



Tutorial : Building a DSL using Kermeta

Logo sample

Haja Rambelontsalama

Abstract

This tutorial is a step-by-step user-guide to build integrated tools with the Kermeta language. The illustrated example here is building the Logo DSL with Kermeta.

Published Build date: 3-November-2010
Revision: \$Date:: 2010-03-05 11:10:01#\$



Preface	vi
Chapter 1. Introduction	1
Chapter 2. Installation	2
2.1. Prerequisites	2
2.2. Install logo projects	2
Chapter 3. Define the metamodel	4
3.1. From Ecore to generated code	4
3.2. Metamodel Ecore with Kermeta	7
Chapter 4. Editor	11
4.1. Dynamic instance	11
4.2. Tree view editor	12
4.3. Textual editor	14
4.4. GMF editor	14
Chapter 5. Model manipulation in Kermeta	15
5.1. Package registry	15
5.2. Serialization	15
Chapter 6. Contract	16
6.1. Static semantics	16
6.2. Implementation in Kermeta	16
Chapter 7. Behaviour	19
7.1. Preparation of the environment	19
7.2. Dynamic semantic	20
7.3. Virtual machine	22
7.4. Operational semantics extra	23
7.4.1. Call to java.lang.Math	23
7.4.2. Implementing a graphical output	24

Chapter 8. Model transformation - Compilation	27
Chapter 9. UI improvements	28
9.1. GUI deployment setup	28
Chapter 10. Using the deployed DSL environment	32
Chapter 11. Conclusion	34

List of Figures

2.1. Installation of the Kermeta Tutorial projects	3
3.1. Project structure after generating the model, edit and editor code	6
3.2. The logo metamodel	7
4.1. Creating a dynamic instance (illustration purpose)	12
4.2. dots.xmi illustration	13
6.1. Kermeta application checkModels output	18
7.1. org.kermeta.kmlogo.logoasm.srcKermeta project structure	20
7.2. Outline view	22
7.3. Call to java Math results	24
7.4. VM testing output	25
11.1. Logo solution project structure.	34

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.

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: v1.2.0.

Tip

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

Introduction

The Logo language is a small programming language to manage a pen-drawer turtle. Wherever the turtle goes, it draw lines to form figures. Logo was initially dedicated to introduce programming concepts to children. You can find more information about logo language on the wikipedia page ([external link](#)).

This tutorial will guide you from the installation of the necessary files to the analysis of the different part of the code so as to build a programming environnement for the Logo.

Note

We **recommend** that you **firstly** read the [10min-reading Building DSL main process](#) tutorial in order to get the "**bird's eye view**" (see Introduction) of what we are going to build here and to have an overview of the **methodology** to follow.

Installation

In this task you'll learn how to install the predefined logo environment, install the predefined logo project for the tutorial.

This tutorial was set up with Eclipse 3.5.1, Kermeta 1.3.2, EMFText 1.2.3

(**Update** : this tutorial was updated using Eclipse 3.6.1, Kermeta 1.4.1, EMFtext 1.3.2)

2.1. Prerequisites

Below is the list of the things that you need to ensure beforehand :

- You should install an eclipse environment with Kermeta (1.3.3 and above). Further information on how to install Kermeta and the Kermeta language;
- You should be familiar with ecore metamodel;
- You should be familiar with eclipse modelling environment.
- You should have notion of aspect oriented programming (AOP).

2.2. Install logo projects

Once you have an Eclipse modelling with Kermeta, get the tutorial projects by selecting **File -> New -> Example... -> Kermeta Samples -> Logo tutorial - base resources**.

At this point, you should have one project inside your workspace as illustrated in the figure below :

- a project named : **org.kermeta.kmlogo.logoasm.tutorial** with a readme and a folder "parts". Inside the folder "parts" are the folders corresponding to each section of this tutorial and containing the needed material for each step.

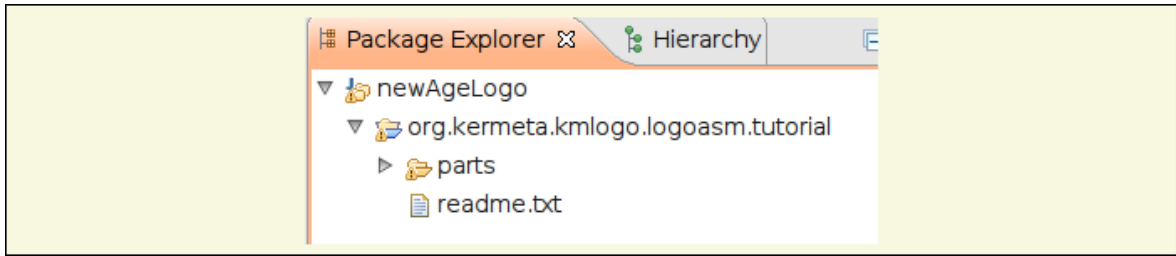


Figure 2.1. Installation of the Kermeta Tutorial projects

Note

If you want to skip this step by step tutorial and directly jump to the whole soluce code, it is already deployed into the Kermeta Eclipse. Please read the conclusion to see how to.

Define the metamodel

The first step of this tutorial is to define the abstract syntax. To do so, we have to define the main concepts. For the logo language, these concepts are the instructions to be executed : pen up, pen down, go forward, rotate left ...

Tip

If you want to learn more about how to create an ecore metamodel with eclipse, please follow the how-to in the next sections or refer to the other available tutorial

3.1. From Ecore to generated code

In order to define the metamodel, we are going to process as shown in the following steps : create an ecore file, set up its nsURI and create its corresponding generator model (.genmodel). Then, we are going to set up the needed informations (base package, model directory, file extension, plugin ID,...) :

1. Inside the project **org.kermeta.kmlogo.logoasm.tutorial**, open the folder **parts/1.metamodel** and open the file **ASMLogo.ecore** which is the metamodel of your language (here logo) and look at its content. In this ecore file you'll notice the various instructions supported by the DSL that we are going to build. Some are primitives, for example : Back, Forward, Left, PenUp, ... Some are expressions, for example : Plus, Minus, ... And some are control structures, for example : If, while, Repeat, ...
2. If not already set in the property view, do not forget to set the properties of our metamodel like in this example :

1. Inside the file **ASMLogo.ecore**;
2. Select **kmLogo** package, right-click on it and choose **show properties view**;
3. Set the **ns URI** to "http://www.kermeta.org/kmLogo" and save it;

Example 3.1. Set up nsURI

3. Then, we need to create the generator model for this ecore. For those who are familiar with creating a genmodel, we've provided a completed one inside the folder **parts/1.metamodel** (see the file **ASMLogo.genmodel**) so that you can rapidly jump to the next step. For the beginners, below is a step

by step guide to create one.

create an **EMF Generator Model** (File > New..) and name it **ASMLogo.genmodel**, hit next and choose **Ecore model** as model importers, hit next and import the previous metamodel (**Browse workspace > org.kermeta.kmlogo.logoasm.tutorial/1.metamodel**), **load** it, hit next and finish. In the example below, you can set up the properties for code generation, if needed you can manually tweak them but normally you should leave them as they are auto-generated (you should result in a file with the same informations as the provided **ASMLogo.genmodel**).

1. Open the file **ASMLogo.genmodel**;
2. Unfold the root element **ASMLogo** and select the **kmLogo** package;
3. Right-click on it and choose **show properties View** (if not already open), set the **base package** (unfold All in property view) property to : **org.kermeta.kmlogo.logoasm.model**;
4. For the **ASM package** (unfold kmLogo package) set its **File Extension** (unfold Model in the property view) property to "logoasm";
5. Right-click on the root element **ASMLogo** and choose **show properties View** (if not already open);
6. In the **property view**, unfold **Model**:
 - Set "Model Directory" to **/org.kermeta.kmlogo.logoasm.model/src**
 - Set "Model Plug-in ID" to **org.kermeta.kmlogo.logoasm.model**
7. In the **property view**, unfold **Edit**:
 - Set "Edit Directory" to **/org.kermeta.kmlogo.logoasm.edit/src**
 - Set "Edit Plug-in Class" to **org.kermeta.kmlogo.logoasm.kmLogo.provider.ASMLogoEditPlugin**
 - Set "Edit plug-in ID" to **org.kermeta.kmlogo.logoasm.edit**
8. In the **property view**, unfold **Editor**:
 - Set "Editor Directory" to **/org.kermeta.kmlogo.logoasm.editor/src**
 - Set "Editor Plug-in Class" to **org.kermeta.kmlogo.logoasm.kmLogo.presentation.ASMLogoEditorPlugin**
 - Set "Editor plug-in ID" to **org.kermeta.kmlogo.logoasm.editor**

Example 3.2. Manually set properties of a .genmodel

4. If you have manually set up the properties, do not forget to save your generator model. Let's directly generate the associated code for our metamodel. Remember the main concepts of our DSL that we've mentioned before, this is step is mandatory as it gives a first implementation of our DSL with EMF :

Inside the **ASMLogo.genmodel**, right-click on kmLogo package and choose **Generate Model Code**. This will add a new project **org.kermeta.kmlogo.logoasm.model** to your structure.

