Developers guide on how to implement additional mapping rules' code for the CIM 2 MODelica Transformation Tool

This guide tries to give details on the developing environment and the technical understanding for the maintenance and enhancement of the transformation tool. Within in this guide, the algorithm for model transformation will be described, as well as a description of the necessary steps for the development of mapping files and for the development of new classes and methods.

I. Eclipse set-up

The current version of the development environment works with Eclipse Neon 4.6.3 and JAVA 1.8. (we suppose that you have available a Git Client, Eclipse and JAVA installed into the system)

1. Download the source code from the repository (https://github.com/ALSETLab/cim2modelica), with a Git Tool, i.e., Github Desktop
    a. Make sure you have downloaded the .project and .classpath file
2. Open Eclipse and Import the project into the current workspace: *File> Import…> Existing project into Workspace*.
3. In the Import dialog, browse on the Select Root Directory field and select the project from the list of Projects

II. Quick view of the project structure

The project contains:

1. A ***dist*** folder to store the code releases (also can be used as test directory)
2. A ***lib*** folder to store the external libraries to be used within the project
3. A ***model*** folder, where the tool stores the output models
4. A ***res*** folder, to store all the resources used by the tool: mapping files and input models
5. A ***src*** folder, with the package structure of the tool. The main packages are 1) cim2model; 2) cim2model.cim; and 3) cim2model.modelica

III. Setup Java Build Path

The project includes the reference of the original build path. Each developer might update their own build path according to per their JAVA configuration. It is recommended not to update any changes on the .project nor .classpath files.

In case you need to modify the JAVA configuration, follow this steps

A completar con otro setup

The project includes also a couple of external libraries that have been originally included in the project build path (Apache JENA and JAVA JAXB[1]). In the Project> Preferences> Java Build Path option the

[1] Due to issues with the version of the library and JAVA at the beginning of the project, we have decided to download the JAXB library from the main Git repository and work with it as external library

project should contains the references to User Libraries JENA and JAXB. If they do not appear in the Java Build Path, you need to create two User Libraries with .jar files from JENA and JAXB
We will create a *User Library* for each folder:

1. Go to Windows> Preferences> Java> Build Path> User Libraries and click on New
2. Give a name to the library
3. Select the recent created library and click on Add JARs
4. Expand the option cim2modelica> libs> apache-jena and select all the .jar files under that folder
5. The .jar files are stored under the new User Library.
6. Do the same for the JAXB library.

Now we add the recent library to the project:

1. Go to Project>Properties> Java Build Path
2. Click on Add Library and select the libraries you have created.
3. The library appears within the project

IV. Run Configuration set-up

To run the transformation tool from the Eclipse, you need to create a Run Configuration.

1. Run> Run Configurations> Java Application> New Launch Configuration
2. Select the corresponding project in Main> Project,
    a. Select the main class, i.e., cim2model.CIM2MOD
3. Specify the arguments of the tool. Arguments> Program Arguments:
    a. <relative_path_CIMFiles_Folder>
    b. <name_of_the_network>

V. Software development in detail

This section describes how to implement new mapping files and how to extend the tool class structure with new JAXB Classes.

I. Create Mapping Files

The folder res.map.openipsl contains the .xml, .xsd and .dtd files that conform the mapping rules for each OpenIPSL component available. To comply with the OpenIPSL packages' organization, the folder res.map.openipsl is organized in the same way as the library (e.g. the folder res.map.openipsl.controls.es contains mapping rules for Excitation System components).
To add a new mapping component rule, follow the next steps:

1. Copy one of the existing *.dtd, .xml, and .xsd* with the new component name. Follow the name syntax of the files. Place the new files into the corresponding folder.
2. Modify the *.dtd* changing the name the component **<!ELEMENT** and **<!ATTLIST** tags with a new name, i.e.: *<!ELEMENT iEEET1Map* replaced by *<!ELEMENT sEXSMap*)
3. Modify the *.xsd* changing the name of the **<xsd:element** main tag, i.e.: *<xs:element name="iEEET1Map"* replaced by *<xs:element name="sEXSMap"*>
4. Modify the .xml changing the name of the .dtd file in the DOCTYPE tag:

```
< !DOCTYPE model SYSTEM "cim_openipsl_sexs.dtd" >
```

5. Modify the *.xml* changing the name of the main tag, and the values of the main tag attributes with the corresponding OpenIPSL names (you can leave the tags *rdf_id* and *rdf_resources* empty)

```
<sEXSMap cim_name="ExcSEXS" rdf_id="" rdf_resource=""
package="OpenIPSL.Electrical.Controls.PSSE.ES" stereotype="class">
```

6. Modify the .xml changing the number of <attributeMap> tags, and their parameter values, per number of parameters of the OpenIPSL component.

```
<attributeMap cim_name="ExcSEXS.tatb" name="T_AT_B" datatype="Real"
variability="parameter" visibility="public"> 0 </attributeMap>
```

II. Create Mapping Files

Using the API provided by the JAXB Library, we can create for each XML mapping rule its corresponding JAVAX class, which will be integrated within the transformation tool. There are two ways of doing this:

1. First option is to create a new *Run Configuration*, within the Eclipse environment, with the class *MappingStructureGenerator.java* as main class. This class is prepared for placing the resulting JAVAX classes into the corresponding tool packages. Create the Run Configuration specifying two program arguments:
   a. Name the package to store the JAVAX class, e.g.: `cim2modelica.cim.map.openipsl.controls.es` (in case of the mapping of a new excitation system component)
   b. Relative path with the name of the mapping schema file, e.g., `./res/map/openipsl/controls/es/cim_openipsl_sexs.xsd`)

2. Second option is to use the JAXB tool, XJC, in the command line. The XJC executable will generate an additional external package, in the folder you have executed the XJC command, with the generated classes. They need to be included into the tool class structure manually.

