta.inv.doc") + De-	EAnnotation.references points to the EAnnotation representing the Constraints. DetailEntry with key=*NameOfTheTag* and value=*ValueOfTheTag*. If this

Kermeta concept	Ecore concept	Notes
	tailEntry	is a /** */ comment, the name of the tag is "documentation".