

Tutorial : Building a DSL using Kermeta

FSM example

François Tanguy, Didier Vojtisek, Zoé Drey, Marie Gouyette

Abstract

This tutorial is part of a serie of tutorials that explain step by step the process explained in the Building DSL main process. In this case, it rely on the classical Finite State Machine example.

Published Build date: 3-November-2010
Last modification: \$Date:: 2010-05-25 08:39:54#\$
19/07/2006



Preface	vi
Chapter 1. Introduction	1
Chapter 2. Installation	2
2.1. Prerequisites	2
2.1.1. Required knowledge	2
2.2. Install FSM project	2
2.2.1. Install FSM tutorial	2
2.2.2. Install the tutorial 's solution	5
Chapter 3. Define the FSM metamodel	6
3.1. Fsm metamodel presentation	6
3.2. Create the fsm meta-model	7
Chapter 4. Editor	9
4.1. Dynamic instances	9
4.2. Tree view editor	10
4.2.1. Generate the editor	10
4.2.2. Use the editor	12
4.3. Textual editors	13
4.4. Graphical editors	13
Chapter 5. Model manipulation in Kermeta	14
5.1. Create a new Kermeta file	14
5.2. Package registry	16
5.3. Serialisation	18
5.3.1. Load an EMF model with Kermeta	18
5.3.2. Modify and save an EMF model with Kermeta	20
5.3.3. Create a model in Kermeta, and save it as an EMF model	20
5.3.4. A template for a complete Kermeta program	21
5.3.5. Load and save for models divided into several resources	22
Chapter 6. Check model	23
Chapter 7. Design by contract	25

7.1. Run configurations	25
7.1.1. An entry point for the program	25
7.1.2. Execution without parameters	26
7.1.3. Execution with parameter(s)	27
7.2. Constraints checking execution sample	29
7.2.1. Pre condition violation	30
7.2.2. Post condition violation	30
Chapter 8. Behaviour	31
8.1. Expected behaviour for this tutorial	31
8.2. Structuration of this behavior with aspects	32
8.3. Behavior algorithms	34
8.3.1. Run algorithm	34
8.3.2. Step algorithm	35
8.3.3. Fire algorithm	35
8.4. Run an fsm example of behaviour	36
Chapter 9. Model Transformation	37
Chapter 10. UI improvements	38
10.1. Customize the generated EMF editor	38
10.1.1. Change the editor's icons	39
10.1.2. Change text presentation	40
Chapter 11. Conclusion	41

List of Figures

2.1. Eclipse window	3
2.2. Wizard for selection	4
2.3. View of the example project	5
3.1. Example of IO/State machines	6
3.2. FSM metamodel	7
3.3. A simple fsm metamodel	8
4.1. Create dynamic instance from an Ecore file	9
4.2. Example of dynamic instance	10
4.3. Model Directory Property	11
4.4. Plugins Generated with EMF	12
5.1. Wizard to create a new Kermeta file	15
5.2. New Kermeta file	16
5.3. Register the meta model into the EMF EPackage Registry	17
5.4. Kermeta icon	17
5.5. Reload icon	18
5.6. Fsm_dyn_sample1.xmi	18
5.7. Fsm_dyn_sample1.xmi content	20
5.8. Fsm_scratch_sample view	21
6.1. Fsm_scratch_sample view.	23
6.2. Invariant error 's trace.	24
7.1. The launcher folder	26
7.2. Execution of minimization example	27
7.3. Run configurations	28
8.1. FSM metamodel with behavior	31
8.2. Example of behaviour execution	36
10.1. Import EMF and Topcased FSM Editor deployed plugins	38
10.2. Customize icons	39
10.3. FSM icon	40
10.4. State icon	40
10.5. Transition icon	40

Preface

Kermeta is a Domain Specific Language dedicated to metamodel engineering. It fills the gap let by MOF which defines only the structure of metamodels, 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 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: v1.3.2

Tip

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

Introduction

This tutorial gives an overview of many concepts used through KerMeta project creation. You can retrieve all the steps defined in the Manual Process to build a DSL on which you can iterate in order to progress in your development. Following this tutorial should give you the knowledge : to create a meta model, check its invariants, run configurations, add pre and post conditions and add behaviour on it. To cover all this features, we decided to start from an existing example. Because it is a complete KerMeta project, you should use it as a reference guide. This example is about the finite state machine which acronym is FSM. Firstly, a section explains how to load files of the example in your Eclipse workspace. The next section is dedicated to the finite state machine we want to model. You will have access to every files. Then we present the features presented above. Moreover, we present here examples of graphical editors but this step comes only when the metamodel is stabilized. In this tutorial we will have access to the base project in order to follow it (cf section Install FSM project) . You can have also have access of the solution of this tutorial (cf section Install the tutorial 's solution)

Installation

2.1. Prerequisites

Caution

KerMeta must be installed (FSM tutorial is only available in this case). If not, please read the "How to install KerMeta" tutorial.

2.1.1. Required knowledge

1. People who read this document need to have some knowledge about either EMOF or ECore metamodels
2. Since ecore is used, people should have a minimal knowledge, as end-users, of Eclipse development environment.

2.2. Install FSM project

2.2.1. Install FSM tutorial

We start from the main Eclipse window.

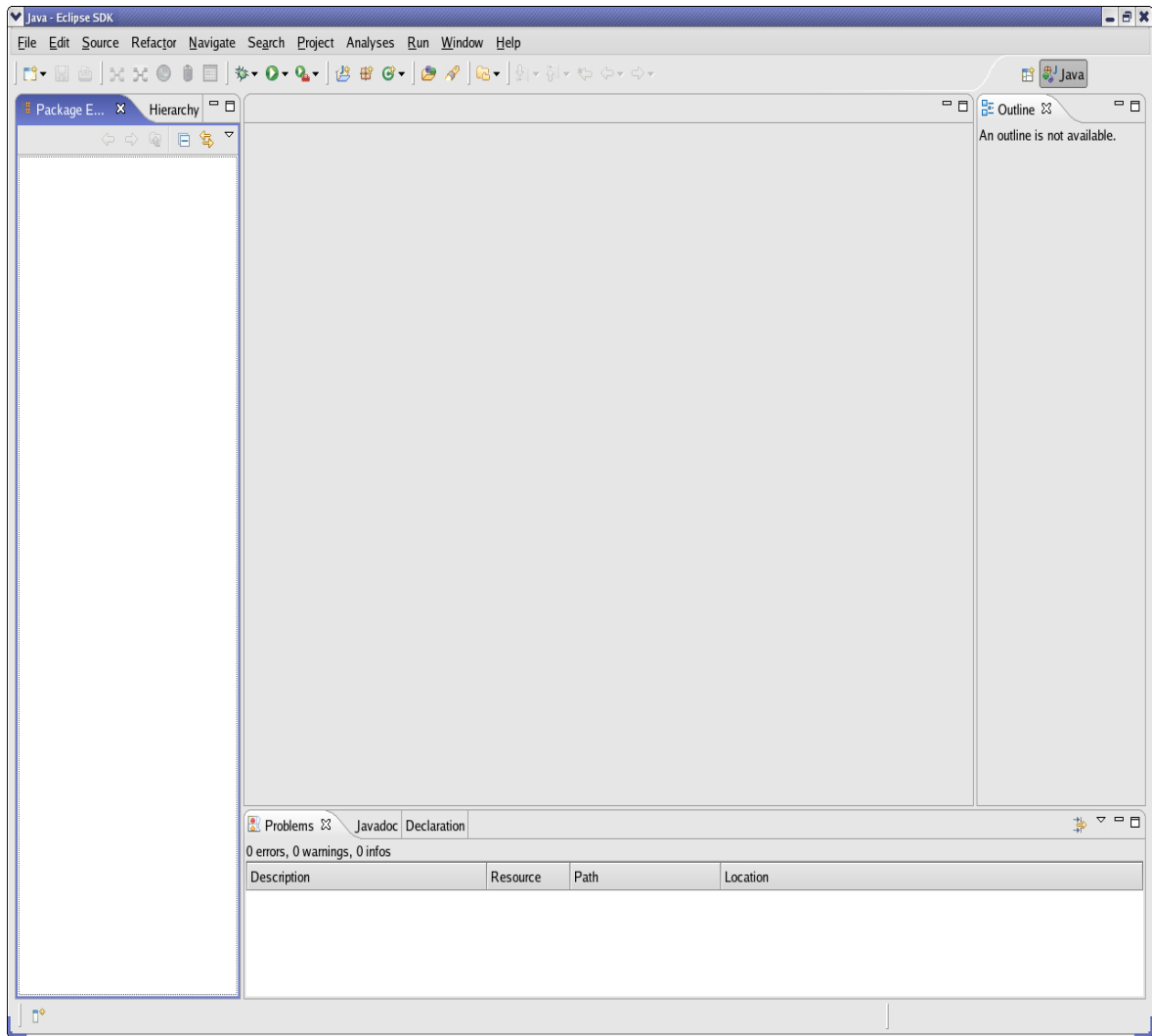


Figure 2.1. Eclipse window

Select : "File" > "New">> "Example.. A window appears. You are asked to choose a wizard for project creation. At the bottom of the list, you will find "Kermeta samples" item. Browse it and select "FSM tutorial Demo". The "Finish" button has been enabled. Click on it.

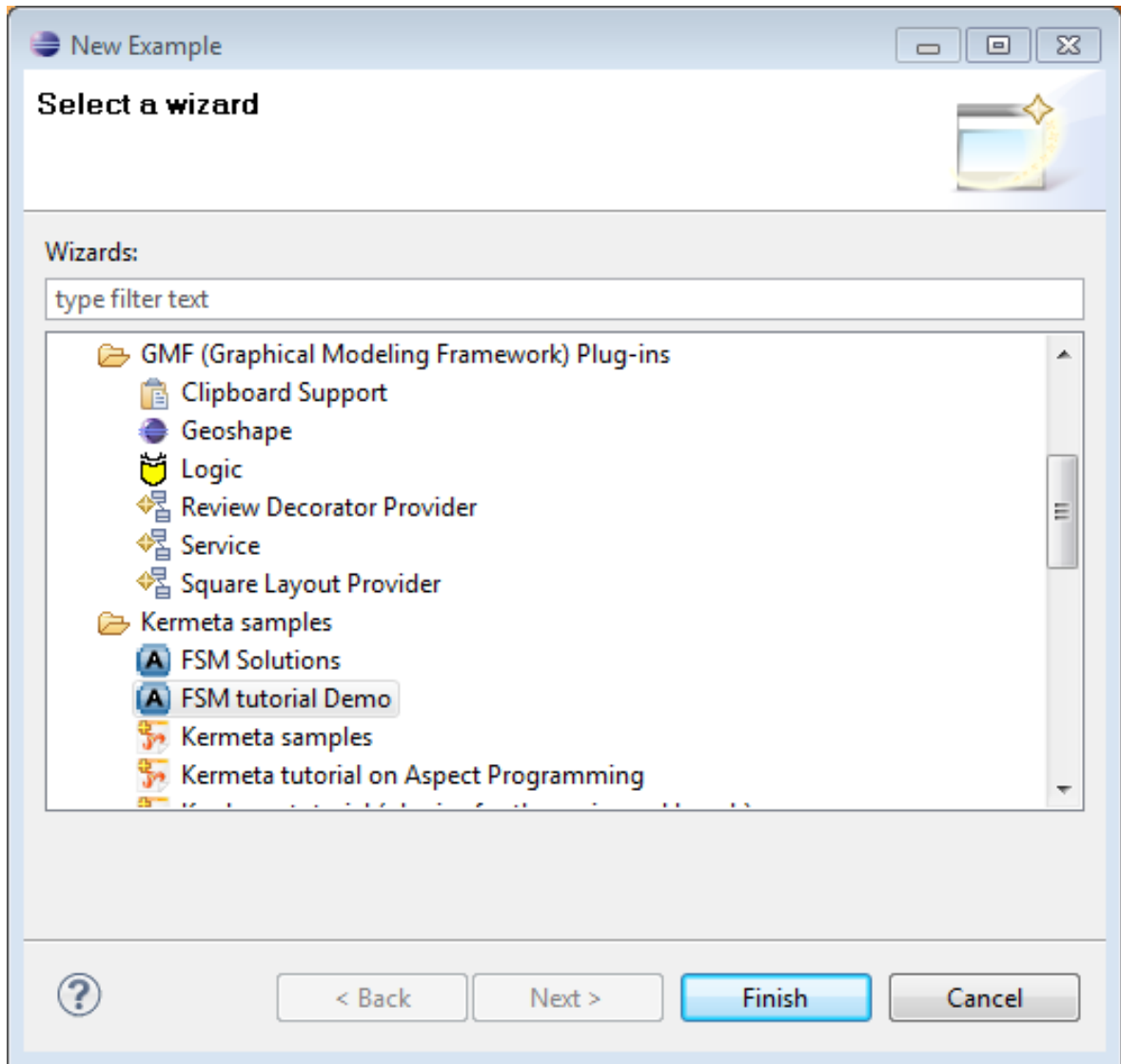


Figure 2.2. Wizard for selection

Thanks to the wizard creation project, a project named "fr.irisatriszell.kermetasamples.fsm.demosAspect" appeared on the left side of Eclipse.