III. Modify the code from the generated JAVAX classes

After the execution of the XJC tool three JAVAX classes are created: *SEXSMap.java* (following the example of the mapping of the excitation system), *AttributeMap.java* and *ObjectFactory.java*. Because most of the mapping rules share the same parameters, the package *cim2modelica.cim.map* contains and abstract class *ComponentMap.java* that contains the general attributes and the *getters/setters* methods. To adapt the generated classes to the Mapping Meta-Model structure of the project, follow these steps.

1. In the SEXSMap.java add the import statement:

```java
import cim2modelica.cim.map.ComponentMap;
```

2. Update the class declaration with:

```java
public class SEXSMap extend ComponentMap
```

3. In case of this component, you can delete all the JAVAX elements and the *getters/setters* methods, leaving the class empty. It inherits every attribute and method from the ComponentMap class. Just leave those attributes and methods that do not appear within the ComponentMap class, i.e.:

```java
package cim2modelica.cim.map.openipsl.controls.es;
import cim2modelica.cim.map.ComponentMap;
import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement(name="sEXSMap")
public class SEXSMap extends ComponentMap
{…}
```

4. The package cim2modelica.cim.map already contains the class AttributeMap.java. Thus, you can delete the newest one.
5. The generated *ObjectFactory.java* class can be discarded because we use the JAXB API to create a specific factory class for the new JAVAX class. This factory class, contains a factory method that unmarshalls the values from the corresponding .xml mapping file into memory.
6. In this example, copy/paste an existing factory method within the same ExcSysMapFactory.java class and adapt its code to the new component name: (Each package of the *Mapping Meta-Model* structure contains a factory class, to group the factory methods per components).

```java
public SEXSMap sexsXMLToObject(String _xmlmap) {
    JAXBContext context;
    Unmarshaller un;
    try {
        context = JAXBContext.newInstance(SEXSMap.class);
        un = context.createUnmarshaller();
        SEXSMap map = (SEXSMap) un.unmarshal(new File(_xmlmap));
        return map;
    } catch (JAXBException e) {
        e.printStackTrace();
        return null;
    }
}
```

IV. Updated controller classes to use the new component map

1. Updated the ModelDesigner.java class, adding a new method to populate the values of the new component map. Just copy one of the existing create_ methods and adapt it to the new mapping object:

```
public SEXSMap create_SEXSModelicaMap(Resource _key, String _source, String _subjectName)
{…}
```

2. Updated the ModelBuilder.java class, adding a new method to create the OpenIPSL component instance with the values of the new component map. Just copy one of the existing create_ methods and adapt it to the new mapping object:

```
public MOClass create_SpecificComponent(SpecificComponentMap _map)
{…}
```

3. See that the component created by the ModelBuilder controller class return objects of type MOClass. In case of a new ExcitationSystem component you can copy the next declaration:

```
public OpenIPSLExcitationSystem create_SEXSComponent(ComponentMap _mapExcSys)
{…}
```

4. Last step is to update the identification process of the CIM classes, within the main CIM2MOD.java class. The algorithm first starts with the identification of the CIM Terminals. Then, it identifies the ConductingEquipement and TopologicalNode classes associated to the Terminal.

```
cimClassResource= cartografo.get_EquipmentClassName(key);
if (cimClassResource[1].equals("Terminal"))
{…
equipmentResource=
   cartografo.get_EquipmentClassName(cartografo.get_CurrentConnectionMap().
   get_ConductingEquipment());
topologyResource=
   cartografo.get_TopoNodeClassName(cartografo.get_CurrentConnectionMap().
   get_TopologicalNode());
…}
```

5. Then, for each equipmentResource and topologyResource their corresponding mapping rule is loaded with the appropriate method from the ModelDesigner class. With the case shown in this guide, the ExcitationSystem component is identified within the static method factory_plant, used when the equipmentResource is a CIM SynchronousMachine class:

```
if (equipmentResource[1].equals("SynchronousMachine"))
{…
    factory_Plant(momachine, machineType, mopin);
}…
/**
* Creates plant object given MachineMap, adds esmap, tgmap and stabmap
* MachinMap can contain ES[0..1], TG[0..1], PSS[0..1]
* @param _momachine
* @param _machineType
* @param _mopin
*/
```

```java
public static void factory_Plant(OpenIPSLMachine _momachine, String _machineType,
                                 MOConnector _mopin)
{…
      switch (excSysData.getKey())
      {
            case "SEXS": mapExcSys= cartografo.create_SEXSModelicaMap(
            excSysData.getValue(),"./res/map/openipsl/controls/es/cim_openipsl_sexs.xml",
            excSysData.getKey());
      …}
      moexcsys = constructor.create_ExcSysComponent(mapExcSys);
…}
```

This guide tries to show the basic code updates that a developer have to do to include a new mapping rule into the architecture. Within the code, there are more details about the parameters and Javadoc description of each method.