Solving the FIXML Case Study using Epsilon and Java

Horacio Hoyos[1], Jaime Chavarriaga[2], and Paola Gomez[2]

[1] Department of Computer Science, University of York, UK.
horacio.hoyos.rodriguez@ieee.org
[2] Universidad de los Andes, Colombia.
{ja.chavarriaga908, pa.gomez398}@uniandes.edu.co

Abstract. The Financial Information eXchange (FIX) protocol is 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 [1]. It consists in a chain of transformation steps that takes a FIXML message and produces first a model of the XML elements in the message, then a model of the classes and objects that represent that message, and finally the corresponding source code in Java, C# and C++. The solution is available as a Github repository[3].

Our solution is implemented using Epsilon[4] invoked from a Java application. Epsilon is an extensible set of languages and tools for model management built atop the Eclipse Modeling Framework (EMF) [2]. Because it interoperates seamlessly with several modelling technologies and file formats, including plain XML files [3], Epsilon is very suitable for solving the above mentioned case study that involves processing of XML files and EMF models, before generating source code based on them.

The remainder of this paper is structured as follows. Section 2 introduces some of the languages and tools in Epsilon, Section 3 specifies the 2014 TTC FIXML Case, including some details about the source code to generate, and Section 4 presents how we use these features to solve the case. Finally, Section 5 presents an evaluation of our solution, and Section 6 concludes the paper.

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

2 Epsilon overview

Epsilon includes several task-specific languages for processing and transforming models, and generating code from models [3]. For instance, among these languages, the following are the Epsilon languages used in our solution:

- The **Epsilon Object Language (EOL)**, a language with the ability to access multiple models, query and update model elements, and build new models from scratch.
- The **Epsilon Transformation language (ETL)**, a declarative language based on EOL that support the models transformation using rules that map elements in the source model with the target model elements. In addition, ETL is capable to transform several source models into several target models. For our solution, we decided to use ETL in order to transform one source model into a one target model.
- The **Epsilon Generation Language (EGL)**, a language for templates that supports the generation of text files combining text fragments and EOL expressions.

Software developers use these languages to create scripts that take one or more files or model instances and transform them into other models or into source code. Later, these scripts can be executed inside the Eclipse IDE or invoked directly from a Java program, i.e., outside Eclipse. This paper presents a Java solution that executes Epsilon scripts and may run outside Eclipse.

3 FIXML Case Study

For 2014, the TTC proposes a transformation case based on the FIXML standard for trading messages. Although the case is explained elsewhere [1], we found some elements in the generated source code that must be further explained and specified. This section describes the case study giving additional information about the features we consider in our solution, in concrete, details about the source code to generate.

3.1 Overview of the transformation chain

The FIXML case comprises a chain of transformation steps that takes an XML message and produces the corresponding source code that represent message elements as classes and message content as an instance created in runtime.

The transformation chain comprises:

- A XML message to XML-model transformation
- A XML-model to Object-model transformation
- An Object-model to source code transformation

3.2 XML message to XML-model transformation

FIXML messages are XML-based documents that includes information about trading transactions. The first task in the contest is processing a FIXML message and creating a corresponding XML-model, i.e., an EMF-based model representing the XML nodes and attributes.

Figure 1 shows the metamodel for the intended XML-model, as it is specified in the case description [1].

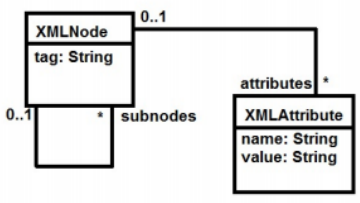

Fig. 1: XML metamodel provided by the FIXML case description

In this metamodel, the type `XMLNode` represents each tag of the XML message, which can have multiples nodes associated to it. In addition, the type `XMLAttribute` represents each value provided by each tag of the XML message.