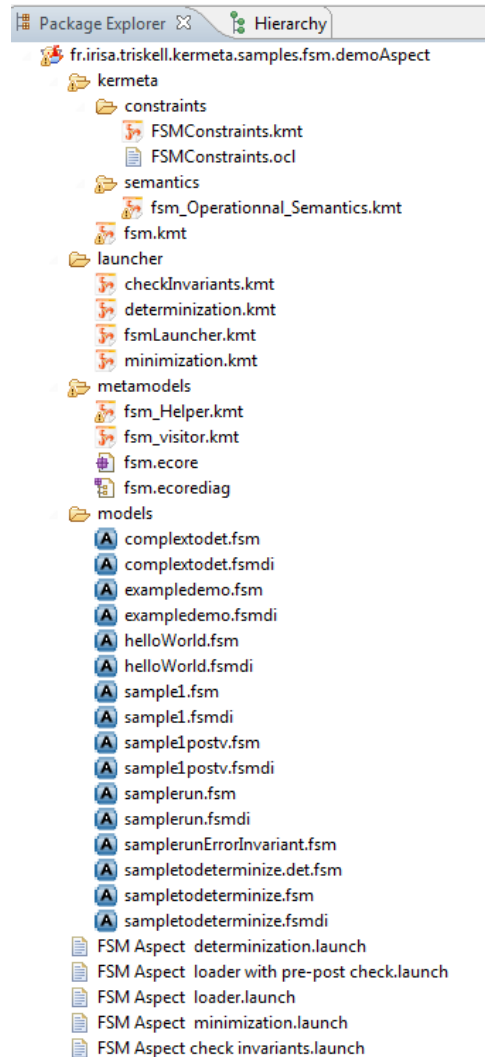


Figure 2.3. View of the example project

2.2.2. Install the tutorial 's solution

If you directly want to see the results, the solutions are available into File -> New -> Example -> Kermeta Samples -> FSM Solutions.

Now, you have the reference project and you can follow the tutorial. The next chapter presents the metamodel used in this tutorial.

Define the FSM metamodel

3.1. Fsm metamodel presentation

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 represented 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".

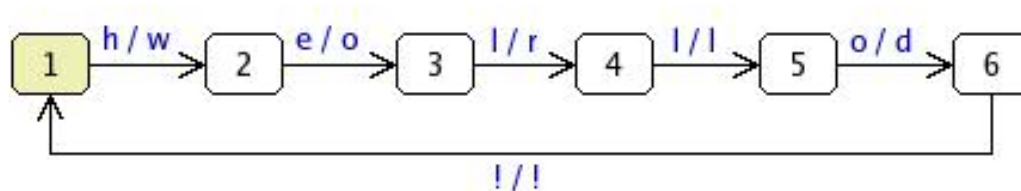


Figure 3.1. Example of IO/State machines

This simple state machine can be modeled and executed easily in Kermeta. See the following meta-model presented in a class diagram syntax. You can retrieve this metamodel named fsm.ecore into `fr.irisa.triskell.kermeta.samples.fsm.demoAspect/metamodels/fsm.ecore`

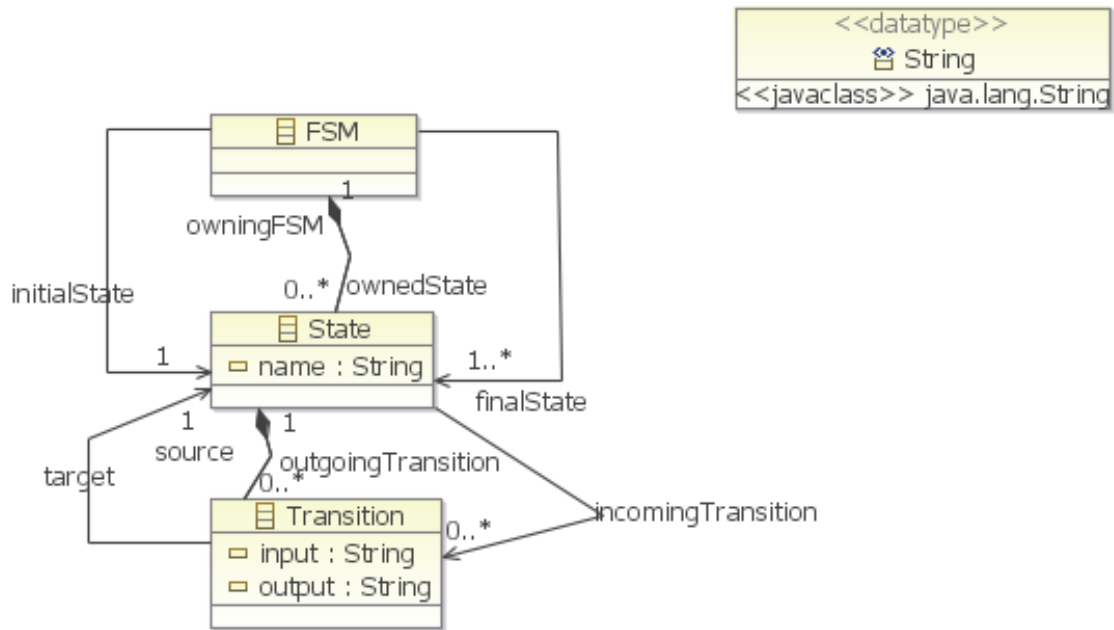


Figure 3.2. FSM metamodel

3.2. Create the fsm meta-model

In this tutorial we want to create the fsm metamodel with EMF. To start with, create a new EMF project (File-> New-> Other -> Eclipse Modeling Framework -> Empty EMF Project) and name it **org.kermeta.fsm.emf**. Then create a new folder model on it. If you want to create an EMF metamodel for fsm from scratch you can follow the tutorial "How to create an EMF meta model?" and use the EMF Sample reflexive editor to create a metamodel like in the following figure into the folder model. Name it *fsm.ecore*. Otherwise, you can simply copy the file fsm.ecore from **fr.irisa.triskell.kermeta.samples.fsm.demoAspect/metamodels/fsm.ecore** to **org.kermeta.fsm.emf/model** folder.

Tip

An FSM EMF editor is ever deployed into the Kermeta Eclipse. It is available in File-> New -> Other -> Kermeta -> FSM Samples -> Fsm Model.

At this stage of the tutorial, you should have the following metamodel:

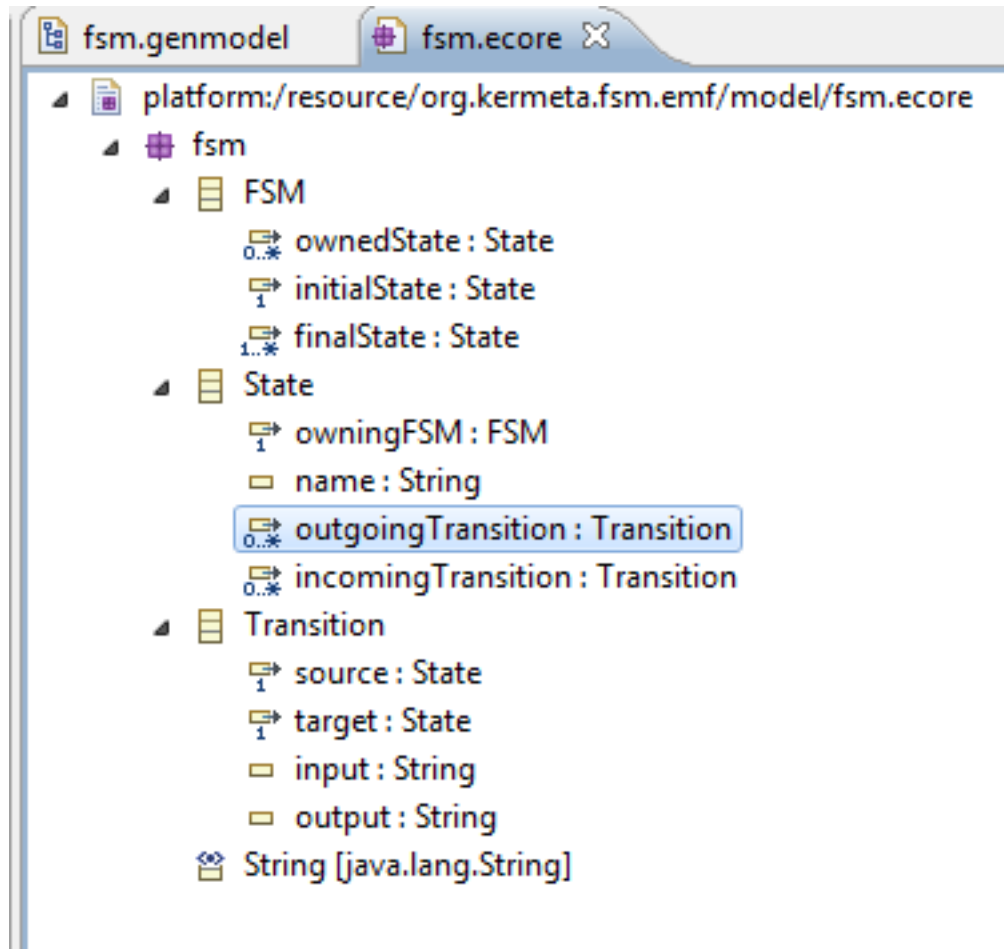


Figure 3.3. A simple fsm metamodel

Then you can use this metamodel to create instances of models conform to it, like detailed in the next section.

In this chapter we present how to create instances of the Fsm metamodel. You can use two ways to do that, creating dynamic instances directly or using an EMF tree editor.

4.1. Dynamic instances

Like shown in the Process tutorial create a dynamic instance is an easy way to create an instance of a metamodel during the first development phases. To do so :

1. Create a new folder dynamic_instance in org.eclipse.fsm.emf .Open the fsm.ecore file with the Sample Ecore Model Editor (Right click on the file -> Open with -> Sample Reflective Ecore Model Editor)
2. To open the content of this file click on the arrows at the left. Then, right click on the FSM metaclass -> Create Dynamic Instance and select the folder dynamic_instance. Name your instance fsm.xmi.

