Solving the FIXML2Code-case study with HenshinTGG

Frank Hermann
Interdisciplinary Centre for Security, Reliability and Trust, Université du Luxembourg, Luxembourg
firstname.lastname@uni.lu

Nico Nachtigall
Interdisciplinary Centre for Security, Reliability and Trust, Université du Luxembourg, Luxembourg
firstname.lastname@uni.lu

Benjamin Braatz
Interdisciplinary Centre for Security, Reliability and Trust, Université du Luxembourg, Luxembourg
firstname.lastname@uni.lu

Thomas Engel
Interdisciplinary Centre for Security, Reliability and Trust, Université du Luxembourg, Luxembourg
firstname.lastname@uni.lu

Triple graph grammars (TGGs) provide a formal framework for bidirectional model transformations. As in practice, TGGs are primarily used in pure model-to-model transformation scenarios, tools for text-to-model and model-to-text transformations make them also applicable in text-to-text transformation contexts. This paper presents a solution for the text-to-text transformation case study of the transformation tool contest 2014 on translating FIXML (XML notation for financial transactions) to source code written in Java, C# or C++. The solution uses the HenshinTGG tool for specifying and executing model-to-model transformations based on the formal concept of TGGs as well as the Xtext tool for parsing xml content to yield its abstract syntax tree (text-to-model transformation) and serialising abstract syntax trees to source code (model-to-text transformation). The approach is evaluated concerning a given set of criteria.

1 Introduction

Triple graph grammars (TGGs) provide a formal framework for specifying consistent integrated models of source and target models in bidirectional model transformations where the correspondences between the elements of source and target models are defined by triple rules, from which operational rules for forward and backward transformations are derived automatically [9, 13]. Several tool implementations for TGGs exist [11] and numerous case studies have proven the applicability of TGGs in model-to-model (M2M) transformation scenarios [8, 5, 6]. In [10], we presented an approach for applying TGGs in a text-to-text (T2T) transformation context in order to translate satellite procedures. In this paper we adapt our approach from [10] to provide a solution for the T2T transformation case study of the transformation tool contest 2014 on translating *FIXML* (XML notation for financial transactions) into source code (Java, C# and C++) [12]. We evaluate the approach based on the following fixed criteria to enable a comparison of our results with other solutions: *a*) complexity, *b*) accuracy, *c*) development effort, *d*) fault tolerance, *e*) execution time, and *f*) modularity of the solution.

As depicted in Fig. 1, in contrast to pure M2M tranformations that are based on TGGs, T2T transformation involve two additional steps: *a*) A text-to-model (T2M) transformation step parses the text and yields its abstract syntax tree (AST) at first (parsing the content of a *FIXML* file to obtain the corresponding AST). Then, a M2M transformation is performed based on a given TGG to convert the source AST into the target AST (translation of a *FIXML* AST into a Java- C#- or C++-AST). *b*) Finally, the target AST is serialised back to text via a model-to-text (M2T) transformation (serialisation of a Java- C#- or C++-AST to source code). In practice, the main challenge is to combine these three steps to a chain of

Submitted to:
TTC 2014

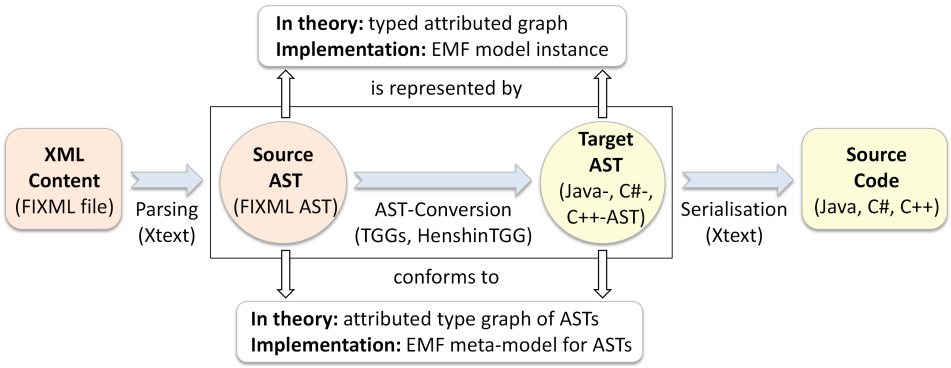

Figure 1: Main phases for the T2T-translation (Text-To-Text)

transformations (T2M, M2M, M2T). We combine *Xtext* [2] with the *HenshinTGG* tool to perform the T2M and M2T steps via *Xtext* and the M2M step via *HenshinTGG*. *Xtext* is a tool for specifying domain specific textual languages and automatically generating parsers and serialisers for them. *HenshinTGG* is an extension of the EMF-Henshin tool [3] and is used for specifying and executing M2M transformations based on the formal concept of TGGs. The solution also has been made available on SHARE[1].

The paper is structured as follows. Sec. 2 introduces TGGs for bidirectional model transformations and describes the TGG tool implementation HenshinTGG, Sec. 3 presents the details of our solution for the case study, Sec. 4 evaluates the solution concerning the given criteria and Sec. 5 provides a conclusion and describes potential extensions.

2 Triple Graph Grammars and HenshinTGG

The main part of the solution involves the *AST-conversion*, i.e., the specification and execution of the M2M transformation from *FIXML* ASTs to ASTs of Java, C# or C++ sourc code. ASTs are specified by attributed graphs [4] (instance graphs) that are typed over a meta-model (attributed type graph with multiplicity constraints) which defines the allowed structure of the instance graphs (cf. Fig. 1). Fig. 2 (right) depicts the meta-model of *FIXML* ASTs.[2] Each *FIXML* AST has a root node of type `Model` with at most one `Header` node that is connected by a `header` edge and with a number of `XMLNodes` that are connected by edges of type `nodes`. The `Header` contains the XML declaration of a *FIXML* file and each `XMLNode` represents a XML empty-element-tag (⟨tag /⟩) or a XML start-`tag` (⟨tag⟩) together with its matching XML end-`tag` (⟨/tag⟩) of a *FIXML* file. `XMLNodes` may have several child elements, i.e., plain text content (attribute `entry`) or a number of XML `subnodes`. Furthermore, each `Header` and `XMLNode` may have several XML `Attributes` of a specific `name` and with a certain `value`. The conversion of *FIXML* ASTs to source code ASTs is performed based on the concept of TGGs that are used for bidirectional M2M transformations.

