

Using oAW's Wombat for Model Transformations

Markus Voelter, voelter@acm.org, www.voelter.de

PRIMARY SPONSORS



SECONDARY SPONSOR



Table of Contents

CONFIGURING ECLIPSE AND OAW.....	3
INSTALLING THE PRE-BUILT TUTORIAL.....	3
CREATING THE PROJECT.....	3
DEFINING THE TRANSFORMATION.....	3
EXAMPLE DATA.....	4
THE WORKFLOW FILE.....	4
RUNNING THE TRANSFORMATION.....	6
COMPLETING THE TRANSFORMATION – MORE FEATURES.....	6

Configuring Eclipse and oAW

For this example, you only need oAW installed. See the *installation* docs for details. You also need to install the *oaw4.demo.emf.datamodel* example project. You should familiarize yourself with that project by reading the *emfHelloWorld* documentation.

Installing the pre-built tutorial

Instead of building the tutorial manually, you can also download the *oaw-samples-emf-4.x.x* package and install the contained Eclipse projects into your workspace (you might want to delete the *ocl* and *atl* demo project for this example). To make the projects compile and run, you have to define the following two Eclipse environment variables:

Variable	... points to
OAW_CORE	oaw4/oaw-core-4.x.x
OAW_LIB	oaw4/oaw-core-4.x.x/lib

Creating the Project

Note that this example here uses the metamodel defined in *oaw4.demo.emf.datamodel*. You can now start by defining a new project. We call it *oaw4.demo.emf.datamodel.wombat*. Make sure it's a Java project. You also have to add a number of dependencies:

- Project Dependency to: *oaw4.demo.emf.datamodel* (defines the metamodel we will work with in the example)
- as usual, all the jars from *oaw4/oaw-core-4.x.x*, also those in it's *lib* subdir.

Defining the transformation

We start by defining the transformation itself. We start by defining a function *copyLibrary* that takes an instance of the metaclass *DataModel*. Since that metaclass is in the default metamodel (see below), we don't have to qualify it by specifying the metamodel identifier before the hashmark.

```
copyLibrary (#DataModel d) :
```

The function in its body creates a new instance of the *Datamodel*; it is populated with the stuff between the curly braces.

```
CREATE (#DataModel: {
```

So, the new *DataModel*'s name is set to the name of the original *DataModel*, ...

```
name <- d.name,
```

... and the set of entities is set to the result of calling *copyEnt* for each member of the set of entities of the passed in *DataModel*, rendered as a collection itself.

```
entity <- { copyEnt(e) | e IN d.entity }
});
```

The *copyEnt* function, for each *Entity* passed in, creates a new Entity with a name similar to the original one, prefixes with X.

```
copyEnt (#Entity e): CREATE(#Entity: {
  name <- "X"+e.name
});
```

This is all for now - a very simple transformation, that for each of the original data model's entities creates a new one, albeit without the entities attributes or references.

Example Data

We need some example data to transform. We use the usual trivial entity, defined in the *src/example.data* file.

```
<?xml version="1.0" encoding="UTF-8"?>
<data:DataModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:data="http://www.openarchitectureware.org/oaw4.demo.emf.datamodel" >
  <entity name="Person">
    <attribute name="name" type="String"/>
    <reference target="//@entity.1" name="autos"/>
  </entity>
  <entity name="Vehicle">
    <attribute name="plate" type="String"/>
  </entity>
</data:DataModel>
```

The Workflow File

The following is the workflow file, called *src/workflow.oaw*.

```
<?xml version="1.0" encoding="windows-1252"?>
<workflow>
  <property file="workflow.properties"/>
```

So, first we load the example model file into a *slot* called model. This is identical to the first step in the oAW code generation example.

```
<component id="xmiParser"
class="org.openarchitectureware.emf.XmiReader">
  <modelFile value="{modelFile}"/>
  <metaModelPackage value="data.DataPackage"/>
  <outputSlot value="model"/>
```

In addition to storing the model (all the contents) into the *model* slot, we also store the first element (the root) of the model into a slot called *rootElement*. This is because later, we need the “root” element (here: an instance of *DataModel*) as an input for the transformation.

```
<outputFirstElementSlot value="rootElement"/>
</component>
```

What follows is the configuration of the Wombat engine. Here it becomes somewhat involved. We define a workflow component that uses the *WombatWorkflowComponent*

```
<component id="transform"
class="org.openarchitectureware.wombat.WombatTransformation">
```

The first parameter we specify is the metamodel. We define a metamodel called *meta* that of type *EmfMetaModel* that uses the *data.DataPackage*. It is necessary to specify which metamodel we use (here: *EmfMetaModel*) since Wombat can also transform metamodels based on other meta metamodels (such as oAW Classic, MDR, etc). Note that this is a simple special case with only one metamodel, typically there are several of them.

```
<metaModel name="meta" scope="data.DataPackage"
implClass="org.openarchitectureware.wombat.adapter.impl.EmfMetaModel"
/>
```

We then define one of our two models. There is a model we call *source* that is an instance of the metamodel *meta* we just declared above. It is an instance of the *EmfTransformableModel* model implementation. Note that this has to be specified for the same reason as above: we can also transform other kinds of models. This model is initialized from the slot “rootElement”, registering all elements that are navigable from the element that is stored there.

```
<model name="source" metamodel="meta" implClass="
org.openarchitectureware.wombat.adapter.impl.EmfTransformableModel">
  <initslot value="rootElement"/>
</model>
```

