Solving the FIXML2Code-case study with HenshinTGG

Frank Hermann Nico Nachtigall Benjamin Braatz Susann Gottmann
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 (an 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. 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 [5, 9]. Several tool implementations for TGGs exist [7]. Numerous case studies have proven the applicability of TGGs in model-to-model (M2M) transformation scenarios [4, 3]. In [6], we presented an approach for applying TGGs in a text-to-text (T2T) transformation context for translating satellite procedures. We adapt this approach to provide a solution for the T2T transformation case study of the TTC 2014 [8]. We evaluate the solution based on fixed criteria: complexity, accuracy, development effort, fault tolerance, execution time, and modularity.

As depicted in Fig. 1, our transformation involves these steps: A text-to-model (T2M) transformation step parses the the content of a *FIXML* file and yields its abstract syntax tree (AST). Then, a M2M transformation is performed based on a given TGG to convert the source AST into the target AST. Finally, the target AST is serialised back to source code via a model-to-text (M2T) transformation. We combine *Xtext* [1] 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 generating parsers and serialisers for them. The parser checks that the input source code is well-formed and the serialiser ensures that the generated output source code is well-defined. *HenshinTGG* is an extension of the EMF-Henshin tool [2] and is used for specifying and executing M2M transformations based on the formal concept of TGGs. The solution is available on SHARE[1].

The paper is structured as follows. Sec. 2 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.

[*]Supported by the Fonds National de la Recherche, Luxembourg (3968135, 4895603).

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

Submitted to:
TTC 2014

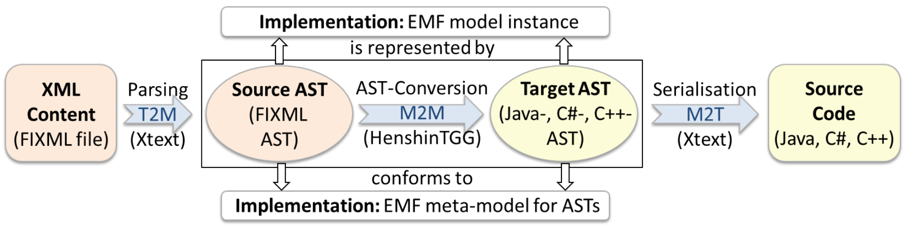

Figure 1: Main phases for the T2T-translation (Text-To-Text)

2 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++ source code. ASTs are specified by instance graphs that are typed over a meta-model which defines the allowed structure of the instance graphs. Fig. 2 (right) depicts the meta-model of *FIXML* ASTs. Each *FIXML* AST has a root node of type `Model` with at most one `Header` node connected by a `header` edge and with a number of `XMLNodes` 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>`). `XMLNodes` may have several child elements, i.e., plain text content (attribute `entry`) or a number of XML `subnodes`. 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 (cf. App. A).

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 [2]. *HenshinTGG* is an Eclipse plugin providing a graphical development and simulation environment for TGGs allowing the specification of triple graphs and triple rules and execution of different kinds of TGG operations.

3 Solution

As already introduced in Sec. 1, we applied the general concept for the T2T-translation depicted in Fig. 1. The presented solution concerns the output language Java only. We are confident that the Xtext grammar could be generalised to a grammar that is capable to handle also C# and C++.

3.1 Parser for *FIXML* ASTs and Serialiser for Java 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. E.g., the root rule `Model` specifies that each *FIXML* file has one optional XML `Header` (line 4) and contains a number of `XMLNodes`. A `Header` contains the content of a usual XML header and a number of `Attributes` (lines 5-7). For a detailed description of this Xtext grammar see Sec. B.1.

