



Project Number: 033471

SPEEDS

Speculative and Exploratory Design in Systems Engineering

Integrated Project

Information Society Technologies

HRC meta-model implementation user guide

Start date of project: May 1st 2006
S. Gebhardt/OFFIS

Duration: 48 months
Revision: 0.2

Revision: 0.2

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Dissemination Level	
PU	Public x
PP	Restricted to other programme participants (including the Commission Services)
RE	Restricted to a group specified by the consortium (including the Commission Services)
CO	Confidential, only for members of the consortium (including the Commission Services)

PROPRIETARY RIGHTS STATEMENT

THIS DOCUMENT CONTAINS INFORMATION, WHICH IS PROPRIETARY TO THE SPEEDS CONSORTIUM. NEITHER THIS DOCUMENT NOR THE INFORMATION CONTAINED HEREIN SHALL BE USED, DUPLICATED OR COMMUNICATED BY ANY MEANS TO ANY THIRD PARTY, IN WHOLE OR IN PARTS, EXCEPT WITH THE PRIOR WRITTEN CONSENT OF THE SPEEDS CONSORTIUM THIS RESTRICTION LEGEND SHALL NOT BE ALTERED OR OBLITERATED ON OR FROM THIS DOCUMENT.

HRC meta-model implementation user guide

The SPEEDS L-1 meta-model (aka HRC meta-model) implementation was created using the Eclipse Modeling Framework (EMF). EMF is a modeling framework with integrated Java code generation facility to support the building of applications based on structured data models.

The tool-chain suggested by EMF is the following:

1. Defining a meta-model (M2) through the meta-meta-model (M3) “Ecore”
2. Generating in Java the implementation of an M2, from its definition in Ecore
3. Creating models (M1) of M2
4. Populating, loading, saving and modifying the M1

Based on this, and following the HRC M2 specification, a definition of HRC was created in Ecore, the generated in Java.

Structure of the generated meta-model implementation

The HRC meta-model consists of three packages, called “ePackage” in ecore: Kernel, Rich connectors and Priorities. The code generated for each ePackage consists of usually two Java packages:

- The first package contains a set of interfaces, representing the client interface to the model.
- The second package contains a set of corresponding implementation classes.

Each MetaClass of an ePackage is defined by one interface “[metaclassName].java” and one implementation “[metaclassName]Impl.java”. The interface declares the methods to access and modify the features of the MetaClass and the class implements them (Figure 1).

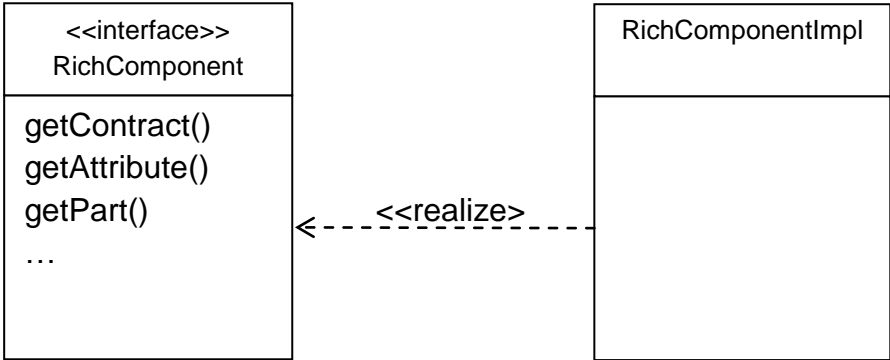


Figure 1 Interface and Implementation Class for MetaClass RichComponent

In addition to the MetaClasses, the ePackage itself is defined by an interface and an implementation which provide accesses to the ePackage metadata.

Then a Factory is generated in the same way, containing all the functions required for creating instances of MetaClasses.

An optional utility package is also created for each model package. These packages contain utility classes which can be overridden to implement extended functionality.

