Kermeta Language Overview

The Triskell Metamodeling Language

Franck Chauvel
Zoe DreyEngineer eclipse (black belt)
Franck Fleurey

Abstract

This document gives an overview of the KerMeta language developped by the Triskell team. It details the specifics of the language. The aim of this docuent is to present an general survey of KerMeta concepts.

Kermeta Language Overview
The Triskell Metamodeling Language
Franck Chauvel
Zoe DreyEngineer eclipse (black belt)
Franck Fleurey
Published Build date: 31-January-2007

Chapter 1. What is Kermeta ? 1

Chapter 2. Kermeta basics 3

2.1. Architecture 3

2.2. Object-Oriented features 5

2.2.1. Operation redefinition 5
2.2.2. Operation specialization 6
2.2.3. Operation overloading 7
2.2.4. Conflicts related to multiple inheritance 7

2.3. Kermeta type system 8

2.3.1. Generic classes 8
2.3.2. Generic operations 9

2.4. Functions in kermeta 9

Chapter 3. Kermeta metamodel 10

3.1. Structure package 10

3.1.1. Packages, subpackages 10
3.1.2. Class 11
3.1.2.1. A basic example 11
3.1.2.2. Abstract class 11
3.1.2.3. Parametric classes and type variable binding 11
3.1.3. Properties 11
3.1.3.1. Attribute and reference 12
3.1.3.2. How to access and control the properties in Kermeta 13
3.1.4. Property 15
3.1.5. Datatypes : primitive types and enumeration 15

3.2. Behavior package 15

3.3. Basic Control Structures 16

3.3.1. Basic block 17
3.3.2. Conditional Statement 17
3.3.3. Loop 17
3.3.4. Exception Handling 17
3.3.4.1. Raising exception 17
3.3.4.2. Catching Exceptions 18

3.4. Using Variables 18

3.5. Call Expressions 19

3.5.1. CallSuperOperation 19

3.5.2. CallVariable 20
3.5.3. CallResult 20
3.5.4. CallFeature and SelfExpression 20

3.6. Assignment 21

3.7. Literals 21

3.8. Lambda Expression 21

Chapter 4. Examples 24

4.1. Hello world 24

4.2. Simple State Machines 24

List of Figures

1.1. Kermeta positionning 1
2.1. Package kermeta::language::structure 4
2.2. Package kermeta::language::behavior 5
3.1. Structure package 10
3.2. Type binding 10
3.3. Attributes and references 12
3.4. Behavior package 16
3.5. Kermeta Control Structures 16
3.6. Use of variables 18
3.7. use of exceptions 19
3.8. Kermeta assignment expression 21
3.9. Kermeta Litteral Expression 21
3.10. Kermeta lambda expressions 22
4.1. sample state machine 25
4.2. Simple State Machine metamodel 25

What is Kermeta ?

Kermeta is a metamodeling language which allows describing both the structure and the behavior of models. It has been designed to be fully compliant with the OMG metamodeling language EMOF (part of the MOF 2.0 specification) and 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.

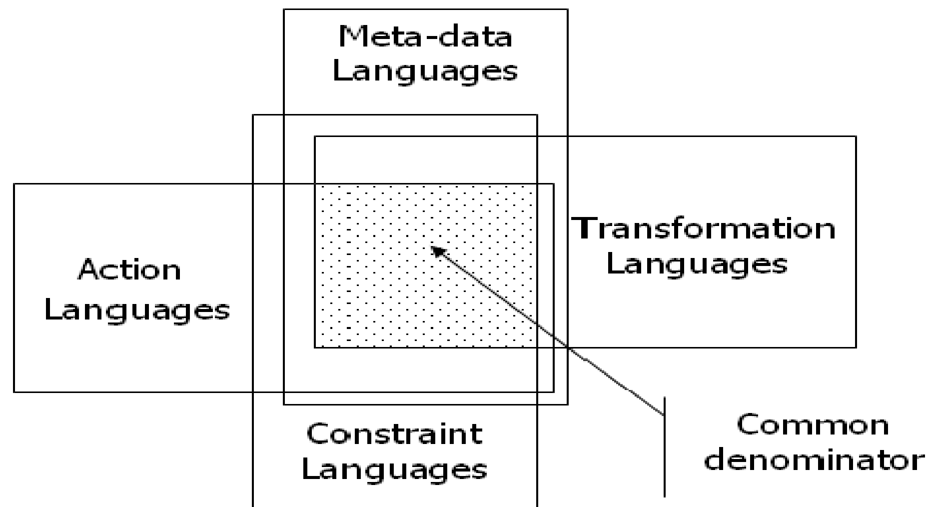

Figure 1.1. Kermeta positionning

In a nutshell, Kermeta is :

-
    - MOF compliant (EMOF compliant to be precise)
    - Imperative
    - Object-Oriented
    - Staticaly Typed

In addition to these characteristics, it includes some typically model-oriented concepts like associations, mul-

tiplicities or object containment management.

This document is presents the main features of the kermeta language. Section 2 presents the general properies of the language, section 3 details the concrete and abstarct syntax of the language and finally section 4 provides some simple programs in kermeta.

Warning

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 bugtracker in the forge:http://gforge.inria.fr/tracker/?group_id=32 Last check: v0.0.16

CHAPTER 2

Kermeta basics

The goal of the KerMeta langage is to provide an action langage for MOF models. The idea is to start from MOF, which provides the structure of the langage, and to add an action model. Using the MOF to define the structure of the KerMeta langage has an important impact on the langage. 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 sementic corresponding to MOF concepts has to be defined in KerMeta. For instance MOF does not provide a sementic for behavior inheritance (concepts like method redefinition, abstract method, ... does not have any sense in the MOF).

This section presents the main characteristics of the kermeta language:

-
  - Object-Oriented Imperative language
  - Type system
  - Functions types

2.1. Architecture

Kermeta has been designed to be fully compatible with the OMG standard meta-data lanquage EMOF. The metamodel of kermeta is divided into two packages : *structure* which corresponds to EMOF and *behavior* which corresponds to the actions. This section gives an overview of these two packages and their relationships.