We briefly review some basic notations for TGGs. Note that the case study of this paper does not use the backward transformation (source code to *FIXML*), but the forward transformation only. However, TGGs still provide an intuitive framework that supports the designer to keep the transformation concise, flexible and maintainable.

[1] http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession...

[2] The meta-model may also serve as a meta-model for the abstract syntax of general XML.

The correspondences between elements of a *FIXML* AST and an AST of source code are made explicit by a triple graph. A triple graph is an integrated model consisting of a source model (*FIXML* AST), a target model (AST of source code) and explicit correspondences between them. More precisely, it consists of three graphs G^S, G^C, and G^T, called source, correspondence, and target graphs, respectively, together with two mappings (graph morphisms) $s_G\colon G^C \to G^S$ and $t_G\colon G^C \to G^T$. The two mappings in G specify a correspondence relation between elements of G^S and elements of G^T.

The correspondences between elements of *FIXML* ASTs and elements of ASTs of source code are specified by triple rules. A triple rule $tr = (tr^S, tr^C, tr^T)$ is an inclusion of triple graphs $tr\colon L \to R$ from the left-hand side $L = L^S \xleftarrow{s_L} L^C \xrightarrow{t_L} L^T$ to the right-hand side $R = R^S \xleftarrow{s_R} R^C \xrightarrow{t_R} R^T$ with $(tr^i\colon L^i \to R^i)_{i\in\{S,C,T\}}, s_R \circ tr^C = tr^S \circ s_L$ and $t_R \circ tr^C = tr^T \circ t_L$. This implies that triple rules do not delete, which ensures that the derived operational rules for the translation do not modify the given input. A triple rule specifies how a given consistent integrated model can be extended simultaneously on all three components yielding again a consistent integrated model. Intuitively, a triple rule specifies a fragment of the source language and its corresponding fragment in the target language together with the links to relevant context elements. A triple rule $tr\colon L \to R$ is applied to a triple graph G via a match morphism $m : L \to G$ resulting in the triple graph H, where L is replaced by R in G. Technically, the transformation step is defined by a pushout diagram [4] and we denote the step by $G \overset{tr,m}{\Longrightarrow} H$. Moreover, triple rules can be extended by negative application conditions (NACs) for restricting their application to specific matches [9, 7]. Thus, NACs can ensure that the rules are only applied in the right contexts. A triple graph grammar $TGG = (TG, S, TR)$ consists of a type triple graph TG, a start triple graph S and a set TR of triple rules, and generates the triple graph language $L(TGG) \subseteq L(TG)$ containing all consistent integrated models. In general, we assume the start graph to be empty in model transformations. For the case study, the type triple graph consists of the type graph for *FIXML* ASTs (cf. Fig. 2 (right)) as the source graph, the type graph for ASTs of source code as the target graph and a correspondence graph containing one node that maps the model elements from source to target.

Example 1 (Triple Rules) *An example of a triple rule of the TGG is presented in Fig. 4. The figure shows an adapted screenshot of the HenshinTGG tool [3] using short notation. Left- and right-hand side of a rule are depicted in one triple graph and the elements to be created have the label* $\langle{+}{+}\rangle$*. The three components of the triple rule are separated by vertical bars, i.e. the source, correspondence and target graphs are visualised from left to right. The rule creates a* `XMLNode` *in the source and its corresponding class (*`class_def` *node) in the target that is linked to an existing* `Model` *node as context element. The correspondence is established via the* `CORR` *node. The NAC* `ClassNameUnique` *ensures that the rule is only applicable if there does not already exist a class with the same name. In view of the other rules for this case study, the depicted rule is of average rule size.*

The operational forward translation rules (FT-rule) for executing forward model transformations are derived automatically from the TGG [9]. A forward translation rule tr_{FT} and its original triple rule tr differ only on the source component. Each source element (node, edge or attribute) is extended by a Boolean valued translation attribute $\langle\texttt{tr}\rangle$. A source element that is created by tr is preserved by tr_{FT} and the translation attribute is changed from $\langle\texttt{tr}\rangle = false$ to $\langle\texttt{tr}\rangle = true$. All preserved source elements in tr are preserved by tr_{FT} and their translation attributes stay unchanged with $\langle\texttt{tr}\rangle = true$.

Example 2 (Operational Translation Rules) *Fig. 5 depicts the corresponding forward translation rule of the triple rule in Fig. 4. The elements to be created are labelled with* $\langle{+}{+}\rangle$ *and translation attributes that change their values are indicated by label* $\langle\texttt{tr}\rangle$*.* △

A forward model transformation is executed by initially marking all elements of the given source model G^S with $\langle\texttt{tr}\rangle = false$ leading to G'^S and applying the forward translation rules as long as possible.

```
1  grammar lu.uni.snt.ttc2014.FIXML
2          with org.eclipse.xtext.common.Terminals
3  generate fIXML "http://www.uni.lu/snt/ttc2014/FIXML"
4  Model      : header=Header? nodes+=XMLNode*;
5  Header     : {Header} "<" "?" "xml"
6               attributes+=Attribute* "?" ">"
7               ("<" "!" ID* STRING* ">")?;
8  XMLNode    : "<" tag=ID attributes+=Attribute*
9               ( ("/" ">") |
10                (">" subnodes+=XMLNode*
11                     entry=Entry? "<" "/" end=ID  ">")
12              );
13 Attribute  : name=ID "=" value=STRING;
14 Entry      : (ID|INT|WS|ANY_OTHER|
15               ML_COMMENT|SL_COMMENT|STRING)+;
```

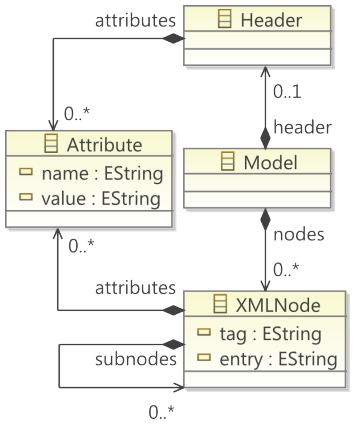

Figure 2: Xtext grammar for *FIXML* parser (left) and corresponding *FIXML* meta-model (right)