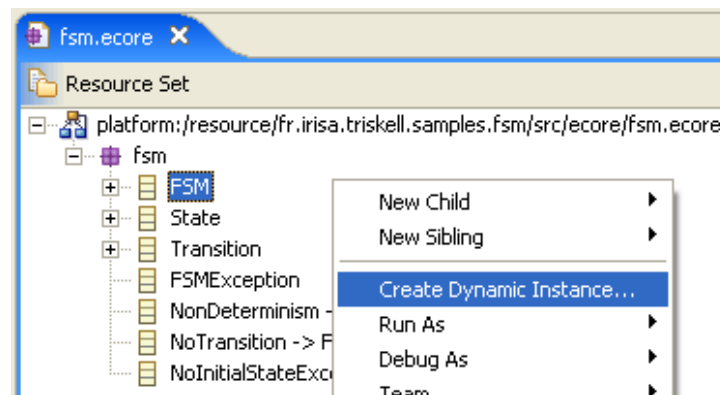


Figure 4.1. Create dynamic instance from an Ecore file

Open this instance with the Sample Reflexive Ecore Model Editor (like shown above). You can add two State st1 and st2 on the FSM class and then a transition a that produce b between st1 and st2 like in the following image :

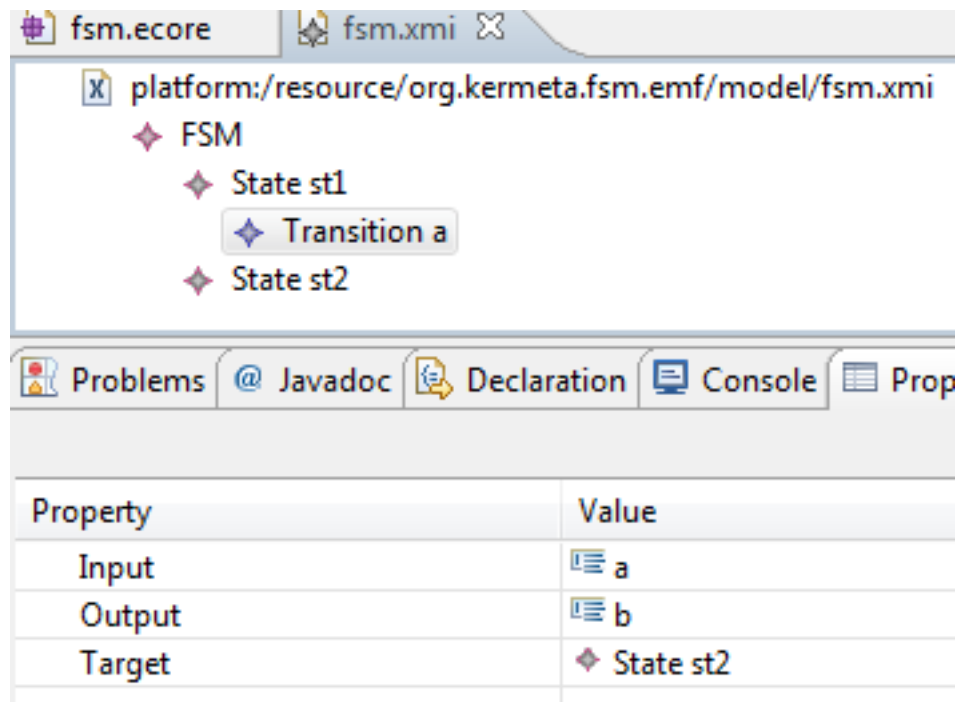


Figure 4.2. Example of dynamic instance

The Logo tutorial (chapter 4 Editor, section 4.1 Dynamic Instance) presents how to create a dynamic instance with an ecorediag. The next section presents another way to create instances of a given metamodel, create a tree view editor with an EMF project.

4.2. Tree view editor

Tree view editors should be favoured for meta-models that are relatively stable and for final development phase. It allows to customize the generated editor to match meta-model specificities. (see the end of this article: Eclipse Corner Article : Using EMF)

4.2.1. Generate the editor

This is the most ergonomic – but longest – way to create an instance of a meta-model. This method is here presented in its main lines:

1. Create a new project and a new ecore metamodel (this step was ever made in the section Create the fsm meta-model)
2. Once the meta-model is created, it is possible to create a model for model generation, called *genmodel*:
 - a. Right click on the file fsm.ecore New > **Other** > **Eclipse Modeling Framework** folder > **EMF Generator Model** and select *Next*;
 - b. Keep the proposed name (*fsm.genmodel*) and click on *Next*. Then, select Ecore model in the following

- wizard and click on *Next*;
- Here, the metamodel is ever selected. Otherwise, select **Browse workspace** button, and find the meta-model file (*fsm.ecore* into *or.kermeta.fsm.emf/model*). Then click on *Load* button and on *Next*;
 - Select the package *fsm* at the top of the window and click on *Finish*
 - In order to avoid weird behaviour (particularly if the current project was not set as a “Java project”) the model directory of the genmodel needs to be changed in the **Properties** tab of the genmodel. For this purpose, click on the root node of the *fsm.genmodel* and change the property called **Model Directory** (in the Model folder) to */org.kermeta.fsm.emf.model/src*, so that the EMF source code is generated in a new empty project that will exclusively contain this source code. You need also to change the property *Model Plug-in ID* to *org.kermeta.fsm.emf.model*.

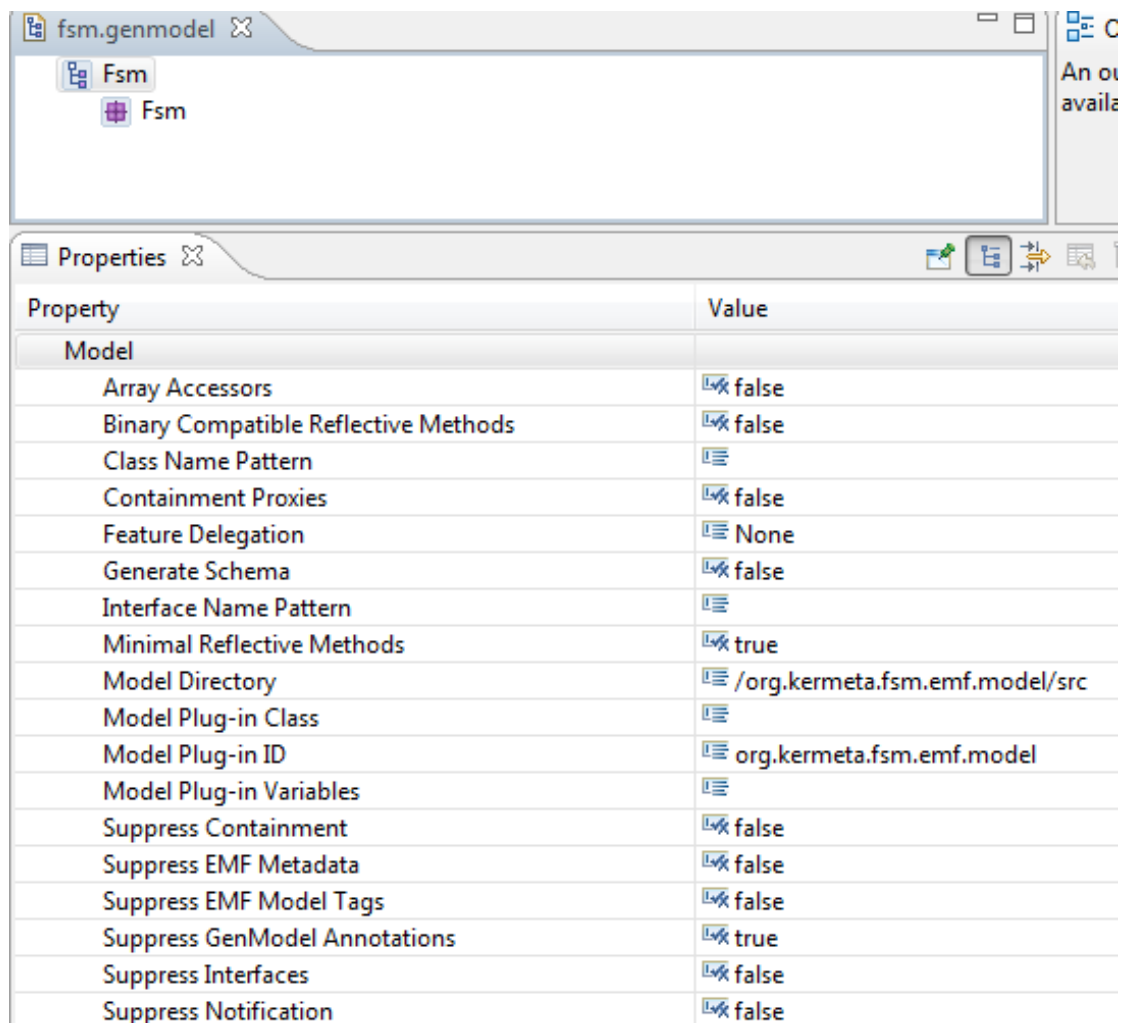


Figure 4.3. Model Directory Property

- Right-click on the package node of the *fsm.genmodel* (second sub node, it corresponds to the *fsm* package), and choose the **Generate all** item.

You should obtain the following plugins (if we not consider the folder `dynamic_instance` which is not linked with the EMF editor creation) :

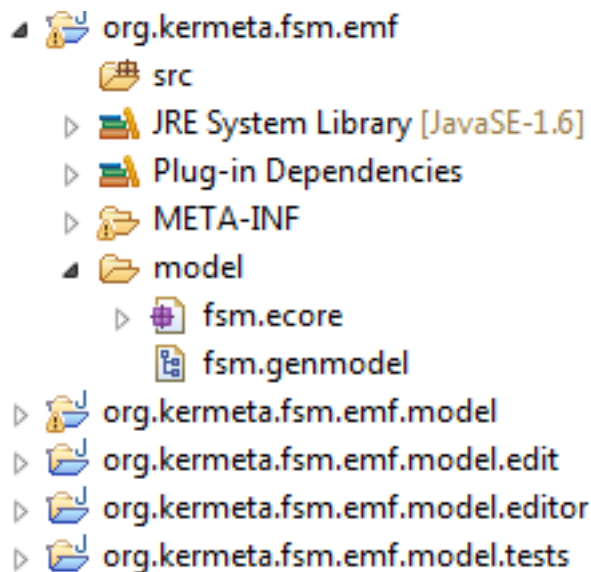


Figure 4.4. Plugins Generated with EMF

You can customize this EMF editor following the example given on the section `Customize the generated EMF editor`. The next section presents how to use this newly created tree view editor.

4.2.2. Use the editor

To be able to use the generated reflexive editor, a new runtime workbench has to be launched through the tool bar menu:

1. In Eclipse Galileo (3.5), right click on `org.kermeta.fsm.emf.editor` and select `Run As -> Eclipse Application`
2. Once the new eclipse application is launched, create a new General project (`File -> New -> Other -> General -> Project`). Call it `MyFirstEMFInstances`). Then select **File > New > Other > Example EMF Model Creation Wizards folder > Fsm Model** to create your instance file and choose `FSM` as Model Object;
3. To add elements to your model just right click on the FSM container and add new Child or new Sibling like in metamodels 's creation steps (cf `fsm.xmi` example in the section `Dynamic Instances` ;

Parallelism between edit a model or a metamodel

Creation of an EMF model follows the same principles that the creation of an Ecore model. Note that the model object, which was **EPackage** in the meta-model creation, becomes **Fsm** in the EMF model creation.

You can use another editors to create instances models such as graphical or textual editors. The following

sections presents rapidly this kind of editors.

4.3. Textual editors

You can also use textualls editors like XText , EMFText, Sintaks (which is part of the Kermeta project) . This textualls editors are not detailed here, please refer to the Logo tutorial (chapter 4 Editor, section 4.3 Textual Editor).

