Introduction to openArchitectureWare 4.1.2

Arno Haase
Independent Consultant
Langemarckstr. 16
53227 Bonn
+49 228 9654443
Arno.Haase@Haase-Consulting.com

Markus Völter
Independent Consultant
Grabenstrasse 4
73033 Goeppingen
+49 171 8601869
voelter@acm.org

Sven Efftinge
Independent Consultant
Am Sophienhof 33
24941 Flensburg
+49 461 9992330
sven@efftinge.de

Bernd Kolb
Independent Consultant
Franz-Marc-Str. 35
89520 Heidenheim
+49 163 7321605
b.kolb@kolbware.de

ABSTRACT

This paper introduces openArchitectureWare 4.1.x based on a common, predefined example. The example has been defined as part of the MDD TIF 07. The idea of this workshop is to showcase various MDD tools and compare their approaches to solving the same example problem. The paper describes using the openArchitectureWare tool chain for defining meta models, code generator, model transformations as well as for building graphical and textual editors.

Keywords

model-driven software development, domain-specific languages, code generation, metamodeling, modeling, templates

1. INTRODUCTION

OpenArchitectureware is an open source tool set for defining and processing models. It provides building blocks for the entire tool chain. It focuses on modularity, making the building blocks usable in other contexts and – more importantly – making it possible to integrate tools that originate outside of oAW.

OpenArchitectureware is currently in the process of moving to Eclipse. More specifically, the workflow engine is becoming a project in its own right while the generator components will be merged into the model-to-text (M2T) project. For simplicity's sake, this paper still uses the term openArchitectureware to denote the collection of tools, this however should not be construed to oppose the integration with Eclipse and the dimishing of the oAW brand.

2. Overview

The approach we took to illustrate oAW's features on the interactive TV example focuses on defining a metamodel to describe interactive content (including its structure), building editors for the corresponding models, checking these models' validity, and generating code (the XML files) from them.

The whole process is tied together using the oAW workflow engine, a script language that specializes in the integration of MDSD building blocks into an actual generator.

The remainder of this paper will look at the different components of oAW with their characteristics, and how they interact to create the interactive TV application.

One of the strengths of oAW is Xtext, a framework for generating a textual modeling language (including metamodel and an Eclipse based IDE with syntax highlighting, context sensitivity etc.) from a simple grammar description. Therefore, we will describe an Xtext-based textual DSL in addition to the graphical DSL required.

3. Metamodel definition

The generator described for this example is entirely based on EMF[1] (Eclipse Modeling Framework), a widely used file and model format. While providing a rich set of useful features, it is by no means the only representation that is used for models, and oAW accesses models through a facade so that models of arbitrary formats can be used – even at the same time, e.g. to transform from one to another or to combine them.

All subsequent steps require the presence of a metamodel as an ecore file, and the Eclipse toolscape provides a constantly growing number of ways to define these. Any of them can be used.

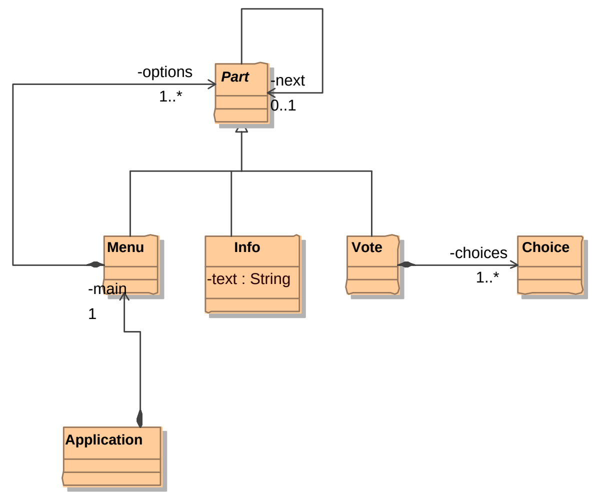
openArchitectureWare however provides a generator called uml2ecore that transforms a UML model – more specifically, an EMF UML2[2] export of a UML model, which is supported as a truly standardized export format by an increasing number of UML tools – into an ecore model.