Formally, a *forward translation sequence* $(G^S, G_0 \overset{tr^*_{FT}}{\Longrightarrow} G_n, G^T)$ is given by an input source model G^S, a transformation sequence $G_0 \overset{tr^*_{FT}}{\Longrightarrow} G_n$ obtained by executing the forward translation rules TR_{FT} on $G_0 = (G'^S \leftarrow \varnothing \rightarrow \varnothing)$, and the resulting target model G^T obtained as restriction to the target component of triple graph $G_n = (G^S_n \leftarrow G^C_n \rightarrow G^T_n)$ with $G^T = G^T_n$. A *model transformation* based on forward translation rules MT: $\mathscr{L}(TG^S) \Rrightarrow \mathscr{L}(TG^T)$ consists of all forward translation sequences with TG^S and TG^T being the restriction of the triple type graph TG to the soure or target componenet, respectively. Note that a given source model G^S may correspond to different target models G^T. In order to ensure unique results, we presented in [9] how to use the automated conflict analysis engine of AGG [1] for checking functional behaviour of model transformations.

In order to perform the M2M transformation from *FIXML* ASTs to source code ASTs, we use the TGG implementation *HenshinTGG* for model transformations which is an extension of the plain graph transformation tool Henshin [3]. *HenshinTGG* is an Eclipse plugin providing a graphical development and simulation environment for TGGs allowing the specification of triple graphs and triple rules. Furthermore, it provides an extended transformation engine for executing different kinds of TGG operations, e.g., automatic derivation of operational forward and backward translation rules from the triple rules of a given TGG and their application to triple graphs for bidirectional model transformations.

3 Solution

As already introduced in Sec. 1, we applied the general concept for the T2T-translation depicted in Fig. 1 which is adapted from the approach we presented in [10] for translating satellite procedures. It consists of the phases *parsing*, *AST-conversion* (main phase), and *serialisation* and is executed using the Eclipse Modeling Framework (EMF) tools *Xtext* and *HenshinTGG*. Xtext supports the syntax specification of textual domain specific languages (DSLs). Based on the EBNF (Extended Backus-Naur Form) grammar specification of a DSL and an optional formatting configuration, the Xtext framework generates the corresponding parser and serialiser. The parser checks that the input source code is well-formed and the serialiser ensures that the generated output source code is well-defined.

3.1 Parser for *FIXML* ASTs

Fig. 2 (left) depicts the Xtext EBNF grammar of input DSL *FIXML*. Each rule of the grammar is identified by a non-terminal symbol separated by a colon from the rule specification body and ends with a semicolon. The root rule `Model` specifies that each *FIXML* file has one optional XML `Header` (line 4) and contains a number of `XMLNodes`. Note that references between rules can be equipped with names, e.g., the reference from rule `Model` to `Header` has the name `header` (`header` = `Header`?). A `Header` contains the content of a usual XML header together with a number of `Attributes` (line 6). A `XMLNode` is an empty-element-tag ($\langle$ID /$\rangle$) of name `ID` and with a number of `Attributes` (lines 8 & 9) or a start-tag ($\langle <$ `ID` $>\rangle$) of name `ID` and with a number of `Attributes` together with its corresponding end-tag ($\langle <$ /`ID` $>\rangle$) (lines 8,10 & 11). `ID`s are imported by line 2 and allow an arbitrary string as terminal that starts with a character or an underline symbol. Note that start-tags and their corresponding end-tags may have different tag names, since, `tag` and `end` allow arbitrary `ID`s. Therefore, in Sec. 4 we introduce an additional Xtext constraint that claims that each start-tag has the name of its corresponding end-tag. `XMLNodes` may have several child nodes (reference `subnodes` in line 10) as well as optional plain text content of type `Entry` (line 11). An `Entry` is a terminal that comprises combinations of the following terminals: `ID`s, Integers, whitespaces (`WS`), any character symbol (`ANY_OTHER`), comments and arbitrary `STRING`s (these rules are imported in line 2). An `Attribute` has a `name` of type `ID` and a value of type `STRING` that are linked by an equality symbol (line 13). Rules that only produce atomic terminals are called terminal rules whereas the other rules are called parser rules.

From the grammar, *Xtext* automatically generates the EMF meta-model in Fig. 2 (right) which serves as the meta-model for *FIXML* ASTs. Each parser rule becomes a node except for `Entry`, since, `Entry` only has unnamed references to terminal rules. Each named reference between two parser rules becomes an edge between the corresponding two nodes, e.g., edge `header` between nodes `Model` and `Header`. Each named reference between a parser and a terminal rule becomes an attribute of the corresponding node, e.g., attribute `tag` of node `XMLNode`. Furthermore, *Xtext* generates a syntax highlighting editor with auto code completion for *FIXML* content and the generic parser that creates *FIXML* ASTs (EMF model instances) from *FIXML* content. Note that in theory meta-models for ASTs are represented by attributed type graphs and ASTs by typed attributed graphs but they are implemented by EMF meta-models or EMF model instances, respectively (cf. Fig. 1). *Xtext* also provides a serialiser from EMF model instances to *FIXML* content, which we do not use in our translation scenario.