4.4. Graphical editors

EMF metamodels can be used with tools like GMF (Graphical Modelling Framework) or Topcased (the FSM Diagram from Kermeta -> Samples FSM was created with Topcased) to create graphical editors for model instances. The tutorial How to create a FSM graphical editor with GMF? explains the creation of an GMF editor with a very simple FSM metamodel. You can retrieve the result of this tutorial and the gmf models into fr.irisa.triskell.kermeta.samples.fsm.gmf.

You know how to create a model conform to the Fsm metamodel. The next chapter explains how to manipulate models with Kermeta.

Model manipulation in Kermeta

Firstly, we present to you how to create a KerMeta file. Then loading a first Kermeta program can be achieved by just copying the code provided in the following sections, following carefully the suggested instructions. Readers who want to directly load their own models should directly go to the section A template for a complete Kermeta program and customize the given template.

You can start with opening the Kermeta perspective with **Window->Open Perspective -> Other -> Kermeta** and create a new Kermeta project (**File -> New -> New Kermeta Project** into the Kermeta perspective). Name it **org.kermeta.fsm.serialisation**, copy the file `fsm.ecore` (from **fr.irisa.triskell.kermeta.samples.fsm.demoAspect/metamodels/fsm.ecore**) into `metamodel` folder. The following section presents how to create a new Kermeta file.

5.1. Create a new Kermeta file

We want to add a new kermeta file into the folder `kermeta` into `org.kermeta.fsm.serialisation/src`.

Tip

This sample is available in the FSM Solutions in the project `fr.irisa.triskell.kermeta.samples.fsm.serialisation`

To do this, select in the main menu of Eclipse

"File" > "New" -> "New Kermeta File"

This action opens the following window.

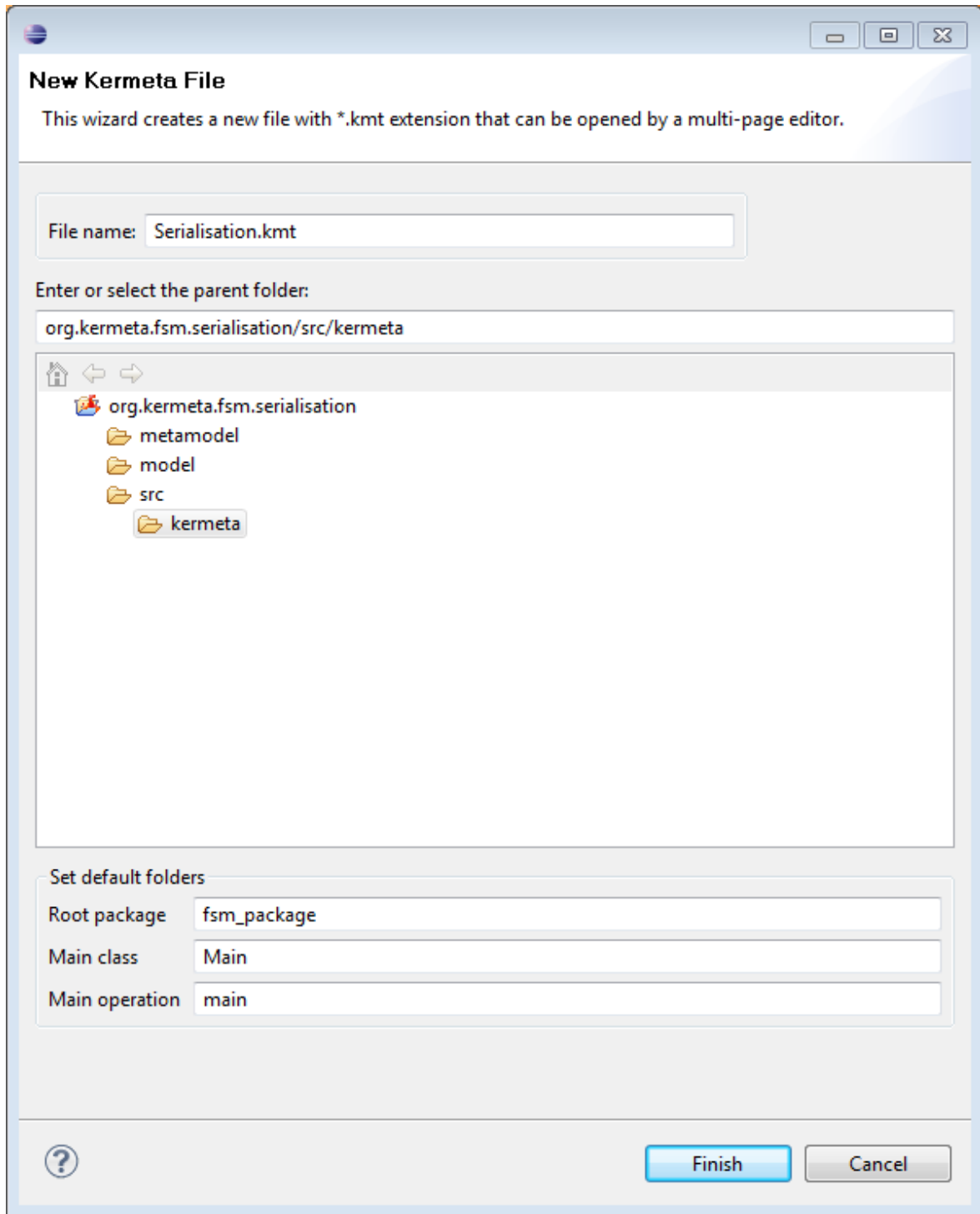


Figure 5.1. Wizard to create a new Kermeta file

Give a name (Serialisation.kmt for instance) to the new file, select a location (The default one is good for us) and change the package to fsm_package. Click on "Finish".

Now your main window should look like the one below. The file is written in KerMeta language. So you can edit this file to add some classes, attributes and so on using the KerMeta language (read Kermeta Manual for more details on KerMeta language).

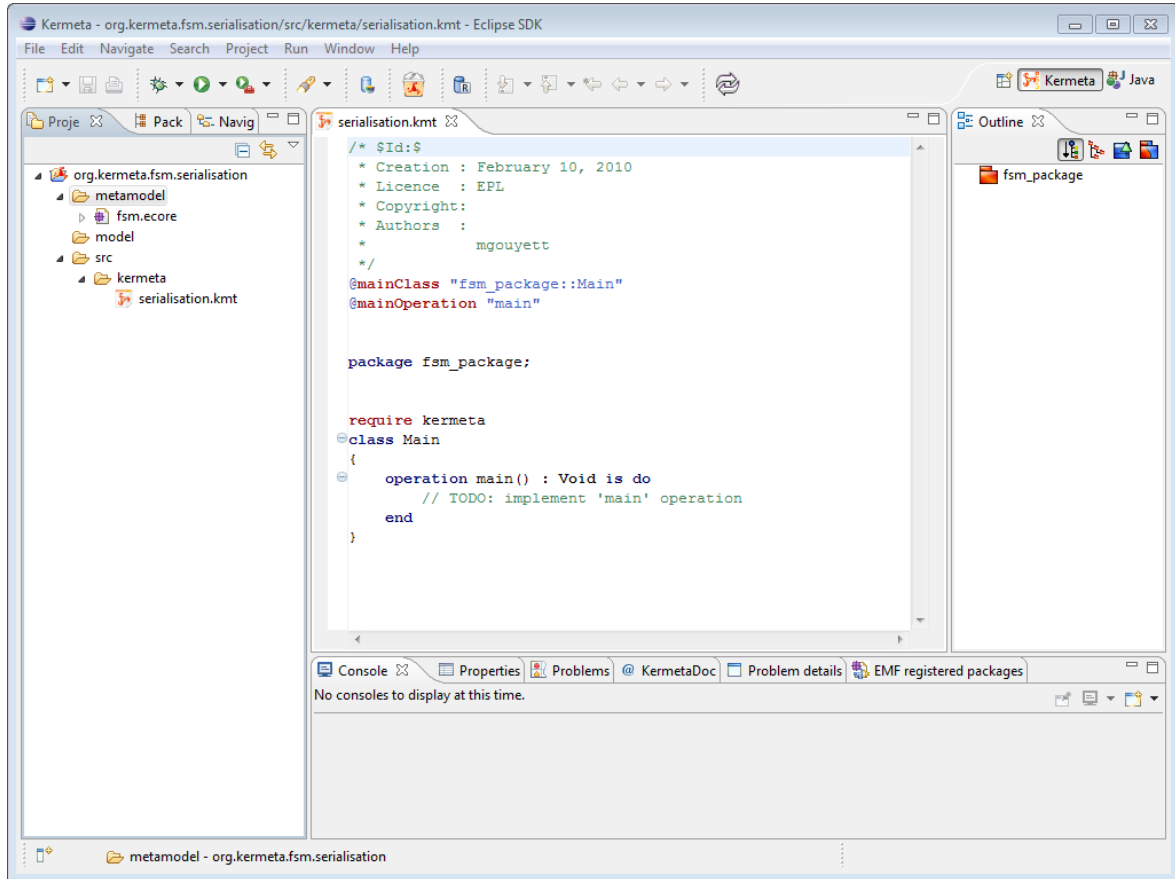


Figure 5.2. New Kermeta file

Create a Kermeta file into another perspective

If you are not into the Kermeta perspective you can create a new Kermeta file with File -> New -> Other -> Kermeta -> New Kermeta File

Add the Kermeta nature to a project

If your project is not a Kermeta project you can add the Kermeta nature on it. Right click on the project Kermeta-> Add Kermeta nature. It permit to check the .kmt files at the beginning of Eclipse.

Now, we add Kermeta code to load and save models compliant with the fsm.ecore metamodel. For this you need to register the EMF metamodel into the Package Registry.

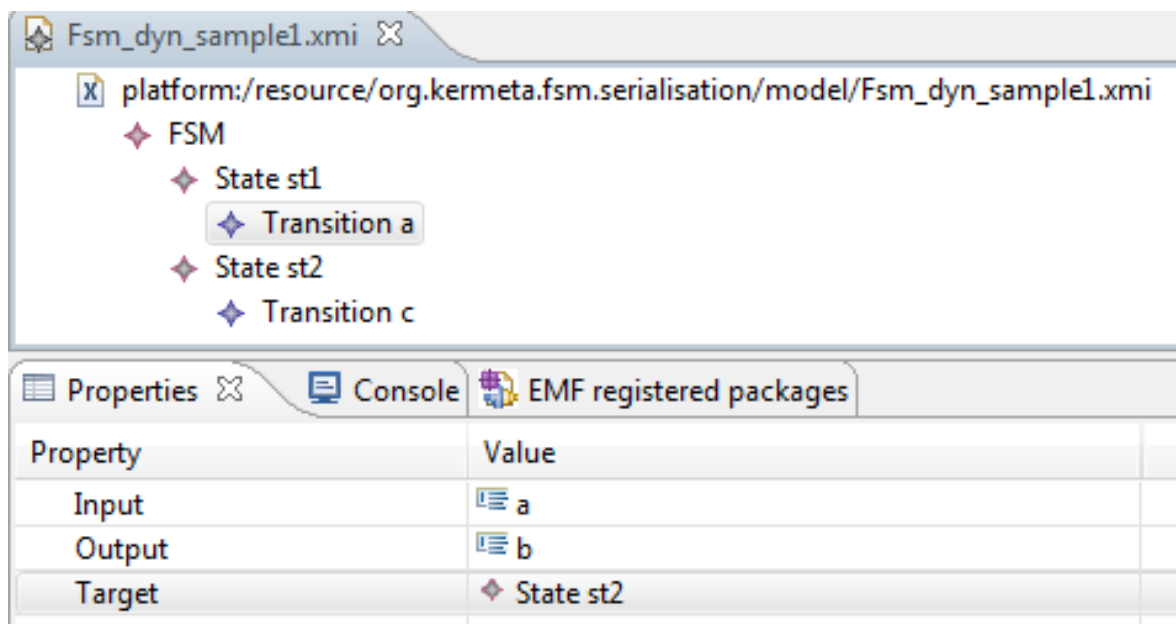
5.2. Package registry

Figure 5.4. Kermeta icon*Figure 5.5. Reload icon*

Now, we use Kermeta to load and save models i.e. to serialise the models. So, register the `fsm.ecore` model as we presented above to use it in the example.