Repeat the step above and choose **Generate Edit code**. This will add a new project **org.kermeta.kmlogo.logoasm.model.edit** to your structure.

Repeat again and choose **Generate Editor Code**. This will add the project **org.kermeta.kmlogo.logoasm.model.editor**.

Copy the files **ASMLogo.ecore** and **ASMLogo.genmodel** inside a folder **org.kermeta.kmlogo.logoasm.model/model** in order to be coherent with the plugin intention (store the model and the model code)

You should now have a project structure as illustrated below :

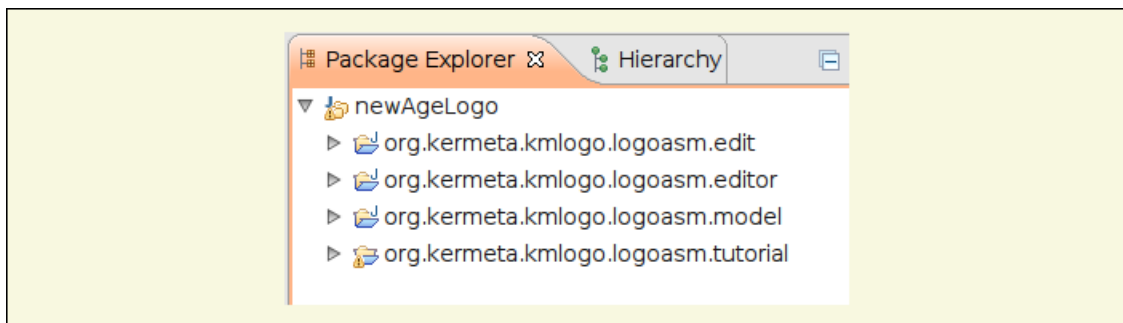


Figure 3.1. Project structure after generating the model, edit and editor code

5. To have an overview of our DSL, go back to the file **org.kermeta.kmlogo.logoasm.model/model/ASMLogo.ecore**:

Right-click on it and initialize the.ecore diagram file.

Name it **ASMLogo.ecorediag**, hit next and choose **ASM** package as a root element and finish to see the graphical representation of your metamodel (if you choose **kmLogo** as a root element then you will have to double-click on **Package ASM** to open it and then **Create** button from the window wizard).

You should obtain an overview of all the concepts in your metamodel (see the figure below -it may differ from actual version but is here for illustration purpose).

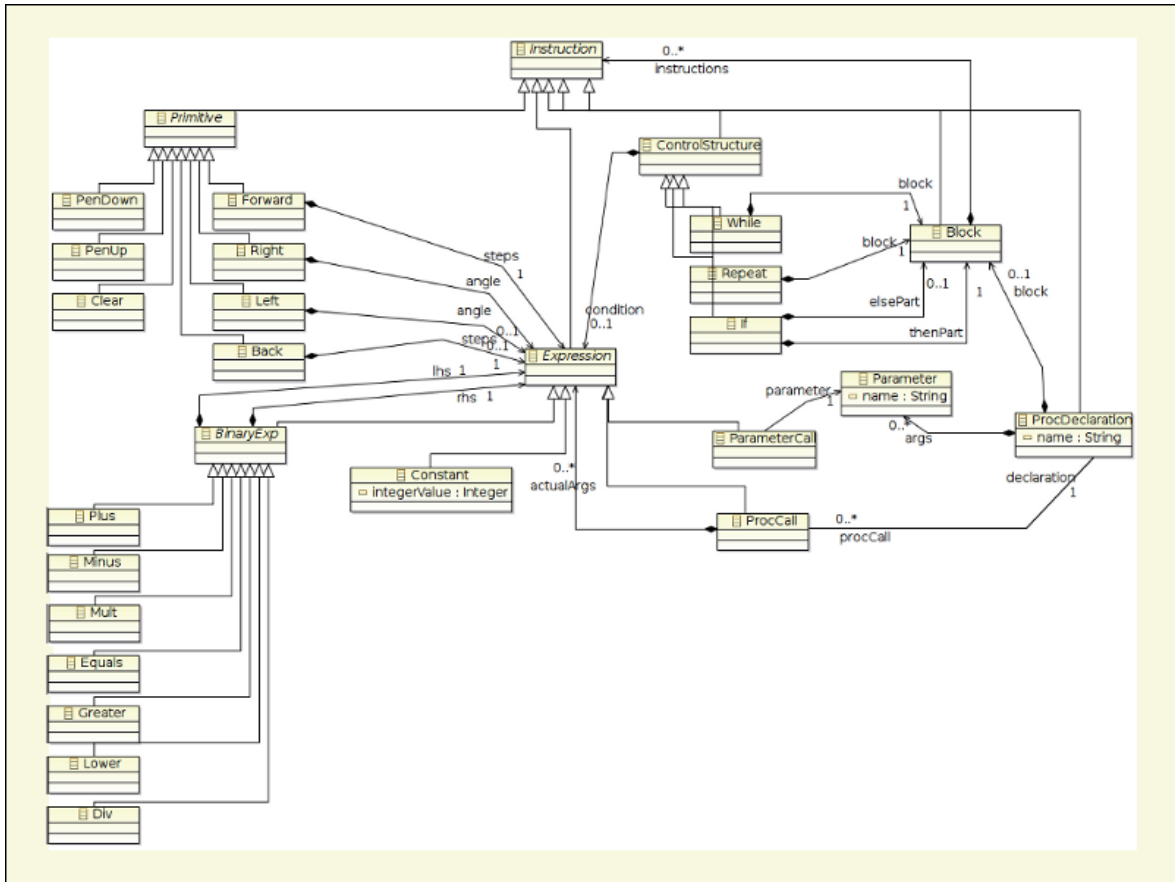


Figure 3.2. The logo metamodel

Important

Before going further in this tutorial, do not forget (if not already done) :

- To set up the nsURI of the **model/ASMLogo.ecore** file;
- To register this.ecore model (In deployed mode (ie. in a runtime workbench), the.ecore is automatically registered by the plugin, in development mode, you need to manually register it).

3.2. Metamodel Ecore with Kermeta

Many.ecore tools allow you to create your metamodel. Kermeta allows you to do it in a "programmatical" way. Analyse the content of the listing below and see what does it stand for :

```
@uri "http://www.kermeta.org/kmLogo"
package kmLogo;

require "kermeta"
```

```
alias Integer : kermeta::standard::Integer;
alias Boolean : kermeta::standard::Boolean;
alias String : kermeta::standard::String;
package ASM
{
  abstract class Instruction
  {
  }
  abstract class Primitive inherits Instruction
  {
  }
  class Back inherits Primitive
  {
    attribute steps : Expression[1..1]
  }
  class Forward inherits Primitive
  {
    attribute steps : Expression[1..1]
  }
  class Left inherits Primitive
  {
    attribute angle : Expression
  }
  class Right inherits Primitive
  {
    attribute angle : Expression
  }
  class PenDown inherits Primitive
  {
  }
  class PenUp inherits Primitive
  {
  }
  class Clear inherits Primitive
  {
  }
  abstract class Expression inherits Instruction
  {
  }
  abstract class BinaryExp inherits Expression
  {
    attribute lhs : Expression[1..1]
    attribute rhs : Expression[1..1]
  }
  class Constant inherits Expression
  {
    attribute integerValue : Integer
  }
  class ProcCall inherits Expression
  {
    attribute actualArgs : Expression[0..*]
    reference declaration : ProcDeclaration[1..1]#procCall
  }
  class ProcDeclaration inherits Instruction
  {
    attribute name : String
    attribute args : Parameter[0..*]
    attribute block : Block
    reference procCall : ProcCall[0..*]#declaration
  }
}
```

```
}
class Block inherits Instruction
{
    attribute instructions : Instruction[0..*]
}
class If inherits ControlStructure
{
    attribute thenPart : Block[1..1]
    attribute elsePart : Block
}
class ControlStructure inherits Instruction
{
    attribute condition : Expression
}
class Repeat inherits ControlStructure
{
    attribute block : Block[1..1]
}
class While inherits ControlStructure
{
    attribute block : Block[1..1]
}
class Parameter
{
    attribute name : String
}
class ParameterCall inherits Expression
{
    reference parameter : Parameter[1..1]
}
class Plus inherits BinaryExp
{
}
class Minus inherits BinaryExp
{
}
class Mult inherits BinaryExp
{
}
class Div inherits BinaryExp
{
}
class Equals inherits BinaryExp
{
}
class Greater inherits BinaryExp
{
}
class Lower inherits BinaryExp
{
}
class LogoProgram
{
    attribute instructions : Instruction[0..*]
}
}
```

Inside **org.kermeta.kmlogo.logoasm.model/model**, create a new folder **srcKermeta**.

