Aspectual Code Generators for Easy Generation of FIXML to OO Mappings TTC 2014 FIXML Case Solution

Steffen Zschaler, Sobhan Yassipour Tehrani
Department of Informatics, King's College London
szschaler@acm.org, sobhan.yassipour_tehrani@kcl.ac.uk

April 30, 2014

Abstract

This paper provides a solution to the TTC 2014 FIXML study case. The case requires the implementation of a relatively straightforward mapping from XML messages in the FIXML format to a set of source files implementing the schema of such a message and, optionally, an instantiation with the data from the message. There is a requirement for producing code in a range of programming languages.

The biggest challenge for transformation design in this study case is the fact that the same tag may occur in multiple places in the FIXML message, but with a different set of attributes. The generator is required to merge all of these occurrences into a single representation in the generated code. We demonstrate how the use of symmetric, language-aware code generators relieves the transformation developer almost entirely from any consideration for this requirement. As a result, the transformation specifications we have written are extremely straightforward and simple. We present generation to Java and C#.

1 Introduction

FIXML is a language used in the financial sector to express financial-transaction information for machine-to-machine communication in electronic trading. For convenience of processing it is useful to implement object-oriented wrappers that are used in end-point systems when reading, constructing, and manipulating FIXML messages. It is possible to introduce new and custom formats for messages and this happens frequently.

The study case asks for implementations of code generators that produce wrapper classes given a specific FIXML message. There are, thus, two parts to the problem posed: 1) to extract the message schema and 2) to generate class code implementing this schema. The case description does, consequently, ask

for the solution to be broken down into two major phases (with an initialisation phase for reading the XML document): 1) extracting the schema into an instance of a programming-language meta-model and 2) generation of source code from the model thus created.

The code-generation phase is almost trivial to implement as it effectively amounts to a textbook case of class-diagram to class-skeleton generation. Schema extraction is a little bit more interesting in that it requires the merging of information from different parts of the XML document: Tags of the same name can occur in different places of the document, but with a different set of attributes and sub-nodes.

In our implementation, there are two design decisions that are worth noting:

1. We use symmetric language-aware aspects [6, 7] in the implementation of our code-generation templates, obviating almost completely the need for any special consideration of the need for merging in schema extraction; and

2. We use a completely target-language independent meta-model of classes and attributes (i.e., of the schema). In fact, because of our use of symmetric aspects, our meta-model does not need to insist on uniqueness of class names and becomes an object model of the FIXML message rather than the extracted schema only. This enables us to easily generate a test method instantiating our generated classes with exactly the data from the given FIXML message.

Our implementation is based on Epsilon [2–5] with our own extensions for symmetric aspects in code generation [6, 7].

The remainder of this paper is structured as follows: We first review some background on symmetric, language-aware code generation in Sect. 2. Section 3 discusses the key points of our solution. Section 4 briefly discusses the results from the test runs required, before Sect. 5 presents metrics for our implementation.

2 Symmetric Aspects for Code Generation

In [6, 7], we have introduced symmetric, language-aware aspects for code-generation templates to enable advanced modularity for code-generation templates. Detailed descriptions are in these papers, but we give a brief summary here to simplify understanding of our solution to the TTC 2014 FIXML case.

Figure 1 shows an overview of the infrastructure for code generation with symmetric aspects. Crucially, results from the interpretation of code-generation templates are not directly written to a file, but are centrally registered against the name of the file they are meant to produce. Later, all texts registered against the same file name are merged before they are finally written to disk.

