Deploying Kermeta code

How to use your Kermeta code in a product

Abstract

This document presents various ways to deploy some code written with Kermeta in order to "hide" kermeta to the end users.

It covers the integration in an eclipse environment and deployment as a standalone application.

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

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.

A MDK (Model Development Kit) is a set of Kermeta code that goes with a given metamodel. Each of the MDKs provides functionalities dedicated to the metamodel it applies to. (Ex: UML, Ecore, Traceability, yourDomainMetamodel, ...)

In this document we will focus on Kermeta MDK. This MDK applies to Kermeta metamodel itself. Ie. it provides functionalities to manipulate Kermeta programs.

> **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: **ht-tp://gforge.inria.fr/tracker/?group_id=32** or using the developer mailing list (kermeta-developers@lists.gforge.inria.fr)

> **Tip**
>
> The most update version of this document is available on line from ht-tp://www.kermeta.org/documents/tutorials/task_oriented_tutorials/deployment_manual .

Contributing to Eclipse user interface

Kermeta isn't dedicated to build user interface, however, you may embed your kermeta application in Eclipse user interface to make it simple to use for the end users. We present here some basic informations for those who which to build such user interface. For more details or for different integration in Eclipse, the reader should refer to the eclipse Platform Plug-in Developer Guide (available in the eclipse help or http://help.eclipse.org/help32/index.jsp .

1.1. Contributing a popup action to eclipse

This should be done from a valid plugin project.

1. Open the `META-INF/MANIFEST.MF` file.

2. Open the tab `Extensions`.

3. Click on `Add` and select the `Extension wizards` tab.

4. You'll follow the steps provided by the `Popup Menu` Wizard and give a meaninful name to your action.

5. You can optionnally set the `nameFilter` wild card in order to restrict the popup to some file extesion only.

You now have a `NewAction.java` file that you can edit to perform the actions you wish, for example by calling kermeta interpreter like explained in Chapter 3, *Launching a kermeta program in interpreted mode*.

> **Tip**
>
> Obviously, you can reuse this wizard on other kind of interface, for example to provide an `Action set` or an `Help content`.

1.2. Deploying a plugin with Kermeta code for interpreted mode

If you wish to deploy some kermeta code for your end users, you must take care of some rules :

It must be put into a bundle (or eclipse plugin). To do so, the project containing the kermeta code must be a valid Eclipse plugin project.

Make sure to deploy the kmt file in the distributes version. In the manifest.mf, in the Build tab, make sure that the files/folders you want to deploy are checked in the Build section. Otherwise, your plugin will work in development mode but will fail once deployed in your user platform.

1.3. Transforming a project into a plugin project

There are some simple steps that you can follow to obtain a valid Eclipse plugin project.

1.3.1. Use the new Plugin project wizard

This is the simpliest approach.

First, if your project already use the name of the plugin you want to create, rename it to another name.

Then, use the New plugin project wizard from Eclipse.

```
File > New  > Project...  > Plugin Project
```

Then, follow the wizard.

If this plugin is a user interface for one of your metamodel, you should consider adding the postfix .ui to your project name.

Tip

Using this wizard, you can also create some template popup, action, help, etc as in Section 1.1, "Contributing a popup action to eclipse"

1.3.2. Manually add the missing elements

You can achieve more or less the same result by adding some data in the following files.

Add the java and plugin nature to your project and the associated buildCommand For this add something similar in the .project file at the root of your eclipse project.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```xml
<projectDescription>
    <name>your.plugin.name.ui</name>
    <comment></comment>
    <projects>
    </projects>
    <buildSpec>
        <buildCommand>
            <name>fr.irisa.triskell.kermeta.kpm.kpmBuilder</name>
```

```
            <arguments>
            </arguments>
        </buildCommand>
        <buildCommand>
            <name>org.eclipse.jdt.core.javabuilder</name>
            <arguments>
            </arguments>
        </buildCommand>
        <buildCommand>
            <name>org.eclipse.pde.ManifestBuilder</name>
            <arguments>
            </arguments>
        </buildCommand>
        <buildCommand>
            <name>org.eclipse.pde.SchemaBuilder</name>
            <arguments>
            </arguments>
        </buildCommand>
    </buildSpec>
    <natures>
        <nature>org.eclipse.pde.PluginNature</nature>
        <nature>org.eclipse.jdt.core.javanature</nature>
    </natures>
</projectDescription>
```

Tip

If you see the .project file, use the Navigator view that show all the files including the hidden ones.

Add a `.classpath` file like:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<classpath>
    <classpathentry kind="con" path="org.eclipse.jdt.launching.JRE_CONTAINER/org.eclipse.jdt.internal.debug.ui.launcher.
    <classpathentry kind="con" path="org.eclipse.pde.core.requiredPlugins"/>
    <classpathentry kind="src" path="src"/>
    <classpathentry kind="output" path="bin"/>
</classpath>
```

