How to create an EMF model and use it in Kermeta

Zoé Drey
Didier Vojtisek
Published Build date: 3-November-2010
$LastChangedDate:: 2010-06-02 10:42:20#$

List of Figures

CHAPTER 1

Introduction

This document is aimed at people who want either to use the Eclipse Modeling Framework in order to create an EMF model, and/or to manipulate them using Kermeta. The use of EMF models in Kermeta is achieved through the following steps, which some of them are optional (they are detailed sections):

1. creating a meta-model in *Ecore* format, following a few constraints that make its instances loadable by Kermeta;
2. creating a model, also called an EMF model, as an instance of the above meta-model;
3. loading this model and manipulating it through a Kermeta program;
4. saving this model, as well as other ones programmatically created from scratch with Kermeta.

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: v0.1.0

Most of the code presented in this tutorial comes from the FSM sample available on the kermeta web site. Have a try to it for a complete working sample!

1.1. Required background

1. Readers are supposed to be familiar to either the EMOF or the Ecore meta-model.
2. The use of Ecore also supposes readers to have a minimal knowledge, as end-users, of the Eclipse development environment.

1.2. Terminology and format used in this tutorial

1. **meta-model** / **Ecore model**: both terms are used to point to the meta-model level. “Ecore model” corresponds to the Eclipse terminology for a meta-model specified in Ecore.
2. **instance**: in the context of this tutorial, the term “instance” is used to refer to an instance of a meta-model,

i.e. of an **Ecore model**. Sometimes we will talk about instance-model (an Ecore model which is an instance of the meta-model), so, don't get confused.

3. **model object / root class**: in the scope of the EMF dynamic editor, these terms designate the model root class that contains all other ones. This root class is the only one in a model that has no container (i.e. which has no “black-diamonded” relation pointing to it).

4. **containment**: this term designates an important property of classes attributes/references. This property has to be appropriately set, in such a way that all the elements of an instance-model should be accessed from the root class of the model, which is aimed at being the highest-level container (you will see that at the 10th step of section ???, p.???, and fig).

1.3. Required environment

The required environment, whatever the underlying operating system, for handling EMF models and being able to run the samples provided in this tutorial includes:

1. Java 1.5;

2. Eclipse, preferably the 3.1.1 version;

3. Ideally, Omondo, that provides a graphical editor of Ecore meta-models;

1. **Kermeta plugin (remote site for a direct eclipse installation: http://www.kermeta.org/update), version 0.1.0;**

2. **EMF modeling framework, version 2.1.0 (remote site: Eclipse update site).**

Required plugins can be found by using Eclipse installation tool, through the tool bar menu:**Help > Find and install > Search for new features to install > Next buttons**

CHAPTER 2

Create a meta-model for Kermeta programs

For simplicity sake, the different steps (meta-model, model creation, as well as model handling through Kermeta) introduced in the scope of this tutorial are illustrated by means of the finite state machines classical example.

2.1. Editing a new meta-model with the EMF sample editor

1. **Create a new simple project (for instance MyFirstEMFSamples).**
2. **Select in the tool bar menu (on top of Eclipse window)File > New > Other > Example EMF Creation Wizards** folder **> Ecore Model**

