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
geza.kulcsar@es.tu-darmstadt.de

Erhan Leblebici
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany
erhan.leblebici@es.tu-darmstadt.de

Anthony Anjorin
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany
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 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 our target meta-model, and (iv) a StringTemplate-based (`www.stringtemplate.org`) code generator for Java, C# and C++. It is available as a virtual machine hosted on Share [5].

1 Introduction

Triple Graph Grammars (TGGs) [4] are a rule-based, declarative language, which can be used for specifying transformations, where both directions (*forward* and *backward* transformation) 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 whole transformation chain is object-oriented program code representing the same data structure that has been 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 between the source and target model instances is also maintained.

In this paper, we will present the features of TGGs and eMoflon by solving the FIXML case study of TTC 2014. For the solution, we demonstrate a relatively new TGG modularity concept, *rule refinement* and show what can be achieved by using it.

Submitted to:

2 Solution With Triple Graph Grammars

The transformation specified by the case study consists of the following steps in our approach: (i) parsing the XML input data into an instance of a *meta-model* for a tree of nodes with attributes (referred to as *source*), (ii) transforming the source using TGGs into a self-specified meta-model (referred to as *target*) tailored to the needs of object-oriented representation, and (iii) generating actual code in Java, C# and C++ from the target using StringTemplate .

In the following, details of the implementation of each step are given.

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:

- it may have a `Folder` as root element (not obligatory),
- a `Folder` can contain `Files` (a `File` can be root as well),
- a `File` is a container of `Nodes`, and
- 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 (in TGG convention, source and target are only designated labels for meta-models and do not refer to the direction of the transformation actually carried out). 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* (deletions never occur).

First, we have to define our target meta-model, which fits the object-oriented structure of our desired output, but still can be mapped to a tree structure via TGGs. The corresponding meta-model diagram can be seen in Fig. 1.

An EMF (Eclipse Modeling Framework) model has to be a single-rooted containment tree, hence our `CompilationUnit` class. It can contain `FixmlClasses` and `FixmlAttributes` referencing each other (contained attribute – containing class). 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.

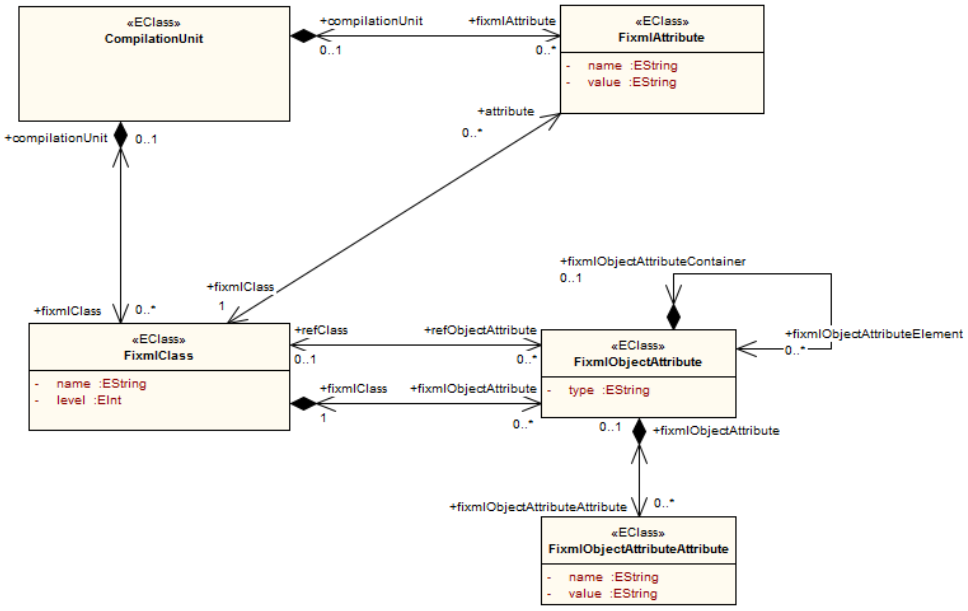

Figure 1: Target meta-model

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, it seems as if we would only have to translate a tree structure (MocaTree) to another, similar one with the single consideration that source child nodes become not only FIXML classes, but object attributes as well. The test input `test1.xml` enhances this feeling as it generates correct output with only three rules: root node to FIXML class, child nodes to FIXML classes and object attributes (recursive rule) and attributes to FIXML attributes.