5.3. Serialisation

By serialisation, we mean here : to save and load models thanks to the persistence library. For these following sections copy and paste on your new file `Serialisation.kmt` or look at the file `serialisation.kmt` into `fr.irisa.triskell.kermeta.sample.fsm.serialisation/kermeta`. Create a new dynamic instance for FSM named **Fsm_dyn_sample1.xmi** in the folder **model** like in the section `Dynamic instances` . Add on it two states `st1` et `st2` and two transitions. The first transition from `st1` to `st2` consume an "a" and produce a "b" whereas the second transition from `st2` to `st2` consume a "c" and produce a "d". This xmi file should look like this :

*Figure 5.6. Fsm_dyn_sample1.xmi*

5.3.1. Load an EMF model with Kermeta

The following code sample load a previously created FSM model. Copy and paste it on `org.kermeta.fsm.serialisation/src/kermeta/serialisation.kmt` or see the `serialisation.kmt` file on `fr.irisa.triskell.kermeta.samples.fsm.serialisation/src/kermeta/serialisation.kmt`.

```

    @mainClass "fsm_package::Main"
    @mainOperation "main"

    package fsm_package;

    require kermeta
    require "http://www.kermeta.org/fsm"
    ❶

    using kermeta::persistence
    ❷

    using kermeta::standard
    class Main
    {
    operation main() : Void is do
    // Input fsm
    var fsm1 : fsm::FSM
    // Create the repository, then the resource
    var repository : EMFRepository init EMFRepository.new
    var resource : EMFResource
    resource ?= repository.getResource("platform:/resource/org.kermeta.fsm.serialisation/model/Fsm_dyn_sample1.xml")
    ❸
    resource.load

    // Load the fsm (we get the instance)
    fsm1 ?= resource.one
    ❹
    // Check that the fsm was correctly loaded
    fsm1.ownedState.each { s | stdio.writeln("-> "+s.name)
        s.outgoingTransition.each { ti | stdio.writeln( "outgoing : " + " source " + ti.source.name + " target " + ti.target.name + " input")
        s.incomingTransition.each { to | stdio.writeln( "incoming : " + " source " + to.source.name + " target " + to.target.name + " input")
    }
    }
    end
    }

```

- ❶ To serialize the class definition, the nsURI of the meta model need to be known.
- ❷ This method permits to retrieve the resource where the model is stored or create a new resource if it does not exist.
- ❸ The persistence permits to load and save EMF models. The key word *using* is used to simplify writing code like import in Java.
- ❹ resource is collection of instances which are in the root file. The first instance of this resource is retrieve by the method one.

Browse through the resource

You can use **resource.each** if you want to browse through the resource.

Create new resource for your instance model

You can create a nex resource to contain your model with the method repository.createResource(NsURI instance model, NsURI metamodel)

repository is an EMFRepository that contain the resource.

Example 5.1. Load an EMF model with Kermeta

To run this file simply right click on it (*Run As -> Run As Kermeta Application*). You should obtain the fol-

lowing trace in the Eclipse's console :

```
-> st1
outgoing : source st1 target st2 input : a output : b
incoming : source st2 target st1 input : c output : d
-> st2
outgoing : source st2 target st1 input : c output : d
incoming : source st1 target st2 input : a output : b
```

Figure 5.7. Fsm_dyn_sample1.xmi content

5.3.2. Modify and save an EMF model with Kermeta

At this stage, it should be interesting to be able to modify a previously loaded model using Kermeta before saving it. The procedure is very simple: do your manipulation as if your loaded FSM model is a Kermeta model (which is, in effect, the case!), and then, simply call a save method on the handling resource. For this purpose, the following code can be added at the end of the *main* operation defined in the above section:

```
var newstate : fsm::State init fsm::State.new
    newstate.name := "s_new"
    fsm1.ownedState.add(newstate)
// save fsm1
resource.save()
```

A new state `s_new` appears on the file `Fsm_dyn_sample1.xmi`.

It is also possible to save the modified model in a new file instead of overwriting the initial one by using the `saveWithNewURI()` method. To this end, just replace the last line of above code (`resource.save()`) by the following one:

```
resource.saveWithNewURI("platform:/resource/org.kermeta.fsm.serialisation/model/modified_dyn_sample1.xmi")
```

5.3.3. Create a model in Kermeta, and save it as an EMF model

Saving a programmatically generated model requires to use a new specific instruction that add the created *Fsm* root class to the destination resource. The following code chunk creates a simple EMF model with 2 states (named "foo", and "bar"), and 2 transitions. Saving it then consists in adding the root class (i.e. the model object) stored in the variable `fsm2` into the resource instances.

```
var another_resource : EMFResource
    another_resource ?= repository.createResource(
        "platform:/resource/org.kermeta.fsm.serialisation/model/Fsm_scratch_sample.xmi",
        "platform:/resource/org.kermeta.fsm.serialisation/metamodel/fsm.ecore")
    var fsm2 : fsm::FSM init fsm::FSM.new
    var s0 : fsm::State init fsm::State.new
    var s1 : fsm::State init fsm::State.new
    var t01 : fsm::Transition init fsm::Transition.new
    var t11 : fsm::Transition init fsm::Transition.new
    s0.name := "foo"
    s1.name := "bar"
```

```

t01.source := s0
t01.target := s1
t11.source := s1
t11.target := s1
fsm2.ownedState.add(s0)
fsm2.ownedState.add(s1)
s0.outgoingTransition.add(t01)
s1.outgoingTransition.add(t11)
// save the from-scratch model!
another_resource.add(fsm2)
another_resource.save()

```

This program should return the following FSM model (viewed with the reflexive editor):

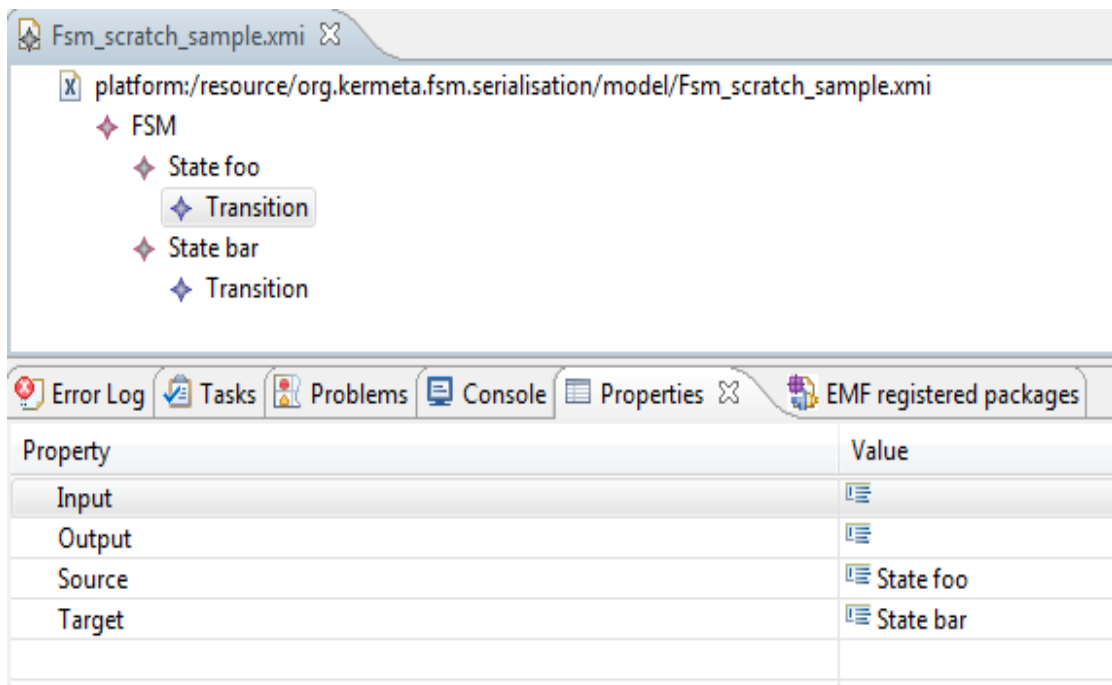


Figure 5.8. Fsm_scratch_sample view

5.3.4. A template for a complete Kermeta program

The following short code sample provides a comprehensive code template (replace the *<words>*) for model loading. Note here that the term *model object* is appropriate (better than *root class!*): loading a model consists in getting the root class, from which, thanks to the containment property (see section, all the contained instances can be accessed.

```

@mainClass "fsm_package::Main"
@mainOperation "main"
package fsm_package;
require kermeta
require "< nsURI of the metamodel >"

using kermeta::persistence
using kermeta::standard
class Main
{

```

```

operation main() : Void is do
  // Variable for your input EMF model
  var <my_model_object> : <type_of_my_model>
  // Create the repository, then the resource
  var <my_rep> : EMFRepository init EMFRepository.new
  var <my_resource> : EMFResource
  <my_resource> ?= repository.getResource(
    "<relative_path_of_my_model_to_load>",
    <my_resource>.load
  // Load the emf model - get the root class
  <my_model_object> ?= resource.one
  // You can now browse your model through its attributes/references
  <my_model_object>.<an_attribute_of_it>.each { o |
    stdio.writeln("-> "+o.toString) } )
  // Save your model in another file
  <my_resource>.saveWithNewUri("<relative_path_of_a_file_where_to_save_model>")
end
}

```

5.3.5. Load and save for models divided into several resources

Depending models on the same EMFRepository

Several models which have links between them *must* be saved into the same EMFRepository. Be careful on saving the resources (unlike load which retrieve all the resources).

If your models have no dependency between them, you can save them into several EMFRepository.

In the following chapters, we will study the sample `fr.irisa.triskell.kermeta.samples.fsm.demoAspect`.