3.2 Serialiser for Java ASTs

Analogously to the parser in Sec. 3.1, the meta-model for Java ASTs (EMF meta-model) and the serialiser from Java ASTs (EMF model instances) to java source code are generated from the Xtext grammar for Java listed in Fig. 3. Note that we only consider that subset of Java which is relevant for the translation. Java source code may include several imports and class definitions (line 4). A class contains a `name` of type `ID` together with a set of `declarations` as `initialDeclarations` and a set of method definitions (`method_def`) (lines 6 & 7). A `declaration` contains an `ID` as `type`, an optional generic `typeParameter`, a variable `name` of type `ID` and a `defaultValue` which can be any expression (lines 10 & 11). An expression (`exp`) is either atomic, a constructor call or a method call (line 15). An atomic expression (`atom`) has a `value` of type `STRING` or `INT` or is the `name` of a variable (line 22). A constructor call (`constructor_call`) contains the terminal `new` together with a method call (line 16). A method call (`methodCall`) contains the `name` of the method and an optional generic `typeParameter` (line 17). A method definition (`method_def`) contains a `name`, a list of `arguments` and a `body` (lines 18 & 19). A

```
1  grammar lu.uni.snt.secan.ttc_java.TTC_Java with org.eclipse.xtext.common.
       Terminals
2  generate tTC_Java "http://www.uni.lu/snt/secan/ttc_java/TTC_Java"
3
4  Model                    : imports+=import_* classes+=class_def*;
5  import_                  : "import" entry=fully_qualified_name ";";
6  class_def                : "class" name=ID "{" initialDeclarations+=stmt*
7                                      => feature+=feature*  "}";
8  feature                  : stmt | method_def;
9  stmt                     : (declaration | assignment)  ";";
10 declaration              : type=ID typeParameter=typeParameter? name=ID
11                                     "=" defaultValue=exp;
12 typeParameter            : ("<" typeP=ID ">");
13 assignment               : var=fully_qualified_name "=" exp=exp;
14 fully_qualified_name     : (ID ("." ID)*);
15 exp                      : atom | constructor_call | methodCall;
16 constructor_call         : "new" method=methodCall;
17 methodCall               : name=ID typeP=typeParameter? "(" ")";
18 method_def               : name=ID "(" (args+=argument ("," args+=argument)
       *)? ")"
19                                     "{" body=body "}";
20 body                     : {body} (stmts+=stmt)*;
21 argument                 : type=ID typeP=typeParameter? name=ID;
22 atom                     : string_val | int_val | variable_name;
23 variable_name            : name=ID;
24 string_val               : value=STRING;
25 int_val                  : value=INT;
```

Figure 3: Xtext grammar for Java serialiser

`body` is a list of statements (`stmt`) (line 20). An argument is of a certain `type` with an optional generic `typeParameter` and has a `name` (line 21).

Adaptations for supporting C# and C++ The presented solution concerns the output language Java only. We are confident that the Xtext grammar above could be generalised to a grammar that is capable to handle also C# and C++. The distinction which language specific tokens would be used can be defined in the Xtext formatter specification. Thus, the presented solution seems to be flexible enough to enable a smooth extension of the serialiser to ouput languages C# and C++, e.g., in Fig. 3 lines 6 & 7, we can add terminals `private` : and `public` : in front of `initialDeclarations` and `feature` to mark the block of variable declarations as private and the block of methods as public in C++.

3.3 M2M Transformation

As the main part of the solution involves the specification and execution of the M2M transformation from *FIXML* ASTs to source code ASTs, we present core forward translation rules for converting *FIXML* ASTs to Java ASTs in this section. Fig. 4 depicts a screenshot of triple rule *Tag2Class* as specified in the HenshinTGG tool and Fig. 5 shows the corresponding forward translation rule that is derived automatically from the triple rule with HenshinTGG. For all other derived forward translation rules in this section, the underlying triple rules are not shown explicitly, since, they can be easily reconstructed from the forward translation rules (cf. Sec. 2). The forward translation rules are applied with HenshinTGG in

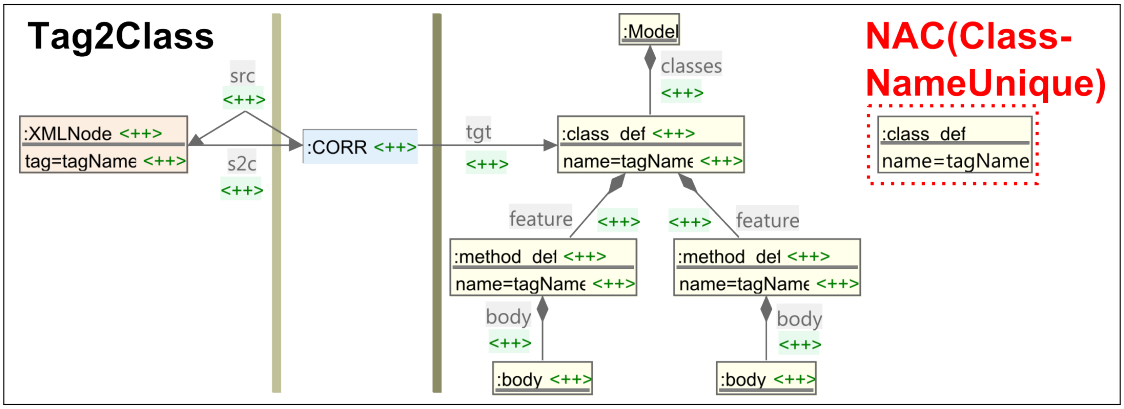

Figure 4: Triple rule *Tag2Class*

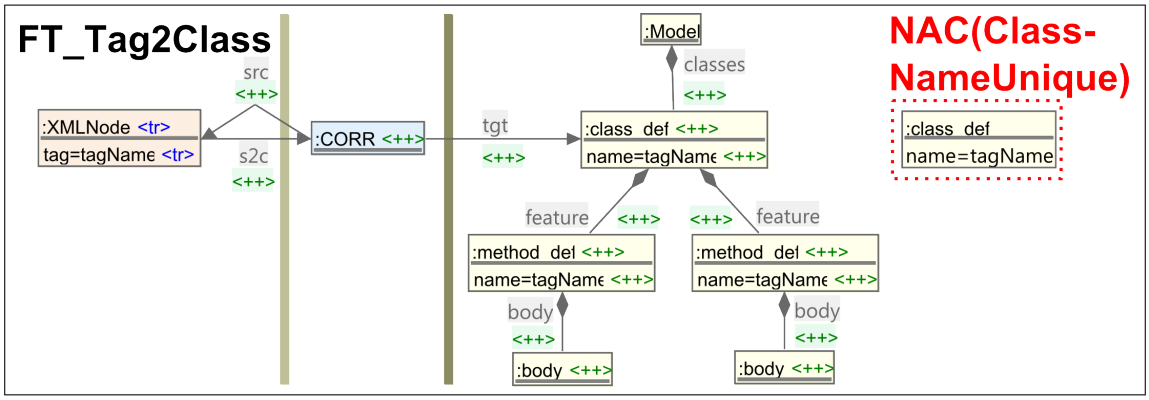

Figure 5: Generated FT-rule *FT_Tag2Class* from triple rule *Tag2Class*

order to convert the ASTs. The rules include the following design decisions.

