Solving the FIXML Case Study using Epsilon and Java

Horacio Hoyos
University of York, UK.
horacio.hoyos.rodriguez@ieee.org

Jaime Chavarriaga
Universidad de los Andes, Colombia.
ja.chavarriaga908@uniandes.edu.co

Paola Gómez
Universidad de los Andes, Colombia.
pa.gomez398@uniandes.edu.co

The Financial Information eXchange (FIX) protocol is the de facto messaging standard for pre-trade and trade communication in the global equity markets. FIXML, the XML-based specification for FIX, is the subject of one of the case studies for the 2014 Transformation Tool Contest. This paper presents our solution to generate Java, C# and C++ source code to support user provided FIXML messages using Java and the Epsilon transformation languages.

1 Introduction

This paper presents a solution to the 2014 Transformation Tool Contest (TTC) FIXML case [2]. It consists of a chain of transformation steps that takes a FIXML message and produces source code that represents the elements of that message as classes and the message content as instances of that classes.

Our solution[1] is implemented using Epsilon[2] and Java. It comprises three steps: the first step produces a model of the XML elements in the message, then another step transforms this model into a model of the classes and objects that represent the original message, and finally the last step produces the corresponding source code in Java, C# and C++.

The remainder of this paper is structured as follows. Section 2 presents how we use Epsilon to solve the case, Section 3 presents an evaluation of our solution, and Section 4 concludes the paper.

2 FIXML Solution using Epsilon

Epsilon is a set of task-specific languages for model management that can be easily integrated in Java. It includes languages such as the Epsilon Transformation Language (ETL) for processing and transforming models, and the Epsilon Generation Language (EGL) for generating code from models [1].

Our solution consists of a transformation chain that comprises: 1. A step that transforms an XML file into a corresponding XML-model implemented using a Java-based model loader, 2. An XML-model to Object-model transformation step implemented using the ETL, and 3. An Object-model to source code transformation implemented using the EGL.

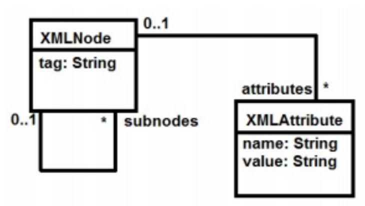
2.1 XML message to XML-model transformation

The first task is processing a FIXML message to create a corresponding XML-model, i.e., an EMF-based model representing the XML nodes and attributes. The resulting XML model must be conform to the

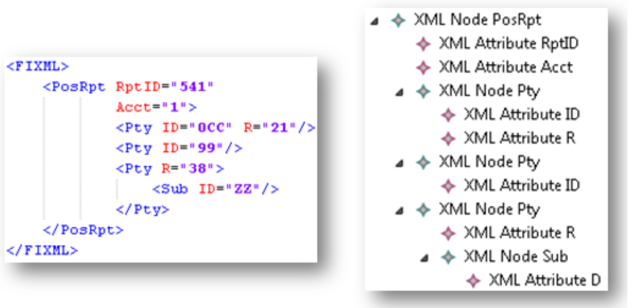
[1]https://github.com/arcanefoam/fixml
[2]http://www.eclipse.org/gmt/epsilon

Submitted to:
TTC 2014

© H. Hoyos, J. Chavarriga & P. Gómez
This work is licensed under the
Creative Commons Attribution License.



(a) XML metamodel



```
<FIXML>
    <PosRpt RptID="541"
            Acct="1">
            <Pty ID="OCC" R="21"/>
            <Pty ID="99"/>
            <Pty R="38">
                <Sub ID="ZZ"/>
            </Pty>
    </PosRpt>
</FIXML>
```

```
XML Node PosRpt
    XML Attribute RptID
    XML Attribute Acct
    XML Node Pty
        XML Attribute ID
        XML Attribute R
    XML Node Pty
        XML Attribute ID
    XML Node Pty
        XML Attribute R
        XML Node Sub
            XML Attribute D
```

(b) FIXML message and the corresponding XML-model

Figure 1: Example Models of the XML-model to Object-model transformation

metamodel specified in the case description [2] and depicted in Figure 1a. For instance, Figure 1b shows a simple FIXML message with six XML tags and the corresponding XML-model. The right-hand model includes an instance of the XML Node meta-class for each tag in the left-hand file: one for top-level `PosRp` tag, three for the inner `Pty` tags, and another one for the `Sub` tag inside the last `Pty` instance. In addition, each XML Node instance includes a set of XML-Attribute instances according to the values in the original XML file. Note in the figure that the three XML node instances for the `Pty` tags include different attributes: one includes XML-attributes for `ID` and `R`, other only an XML-attribute for `ID`, and the other only one for `R`.

Although Epsilon provides facilities to process XML files[3], we implement this first step using a Java SAX XML parser and the EMF framework. We opt for the SAX XML parser because it gives us more control about how the XML syntax errors are detected and how the application will inform the user of malformed input files. Our implementation consists of a Java class with handlers for each XML processing event in the SAX parser[4]: Each time the SAX parser detects an XML tag our class uses the EMF to create a XML-Node instance, and each time the parser detects an attribute our class creates an XML-Attribute instance. In consequence, when the SAX parser ends the processing of the XML file, our class has produced the complete XML-model as specified in the case study.