For the merging step, we use an implementation of superimposition; specifically, FeatureHouse [1]. FeatureHouse comes with our implementation by

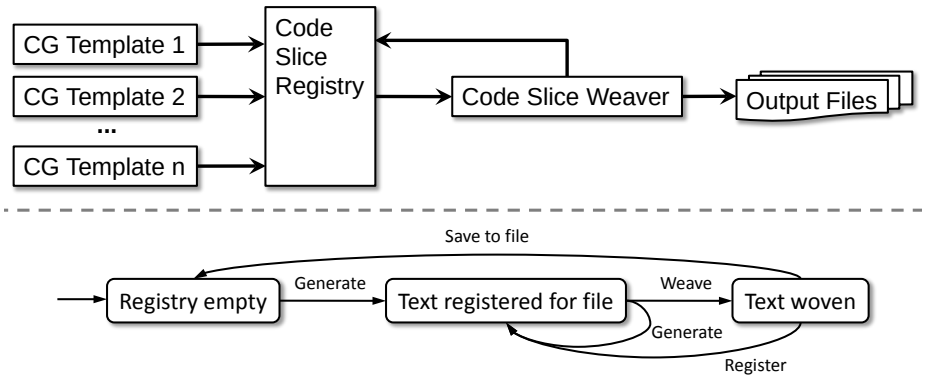

Figure 1: Infrastructure for symmetric, language-aware aspects for code-generation (from [6])

default, but other merging strategies can be implemented and provided. FeatureHouse merges two texts in two steps: First, the texts are parsed using a coarse-grained grammar for the particular language they are written in. The aim is to extract named entities in the code; details of the implementation (e.g., method bodies) are kept as opaque blocks of code. Two such *feature-structure trees* are then combined by merging the contents of nodes of the same name. Where these contents are opaque blocks, FeatureHouse calls out to language-specific semantic merge operators. For example, two Java method bodies are merged by inserting the second in any place where the first mentions the special invocation of '`origin()`'.

As a result, more than one code-generation template can contribute to a given file. If each template is written to be computationally complete, they can be swapped in or out of a transformation workflow completely independently of each other, giving great flexibility for transformation reuse, but also for debugging. Because the templates are standard generation templates (written in EGL [5] in our case), they can alternatively also be run by the standard EGL engine and the result written to disk directly, making it accessible for debugging.

Symmetric language-aware aspects for code generation have been implemented as an extension to EGL and are available from EpsilonLabs.[1]

3 Solution

We first describe the complete solution for generating Java code, before discussing the changes needed for generating C# code.

[1]EpsilonLabs is available at `http://epsilonlabs.googlecode.com/`. The update site for the symmetric aspects for code-generation plugins is `http://epsilonlabs.googlecode.com/svn/trunk/org.eclipse.epsilon.egl.symmetric_ao.updatesite`.

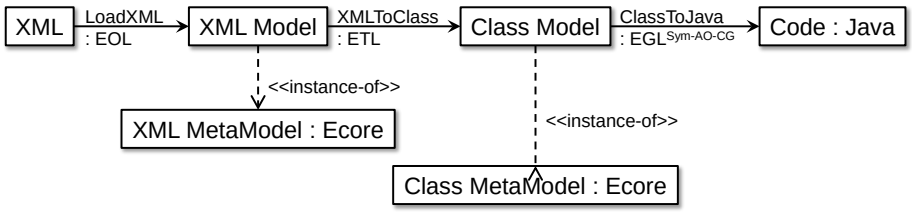

Figure 2: Transformation architecture. Boxes correspond to artefacts (with their language expressed after a colon or using an `<<instance-of>>` relation) and arrows describe transformations

3.1 Transformation to Java

Figure 2 gives an overview of the complete transformation architecture implemented for generating Java code from FIXML messages. In a first step, the XML is parsed and translated into an instance of the XML metamodel defined by the task specification. This model is then translated into a model of classes and attributes, before code is finally generated. In the following, we will discuss each of these steps in some more detail.

3.1.1 LoadXML

The case specification provided a meta-model for XML documents and required that transformation architectures use this as a intermediary storage format for the FIXML message to be processed. We have encoded the given meta-model in Ecore and have written a simple EOL [2] program to parse XML documents into instances of this meta-model. This is simplified by the fact that Epsilon already comes with an XML parser, called a *model driver*, exposing the contents of an XML file to model-management operations through a naming convention [4]. The complete EOL program for `LoadXML` can be found in Listing 1.