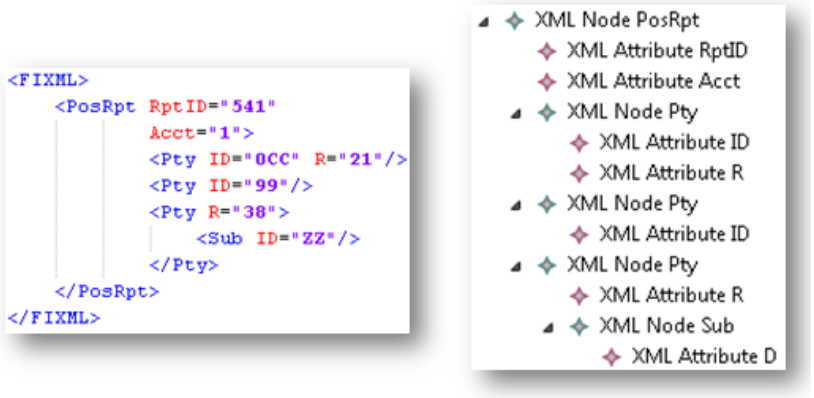

Fig. 2: FIXML messsage and the corresponding XML-model

For instance, Figure 2 depicts a simple FIXML message and the corresponding XML-model. On the left side, the FIXML message contains the tag `PosRpt` with the attributes `RptID` and `Acct` whose values are `541` and `1` respectively. In addition, this tag `PosRpt` contains three inner tags named `Pty`, each one with

different values for its attributes: 1. the first tag, contains the attributes `ID` and `R` with the values `OCC` and `21`, 2. the second tag, contains the attribute `ID` with a different value (`99`), and 3. the third tag, besides having the attribute `R`, contains another tag called `Sub` with the respective attribute information.

Also in the Figure 2, the right side shows the corresponding XML-model, which is conform to the XML metamodel provided by the case description. This model includes, for each tag of the original message, an instance of the XML Node meta-class: 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 turn, each of these XML Node instances includes a set of XML-Attribute instances according to the values in the original XML file. For instance, note 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`.

3.3 XML-model to Object-Model transformation

Once an XML-model is created based on the FIXML message, the next step is transform this XML-model into a corresponding model that Object-Model, i.e., an EMF-based model representing the classes that support the message structure and the objects that are part of the message.

The FIMXL case description [1] does not specify a concrete metamodel for the Object-Model. Figure 3 describes the metamodel we are using in our solution.

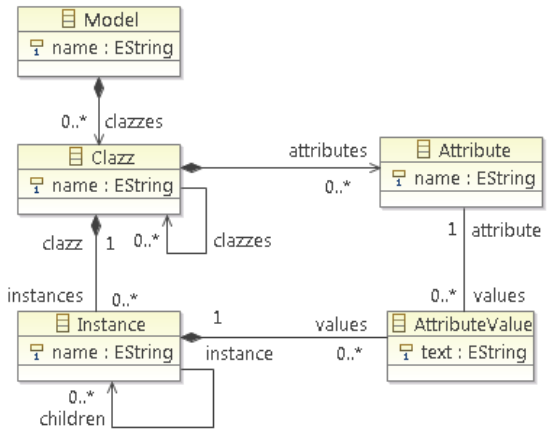

Fig. 3: Object metamodel used in our solution

The root of our Object metamodel 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 mentioned case description [1] defines some informal rules about how the XML-model must be transformed into a corresponding Object-Model:

- XML tags must be translated into Classes in the target model.
- XML attributes must be mapped to Attributes
- Nested XML tags become Properties (i.e., as member objects or relationships to other Classes)

Although the mentioned case description does not mention rules about how to transform the data in the XML to object instances, it provides some examples that we use to define some additional rules:

- XML nodes must be transformed into Instances of the Class that correspond to the XML tag.
- values of the XML Attributes must be mapped to Attribute Values of the Instances.
- In a XML node, nested XML nodes must be transformed into relationships between the parent instance and the children instances.

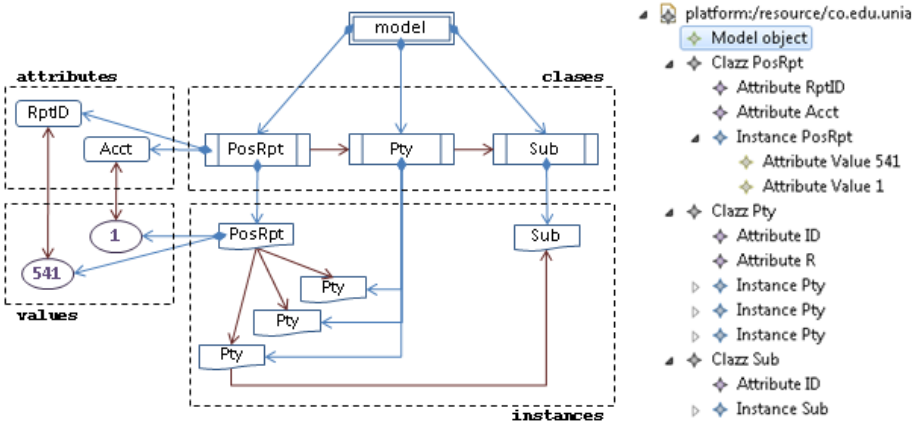

Fig. 4: Object Model corresponding to the above FIXML message

For instance, Figure 4 depicts the Object model that corresponds to the XML model presented in the Figure 2. The model includes a set of `Clazz` elements, one for each new XML Node name. In turn, each `Clazz` contains a set of `Instances` representing each XML node. Note that the model includes a `Clazz` named `Pty` that includes three `Instances`, one for each node of that tag in the original XML file.