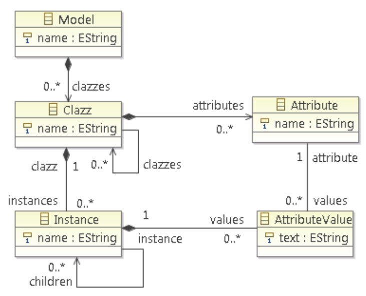
2.2 XML-model to Object-Model transformation

Once the XML-model is created, it is transformed into a corresponding Object-Model, i.e., an EMF-based model representing the classes that correspond to structure of the message structure, and the objects that correspond to the data in the message.

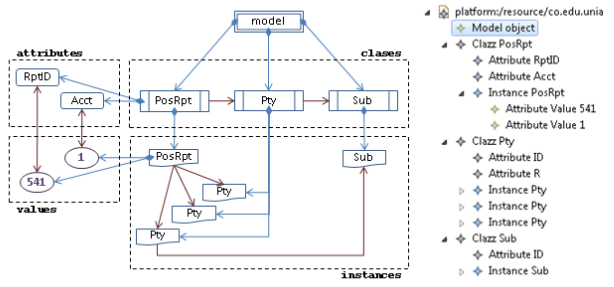
The FIMXL case description [2] does not specify a metamodel for the Object-Model. Figure 2a describes the object metamodel we are using in our solution. The root is a meta-class named `Model`, which serves as a container of all elements. This `Model` contains a set of `Clazz`, a meta-class that represents each class to be created. In turn, each `Clazz` may be related to another `Clazz`, to a set of `Attributes` and to a set of `Instances`. Finally, each `Instance` may contain a set of `AttributeValues`.

The case description [2] defines some informal rules about how the XML-model must be transformed into a corresponding Object-Model: 1. XML tags must be translated into Classes in the target model, 2. XML attributes must be mapped to Attributes, and 3. Nested XML tags become Properties (i.e., as

[3] http://www.eclipse.org/epsilon/doc/articles/plain-xml/
[4] https://www.jcp.org/en/jsr/detail?id=206



(a) Object metamodel used in our solution



(b) Object Model corresponding to the FIXML message in Figure 1

Figure 2: Example Models of the XML-model to Object-model transformation

member objects or relationships to other Classes). In addition, based on the examples included in the description, we define a set of additional rules: 1. XML nodes must be transformed into Instances of the Class that correspond to the XML tag, 2. the values of the XML Attributes must be mapped to Attribute Values of the Instances, and 3. in an XML node, nested XML nodes must be transformed into relationships between the parent instance and the children instances.

We implement this second step using an ETL Transformation. This transformation has rules to create `clazzes` and instances for each tag in the FIXML model. Basically each of the FIXML types is transformed into a set. Each set contains both the object description and the object instance. Thus, for example, a FIXML Node is transformed into a `Clazz` and an Instance. A look-up of previously defined `Clazzes` ensures that `Clazzes` are not duplicated. The same logic applies for Attributes.

2.3 Object-Model to source code transformation

The final step comprises the generation of the Java, C# and C++ source code that correspond to the Object-Model obtained before.

Based on the examples provided in the description, we define a set of general rules to generate the code: (a) Every `Clazz` of the Object-Model must be generated into a class, (b) the `Attributes` of a `Clazz` must be typed as String and declared as private in the corresponding class, (c) for each class, the relationships to other `Clazzes` are implemented as typed lists of objects, (d) each class includes additional methods to add objects to the object lists, (e) the default constructor for every class creates an instance with attribute values and relationships that corresponds to the first XML node of the Object-Model, and (f) an additional constructor with parameters assigns values to the class attributes.

These rules are adapted to the peculiarities of each language. For instance, for Java, each class is generated in a different ".java" file. For C#, each class is generated in a ".cs" file. And for C++, each class is generated in a ".h" file for the class interface and a ".cpp" file with the implementation. Also, these rules consider other limitations imposed by these languages, e.g., Java does not accept more than 256 arguments in each method.

We implement these transformation rules using three different EGL templates: one for Java, other for C# and another for C++. Basically each template consists of fragments of source code with marks that are replaced by the values in the elements of the object model during code generation. In our implementation, the generation of the three languages are launched in parallel using java threads.

3 Evaluation of the Solution

The 2014 TTC FIXML case description [2] defines seven (7) measures to evaluate the solutions systematically: Abstraction level, Complexity, Accuracy, Development effort, Fault Tolerance, Execution Time and Modularity. The following are the results of evaluating our solution based on these measures.

Execution time. The execution time is measured as the milliseconds spent for executing each of the three stages with the provided FIXML files. The following table shows the average execution time of ten (10) consecutive executions of our solution using the sample FIXML files provided in the description.