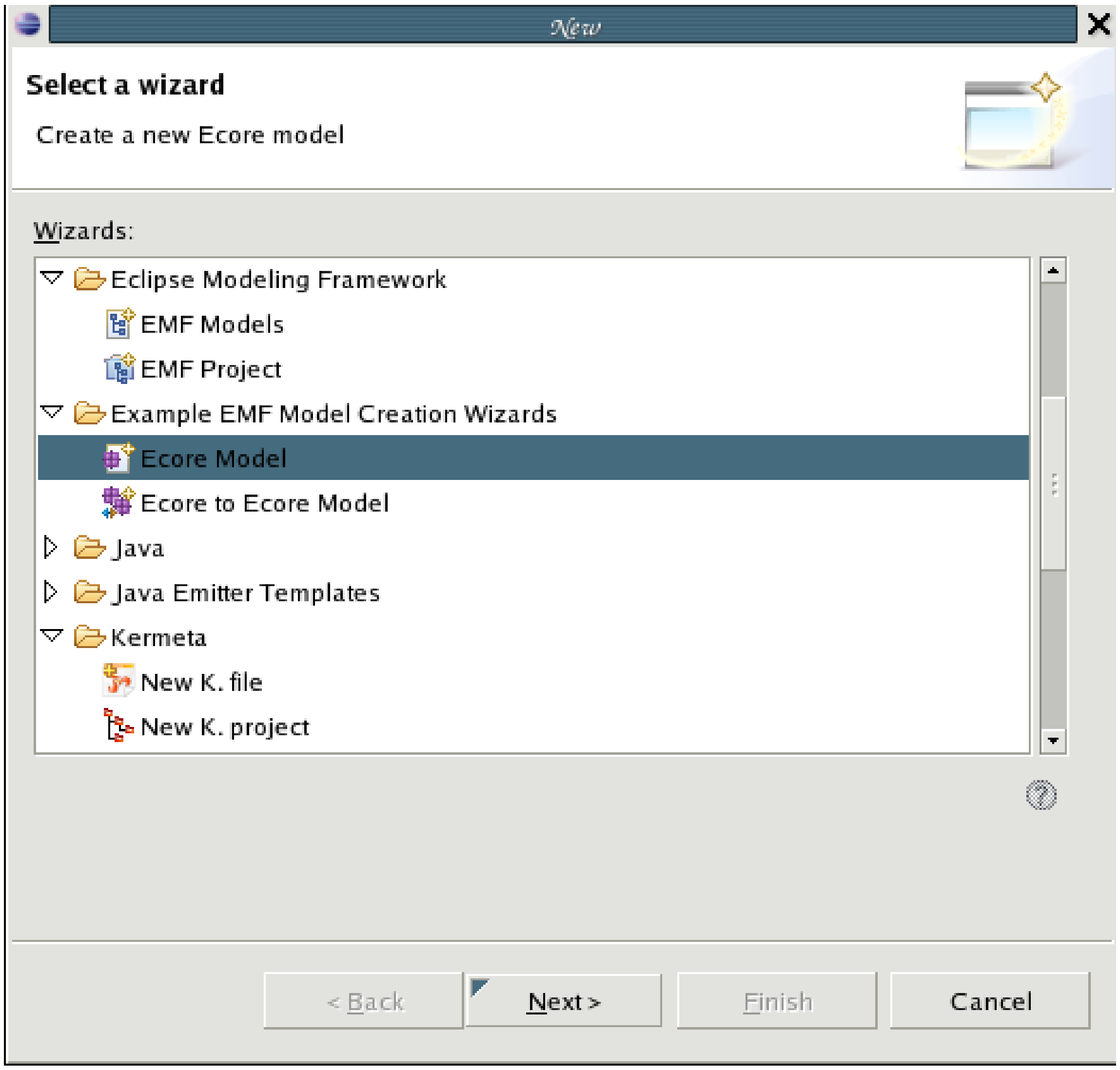
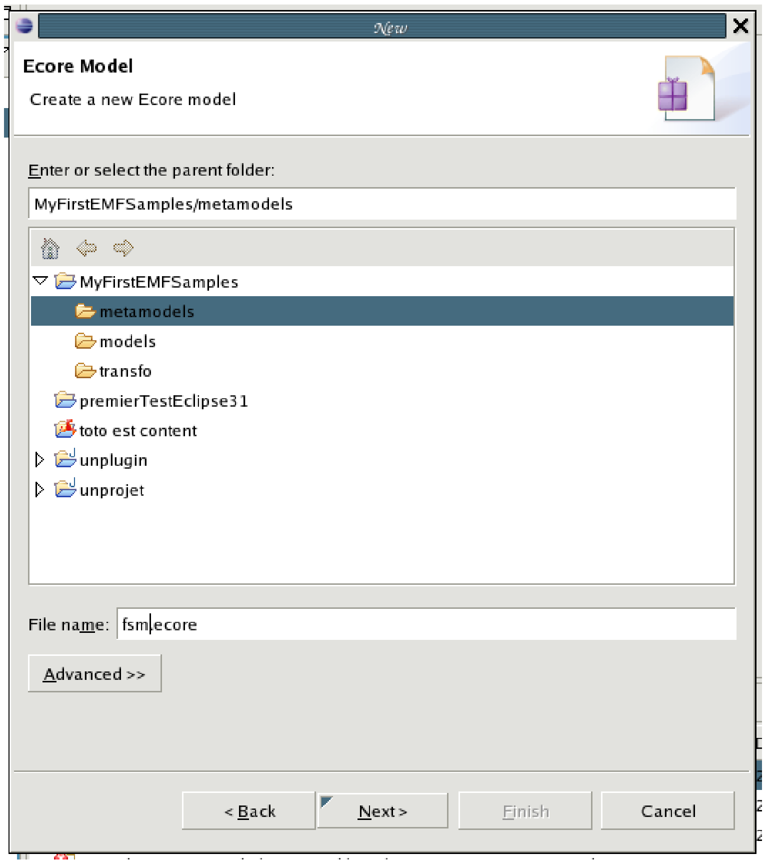


Figure 2.1. Selection of the model object (root element)

3.