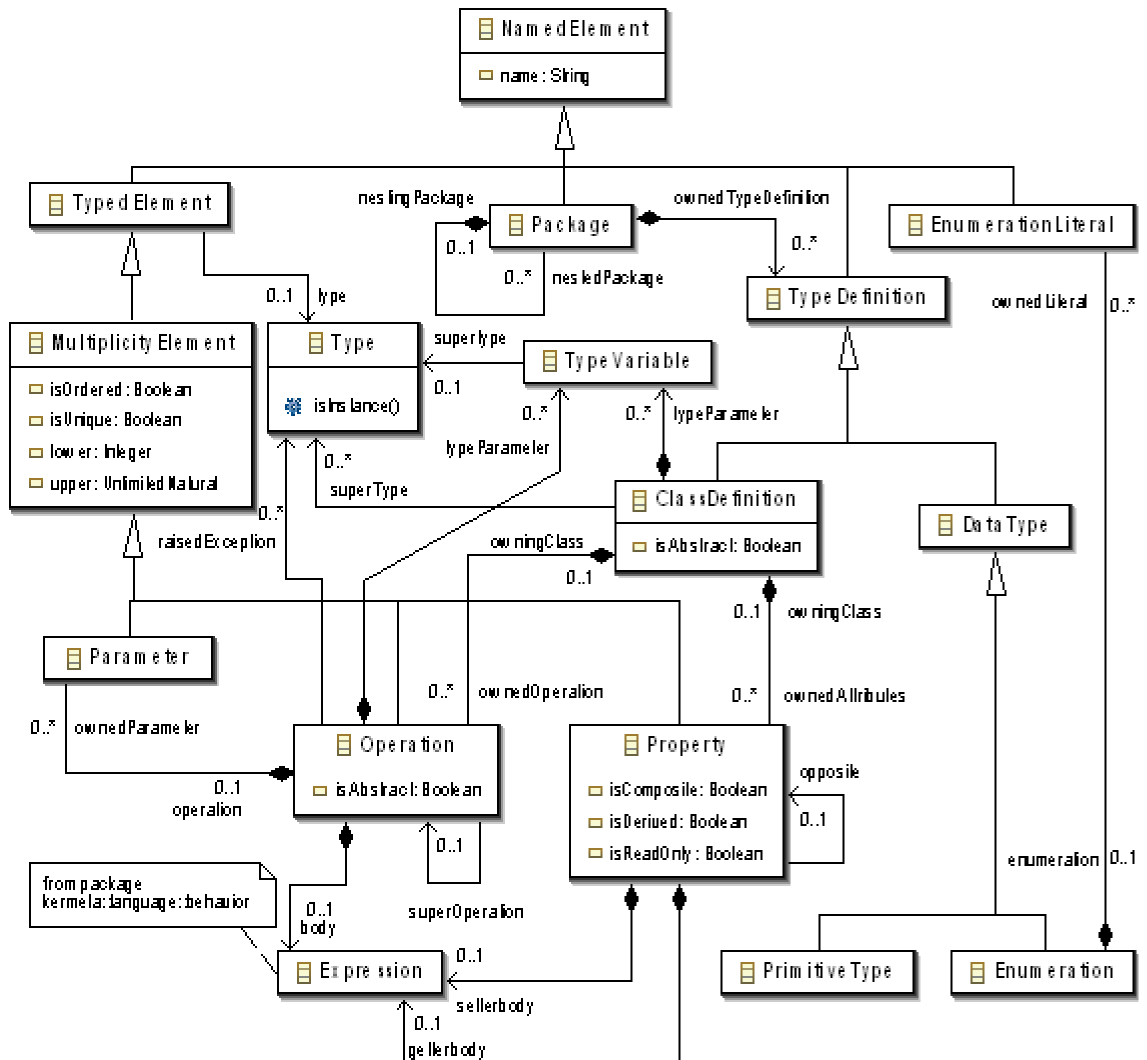

Figure 2.1. Package kermeta::language::structure

Figure 3 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-suported by tools such as Eclipse/EMF.

As MOF is not initialy designed to be executable, several concepts has to be completed 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 ???. In practice, Kermeta expressions have been designed by adding model modification capabilities (like assignement of properties for instance) to OCL expressions.

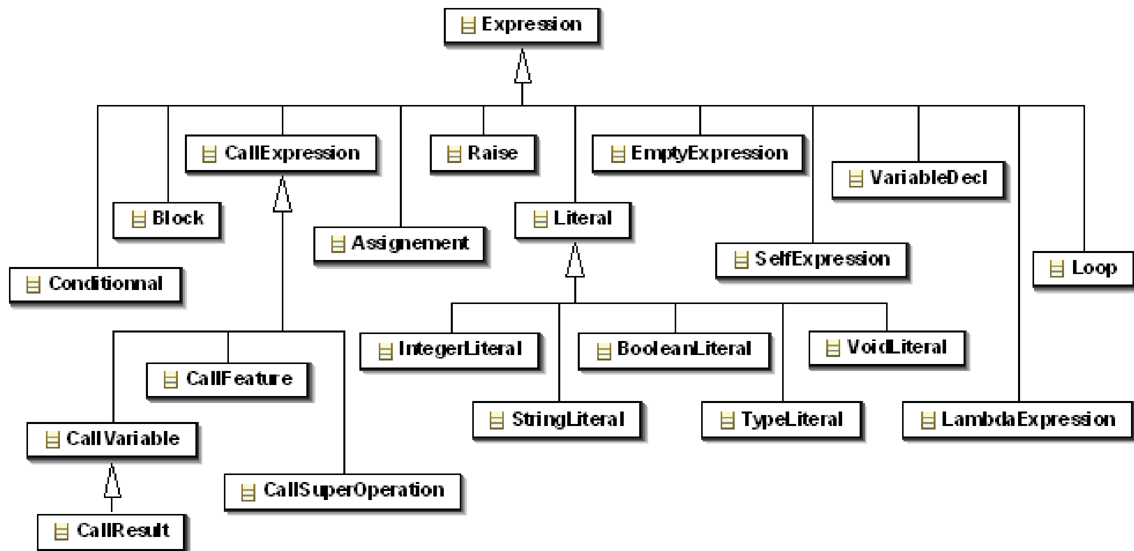

Figure 2.2. Package kermeta::language::behavior

The link between structure and behavior is made throught the property « body » of class Operation which allows defining the behavior of an operation using a Kermeta expression. Yet, in order to fulfil the requirements presented before a few more extensions has to be performed on EMOF. These are detailed in the following sections.

2.2. Object-Oriented features

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.

2.2.1. 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.