Test file	**Init EMF**	**XML to XmlModel**	**XmlModel to ObjectModel**	**ObjectModel to code**
test1.xml	767.9	249.7	696.0	804.1
test2.xml	751.0	256.9	901.0	1055.9
test3.xml	770.2	262.7	796.5	1256.8
test4.xml	745.4	375.5	2995.9	2382.7
test5.xml	779.8	323.6	1643.9	1471.9
test6.xml	745.4	375.5	2995.9	2382.7

Table 1: Execution time (in milliseconds) of processing the example FIMXL messages

Abstraction Level. Our solution combines imperative code in Java and declarative scripts in ETL and EGL. As a result, according to the evaluation criteria [2], the abstraction level of (a) the Java code to launch transformations is low, (b) XML to XML-model transformation is low, (c) XML-model to Object-model transformation is high, (d) Object-model to code transformation is high, and (e) the overall solution is medium

Complexity. For the Java code used to take the XML file and create an XML-model, we measure the complexity c as the sum of e, the number of expressions and instructions involved in processing the XML tags and create the corresponding model; r_c, the number of references to meta-classes and r_p, the number of references to meta-class properties. The corresponding values: a) 18 for e, b) 2 for r_c, and c) 8 for r_p, provide a complexity of 28 for the first transformation step.

For the transformation scripts in ETL and EGL, we measure the complexity c as the sum of e, the number of EOL expressions and functions; r_c, the number of references to meta-classes and r_p, the number of references to meta-class properties. As a result, the complexity to take the XML-model and create an Object-model was 66 ($e = 35$, $r_c = 8$, $r_p = 23$), and to take this model in order to create the java code was 76 ($e = 24$, $r_c = 3$, $r_p = 49$), the C# code was 71 ($e = 22$, $r_c = 3$, $r_p = 46$), and the C++ code was 112 ($e = 41$, $r_c = 6$, $r_p = 64$). Finally, the sum of all complexities provides the overall complexity which is 353.

Accuracy. According to the evaluation criteria, we consider our solution **accurate** after performing a set of testing procedures that includes processing the set of FIXML messages provided in the case description and compiling the resulting code using the JDK compiler[5] for the Java code, the Mono compiler[6] for the .NET code and GCC/MingW[7] for the C++ code.

Fault tolerance. According to the evaluation criteria [2], our solution is **High**: it detects erroneous XML files and present information about the error.

[5]`https://jdk7.java.net/download.html`

[6]`http://www.mono-project.com/CSharp_Compiler`

[7]`http://www.mingw.org/`

Modularity. Modularity m is measured as $m = 1 - (d/r)$, where d is the number of dependencies between rules (implicit or explicit calls, ordering dependencies, inheritance or other forms of control or data dependence) and r is the number of rules.

For the XML to XML-model transformation, the Java code consists of a class with event-handler methods, i.e., a class with methods that are invoked during the processing of an XML document. We measure the number of rules as the number of event-handler methods (i.e., $r = 4$). And, because these methods does not invoke one to the other, we consider that there is not dependencies among these rules (i.e., $d = 0$). Thus, the modularity corresponds to 1.

In ETL, each rule uses an *equivalents* method to obtain the model elements produced by other rules. We measure the dependencies d as the number of times that the *equivalents* method is used in all the rules. For instance, the XML-model to Object-model transformation uses only three rules (i.e., $r = 3$) but all these rules uses the *equivalents* method four times (i.e., $d = 4$). Thus, the modularity is $-0.\overline{33}$.

In EGL, an *operation* is a reusable text template that can be included as a part of any other template. Thus, we measure r as the number of *operations*, including the main template, and d as the number of times that an operation invokes another operation. For instance, the Object-model to Java code transformation includes a main template and three operations (i.e., $r = 4$) and all these operations invoke other operations five times (i.e., $d = 5$). That means that the modularity corresponds to -0.25.

The following table details the measures for modularity of our solution.

Element		r	d	**Modularity**
XML to XML-model		4	0	1
XML-model to Object-model		3	4	$-0.\overline{33}$
Object-model to code	Java	4	5	-0.25
	C#	4	5	-0.25
	C++	8	10	-0.25

Table 2: Modularity of each transformation step of the solution

Development effort. The effort of developing each element, measured in person-hours, was : 4h for the XML to XML-model transformation, 2h for the XML-model to Object-model step, 2h for the code generation in java, 1h for the generation in c#, and 4h for the code in C++. We must clarify that the first transformation we create was the Object-Model to Java transformation, and we later use that transformation as a foundation to create the transformations for the other languages.

4 Conclusions

In this paper, we have discussed our solution to the TTC 2014 FIXML case based on Epsilon. This solution is structured as requested (i.e., there is a generic XML-to-XMLModel transformation, an XMLModel-to-ObjectModel transformation, and an ObjectModel-to-Text transformation) and evaluated using the criteria defined in the case description.

References

[1] Dimitris Kolovos, Louis Rose, Antonio Garcia-Dominguez & Richard Paige (2014): *The Epsilon Book*. Available at `http://www.eclipse.org/epsilon/doc/book/`.

[2] K. Lano, S. Yassipour-Tehrani & K. Maroukian (2014): *Case study: FIXML to Java, C# and C++*. In: *Transformation Tool Contest 2014*.