Inside this newly created folder, create a new Kermeta file (**.kmt**) and type the listing above inside (to save time, after analysing the listing above, you may copy/paste to replace the generated code).

After saving, right-click on it and choose **Kermeta > Generate Ecore**.

You should retrieve our logo language's .ecore representation that we 've just seen before. Congratulation you've just "written" your first kermeta program of this tutorial, and you have define the famous main concepts of your DSL!

Editor

In this task, you will learn how to provide editors to your DSL in order to manipulate **model instances** conform to your DSL's metamodel. There are many ways to do so and as these techniques are independent from each other, you can experiment each of them according to your needs. However in this tutorial we will focus more on the textual editor. Advanced users (i.e.: those familiar with creating .xmi files from scratch) can directly jump to the texteditor generation section using dedicated framework.

4.1. Dynamic instance

Before all, let's create our first model instance with the dynamic instance :

1. Open the file **ASMLogo.ecorediag** inside the folder **org.kermeta.kmlogo.logoasm.model/model**.
2. Go to the **outline view**. Select the top level element inside your metamodel (for example here the Logo-Program as it is the entry point).
3. Right-click on this element and choose **Create a dynamic instance**.

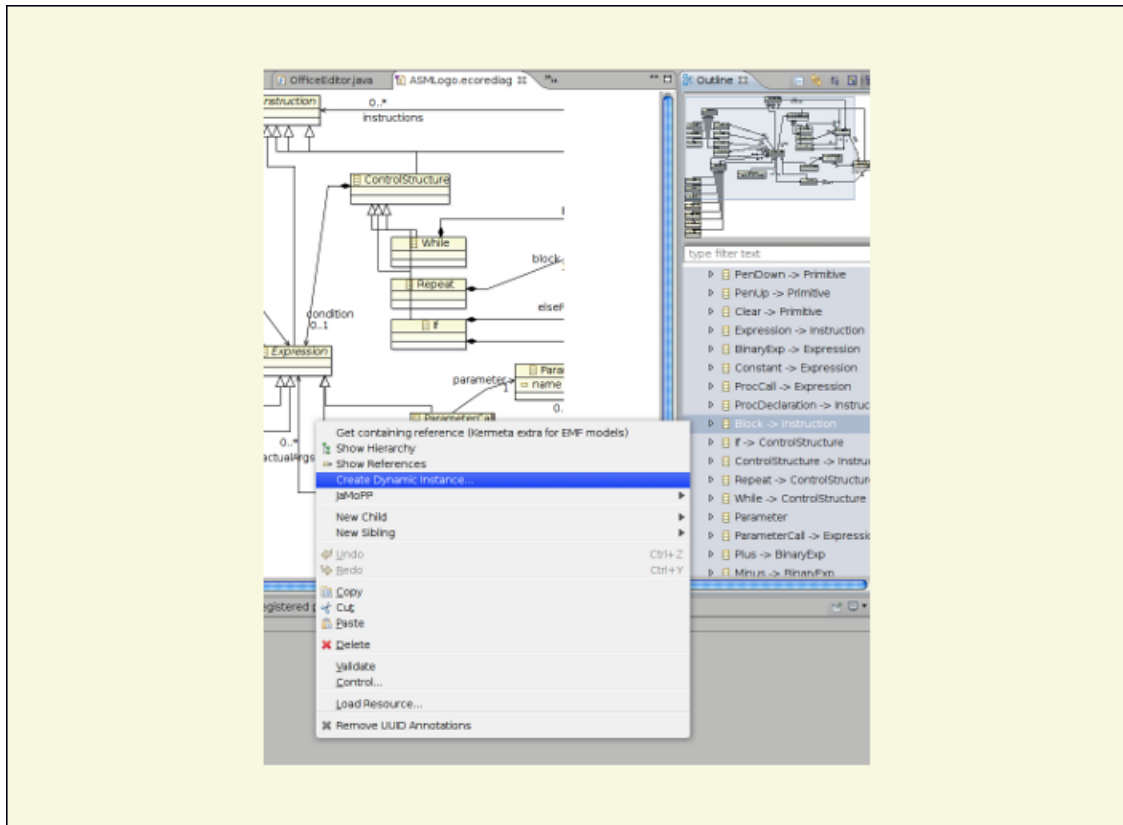


Figure 4.1. Creating a dynamic instance (illustration purpose)

Name your dynamic model "dots.xml" and hit finish.

4.2. Tree view editor

Using the tree-view editor is a way to manipulate model instances. Once you open the created ".xml" file with the "Sample Reflective Ecore Model Editor" then you can add childs or sibling for the element to construct you model instance. At the end, you can obtain a model conform to the logo language in xml format.

If you want to learn more, follow the complete example below or check out the other available examples

1. Right-click on the file "dots.xml" and choose "open with .. > Sample Reflective Ecore Model Editor".
2. In the editor, unfold "...dots.xml" and right-click on Block element. Choose **New child > Instructions Right** to position the turtle.
3. Then, right-click on "Instruction Right" and choose **New child > Angle Constant** to define the angle of the rotation. Select the element "Constant" and in the properties view (right clic show property view), set

the **Integer Value** to 90.

4. We set the first instruction. In the editor, right-click and choose **Validate** to check whether your model instance newly created is conform to the defined metamodel.
5. Then you can keep on building your model instance : Right-click on "Right" element, choose **New Sibling > Instructions penDown** to tell the turtle to be "ready to write".
6. After that, Right-click on "PenDown" and choose **New Sibling > Instructions Forward** then like we did before, create the **steps Constant** child and set the value to 10. That tell the turtle to "draw" the line on 10 of the given metric (pixel).
7. Repeat above steps to complete your model instance like the illustration below and do not forget to **Validate** to check conformance from time to time.

