Kermeta tutorial

How to add behavior to a metamodel

François Tanguy, Didier Vojtisek, Cyril Faucher

Abstract

This tutorial is part of a serie of tutorials that explain step by step one of the aspect of Kermeta. This one will explain you how to add operational behavior to a metamodel.

Kermeta tutorial
How to add behavior to a metamodel
François Tanguy, Didier Vojtisek, Cyril Faucher
Published Build date: 3-November-2006

List of Figures

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 presents various aspects of the language, including the textual syntax, the metamodel (which can be viewed as the abstract syntax) and some more advanced features typically included in its framework.

Important

Kermeta is an evolving software and d espite that we put a lot of attention to this document, it may contain errors (more likely in the code samples). If you find any error or have some information that improves this document, please send it to us using the bug tracker in the forge: http://gforge.inria.fr/tracker/?group_id=32 or using the developer mailing list (kermeta-developers@lists.gforge.inria.fr) Last check: v0.3.1

Tip

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

CHAPTER 1

Prerequisites

This tutorial is one part of a bigger tutorial based on a Finite State Machine example. Even if some of its aspects are reexplained here, the reader may lack information he can find in the big tutorial.

To understand this tutorial, you must be familiar with meta model creation. If not, please read the the dedicated tutorial on "How to create KerMeta meta model".

> **Important**
>
> This tutorial does not cover the study of KerMeta language. Please have a look at the KerMeta manual located at: http://www.kermeta.org/documents/manual/

CHAPTER 2

Objectives

In this tutorial, we will see some uses of KerMeta language in order to add behavior to KerMeta meta model. We start by remembering the reader the FSM specifications. And then, we explain step by step how to add behavior to the meta model.

CHAPTER 3

Finite State Machine Specifications

We want to represent IO/state machines. Inputs and outputs can be attached on each transition. To illustrate finite state machine, here is a simple example. This state machine recognizes the "hello" motif and produces the "world" motif.

Here, we present this finite-state machine in a specific graphical syntax where states are presented as squares and transitions by arrow between squares. Input and outputs are present above transitions. Here, "h/w" says that we consume an "h" to produce a "w".

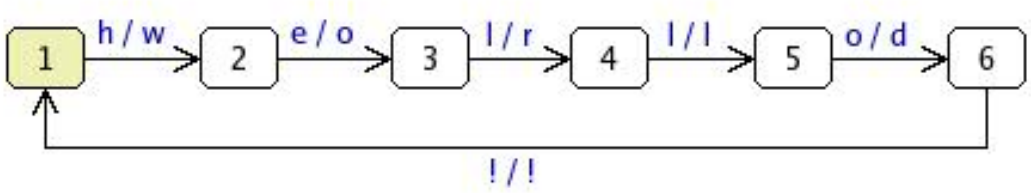

Figure 3.1.

This simple state machine can be modeled and executed easily in Kermeta. See the following meta-model presented in a class diagram syntax.

CHAPTER 4

The coding view of a meta model

It will be easier to add behavior to your meta model thanks the coding view because you can clearly see blocks of code (conditions, loops...).

