A Solution to the FIXML Case Study using Triple Graph Grammars and eMoflon

Géza Kulcsár
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany

Erhan Leblebici
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany

Anthony Anjorin
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany

{geza.kulcsar|erhan.leblebici|anthony.anjorin}@es.tu-darmstadt.de

Triple Graph Grammars (TGGs) are a bidirectional model transformation language, which has been successfully used in different application scenarios over the years.

Our solution for the *FIXML case study* of the Transformation Tool Contest (TTC 2014) is implemented using TGGs and eMoflon (`www.emoflon.org`), a meta-modelling and model transformation tool developed at the Real-Time Systems Lab of TU Darmstadt.

The solution, available as a virtual machine hosted on Share [5], includes the following: (i) an XML parser to a generic tree model called *MocaTree* (already a built-in feature of eMoflon), (ii) a *target meta-model* specification, (iii) TGG rules describing a bidirectional transformation between MocaTree and the target meta-model, and (iv) a StringTemplate-based (`www.stringtemplate.org`) code generator for Java, C# and C++.

1 Introduction

Triple Graph Grammars (TGGs) [4] are a rule-based, declarative language, used for specifying transformations, where both directions (*forward* and *backward*) can be derived from the same specification.

The FIXML case study [3] is a text-to-text transformation based on the FIX (Financial Information eXchange) message format and its XML representation. The target format of the transformation is object-oriented code representing the same data structure originally expressed by the input FIXML data.

Such applications, where an input (tree- or graph-like) model should be transformed to another structure according to some mapping between the elements, are effectively solved using TGGs. Additionally, given such a transformation, consisting of a set of TGG *rules*, a correspondence model representing traceability links between the source and target model instances is also maintained.

In this paper, we present the latest TGG-features provided by eMoflon by solving the FIXML case study of TTC 2014 and evaluating our solution. Using the solution, we demonstrate a relatively new TGG modularity concept, *rule refinement* and show what can be achieved with it.

2 Solution With Triple Graph Grammars

The case study consists of the following steps: (i) parsing the XML input data into an instance of a *source meta-model* for a tree of nodes with attributes, (ii) transforming the source model using TGGs into an instance of a self-specified target meta-model tailored to the needs of object-oriented languages, and (iii) generating code in Java, C# and C++ from the target model using StringTemplate. In the following, details of the implementation of each step are given.

Submitted to:
TTC 2014

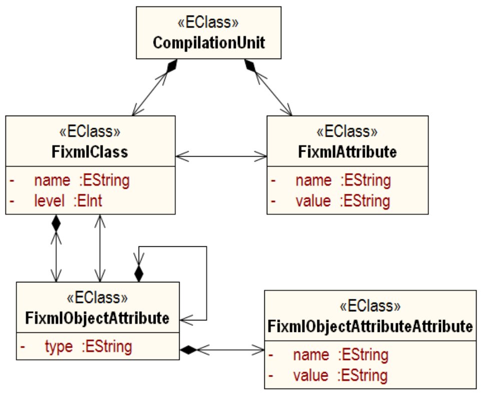

Figure 1: Target meta-model

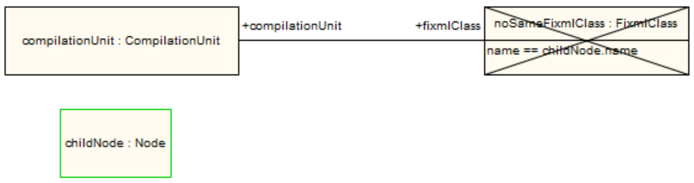

Figure 2: `First` rule

2.1 Step I: XML to Source

For this transformation, eMoflon already provides an XML adapter (a parser and an unparser) which, given an XML tree, can create an instance of a generic tree meta-model called MocaTree. The structure of a MocaTree is the following: (i) it may have a `Folder` as root element (not obligatory), (ii) a `Folder` can contain `Files` (a `File` can be root as well), (iii) a `File` is a container of `Nodes`, and (iv) a `Node` can have `Attributes`. In our transformation, there is always a single `File` root representing the file containing the XML data, the XML tags are the `Nodes` of the tree and XML attributes become `Attributes` of the corresponding `Node`.

2.2 Step II: Source to Target

This part of the transformation is implemented with TGGs. A TGG consists of a set of *rules* which describe how two models (instantiating two different meta-models) are built up simultaneously. The mediator graph describing the mapping between source and target model elements is called *correspondence graph*. Such a rule set immediately defines both source-to-target and target-to-source transformations. A rule prescribes the context (model parts that have to exist before rule application) and the elements that are added to the models and the correspondence graph during rule application. In this sense, TGG transformations are *monotonic* (do not delete).