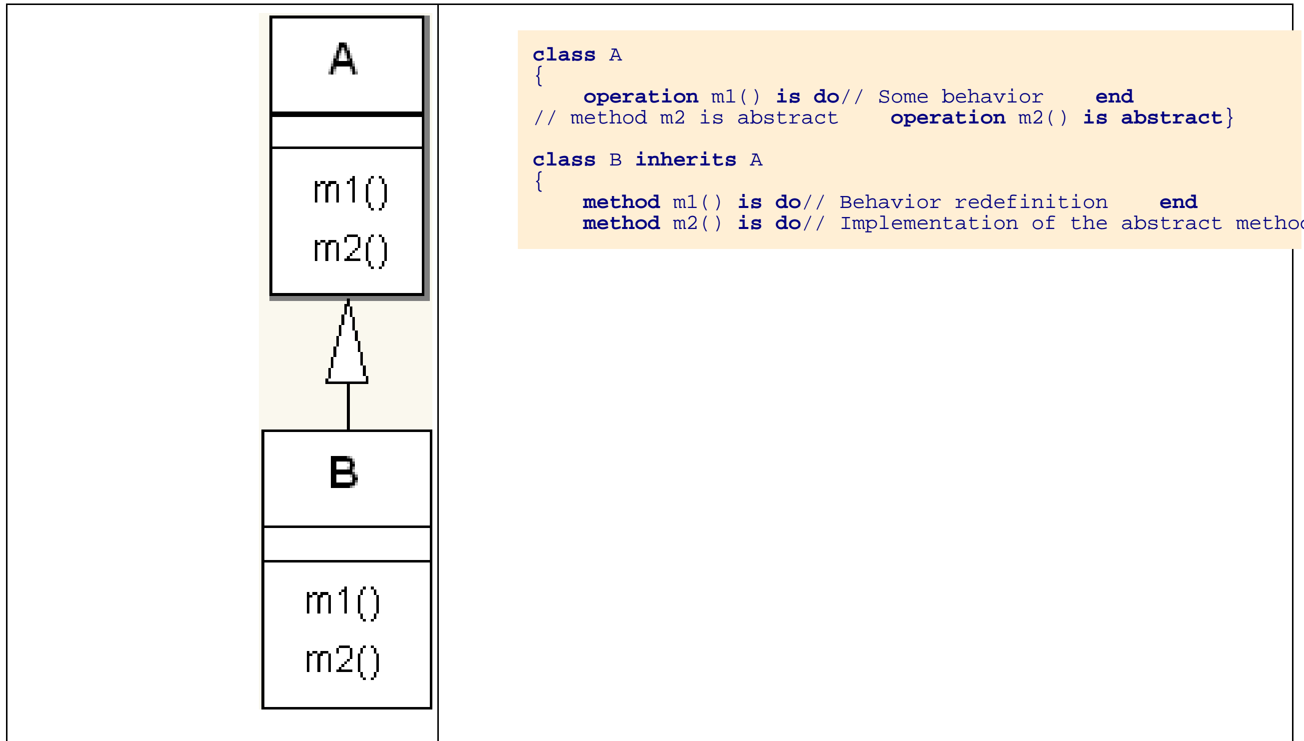

Table 2.1. Operation redefinition in Kermeta

2.2.2. 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 therory 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 precisly 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.

2.2.3. Operation overloading

Overloading 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 convenient way for writing operation whose behaviors differs 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 have chosen to exclude overloading from Kermeta.

2.2.4. 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 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 solution to resolve these conflicts. The first one is to have an implicit resolution mechanism which arbitrarily 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
{
// Explicit selection of the implementation// to inherit    metho
```

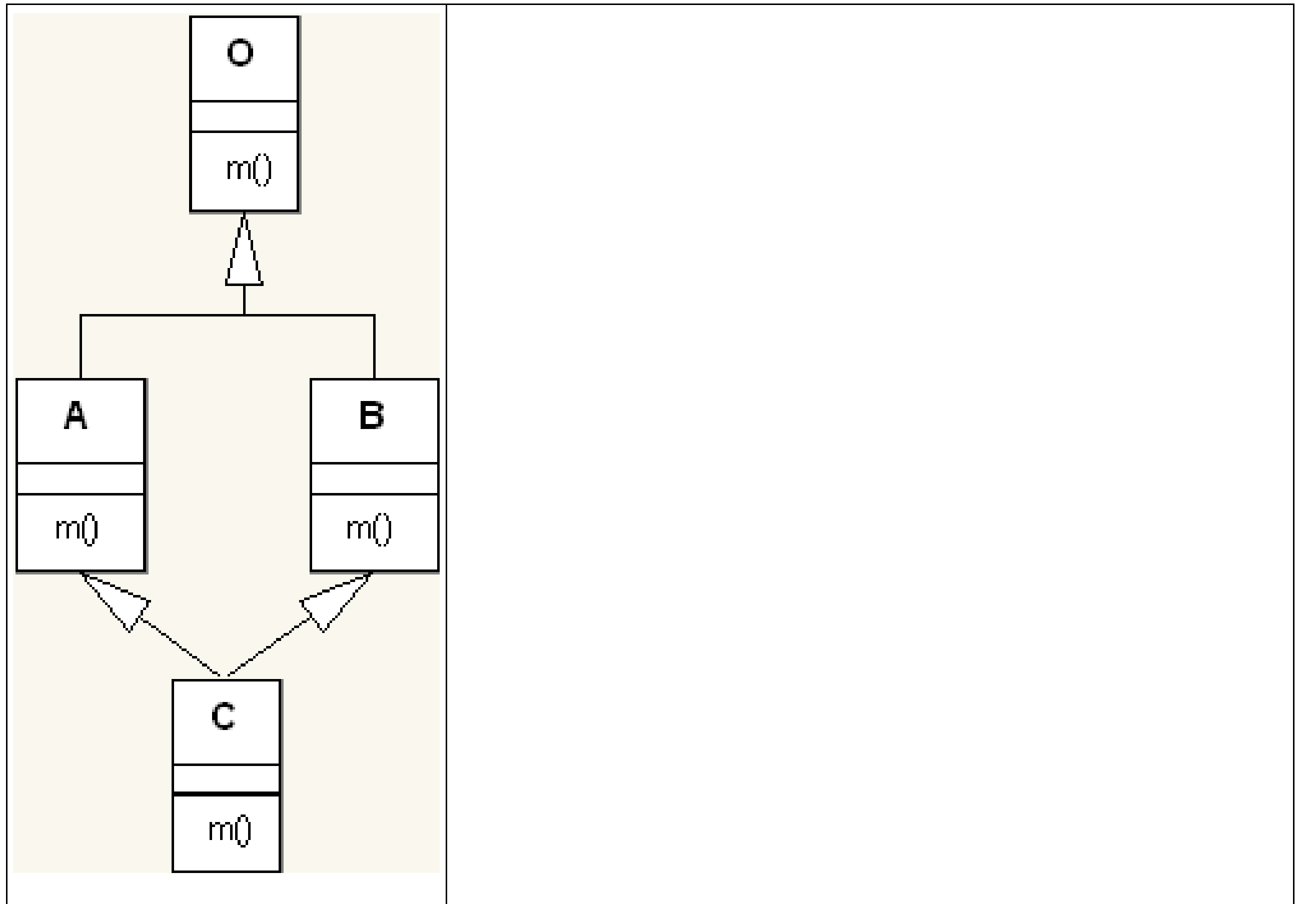

Table 2.2. Explicit selection of super operation in Kermeta

2.3. Kermeta type system

One of the core charateristics of kermeta is to be statically typed. In order to allow static typing of OCL-like expression, 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., to be presented at the Models05 conference).

As a result to these modification 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.

2.3.1. 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 be constraind by a type, in that case, the type variable can only be substituted by a sub-type of this type. The fol-

lowing code demonstrate how to create generic classes.