Ecore Model

Create a new Ecore model

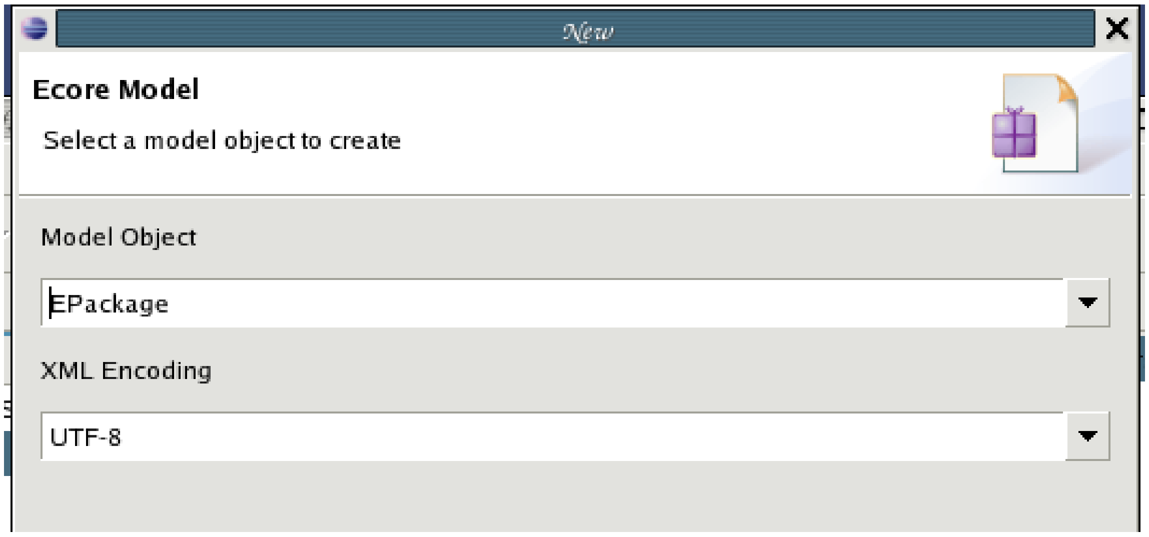
Enter or select the parent folder: MyFirstEMFSamples/metamodels