Design decisions *FIXML* input files may contain lists of XML tags with the same name. In our solution, all these list elements are visited and all occcurring features of these tags are integrated within the class definition. It would have been possible to use the input list as an example list for the empty constructor of the class definition. However, we assume that the best solution is to have an empty constructor that creates initially empty lists. In our view, any content in the list created by the empty constructor would be non-intuitive for the user of the generated Java code. We took the design decision in our implemented solution to create empty lists within the empty constructors.

Forward translation rule *FT_Tag2Class* specifies the translation of a `XMLNode` with a certain `tagName` into a class (`class_def`) of the same `name` and links the created class to the root node `Model` of the target AST with edge `classes`. Furthermore, two constructors (`method_def` nodes having the name of the class) are created for the class with an empty `body` each. The rule is only applied if there does not already exist a class with the same name (NAC *ClassNameUnique*), i.e., if the *FIXML* file contains several XML nodes of the same name, only one of these nodes is translated by this rule. A second rule *FT_Tag2ExistingClass* translates the other XML nodes by creating correspondences between the XML nodes and the created class with *CORR* nodes only. After the translation of a `XMLNode` and its attribute

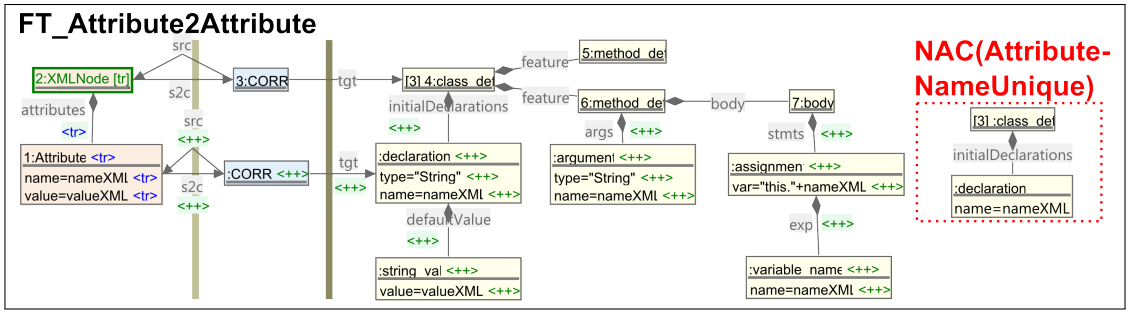

Figure 6: FT-rule *FT_Attribute2Attribute*

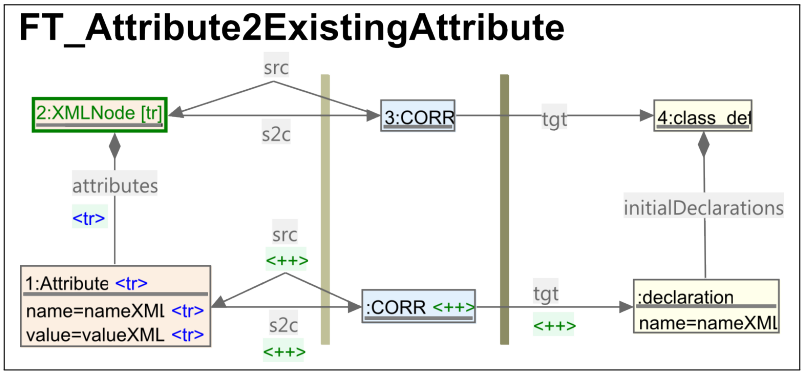

Figure 7: FT-Rule *FT_Attribute2ExistingAttribute*

tag, their translation attributes are set from $false$ to $true$ as indicated by labels ⟨`tr`⟩.

Rule *FT_Attribute2Attribute* takes an `XMLNode` that is already translated into a class and translates each XML `Attribute` with `nameXML` and `valueXML` of the `XMLNode` to a member variable (node of type `declaration`) of the class by linking the variable to the class with edge `initialDeclarations`. Already translated elements are indicated by labels [`tr`]. The created member variable has the same `name` and `value` as `defaultValue` than the XML `Attribute`. The `type` of the variable is set to `String`. Furthermore, the constructor of the class is extended by an `argument` of `type` `String` having the `name` of the created member variable. The `body` of the constructor is extended by an `assignment` which assigns the `argument` to the created member variable (`assignment` node). Note that HenshinTGG stores the nodes of graphs in lists accordingly to their mapping numbers, i.e., the same constructor (node `6 : method_def`) will always been matched for extension while the other constructor (node `5 : method_def`) stays unmodified and empty. In combination with rules *FT_Tag2Class* and *FT_Tag2ExistingClass*, this rule will collect all XML attributes of `XMLNodes` with a certain name and will append them to the corresponding class as member variables. The constructor is extended correspondingly. The rule is only applicable if there does not already exist a member variable of the same `nameXML` for the class (NAC *AttributeNameUnique*).

Similarly to rule *FT_Tag2Class*, if the *FIXML* file contains several XML tags of the same name that share XML attributes of the same name, only one of these attributes is translated by rule *FT_Attribute2Attribute* due to the NAC. Rule *FT_Attribute2ExistingAttribute* translates the other `Attributes` by creating correspondences between the `Attributes` and the created member variable (node of type `declaration`) with `CORR` nodes only.

Adaptations for supporting C# and C++ The `String` type in Java is written `string` in C# which can be accomplished easily by substituting `String` by `string` for `type` in rule *FT_Attribute2Attribute*. Similarly, the star symbol for pointers can be added to types in C++. For C++ compilers it is necessary to declare classes before they are used in other classes. A simple syntactical ordering of classes accordingly to their usage is not sufficient due to possible circular dependencies between classes. A simple solution would be to add an empty class declaration (`class className;`) for each class at the beginning of a C++ file. Rule *FT_Tag2Class* must be modified so that it additionally creates a declaration node for each class linked to the `Model` node. Furthermore, the Java Xtext grammar must be extended by a rule for declarations such that the declarations precede the class definitions syntactically. A separation from class declarations and implementations into header and implementation files is also realisable without great effort. The forward translation rules would maintain a $\texttt{Model}-\texttt{Header}$ node for the header file and a $\texttt{Model}-\texttt{Impl}$ node for the implementation file of the classes instead of one `Model` node only, i.e., C++ EMF model instances would contain not one but two ASTs that can be serialised into separate files.