While examining a more complex test (e.g., `test2.xml`), we realized that there is an additional concern when representing information contained by an XML data in an object-oriented manner: subtags (child nodes) may be of the same type, can contain different attribute values, and can have further member object attributes. These values and containments all have to appear in the generated code as parametrized constructors of their respective classes. This means that, in this respect, 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 order to keep rules as clean as possible, we can also take care of the ”first level” (classes and their direct members) and the ”1+ levels” (nested object attributes and their attributes) with separate rules. In the following, FIXML classes are simply referred to as *classes*.

The TGG rules required for this transformation have therefore a lot in common: for example, first level and 1+ level rules differ only slightly in context (at first level, we need a class with level 0 as parent object, at lower levels, we need an object attribute what represents the parent object of the one being processed). For such transformation tasks, the TGG modularization technique *rule refinement* can be applied effectively. 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.

Here, we only give an overview how rule refinement is applied to the case study. For further details and background, we refer to [1] our eMoflon handbook [2]. In the following, a semantic description of the required rules is given; after that, we present the rule diagram of an implementation using rule refinement and give an actual example of a TGG rule.

Root rule. Our first rule is straightforward: we have to map the XML tag right after the `<FIXML>` element (i.e., a direct child node of a node whose name is "FIXML") to a `FixmlClass` contained by a `CompilationUnit`.

First level child node rules. We can state the following of a child node of the root node: we have to create an object attribute of the root class and a new class, if there is no existing class for this type of node (i.e., no class with this name, as node names are used as class names); if there is already a class for the child node being processed, we only have to create a new object attribute in the root class. This can be expressed using two rules, which differ only regarding context class existence, so they fit for a refinement-based realization.

First level attribute rules. This task can be covered with only one rule, which simply maps all attributes of the root node in source to a FIXML attribute of the root class.

1+ level child node rules. On lower levels, we also have to differentiate the case where there is already a class created for the node being processed and where there is not (as we can discover new node types arbitrarily deep in the tree). If a `fixmlObjectAttribute` connected to the parent of the node being processed is part of the context, it also means that we are below level 1 without any further context elements.

In the case where there is no class present, the following target elements have to be created for a single child node in source: a new child object attribute of the context object attribute, a new object attribute of its referred class and the new class itself (referred by the new child object attribute). In the other case, a class obviously does not have to be created. As for multiple object attribute occurrences, a union of the contained object attributes is requested in the case study, an object attribute for the referred class should be created each time.

1+ level attribute rules. Each attribute in lower levels requires the same context as child nodes and has to be mapped to an *object attribute attribute* of the parent object attribute and a FIXML attribute of the corresponding referred class. This mapping can be specified with a single TGG rule, which can be moreover easily inherited from a first level attribute rule.

Rule diagram. The diagram how the transformation rule set has been implemented can be seen in Figure 2. 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 system.

Example: `Level0` *rule.* This rule (Figure 3) is a common ancestor of all the rules dealing with child nodes and therefore captures the commonalities of those.

The black boxes in the upper part of Fig. 3 represent the rule context: there is a(n already processed)