Add a `META-INF/MANIFEST.MF` file like:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Your Plug-in
Bundle-SymbolicName: your.plugin.name.ui
Bundle-Version: 1.0.0
Bundle-Activator: your.plugin.name.ui.Activator
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

Preparing a kermeta program for a launch

Here is some tips for preparing a kermeta program in order to be launched by a user interface.

Use String parameters. The main operation that'll be started should take only String parameters.

```
operation myMain( inputFileURI : String, outputFileURI : String, option : String) is do
// ...
end
```

Example 2.1. Typical kermeta operation used as an entry point

If the code must be interpreted, the easiest way is to deploy the kermeta code in a plugin. all the required kermeta code, kmt, km, ecore must be in plugins as explained in Section 1.2, "Deploying a plugin with Kermeta code for interpreted mode". The only exception is a bout the registered Ecore, which follow EMF way for deploying the java code for the ecore file.

> **Note**
>
> There are some ways to pass to the launched program something more complex than String, however this requires a more coupled interaction and knownledge of the Interpreted or the Compiled code. If you need this, please ask on the user mailing list and explain you use case for further assistance. We'll be glad to guide you.

Launching a kermeta program in interpreted mode

You can start Kermeta interpreter on you kermeta code from a java program, however you must be aware of the run context. You'll probably run the interpreter in Eclipse environment or as a standard java application. Actually, the simpliest is to run in eclipse because you don't have to bother about the jar used by the interpreter and eclipse itself but you'll have to do some GUI integration (create a popup menu for example). On the other hand, you won't have to create a GUI, but you have to care about file name resolution and jar location.

Tip

Do not be confused, you can run a standard java application from Eclipse IDE. In this case, you need to follow the explanation of Section 3.2, "Interpreted mode in standard java".

3.1. Interpreted mode in Eclipse environment

Here is a typical code for starting a kermeta interpreter on a kermeta program.

Note

This sample is extracted from the kmLogo example which is distributed with Kermeta. You can retrieve this sample by doing the following steps.

File > new > Example... > Kermeta samples > km Logo tutorial (plugins for the main workbench)