Check model

Check all the invariants for a model can be useful if you use a model provided by a third party or resulting from a transformation, it permits to check well-formed rules. For more informations about it, please refer to the Model Checking Manual. Look at the file `checkInvariants.kmt` in `fr.irisa.triskell.kermeta.samples.fsm.demoAspect/launcher/checkInvariants.kmt`. This class permits to check all the invariants from the meta model thanks to the method `checkAllInvariants`.

```
class InvariantChecker
{
  operation main(input_automaton : String) : Void is do
  var rep : EMFRepository init EMFRepository.new
  var theFSM : FSM init AutomatonHelper.new.loadEMFAutomaton(rep, input_automaton, "http://www.kermeta.org/fsm")

  // To check all contained elements by "theFSM"
  stdio.writeln("> call of the checkAllInvariants method")
  theFSM.checkAllInvariants
  // To check only the states that are contained in "theFSM"
  stdio.writeln("> call of the checkInvariants method")
end
}
```

Look at the Run Configuration FSM check invariant. If you run it you should obtain the following trace :

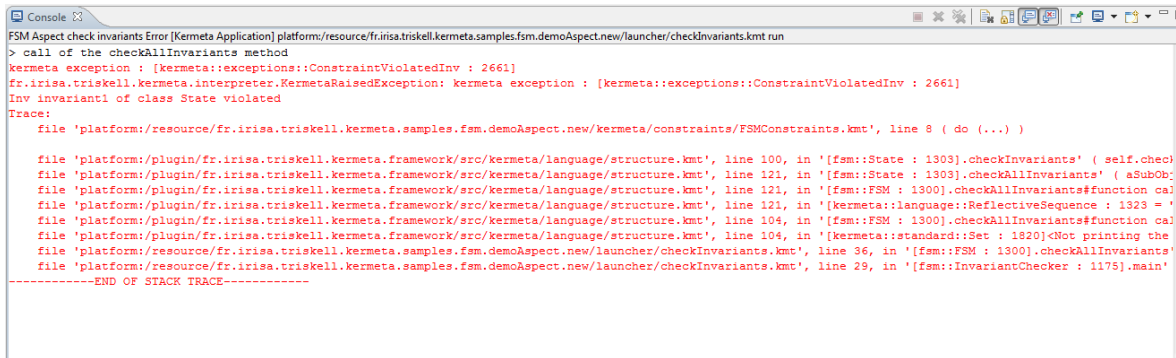
```
> call of the checkAllInvariants method
> call of the checkInvariants method
Check WFR : start
Check WFR : end
```

Figure 6.1. Fsm_scratch_sample view

Now, we will study the case of an error which violate an invariant. Look at the file `FSMConstraints.kmt` into `fr.triskell.kermeta.fsm.demo/kermeta/constraints/FSMConstraints.kmt`. You will see the following invariant

```
aspect class State{ inv invariant1 is do self.outgoingTransition.forAll{ tr1 | self.outgoingTransition.forAll{ tr2 | tr2.input.equals(tr1.input
```

It means that several outgoing transitions from a state cannot have the same input. Look at the file `samplerunErrorInvariant.fsm`. Right click on Aspect `check invariantsError` and click on `Apply` and `Run`. This fsm model have two transitions with the same input. Like invariant is violated, Kermeta raises an exception. So, you should obtain the following trace :



```
FSM Aspect check invariants Error [Kermeta Application] platform:/resource/fr.irisa.triskell.kermeta.samples.fsm.demoAspect.new/launcher/checkInvariants.kmt run
> call of the checkAllInvariants method
kermeta exception : [kermeta::exceptions::ConstraintViolatedInv : 2661]
fr.irisa.triskell.kermeta.interpreter.KermetaRaisedException: kermeta exception : [kermeta::exceptions::ConstraintViolatedInv : 2661]
Inv invariant1 of class State violated
Trace:
file 'platform:/resource/fr.irisa.triskell.kermeta.samples.fsm.demoAspect.new/kermeta/constraints/FSMConstraints.kmt', line 8 ( do (...) )
file 'platform:/plugin/fr.irisa.triskell.kermeta.framework/src/kermeta/language/structure.kmt', line 100, in '[fsm::State : 1303].checkInvariants' ( self.check
file 'platform:/plugin/fr.irisa.triskell.kermeta.framework/src/kermeta/language/structure.kmt', line 121, in '[fsm::State : 1303].checkAllInvariants' ( aSubOb
file 'platform:/plugin/fr.irisa.triskell.kermeta.framework/src/kermeta/language/structure.kmt', line 121, in '[fsm::FSM : 1300].checkAllInvariants#function call
file 'platform:/plugin/fr.irisa.triskell.kermeta.framework/src/kermeta/language/structure.kmt', line 121, in '[kermeta::language::ReflectiveSequence : 1323 = '
file 'platform:/plugin/fr.irisa.triskell.kermeta.framework/src/kermeta/language/structure.kmt', line 104, in '[fsm::FSM : 1300].checkAllInvariants#function call
file 'platform:/plugin/fr.irisa.triskell.kermeta.framework/src/kermeta/language/structure.kmt', line 104, in '[kermeta::standard::Set : 1820]<Not printing the
file 'platform:/resource/fr.irisa.triskell.kermeta.samples.fsm.demoAspect.new/launcher/checkInvariants.kmt', line 36, in '[fsm::FSM : 1300].checkAllInvariants'
file 'platform:/resource/fr.irisa.triskell.kermeta.samples.fsm.demoAspect.new/launcher/checkInvariants.kmt', line 29, in '[fsm::InvariantChecker : 1175].main'
-----END OF STACK TRACE-----
```

Figure 6.2. Invariant error 's trace

Now you know how to check models. The next section explains how you can improve kermeta programs adding on them Design by contract which consist in adding pre and post conditions.

Design by contract

By Design by contract, we mean add pre or post condition to our ecore model in order to add constraints on the Fsm's execution. It permits to use good practices into kermeta development. You can retrieve the Design by Contract concepts languages like Eiffel. Firstly, we present how to run configurations to parametrize this pre or post conditions. Then we present examples of pre and post conditions. In this section we study the example available in `fr.irisa.kemeta.samples.fsm.demoAspect`.

7.1. Run configurations

7.1.1. An entry point for the program

We want to execute an FSM model. To do that we must call the "run" operation of the "FSM" class. We are going to do that thanks to a KerMeta script. This script will : load an instance of the FSM meta model stored in a file and call the run operation of these instance.

Tip

To launch a script, the interpreter must know the entry point of the program. It can "ask" it to the user thanks to an Eclipse window. Another way might be to add the following statements into your kermeta code :

- `@mainClass` which stands for the main class,
- `@mainOperation` which stands for the main operation of the main class.

In the FSM example (`fr.irisa.triskell.kemeta.samples.fsm.demoAspect`), those scripts are in the "launcher" directory. Look at "minimization.kmt" script. Here the interpreter knows that entry point of the program is the operation "main" in the "Minimization" class.

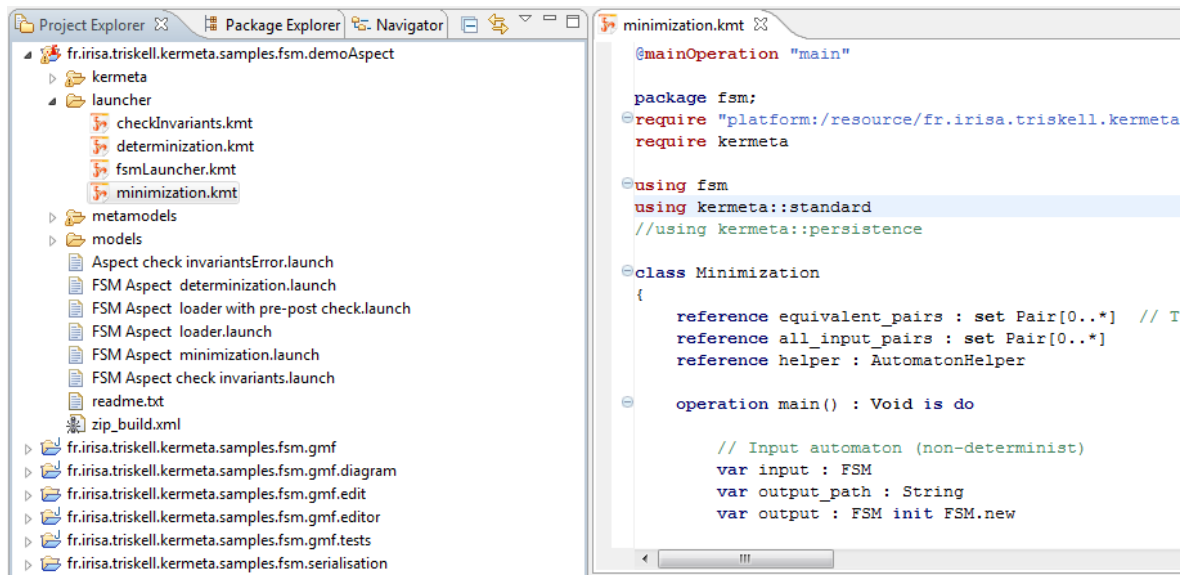


Figure 7.1. The launcher folder

7.1.2. Execution without parameters

Let's have a look at the file named "minimization.kmt". Open it. Look at the code of the main operation. There is no parameter. To run this script with constraint checking, right click on "minimization.kmt" and select "Run As" and "Kermeta Constrained Application". To run this script without constraint checking, right click on "minimization.kmt" and select "Run As" and "Run As Kermeta Application".

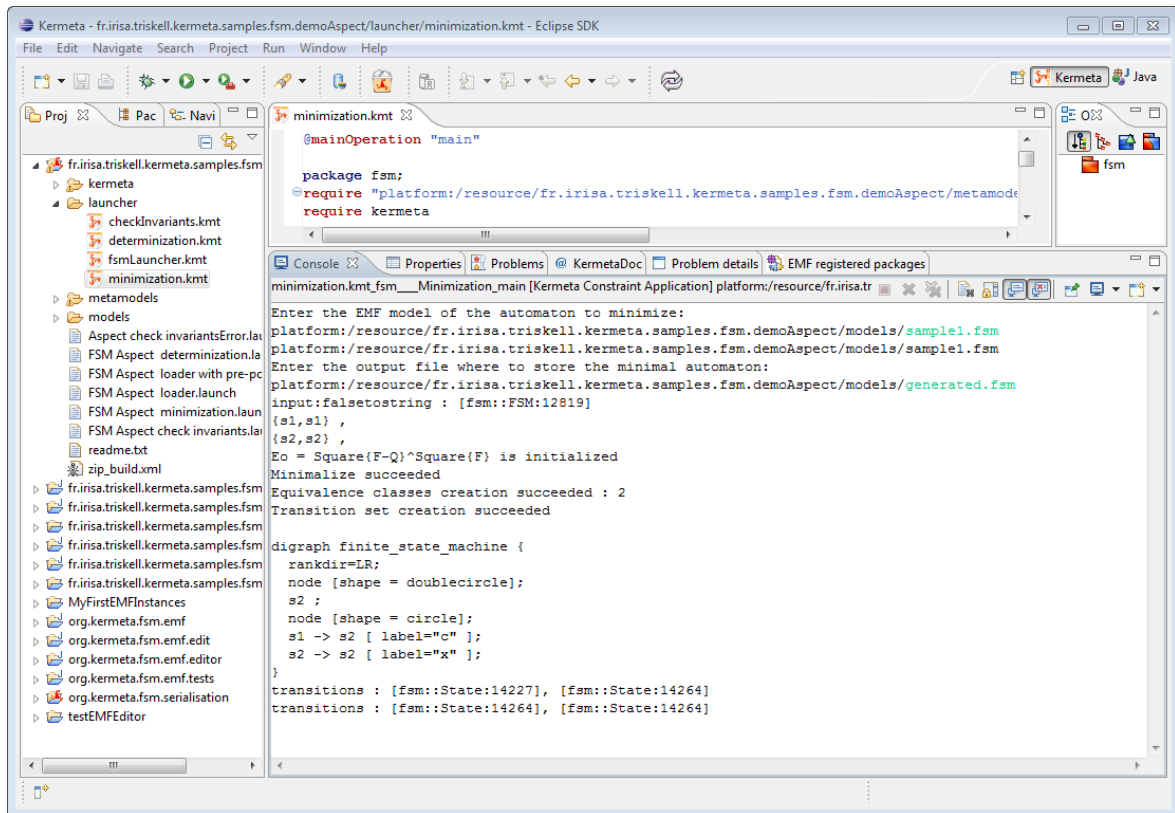


Figure 7.2. Execution of minimization example

The program asks you for a filename. Put in `./models/sample1.fsm` for example. You are lastly asked for a filename which will correspond to the file generated by the program. Put in `./generated.fsm` and see the execution.

7.1.3. Execution with parameter(s)

Now if you have a look at the three others scripts (checkInvariants, determinization and fsmLauncher) into **fr.triskell.kermeta.samples.fsm.demoAspect/launcher** you will notice that the main operation of the main class takes one argument. Let's focus on "fsmLauncher.kmt" launcher. The main operation takes one parameter which is the name of the file containing the FSM model. It loads the model, prints it and runs it. If you try the running method above, an exception is raised because the parameterized file does not exist. Indeed we did not specify any filename to the program. So, you cannot use the method above to run those kind of script. That is the reason why we are going to use run configurations. Then right click on "fsmLauncher.kmt" file and select "Run As" and "Run Configurations...". A window appears like the one below. Select the run configuration named "loaderFSM" and look at the different options. Have a special look at the file parameters :

- "Location of your program file", here this is "fsmLauncher.kmt" filename relative to the project's root directory.
- "Class qualified name", that is to say the main class of the program.
- "Operation name", that is to say the main operation of the main class.