Example 4.1. Creation of the dots.xmi

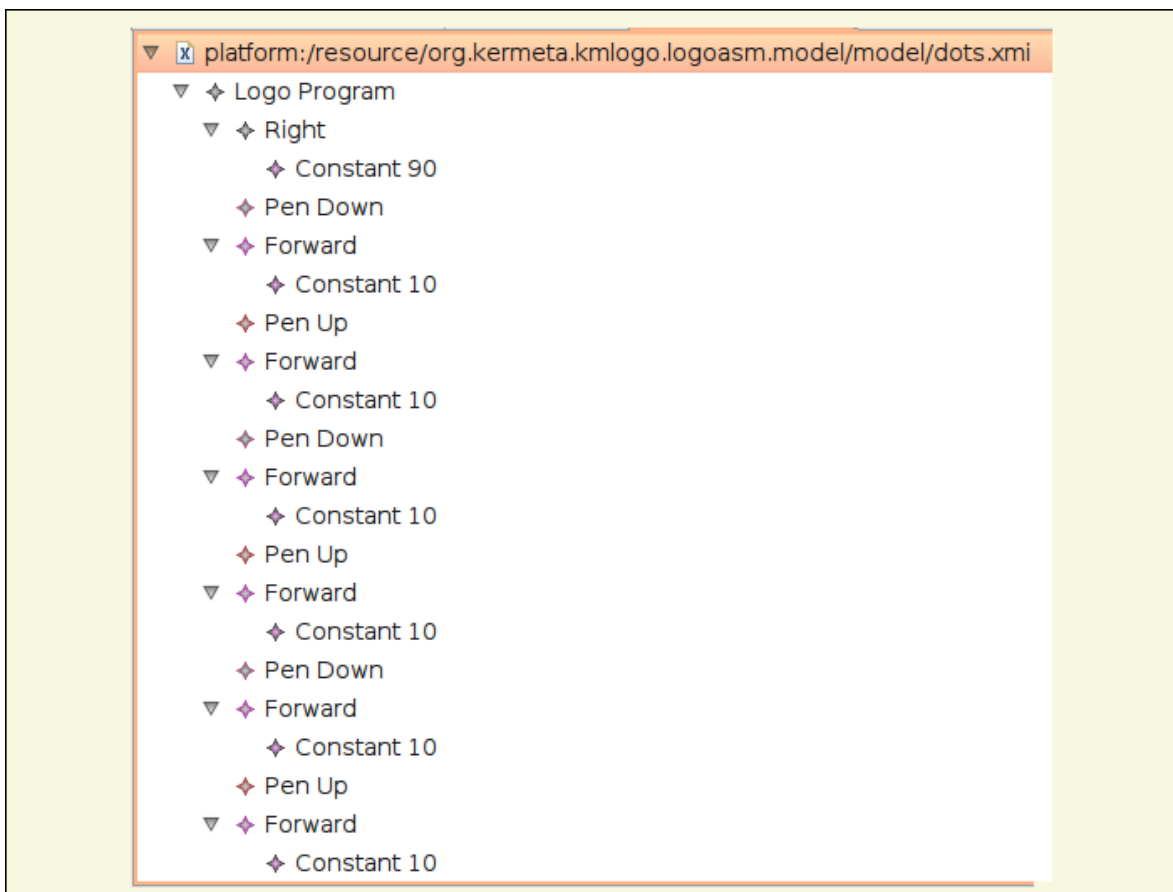


Figure 4.2. dots.xmi illustration

You have just finished creating your first model instance, save it and keep an eye on it as we are going to re-use it later in this tutorial.

4.3. Textual editor

A textual editor is also available in order to create a model instance conform to our metamodel and can be processed by the interpreter. In this example we will use EMFText (external link) to generate the concrete syntax of our metamodel.

Follow these steps to set up the concrete syntax of your language and its associated textual editor :

1. Create a java project and name it : **org.kermeta.kmlogo.logoasm.emftexteditor**. Create a folder **concrete_syntax** inside.
2. Back to the project **org.kermeta.kmlogo.logoasm.tutorial**, open the folder **1.metamodel/concrete_syntax**.
3. Copy and paste the file **ASMLogo.cs** into the newly created folder **org.kermeta.kmlogo.logoasm.emftexteditor/concrete_syntax/**.

Right-click on the file **ASMLogo.cs** and choose "Open With ..." > "EMFText cs editor". Analyse its content : it follows the structure of .cs file as described in the emftext CS language reference (external link).

Here, we defined the file extension of our DSL (logo), link it to our metamodel (through the nsUri) and define the style and the concrete rules for each meta-class concept that we want to map.

Tip

If you meet with some errors, be sure to register your ecore and ensure that the path to your gen-model is correct.

4. Right-click on **ASMLogo.cs** file and choose **generate Text Ressource**.

This would add two project in your project structure : **org.emftext.common.antlr3_1_1** and **org.kermeta.kmlogo.logoasm.model.kmLogo.resource.logo**.

(**Update** : Since EMFtext 1.3, this would add three projects in your project structure : **org.emftext.common.antlr3_2_0**, **org.kermeta.kmlogo.logoasm.model.kmLogo.resource.logo** (core resource for texteditor) and **org.kermeta.kmlogo.logoasm.model.kmLogo.resource.logo.ui** (eclipse ui interaction).)

You have just generated the code for your text editor in "src" and "src-gen" of each project. Keep going through this tutorial to test it inside a real interpreter.

4.4. GMF editor

The graphical way to manipulate your metamodel can be done with the GMF editor.

For further information, please refer to the FSM tutorial on how to graphically edit your metamodel.

Model manipulation in Kermeta

These are few steps on how to manipulate metamodel in Kermeta. This is about why to register your metamodel and how to load/save it. We invite the reader to check inside the **Building DSL main process** tutorial (See Building DSL with kermeta tutorials) for the explanation of these model manipulations.

5.1. Package registry

An example is fully covered in section 5.2 of FSM tutorial, for further explanation of this step please refer to the section 4.2 of process tutorial

5.2. Serialization

The section 5.1 of FSM tutorial provides a complete example for this section, explanation can be found inside the section 4.1 of process tutorial

Contract

Let's now add some **static semantic** to our metamodel i.e. add missing pre/post-conditions to our .ecore in order to express the constraints specified in the metamodel. From now on, we are going to learn how to weave aspects with Kermeta.

6.1. Static semantics

For this step we are going to add two contracts to our metamodel:

- "no two formal parameters of a procedure may have the same name";
- "each procedure call provide the same number of arguments as specified in its declaration".

In OCL you will have the listing below to express it (you can find the corresponding OCL file inside **org.kermeta.kmlogo.logoasm.tutorial/parts/2.contracts**) :

```
package kmLogo::ASM

context ProcCall
inv appropriate_number_of_actual_parameters :
    actualArgs->size() = declaration.args->size()

context ProcDeclaration
inv unique_names_for_formal_arguments :
    args->forAll ( a1 | args->forAll ( a2 | a1.name = a2.name implies a1 = a2 ))

endpackage
```

6.2. Implementation in Kermeta

Kermeta offers same mechanism as OCL to navigate inside the elements (<Collection>.each, <Collection>.forAll, ...). Through aspect weaving, Kermeta allow to reopen a class and weave an aspect inside in order to add this pre/post conditions. Let's implement the example above in Kermeta :

1. Create a new plug-in project **org.kermeta.kmlogo.logoasm.srcKermeta** (set all by default, hit next and finish, if you're not familiar with plug-in perspective choose "no" if asked).

Create a folder **kermeta** on the root of this project. This is the folder where we are going to put all kermeta sources for this tutorial.

2. Inside the folder **kermeta**, create a folder **2.Constraints**.

Inside the folder **2.Constraints**, create a new kermeta file **ASMLoStaticSemantics.kmt**. Analyse the content of the following listing and then copy/paste into the newly created file :

```
package kmLogo::ASM;
require kermeta
require "http://www.kermeta.org/kmLogo"

aspect class ProcDeclaration{
  /**
   * No two formal parameters of a procedure may have the same name
   */
  inv unique_names_for_formal_arguments is
  do
    args.forAll{ a1 | args.forAll{ a2 |
      a1.name.equals(a2.name).implies(a1.equals(a2))}
    }
  end
}

aspect class ProcCall{
  /**
   * A procedure is called with the same number of arguments as specified in its declaration
   */
  inv same_number_of_formals_and_actuais is do
    actualArgs.size == declaration.args.size
  end
}
```

It reopen the ProcDeclaration and ProcCall classes and weave invariants as aspects into it

Note

If you get errors on "http://www.kermeta.org/kmLogo", check your package registration. you may hit the **check this file** button (on Kermeta's perspective toolbar) to refresh.

3. After saving it, let's see the results.

To do so go back to the project **org.kermeta.kmlogo.logoasm.tutorial** for a while.

Copy the folders:

- **org.kermeta.kmlogo.logoasm.tutorial/parts/2.contracts/models** which contains the models conform to our ecore metamodel;
- **org.kermeta.kmlogo.logoasm.tutorial/parts/2.contracts/tests** which contains the main operation for testing the constraints on these models.

Paste these folders and their content into **org.kermeta.kmlogo.logoasm.srcKermeta/kermeta/2.Constraints**.

4. Open the file **tests/CheckModels.kmt**, remove the two comment blocs inside and analyse its purpose.

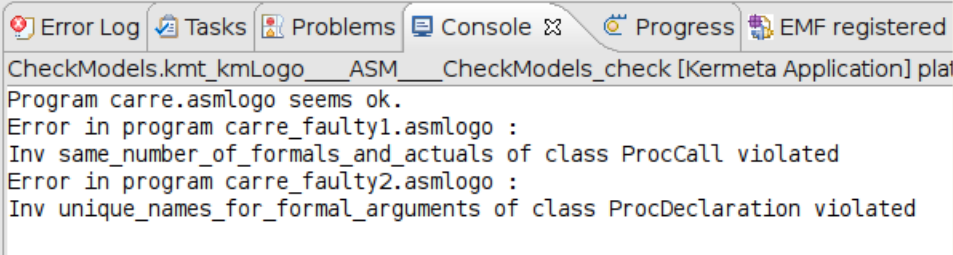
It loads the models inside the models folder and require the kermeta file that we've created before.

5. Let's now run our first Kermeta application : Right-click on the file **CheckModels.kmt** and choose **Run as > Run as Kermeta application**.