Our target meta-model, chosen to fit the object-oriented structure of our desired output is depicted in Fig. 1. Our `CompilationUnit` class serves as the root container can contain `FixmlClasses` and `FixmlAttributes` referencing each other. Those class attributes, which are not single variables but are also objects themselves are contained by the corresponding class as `FixmlObjectAttribute`. Besides this containment reference, they also need another one showing which class they instantiate. They also have a containment self-reference as an object containment chain can be arbitrarily long. Finally, `FixmlObjectAttribute` can contain `FixmlObjectAttributeAttributes` – although classes already contain the attribute list, actual member object instances may have different values which we have to include in the model. (`FixmlAttributes` and `FixmlObjectAttributeAttributes` are technically the same - they have been separated only for the purpose of code generation as they are handled differently.)

Designing a TGG begins by identifying the semantic correspondences between model elements. As the TGG language is fully declarative, rules have to be declared so that they are applied only in the intended context and a sequence of rule applications always results in a correct model.

Regarding the case study, we can conceive our transformation as fulfilling two tasks simultaneously: building a rooted tree of (attributed) object attributes, i.e., expanding the model vertically, and for all child nodes, creating a class if it does not exist, i.e., expanding the model horizontally.

In our experience, that is the main challenge when realizing this transformation with TGGs: TGG rules translate each model element only once, so rules have to be formulated in such a way that all corresponding elements (in both forward and backward directions) have to be immediately added to the target model if the context for processing a new source element (node or attribute) is present. This requirement of a transformation with TGGs calls for carefully specified rule contexts.

The TGG Rules. We can examine the tasks of translating child nodes and translating attributes separately. In the following, FIXML classes are simply referred to as *classes*.

Using rule refinement, one is able to specify the common parts of TGG rules as separate rules, and then later derive the actual rules of it using a kind of inheritance, where the inheriting rule has to contain only what differentiates it from its ancestor. This results in more rules but a decreased amount of objects within rules what makes them more comprehensible. In addition, the rule diagram showing inheritances reflects the logical structure of the TGG.

Another advantage is that rule refinement relies on the existing TGG rule pattern syntax as opposed to, e.g., a template-based solution which would result in an additional layer on top of the TGG specification. The technique is flexible and has only a limited amount of restrictions; we have to note that this can also lead to misuse and, thus, overcomplicated diagrams. This is the general drawback of refinements: their application is not always trivial and getting used to thinking in refinement diagrams requires some practice. Finally, while the resulting overview diagram can be a valuable tool for maintenance, it might be challenging to design it.

For further details on rule refinement, we refer to [1] and our eMoflon handbook [2]. In the following, a semantic description of our rule set is given and the rule diagram of our implementation is shown; the detailed presentation of the single rules is omitted because of space restrictions.

Root rule. Our first rule is straightforward: we have to map the XML tag right after the `<FIXML>` element to a `FixmlClass` contained by a `CompilationUnit`.

Attribute rules. This task can be covered with two rules. A *Level 0* attribute rule simply maps all attributes of the root node in source to a FIXML attribute of the root class.

Each attribute in lower levels (*Level N*) has to be mapped to an *object attribute attribute* of the parent object attribute and a FIXML attribute of the corresponding referred class (as in the Level 0 case). This mapping can be specified with a TGG rule which inherits from a first level attribute rule.

Node rules. We have to separate the nodes along two dimensions when specifying the rules we need for handling nodes: (1) if the node processed is *Level 0* (direct descendant of the root node) or *Level N* and (2) if it is the *first* occurrence of this node name or not (*rest*). We always have to create a new object attribute for a node and a new class, if there is no existing class for this type of node; Level 0 object attributes are direct descendants of a class, while Level N ones are children of another object attribute.

Figure 2 shows the `First` rule, one of the abstract rules, which specifies the context for a first occurrence of a node. The black boxes (in the upper part) represent the *context* in the visual syntax of eMoflon. Green boxes (`childNode` in the lower part) are the newly created elements. A crossed-out box means a negative application condition: the object can not be part of the context. This (abstract) rule requires, that when a new node from the source is processed, there is no FIXML class already there with the same name. (In this rule, there are no correspondence links directly present.)

Rule diagram. The diagram how the transformation rule set has been implemented can be seen in Figure 3. Using rule refinement, it can be specified which rules should be actually generated from the description for transformation purposes, avoiding having too general rules included in the transformation

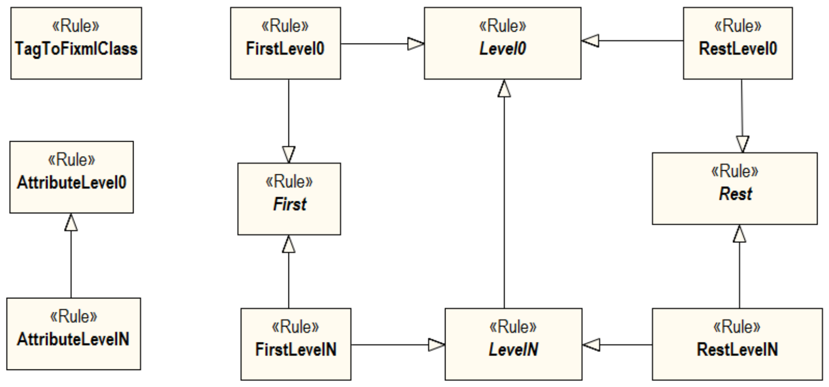

Figure 3: Rule diagram for source-to-target model transformation

system (the names of abstract rules are in italics).