From the grammar, *Xtext* automatically generates the EMF meta-model in Fig. 2 (right) which serves as meta-model for *FIXML* ASTs. Each parser rule becomes a node except for `Entry`, because `Entry` has only unnamed references to terminal rules. Each named reference between two parser rules becomes an

```
grammar lu.uni.snt.ttc2014.FIXML
        with org.eclipse.xtext.common.Terminals
generate fIXML "http://www.uni.lu/snt/ttc2014/FIXML"
Model     : header=Header? nodes+=XMLNode*;
Header    : {Header} "<" "?" "xml"
            attributes+=Attribute* "?" ">"
            ("<" "!" ID* STRING* ">")?;
XMLNode   : "<" tag=ID attributes+=Attribute*
            ( ("/" ">") |
              (">" subnodes+=XMLNode*
                   entry=Entry? "<" "/" end=ID  ">")
            );
Attribute : name=ID "=" value=STRING;
Entry     : (ID|INT|WS|ANY_OTHER|
             ML_COMMENT|SL_COMMENT|STRING)+;
```

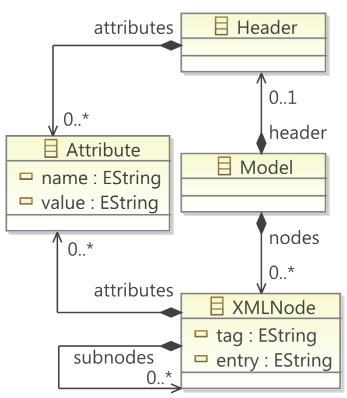

Figure 2: Xtext grammar for *FIXML* parser (left) and corresponding *FIXML* meta-model (right)

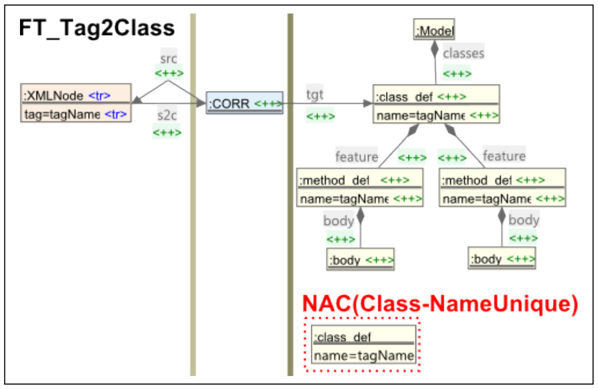

Figure 3: Generated FT-rule *FT_Tag2Class*

	Parsing (in ms)	AST-Conversion (in ms)	Serialising (in ms)
test1.xml	147	500	1204
test2.xml	199	1063	1782
test4.xml	174	1478	3307
test5.xml	1012	5489	1749
test6.xml	2082	11935	596

Figure 4: Execution times on SHARE

edge between the corresponding two nodes. Each named reference between a parser and a terminal rule becomes an attribute of the corresponding node.

Analogously to the parser, 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 (cf. App. B). Note that we only consider that subset of Java which is relevant for the translation.

3.2 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 one forward translation rule *FT_Tag2Class* for converting parts of *FIXML* ASTs to parts of Java ASTs in this section. Forward translation rules are derived automatically from triple rules that we specified with HenshinTGG. The forward translation rules are applied with HenshinTGG in order to convert the ASTs. The rules include the following 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 occurring features of these tags are integrated within the class definition. We 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.

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.

4 Analysis

The approach is evaluated concerning the following criteria that are fixed for the case study [8].