- MyFirstEMFSamples
  - metamodels
  - models
  - transfo
- premierTestEclipse31
- toto est content
- unplugin
- unprojet

File name: fsm.ecore

Enter a name for the meta-model file (e.g. *fsm.ecore*), then click on **Next** button.

4. Choose **EPackage** as the Model Object (i.e. the root of the meta-model), and click on the **Finish** button.



Ecore Model

Select a model object to create

Model Object: EPackage

XML Encoding: UTF-8

Figure 2.2. Selection of the model object (root element)

5. If Omondo is installed, the meta-model can be edited by means of the Omondo graphical editor. However, this tutorial focuses on the creation of meta-models using the **Sample Ecore Editor**. In case the Omondo editor is set as the default editor, the sample Ecore editor can be opened (after having closed the Omondo editor) by right-clicking onto the Ecore file and selecting **Open with** > **Sample Ecore Editor**.

6. If the **Properties** view is not visible at the bottom (or left, or right) part of the Eclipse window, open it using the tool bar menu.**Window** > **Show View** > **Other** > **General** folder > **Properties**;

7.

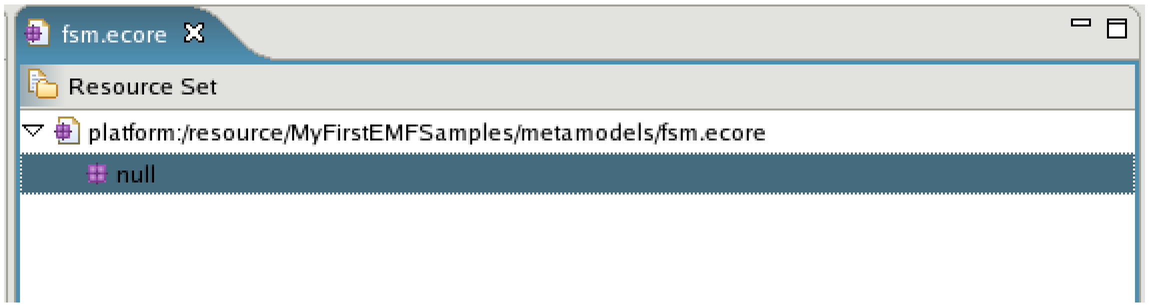


Figure 2.3. New empty ecore model

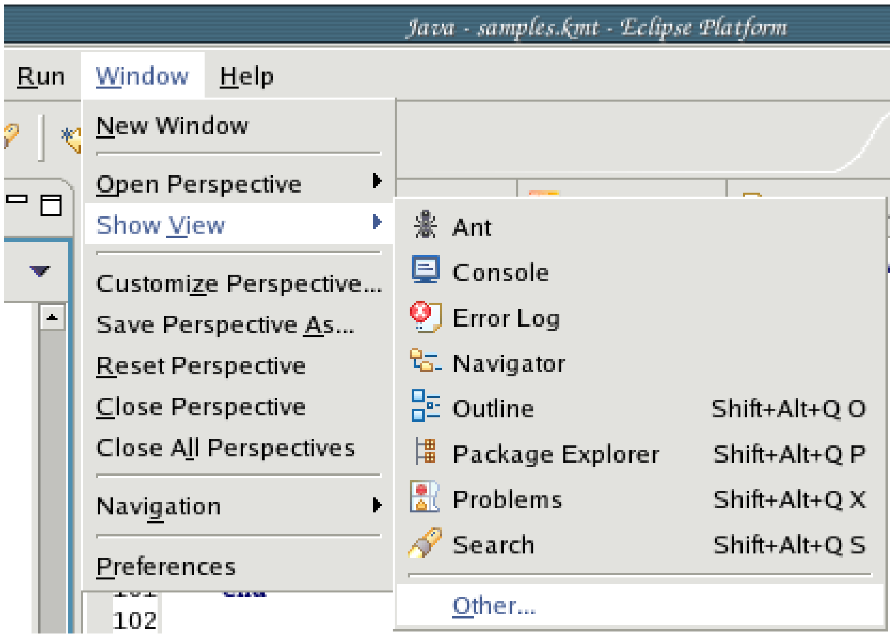


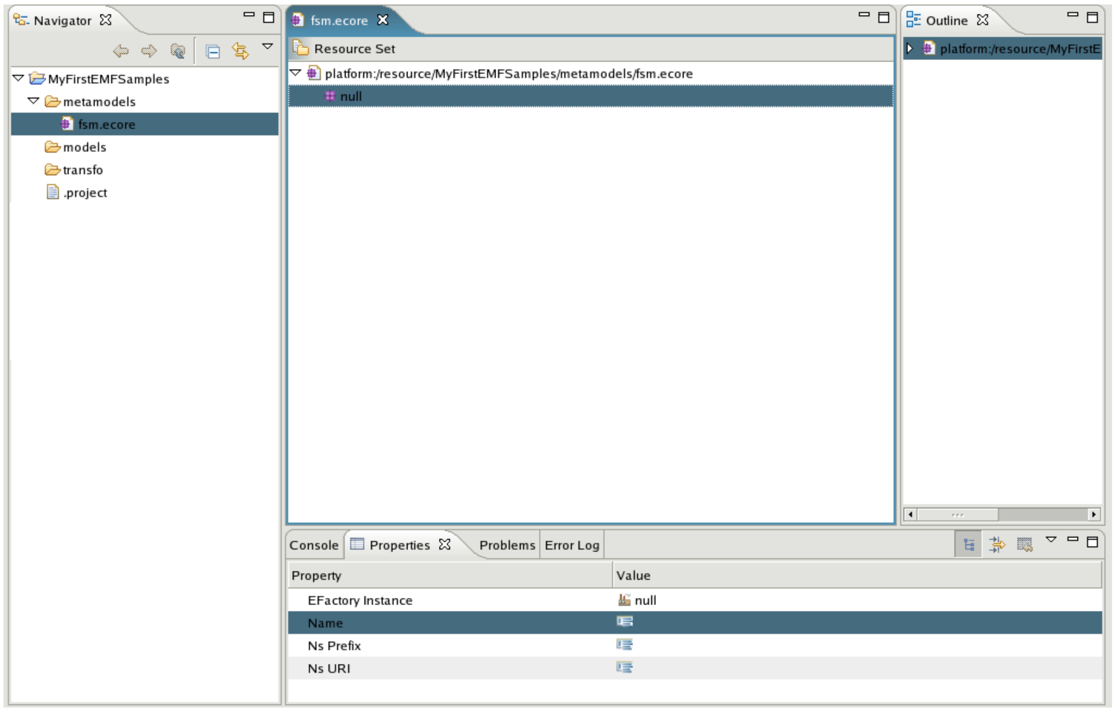
Figure 2.4. How to show properties view

8. Two properties of the created EPackage (which is initially displayed as *null*) have to be set through the **Properties** tab:

1.

a. its **Name**: *fsm*, for example;

b. its **Ns URI**: the namespace URI of the Ecore model is mandatory to allow Kermeta being able to correctly load its potential instances, as well as for the Dynamic creation of instances tool (see section 3.1 p.11). It is strongly recommended to set an absolute Eclipse URI (this kind of URI is actually relative to your Eclipse project), e.g. *platform:/resource/MyFirstEMFSamples/metamodels/fsm.ecore*



Property	Value
EFactory Instance	null
Name	
Ns Prefix	
Ns URI	

3. At this stage, it is now possible to add children to the created root (which is *fsm* EPackage in the considered example). This could be achieved by right-clicking on the class and choosing the **New child** item. In the scope of the FSM example, three classes are added, one of them (the class *Fsm*) being considered as the “root class” (this should not be not mandatory, but allows a better EMF working):