```
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}

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}
```

2.3.2. Generic operations

Kermeta operations can contain type parameters. Like for classes these type parameters can be constrained by a super type. However, unlike for classes for which the bindigs to these type parameters is explicit, for operations the actual type to bound to the variable is statically infered 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 en
```

2.4. Functions in kermeta

In order to implement and statically type check OCL-like iterators, kermeta includes some limited functional features. See section 3.8 for detailed informations.

CHAPTER 3

Kermeta metamodel

3.1. Structure package

Figure 3.1. Structure package

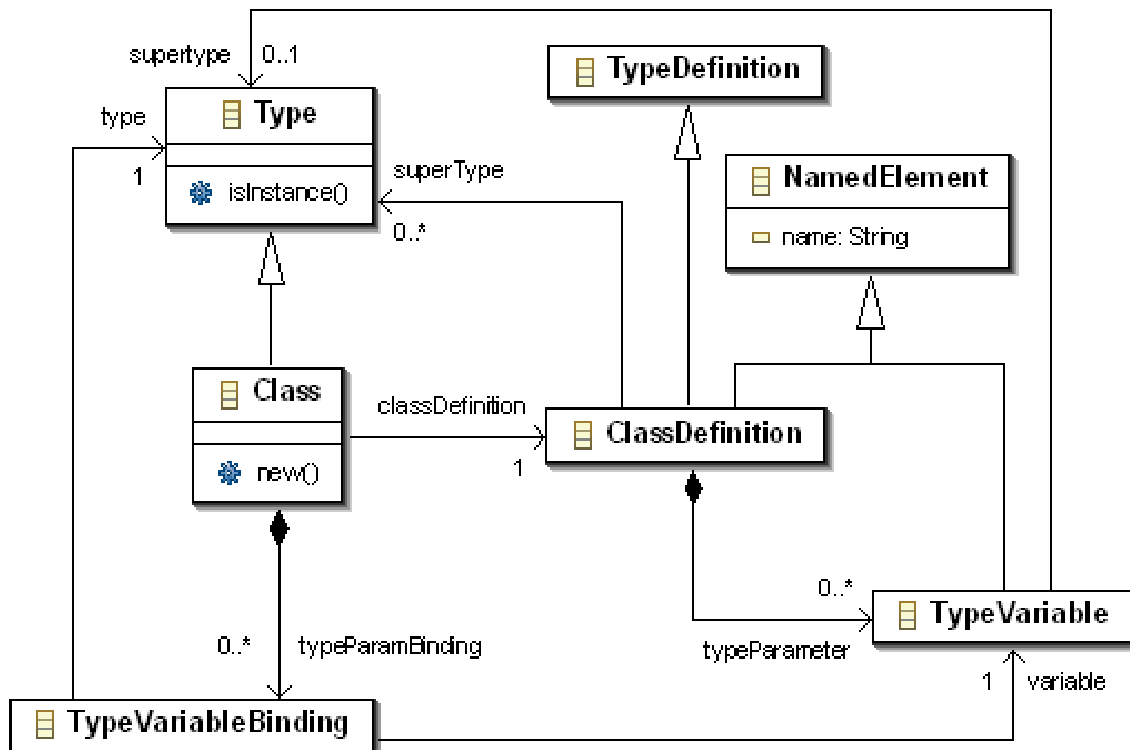

Figure 3.2. Type binding

3.1.1. Packages, subpackages

```
// the root package is unique, and specified by the ";"
package rootPackage;

package nestedPackage
{
    package nestedInNestedPackage {}
}
```

3.1.2. Class

3.1.2.1. A basic example

```
// This is the class definition

class A
{
 // Properties
 attribute b : B#a // A is a composite. "b" is its component
 reference c : C#a // A and C are linked by an association
 attribute i : Integer;
 property d : Integer
    getter is do
        result := i + 1
    end


 // An Operation with one Parameter.
 operation f(ownedParam : typeOfOwnedParam) : Type is do
 end

}

class B
{
    reference a : A
}
class C
{
    reference a : A
}
```

3.1.2.2. Abstract class

```
// This class is abstract (its property isAbstract equals True!)
abstract class A {}
```

3.1.2.3. Parametric classes and type variable binding

```
// 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
```

3.1.3. Properties

A property can be expressed in three ways : as a *property* (which is derivated or calculated), as an *attribute*, or as a *reference*. We introduce in this section the 2 last cases, which are relationships between two concrete entities.

3.1.3.1. Attribute and reference

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

> **Note**
>
> NOTE : a bi-composition is not valid in MOF. So, only one entity can be the component of the other.

- *reference*: a reference defines a association between two entities.
- *opposite*: the opposite [property] of a property is expressed by a sharp #. The following example means that container is the opposite property of the entity of type Contained3 and referenced by the name *contained*.