Complexity	The TGG comprises 14 triple rules altogether 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. Both Xtext grammars comprise 24 parser rules with 58 references between them. In total, this results in a complexity score of 322.
Accuracy	The syntactical correctness of the translation is ensured by its formal definition based on forward translation rules [5], 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 are expressed by graph constraints and are translated to application conditions of the triple rules. So, the translation of *FIXML* ASTs ensures that target ASTs fulfill the constraints.
Development effort	We spent 8 person-hours for writing and debugging the solution. In detail: 1 hour for the grammar of the parser, 2 hours for the grammar of the serialiser, and 5 hours for the TGG. The experience level of our developers is: Expert.
Fault tolerance	Files Test 7 & 8 of the *FIXML* case study [8] have syntax errors and should be identified as invalid by the translation. The fault tolerance of our solution is classified as High. Invalid *FIXML* input files that do not correspond to the *FIXML* Xtext grammar lead to Xtext parsing errors which are displayed on the console and the translation is aborted. Test 8 is successfully detected as being invalid because the *FIXML* grammar claims that each XML start-tag has a corresponding end-tag. Syntactical restrictions that cannot be expressed by the grammar are defined by constraints in a custom Xtext validator. We defined a constraint claiming that each start-tag has the name of its corresponding end-tag. Test 7 does not satisfy this constraint and is classified as being invalid. HenshinTGG GUI visualises those fragments of a *FIXML* AST that cannot be translated by marking them red. This allows debugging and the adaptation of the triple rules to obtain a complete translation.
Execution time	The execution times of the translation steps for each test are as listed in Fig. 4.
Modularity	The TGG has a score of -0.5 (21 dependencies between 14 triple rules). The Xtext parsing grammar for XML has a score of -0.71 (12 references between 7 grammar rules). The Xtext serialisation grammar for Java has a score of -0.64 (36 references between 22 grammar rules). In total, this results in a score of -0.62.
Abstraction level	The abstraction level of the presented specification is classified as High, since, TGGs together with EBNF grammars are a declarative approach to specify the T2T transformation.

5 Conclusion

The paper provides a T2T transformation solution to the *FIXML2Code* case study of the TTC 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.

The following extensions to the solution were proposed by the case study [8]. The presented approach is flexible enough to cover these extensions.

(1) **Selection of appropriate data types:** In order to enable a distinction between data types in *FIXML* ASTs, parser rule `Attribute` of the *FIXML* grammar must be modified with $(\texttt{valueS} = \texttt{STRING}|\ \texttt{valueI} = \texttt{INT}|\ldots)$ for $\texttt{value} = \texttt{STRING}$. For each possible XML attribute type, two separate transformation rules must be defined such that XML attributes are transformed to member variables of correct types in the source code.

(2) **Generic transformation:** The solution generates Java classes from *FIXML* sample files that 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 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] The Eclipse Foundation (2012): *Xtext – Language Development Framework – Version 2.2.1*. Available at `http://www.eclipse.org/Xtext/`.

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

[3] H. Giese, S. Hildebrandt & S. Neumann (2010): *Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent*. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr & B. Westfechtel, editors: *Graph Transformations and Model-Driven Engineering*, *LNCS* 5765, Springer, pp. 555–579.

[4] J. Greenyer & J. Rieke (2012): *Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata*. In A. Schürr, D. Varró & G. Varró, editors: *Applications of Graph Transformations with Industrial Relevance*, *LNCS* 7233, Springer, pp. 222–237.

[5] F. Hermann, H. Ehrig, U. Golas & F. Orejas (2010): *Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars*. In: *MDI 2010*, ACM, pp. 22–31.

[6] F. Hermann, S. Gottmann, N. Nachtigall, H. Ehrig, B. Braatz & T. Engel (2014): *Triple Graph Grammars in the Large for Translating Satellite Procedures*. In: *Theory and Practice of Model Transformations*, *LNCS* 7909, Springer, pp. 50–51.

[7] S. Hildebrandt, L. Lambers, H. Giese, J. Rieke, J. Greenyer, W. Schäfer, M. Lauder, A. Anjorin & A. Schürr (2013): *A Survey of Triple Graph Grammar Tools*. *ECEASST* 57.

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

[9] A. Schürr & F. Klar (2008): *15 Years of Triple Graph Grammars*. In H. Ehrig, R. Heckel, G. Rozenberg & G. Taentzer, editors: *Graph Transformations*, *LNCS* 5214, Springer, pp. 411–425.

A Triple Graph Grammars

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.

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 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 [11] and we denote the step by $G \xRightarrow{tr,m} H$. Moreover, triple rules can be extended by negative application conditions (NACs) for restricting their application to specific matches [5, 12]. Thus, NACs can ensure that the rules are only applied in the right contexts. A triple graph grammar $TGG = (TG, SG, TR)$ consists of a type triple graph TG, a start triple graph SG 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. 7. The figure shows an adapted screenshot of the HenshinTGG tool [2] 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 [5]. 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 $\langle\texttt{tr}\rangle = \mathit{false}$ to $\langle\texttt{tr}\rangle = \mathit{true}$. All preserved source elements in *tr* are preserved by tr_{FT} and their translation attributes stay unchanged with $\langle\texttt{tr}\rangle = \mathit{true}$.