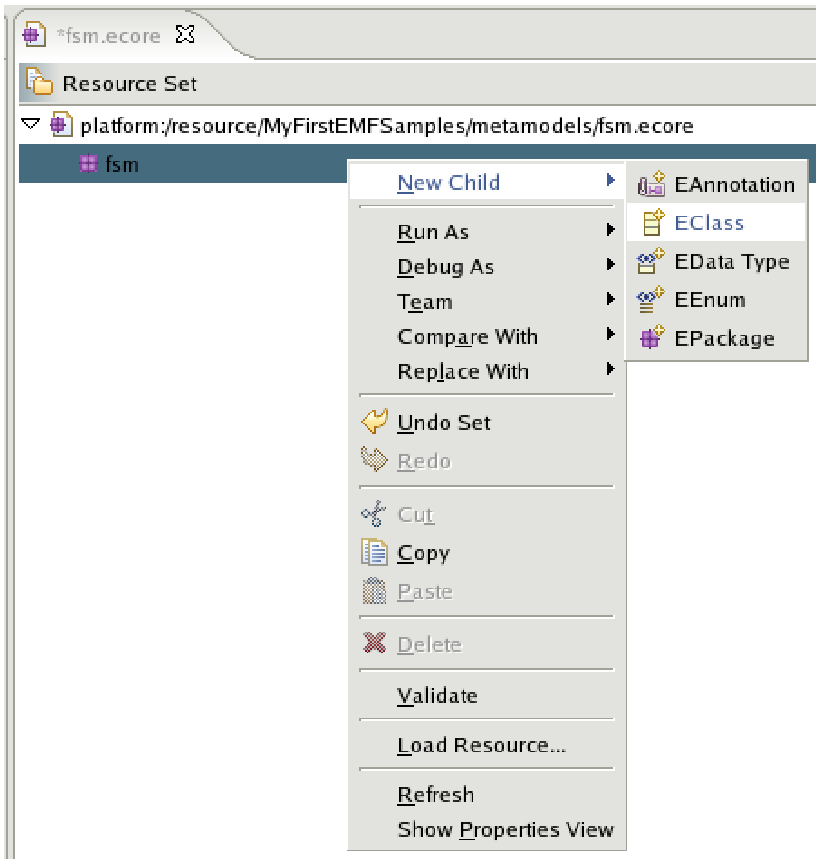


Figure 2.5. New child on EPackage root node

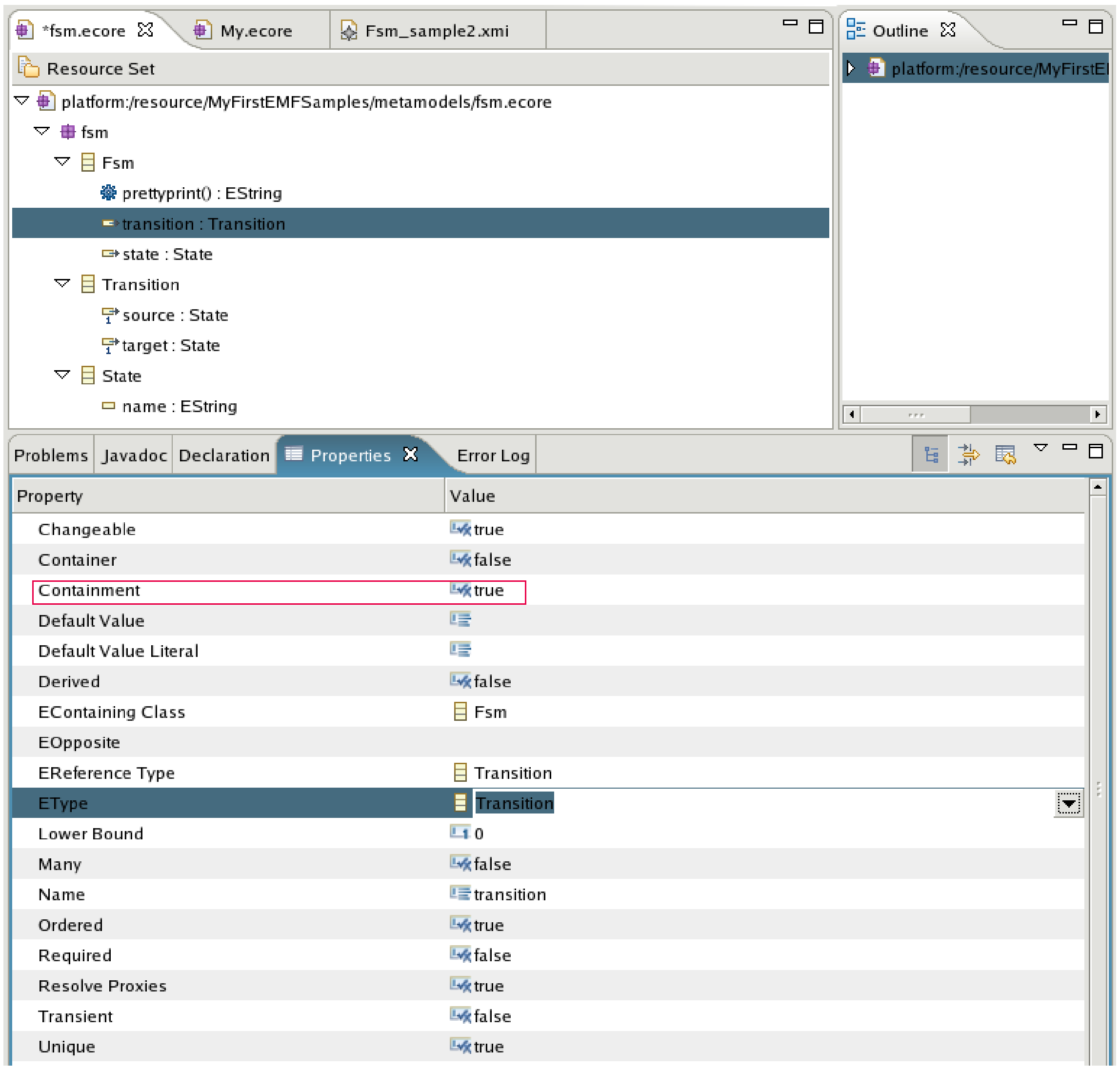
1.
   a. a *Fsm* class, for which the **Name** attribute (in the properties view) has to be set (other properties do not need to be considered at this stage);

   b. a *Transition* class (idem as for the *Fsm* class);

   c. a *State* class (idem as for previous classes).

4. Adding operations, attributes, or references to the newly created classes is achieved in the same way classes have been added to the root EPackage, i.e. using **New child** on each created element.

5. Still through the **Properties** tab, the **EContainment** property will have to be set to *true* for each reference which intended to contain instances. In the scope of the automaton example, the *transition* and *state* references of the *Fsm* class are defined as containment reference. By this mean, it will be possible to create, in an EMF model of it, a collection of transitions and states. Refer to the EMF model creation step (section 3, p.11) for further details.



```
Resource Set
  platform:/resource/MyFirstEMFSamples/metamodels/fsm.ecore
    fsm
      Fsm
        prettyprint() : EString
        transition : Transition
        state : State
      Transition
        source : State
        target : State
      State
        name : EString
```

Property	Value
Changeable	true
Container	false
Containment	true
Default Value	
Default Value Literal	
Derived	false
EContaining Class	Fsm
EOpposite	
EReference Type	Transition
EType	Transition
Lower Bound	0
Many	false
Name	transition
Ordered	true
Required	false
Resolve Proxies	true
Transient	false
Unique	true

Figure 2.6. Setting the properties of an EReference

6. Setting properties, such as the upper and lower bounds, the type (**EType**), of the attributes, operations, and references is achieved through the **Properties** view. The main properties to consider are: ESuperType, EType, Name, Upper Bound, Lower Bound, Containment (for the diamond-ed associations), Ordered, Unique, and EOpposite (opposite property). For simplicity purpose, other properties can be ignored in the scope of this tutorial.