4 Analysis

The approach is evaluated concerning the following criteria that are fixed for the case study [12].

Complexity The Xtext grammars for *FIXML* and Java together with the TGG are the specification of the T2T transformation. The TGG comprises 14 triple rules together containing 27 nodes, 14 node attributes and 12 edges in source graphs, 65 nodes, 40 node attributes and 48 edges in target graphs as well as 20 nodes in correspondence graphs (in total 112 nodes, 54 node attributes and 60 edges). Both Xtext grammars comprises 24 parser rules with 58 references between them.

Accuracy The syntactical correctness of the translation is ensured by its formal definition based on forward translation rules [9], i.e., each *FIXML* file that is completely translated yields source code that is correctly typed over the meta-model of the target programming language. The constraints of the target language (different classes have different names, etc.) are expressed by graph constraints and are translated to application conditions of the triple rules of the TGG [4]. This ensures that the translation of *FIXML* ASTs yields target ASTs only that fulfill the contraints, e.g., the target NAC *AttributeNameUnique* of rule *FT_Attribute2Attribute* in Fig. 6 ensures that different member variables of the same class have different names. Moreover, triple rules specify a fragment of the source language together with its corresponding fragment in the target language. This simplifies to check that source code fragments correctly represent the *FIXML* elements (semantic preservation).

Development effort The following person-hours were spent for writing und debugging the specifications: (a) *0.5 person-hours* for the Xtext grammar of the parser (cf. Sec. 3.1), (b) *1.0 person-hours* for the Xtext grammar of the serialiser (cf. Sec. 3.2), and (c) *2.5 person-hours* for the TGG of the M2M translation (cf. Sec. 3.3) (in total *4.0 person-hours*).

Fault tolerance The case study [12] contains two *FIXML* files, i.e., Test 7 & 8, that have syntax errors and should be identified as being invalid by the translation. The fault tolerance of our solution is classified as High, since, invalid *FIXML* input files that do not correspond to the *FIXML* Xtext grammar in Fig. 2 lead to Xtext parsing errors which are displayed on the console. Test 8 is successfully detected as being

invalid with the following error message, since, the *FIXML* grammar claims that each XML start-tag has a corresponding end-tag.

```
An internal error occurred during: "Translating test8.xml".
Translation failed. No output was generated.
The following syntax errors occured while parsing:
Line 19: mismatched input '<EOF>' expecting '<'
```

Syntactical restrictions that cannot be expressed by the grammar are defined by constraints in a custom Xtext validator for *FIXML* based on the elements of the grammar. We defined the constraint `checkXMLNodeHasStartEndTagsOfSameName` that claims that each start-tag (`xmlnode.tag`) has the name of its corresponding end-tag (`xmlnode.end`), i.e., `xmlnode.tag.equals(xmlnode.end)` (cf. Fig. 8). If the constraint is not satisfied, an error is raised. Test 7 does not satisfy this constraint and is classified as being invalid with the following error message.

```
An internal error occurred during: "Translating test7.xml".
Translation failed. No output was generated.
The following syntax errors occured while parsing:
Start-tag must have the name of its corresponding end-tag for start-tag (Hdr) and
end-tag (Order)
-------------------------------
Start-tag must have the name of its corresponding end-tag for start-tag (Sndr) and
end-tag (Hdr)
```

If syntax errors occur in *FIXML* files, then the translation is aborted. Moreover, *FIXML* files that are not translated completely can be debugged with the HenshinTGG GUI. The GUI visualises those fragments of a *FIXML* AST that cannot be translated by marking them red. This allows the adaptation of the triple rules to obtain a complete translation.

```
@Check
def checkXMLNodeHasStartEndTagsOfSameName(XMLNode xmlnode) {
    if (xmlnode.end != null && !xmlnode.tag.equals(xmlnode.end)) {
            error("Start-tag must have the name of its corresponding
                   end-tag.", TTC_XMLPackage.Literals::XML_NODE__END);
            return;
    }
}
```

Figure 8: Xtext validator for *FIXML* syntax - constraint `checkXMLNodeHasStartEndTagsOfSameName`

Execution time The execution times of the translation steps for each test are as follows (measured in milliseconds). Note that we adapted Test 3, since, initially it contained a syntax error (the second `Order` tag must be an end-tag). For translating Tests 7 & 8, we also fixed the syntax errors in order to get results for the translation.

	Parsing (in ms)	AST-Conversion (in ms)	Serialisation (in ms)
test1.xml	19	46	146
test2.xml	26	210	312
test3.xml	20	274	218
test4.xml	24	182	279
test5.xml	99	754	320
test6.xml	97	3183	278
test7.xml	35	110	192
test8.xml	9	120	255

Modularity As the specification of the M2M transformation of ASTs is the main part of the solution, we focus on the sequential dependencies between the triple rules of the TGG to obtain a measure for modularity. Note that triple rules are non-deleting and parallel dependencies between rules do not exist. Between the 14 triple rules of the specification, 21 sequential dependencies do exist ($1 - \frac{|dependencies|}{|rules|} = -0.5$).

Abstraction level : The abstraction level of the presented specification is classified as High, since, TGGs together with EBNF grammars provide a primarily declarative approach to specify the T2T transformation.

5 Conclusion

The paper adapted the approach for T2T transformations from [10] to provide a solution to the *FIXML2Code* case study of the transformation tool contest 2014 by using the EMF tools Xtext and HenshinTGG. Xtext is used to parse *FIXML* content to an AST and to serialise Java ASTs to Java source code. HenshinTGG is used to perform the main task of translating *FIXML* ASTs into Java ASTs based on the formal concept of TGGs. This allowed the use of existing formal results in order to ensure syntactical correctness of the translation. The approach was evaluated based on a given set of fixed criteria which enables a comparison with other solutions to the case study in future work.