Example 2 (Operational Translation Rules) *Fig. 8 depicts the corresponding forward translation rule of the triple rule in Fig. 7. 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 $\langle\texttt{tr}\rangle = \mathit{false}$ leading to G'^S and applying the forward translation rules as long as possible. Formally, a *forward translation sequence* $(G^S, G_0 \xRightarrow{tr^*_{FT}} G_n, G^T)$ is given by an input source model G^S, a transformation sequence $G_0 \xRightarrow{tr^*_{FT}} G_n$ 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 $MT\colon \mathscr{L}(TG^S) \Rrightarrow \mathscr{L}(TG^T)$ consists of all forward translation sequences with TG^S and TG^T being the restriction of the triple type graph TG to the source or target component, 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 [5] how to use the automated conflict analysis engine of AGG [14] for checking functional behaviour of model transformations.

B Deeper Insights into our 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 [6] 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*. The Xtext framework supports the syntax specification of textual domain specific languages (DSLs) and generates an optional formatting configuration, based on the EBNF (Extended Backus-Naur Form) grammar specification of a DSL. I addition, 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. *HenshinTGG* is an Eclipse plugin supporting the visual specification and execution of EMF transformation systems, which is used for the main phase (AST conversion). The presented solution concerns the output language Java only, but the presented solution seems to be flexible enough to enable a smooth extension of the serialiser to output languages C# and C++.

B.1 Parser for *FIXML*

In Sec. 3.1, we broached the description of the Xtext EBNF grammar of input DSL *FIXML*. In this section, we will complete this explanation.

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` 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 underscore symbol. Note that start-tags and their end-tags may have different tag names, since, `tag` and `end` allow arbitrary `ID`s. Therefore, we introduce an additional Xtext constraint that claims that each start-tag has the name of its corresponding end-tag (`xmlnode.tag.equals(xmlnode.end)`) Fig. 5.

```
@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 5: Xtext validator for *FIXML* syntax - constraint `checkXMLNodeHasStartEndTagsOfSameName`

`XMLNodes` may have several child nodes (reference `subnodes` in line 10) `subnodes` as well as optional plain text content of type `Entry` (lines 8-12). 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. An `Attribute` has a `name` of type `ID` and a value of type `STRING`.

B.2 Serialiser for Java ASTs

```
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 6: Xtext grammar for Java serialiser

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. 6. 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

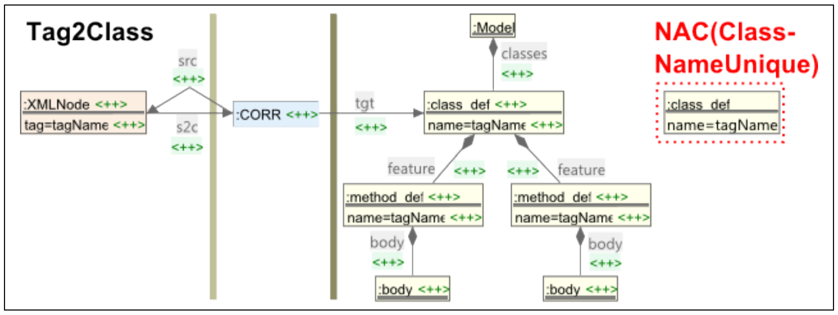

Figure 7: Triple rule *Tag2Class*

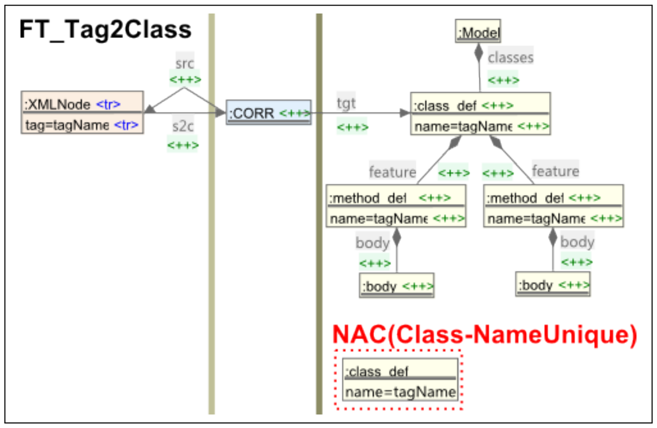

Figure 8: Triple rule *Tag2Class*

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 `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 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 output languages C# and C++, e.g., in Fig. 6 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++.

