Validating Ecore models using oAW Workflow and OCL

Peter Friese
Gentleware AG
Ludwigstraße 12
20357 Hamburg, Germany
peter.friese@gentleware.com

Bernd Kolb
Kolbware
Franz-Marc Straße 35
89520 Heidenheim, Germany
b.kolb@kolbware.de

ABSTRACT
This paper describes how to integrate the Eclipse OCL framework with the openArchitectureWare workflow in order to use OCL for model validation.

Keywords
Eclipse Summit Europe 2007, Eclipse, OCL, model, validation, workflow, openArchitectureWare

1. INTRODUCTION
During the last few years, more and more projects and companies have adopted modeling technologies to drive their software development process. Having started mostly as a way to describe and document software, models are increasingly being used to generate software.

Having a look at the Eclipse Modeling Project (`http://www.eclipse.org/modeling/`), it can be seen that there is a large number of modeling related projects at Eclipse alone.

However, many of these projects are islands and are not integrated with each other.

In this paper, we propose to use the openArchitectureWare workflow engine to act as an integration layer between various projects. As a sample, we show how to integrate the Eclipse OCL component (`http://www.eclipse.org/modeling/mdt/?project=ocl#ocl`) with openArchitectureWare.

2. OPENARCHITECTUREWARE (OAW)
openArchitectureWare (oAW) is a modular MDSD framework that enables software engineers to create model driven code generators based on an array of building blocks.

Although oAW is written in Java, it supports any target platform, thus enabling software engineers to build code generators for J2EE/JEE, .NET, automotive platforms, microcontrollers and so forth.

oAW can operate on arbitrary metamodels, such as UML, UML2, EMF/Ecore, XML or even JavaBeans-based metamodels.

Models based on these metamodels can be validated using the OCL-inspired *Check* language.

oAW allows users to transform models using both model to model (M2M) and model to code (M2T) transformations. Model to model transformations are described using the *Xtend* language, while model to text transformations are decribed in templates written in *Xpand*.

All languages are fully polymorphic and adapt to the respective metamodel.

To facilitate work with the various languages, configuration files and workflows, oAW provides a tight integration with Eclipse. Wizards, cheat sheets, online help and Welcome page contributions help users getting started with openArchitectureWare quickly. Editors for the various languages feature syntax coloring, code completion, code outlining and hyperlink navigation.

An oAW-based code generator consists of a user-supplied metamodel, a collection of metamodel-tied templates and possibly a collection of model to model transformations. Last, but not least, operation of the code generator is controlled using a workflow that is described in XML and will be executed by the oAW workflow engine.

3. OAW WORKFLOW
The oAW workflow engine is a declaratively configured generator engine that is configured using XML. Workflows are made up of workflow components which are POJOs that conform to a oAW-supplied base class.

oAW sports a WTP-based XML editor which provides introspection-driven code completion, thus making it easy to support custom-written workflow components.

Workflow components delivered with openArchitectureWare include serveral model readers (for various metamodels such as UML and Ecore), a model validator based on the oAW-related *Check* language and a template engine for the *Xpand* language.

Custom workflow components can be developed based on a

set of base classes such as *WorkflowComponent*, *Component*, *SubComponent* and *CompositeComponent* .

```
<workflow>
    <component class="my.first.WorkflowComponent">
        <aProp value="test"/>
    </component>
    <component class="my.second.WorkflowComponent">
        <anotherProp value="test2"/>
    </component>
    <component class="my.third.WorkflowComponent">
        <prop value="test"/>
    </component>
</workflow>
```

Workflows can be nested, thus enabling openArchitetcureWare users to create re-useable generator components (also known as "cartridges"):

```
<workflow>
    <cartridge
        file="compiler.oaw"
        srcDir="_test.tmp"
        targetDir="src-gen-test"
        packagePrefix="testpkg.pkg"
    />
</workflow>
```

4. THE OBJECT CONSTRAINT LANGUAGE (OCL)

The Object Constraint Language (OCL) is a formal language that enables modelers to specify expressions and constraints on object-oriented models. Expressions can be used to specify values or to select them from a model instance. Constraints are restrictions of (part of) an object-oriented model.

OCL is part of the UML specification of the OMG and has been specified by the OMG in [2].

Although the OMG envisions OCL as the constraint language for UML-based models, it can be used in other contexts as well.

In this paper we will show how to use OCL to ensure models meet certain constraints.

As code generators rely on formal and precise models, it is essential to ensure models do not contain ambiguous information.

Using metamodels is a corner stone in ensuring formal models. However, some constraints cannot be expressed in the metamodel due to limitations of the metametamodel being used.

There are two countermeasures we can take to avoid unprecise models: either the modeling tool must ensure the user cannot create illegal models (e.g. by disallowing to draw illegal dependencies) or the generator must validate the model before processing it.