The action file corresponding to this section are in

fr.irisa.triskellkmlogo.ui/src/fr.irisa.triskell.kmlogo.ui.popup.actions

```java
package fr.irisa.triskell.kmlogo.ui.popup.actions;

import java.util.Iterator;

import org.eclipse.core.resources.IFile;
import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.StructuredSelection;
import org.eclipse.ui.IActionDelegate;
```

```java
import org.eclipse.ui.IObjectActionDelegate;
import org.eclipse.ui.IWorkbenchPart;

import fr.irisa.triskell.eclipse.console.EclipseConsole;
import fr.irisa.triskell.eclipse.console.IOConsole;
import fr.irisa.triskell.eclipse.console.messages.ErrorMessage;
import fr.irisa.triskell.eclipse.console.messages.InfoMessage;
import fr.irisa.triskell.eclipse.console.messages.OKMessage;
import fr.irisa.triskell.eclipse.console.messages.ThrowableMessage;
import fr.irisa.triskell.kmlogo.ui.RunLogoK;

/**
 * The action may take a long time, so the concrete action is embedded in a Runnable Thread
 *
 */
public class RunLogo implements IObjectActionDelegate, Runnable {

  protected StructuredSelection currentSelection;
  protected IFile logoFile;

  public static final String LOGO_SIMULATOR_KERMETA_CODE =
    "platform:/plugin/fr.irisa.triskell.kmlogo.model/logo/5.Simulator/LogoSimulator.kmt"; ❶

  /**
   * Constructor for Action1.
   */
  public RunLogo() {
    super();
  }

  public void run() {

    IOConsole console = new EclipseConsole("Logo Simulator"); ❷
    console.println(new InfoMessage("Launching logo interpreter on file : " + logoFile.getLocation().toOSString() + "..."));

    try {

      String file_uri = "file:/" + logoFile.getLocation().toOSString();

      Interpreter interpreter = new Interpreter(LOGO_SIMULATOR_KERMETA_CODE, InterpreterMode.RUN, null);
      interpreter.setStreams(console);        ❸
      interpreter.setEntryPoint("kmLogo::Interpreter", "execute");❹
      String[] parameters = new String[]; ❺
      parameters[] = file_uri;
      interpreter.setParameters(parameters);

      // Add some URL to the classloader of the interpreter : needed if your code use some extra java classes (via extern for example)
      // use a set for building the URL (in case some may fail due to malformed URL
      // Note : URL must end with a / if this is a directory, if not, this is considered as a jar by the classloader
      Set<URL> urlsSet = new LinkedHashSet<URL>();
      // URL used when run in a runtimeworkbench, this allows to debug the plugin
      safeAddURLAsString(urlsSet, "file://" + FileLocator.resolve(Platform.getBundle("fr.irisa.triskell.kmlogo.model").getEntry("/bin/")));
      // add this plugin as a deployed plugin
      Bundle bundle = org.eclipse.core.runtime.Platform.getBundle("fr.irisa.triskell.kmlogo.model");
      if(bundle != null){
        urlsSet.add(FileLocator.resolve(bundle.getEntry("/")));
      }
      // convert the set into an array
      URL[] urls = new URL[urlsSet.size()];
      int i = ;
      for (URL url : urlsSet) {
        urls[i] = url;
        i++;
      }
      URLClassLoader cl = new URLClassLoader(urls, interpreter.getClass().getClassLoader());
      Thread.currentThread().setContextClassLoader(cl);

      interpreter.launch(); ❻

      console.println(new OKMessage("Execution terminated successfully."));
```

```
      } catch (Throwable e) { ❷
        console.println(new ErrorMessage("Logo runtime error : "));
        console.println(new ThrowableMessage(e));
        e.printStackTrace();
      }
    }

    /**
     * @see IObjectActionDelegate#setActivePart(IAction, IWorkbenchPart)
     */
    public void setActivePart(IAction action, IWorkbenchPart targetPart) {
    }

    /**
     * @see IActionDelegate#run(IAction)
     * Create a new thread for this concrete action
     */
    public void run(IAction action) {
      new Thread(this).start();
    }

    /**
     * @see IActionDelegate#selectionChanged(IAction, ISelection)
     * This allow to retrieve the file selected by the user when activating the popup
     */
    public void selectionChanged(IAction action, ISelection selection) {

      if (selection instanceof StructuredSelection)
      {
        // the selection should be a single file
        currentSelection = (StructuredSelection)selection;
        Iterator it = currentSelection.iterator();

        while(it.hasNext()) {
          logoFile = (IFile)it.next();
        }
      }
    }
    /**
     * add a new URL to the set
     * Doesn't fail if the URL is malformed, in that case, only a warning is raised,
     * @param urlsSet : set that will contain the URL built
     * @param url : String of the URL to build
     */
    private static void safeAddURLAsString(Set<URL> urlsSet, String url){
      try{
        urlsSet.add(new URL(url));
      } catch (MalformedURLException e) {
        Activator.logWarningMessage(
          "problem adding an entry of the classpath, "
          + url + " cannot be added in classloader", e);
      }
    }

}
```

❶ Path to the kermeta program that will be run.
❷ Kermeta stdio.writeln outputs must be redirected to an IOConsole. This EclipseConsole will print to the user console in eclipse.

The plugin `fr.irisa.triskell.eclipse.util` also provides so other console kind like `Loc-alIOConsole` that prints to the standard streams.
❸ Tells the interpreter where the stdio will be written.
❹ Indicates the main class and the operation that will be started.
❺ Pass the String parameters to the interpreter. This must conforms to the expected parameters of the main

operation specified as entry point.

❻   Finally starts the interpreter and wait for the end of the kermeta program.

> **Note**
>
> In the Interpreter API, it exist some other way to start the program, for example `launchAndWait()` which doesn't release the interpreter memory at the end of the operation which is useful in some special applications. Feel free to ask question to the development team.

❼   Use various means to tell the user that something wrong occurred. However, if you expect some exception and haven't catched them in the kermeta program (for example when checking a model invariant, you may add here some code to report a better message to your users).

3.2. Interpreted mode in standard java

Most of the elements seen in the previous section is still correct (especially how to initialize the Interpreter with your kermeta program and passing the parameters). In addition to that you must be aware of some point.