- "Operation arguments", the parameters you want to send to the main operation.

Here, we give the string `../models/sample1.fsm` as a parameter to `mainLoadFSM` operation to `fsm::Main` class. By clicking on "Run" button, it will start the execution. You can create yourself some new run configurations. Just by left clicking on "Kermeta Application" or "Kermeta Constraint Application" (depending on the constraint checking you want) and select "New" and fill in the required fields.

Caution

Eclipse is slash sensible. It only accepts front slash and no backslash. Then `/fr.irisa.triskell.kermeta.samples.fsm.demo/launcher/fsmLauncher.kmt` is a valid filename whereas `\fr.irisa.triskell.kermeta.samples.fsm.demo\launcher\fsmLauncher.kmt` is not.

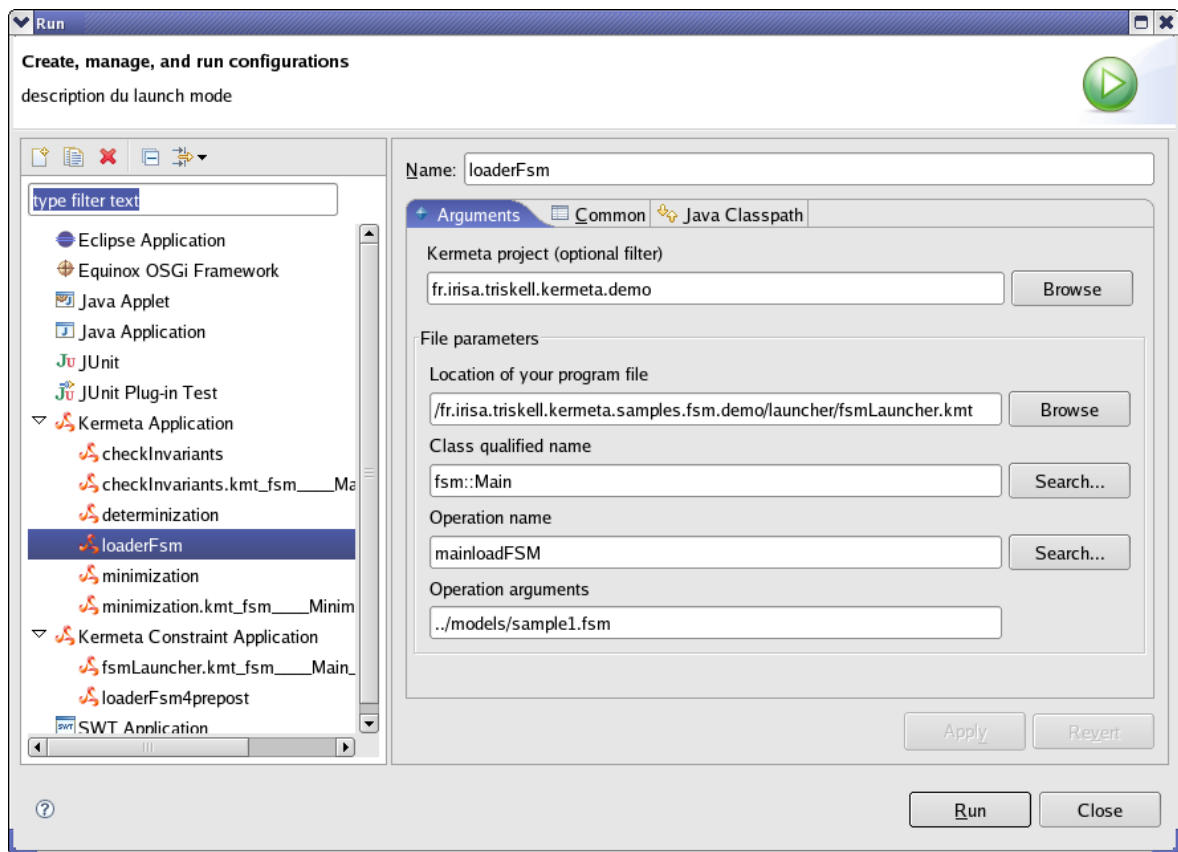


Figure 7.3. Run configurations

Tip

If you want to create a new configuration, just right click on Kermeta Application, or Kermeta Constrained Application and New. Then, fill the fields like above.

7.2. Constraints checking execution sample

This section will present an example of constraint checking execution. Have a look at the "fsm.kmt" file (into `fr.triskell.kermeta.samples.fsm.demo/kermeta`). This kermeta file requires two files :

1. `fsm_Operationnal_Semantics.kmt` from `fr.triskell.kermeta.samples.fsm.demo/kermeta/semantics/` which defines the fsm 's behaviour (cf section Behaviour)

The following code shows the behaviour of the step operation :

```
aspect class State {
    reference combination : Set<State>

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

    // Create a new state from self state
    operation copy() : State is do
        result := State.new
        result.name := String.clone(name)
        result.combination := Set<State>.new
    end
end
}
```

Tip

The key-word **aspect** (cf Behaviour) permits to add some operations.

2. `FSMConstraints.kmt` from `fr.triskell.kermeta.samples.fsm.demo/kermeta/constraints/` which defines an invariant and pre and post conditions.

```
aspect class State{
    inv invariant1 is
    do
    self.outgoingTransition.forAll{ tr1 | self.outgoingTransition.forAll{ tr2 | tr2.input.equals(tr1.input).equals(tr1.equals(tr2))}}
    end
}
aspect class State{

    operation step(c : String) :String
    pre pre2 is do
        c.equals(void).~not.~and(c.size.isGreater(0))
    end
    post post3 is do
        result.equals(void).~not.~and(result.size.isGreater(0))
    end
end
}
```

```
is abstract
```

```
}
```

There is a pre condition (**pre2**) which says that the character given as a parameter must not be void or empty string. The post condition (**post3**) says that the result must not be void or empty string. For each "step" method call, the pre and post conditions will be checked. If there are evaluated as false, the program is aborted otherwise the program goes on. Look at the run configuration named "FSM Aspect loader with pre-post check". If you do not retrieve this configuration right click on FSM Aspect loader with pre-post check.launch Run As -> FSM Aspect loader with pre-post check. Open the file (../models/sample1postv.fsm) used as parameter for this configuration. Observe the finite state diagram.

7.2.1. Pre condition violation

Execute "FSM Aspect loader with pre-post check" configuration. When you are asked for a letter , just press enter to send an empty string. Normally, it should provoke the violation of the pre condition. You should obtain the following trace into the console :

```
State : s1
  Transition : s1-(c/NC)->s2
State : s2
  Transition : s2-(x/y)->s2
Current state : s1
give me a letter :

stepping...
[kermeta::exceptions::ConstraintViolatedPre:4603]
pre pre2 of operation step of class State violated
```

7.2.2. Post condition violation

Execute "FSM Aspect loader with pre-post check" configuration. When you are asked for a letter , press c and then press enter. Normally, the post condition will be violated because the result will be an empty string. You should obtain the following trace into the console :

```
State : s1
  Transition : s1-(c/NC)->s2
State : s2
  Transition : s2-(x/y)->s2
Current state : s1
give me a letter : c
c
stepping...
[kermeta::exceptions::ConstraintViolatedPost:5548]
post post3 of operation step of class State violated
```

This chapter presented the use of pre and post condition on an execution. The next chapter explain how you can simulate an execution of the modelling system thank to behaviour.

Behaviour

8.1. Expected behaviour for this tutorial

Adding a behaviour to the FSM meta-model consists in to make a simulation of execution with operations and an execution context represented by the current state of the FSM. That's why you need to add a current-State reference and three operation :

1. run() for FSM class,
2. step(String): String for State class,
3. and fire(): String for Transition class

Adding behavior to this meta model look like change the meta model according the following schema :

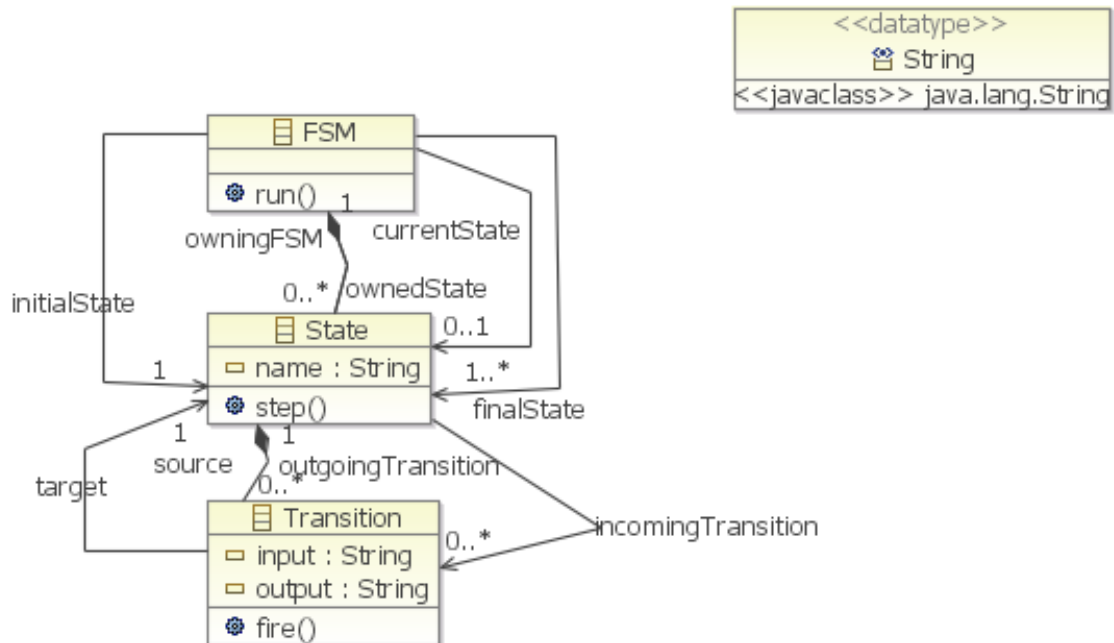


Figure 8.1. FSM metamodel with behavior

The old way to add behavior in Kermeta was transform the ecore meta model into a Kermeta file and add the code for the methods. This tutorial presents the new approach, use aspect to add this new operations. With aspect, you can add new elements and new operations to a fixed meta-model. You can also combine several aspects. This section show you how to add behaviour with aspect into a metamodel. Have a look on the file fsm_Operational_Semantics.kmt from fr.irisa.triskell.samples.fsm.demo/kermeta/semantics which contains the fsm 's behaviour operations.

8.2. Structuration of this behavior with aspects

So, this part presents the use of aspects to implement behavior. An aspect into a KerMeta file can be created simply with :

```
require kermeta
require "http://www.kermeta.org/fsm"
❶

using fsm
❷
using kermeta::standard

aspect class Transition
❸
{
  // Fire the transition
  operation fire() : String is do
  ❹
    [...]
  end
}

}
```

- ❶ You need to load the meta model where you will weave aspects.
- ❷ The key word using fsm permits to simplify the writing of the elements from the fsm meta model like an import in the Java language.
- ❸ The key word aspect is used to add attribute or operation to an existing metaclass (in this case Transition) of the loaded meta model.
- ❹ Now, you can add attribute or operation like in a classical Kermeta file.

Example 8.1. Aspect in Kermeta

None of the elements added by aspect take part of the ecore meta model, so these elements are transient that's why it cannot be serialized by this way. If you want to serialize a new element to the meta model you need to add it into.