Note

If you get errors, again, check that you did not forget to register you ecore. Maybe it is necessary to clean all the projects or launch the kermeta cleaner (trash bin with kermeta icon on your toolbar) or hit the kermeta validator ("check this file" button in Kermeta perspective toolbar)

On your console view, you should view the result below though the model instances are conform to our metamodel (you can right-click and "validate" to see its conformance to the ecore) :



```
CheckModels.kmt_kmLogo__ASM__CheckModels_check [Kermeta Application] platform
Program carre.asmlogo seems ok.
Error in program carre_faulty1.asmlogo :
Inv same_number_of_formals_and_actuals of class ProcCall violated
Error in program carre_faulty2.asmlogo :
Inv unique_names_for_formal_arguments of class ProcDeclaration violated
```

Figure 6.1. Kermeta application checkModels output

Important

Remember : To check the constraints in Kermeta, you should call the method **checkAllInvariants()** on root element of the model (here the one from **models/carre.logoasm**)

Behaviour

Let us now add some **operational semantics** to our metamodel. To do so, we are going to add the actions we wish to implement thanks to the weaving mechanism in Kermeta that we've just seen before.

7.1. Preparation of the environment

First of all let's prepare the appropriate programming environment needed for this implementation. **org.kermeta.kmlogo.logoasm.srcKermeta** will be the concerned project during this chapter. We'll setup a graphical UI using *java AWT* and a *Java Math* wrapper in order to add an UI to our DSL and add some math operations to Kermeta.

1. Copy/paste the folders and their content from **org.kermeta.kmlogo.logoasm.tutorial/parts/3.behaviour/...** into **org.kermeta.kmlogo.logoasm.srcKermeta/kermeta/** i.e.:

- **1.Models** where are stored the models we are going to simulate;
- **3.JavaInterfaces** where we can see the needed java program for this tutorial;
- **4.VirtualMachine** where is defined the application domain;
- **5.Simulator** where the execution program is provided.

2. Inside the **org.kermeta.kmlogo.logoasm.srcKermeta/src** create two packages **org.kermeta.kmLogo.gui** and **org.kermeta.kmLogo.wrapper**.

Move the files from :

- **3.JavaInterfaces/gui/** into the package **org.kermeta.kmLogo.gui**;
- **3.JavaInterfaces/wrapper/** into the package **org.kermeta.kmLogowrapper** (as they are now empty, you may delete the folder **3.JavaInterfaces/gui/** and **3.JavaInterfaces/wrapper/** after).

- 3.

Caution

At this point, you should have errors on your wrapper package. It is normal, we haven't set up the

plugin yet. Though these next steps are not really part of Kermeta processing, we have to fix these errors to keep on.

Replace the file **META-INF/MANIFEST.MF** by the one you can find in **3.JavaInterfaces/plugin**. The manifest tell the project about its configuration (dependencies, exported package, runtime, ...). Have a look at its content and see how to solve the problem (If errors still persist, continue next step).

Copy the file **plugin.xml** in **3.JavaInterfaces/plugin** and paste it on the root of your project. This is the file which handle the plug-in nature of the project. Observe its content (extension, ..). (you may delete the folder **3.JavaInterfaces/plugin** and the package **src/org.kermeta.kmlogo.logoasm.srcKermeta** since we don't use it).

4. At this point your project **org.kermeta.kmlogo.logoasm.srcKermeta** should have no error and present the structure illustrated below:

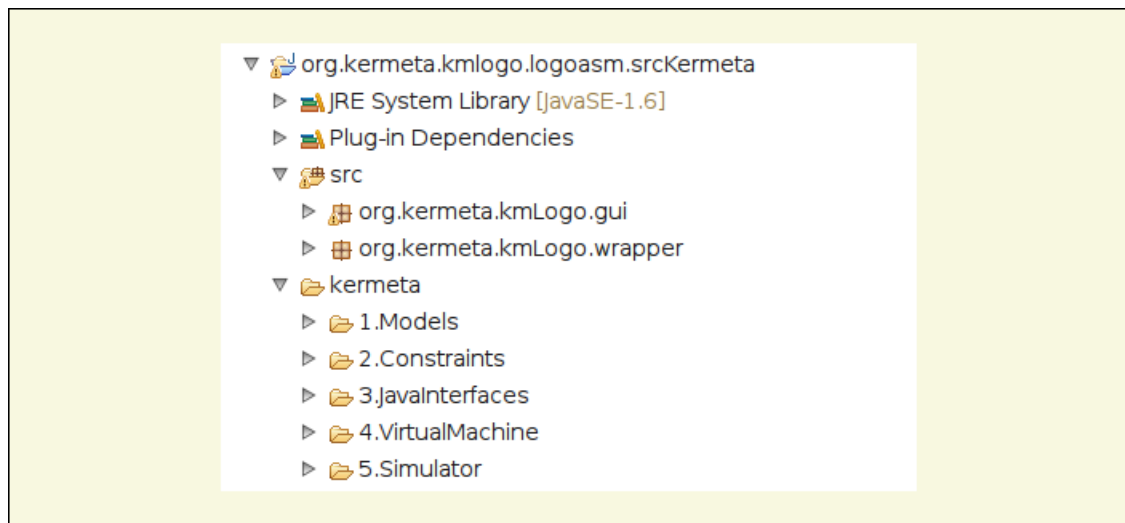


Figure 7.1. org.kermeta.kmlogo.logoasm.srcKermeta project structure

7.2. Dynamic semantic

Now, still using aspect weaving, we are going to add some operations and properties to the ASMLoogo.ecore in order to provide an operational semantics to Logo. We'll also see additional Kermeta features on aspect weaving.

We will weave an interpreter to the ASM by adding "eval" operations to the ASM metaclass where there's a behaviour. This eval operation will pass the actual context of the program.

1. Open **org.kermeta.kmlogo.logoasm.srcKermeta/kermeta/5.Simulator/LogoDynSemantics.kmt** and observe how the weaving (obtained through the require statement at the beginning of the LogoDynSe-

mantics.kmt) adds some operations and properties to the ASMLogo.ecore.

2. In the top right corner of the **outline view**, unfold **kmlogo::ASM package**. Then use the red/blue package icon to see what comes from the opened kmt file (in **red**) and what comes from the imported files (in **blue**). So the "blue-red-mix" colored icon tell you that the class has an aspect weaved into it.
3. Find the class **Repeat** and observe that it only has a Block property imported from the.ecore.

Inside this class, let's now simply add operational behavior in the eval operation as following:

```
aspect class Repeat
{
  method eval(context : Context) : Integer is do
    from var i : Integer init condition.eval(context)
    until i < 1
    loop
      result := block.eval(context)
      i := i - 1
    end
  end
}
```

Save the file and observe that the method eval was added in the outline view of the class Repeat. And because this aspect comes from this opened kmt file it is colored in red (see figure below).