In the model, each `Clazz` also includes `Attributes`, one for each attribute in the corresponding XML nodes. In addition, each `Instance` member of a class may include an `Attribute Value` for each of these attributes. Note that the model includes, as the root element in the XML document, a `Clazz` for the `PosRpt` tag with the attributes `RptID` and `Acct`. At the same time, this `Clazz` includes an `Instance` with the set of `Attribute Values` `541` and `1` which correspond with the attributes `RptID` and `Acct` respectively. In a similar way, the attributes and attributes values are treated for each `Instance`.

3.4 Object-Model to source code transformation

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

The structure and features of the source code to generate is not completely specified in the FIMXL case description [1]. This description includes some code examples that we use to establish some requirements about the code to generate. The following are the requirements we considered in our solution:

For the Java source code

- Every `Clazz` of the Object-Model must be generated into a different file named as the name of the class, and with the extension ".java".
- The `Attributes` of a `Clazz` must be typed as String and declared as private attributes in the corresponding class.
- For each class, the relationships to other `Clazzess` are implemented as typed lists of objects.
- 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.
- An additional constructor with parameters assigns values to the class attributes.
- Each class includes additional methods to add objects to the object lists.
- The generated classes does not include getters and setters for the other attributes

For the C# source code

- Classes must be generated in individual ".cs" files.
- The default constructor for every class corresponds the attribute values and relationships to the first XML node of the Object-Model.
- An additional constructor with parameters assigns the values given to the corresponding attributes, its does not consider the relationships.
- In every class, the relationships to other classes must be implemented as a typed list of objects. An additional method to add objects to the list must be included in the class.
- The generated classes will not include getters and setters for the other attributes.

For the C++ source code

- Classes must be generated in pairs of files, a ".h" file with the class interface, and a ".c++" file with the class methods implementation.
- The default constructor for every class must create an object instance with the attribute values and relationships of the first node of the corresponding XML tag in the original XML document.
- An additional constructor with parameters must create an object instance with the specified attribute values.
- In every class, the relationships to other classes must be implemented using multiple attributes (but not a list). These attributes must be named using the name of the other class and a consecutive number.

4 Solving the FIXML Case Study using EMF and Epsilon

Our solution was developed with the purpose of providing a stand-alone application that accepts an FIXML file and an output folder path as inputs and generates the required code.

Our solution implements the three-step model transformation chain (MTC):

1. FIXML text to FIXML model transformation
2. FIXML model to Object model
3. Object model to code

4.1 XML message to XML-model transformation

For the first step we used the Java SAX XML parser and EMF to populate the FIXML model. In the second phase we used an ETL model to model transformation and in the third phase we used EGL templates for code generation. In the first phase the SAX XML parser provides XML syntax error detection to inform the user of malformed input files. Since the SAX parser is event based, each of the events is used to identify what information has been parsed and using EMF we create and populate the FIXML model.

4.2 XML-model to Object-Model transformation

In the second step, we use ETL to transform the FIXML model into an Object model. This transformation deals with the fact that we have to both create clazzes and instances for each of the tags in the FIXML model. The transformation is a simple two rule description which uses some of the advanced features of ETL to accomplish the transformation. 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.

4.3 Object-Model to source code transformation

The third step consists of the code generation for which we provide three separate EGL templates: Java, C# and C++. For this phase, the three generations are launched in parallel using java threads.

4.4 Executing the solution

The complete solution uses the stand-alone versions of ETL and EGL and can be runned from the command line. It receives two arguments:

1. Path to the FIXML text file
2. Path to the output folder for generated code (optional)

If the second argument is not supplied the code would be generated in the same location as the FIXML file adding sub-folders for each of the languages.

5 Evaluation of the Solution

The 2014 TTC FIXML case description [1] defines a set of 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.

Abstraction Level. According to the evaluation criteria, the abstraction level should be evaluated as: 1. **High** for primarily declarative solutions, 2. **Medium** for declarative-imperative solutions, and 3. **Low** for primarily imperative solutions.