  class A {

  reference b : B#a

  }

  class B {

  reference a : A#b

  }

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

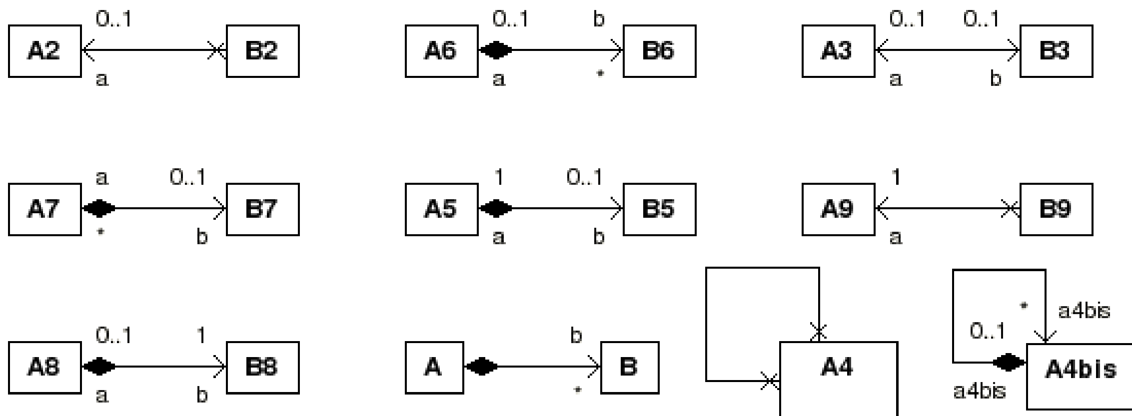

Figure 3.3. Attributes and references

```
package root;
```

```
class A {
    attribute b : B[0..*]

}
class B {}
class A2 {}
class B2 {
    reference a : A2
}
class A3 {
    reference b : B3#a
}
class B3 {
    reference a : A3#b
}
class A4 {}
class A5 {
    attribute b : B5#a
}
```

```
class B5 {
    reference a : A5[1..1]#b
}
class A6 {
    attribute b : B6[0..*]#a
}
class B6 {
    reference a : A6#b
}
class A7 {
    attribute b : B7#a
}
class B7 {
    reference a : A7[0..*]#b
}
class A8 {
    attribute b : B8[1..1]#a
}
class B8 {
    reference a : A8#b
}
class A9 {}
class B9 {
    reference a : A9[1..1]
}
class A4bis {
    reference a4bis : A4bis#a4bis
}
```

Note

For every cases where the upper bound is upper to 1, the type of the reference is OrderedSet. The reader will refer to the Language basics chapter (except the bag type) to have the available types for a [m..n](n>1) multiplicity property.

3.1.3.2. How to access and control the properties in Kermeta

let's take the example with A6 and B6 :

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

- Get the attribute of an instance:

```
var a6 : A6 init A6.new
var b6 : Set<B6>
// get the b attribute (if b6 was a 1-multiplicity element, its type // would have been “B6”)
b6 := a6.b
```

- Add/remove an element to a property with a [m..n] multiplicity

```
var aa6 : A6 init A6.new
var ab6 : B6 init B6.new
// add ab6 to the attribute “b”
aa6.b.add(ab6)
// remove ab6 : telling the index of the element to remove.
aa6.b.remove(0)
```

- Get the opposite of a property

Let's take a simple class:

```
class A {reference refb : B#refa}class B {reference refa : A}
```

We access the opposite of b following this way:

```
var vara : A init A.new
var varb : B init B.new
// add b to the attribute “b”
vara.refb := varb
// this assertion is true.
assert(varb.refa == vara)
```

It is not different with references that have a [m..n] (m>n and n>1) multiplicity:

```
class A {reference refb : B[0..*]#refa}class B {reference refa : A}
```

We access the opposite of b following this way:

```
var vara : A init A.new
var varb : B init B.new
// add b to the attribute “b”
vara.refb.add(varb)
// this assertion is true.
assert(varb.refa == vara)
```

- Get the container of a property

```
var aa6 : A6 init A6.new
var ab6 : B6
// add ab6 to the attribute “b”
aa6.b.add(ab6)
var a6cont : A6
a6cont := ab6.container()
assert(a6cont.equals(aa6))
```

3.1.4. Property

The specific property defined by the keyword property is a derived property. This means that it does not reference to a concrete entity : it is indeed calculated, through the accessor operations getter and setter.

Let's take the following class definitions :

```
// readonly property : it has no setter
class A :
    attribute period : Real
    property readonly frequency : Real
        getter is do
            result := 1/period
        end

// modifiable property :
class A :
    attribute period : Real
    property frequency : Real :
        getter is do
            result := 1/period
        end
        setter is do
            period := 1/value
        end
```

3.1.5. Datatypes : primitive types and enumeration

Here is simple datatypes examples :

```
var myVar1 : Integer init 10
var myVar2 : Integer
var myVar3 : Real init 3.14
var myVar4 : String init "a new string value"
var myVar5 : boolean
```

And here is an Enumeration simple type :

```
Enumeration Size
{
    small;
    normal;
    big
}
// An enumLiteral :
Size.small
```

3.2. Behavior package

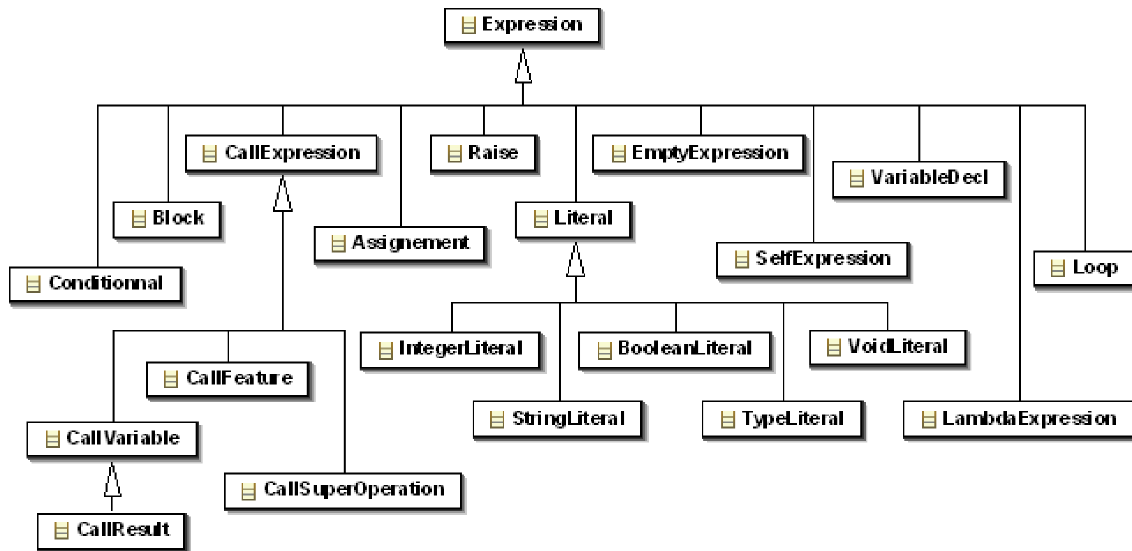

Figure 3.4. Behavior package

3.3. Basic Control Structures

Kermeta provides basic control structures : block, conditionnal branch, loop, and execption handling. Here there an exerpt of the Meta-model describing control structures. Each basic control structures derives fromo the Expression concept.