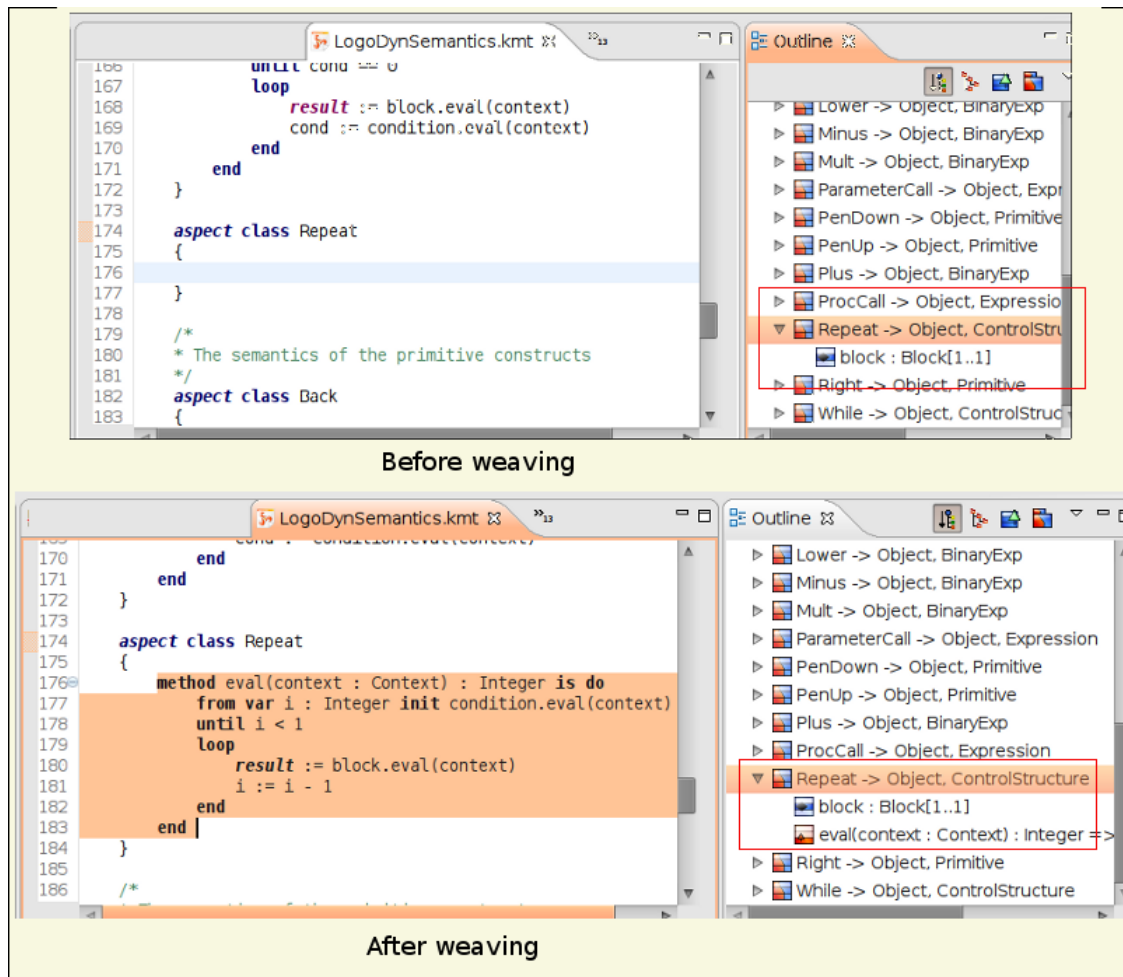


Figure 7.2. Outline view

4. Finally, find the class `Context`, and observe what is passed between `eval()` calls.

At this point, you know how to weave operational semantic to Kermeta and rapidly identify them in the Outline view. **Aspected classes** are represented in Red/Blue color.

7.3. Virtual machine

In order to provide the behaviour, the interpreter needs an application domain. In this sample we call it Virtual Machine (VM).

1. Open the file `org.kermeta.kmlogo.logoasm.srcKermeta/kermeta/4.VirtualMachine/VMLogo.ecore`,

initialize its ecore diagram file to see its representation.

In this ecore file you'll notice the various notions needed for this VM : turtle, segment and points.

2. Open **4.VirtualMachine/LogoVMSemantics.kmt** and observe how the weaving (obtained through the require statement at the beginning of the LogoVMSemantics.kmt) adds some operations and properties to the VMLogo.ecore. Those operations have some impacts on the GUI through command like `stdio.writeln`.
3. Like in the section before, use the outline view to identify what has been added (in red) to the structure (in blue).
4. Find the operation **move** in the class **Turtle** and observe some interactions with the UI (`stdio.out`).

Important

As you may have noticed inside **LogoVMSemantics.kmt**, there is a call (require to the beginning of the file) to **Math.kmt** which is a Java interface needed to properly execute the simulator. So **before moving onto the testing the simulator**, let's have a look at how to set up this interface into the next section.

7.4. Operational semantics extra

In this step, we will see how some extra feature has been implemented in the operational semantics. It will explain how to call some java code to implement extra features.

7.4.1. Call to `java.lang.Math`

This step will show you how a call to java may workaround some missing feature in Kermeta. It will explain how to call **java.lang.Math** from Kermeta. This is useful since the Kermeta doesn't provides **sin**, **cos**, **tan** operators by default. This may also be useful in case of major performance issue for a specific task.

1. Open **org.kermeta.kmlogo.logoasm.srcKermeta/kermeta/3.JavaInterfaces/Math.kmt** which is the Kermeta wrapper. It defines the operations as seen by a kermeta code. The calls to java are done using the **extern** keyword. It will call the static java operation with the given qualified name.
2. Then, open the file **src/org.kermeta.kmlogo.wrapper/Math.java** the Java wrapper. It defines the static operations that the interpreter can call. All parameters must be of type **RuntimeObject**. This class is in charge of:
 - translating the RuntimeObject to and from classical java object (here `java.lang.Double`);
 - calling the appropriate java code

Important

Remember that the objects in java side must implement a `RuntimeObject` in order to be manipulated with Kermeta. This is why all static method contains `RuntimeObject` parameters which is the Kermeta object kept in memory at runtime, and at last, must return a `RuntimeObject`.

Also notice that the conversion to java types is done by the `TYPE.create()` method.

3. In order to use this wrapper, simply **import** the `Math.kmt` (see `LogoVMSemantics.kmt`). Then, use it as a normal kermeta class. Since both java code is contained by the same plugin, there is no need of classpath declaration. If you need to use java code from another plugin, make sure that your plugin has a dependency to it.
4. To test this section, open the kermeta file `3.JavaInterfaces/test/testMath.kmt` which contains the main operation to launch the java wrapper. Observe how a main class is defined in Kermeta and the use of `Math.kmt`.

Right-click on `testMath.kmt` and choose **Run > Run as Kermeta Application**, you should obtain the results illustrated below :

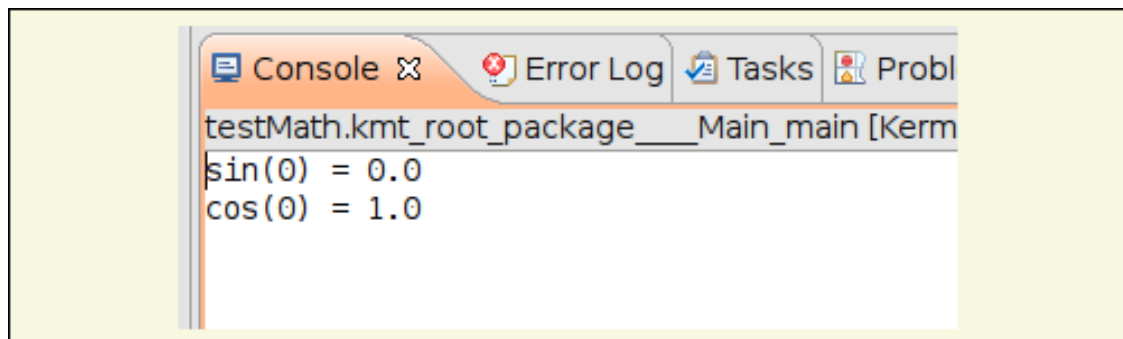


Figure 7.3. Call to java Math results

7.4.2. Implementing a graphical output

This section will guide you through the implementation of a graphical interface for the logo turtle. It is similar to the call to `Java.Math` previously seen.

1. Open the file `org.kermeta.kmlogo.logoasm.srcKermeta/kermeta/3.JavaInterfaces/TurtleGUI.kmt`. Like the java call that we have previously seen, it correspond to the kermeta wrapper. It defines the classes as seen by kermeta programs. For this application, it provides operations like **drawLine**, **drawTurtle** or **clearDrawing**. Note that it need an "initialize" operation in order to correctly create the object.
2. Then open the file `org.kermeta.kmlogo.wrapper/TurtleGUIWrapper.java`. It defines the static operations that the interpreter can call. All parameters must be of type `RuntimeObject`. This class is in charge of :

- translating the `RuntimeObject` to and from classical java object;
- calling the appropriate java code. The associated java object (here a `ITurtleGUI`) is stored into the `UserData` of the `RuntimeObject`.

Tip

Again, remember that the objects in java side must implement a `RuntimeObject` in order to be manipulated with Kermeta. This is why all static method contains `RuntimeObject` parameters (which are the Kermeta object kept in memory at runtime), and must return a `RuntimeObject` object type.

Also notice that the `initialize()` method allow the "conversion" into java side i.e.: it creates a java object from the given name so as to represent the GUI (`turtleGUI`). Then it sets this object inside the `RuntimeObject` by the method `setUserData()` to store it for later re-use which is done with `getUserData()`. The conversion to java types is done by the `TYPE.getValue()` method in this direction.

So if you want to implement yours, be sure that all your java object are not external to the Kermeta (`RuntimeObject`) domain, otherwise you must implement an internal controller to set up the glue (i.e.: the mapping).

