FIXML to Java, C# and C++ Transformations with QVTR-XSLT

Dan Li, Danning Li
Guizhou Academy of Sciences, Guiyang, China

Xiaoshan Li
Faculty of Science and Technology, University of Macau, China

Volker Stolz
Bergen University College, Norway

QVTR-XSLT is a tool for design and execution of transformations based on the graphical notation of QVT Relation. In this paper, we present a solution to the "FIXML to Java, C# and C++" case study of the Transformation Tool Contest (TTC) 2014 using the QVTR-XSLT tool.

1 Introduction

The "FIXML to Java, C# and C++" case study of the Transformation Tool Contest (TTC) 2014 addresses the problem of automatically synthesizing program code from financial messages expressed in FIX (Financial Information eXchange) format. The problem can be broken down into three tasks: 1) generating FIX model from FIX text file, 2) producing a model of the program language from the FIX model, and 3) converting the program model to program code of Java, C# or C++. In this paper, the transformation tasks are tackled with QVTR-XSLT [1], a tool that supports editing and execution of the graphical notation of QVT Relations (QVT-R) language [3].

As part of the model transformation standard proposed by the Object Management Group (OMG), QVT-R is a high-level, declarative transformation language. Its graphical notation provides a concise, intuitive, and yet powerful way to define model transformations. In QVT-R, a transformation is defined as a set of *relations* (rules) between source and target metamodels, where a relation specifies how two object diagrams, called *domain patterns*, relate to each other. Optionally, a relation may have a pair of *when*- and *where*-clauses specified with an extended subset of Object Constraint Language (OCL) to define the pre- and postconditions of the relation, respectively. A transformation may also include *queries* and *functions*. Transformations are driven by a single, designated top-level relation.

QVTR-XSLT supports the graphical notation of QVT-R and the execution of a subset of QVT-R by means of XSLT [4]. The tool supports unidirectional non-incremental enforcement model-to-model transformations of QVT-R. Features supported include transformation inheritance through rule overriding, traceability of transformation executions, multiple input and output models, and in-place transformations. In addition, we extend QVT-R with additional transformation parameter, conditional relation call and graphical model query [2]. The tool provides a *graphical editor* in which metamodels and transformations can be specified using the graphical syntax, and a *code generator* that automatically generates executable XSLT stylesheets for the transformations. A *transformation runner* is also developed to execute a single or a chain of generated XSLT transformations by invoking a Saxon XSLT processor. It can display the execution time and generate the execution trace if required.

The rest of the paper is structured as follows: Section 2 introduces the design of a solution for the case study. We discuss the experimental result and evaluation of the solution against the criteria given in the case specification in Section 3.

Submitted to:
TTC 2014

2 Solution design

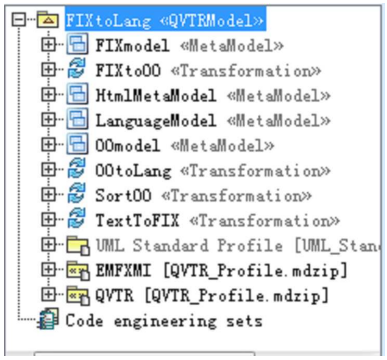

Figure 1: Solution overview.

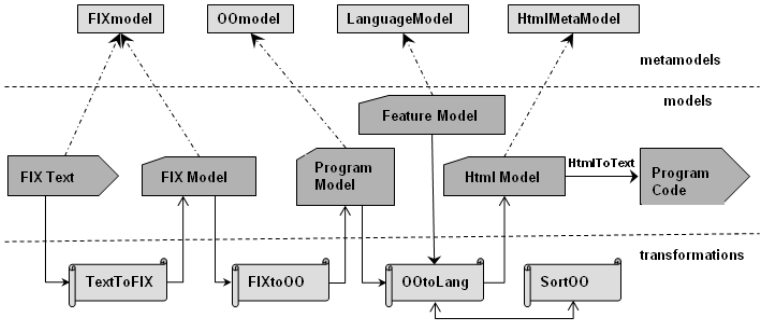

Figure 2: Overall transformation process.

Using the graphical editor of QVTR-XSLT, the solution for the case study is designed as a QVT-R transformation model *FIXtoLang* whose outline is shown in Fig. 1. It consists of 4 metamodels and 4 transformations. Among the metamodels, *FIXmodel* specifies the structures of both FIX text model and FIX model, *OOmodel* defines the abstract model for the OO program languages, and the *LanguageModel* provides the concrete syntax features for each language.

To complete the tasks of the case study, transformation *TextToFIX* reads a FIX text file and transforms it to a FIX model (task 1, see Section 2.1), which is subsequently converted into an abstract program model by the *FIXtoOO* transformation (task 2, see Section 2.2). In case of C++, the classes defined in the program model need to be sorted to ensure a class is declared before being called. Transformation *SortOO* is dedicated to this purpose. For the next task, as QVTR-XSLT is mainly designed for model-to-model transformations, the program model, along with the language concrete feature model, are first transformed to program code represented as an HTML model that conforms to the *HtmlMetaModel* of Fig. 1. Then, a pre-defined XSLT stylesheet generates a plain text file of the program code from the HTML model (see Section 2.3). This transformation process, the various artifacts and their relation to each other, are shown in Fig. 2.