3.1.2 XMLToClass

As a next step, we need to extract the message schema from the concrete message given. In our implementation, this amounts to a very straightforward copying of the XML model into a model of classes and their attributes, differentiating between string-typed and class-typed attributes.[2] The resulting model does not actually describe a schema, but represents the actual object structure of the message given. The only change made at this step is for the transformation to ensure that attribute names are unique *within an object* (although not necessarily between different objects of the same class). This will work together with

[2]Here we would also do any type analysis if we were providing a solution for that additional requirement.

Listing 1: LoadXML implementation in EOL

```
generateFor (XMLDoc.root);

operation  generateFor (e : Element) : XML!XMLNode {
  var node : XML!XMLNode = new XML!XMLNode;
  node.tag = e.tagName;

  if (e.getAttributes().length > 0) {
    for (idx in Sequence{1..e.getAttributes().length}) {
      var attr = e.getAttributes().item (idx - 1);

      var xmlAttr : XML!XMLAttribute = new XML!XMLAttribute;
      node.attributes = node.attributes->including (xmlAttr);
      xmlAttr.name = attr.nodeName;
      xmlAttr.value = attr.nodeValue;
    }
  }

  for (elt in e.children) {
    node.subnodes = node.subnodes
                        ->including (generateFor (elt));
  }

  return node;
}
```

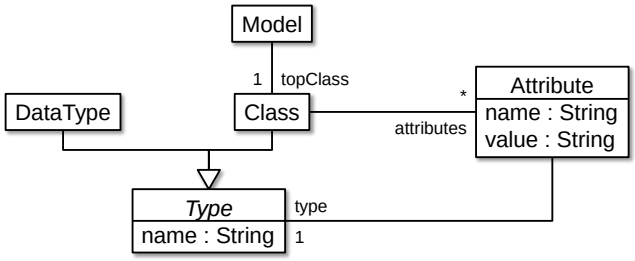

Figure 3: Class meta-model diagram

name-based merging to ensure generation of minimal code. Additionally, we also keep track of the top-level element in the object structure.

This transformation is written in ETL [3] and produces instances of the meta-model shown in Fig. 3. The code of the transformation is shown in Listings 2 and 3.

It is worth noting that the transformation does not actually merge different occurrences of the a tag of the same name into one class definition in the class model. As a result, the model produced may contain multiple classes of the same name. Their definitions will be merged automatically once code has been generated.

3.1.3 ClassToJava

The final step of the transformation chain produces Java code from the class model. The template is written in EGL and is extremely straightforward. It consists of a controller template (*cf.* Listing 4) that instantiates a second template for every class in the model. That second template (*cf.* Listing 5) simply generates a class skeleton including all attributes and references as well as a default constructor and a constructor for the attributes and references found.

Note that the name of the file to generate is derived from the name of the class in Listing 4. This may lead to multiple versions of the same file being generated. However, the build workflow shown in Listing 6 invokes the template using `eglRegister` rather than `egl`, thus registering all generated code in the central registry. Only the call to `eglMerge` combines all code produced for a particular class. Because elements of the same name are unified in the merging process, the requirement of the study case is implicitly satisfied.

Because of this ability to merge different generated code, we were also able to provide a modular definition of an additional feature, namely the generation of a main method showing how to instantiate the classes created to represent the FIXML message from which it was generated. To this end, we defined two separate code-generation templates: the first (*cf.* Listing 7) is a controller template identifying the top class in a given class model and invoking the second template for this class. The second template (*cf.* Listing 8) then generates