Extensions The following extensions to the solution were proposed by the case study [12]. The presented approach is flexible enough to cover these extensions. (1) **Selection of appropriate data types:** In order to map XML attributes to member variables of correct types and not only to member variables of type `String`, parser rule `Attribute` of the *FIXML* grammar in Fig. 2 must be modified with $(\texttt{valueS} = \texttt{STRING} | \texttt{valueI} = \texttt{INT} | \texttt{valueD} = \texttt{DOUBLE} | \ldots)$ for $\texttt{value} = \texttt{STRING}$. This enables a distinction between data types in *FIXML* ASTs. Then, for each specified data type separate rules *FT_Attribute2Attribute* and *FT_Attribute2ExistingAttribute* must be defined such that `Attribute` nodes with attributes `valueS`, `Attribute` nodes with attributes `valueI`, and `Attribute` nodes with attributes `valueD` are translated separately to `declaration` nodes with correct `types`. (2) **Generic transformation:** The solution generates Java classes from *FIXML* sample files where the classes shall reflect the general structure of *FIXML* files. A generation of classes based on *FIXML* schema definitions is more appropriate in order to obtain a source code representation of the general structure. The presented approach can be adopted, since, Eclipse supports the automatic generation of EMF meta-models from XML schemas which then serve as meta-models for input ASTs, i.e., no Xtext grammar for parsing is

required. The Xtext grammar for serialisation does not need to be modified but the triple rules need to be adapted accordingly to the new input EMF meta-model.

References

[1] (2014): *AGG – Version 2.0.6*. Available at `http://user.cs.tu-berlin.de/~gragra/agg/`.

[2] The Eclipse Foundation (2012): *Xtext – Language Development Framework – Version 2.2.1*. Available at `http://www.eclipse.org/Xtext/`.

[3] The Eclipse Foundation (2013): *EMF Henshin – Version 0.9.4*. Available at `http://www.eclipse.org/modeling/emft/henshin/`.

[4] H. Ehrig, K. Ehrig, U. Prange & G. Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science, Springer.

[5] Hartmut Ehrig & Ulrike Prange (2008): *Formal Analysis of Model Transformations Based on Triple Graph Rules with Kernels*. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg & Gabriele Taentzer, editors: *Graph Transformations*, *Lecture Notes in Computer Science* 5214, Springer Berlin Heidelberg, pp. 178–193.

[6] Holger Giese, Stephan Hildebrandt & Stefan Neumann (2010): *Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent*. In Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr & Bernhard Westfechtel, editors: *Graph Transformations and Model-Driven Engineering*, *Lecture Notes in Computer Science* 5765, Springer Berlin Heidelberg, pp. 555–579.

[7] Ulrike Golas, Hartmut Ehrig & Frank Hermann (2011): *Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions*. *ECEASST*.

[8] Joel Greenyer & Jan Rieke (2012): *Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata*. In Andy Schürr, Dániel Varró & Gergely Varró, editors: *Applications of Graph Transformations with Industrial Relevance*, *Lecture Notes in Computer Science* 7233, Springer Berlin Heidelberg, pp. 222–237.

[9] Frank Hermann, Hartmut Ehrig, Ulrike Golas & Fernando Orejas (2010): *Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars*. In: *Model Driven Interoperability (MDI 2010)*, ACM, pp. 22–31.

[10] Frank Hermann, Susann Gottmann, Nico Nachtigall, Hartmut Ehrig, Benjamin Braatz & Thomas Engel (2014): *Triple Graph Grammars in the Large for Translating Satellite Procedures*. In: *Theory and Practice of Model Transformations*, LNCS, Springer.

[11] Stephan Hildebrandt, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin & Andy Schürr (2013): *A Survey of Triple Graph Grammar Tools*. *ECEASST* 57.

[12] Kevin Lano, Sobhan Yassipour-Tehrani & Krikor Maroukian (2014): *Case study: FIXML to Java, C# and C++*. In: *7th Transformation Tool Contest (TTC 2014)*, this volume, WS-CEUR.

[13] Andy Schürr & Felix Klar (2008): *15 Years of Triple Graph Grammars*. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg & Gabriele Taentzer, editors: *Graph Transformations*, *Lecture Notes in Computer Science* 5214, Springer Berlin / Heidelberg, pp. 411–425. 10.1007/978-3-540-87405-$8_2$8.