   Lower and upper bound properties: 0, 1, -1 (stands for *) are allowed.

7. Do not forget to save the created meta-model.

2.2. Resulting meta-model

At this stage of the tutorial, the designed meta-model should look like the following meta-model:

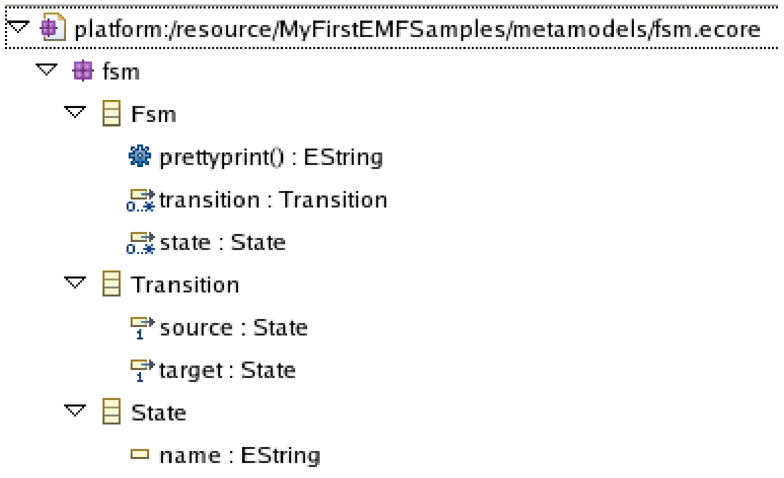

platform:/resource/MyFirstEMFSamples/metamodels/fsm.ecore
fsm
Fsm
prettyprint() : EString
transition : Transition
state : State
Transition
source : State
target : State
State
name : EString

Figure 2.7. A simple FSM meta-model

2.3. EMF meta-model creation tips

Creating a good meta-model is sometime difficult due to limitations of the used tools. For a better experience in using EMF tools and Kermeta, it is advised to respect the following rules:

1. Create an element that will contain directly or indirectly all the other elements. The reflexive editor and the editor generated by EMF allow to create only one root element and then, from this element, create contained elements.

   This problem occurs only for model element creation from the editors. The editors correctly display models from meta-models that do not follow this rule if you are able to create such models by another mean. Kermeta is **not** affected by this constraint.

2.4. External documentation

Additional documentation on EMF can be found at the following links:

1. www.eclipsecon.org/2005/presentations/EclipseCon2005_Tutorial28.pdf
2. http://dev.eclipse.org/viewcvs/indextools.cgi/emf-home/Attic/faq.html?rev=1.3
3. http://eclipse.org/emf/docs.php?doc=docs/tutorials/1.1/xlibmod/xlibmod_emf1.1.html

1. **http://www.eclipse.org/articles/Article-Using%20EMF/using-emf.html**

More generally, most of EMF documentation can be found onto the Eclipse website.

2.5. Alternative ways to create a meta-model

EMF tools are not the only way to create Ecore meta-models. Any tool that can manipulate Ecore can do the same. Here is a small list of tools that can be used to create your meta-model:

1. **Omondo / Eclipse UML has a nice graphical editor for Ecore models. It even enables to directly generate the EMF editor** from this tool.
2. **Kermeta allows users to specify meta-models by means of the Kermeta syntax and then translate them into Ecore using the Kermeta2Ecore function.**

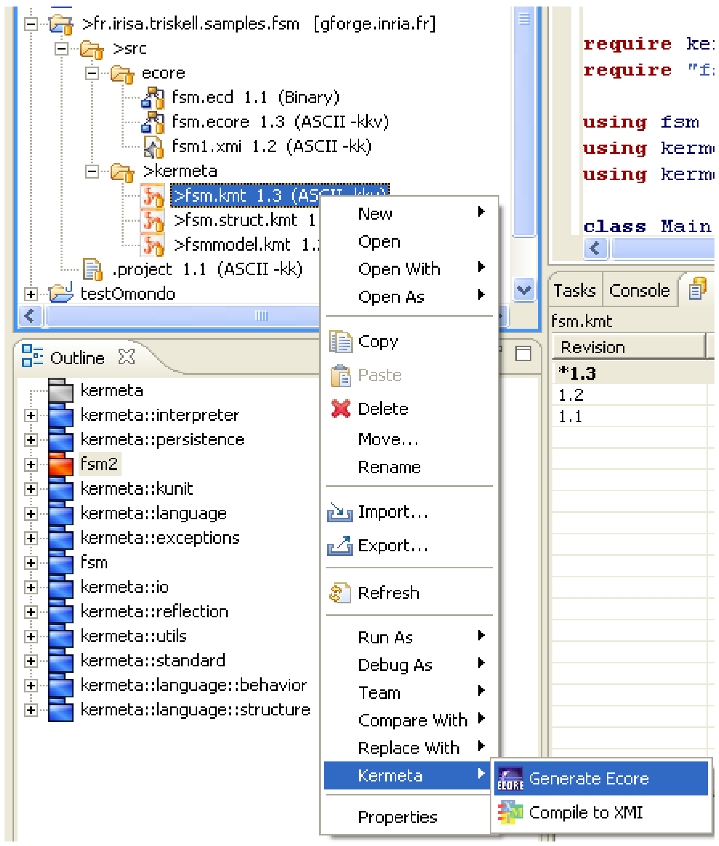


Figure 2.8. Generate Ecore meta-model from Kermeta

The Ecore import/export function and the Omondo editor can be used in order to graphically display the Kermeta classes of a model. It will act as a basic manual roundtrip editor.