Using UML class diagrams to define a metamodel allows the metamodel definition to be split into several diagrams, each of them presenting a different partition of the metamodel, and thus providing benefit for very large metamodels.

The following diagram shows the metamodel on which our subsequent tooling is based:

[1] http://www.eclipse.org/emf

[2] http://www.eclipse.org/uml2



The underlying assumption is that all stakeholders will work on instances of this metamodel.

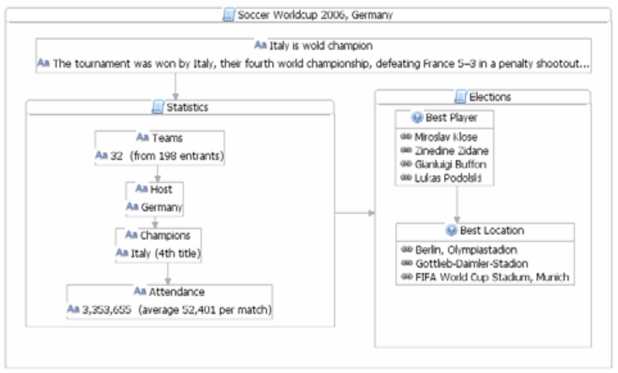
4. Graphical model editor

In order to provide a graphical editor for models based on the above described metamodel, we created an editor using the Eclipse Graphical Modeling Framework[3] (GMF).

Another approach would have been to use a given graphical editor for a more or less similar metamodel – e.g. using a UML tool with a profile, or Microsoft Visio with guidance as to its usage – and use a model transformation to convert the resulting models to our metamodel (see. section 6). However we chose to use GMF in order to illustrate the integration with the Eclipse platform.

GMF is designed to generate a graphical editor for EMF Meta-metamodels. In order to do this, the developer has to create a graphical description of the editor, a tooling description of the editor and last, but not least create a mapping where the tooling definition is bound to the graphical definition and the metamodel.

The generated editor can be customized using different extension-points and template AOP provided by the XPand template engine. The integration of different check languages is possible.



3 http://www.eclipse.org/gmf

5. Textual model editor

While the requirements specify the use of a graphical editor, we often find textual languages more useful, especially in (but by no means restricted to) project settings where technical people use them to specify parts of a system.

To illustrate this feature, we redundantly created a textual DSL with a textual editor for the itv metamodel. It is based on the Xtext framework and requires a grammar definition in a simplified EBNF format as its only input. The following listing shows this grammar. Note that xText comes with a custom editor that provides code completion, syntax highlighting and the like.

```
ApplicationAST :
      name=STRING
      mainMenu = MenuAST;

PartAST :
      MenuAST | InfoAST | VoteAST;

MenuAST :
      "Menu" name=STRING "{"
              (parts+=PartAST)*
      "}";

InfoAST :
      name=STRING ":" text=STRING;

VoteAST :
      "Vote" name=STRING
      (choices+=ChoiceAST)*;

ChoiceAST :
      "-" name=STRING;
```

Taking this grammar as input, Xtext generates an editor with syntax highlighting, code completion, folding support, cross referencing, an outline view and other features expected from a modern IDE.

The following listing shows a simple definition of a sample itv application:

```
"World Cup"

Menu "choice" {

  Vote "all times favorite player" {
    - "Schumacher"
    - "Beckenbauer"
    - "Rummenigge"
  }

  Menu "player info" {
        "Schumacher": "Goalie"
        "Beckenbauer":  "honorary  president  of
Bayern München"
   }

   "Background Information":
      "Football is about winning. "

}
```

In addition to the editor, Xtext generates a technical metamodel that corresponds to the AST of the grammar. This metamodel is an ecore file.