Listing 2: XMLtoClass implementation in ETL

```
pre {
  var STRING_TYPE : Classes!DataType = new Classes!DataType;
  STRING_TYPE.name = "String";
}

rule NodeToClass
        transform s : XML!XMLNode
        to t : Classes!Class {

  t.name = s.tag;

  var uniqueID = new Map;
  for (attr in s.attributes) {
    var newAttr = attr.equivalent();
    newAttr.name = newAttr.name.getUniqueVersion(uniqueID);
    t.attributes = t.attributes->including (newAttr);
  }

  for (elt in s.subnodes) {
    var attr : Classes!Attribute = new Classes!Attribute;
    t.attributes = t.attributes->including(attr);
    attr.name = elt.tag.getUniqueVersion(uniqueID);
    attr.type ::= elt;
  }
}

rule AttrToAttr
     transform s : XML!XMLAttribute
     to t : Classes!Attribute {

  t.name = s.name;
  t.value = s.value;

  t.type = STRING_TYPE;
}

post  {
  var mdl : Classes!Model = new Classes!Model;
  mdl.topClass ::= getTopNode();
}
```

Listing 3: XMLtoClass implementation in ETL (ctd.)

```
operation getTopNode() : XML!XMLNode {
  var resultSet = XML!XMLNode.all;
  for (node in XML!XMLNode.all) {
    resultSet = resultSet->excludingAll(node.subnodes);
  }

  return resultSet.random();
}

operation String getUniqueVersion(uniqueID) : String {
  var result : Integer = 0;
  if (uniqueID.containsKey(self)) {
    result = uniqueID.get(self);
    uniqueID.put(self, result + 1);
  }
  else {
    uniqueID.put(self, 1);
  }

  return self + result;
}
```

Listing 4: ClassToJava controller template

```
[%
  for (cl in Model!Class.all()) {
    var t := TemplateFactory.load('JavaOneClass.egl');
    t.populate('currentClass', cl);
    t.generate(tgtdir + cl.name + '.java');
  }
%]
```

Listing 5: ClassToJava per-class template

```
package [%=pck%];

public class [%=currentClass.name%] {
[%
  for (prop : Model!Attribute in currentClass.attributes) {
    %]
  private [%=prop.type.name%] [%=prop.name%] =
      [%if (prop.type.isKindOf(Model!DataType)) {
      %] "[%=prop.value %]" [%
      } else {
      %] new [%=prop.type.name%] ()[%}%];
    [%
  }
%]

  public [%=currentClass.name%]() {}

  [%if ((not currentClass.attributes->isEmpty()) and
        // Java is not happy with too many parameters
        (currentClass.attributes->size() <= 200)) {%]
  public [%=currentClass.name%]([%
    var first = true;
    for (prop : Model!Attribute in currentClass.attributes) {
      if (not first) {%], [%}
      else {first = false;}
      %][%=prop.type.name%] [%=prop.name%][%
    }%]) {
      [%
        for (prop : Model!Attribute in
             currentClass.attributes) {
       %] this.[%=prop.name%] = [%=prop.name%];
      [%}%]
  }
  [%}%]
}
```

Listing 6: Build workflow

```
...

<target name="generate-java" depends="generate-general">
  <epsilon.eglRegister
        src="transformations/java/GenerateMain.egl">
    <model ref="classes" as="Model"/>
    <parameter name="tgtdir" value="${generate-tgt}/java/"/>
    <parameter name="pck" value="${tgtsubdir}.java"/>
  </epsilon.eglRegister>

  <epsilon.eglRegister
        src="transformations/java/ToJava.egl">
    <model ref="classes" as="Model"/>
    <parameter name="tgtdir" value="${generate-tgt}/java/"/>
    <parameter name="pck" value="${tgtsubdir}.java"/>
  </epsilon.eglRegister>

  <epsilon.eglMerge>
    <file>
      <include name="${generate-tgt}/java/*.java" />

      <superimpose artifactHandler="java15" />
    </file>
  </epsilon.eglMerge>
</target>

...
```

Listing 7: Java main method controller template

```
[%
  for (mdl in Model!Model) {
    var t := TemplateFactory.load('JavaMainMethod.egl');
    t.populate ('currentClass', mdl.topClass);
    t.generate (tgtdir + mdl.topClass.name + '.java');
  }
%]
```

an empty class body with only a main method with a recursively constructed constructor call in it. Note that because of a limitation in Java we are not generating custom constructors when there are more than 200 attributes in a class. This is to avoid compilation errors, because there is a maximum number of parameters that can be passed to a constructor. Alternatively, we could have translated such large repetition of attributes (as occurs, for example, in test case 6) into arrays. However, that would have been a deviation from the prescribed behaviour.