B.3 M2M Transformation

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. 7 depicts a screenshot of triple rule *Tag2Class* as specified in the HenshinTGG tool and Fig. 8 shows the corresponding forward translation rule that is derived automatically from the triple rule *Tag2Class* that we specified with HenshinTGG. For all other derived

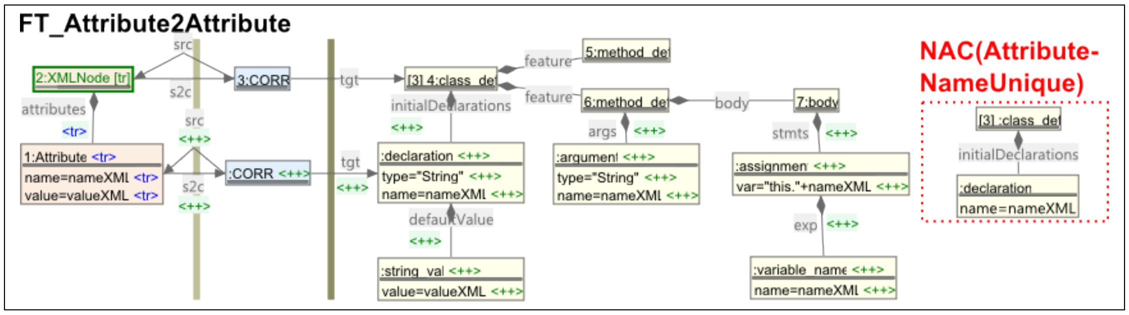

Figure 9: FT-rule *FT_Attribute2Attribute*

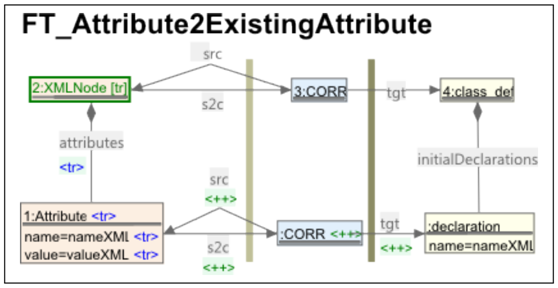

Figure 10: FT-Rule *FT_Attribute2ExistingAttribute*

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).

Forward translation rule *FT_Tag2Class* is already presented in Sec. 3.2.

Rule *FT_Attribute2Attribute* (Fig. 9) 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 variables `name` and `value` get the same `defaultValues` (i.e., `nameXML` and `valueXML`) as 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 be 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 yet exist a member variable of the same `nameXML` for the class (NAC *AttributeNameUnique*).

Due to the NAC, only one of these attributes is translated by rule *FT_Attribute2Attribute*, if the *FIXML* file contains several XML tags of the same name that share XML attributes of the same name. 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 of class declarations and implementations into header and implementation files is also realisable without large efforts. The forward translation rules would maintain a `Model` − `Header` node for the header file and a `Model` − `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.