Since most modeling tools miserably fail at ensuring the user creates valid models, it is essential the code generator performs model validation before further processing. This is where our OCL integration comes into play

5. INTEGRATING OAW AND OCL

To integrate the OCL component and openArchitectureWare, we need to write a workflow component. This workflow component will be invoked during the openArchitectureWare workflow run. The OCL workflow component will take an OCL constraint expression as a parameter. When executed, it will evaluate this expression against the model.

Workflow components support properties. In order to support simple properties, a workflow component has to supply a `set<propertyname>` method. If you want to set a list of values you need to supply a `add<propertyname>` method.

Workflow components have to communicate among each other. For example, if an XMIReader component reads a model that a constraint checker component wants to check, the model must be passed from the reader to the checker. The way this happens is as follows. In the invoke operation, a workflow component has access to the so-called workflow context. This context contains any number of named slots. In order to communicate, two components agree on a slot name, the first component puts an object into that slot and the second component takes it from there. Basically, slots are named variables global to the workflow. The slot names are configured from the workflow file. Here is an example:

```
<?xml version="1.0" encoding="windows-1252"?>
<workflow>
    <property file="workflow.properties"/>
    <component id="xmiParser"
        class="org.openarchitectureware.emf.XmiReader">
        <outputSlot value="model"/>
    </component>
    <component
        id="checker"
        class="datamodel.generator.Checker">
        <modelSlot value="model"/>
    </component>
</workflow>
```

If some error occurrs during processing, a workflow component can use the *issues* API to communicate this problem to the surrounding workflow. This is used in our implemention to return the result of the evaluation to the workflow:

```
boolean result =
  eval(issues, context, ocl, helper, exp);
if (!result) {
  issues.addError(exp + " evaluated to false");
}
```

Here is the source code for the OCL workflow component:

```
public class OCLWorkflowComponent
```

```
  extends WorkflowComponentWithModelSlot {

  private IOCLFactory<Object> oclFactory;

  private boolean isEMF = true;

private String exp;

  public void invoke(WorkflowContext ctx,
    ProgressMonitor monitor, Issues issues) {
    Object object = ctx.get(getModelSlot());
    EObject context;
    if (object instanceof EList) {
      EList l = (EList) object;
      context = (EObject) l.get(0);
    } else {
      context = (EObject) object;
    }
    if (isEMF) {
      oclFactory = new EcoreOCLFactory(context);
    } else {
      oclFactory = new UMLOCLFactory(context);
    }

    OCL<?, Object, ?, ?, ?, ?, ?, ?, ?,
      Constraint, ?, ?> ocl = oclFactory.createOCL();
    OCLHelper<Object, ?, ?, Constraint> helper =
      ocl.createOCLHelper();

    boolean result =
      eval(issues, context, ocl, helper, exp);
    if (!result) {
      issues.addError(exp + " evaluated to false");
    }
  }

  private boolean eval(Issues issues, EObject context,
    OCL ocl, OCLHelper helper, String exp) {
    // set our helper's context classifier to
    // parse against it
    helper.setContext(
      oclFactory.getContextClassifier(context));

    try {
      Constraint parsed =
        (Constraint) helper.createInvariant(exp);

      return ocl.check(context, parsed);

    } catch (Exception e) {
      issues.addError("Exception while evaluating " +
        exp + ": " + e.getMessage(), e);
    }
    return false;
  }

  public void setIsEMF(boolean isEMF) {
    this.isEMF = isEMF;
  }

  public void setOCLExpression(String exp) {
    this.exp = exp;
  }
}
```

Here is a sample workflow file that loads a model and evaluates two different OCL expressions against that model:

```
<workflow>

  <component file='org/example/dsl/parser/Parser.oaw'>
    <modelFile value='model/xxx.dsl' />
    <outputSlot value='test' />
  </component>

  <component
    class="...adapter.ocl.OCLWorkflowComponent">
    <OCLExpression value="contents -> size() = 1" />
    <modelSlot value="test" />
    <isEMF value="true" />
  </component>


  <!-- will fail -->
  <component
    class="...adapter.ocl.OCLWorkflowComponent">
    <OCLExpression value="name = 'lalalala'" />
    <modelSlot value="test" />
    <isEMF value="true" />
  </component>

</workflow>
```

6. CONCLUSIONS

In this paper, we have motivated the reasons for validating models and have shown how to integrate the Eclipse OCL engine and openArchitectureWare by means of writing a custom openArchitectureWare workflow component.

As can be seen, the openArchitectureWare workflow engine can be used as an integration layer and thus helps building highly customized code generators, which is major benefit of openArchitectureWare.

7. REFERENCES

[1] oAW Developers. *openArchitectureWare Documentation*. openArchitectureWare.org, 2007.

[2] *Object Constraint Language Specification*. OMG, May 2006.