3.2 Transformation to C#

C# and Java are quite similar programming languages. The syntax of both languages is based on C/C++. They are both object-oriented and strongly typed languages. In general, the overall structure of C# and Java are almost identical for this FIXML transformation. The only real difference is the need to use 'using System;' at the beginning of each file to allow for the use of upper-case 'String' as a type name.

Because neither the class model nor the XML model contain any information specific to the target language, the early transformations can be kept unchanged. Only the final code-generation needs to be adjusted by 1) using the C#-specific template and 2) changing the language handler for the invocation of `eglMerge` to `csharp`. Language handlers encapsulate language-specific information like the feature-structure grammar and semantic merge-operators for unparsed blocks. A C# language handler did not exist in the original version of symmetric aspects for code generation as presented in [6, 7]. However, as the architectures of the generation infrastructure and the underlying FeatureHouse system are designed to be extensible, adding one was a matter of a few minutes.

4 Test runs

Eight test cases, all in xml format, were provided as inputs for this transformation. Amongst the test cases there were files which were deliberately wrong in order to check whether the transformation solution could detect such cases and give appropriate error messages. For this transformation, we have managed to

Listing 8: Java main method template

```
package [%=pck%];

public class [%=currentClass.name%] {
  public static void main (String[] args) {
    [%=currentClass.name%] top
        = [%=currentClass.generateConstructorCall()%];
  }
}

[%
  operation Model!Class generateConstructorCall() : String {
    var result : String = "new " + self.name + " (";

    // Java doesn't like too many parameters
    if (self.attributes->size() <= 200) {
      var first = true;
      for (attr in self.attributes) {
        if (not first) {
          result = result + ", ";
        }
        else {
          first = false;
        }

        if (attr.type.isKindOf(Model!DataType)) {
          result = result + '"' + attr.value + '"';
        }
        else {
          result = result +
                   attr.type.generateConstructorCall();
        }
      }
    }

    result = result + ")";
    return result;
  }
%]
```

TestCase 3:	`[epsilon.xml.loadModel] [Fatal Error] test3.xml:25:3: The element type "Order" must be terminated by the matching end-tag "</Order>".`
TestCase 7:	`[epsilon.xml.loadModel] [Fatal Error] test7.xml:14:12: The element type "Sndr" must be terminated by the matching end-tag "</Sndr>".`
TestCase 8:	`[epsilon.xml.loadModel] [Fatal Error] test8.xml:19:10: The element type "Hdr" must be terminated by the matching end-tag "</Hdr>".`

Table 1: Error messages

produce the correct output as the result of the transformations. All test cases were validated manually and compared to the output result by another approach (UML-RSDS). Test cases 3, 7 and 8 did not have the right XML format. The respective error messages are shown in Table 1. These errors are all caught by the XML driver for Epsilon [4] already.

5 Metrics

Tables 2 and 3 show the results for the various metrics requested in the case specification.

6 Conclusions and Outlook

We have presented a solution to the TTC 2014 FIXML case using symmetric aspects for code generation. The key feature of our solution is that our implementation could be largely built language independently and with almost no concern for schema derivation issues. We have not implemented the generator for C++. However, this could be easily realised following the same ideas by adding an appropriate set of code-generator templates.

References

[1] Sven Apel, Christian Kästner, and Christian Lengauer. FeatureHouse: Language-independent, automated software composition. In Stephen Fickas, Joanne Atlee, and Paola Inverardi, editors, *Proc. 31st Int'l Conf. on Software Engineering (ICSE'09)*, pages 221–231. IEEE Computer Society, 2009.