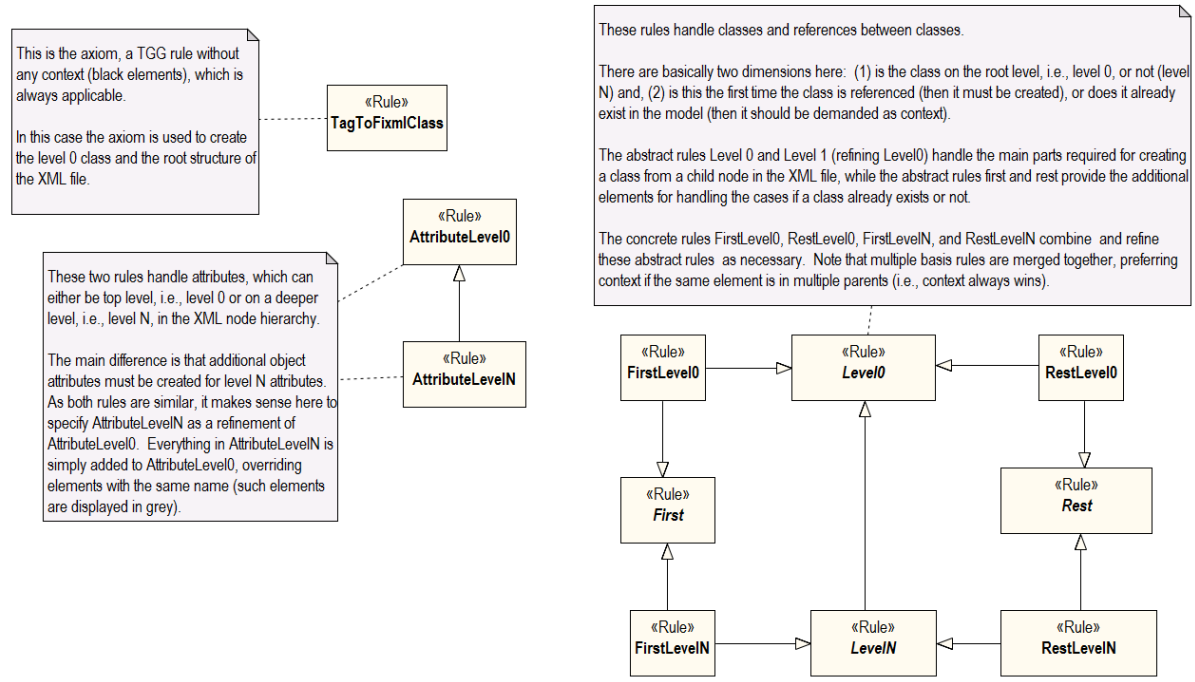

Figure 2: Rule diagram for our transformation including rule purpose description

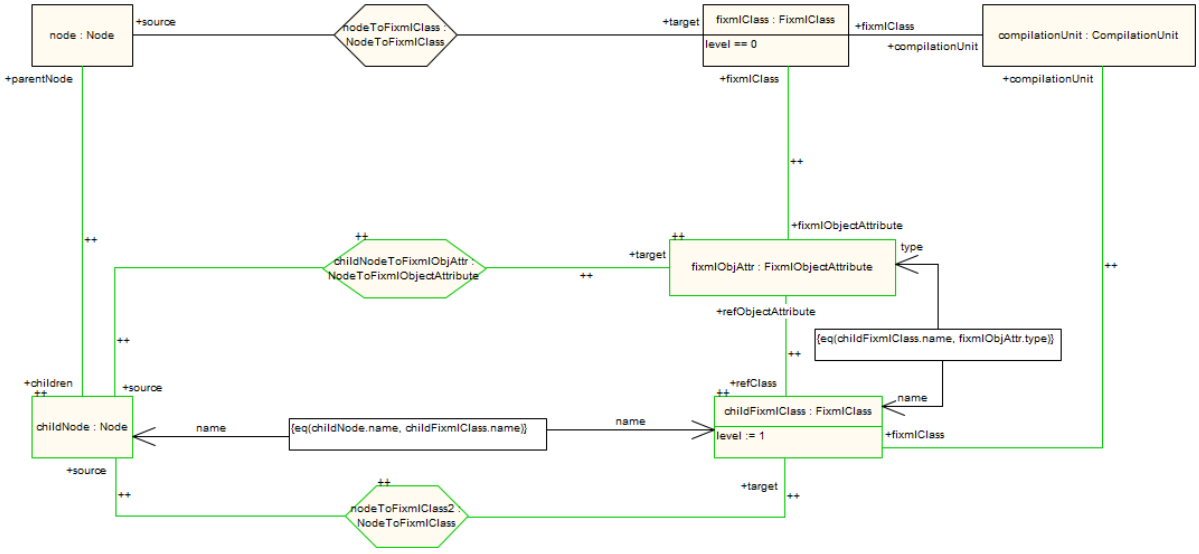

Figure 3: Rule `Level0`

node in the source that is represented by a class whose level is 0 (i.e., it is root). Hexagonal boxes represent the elements of the correspondence graph; source and target nodes connected to them are mapped to each other by the transformation. Green elements (also marked with '++') represent those which are created while applying the rule (technically, it means that during forwards transformation, if a non-processed element fitting the context is found in the input, it will be processed by adding the corresponding elements to both models). In this rule, we map an object attribute and a new class to a MocaTree node (these two elements are created in each variant of child node rules, therefore they are included in this abstract rule). The black boxes with one expression and two outgoing arrows each are *TGG constraints* that express constraints which the nodes have to fulfill. Here, they state that the names of the corresponding source and target objects have to be equal.

Although rule refinement can be a powerful tool for transformation designers when specifying TGG transformations with a lot of similar rules, the resulting rule set will be exactly the same as if we have been specified the required rules one by one without modularization.

2.3 Step III: target to code

This step has been implemented using StringTemplate, a template language and engine for generating source code. Templates consist of rules which can have parameters and define the output string, where given sub-strings are specified by the parameters passed to the template from the Java code where the generation is involved.

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 for containing some additional elements required for generating valid program code. In cases where an object attribute has less contained attributes and/or object attributes than the class it instantiates (i.e., its attribute list is incomplete), we have to create empty objects as "placeholders" for non-existing member elements, because we also have to include empty attributes and object attributes as parameters (empty strings or `null` objects, respectively) in the constructor calls generated for these object attributes. This case, where non-existent source objects should have actual correspondences in target, is a known limitation of TGGs and such requirements can not be directly formulated by TGG rules.

Another post-processing step is necessary for attributes, as we did not make a restriction if an attribute already exists in a class or not. This decision proved useful for keeping the transformation size reasonable and the transformation itself clean and correct, but can result in multiple attributes of the same name in the target classes. These multiple occurrences can be reduced to one attribute of each kind with a post-processing step.