This metamodel is structurally very similar to the metamodel described in section 3, but it must be noted that it is not the same.

Xtext currently does not take a given metamodel as input, making a model transformation to the "real" metamodel necessary.

6. Model transformation

Xtend is one of the oAW languages. It is a functional language suited for a number of uses.

One of these uses is model transformation (i.e. creating a new model based on one or more existing models), others are model modification (i.e. adding to, removing from or just modifying an existing model) or metamodel extension (see section 9).

We illustrate its model transformation capabilities by showing the Xtend script that transforms from the textual DSL to the "full" DSL that is used by the GMF editor and on which the code geheration is based:

```
import itv;

extension
org::openarchitectureware::util::stdlib::io;

create Application
    transform (ApplicationAST ast):
  setName (ast.name) ->
  setMain (ast.mainMenu.transform ());

transform (PartAST ast):
  syserr (ast, "unsupported kind of Part");

create Menu transform (MenuAST ast):
    setName (ast.name) ->
    setOptions (ast.parts.transform ());

create Info transform (InfoAST ast):
    setName (ast.name) ->
    setText (ast.text);

create Vote transform (VoteAST ast):
    setName (ast.name) ->
    setChoices (ast.choices.transform ());

create Choice transform (ChoiceAST ast)
  setName (ast.name);
```

The first line declares implicit namespace resolution for the rest of the file – whenever a type is resolved, it is looked for in the "itv::" namespace which incidentally contains both the source and destination metamodel elements.

The extension statement in the second line includes another extension file, in this case a standard library that provides the syserr extension we use for error reporting.

The actual "extensions" (oAW terminology for functions defined in an Xtend file) consist of an optional return type followed by the extension name and a parameter list. After this signature part, there is a single expression which defines the value of the extension when it is evaluated.

The "->" operator is the oAW way to control execution order of side effects. It chains several expressions, evaluation them from left to right and returning the value of the last of them.

The "create" keyword turns an extension into a create extension, adding special features that are very useful for controling model structure when creating models. The first of these is that it implicitly creates a new model element of the specified type, binding it to "this" so that all feature accesses access it by default.

Secondly, it caches the results of each execution. Any subsequent call to the same extension with the same parameters will result in the same object being returned rather than a new object being created. This allows elegant control over graph structure: Whenever a model element is to be referenced from several places in the model, the transformation can "create" it from each of the referencing places – only the first such "creation" will actually create a new element, and all subsequent "creations" will automatically result in the already created instance.

Thirdly, the newly created element is bound to the cache before the initialization is executed. This deferred initialization deals gracefully with cyclic object dependencies: If object A is created and initialized to reference newly created object B which in turn references A, naively eager initialization would result in endless recursion while deferred initialization as described yields the intended results.

Invocation of extensions is polymorphic in all parameters. The concrete extension to be invoked is chosen at runtime based on the object types of all its parameters. In case of ambiguity, an exception is thrown at runtime.

The invocation of an extension on a collection – such as the invocation of transform() on ast.parts in the MenuAST transformation – serves as a shorthand notation for creating a collection of the results of invoking the extension on all elements of the original collection.

oAW has special tools to support model weaving – i.e. enhancing one model based on an aspect model defining pointcuts to define which parts it applies to – and model merging – i.e. joining several models that conform to the same metamodel into a single larger model. Both features are however not used for the interactive tv example.

7. Model validation

Models can be semantically invalid even if they are syntactically correct. Rather than having every component operating on a model guard against these semantic constraints in an ad hoc manner – or, worse, ignoring the issue and making implicit assumptions – oAW provides a DSL to specify such constraints and checking them explicitly in a separate step of the tool chain.