C Some generated outputs

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

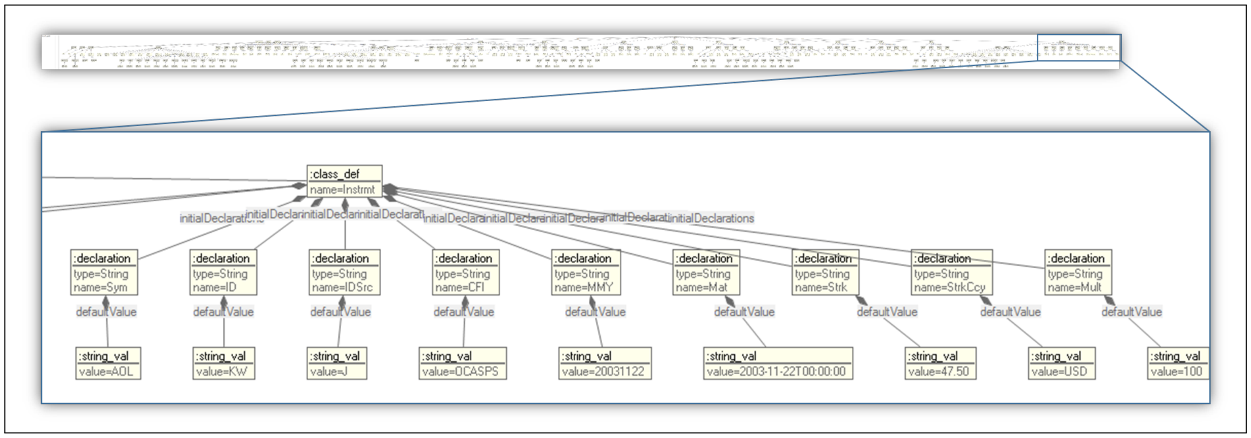
:class_def
name=Instrmt
initialDeclarations
:declaration type=String name=Sym
:declaration type=String name=ID
:declaration type=String name=IDSrc
:declaration type=String name=CFI
:declaration type=String name=MMY
:declaration type=String name=Mat
:declaration type=String name=Strk
:declaration type=String name=StrkCcy
:declaration type=String name=Mult
defaultValue
:string_val value=AOL
:string_val value=KW
:string_val value=J
:string_val value=OCASPS
:string_val value=20031122
:string_val value=2003-11-22T00:00:00
:string_val value=47.50
:string_val value=USD
:string_val value=100

Figure 11: Generated output Java AST (EMF model instance) for test5.xml.txt

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

```java
import java.util.Vector;

class FIXML {
    PosRpt PosRpt_object = new PosRpt();

    FIXML() {
    }

    FIXML(PosRpt PosRpt_) {
        this.PosRpt_object = PosRpt_;
    }
}