Output is generated separately for each class created by the transformation. In our StringTemplate template, there exists a main rule for each languages, which receives the class, its attributes and its object attributes as parameters and then performs the code generation according to the syntax of the actual programming language.

In the case of C++, an additional consideration is necessary, i.e., we have to order the class outputs so that a class definition is always placed before a pointer to it is declared. We can easily handle this condition by using the `level` attribute of the classes: we have to take the classes for code generation in decreasing level order (classes on the same level can be processed in arbitrary order as they can not

contain each other).

2.4 A TGG advantage: backward transformation

In our solution, we also included another operation for demonstrating one of the advantages 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 target model instances should be translated to FIXML descriptions. By building a parser for object-oriented program code generating target models, actual Java, C# or C++ program structures could be translated to FIXML trees.

3 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) and eMoflon as tool for implementing the TGG. We demonstrated a relatively new TGG concept and eMoflon feature, rule refinement, which enables the creation of cleaner and more structured TGG rule sets with less objects within rules. In addition to the required transformation from an XML description to object-oriented code in different languages, we have also shown that a TGG transformation specification immediately provides a backward transformation as well, allowing us to produce proper FIXML trees (using the unparser of the XML adapter of eMoflon to create XML from MocaTree) from any given target model instances.

Our future plans therefore include building a target model generator (which task can be carried out using eMoflon as well) and performing scalability measurements for TGGs based on our FIXML case study solution, which we can then use for identifying performance bottlenecks in the TGG implementation of eMoflon. Our process of tailoring a transformation-based model generator to an already existing meta-model and a TGG transformation may provide us important experience for a farther goal: the ability to derive such generators automatically from a TGG transformation.

References

[1] Anthony Anjorin, Karsten Saller, Malte Lochau, and Andy Schürr. Modularizing triple graph grammars using rule refinement. In *FASE*, pages 340–354, 2014.

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

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

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

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