Although our solution combines imperative code in Java to start the execution of the transformation scripts, these scripts are primarily declarative specifications. The following table shows the abstraction level for each element in the solution and an overall value for all the solution.

Element	**Abstraction Level**
Java code to launch transformations	Low
XML to XML-model transformation	Low
XML-model to Object-model transformation	High
Object-model to code transformation	High
Overall solution	Medium

Complexity. The solution complexity should be measured as the sum of number of operator occurrences and feature and entity type name references in the specification expressions.

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.

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.

The following table shows the complexity for each element in the solution and an overall value for all the solution.

Element		e	r_c	r_p	**Complexity**
XML to XML-model		18	2	8	28
XML-model to Object-model		35	8	23	66
Object-model to code	Java	24	3	49	76
	C#	22	3	46	71
	C++	41	6	64	112
Overall solution					353

Accuracy. According to the evaluation criteria, a solution is valid when 1. the resulting code is valid in the target languages (syntactic correctness), and 2. that code represents the source XML data structure and elements (semantic preservation)..

Our solution is **accurate**. In order to verify this, we have compiled 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.

Development effort. The effort is measured as the time in person-hours spent in writing and debugging the transformation scripts. The following is a table detailing the effort of developing each element of our solution.

Regarding the effort spent for the transformations to code, 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 and debug the other transformations to code.

[5] `https://jdk7.java.net/download.html`
[6] `http://www.mono-project.com/CSharp_Compiler`
[7] `http://www.mingw.org/`

Element		Development effort
XML to XML-model		4h
XML-model to Object-model		2h
Object-model to code	Java	2h
	C#	1h
	C++	4h

Fault tolerance. Fault tolerance is classified as 1. **High** if transformation is able to detect invalid input XML and produce accurate error messages; 2. **Medium** if erroneous files produce a failed execution with an indication that some error occurred; and 3. **Low** if such files are processed and output produced without warnings being issued.

Regarding Fault Tolerance, our solution is **High**. Basically, it detects erroneous XML files and present information about the error. For instance, the "test7.xml" and "test8.xml" are erroneous XML file. Our application reports the following errors for these files:

Test file	Error message
test7.xml	`Fatal Error: URI=file:/.../test7.xml Line=14: The element type "Sndr" must be terminated by the matching end-tag "</Sndr>". The respective classes where not created.`
test8.xml	`Fatal Error: URI=file:/.../test8.xml Line=19: The end-tag for element type "Order" must end with a '>' delimiter. The respective classes where not created.`

Execution time. The execution time is measured as the milliseconds spent for executing each of the three stages with the provided FIXML files. In addition, we measure the time spent for initializing the EMF metamodels, a task that is performed once in our application but must be executed for each step when these steps run separately.

The following table shows the execution time of our solution. For each transformation time, this measures include the loading of models and the printing of output code from the application. These measures is the average execution time of ten (10) consecutive executions of the application with the specified message.

Modularity. According to the evaluation criteria, the modularity m should be 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 must be invoked, during the processing of an XML document, at the start of the document, at the

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

start of a node, at the end of a node and at the end of the document. Considering this, we measure the number of rules as the number of event-handler methods (i.e., $r = 4$). Because these methods does not invoke one to the other, we consider that there is not any dependencies among these rules (i.e., $d = 0$).

In ETL, each rule can use the *equivalents* method to obtain the model elements produced by other rules. Thus, 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., $d = 4$). Using the above mentioned formula, the modularity correspond to $-0.\overline{33}$.

In EGL, the code is generated using text templates where special tags denote where some text must be replaced by values from the model. In addition, an *operation* is a reusable text template that can be included as a part of any other template. For the EGL scripts, 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 invokes other operations five times (i.e., $d = 5$). That means that the modularity correspond 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

6 Conclusions

We have developed and discussed an Epsilon-based solution to the TTC 2014 FIXML case. It includes a series of transformation scripts structured as requested by the case description, i.e., there is a generic XML-to-XMLModel transformation,

an XMLModel-to-ObjectModel transformation, and an ObjectModel-to-Text transformation.

In addition, we have presented an evaluation of our solution considering a set of predefined quality attributes. This evaluation will be useful to compare our solution to other proposed in other languages or by other authors.

References

1. Lano, K., Yassipour-Tehrani, S., Maroukian, K.: Case study: FIXML to Java, C# and C++. In: Transformation Tool Contest 2014. (2014)
2. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. 2 edn. Addison-Wesley Professional (2008)
3. Kolovos, D., Rose, L., Garcia-Dominguez, A., Paige, R.: The Epsilon Book (2014)