CHAPTER 3

Create an instance-model from a meta-model

3.1. With the dynamic instance creation tool

This is the most simple way for creating an instance of a meta-model. Creation is accessible by right-clicking onto the root class/model object of the meta-model (in the scope of the FSM example, it is the *Fsm* class). Because of the containment property, classes are only available for creation through this root class.

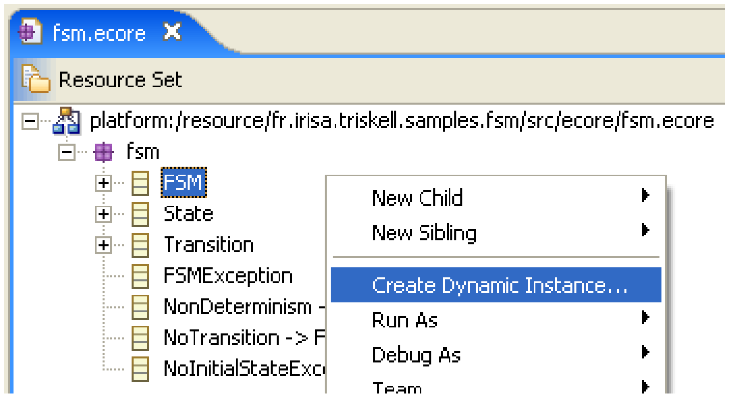


Figure 3.1. Create dynamic instance from an Ecore file

3.2. Using a reflexive editor

Since it allows to customize the generated editor to match meta-model specificities. (see the end of this article: http://www.eclipse.org/articles/Article-Using%20EMF/using-emf.html), this solution should be favoured for meta-models that are relatively stable.

3.2.1. 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. Once the meta-model is created, it is possible to create a model for model generation, called *genmodel*:

1.
   a. **File > New > Other > Eclipse Modeling Framework** folder **> EMF Models**;
   b. Choose a name for your genmodel (*fsm.genmodel* is ok);
   c. Select **Load from an EMF core model** button, and find the meta-model file (*fsm.ecore*);
   d. Select the unique package *fsm*.

5. 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, change the property called **Model Directory** (in the Model folder), to */MyFirstEMFSamples.model/src*, so that the EMF source code is generated in a new empty project that will exclusively contain this source code.
6. Right-click on the root node of the *fsm.genmodel*, and choose the **Generate all** item.

3.2.2. Use the editor

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

1.
   a. **Run > Run As > Run-time Workbench** (in Eclipse 3.0.2);
   b. or **Run > Run... > Eclipse application item in the right part > New** button **> Apply** button **> Run** button at the bottom (Eclipse 3.1).
   c. or **Run > Open Run Dialog... > Eclipse application item in the left part > New** button **> Apply** button **> Run** button at the bottom (Eclipse 3.3).

4. Once the new eclipse application is launched, create a new simple project (e.g. called *MyFirstEMFInstances*), and select **File > New > Other > Example EMF Creation Wizards** folder **> Fsm Model**;
5. Creation of an EMF model follows the same principles that the creation of an Ecore model. Please refer to the meta-model creation steps, section ???, p.???. Note that the model object, which was **EPackage** in the meta-model creation, becomes **Fsm** in the EMF model creation;
6. As you work in another Eclipse environment, you will probably want to copy the models that you created this way in your initial project (the one named *MyFirstEMFSamples*). Simply do it.

CHAPTER 4

Manipulate instance-models with Kermeta

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 4.5, p.15 and copy the given template.

4.1. Preparing a Kermeta program

4.1.1. The persistence library

The persistence library is inspired from the resource manager of EMF models. There is a repository (called *EMFRepository*), that is aimed at containing a set of resources (the *EMFResources*). Each resource contains a reference called *instances*, that contains all the root classes of the loaded model (there is usually only have one root class). So, the procedure of creation of a resource that will handle EMF models is the following (code example is provided in next sections):

1. Instanciate an EMFRepository;
2. Create a new EMF resource in this repository;
3. Load this resource;
4. Get the instances, i.e. the root class(es). All other instances can then be accessed by navigating the root class(es) properties.

4.1.2. A first Kermeta program using persistence

As a reminder, here is the common skeleton of a Kermeta program, inside which can must be added any library that is necessary to load and save the FSM samples.

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

package fsm_package;

require kermeta
require "../metamodels/fsm.ecore"
```

```
using kermeta::persistence // <- used to load and save EMF models.
using kermeta::standard

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

4.2. Load an EMF model with Kermeta

The following code sample load a previously created FSM model:

```
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.createResource("../models/Fsm_dyn_sample1.xmi", "../metamodels/fsm.ecore")
 resource.load

 // Load the fsm (we get the instance)
 fsm1 ?= resource.instances.one
 // Check that the fsm was correctly loaded
 fsm1.state.each { s | stdio.writeln("-> "+s.name) }
 fsm1.transition.each { t | stdio.writeln( t.source.name+ " -- " +t.target.name ) }
end
```

4.3. 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.state.add(newstate)
// save fsm1
resource.save()
```

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("../models/modified_dyn_sample1.xmi")
```