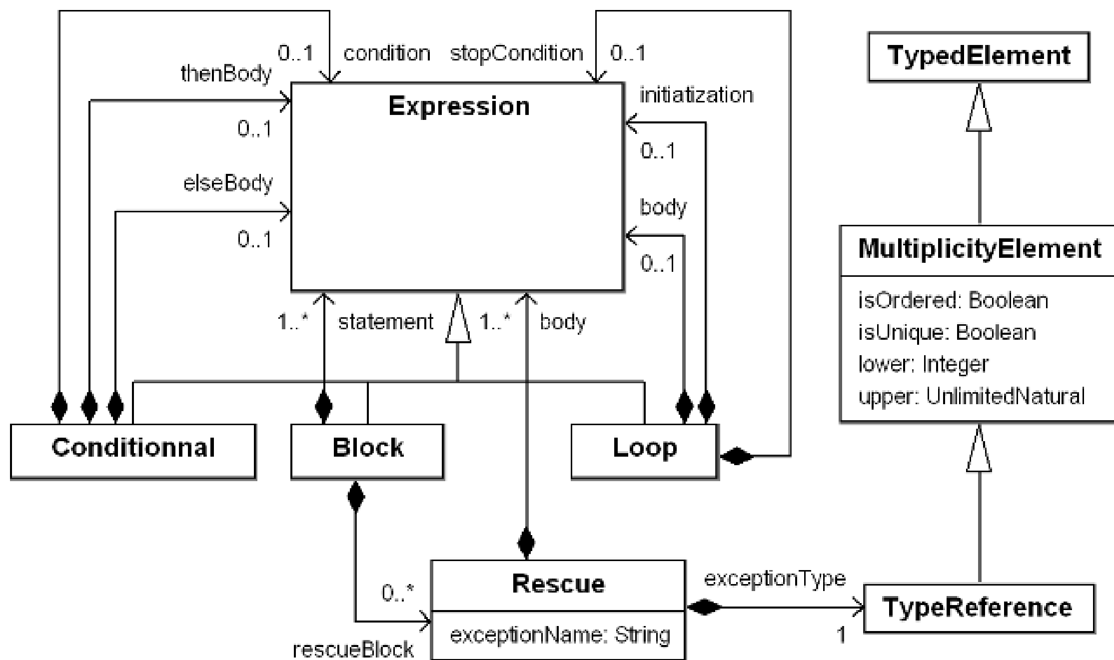

Figure 3.5. Kermeta Control Structures

In the following of this section, each basic control structure is presented in a “Eiffel-like” syntax.

3.3.1. Basic block

Basic block allow programmers to manage variable scope. As in others langage, a variable can be used in the block where it was defined.

```
do
    var v1 : Integer init 15
    do
        var v2 : Integer init 13
        do
            var res : Integer
            res := v1 + v2
        end
    end
end
```

3.3.2. Conditional Statement

Conditionnal expression allow user to switch on an boolean expression.

```
Do
    var v1 : integer init 15
    var v2 : integer init 16

    if (v1 < v2) then
        stdio.writeln("V2 est plus grand que V1")
    else
        stdio.writeln("V1 est plus grand que V2")
    end
end
```

3.3.3. Loop

Returns void

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

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

3.3.4. Exception Handling

3.3.4.1. Raising exception

```
do
    var excep : Exception

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

```
    raise excep
end
```

3.3.4.2. Catching Exceptions

Block structure can actually catch exception with the following syntax.

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

do
    var v3 : integer
    v3 := v1 + v2
rescue (myError : Exception)
    // something with myError
    // ...
end
```

3.4. Using Variables

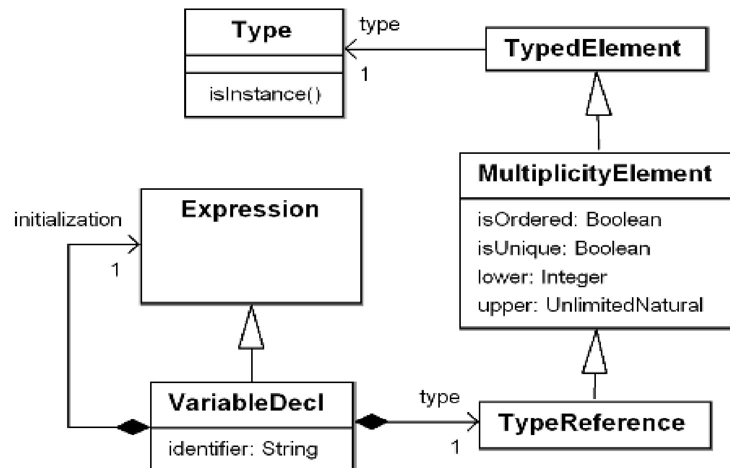

Figure 3.6. Use of variables

```
do
    // this is a VariableDecl, which initialization is 14
    var v1 : integer init 14
    var v2 : integer init 145 * v1

    var tab : integer[0..*]
end
```

In the previous example, we define 3 variables of type integer. The first is initialized with the “14” litteral value, the second is initialized with an expression using v1. For the last variable, we use a multiplicity notation to specifiy an ordered set of integer.