The other model is called *dest*, and since we use the same metamodel for input and output (we basically copy the model), the declarations are the same as for *source*. Since this model is intended as the target of the transformation, we do not initialize it with elements from a slot.

```
<model name="dest" metamodel="meta" implClass="
org.openarchitectureware.wombat.adapter.impl.EmfTransformableModel"/>
```

We then declare that the *meta* metamodel is the default metamodel. So whenever we do not qualify a metaclass in the transformation, it is assumed to be part of that metamodel (that is why we can write *#DataModel* and not *meta#DataModel* in the transformation script).

```
<defaultMetaModel value="meta" />
```

Then we define the default model. Newly created elements are put into this model.

```
<defaultModel value="dest" />
```

We then declare which file contains the transformation we are about to start.

```
<script value="src/demo.wbt" />
```

Finally, we specify that the transformer should run the *copyLibrary* function in the script, taking the contents of the *rootElement* slot as an input and storing the result in the *newModel* slot.

```
<doTransform targetSlot="newModel"
  execute="copyLibrary (slot:rootElement)" />
</component>
```

That's all for the transformation. To see what we actually did, we subsequently write the content of the *newModel* slot out to a file.

```
<component id="xmiWriter"
  class="org.openarchitectureware.emf.XmiWriter">
  <inputSlot value="newModel"/>
  <modelFile value="{newModelFile}"/>
</component>
</workflow>
```

To make the workflow work, we also have to define the properties:

```
modelFile=example.data
newModelFile=exampleChanged.data
workspaceRoot=L:/workspace-oaw4-doku
```

Make sure you adapt the workspace root accordingly...!

Running the transformation

You run the transformation in the usual way; you just start the workflow file. As a result, in the root of the *oaw4.demo.emf.datamodel.wombat* project you should get the following XML in a file called *exampleChanged.data*.

```
<?xml version="1.0" encoding="ASCII"?>
<data:DataModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:data="http://www.openarchitectureware.org/oaw4.demo.emf.datamodel">
  <entity name="XVehicle"/>
  <entity name="XPerson"/>
</data:DataModel>
```

Completing the transformation – more Features

The above example copied only the entities themselves. Attributes and references were ignored. Let's change this now. Let's first look at copying attributes. Nothing special so far, the algorithm to copy attributes is basically the same as for the entities themselves.

```
copyEnt (#Entity e): CREATE (#Entity: {
```

```

    name <- "X"+e.name,
    attribute <- { copyAttr(a) | a IN e.attribute }
});

copyAttr (#Attribute a): CREATE(#Attribute: {
    name <- "X"+a.name
});

```

It becomes much more interesting when we transform the references. Here's the problem: if, for each reference of the old entity, I want to create a new reference in the new entity, I have to „connect“ objects that I am just about to create when iterating over the entities. It might happen that the target of an entity reference is not yet created because it's the 5th object in the list and we are just „wiring“ the 3rd one. So, typically, what one does is do two passes:

- the first pass of the transformation creates all entities
- the second pass creates the references; at this point, all entities are already created and thus references can be made between them.

This approach does have its problems, though. First of all, it's more effort since you have to run two passes. Second, you must be able to identify the created entities somehow, e.g. by name, in order to refind them in the second pass.

Using Wombat, this is much simpler. Take a look at the following:

```

copyEnt (#Entity e): CREATE(#Entity: {
    name <- "X"+e.name,
    reference <- { copyRef(r) | r IN e.reference } // line 3
});

copyRef (#EntityReference r): CREATE(#EntityReference: {
    name <- "X"+r.name,
    target <- copyEnt(r.target) // line 8
});

```

So, in line three we assign the new references to be the copy of the old references. So far so good. In the *copyRef* function we have a problem, though. To be able to assign the target of the reference (line 8) we need to be able to find the entity that has been created for the original target. To do this, we simply call the *copyEnt* operation for the target object again! And no, this will not result in infinite loops – Wombat takes care of this.

About our Sponsors

itemis GmbH & Co. KG is an independent IT service company with an emphasis on consulting, coaching, and software development. Every single itemis expert provides many years of project experience and widespread knowledge about all object oriented and component based software development issues - especially in the field of model driven software development.

b+m is the founder of the openArchitectureWare project. The software was originally developed within the scope of many successful projects. b+m opened the software to the community in late 2003. All of the paradigms of Model-Driven Software Development including Product Line Engineering and not only the generator framework have become a key concept for product and customer specific development at b+m. b+m customers can make use of long time experience and substantial know-how in that field. Located at the company headquarters in Melsdorf/Kiel and at its subsidiaries in Berlin, Cottbus, Hamburg, Hanover and Kiel the b+m staff of 205 provides practical solutions for customized business applications, business process optimization and comprehensive architecture, project and quality management.

oose Innovative Informatik GmbH offers coaching, consulting and training in all themes about software engineering. The main focus of their activities are software architecture, requirements engineering and project-management. oose have first-hand information and experience, because our staff take actively part with others in actual trends, standards and innovations. Our staff support this and pass their know-how regularly on by writing and publishing books or being speaker at conferences, etc. Within the OMG oose collaborate actively on the specifications of the UML and also the SysML.

MID Enterprise Software Solutions GmbH is a leading supplier of optimized tool environments for standardsbased and model-centric software development as well as business process modeling. This includes professional tool consulting and tool components to build a complete tool environment using the best techniques and tool modules available - Architectural and Operational Excellence. With innovatorAOX, MID provides a holistic standard tool environment for object- and function-oriented software development as well as business process and data modeling to help its customers establish highly efficient processes and tool environments for software production, The unique and seamless integration of business process modeling into the development process ensures an unprecedented level of convergence of business requirements and implemented IT systems. Project members from all departments speak the same language and all requirements are clearly described.