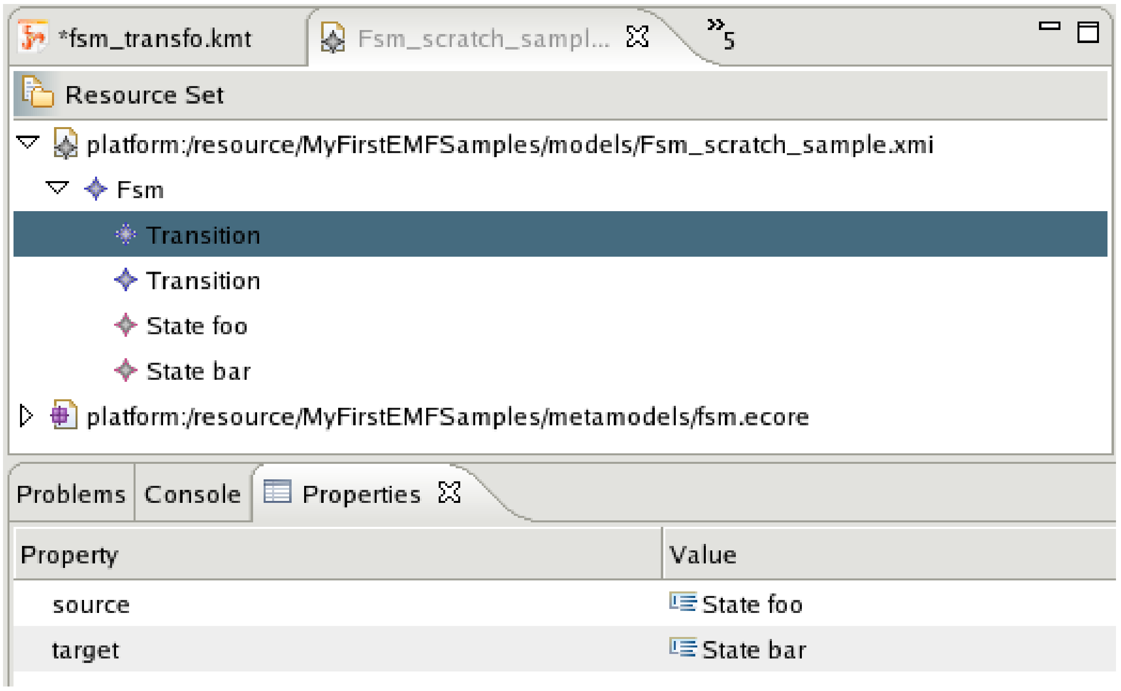
4.4. Create a model in Kermeta, and save it as an EMF mod-

el

Saving a programmatically generated model requires to use a new specific instruction that addq 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(
      "../models/Fsm_scratch_sample.xmi",
      "../metamodels/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.state.add(s0)
fsm2.state.add(s1)
fsm2.transition.add(t01)
fsm2.transition.add(t11)
// save the from-scratch model!
another_resource.instances.add(fsm2)
another_resource.save()
```

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



platform:/resource/MyFirstEMFSamples/models/Fsm_scratch_sample.xmi
Fsm
Transition
Transition
State foo
State bar
platform:/resource/MyFirstEMFSamples/metamodels/fsm.ecore

Property	Value
source	State foo
target	State bar

Figure 4.1. Fsm_scratch_sample view

4.5. 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 1.2, p.2), all the contained instances can be accessed.

```
@mainClass "fsm_package::Main"
@mainOperation "main"
package fsm_package;
require kermeta
require "<relative_path_of_the_metamodel>" // NOTE : same as param of createResource

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.createResource(
      "<relative_path_of_my_model_to_load>",
      "<relative_path_of_the_metamodel>")
      <my_resource>.load
      // Load the emf model - get the root class
      <my_model_object> ?= resource.instances.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
}
```

CHAPTER 5

Meta-model with behaviour and persistency

Kermeta makes it possible to add behaviour to a meta-model. Developers may therefore be interested in loading models that conform to such a meta-model. The following section explains one of the simplest way to achieve this goal. This is a very small variation on the code presented in the previous sections.

At current stage, this is based on Kermeta ability to “require” class definition written in several formats. Currently (version v0.0.16) supported formats are:

1. kmt files (Kermeta textual syntax);
2. km files (Kermeta model in XMI 2.0);
3. Ecore files (Ecore model in XMI 2.0);
4. emfatic files (Ecore textual syntax developed by IBM).

The transformations Kermeta2Ecore and Ecore2Kermeta enables to obtain two versions of a same meta-model, one in Ecore, the other in Kermeta. From a structural point of view, they will be equivalent and compatible in Kermeta.

Then, these two syntaxes are structurally equivalent.

```
require "fsm.kmt" // if you use this one, you'll have the FSM behavior
```

is equivalent to

```
require "fsm.ecore" // if you use this one, you won't be able to use the FSM behavior
```

If you have generated the Ecore from a kmt that defines a behaviour, this Ecore will also contain the specified behaviour.

Model loading and saving must be achieved using the Ecore version, so that EMF will know how to serialize/deserialize the models.

```
var my_resource : EMFResource init repository.createResource( "./my_fsm_usermodel.fsm", "./fsm.ecore")
```

The Kermeta version of the meta-model will be used to specify the behaviour, the Ecore version will be used by EMF for the persistency aspects.

Note

Kermeta2Ecore and Ecore2Kermeta transformations are available in the workbench by respectively right-clicking onto a Kermeta or an Ecore file).