2.1 FIX text to FIX model transformation

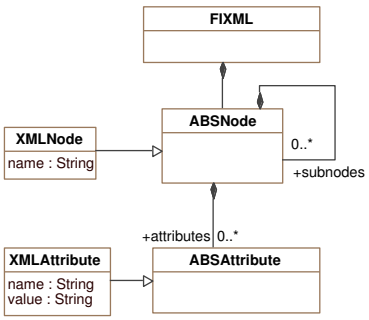

Figure 3: FIX metamodel.

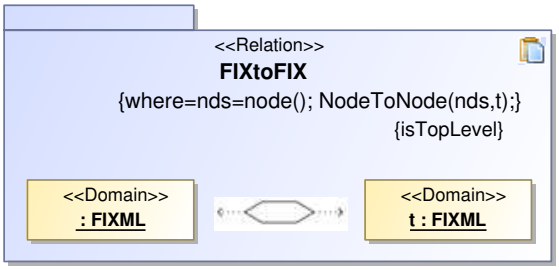

Figure 4: Top relation *FIXtoFIX*.

The very first transformation *TextToFIX* takes as input an XML text file and outputs a model of FIX format. As shown in Fig. 3, we define a single metamodel *FIXmodel* for both the source and target

models. QVTR-XSLT uses simple UML class diagrams to define metamodels, and requires that a model has a unique root element, such as the *FIXML* shown in the Fig. 3. In the metamodel, two elements, *ABSNode* and *ABSAttribute*, specify the structure of the source text model. Their sub-classes, *XMLNode* and *XMLAttribute*, defines the metamodel of the target FIX model. Slightly different from the metamodel given in the case specification, we use *name* property instead of *tag* to specify the tag of a FIX node.

The transformation itself is simple and straightforward. It starts from the top relation *FIXtoFIX* (Fig. 4), which matches the *FIXML* element (the root of the source text model) in its left-hand part, and constructs the root *FIXML* element of the target model in its right-hand part. In the *where* clause, function *node()* is used to obtain all direct subnodes owned by the root of the source model, and another relation *NodeToNode* is invoked to subsequently map these subnodes. The mapping is mostly one-to-one.

2.2 FIX model to program model transformation

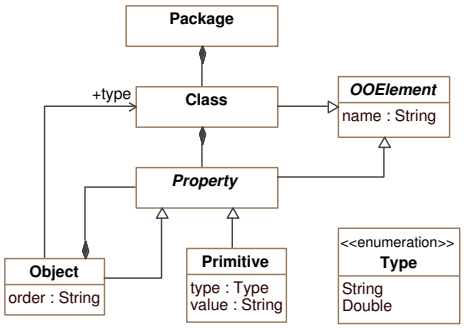

Figure 5: Metamodel of program model.

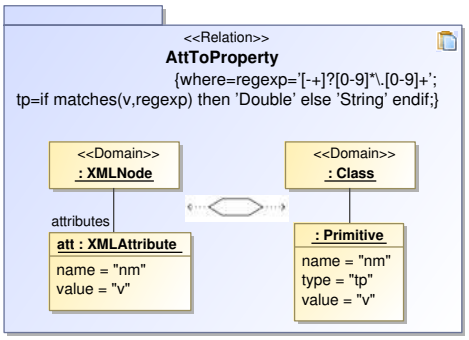

Figure 6: Relation *AttToProperty*.

Fig. 5 illustrates the metamodel of the program model, which serves as the target metamodel of the transformation *FIXtoOO*. The three programming languages share the same abstract syntax definitions. In the metamodel, we define a root element *Package* that contains a set of *Classes*. A class owns *Properties* which could be either of a *Primitive* type (e.g., *String* or *Double*) or an *Object* of class type. The *order* property in *Object* elements indicates the order of an object if there are multiple objects with the same name.

The challenge of the transformation is that in the source model there may be multiple nodes with the same tag name. These nodes are distributed throughout the model, and each of them may have a different set of subnodes. We have to search the whole model to collect all occurrences of this node, union all of their subnodes to obtain a largest set, and convert the set to the properties of corresponding class in the target model. As multiple subnodes with the same tag name may exist within the same node, a function is used to count the order of the subnodes, and store the order in the *order* property of the *Object* element.

We tackle the task of Extensions 3.1 (selecting appropriate data types) in the relation that transforms attribute nodes of the source model into primitive properties of target model, as shown in Fig. 6. In the *where* clause, a regular expression *regexp* is used in the *matches* function to decide if the value *v* is of type *Double*, otherwise it is of type *String*.