3. Then, the extra step here is implementing the graphical user interface. This is done by the classes in `org.kermeta.kmlogo.logoasm.srcKermeta/src/org.kermeta.kmlogo.gui/`. In order to allow some extensibility, it has been splitted into an interface `ITurtleGUI` and a simple concrete AWT implementation `TurtleSimpleAWTGUI`.
4. The last step before testing is to provide a simple controller that asks to the GUI to update the graphical view of a given Turtle. This is done by the file: `4.VirtualMachine/TurtleControler.kmt` which implements `4.VirtualMachine/MoveListener.kmt` (merely a turtle movement listener).
5. After succesfully testing the call to `Java.Math`, let's now test the virtual machine that we saw in section 7.3. To do so, open `4.VirtualMachine/tests/testVM.kmt`. This file will initiate a turtle and draw a square using the kermeta wrappers and the java wrappers that we saw before (look at the required files define at the beginning of the kmt to see the interaction between the files). Right-click on this file an **run it as a Kermeta application** to see the graphical representation illustred below (notice by the way the console output that illustrate the impact of the GUI as mentioned in the step 2 of section 7.3):

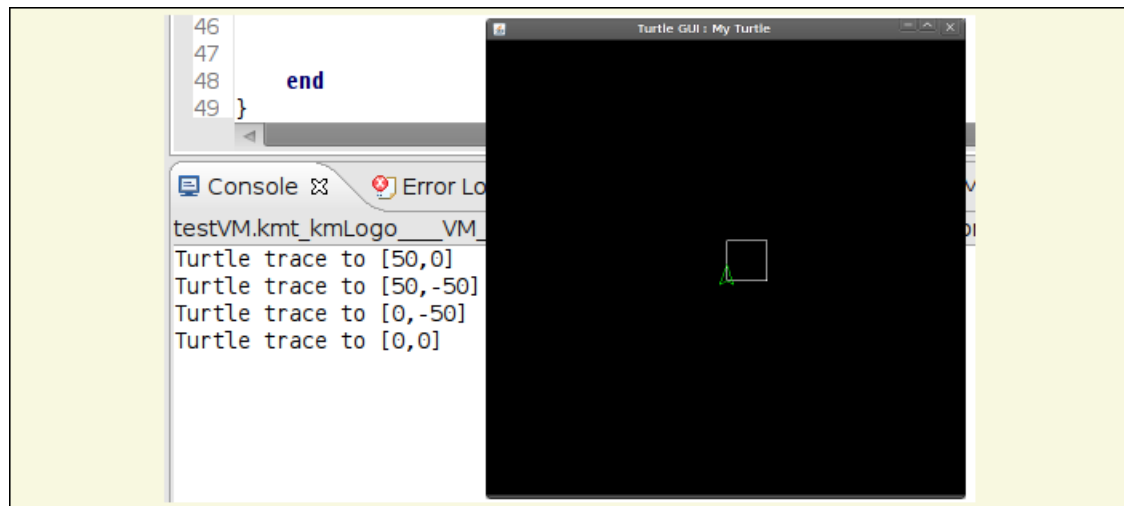


Figure 7.4. VM testing output

6. Open **org.kermeta.kmlogo.logoasm.srcKermeta/5.Simulator/ LogoSimulator.kmt**. This class provides an *execute* operation. This operation allows to load a model that conforms to the ASMLogo.ecore and start the evaluation of the logo instructions. Finally, it asks to the controller to update the graphical view. This is the main entry and the loader for a logo model instance.

Now that we saw in "step 5" that the VM is working correctly, let's test the whole interpreter that we have set up. To do so, open the file **5.Simulator/tests/carre.kmt**. It will instantiate a simulator, load an instance model from **1.Models/carre.logoasm** and execute the interpreter. Right-click the file and **run as kermeta application**, you should obtain the same illustration as before.

7. Let's have another test. Remember the "dots.xmi" that you've created before inside **org.kermeta.kmlogo.logoasm.model/model** (if not, a copy is available at "org.kermeta.kmlogo.logoasm.tutorial/parts/1.metamodel/model"). Move it to the folder **org.kermeta.kmlogo.logoasm.srcKermeta/kermeta/1.Models/**. Open it and save it as **dots.logoasm** so as to convert it as a model instance understandable by our interpreter (remember the file extension that we've set up in the .genmodel). Now, open the file **5.Simulator/tests/dots.kmt** and see that it calls the model we've just renamed. Run it as a Kermeta application to see the results (you can test the other models (*.logoasm) in the folder **5.Simulator/test**).

Model transformation - Compilation

In this next step, we'll discover how to map a logo language to lego Mindstorm environment. Logo language here is like a *Platform Independent Model* (PIM) and the target program is like a *Platform Specific Model* (PSM).

Kermeta allow to weave a "compilation" aspect into the logo metamodel.

1. Copy the folder **org.kermeta.kmlogo.logoasm.tutorial/parts/4.compiler/6.NXTCompiler** to **org.kermeta.kmlogo.logoasm.srcKermeta/kermeta**
2. Open the file **6.NXTCompiler/LogoNXCCompiler.kmt**. It adds some operations and properties to the ASMLogo.ecore. In order to provide a compiler to Logo, it weaves "compileToNXC" operations to the ASM metaclasses. The compiled code uses a predefined API specific to the target platform (here a Lego Mindstorm robot).
3. Find operation "compileToNXC" of class "If" and observe how the code is compiled (remember that you can navigate through the outline view).
4. Find operation compileToStdOut() of class "NXCCompiler". Normally it should call an operation "get-API" to define the base API of the target platform but right now it is hard coded until we fix the compiler.

To launch compiling, refer to the "chapter 10 Using the deployed DSL environment".

UI improvements

This chapter introduces the code needed to deploy a Kermeta code for a end user (eclipse GUI). It will use the interpreter as sample.

9.1. GUI deployment setup

We need to add some GUI elements to interact with the interpreter sample through eclipse GUI. To do so, we are going to set up 2 plugin projects one to handle ui settings and one to handle needed libraries :

1. Create a new plug-in project and name it : **org.kermeta.kmlogo.logoasm.ui**. This plugin project contains the eclipse extension and the code that allows to launch a kermeta program from Eclipse GUI. This hides the language in which the transformation is written in to the end user.

On the creational wizard : Be sure that you set the activator at **org.kermeta.kmlogo.logoasm.ui.Activator** then hit next.

Check **custom plug-in wizard** and hit next, select **New File Wizard** and **Popup Menu** then hit next.

2. In the **New Wizard Option** :

- set the Java package name to **org.kermeta.kmlogo.logoasm.ui.tools.wizards**;
- set Wizard Category Name to **Kermeta samples**;
- set WizardClass Name to **KmLogoExampleWizard**;
- set WizardPage Class Name to **KmLogoExampleWizardPage**
- set Wizard name to **KmLogo samples**;
- file extension to **kmt** and initial file name to **new_file.kmt**.

In the **Sample Popup Menu**

- set Submenu name to **Logo**;
- set Action Label to **Run Logo Simulator**;
- set Java package name to **org.kermeta.kmlogo.logoasm.ui.popup.actions**;

- set Action Class to **RunLogo** and hit finish;

If you are asked whether to open the plug-in perspective which you are not familiar with, you can choose "No".

3. Copy (and replace) all the files :

- **CompileNXCLogoK.java, ExecHelper.java, RunLogoK.java, ConvertToLogoasmK, ConvertToLogoK, Activator.java** from the folder **org.kermeta.kmlogo.logoasm.tutorial/parts/5.ui/uiSetting/ui** into the package **org.kermeta.kmlogo.logoasm.ui**;
- **CompileLogoNXC.java, RunLogo.java, ConvertToLogoasm, ConvertToLogo**, from the folder **org.kermeta.kmlogo.logoasm.tutorial/parts/5.ui/uiSetting/popup** into the package **org.kermeta.kmlogo.logoasm.ui.popup.actions**;
- **KmLogoExampleWizard.java** into the package **org.kermeta.kmlogo.logoasm.ui.tools.wizard**.

Inside the package **org.kermeta.kmlogo.logoasm.ui.tools.wizard**, delete the file **KmLogoExampleWizardPage.java** since we do not use it.

(**Update** : Inside the package **org.kermeta.kmlogo.logoasm.ui.popup.actions**, delete the file **NewAction.java** if it exists).

It is normal that you got errors at this time just because there are needed libraries that we should define inside steps ahead.

4. Copy the file **org.kermeta.kmlogo.logoasm.tutorial/parts/5.ui/uiSetting/plugin/plugin.xml** and overwrite the one in the created project.

Open the file and have a look to the popupMenus extension. They declare the actions for the end user.

5. Copy the folder **org.kermeta.kmlogo.logoasm.tutorial/parts/5.ui/uiSetting/icons** to the root of the project **org.kermeta.kmlogo.logoasm.ui** so as to set up the correct icons for the ui.

Copy (and replace) the folder **org.kermeta.kmlogo.logoasm.tutorial/parts/5.ui/uiSetting/editor/icons** to the root of the project **org.kermeta.kmlogo.logoasm.editor** so as to set up the correct icons for the ui.