3.5. Call Expressions

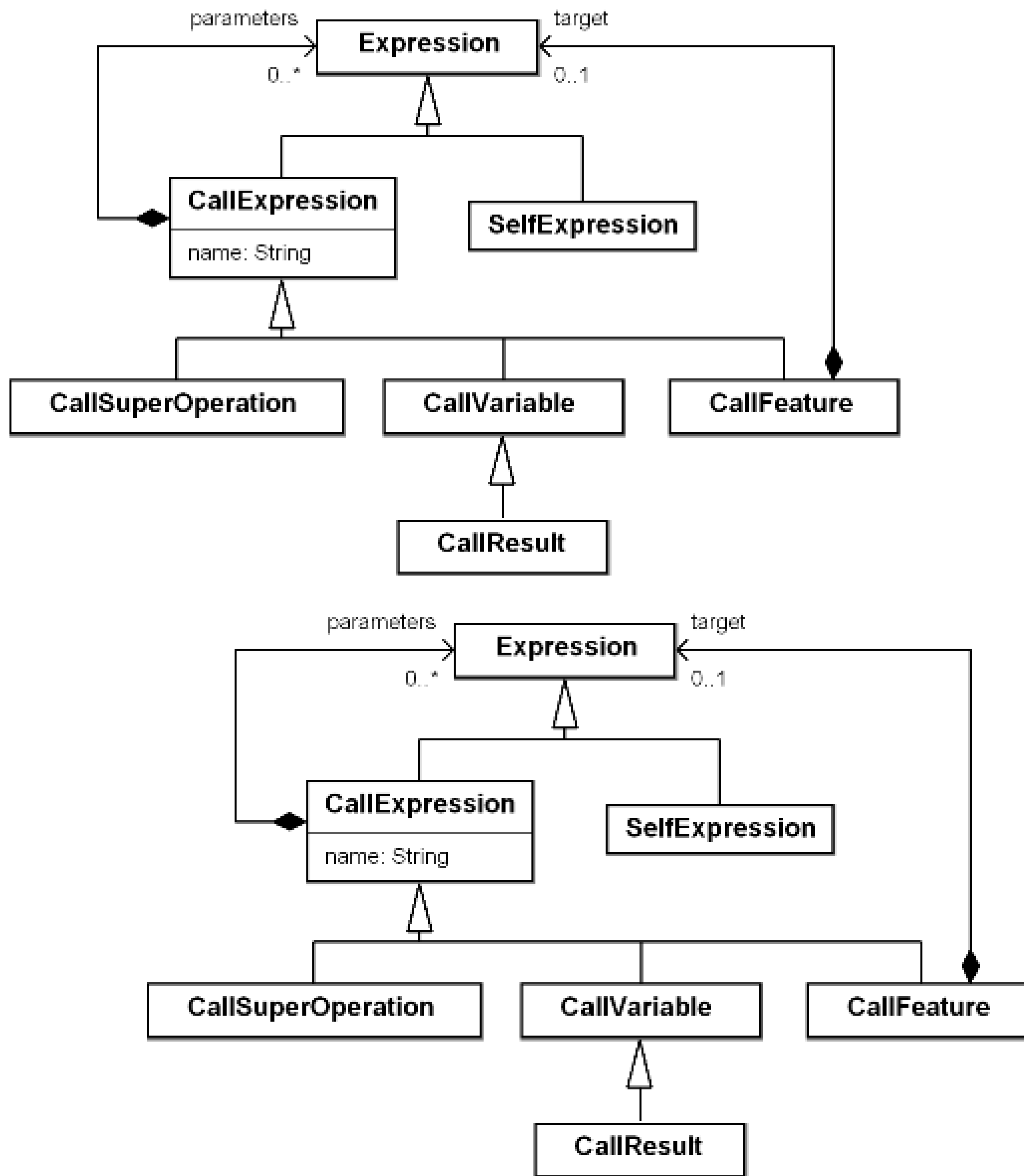

Figure 3.7. use of exceptions

3.5.1. 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}
```

3.5.2. 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 : Integer
lf := function { i : Integer | i.plus(1) }

// The type of lf, below, is CallVariableres := lf(4)
```

3.5.3. CallResult

The type of *result* is *CallResult*

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

3.5.4. 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
}
```

3.6. Assignment

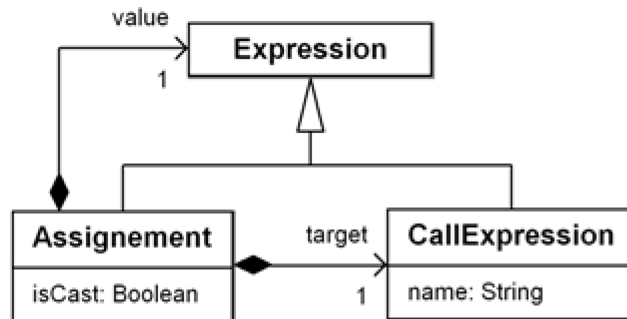

Figure 3.8. Kermeta assignment expression

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
```

3.7. Literals

Figure 3.9. Kermeta Litteral 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
```

3.8. Lambda Expression

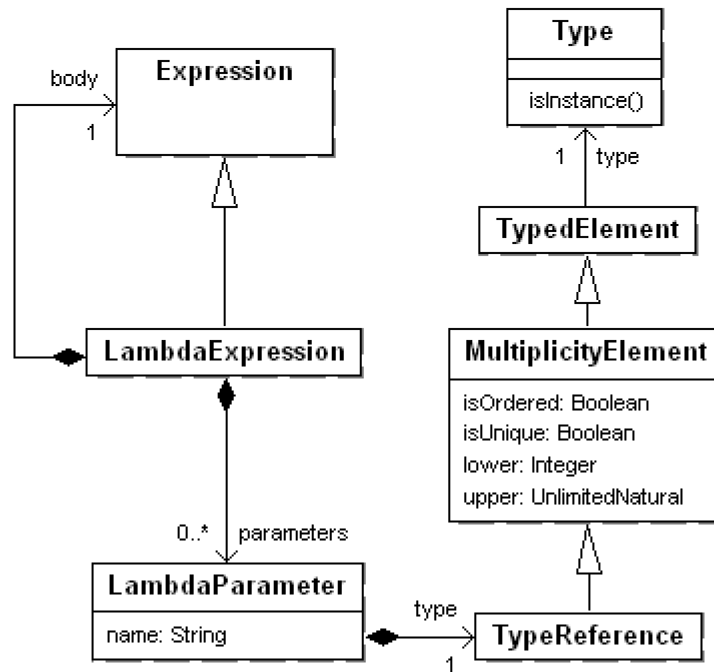

Figure 3.10. Kermeta lambda expressions

- A basic lambda expression.
  - *i* is a LambdaParameter, which *type* is *Integer*
  - *i.plus(4)* is the body of thie lambda expression

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

- A lambda expression with many 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}
```

CHAPTER 4

Examples

4.1. Hello world

Here, there is the well-known “helloworld” example. In this example, we define a class called “Hello” which contains a simple operation named “sayHello”. This operation doesn't take any parameters and just prints “Hello the world” on the terminal.

```
@mainClass “HelloWorld::Hello”
@mainOperation “sayHello”

package HelloWorld;

using kermeta::standard

class Hello
{
    operation sayHello() is
    do
        stdio.writeln(“Hello the World !”)
    end
}
```

As we explain in previous section, KerMeta describes meta-models with their operationnal semantic. So, contrary to a Java program, there is no implicit entry point in your model and your have to use the two specific annotations “mainClass” and “mainOperation” to specify which method is the entry point of the execution.

4.2. Simple State Machines

Here there is a more complicated example which provide a Finite State Machine (FSM) Meta-Model. A finite state is defined by a set of state (containing the initial state) and a set of transition which link two states together. Each transition are describe with a character they require as input and the caracter they produce as output.

Here, we present this finite-state machine in a specific graphical syntax where states are represented as circles and transitions by arrow between circles. Inputs and outputs are present above transitions. Here, “a/b” say that we consume an “a” to produce a. The following state machine works on simple words built with the {a,b} alphabet and replaces “ab” sequence by “ba” sequences and vice-versa.