This constraint validation language is called Check, and the following listing shows two such constraints for an interactive tv application:

```
import itv;

context Vote WARNING
      "Must have at least two choices..." :
   this.choices.size>=2;

context Vote ERROR
      "Duplicate choice '"+name+"'" :
  eContainer.eContents.typeSelect(VoteAST).
     select(e|e.name==name).size==1;

context Part ERROR
      "names must have three chars or more":
  name.length >= 3;
```

Every constraint declares the type of model elements it applies to, followed by either ERROR or WARNING to indicate the severity of a failed validation.

The next part of a check declaration is an expression that defines the error message in case of a failure. This message is intended to describe the problem at the metamodel level of abstraction, enabling modelers to understand it and address the course. This is aligned with our conviction that models are first class source code and modelers basically have the same role as programmers.

Since the messages are not just string literals but full fledged expressions, it is possible to include attributes of the checked elements in the message, making for more informative and readable texts.

The actual specification of the constraint is a predicate that is written as an invariant of the metatype, i.e. it must evaluate as “true” for the model element to be valid.

These predicates use the same expression language as the Xtend language, which is based on OCL. The “Duplicate choice” check illustrates the powerful collection operations it provides.

The “typeSelect” operation selects all elements of a collection that conform to a given type. This operation is not part of standard OCL but a custom enhancement we added since selecting elements based on their type is a frequent operation when dealing with models.

It is possible to reference extensions in checks so that common functionality can be factored out and reused across different checks or even several of the oAW languages.

It is possible to specify a supertype as the context of a check, in which case it is checked against all elements of that type or any of its subtypes. The last check is an example of this. It checks the length of the name for all kinds of Parts, i.e. Menu, Info and Vote elements.

8. Code Generation

After the input models are transformed, merged and validated – not necessarily in this sequence – they are finally fit to be used as input for code generation. For this purpose, oAW provides the Xpand template language. As any of the other tools in the tool chain, the generator can be exchanged for any other generator component.

Xpand is a dedicated template language in that it is optimized for producing textual output. The following listing shows the generator for the interactive tv application.

```
«IMPORT itv»

«EXTENSION
org::openarchitectureware::util::stdlib::io»

«DEFINE root FOR Application»
  «FILE name+".xml" xml»
    <?xml version="1.0" encoding="utf-8">
      <TVApp name="«name»">
         «EXPAND part FOR main»
      </TVApp>
  «ENDFILE»
«ENDDEFINE»

«DEFINE part FOR Part»
  «syserr (this, “undefined kind of Part”)»
«ENDDEFINE»

«DEFINE part FOR Menu»
  <Menu name="«name»">
    «EXPAND part FOREACH options»
  </Menu>
«ENDDEFINE»

«DEFINE part FOR Info»
  <Text name="«name»">
    «text»
  </Text>
«ENDDEFINE»

«DEFINE part FOR Vote»
  <Vote name="«name»">
    «FOREACH choices AS c»
      <Choice name="«c.name»"/>
    «ENDFOREACH»
  </Vote>
«ENDDEFINE»
```

The first syntactic peculiarity of Xpand files that meets the eye is the french quotation marks « and ». They serve to distinguish between the output that becomes part of the output 1:1, and escaped control code that is interpreted. The french quotation marks were chosen because they very rarely appear in generated output, which makes for far more readable templates.

Xpand files consist of one or more DEFINEs which are the template building blocks. Every such DEFINE has one primary parameter the type of which stands after the FOR and which is bound to “this”, and any number of additional “regular” parameters (for which there is no example in this very simple template file).

In our case, the “root FOR Application” template serves as the entry point since it is called from the workflow file (see section 10).

The outermost statement in this template definition is the FILE statement which defines a scope for a file into which output is written. The name of the file is given by an arbitrary expression that can – and usually does – contain references to model properties. Xpand intentionally moved control over the output files into the template language (rather than making it part of the call interface for the generator as many other template languages do) because we found that path and file names often have significant dependencies on model properties. This also addresses the problem of controlling which files to generate when – FILE statements can be executed conditionally.