6. Copy (and replace) the file **META-INF/MANIFEST.MF** by the one from **org.kermeta.kmlogo.logoasm.tutorial/parts/5.ui/uiSetting/plugin**.

Open the file and have a look at the dependencies and the export package of the project. We've done with the first plugin project, let's now move on to the second one.

Caution

RunLogo.java is the code to call the Logo interpreter (the operational behavior we've defined for it). Open it and look at how is manipulated the kmt file;

RunLogoK.java is the code that calls Kermeta interpreter with a given kermeta program and parameters. In addition to initializing the interpreter, this code also sets the correct java classpath. Otherwise, Kermeta interpreter will not correctly find the java code called via the "extern".

You should now have no errors (if there's one left referring to an unresolved "LogoResourceFactory" regenerate the code for texteditor (repeat step 4 of 4.3))

7. Create a new plug-in project and name it : **org.kermeta.kmlogo.logoasm.ui.osspecific**. This plugin project handles the implementation of specific missing libraries according to each platform.

On the creational wizard : Be sure that you set the activator at **org.kermeta.kmlogo.logoasm.ui.osspecific.Activator** then hit next.

Select **custom plug-in wizard** and hit next, select **Popup Menu** then hit next.

8. In the **Sample Popup Menu**

- set Submenu name to **Logo**;
- set Action Label to **Compile to NXT Binary (*.rx)**;
- set Java package name to **org.kermeta.kmlogo.logoasm.ui.popup.actions**;
- set Action Class to **NXC2RXE** then hit finish;

If you are asked whether to open the plug-in perspective which you are not familiar with, you can choose "No".

9. Copy (and replace) all the files from the folder :

- **Activator.java** from the folder **org.kermeta.kmlogo.logoasm.tutorial/parts/5.ui/uiOsSpecific/ui** into the package **org.kermeta.kmlogo.logoasm.ui.osspecific**;
- **UploadRXE.java** from the folder **org.kermeta.kmlogo.logoasm.tutorial/parts/5.ui/uiOsSpecific/popup** into the package **org.kermeta.kmlogo.logoasm.ui.osspecific.actions**;

It is normal that you got errors at this time just because there are needed libraries that we should define in next step.

10. Set up the missing libraries : Copy the folder **org.kermeta.kmlogo.logoasm.tutorial/parts/5.ui/uiOsSpecific/lib** and paste it into your root project.

Unfold this folder, add **bluecove.jar**, **icommand.jar**, **RXTXcomm.jar** to the Java build path (right-click) and move to the next step.

11. Copy the file **org.kermeta.kmlogo.logoasm.tutorial/parts/5.ui/uiOsSpecific/plugin/plugin.xml** and overwrite the one in the created project.

Open the file and have a look to the popupMenus extension. They declares the actions for the end user.

Note

You can customize with your own action by opening the **Extensions** tab of **plugin.xml** file and choose **add** button to add your action.

12. Copy and replace the file **META-INF/MANIFEST.MF** by the one from

org.kermeta.kmlogo.logoasm.tutorial/parts/5.ui/uiOsSpecific/plugin.

Open the file and have a look at the dependencies and the export package of the project.

To be sure that the editor's configuration has not been overridden, right-click on **org.kermeta.kmlogo.logoasm.emftexteditor/concrete_syntax/ASMLogo.cs** and choose to **Generate Text Ressource** again.

Now that everything is correctly configured, let's jump into the last step of this tutorial which is the deployment the Logo demo.

Using the deployed DSL environment

Now that we've set everything up correctly, it's time for testing the whole! This last step describes the way you use your DSL within an eclipse workbench.

1. Start a new eclipse worbench **Run > Run Configuration > new Eclipse application**
2. Create a simple project and name it **example**
3. Go to **File > New > EMFText > EMFText .logo file** and create a file **carre.logo** in this newly created project.
4. Type in the example code below to tell the turtle to draw a square. If you get some errors, hit **CtrlSpace** and enjoy the built-in context assistant to help you resolving them.

```
to carre :size
  Repeat 4 [
    Forward :size
    Right 90
  ]
end
Clear
PenDown
carre ( 50 )
PenUp
```

Save your program and right-click on it and choose **Logo > Logo to logoasm** in our created pop-up menu. Look at your project directory : a new file **carre.logoasm** was created.

5. Rihgt-click on the file **carre.logoasm** and select **Logo > Run Logo Simulator**.

You should see our graphical representation of a turtle drawing a square like in section 7.4.2.

6. As you can see in the popup menu there are other available actions that we've defined early inside the extension of the plugin.xml file. Now if you want to compile, right clic on the file **carre.logoasm**, and choose **Generate NXC Code** (If you want an xmi version you can hit "File > Save" to save your file in **.xmi** type). You should see a newly created file **carre.nxc** which is a C-like language containing the instructions.

Note

For some reason (an uri converter, path setup, ..) some may not find the *.nxc file inside their project but into a folder "platform:" and its subfolder in their workspace. You should manually copy it into your workspace

7. From that .nxc file, you can convert it to a binary file ".rxex" one by choosing **Logo > Compile to NXT Binary**. Finally you got a ".rxex" that you can upload by bluetooth to your Logo turtle device by the same menu **Logo > Upload RXEX**. If the above submenu doesn't show, see the caution below.

Caution

For this last step, you need to set up a corresponding environment (which is not covered by the target of this tutorial) if you want to implement the compiler.

Maybe you need to check out the solution that we provide in the conclusion in order to successfully achieve this last step and see the complete version including the linux, macos, and windows integration. Additionally, you may also follow instructions inside **lib/INSTALL.rxtx** inside the project **org.kermeta.kmlogo.logoasm.ui.osspecific**.

Conclusion

This is the end of this tutorial. Now you should be able to implement your own DSL language using Kermeta. Through this example, we provided a Model Develoement Kit (MDK) in order to help you build your own one from **"just" an .ecore file and Kermeta**. Like the FSM tutorial, this was a concrete example, the process is described in the **Building DSL main process** tutorial (See Building DSL with kermeta tutorials) with the **"big picture"** (See Introduction chapter) of what we've built here. Information on Kermeta language is available inside the manual and other tutorials on Kermeta are available on tutorial page (external link).

Note

If you want to compare your results with the solution code deployed inside the Kermeta Eclipse or if you want to shortcut to the final results directly, you may launch **File -> New -> Example... -> Kermeta Samples -> Logo tutorial - solution**. It will load a bunch of project as shown in the project structure below :

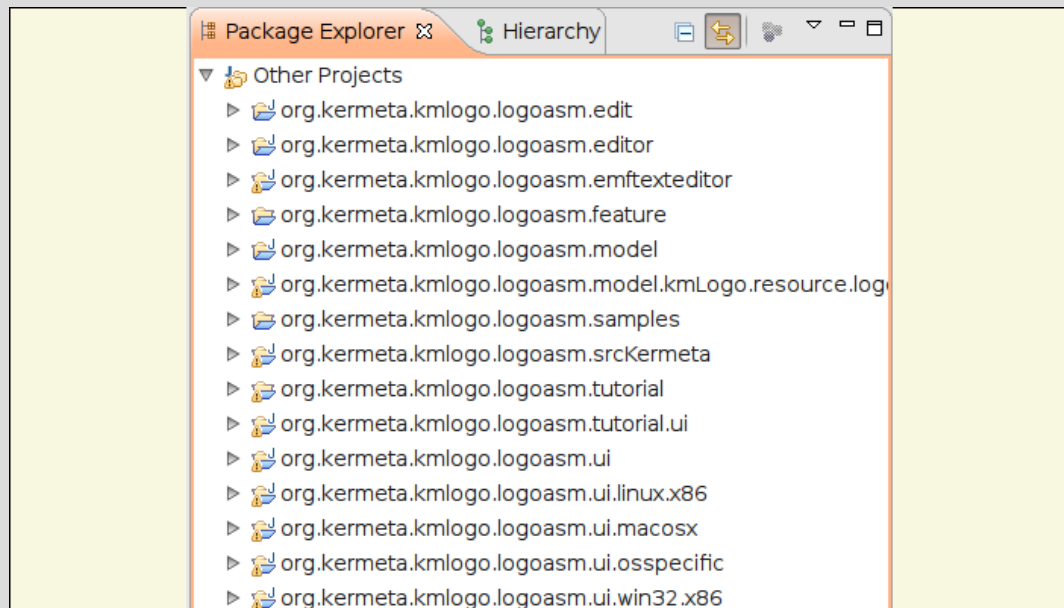


Figure 11.1. Logo solution project structure