2.3 Step III: Target to Code

This step has been implemented using StringTemplate, a template language and engine for generating source code. In general, StringTemplate is a very simple, minimalistic template language, that enforces a strict separation between logic (i.e., the actual transformation) and a view of the model. This fits well to our approach as we focus on TGG rules and handle the complexity of the transformation there and not in the templates.

Although our target model resembles our expectations of a model representing structural information in object-oriented format, it still has to be post-processed in order to (1) contain empty attributes where an object attribute has less contained attributes and/or object attributes than the class it instantiates and (2) deleting multiple neighbour attributes of the same name.

2.4 A TGG Advantage: Backward Transformation

In our solution, we also included another operation for demonstrating one of the advantages of TGGs: without any further efforts, a backward transformation on the target outputs of the given test cases can be performed. Utilizing the built-in XML unparser of eMoflon, we are capable of recreating the original XML input of the transformation. This step included in the solution is solely for demonstration purposes, but the backward transformation provided here could be actually applied in a possible application where actual model instances are, for instance, refactored and are to be translated back to FIXML descriptions.

3 Evaluation

In this chapter, we give an evaluation of our solution, taking the different aspects, as specified by the case study, into consideration.

The *abstraction level* of the solution is high as the main transformation is specified in a fully declarative way. Its *complexity* can not be evaluated in this context as it is not clear how ”operator” and ”reference” should be interpreted in our TGG visual syntax. The resulting program code is *syntactically and semantically correct*, although it is not directly compilable as some language-specific details have

	`test1.xml`	`test2.xml`	`test3.xml`	`test4.xml`	`test5.xml`	`test6.xml`
FWD	0.862	0.4265	0.2254	0.2172	2.6568	52.4381
BWD	0.3302	0.3064	0.119	0.2116	5.2227	73.1619

Table 1: Execution times for the test cases `test1.xml`-`test6.xml` in both directions

been omitted (as in the case study as well). The solution has been developed by a newcomer to TGGs, so the *development effort* was higher than usual: approx. 30 person-hours of which 25 was spent with the TGG specification and learning TGGs.

The solution is highly *fault tolerant* as it does not accept invalid XMLs as input and accurate error descriptions are shown.

The aspect of *modularity* can be addressed in our solution in the following way: r is the number of TGG rules and d is the number of inheritance arrows. This results in a modularity score of $1 - \frac{10}{11} \approx 0.1$.

The average execution times (in seconds) of 10 runs in our SHARE environment for the test cases `test1.xml` to `test6.xml` in forward (FWD) and backward (BWD) direction are summarized in Table 1.

4 Conclusion and Future Work

In this paper, we presented our solution for the FIXML case study of Transformation Tool Contest 2014 using Triple Graph Grammars (TGGs) using eMoflon. We demonstrated a relatively new TGG concept, rule refinement, which enables more structured TGG rule sets. In addition to the required transformation from FIXML to object-oriented code, we have also shown that a TGG specification immediately provides a backward transformation as well, allowing us to produce FIXML trees from target models.

Our future plans include performing scalability measurements based on our FIXML case study solution, which we can then use for identifying performance bottlenecks. The process of tailoring a transformation-based model generator to an already existing meta-model and a TGG may provide us important experience for a further goal: the ability to derive such model generators automatically from a TGG transformation.

References

[1] Anthony Anjorin, Karsten Saller, Malte Lochau & Andy Schürr (2014): *Modularizing Triple Graph Grammars Using Rule Refinement*. In: *FASE*, pp. 340–354.

[2] eMoflon online handbook (2014): *http://www.moflon.org/fileadmin/download/moflon-ide/eclipse-plugin/documents/release/eMoflonTutorial.pdf*.

[3] Kevin Lano, Sobhan Yassipour-Tehrani & Krikor Maroukian (2014): *Case study: FIXML to Java, C# and C++*.

[4] Andy Schürr (1994): *Specification of Graph Translators with Triple Graph Grammars*. In E. Mayr, G. Schmidt & G. Tinhofer, editors: *20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, *Lecture Notes in Computer Science (LNCS)* 903, Springer Verlag, Heidelberg, pp. 151–163.

[5] FIXML Transformation Solution with eMoflon hosted on Share (2014): *http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TGG-Comparison_eMoflon_08_05_2013_TTC14_eMoflon_FIXML.vdi*.