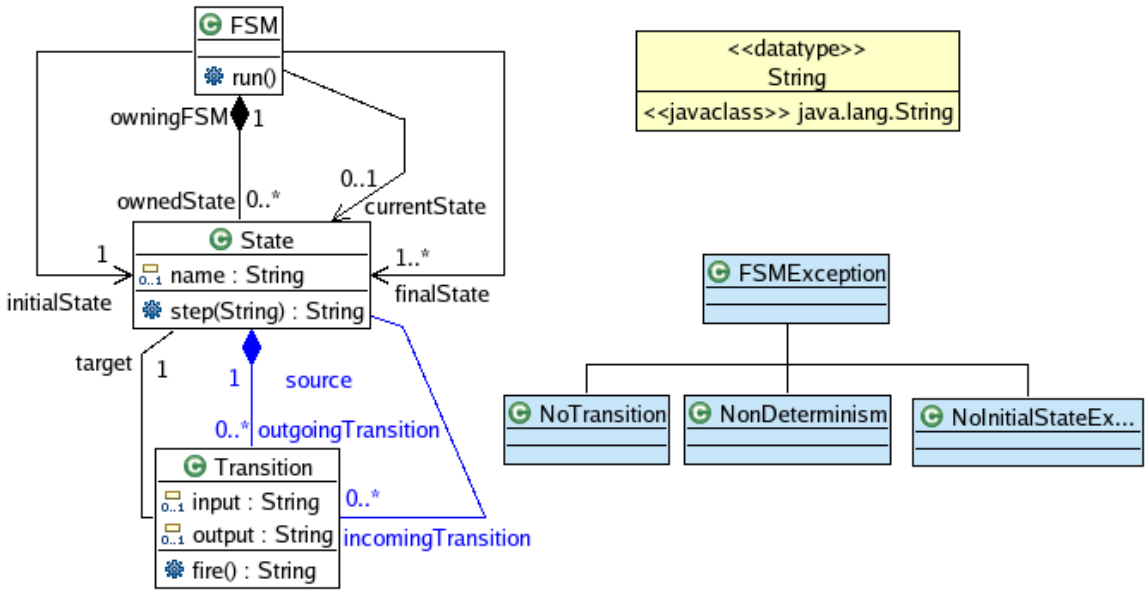

Figure 4.1.

Then have a look at the "fsm.kmt" file in the "kermeta" directory.

The class diagram suggests that we must fill in three operations :

- run() for FSM class,
- step(String): String for State class,
- and fire(): String for Transition class.

CHAPTER 5

The run/step/fire operations

- The run operation is used as a user interface. Thanks to this operation, we are going to display information about the finite state machine, read user input and process steps.
- The step operation is used to go to an other state depending on the user's input and the transitions available from the current state.
- The fire operation is used to change the current state and to get the produced string.

Let us see the behavior of these three operations.

The run algorithm

Behavior :

1 – initialize current state

2 – loop until the user's input equal to "quit"

print the current state

read a string

process a step

catch exceptions if there are some and exit the program displaying the error.

Here is the code of the operation :

```
operation run() : Void raises FSMException is do
  // reset if there is no current state
  if self.currentState == void then self.currentState := self.initialState end
  self
  from var str : String init "init"
  until str == "quit"
  loop
    stdio.writeln("Current state : " + self.currentState.name)
    str := stdio.read("give me a letter : ")
    if str == "quit" then
      stdio.writeln("")
      stdio.writeln("quitting ...")
    else
      if str == "print" then
        stdio.writeln("")
      else
        stdio.writeln(str)
        stdio.writeln("stepping...")
        do
          var textRes : String
          textRes := self.currentState.step(str)
          if( textRes == void or textRes == "" ) then
            textRes := "NC"
```

```
          end

          stdio.writeln("string produced : " + textRes)
          rescue (err : ConstraintViolatedPre)
            stdio.writeln(err.toString)
            stdio.writeln(err.message)
            str := "quit"
          rescue (err : ConstraintViolatedPost)
            stdio.writeln(err.toString)
            stdio.writeln(err.message)
            str := "quit"
          rescue(err : NonDeterminism)
            stdio.writeln(err.toString)
            str := "quit"
          rescue(err : NoTransition)
            stdio.writeln(err.toString)
            str := "quit"
          end
        end
      end
    end
end
```

The step algorithm

In this operation, there are pre and post conditions. These are conditions checked each time the operation is called. If they are evaluated to false an exception is raised. You can choose to check them or not. Please refer to the corresponding tutorial on how to use run configurations.

Behavior :

1 – Select the possible transitions.

2 – If there is none raise a NoTransition exception.

If there is more than one raise a NonDeterminism exception.

3 – If there is only one transition, call its fire operation and return its result.

```
// Go to the next state
operation step(c : String) : String raises FSMException is

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

  do
    // Get the valid transitions
    var validTransitions : Collection<Transition>
    validTransitions := outgoingTransition.select { 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

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

The fire algorithm

Behavior :

1 – change the current state of the FSM

2 – return the produced string

```
// Fire the transition
operation fire() : String is do
  // update FSM current state
  source.owningFSM.currentState := target
  result := output
end
```