2.3 Program model to program code transformation

This task is comprised of three steps: 1) sorting class declarations of the program model; 2) transforming the program model into an HTML model of a particular programming language; 3) rendering the HTML

model to a text file.

Sorting program model. For C++, the class declarations should be ordered so that classes are always declared before they are used. We design transformation *SortOO* for that purpose. It takes *OOmodel* as the source- and the target metamodel. The transformation adopts a typical bubble sort algorithm. The following function is defined for comparison of the pair of adjacent classes:

```
function Compare(c1:Class, c2:Class) {
    result=if c2.#Object.type→includes(c1.name) then c1→union(c2) else c2→union(c1) endif;
}
```

where the input parameter *c1* is located before *c2* in the source model. However, if class *c2* does not include any object of type *c1*, we consider *c2* is "smaller" than *c1* and swap them.

Program model to HTML model. This transformation *OOtoLang* takes as input a program model and a feature model, and generates an HTML model for the code of the particular programming language. It calls the sorting function defined in *SortOO* if needed. The feature model, which conforms to the metamodel *LanguageModel*, defines the concrete syntax features for each language:

```
<LanguageDef>
    <LangDef name="Java" this="this." String="String" Double="Double" iniVar="true" nul='null' orderClass="false" .../>
    <LangDef name="C#" this="this." String="string" Double="double" iniVar="true" nul='null' orderClass="false" .../>
    <LangDef name="C++" this="" String="string" Double="double" iniVar="false" nul='NIL' orderClass="true" .../>
</LanguageDef>
```

In addition, a parameter file is used for the transformation to indicate which language is currently wanted and the file name of the feature model:

```
<parameterRoot>
    <currentLang>C++</currentLang>
    <sourceTypedModel name="languageSpec" file="LanguageDef.xml"/>
</parameterRoot>
```

HTML model to plain text. A pre-defined simple XSLT stylesheet of about 20 lines of XSLT code is used to convert the HTML model of the program code into a plain text file.

3 Experiments and Evaluation

Using the QVTR-XSLT code generator, we load the QVT-R transformation model and generate for each transformation a XSLT stylesheet. Some measures of the transformations, such as lines of generated XSLT code, development efforts, and model modularity, are shown in Table 1.

Table 1: Measures of the transformations.

Name	Number of relations /queries/functions	Lines of XSLT code	Develop person-hours	Modularity
TextToFix	3	81	3	0
FIXtoOO	6/3/1	181	10	- 0.2
SortOO	1/3/3	117	7	0
OOtoLang	10/6/1	444	20	- 0.56
Total	20/12/5	857	40	- 0.31

With the transformation runner, we load and execute a batch file that chains all the transformations, as well as individual XSLT transformations, on the examples provided by the case study in a laptop of Intel M330 2.13 GHz CPU, 3 GB memory, and running Windows 7 Home. The sizes of examples and the execution times for generating C++ code are shown in Table 2. The execution time includes loading and saving model files from/to disk. The DTD definition (second line) of test4.xml has to be removed first. Examples test7 and test8 are rejected because they are invalid XML files.

Table 2: Experimental results

Example	Size (kb)	Batch (ms)	TextToFIX (ms)	FIXtoOO (ms)	OOtoLang (ms)
test1	0.65	16	< 1	< 1	15
test2	0.92	31	< 1	15	16
test3	0.56	25	< 1	8	16
test4	0.83	47	< 1	16	31
test5	5.0	265	3	120	141
test6	12.4	1200	15	590	593

The generated programs are syntactically correct by checked in the IDEs of corresponding languages. For *test1* and *test2*, comparing the generated programs with the program text files provided by the case study shows equivalent structure. We also manually verify the generated program code with the original XML examples. So there is a high confidence that the transformations produce semantics preserving results. As we can see from Table 2, the solution works well, but the transformation algorithm also needs to be optimized to convert larger models more efficiently.

Conclusion

We presented a solution for the ”FIXML to Java, C# and C++” case study of TTC 2014. Our solution is founded on the standards introduced by OMG and W3C, and makes use of well-known and commonly adopted CASE tools and languages. We hope the case study will help to demonstrate that the graphical notation of QVT-R, a combination of UML object diagrams and essential OCL expressions, as well as the QVTR-XSLT tool, can be efficiently applied to model transformations in practice.

References

[1] Dan Li, Xiaoshan Li & Volker Stolz (2011): *QVT-based model transformation using XSLT. ACM SIGSOFT Softw. Eng. Notes* 36, pp. 1–8, doi:10.1145/1921532.1921563.

[2] Dan Li, Xiaoshan Li & Volker Stolz (2012): *Model querying with graphical notation of QVT relations. ACM SIGSOFT Softw. Eng. Notes* 37(4), pp. 1–8, doi:10.1145/2237796.2237808.

[3] Object Management Group (2011): *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.1*.

[4] WWW Consortium (2007): *XSL Transformations (XSLT) Version 2.0, W3C Recommendation*. Available at http://www.w3.org/TR/2007/REC-xslt20-20070123/.