[2] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon object language (EOL). In Arend Rensink and Jos Warmer, editors, *Proc. ECMDA-FA 2006*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.

<table>
<tr><td>Complexity</td><td>It is not entirely clear what is meant by an operator or an entity/feature reference in this context, so the below values are approximations:

LoadXML – 35

XMLToClass – 61

ClassToJava – 12 (controller template) + 34 (per-class template) + 11 (main-method controller template) + 29 (main-method generation) = 86

ClassToCS – 12 (controller template) + 31 (per-class template) + 11 (main-method controller template) + 26 (main-method generation) = 80</td></tr>
<tr><td>Accuracy</td><td>Syntactic correctness and semantic preservation are achieved. Uniqueness of attribute names is guaranteed by <code>XMLToClass</code>.</td></tr>
<tr><td>Development effort</td><td>Approx. 3.5 person hours for Java; approx. 0.5 additional person hours for C#; approx. 1 person hour for a generalised build script (optional).</td></tr>
<tr><td>Fault tolerance</td><td>High – the transformation accurately reports errors in the XML files.</td></tr>
<tr><td>Execution time</td><td>The following times (in milliseconds) were measured when running all test cases on a TravelMate laptop with i5 CPU running at 2.4GHz and 4GB of main memory.
<table>
<tr><th>Stage</th><th>Minimum</th><th>Average</th><th>Maximum</th></tr>
<tr><td>LoadXML</td><td>78</td><td>299</td><td>1062</td></tr>
<tr><td>XMLToClass</td><td>31</td><td>304</td><td>1451</td></tr>
<tr><td>ClassToCSharp</td><td>63</td><td>1929</td><td>7317</td></tr>
<tr><td>ClassToJava</td><td>218</td><td>1713</td><td>5647</td></tr>
</table>
It should be noted that the times shown can vary substantially between runs of the experiment set. The code-generation stage takes the most time, which is in line with the fact that the main processing happens here. Further breakdown of the timing for Java generation reveals the following for the same run as above:
<table>
<tr><th>Stage</th><th>Minimum</th><th>Average</th><th>Maximum</th></tr>
<tr><td>RegisterJava</td><td>109</td><td>819</td><td>2636</td></tr>
<tr><td>MergeJava</td><td>109</td><td>894</td><td>3011</td></tr>
</table></td></tr>
</table>

Table 2: Metrics

Abstraction level	Medium as this is a declarative-imperative solution.
Modularity	Below are approximate values making assumptions about the meaning of 'rule':
	LoadXML – $1 - 1/1 = 0$
	XMLToClass – $1 - 5/6 = 1/6$
	ClassToJava – $1 - 3/4 = 1/4$
	ClassToCS – $1 - 3/4 = 1/4$

Table 3: Metrics (ctd.)

[3] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. The Epsilon transformation language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Proc. 1st Int'l. Conf. on Theory and Practice of Model Transformations (ICMT'08)*, volume 5063 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2008.

[4] Dimitrios S. Kolovos, Louis M. Rose, James Williams, Nicholas Matragkas, and Richard F. Paige. A lightweight approach for managing XML documents with MDE languages. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitris Kolovos, editors, *Proc. 8th European Conf. on Modelling Foundations and Applications (ECMFA'12)*, volume 7349 of *LNCS*, pages 118–132. Springer, 2012.

[5] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. Polack. The Epsilon generation language. In Ina Schieferdecker and Alan Hartman, editors, *Proc. 4th European Conf. on Model Driven Architecture (ECMDA-FA'08)*, pages 1–16. Springer, 2008.

[6] Steffen Zschaler and Awais Rashid. Symmetric language-aware aspects for modular code generators. Technical Report TR-11-01, King's College London, Department of Informatics, 2011.

[7] Steffen Zschaler and Awais Rashid. Towards modular code generators using symmetric language-aware aspects. In *Proceedings of the 1st International Workshop on Free Composition*, FREECO '11, pages 6:1–6:5, New York, NY, USA, 2011. ACM.