In order to include the value of model properties in the output, these must be escaped using the french quotation marks, as in <TVApp name="«name»">. This fragment inserts the value of the application's name element in double quotes as the name attribute of a TVApp tag.

Templates can call other templates as exemplified by the “EXPAND part FOR main” call. this causes the output of the called template to be inserted in the place of the call. If a template is to be expanded for all elements in a collection rather than a single element, FOREACH must be used instead of FOR.

If there are several templtes of the same name, a call is resolved polymorphically based on the object type of the parameters. This allows elegant template code without explicit conditional code – the expansion of “part” for the options stored in a menu chooses the appropriate template depending on whether a given part is a Menu, an Info or a Vote.

There is also support for loops without calling templates. This need is catered to by the FOREACH element.

The EXTENSION declaration in the header of the file makes the contents of the standard library for io available to the template. It is used to report a runtime error in the "part" template for Part.

The template file for the interactive tv example is laid out for readability of the template definitions. This results in output that is badly layouted, mostly due to a big number of empty lines and counterintuitive indentation.

This trade-off between template and output readability is typical of template languages, and the oAW approach is to go for template readability and apply a code beautifier afterwards. There is a hook in the workflow call of the generator (see section 10).

9. Metamodel extension

The Xtend language appeared in three contexts so far, first as a means for model transformation and then as part of constraint checking and templates. On a more general note, it serves to dynamically add features to metamodel elements.

In principle, it is possible to add all functionality needed by any metamodel type to the type directly – e.g. as operations. The interactive tv application is so simple that it does not really require this, but it frequently happens in more complex metamodels.

Let us assume a metamodel that represents type definitions in some kind of namespace. Let us further assume we want to generate Java classes from these definitions. Then we need a concept of the fully qualified name of the Java type, which basically is a derived attribute based on the names of the type and the surrounding namespaces.

It is usually a bad idea to implement this logic in the templates because that reduces readability and, worse, enforces copy and paste programming and quickly results in a maintenance nightmare when consistent changes of all occurrences are required.

Another approach would be to implement it as an operation "getJavaFQN()" on the metaclass, assuming the metamodel type is actually represented by a class. This approach works – and is supported by oAW – but it has two major drawbacks.

Firstly, it works only if the metamodel is represented by classes and these classes are available for customization. This would for example preclude the use of dynamic EMF or Ecore models generated from XSDs.

Secondly, it does not scale well. Let us assume that our effort is part of a company-wide integration effort and we generate not only Java but also C++, Corba IDL and XSDs from the same models. Then the metaclasses would be burdened with helper functionality for every single target platform, bloating them and making them highly volatile.

Because of these issues, oAW introduced the concepts of extenal metaclass extensions using the Xtend language. It allows the definition of additional operations and derived attributes without actually touching the metamodel itself, yet making these extensions available as if they were part of the metamodel definition itself.

The extensions can be loaded selectively, and only those extensions that are loaded for a given scope are present. So a template generating Java code can add the Java specific extensions while a C++ template can ignore these and add the C++ specific extensions instead.

This concept of extending the metamodel is reflected in two alternative mode of invocation of extensions.

The first of these is to treat them as regular functions (as opposed to methods / features of a metamodel type). Using this call syntax, the code to compute the fully qualified name of a type would look as follows:

```
getJavaFQN (type)
```

But oAW integrates extensions into the type system, making them actual features of the first parameter type. That makes the following syntax possible, which has identical semantics to the previous one:

```
type.getJavaFQN ()
```

Depending on the context, one or the other of these options is more intuitive, but both are always available.

10. Integration of the steps

So far this paper looked at the different building blocks of a generator – model editor, validator, transformation engine, and the generator proper. These building blocks expect models as input, process models, and provide models as output.

In order to combine these modular tools into a complete generator, some sort of execution engine is required. The oAW workflow engine provides this functionality.

It executes a workflow file which is an XML definition of a sequence of components. Each component is described by its type – a fully qualified Java class name – and parameters, the instantiation being done by reflection along the lines of a Spring application initialization.