A Some generated outputs

A.1 Generated Java AST (EMF model instance) for test5.xml.txt

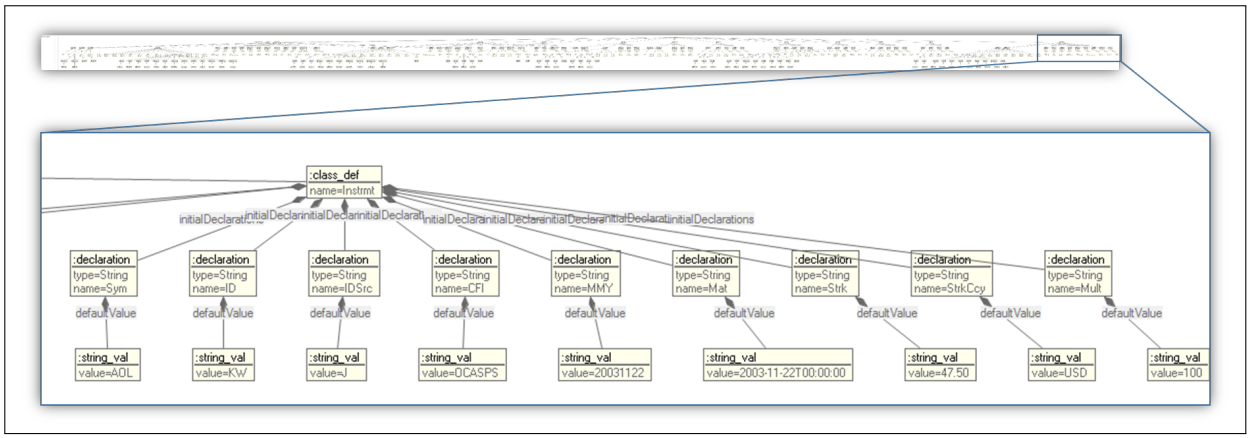

Figure 9: Generated output Java AST (EMF model instance) for test5.xml.txt

A.2 Generated Java source code for test5.xml.txt

```
1  import java.util.Vector;
2
3  class FIXML {
4          FIXML() {
5          }
6  }
7
8  class PosRpt {
9          String RptID = "541386431";
10         String Rslt = "0";
11         String BizDt = "2003-09-10T00:00:00";
12         String Acct = "1";
13         String AcctTyp = "1";
14         String SetPx = "0.00";
15         String SetPxTyp = "1";
16         String PriSetPx = "0.00";
17         String ReqTyp = "0";
18         String Ccy = "USD";
19         Vector<Pty> Pty_objects = new Vector<Pty>();
20         Vector<Qty> Qty_objects = new Vector<Qty>();
21         Hdr Hdr_object = new Hdr();
22         Amt Amt_object = new Amt();
23         Instrmt Instrmt_object = new Instrmt();
24
25         PosRpt(Vector<Pty> Pty_list, Vector<Qty> Qty_list) {
26                 this.Pty_objects = Pty_list;
27                 this.Qty_objects = Qty_list;
28         }
29
30         PosRpt(String RptID, String Rslt, String BizDt, String Acct,
31                         String AcctTyp, String SetPx, String SetPxTyp, String PriSetPx,
32                         String ReqTyp, String Ccy, Hdr Hdr_, Amt Amt_, Instrmt Instrmt_) {
33                 this.RptID = RptID;
34                 this.Rslt = Rslt;
```

```
35                  this.BizDt = BizDt;
36                  this.Acct = Acct;
37                  this.AcctTyp = AcctTyp;
38                  this.SetPx = SetPx;
39                  this.SetPxTyp = SetPxTyp;
40                  this.PriSetPx = PriSetPx;
41                  this.ReqTyp = ReqTyp;
42                  this.Ccy = Ccy;
43                  this.Hdr_object = Hdr_;
44                  this.Amt_object = Amt_;
45                  this.Instrmt_object = Instrmt_;
46          }
47
48          FIXML() {
49          }
50  }
51
52  class Hdr {
53          String Snt = "2001-12-17T09:30:47-05:00";
54          String PosDup = "N";
55          String PosRsnd = "N";
56          String SeqNum = "1002";
57          Sndr Sndr_object = new Sndr();
58          Tgt Tgt_object = new Tgt();
59          OnBhlfOf OnBhlfOf_object = new OnBhlfOf();
60          DlvrTo DlvrTo_object = new DlvrTo();
61
62          Hdr() {
63          }
64
65          Hdr(String Snt, String PosDup, String PosRsnd, String SeqNum, Sndr Sndr_,
66                          Tgt Tgt_, OnBhlfOf OnBhlfOf_, DlvrTo DlvrTo_) {
67                  this.Snt = Snt;
68                  this.PosDup = PosDup;
69                  this.PosRsnd = PosRsnd;
70                  this.SeqNum = SeqNum;
71                  this.Sndr_object = Sndr_;
72                  this.Tgt_object = Tgt_;
73                  this.OnBhlfOf_object = OnBhlfOf_;
74                  this.DlvrTo_object = DlvrTo_;
75          }
76  }
77
78  class Sndr {
79          String ID = "String";
80          String Sub = "String";
81          String Loc = "String";
82
83          Sndr() {
84          }
85
86          Sndr(String ID, String Sub, String Loc) {
87                  this.ID = ID;
88                  this.Sub = Sub;
89                  this.Loc = Loc;
90          }
91  }
92
93  class Tgt {
94          String ID = "String";
95          String Sub = "String";
96          String Loc = "String";
97
98          Tgt() {
99          }
```

```
100
101         Tgt(String ID, String Sub, String Loc) {
102                 this.ID = ID;
103                 this.Sub = Sub;
104                 this.Loc = Loc;
105         }
106 }
107
108 class OnBhlfOf {
109         String ID = "String";
110         String Sub = "String";
111         String Loc = "String";
112
113         OnBhlfOf() {
114         }
115
116         OnBhlfOf(String ID, String Sub, String Loc) {
117                 this.ID = ID;
118                 this.Sub = Sub;
119                 this.Loc = Loc;
120         }
121 }
122
123 class DlvrTo {
124         String ID = "String";
125         String Sub = "String";
126         String Loc = "String";
127
128         DlvrTo() {
129         }
130
131         DlvrTo(String ID, String Sub, String Loc) {
132                 this.ID = ID;
133                 this.Sub = Sub;
134                 this.Loc = Loc;
135         }
136 }
137
138 class Pty {
139         String ID = "OCC";
140         String R = "21";
141         Sub Sub_object = new Sub();
142
143         Pty() {
144         }
145
146         Pty(String ID, String R, Sub Sub_) {
147                 this.ID = ID;
148                 this.R = R;
149                 this.Sub_object = Sub_;
150         }
151 }
152
153 class Sub {
154         String ID = "ZZZ";
155         String Typ = "2";
156
157         Sub() {
158         }
159
160         Sub(String ID, String Typ) {
161                 this.ID = ID;
162                 this.Typ = Typ;
163         }
164 }
```

```
165
166 class Qty {
167         String Typ = "SOD";
168         String Long = "35";
169         String Short = "0";
170
171         Qty() {
172         }
173
174         Qty(String Typ, String Long, String Short) {
175                 this.Typ = Typ;
176                 this.Long = Long;
177                 this.Short = Short;
178         }
179 }
180
181 class Amt {
182         String Typ = "FMTM";
183         String Amt = "0.00";
184
185         Amt() {
186         }
187
188         Amt(String Typ, String Amt) {
189                 this.Typ = Typ;
190                 this.Amt = Amt;
191         }
192 }
193
194 class Instrmt {
195         String Sym = "AOL";
196         String ID = "KW";
197         String IDSrc = "J";
198         String CFI = "OCASPS";
199         String MMY = "20031122";
200         String Mat = "2003-11-22T00:00:00";
201         String Strk = "47.50";
202         String StrkCcy = "USD";
203         String Mult = "100";
204
205         Instrmt() {
206         }
207
208         Instrmt(String Sym, String ID, String IDSrc, String CFI, String MMY,
209                         String Mat, String Strk, String StrkCcy, String Mult) {
210                 this.Sym = Sym;
211                 this.ID = ID;
212                 this.IDSrc = IDSrc;
213                 this.CFI = CFI;
214                 this.MMY = MMY;
215                 this.Mat = Mat;
216                 this.Strk = Strk;
217                 this.StrkCcy = StrkCcy;
218                 this.Mult = Mult;
219         }
220 }
```