> **Tip**
>
> A good sample for building your own standard java calling kermeta interpreter is to look at the Run-CommandLine.java class which implements a command line argument parser and calls the Interpreter on top of a standalone distribution. http://gforge.inria.fr/scm/viewvc.php/trunk/kermeta_projects/fr.irisa.triskell.kermeta.interpreter/src/java/org/kermeta/interpreter/api/RunCommandLine.java?view=markup&root=kermeta

3.2.1. `platform:/resource` and `platform:/plugin` resolution

These URI are specific to Eclipse and aren't known by default in Java. You'll need to guide indicate how to map them to the classical `file://` URIs

This is done by adding some URI map entry in EMF.

```java
import org.eclipse.emf.ecore.resource.impl.ExtensibleURIConverterImpl;

ExtensibleURIConverterImpl.URI_MAP.put(URI.createURI("platform:/plugin/"), URI.createURI("file://path_to_you_folder_containing_a_
ExtensibleURIConverterImpl.URI_MAP.put(URI.createURI("platform:/resource/"), URI.createURI("file://path_to_you_folder_containing_
```

Example 3.1. adding URI map entries in EMF (java)

> **Tip**
>
> In the above sample, we map only two folders, however you can reach more complex mapping scheme. Please have a look in eclispe help for more details. http://help.eclipse.org/galileo/topic/org.eclipse.emf.doc/references/javadoc/org/eclipse/emf/ecore/resource/impl/ExtensibleURIConverterImpl.html#getURIMap()

3.2.2. classpath

Obviously, as you run as a standard java application you'll need to include inthe classpath all the jar that constitute Kermeta interpreter and their dependencies to eclipse jars.

If your code use `extern` call, you'll also need to provide the jar that contains the java implementation of those extern operation.

> **Tip**
>
> The standalone distribution of kermeta is a repackaging of kermeta and all its dependencies in a convenient jar (or set of jars) that doesn't include those nasty version number eclipse use. It's more convenient to build your classpath from it.

Registering Ecore file

Usually, when deploying a code for a given metamodel, we deploy this metamodel in a EMF plugin that contains the generated java code for this metamodel. When using this, Eclipse takes care of registering the EPackage in its registry.

If for some reason you don't want to deploy the java version of the metamodel, you can also manually register it directly from an ecore file.

In the graphical interface, this will be done by

> right click on the ecore file > EPackage registration > Register EPackage into repository

.

If you need to add an ecore file into the registry programmatically, you can do it in two ways:

- In kermeta, use the `registerEcoreFile(`*`fileUri`*` :` String) operation provided by `ker-
meta::persistence::EMFRepository`

> kermeta::persistence::EMFRepository.*new*.registerEcoreFile(*"MyMetamodel.ecore"*)

- In java, use the API provided by the `org.eclipse.emf.ecoretools.registration` plugin.

> **import** org.eclipse.emf.ecoretools.registration.EcoreRegistering;
> *// ...*
> EcoreRegistering.registerPackages(anEcoreIFile);

> **Note**
>
> despite the name of this plugin, it is provided by kermeta team, this was an attempt to contribute to ecore tools project, but this contribution is still in the pipeline in Eclipse :-(

Launching a compiled kermeta program (Compiler V1)

For each operation having like parameters a sequence of String, a Java "main" method is generated at compilation time in a Java package denoted "runner". A Java package "runner" is generated for all Kermeta or Ecore packages. Thus this main is usable as a classical Java Main method.

An example of Java Main Class for the operation: event::manager::EventManager::execute(filePath : String)

```java
package event.manager.runner;

import org.kermeta.compil.runtime.ExecutionContext;

/**
 * A Stub of a Java Application Runner for a Kermeta runnable operation
 *
 * @generated
 */
public class EventManager__execute__Runner {

    /**
     * @generated
     * Main to run the execution
     */
    public static void main(String[] args) {

        //Initialize the reflection
        ExecutionContext.getInstance().lazyInitialize();

        event.manager.EventManager anExec = (event.manager.EventManager) org.kermeta.compil.runtime.helper.language.ClassU
                .newObject(event.manager.ManagerPackage.eINSTANCE
                        .getEventManager());

        // args[0] is the parameter filePath
        // call the operation "execute" on an EventManager
        anExec.execute(args[]);

    }

}
```

If the Runner operation has not been generated at compilation time (parameters having types different of String), you should write the previous source code by the hand and adapt it following the operation.