The components are executed in the sequence in which they appear in the workflow definition. They communicate with each other through so-called slots, i.e. entries in a global Map that is controlled by the workflow engine and to which every component has read and write access.

The following listing shows a simple workflow definition for the interactive tv exmple:

```
<workflow>

  <component id="read"
         class="oaw.emf.XmiReader">
    <outputSlot value="model"/>
    <firstElementOnly value="true"/>
    <modelFile value="models/modelFile.xmi"/>
  </component>

  <component class="oaw.check.CheckComponent">
      <metaModel id="mm"
         class="org.openarchitectureware.type.
                     emf.EmfMetaModel">
         <metaModelFile value="itv.ecore"/>
      </metaModel>
    <checkFile value="itvConstraints"/>
    <emfAllChildrenSlot value="model"/>
  </component>

  <component id="gen"
       class="oaw.xpand2.Generator"
       kipOnErrors="true">
    <metaModel idRef="mm"/>
    <outlet path="src-gen"
      fileEncoding="iso-8859-1"/>
    <outlet name="xml" path="src-gen"
            fileEncoding="utf-8" />
      <expand  value="templates::xml::root  FOR
model"/>
    <beautifier
     class="oaw.xpand2.output.XmlBeautifier"/>
  </component>

</workflow>
```

The components in the workflow definition are executed in the order in which they appear. The first component is an XmiReader, it reads an EMF model from the file “models/modelFile.xmi” and stores it in the slot “model”.

The second component validates the contents of the slot “model” based on the constraint definitions in the file “itvConstraints.chk” (the file extension is appended automatically), and the third and last component starts the actual generator.

The Generator component is declared to be skipped if a previous component – in our case, that would be the CheckComponent – reported any errors. That is reasonable behavior since we would not want to attempt generating code based on an invalid model – that is after all what the constraint checks are there for.

We then define two outlets, one for regular source code (which we currently do not use and just included for illustration purposes) with a file encoding of iso-8859-1, and another for xml files with a file encoding of utf-8. Every file statement in an Xpand template (see section 8) can declare which outlet the file should be written to. The outlet also defines the root directory relative to which the FILE path is evaluated – in our case both outlets have “src-gen” as their root directory.

Another feature of the outlets is to specify if files should be overwritten when they are generated – which is the default – or generated only once, skipping the generation altogether if the file already exists. The latter option is useful e.g. for generating default implemenations that are intended for illustration purposes rather than providing useful production code.

The GeneratorComponent also specifies a number of beautifiers which are applied to generated files.

Implementing customized workflow components is quite simple. Parameters are passed in based on reflection, so once the workflow component has setter methods for a property, it is available for configuration in the workflow file. Workflow components need to extend a given abstract class and implement two methods, one to check valid initialization of the component and the other to actually do its job.

The workflow engine supports the concept of cartridges, i.e. workflow files that are invoked by other workflow files. This feature is useful for modularizing larger projects, and for extracting reusable generator fragments.

In the context of such reusable fragments, oAW provides support for Aspect-Oriented Programming (AOP). It is possible to specify template code that replaces (more specifically, is executed “around”) other template code. This feature makes it possible to customize existing templates without having to modify them, which we see as a necessary precondition for having useful reusable templates.

11. Conclusion

We showed that openArchitectureware provides a tool box that supports a large variety of steps in the MDSD tool chain. These building blocks can be combined in a large number of ways, and it is possible to complement or replace them with tools from another origin on a case-by-case basis.

Specifically, oAW provides support for integrating graphical models defined by GMF, and for creating editors for textual DSLs based on a simplified EBNF grammar. The Check language allows declarative checks of model validity, Xpand defines templates, and Xtend both extends metamodels dynamically and can specify model transformations and modifications.

These building blocks are tied together using the oAW workflow engine which serves as a backbone for creating customized generators.