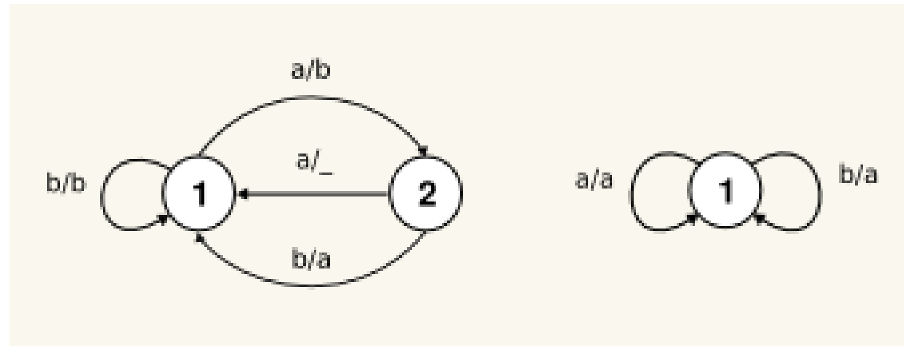

Figure 4.1. sample state machine

As FSM consume and produce caracters, we can express intuitivelly the operationnal semantic as follow :

“For all character of the input string, we need to find a transition “t” among the outgoing transitions of the current state and produce the output character of t. Then, we move to the target state of the transition t.”

In KerMeta we need to express the structure and the operationnal semantic in an object oriented way. To do that, we define a class FSM which refers a set of state, an initial state and a current state. Each state refers a set of outgoing transition and a set of incoming transition. To express the semantic, we define a “run” operation in FSM class, a “step” operation in the State class and a “fire” operation in the transition class. The fire operation consume a the input character and produce the output character. The “step” operation select in the outgoing transition set, a transition triggered by input character. The run operation processes each character of the input string.

This metamodel is presented in the following figure in a UML class diagram syntax.

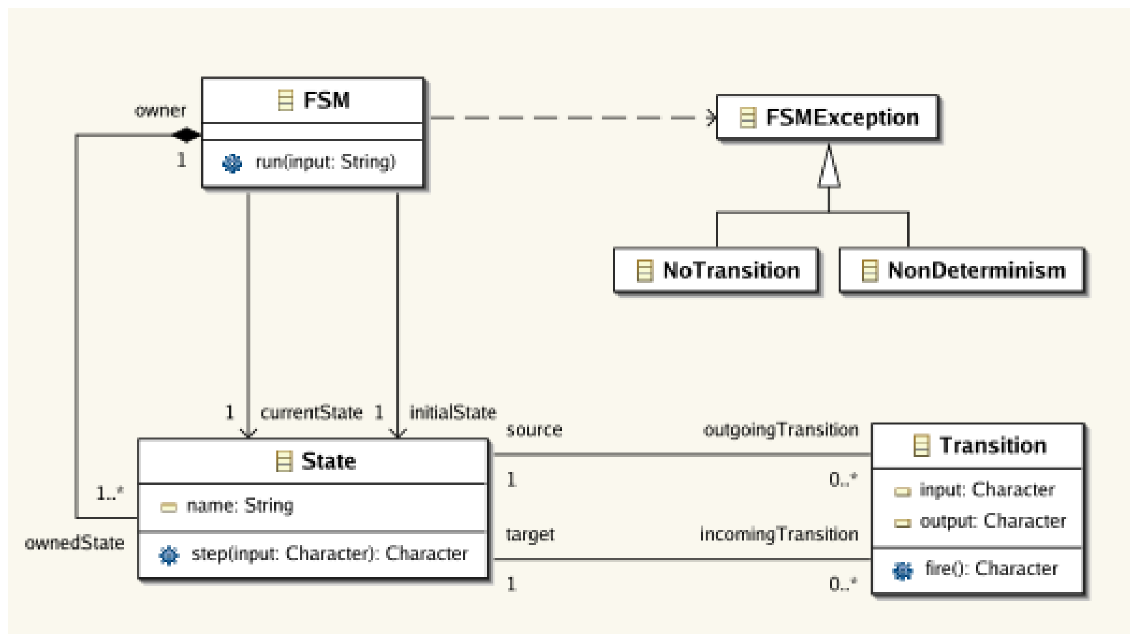

Figure 4.2. Simple State Machine metamodel

```
package fsm;

using kermeta::standard #

class FSM #
{
    attribut ownedState : set State[0..*]#owningFSM
    reference initialState : State[1..1]
    reference currentState : State

    operation run(input : String) : String raises FSMException is #     do
        // reset if there is no current state
        if currentState == void then reset end
        // initialise result
        result := ""
        from var i : Integer init 0
        until input.size == i
        loop
            result := result +
```

```
                currentState.step(input.charAt(i))
            i := i + 1
        end
    end

    operation reset() : Void is #
    do
        currentState := initialState
    end
}
```

First, we need to define a package that will contain all our classes (cf. #). After, the package declaration, all classes and package define in the file will be part of this first package.

Then, we define a class FSM (cf. #) that represente a root element. Here, a FSM object contains its states and its transitions. We define the mehod “run” (cf. #) that we present previously and we add a operation called “reset” which restart the FSM (cf. #).

```
class State
```

```
{
    attribut name : String #
    reference owningFSM : FSM#ownedState
    attribut outgoingTransition : set Transition[0..*]#source
    reference incomingTransition : set Transition[0..*]#target

    operation step(c : Character) : Character raises FSMException
    is do #
        // Get the valid transitions
        var validTransitions : Collection<Transition>
        validTransitions :=
        outgoingTransition.collect { t | t.input.equals(c) }

        // Check if there is one and only one valid transition
        if validTransitions.empty then
            raise NoTransition.new
        end

        if validTransitions.size > 1 then
            raise NonDeterminism.new
        end

        // fire the transition
        result := validTransitions.one.fire
    end
}
```

Here we define the class State (cf. #) which thee appropriate attributes and references (cf. #). Then we define

the operation “step” which select the transition to fire (cf. #).

```
class Transition {
    reference source : State[1..1]#outgoingTransition
    reference target : State[1..1]#incomingTransition
    attribut output : Character
    attribut input : Character

    operation fire() : Character is do
        // update FSM current state
        source.owningFSM.currentState := target
        result := output
    end
}

abstract class FSMException {}
class NonDeterminism inherits FSMException {}
class NoTransition inherits FSMException {}
```

Finally, we define the class transition we define the class transition with the “fire” operation. We define too the “FSMException” which is an abstract class and the two exception “NonDeterminism” and “NoTransition” which inherits from FSMException.