Have a look on the Kermeta file fsm_Operational_Semantics.kmt in the folder kermeta/semantics of fr.irisa.triskell.kermeta.samples.fsm.demo. You can add several aspects in the same file. So, this code is structured like this :

```

package fsm;

require kermeta
require "http://www.kermeta.org/fsm"
using fsm
using kermeta::standard
using kermeta::persistence
using kermeta::exceptions

aspect class FSM
{
reference currentState : State
❶

// Operational semantic
operation run() : Void raises FSMException is do
❷
// [...]
end

/** Initialize a new automaton from an existing one
 * param :
 *   p_state : the initial state
 *   isInitComb
 */
operation initialize(p_state : State, isInitComb : Boolean) is do
// [...]
end
}
❸

aspect class State {
// Go to the next state
operation step(c : String) : String raises FSMException
// [...]
end
}

aspect class Transition
{
// Fire the transition
operation fire() : String is do
// [...]
end
}

```

- ❶ The reference currentState is added by aspect into the fsm meta model.
- ❷ The key word raises declare an exception that an operation can throw.
- ❸ Don't forget to close the brace at the and of an aspect definition.

Example 8.2. Structuration of the fsm_Operational_Semantics.kmt file

The next section presents the differents algorithms in details.

8.3. Behavior algorithms

Now, we will present in details the algorithm for the operations `run()`, `step(String)` and `fire (String)`.

- 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.

8.3.1. 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
        end
    end
  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

```

8.3.2. 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. The following chapter presents how to 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 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

```

8.3.3. 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
```

8.4. Run an fsm example of behaviour

In this example we execute a step into the fsm behaviour algorithm with the file `samplerun.fsm` stored into `fr.irisa.triskell.kermeta.samples.fsm.demo/models/samplerun.fsm`. The following image presents the file `samplerun.fsm`.

In this example, you use the transition `c` to go to `s1` to `s2` and produce the string `v`. the behaviour can be produced thanks to the operations `run()`, `step()` and `fire()` defined in the last section. To run this behaviour right click on *FSM Aspect Behaviour* -> Run as -> FSM Aspect Behaviour.

You should obtain the following trace :

```
State : s1
  Transition : s1-(c/v)->s2
  Transition : s1-(a/i)->s2
State : s2
  Transition : s2-(v/n)->s2
  Transition : s2-(a/i)->s2
  Transition : s2-(e/s)->s1
Current state : s1
give me a letter : c
c
stepping...
string produced : v
Current state : s2
give me a letter :
```

Figure 8.2. Example of behaviour execution

You can continue the behaviour of finite state machine if you want. This section has presented how to add a behaviour on the meta model. The next section present how to use model transformation.

Model Transformation

The files `minimization.kmt` and `determinization.kmt` from **fr.irisa.triskell.kermeta.samples.fsm.demo/launcher** are examples of transformation of one model to one another, let you have a look on it. These examples are separated into three main steps :

1. load the model to transform and initialize the output model
2. treatment of the transformation (use an algorithm to determine the new results and add it into other output model).
3. save the output model

You can execute these examples following the section Run configurations. This section give you an example of model transformation. The next section will present how to customize the EMF editor presented on section Tree view editor.

UI improvements

You can customize your user interface like shown below. For more samples please refer to the Logo tutorial (chapter 9 : UI Improvement).

10.1. Customize the generated EMF editor

Generated code can be modified for customization. An EMF editor for FSM models is ever deployed into Eclipse Kermeta. Retrieve the plugin code source with File -> Import -> Plug-in Development -> Plug-ins and Fragments and click two times on Next. Then choose the following plugins :

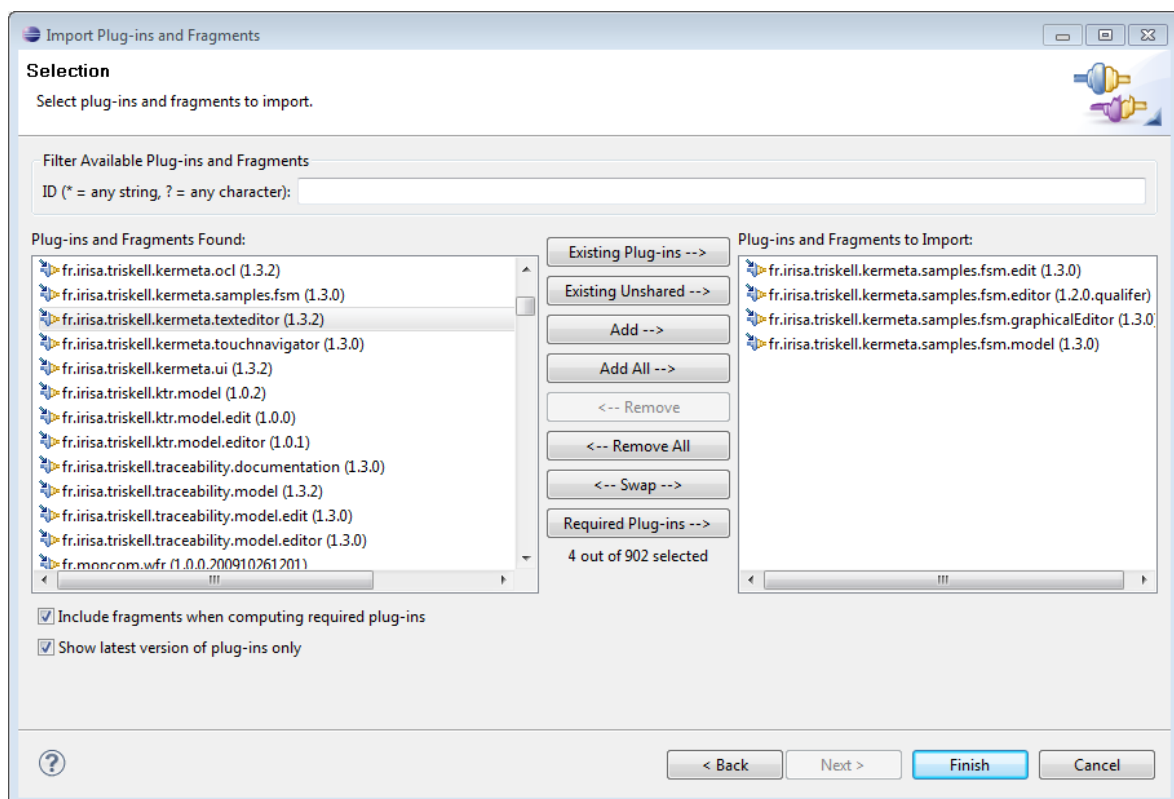


Figure 10.1. Import EMF and Topcased FSM Editor deployed plugins

The plugin .model, .edit and .editor correspond to the fsm EMF editor whereas the .graphicalEditor repres-

ents the Topcased editor.

Add custom code

To keep your customized code for each generation you need to add

```
/**
 *
 * @generated NOT
 */
```

just before the name of the method you want to customize.

10.1.1. Change the editor's icons

To change an icon you need only to replace the icon from `fr.triskell.kermeta.samples.fsm.edit/icons/full/obj16` to another with the same name and the same extension. It is the case on this deployed plugin. The following image presents this plugin, the `.gif` images have been changed from the initial EMF generation.

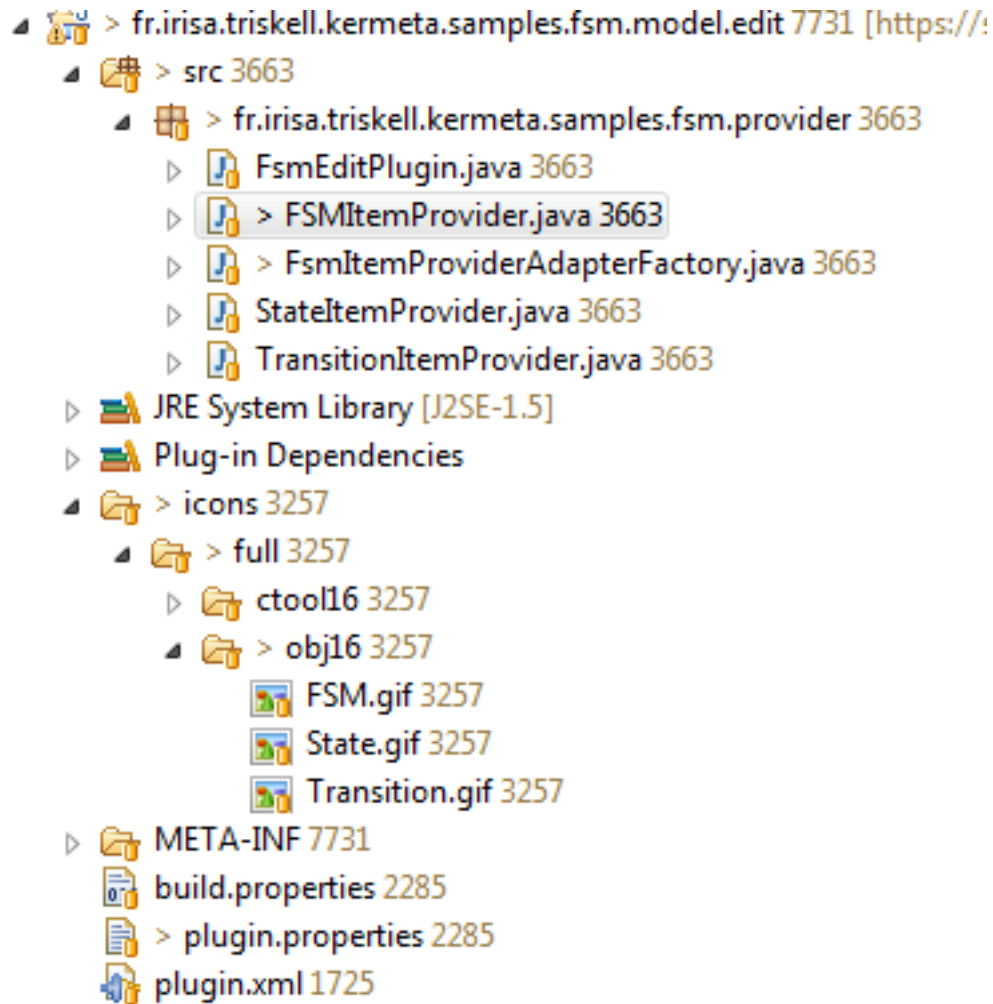


Figure 10.2. Customize icons

Respectively the icons FSM.gif, State.gif and Transition.gif look like this :



Figure 10.3. FSM icon



Figure 10.4. State icon



Figure 10.5. Transition icon

Note

Like GMF use EMF icons for the palette you can customize the palette icons by this way.

You will see the change when you will use the generated editor like in the following section.

10.1.2. Change text presentation

The following method comes from FSMItemProvider in the package fr.triskell.kermeta.samples.fsm.provider in the project fr.irisa.triskell.kermeta.samples.fsm.model.edit. It permits to display the initial state or if it is missing or if the final state is missing.

```

/**
 * This returns the label text for the adapted class.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated NOT
 */
public String getText(Object object) {

    String initialState = "[an initial state is required]";
    if(object instanceof FSM && ((FSM) object).getInitialState()!=null) {
        initialState = "[initial state = " + ((FSM) object).getInitialState().getName() + "]";
    }
    String finalState = "- [at least one final state is required]";
    if(object instanceof FSM && ((FSM) object).getFinalState().size(>=) 1) {
        finalState = "";
    }
    return getString("_UI_FSM_type") + initialState + finalState;
}

```

CHAPTER 11

Conclusion

This tutorial give an illustration of the steps presented into the Processes to build a DSL Document (cf hyperlien), in particular it focus on traitment of an ecore meta model into Kermeta. For more informations please refer to others tutorials available <http://www.kermeta.org/documents/tutorials> .