Using the generated implementation

In the following sections it is briefly shown how to use the generated HRC meta-model implementation to create a model instance and to save/load it to/from a file. To compile and execute the shown code-fragments from outside of Eclipse you need an installed Java SDK version 1.5 or higher and you have to add of course the jar-archive (com.eu.speeds.model_[version].jar) of the meta-model implementation to the Java classpath and also the following jar-archives of EMF (at least version 2.3):

- org.eclipse.emf.common_[version].jar
- org.eclipse.emf.ecore_[version].jar
- org.eclipse.emf.ecore.xmi_[version].jar

Creating and accessing a model instance

Let's have a look at the following HRC model:

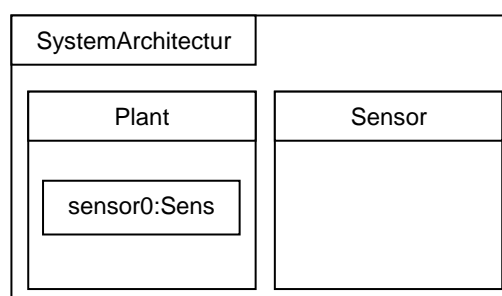


Figure 2 Example HRC Model Instance

HRC meta-model implementation user guide

The model consists of an HRC Package containing two Rich Components. To create this programmatically, the following Java statements are sufficient:

```
1 // get the Kernel Factory
2 KernelFactory factory = KernelFactory.eINSTANCE;
3
4 // create the Package "SystemArchitecure"
5 Package p0 = factory.createPackage();
6 p0.setName("SystemArchitecture");
7
8 // create the Rich Component "Sensor"
9 RichComponent rc0 = factory.createRichComponent();
10 rc0.setName("Sensor");
11 p0.getDeclared().add(rc0);
12
13 // create the Rich Component "Plant"
14 RichComponent rc1 = factory.createRichComponent();
15 rc1.setName("Plant");
16 p0.getDeclared().add(rc1);
17
18 // create the Rich Component Property "sensor0"
19 RichComponentProperty rcp0 = factory.createRichComponentProperty();
20 rcp0.setName("sensor0");
21 rcp0.setType(rc0);
22 rc1.getPart().add(rcp0);
```

The second line is just a shortcut to access the factory implementation for the Kernel package of the HRC meta-model.

The following lines of the code fragment then create the HRC Package (lines 5-6) containing two HRC Rich Components (lines 9-11 and 14-16) and the HRC Rich Component Property (lines 19-22) as part of the second Rich Component.

Saving and loading

To store the above model in the file `example.kernel`, an EMF resource, a container for arbitrary model artifacts, must be created. Therefore also an EMF resource set is needed. It manages the creation, serialization, and resolution of (cross resource) references for its resources. The following code-fragment shows how to obtain a resource and how to use it to store the above model:

HRC meta-model implementation user guide

```
1 // create a new resource set
2 ResourceSet resourceSet = new ResourceSetImpl();
3
4 // register the default XMI serializer
5 resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().
6     put(Resource.Factory.Registry.DEFAULT_EXTENSION,
7         new XMIResourceFactoryImpl());
8
9 // create a new resource as file "example.kernel"
10 URI fileURI = URI.createFileURI(new File("example.kernel").
11     getAbsolutePath());
12 Resource resource = resourceSet.createResource(fileURI);
13
14 // add the Package to the resource
15 resource.getContents().add(p0);
16
17 // save the resource
18 try {
19     resource.save(Collections.EMPTY_MAP);
20 } catch (IOException e) {}
```

The resource set is created in the second line. The next step is to register appropriate resource factories for extensions. Using a registry, the resource set can create the right type of resource for a given URI. In the code above, the XMI resource factory is registered as default (lines 5-7). With the prepared resource set, then a new resource can be created (line 12) and the above model can be added (line 15). Due to the containment relations in the HRC meta-model classes, it is sufficient to add the Package to the resource. Finally, the resource can be saved (line 19).

To load the model from the file `example.kernel`, as saved above, one first has to set up a resource set similar to the code above. Then a given resource can be loaded into the resource set as follows:

```
1 // create a new resource set
2 ResourceSet resourceSet = new ResourceSetImpl();
3
4 // register the default XMI serializer
5 resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().
6     put(Resource.Factory.Registry.DEFAULT_EXTENSION,
7         new XMIResourceFactoryImpl());
8
9 // register the Kernel ePackage
10 resourceSet.getPackageRegistry().put(
11     KernelPackage.eNS_URI, KernelPackage.eINSTANCE);
12
13 // load the content of file "example.kernel" into a resource
14 URI fileURI = URI.createFileURI(new File("example.kernel").
15     getAbsolutePath());
16 Resource resource = resourceSet.getResource(fileURI, true);
17
18 // access the resource content
19 Package p0 = (Package) resource.getContents().get(0);
```

When the content of "example.kernel" is parsed, the resource set needs to know how to restore the serialized elements. This is the reason why the ePackage Kernel has to be provided to the resource (lines 10-11). Then the resource can be loaded by the resource set (line 16) and its content can be accessed (line 19).