```

```java
class PosRpt {
    String RptID = "541386431";
    String Rslt = "0";
    String BizDt = "2003-09-10T00:00:00";
    String Acct = "1";
    String AcctTyp = "1";
    String SetPx = "0.00";
    String SetPxTyp = "1";
    String PriSetPx = "0.00";
    String ReqTyp = "0";
    String Ccy = "USD";
    Vector<Pty> Pty_objects = new Vector<Pty>();
    Vector<Qty> Qty_objects = new Vector<Qty>();
    Hdr Hdr_object = new Hdr();
    Amt Amt_object = new Amt();
    Instrmt Instrmt_object = new Instrmt();

    PosRpt() {
    }

    PosRpt(String RptID, String Rslt, String BizDt, String Acct,
            String AcctTyp, String SetPx, String SetPxTyp, String PriSetPx,
            String ReqTyp, String Ccy, Vector<Pty> Pty_list,
            Vector<Qty> Qty_list, Hdr Hdr_, Amt Amt_, Instrmt Instrmt_) {
        this.RptID = RptID;
        this.Rslt = Rslt;
        this.BizDt = BizDt;
        this.Acct = Acct;
        this.AcctTyp = AcctTyp;
        this.SetPx = SetPx;
        this.SetPxTyp = SetPxTyp;
        this.PriSetPx = PriSetPx;
        this.ReqTyp = ReqTyp;
        this.Ccy = Ccy;
        this.Pty_objects = Pty_list;
        this.Qty_objects = Qty_list;
        this.Hdr_object = Hdr_;
        this.Amt_object = Amt_;
        this.Instrmt_object = Instrmt_;
    }
}

class Hdr {
    String Snt = "2001-12-17T09:30:47-05:00";
    String PosDup = "N";
    String PosRsnd = "N";
    String SeqNum = "1002";
    Sndr Sndr_object = new Sndr();
    Tgt Tgt_object = new Tgt();
    OnBhlfOf OnBhlfOf_object = new OnBhlfOf();
    DlvrTo DlvrTo_object = new DlvrTo();

    Hdr() {
    }

    Hdr(String Snt, String PosDup, String PosRsnd, String SeqNum, Sndr Sndr_,
            Tgt Tgt_, OnBhlfOf OnBhlfOf_, DlvrTo DlvrTo_) {
        this.Snt = Snt;
        this.PosDup = PosDup;
        this.PosRsnd = PosRsnd;
        this.SeqNum = SeqNum;
        this.Sndr_object = Sndr_;
        this.Tgt_object = Tgt_;
        this.OnBhlfOf_object = OnBhlfOf_;
        this.DlvrTo_object = DlvrTo_;
    }
}

class Pty {
    String ID = "OCC";
    String R = "21";
    Sub Sub_object = new Sub();

    Pty() {
    }

    Pty(String ID, String R, Sub Sub_) {
        this.ID = ID;
        this.R = R;
        this.Sub_object = Sub_;
    }
}

class Qty {
    String Typ = "SOD";
    String Long = "35";
    String Short = "0";

    Qty() {
    }

    Qty(String Typ, String Long, String Short) {
        this.Typ = Typ;
        this.Long = Long;
        this.Short = Short;
    }
}

class Amt {
    String Typ = "FMTM";
    String Amt = "0.00";

    Amt() {
    }

    Amt(String Typ, String Amt) {
        this.Typ = Typ;
        this.Amt = Amt;
    }
}

class Instrmt {
    String Sym = "AOL";
    String ID = "KW";
    String IDSrc = "J";
    String CFI = "OCASPS";
    String MMY = "20031122";
    String Mat = "2003-11-22T00:00:00";
    String Strk = "47.50";
    String StrkCcy = "USD";
    String Mult = "100";

    Instrmt() {
    }

    Instrmt(String Sym, String ID, String IDSrc, String CFI, String MMY,
            String Mat, String Strk, String StrkCcy, String Mult) {
        this.Sym = Sym;
        this.ID = ID;
        this.IDSrc = IDSrc;
        this.CFI = CFI;
        this.MMY = MMY;
        this.Mat = Mat;
        this.Strk = Strk;
        this.StrkCcy = StrkCcy;
        this.Mult = Mult;
    }
}

class Sndr {
    String ID = "String";
    String Sub = "String";
    String Loc = "String";

    Sndr() {
    }

    Sndr(String ID, String Sub, String Loc) {
        this.ID = ID;
        this.Sub = Sub;
        this.Loc = Loc;
    }
}

class Tgt {
    String ID = "String";
    String Sub = "String";
    String Loc = "String";

    Tgt() {
    }

    Tgt(String ID, String Sub, String Loc) {
        this.ID = ID;
        this.Sub = Sub;
        this.Loc = Loc;
    }
}

class OnBhlfOf {
    String ID = "String";
    String Sub = "String";
    String Loc = "String";

    OnBhlfOf() {
    }

    OnBhlfOf(String ID, String Sub, String Loc) {
        this.ID = ID;
        this.Sub = Sub;
        this.Loc = Loc;
    }
}

class DlvrTo {
    String ID = "String";
    String Sub = "String";
    String Loc = "String";

    DlvrTo() {
    }

    DlvrTo(String ID, String Sub, String Loc) {
        this.ID = ID;
        this.Sub = Sub;
        this.Loc = Loc;
    }
}

class Sub {
    String ID = "ZZZ";
    String Typ = "2";

    Sub() {
    }

    Sub(String ID, String Typ) {
        this.ID = ID;
        this.Typ = Typ;
    }
}
```

Appendix References

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

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

[12] U. Golas, H. Ehrig & F. Hermann (2011): *Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions*. *ECEASST*.

[13] F. Hermann, H. Ehrig, U. Golas & F. Orejas (2010): *Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars*. In: *MDI 2010*, ACM, pp. 22–31.

[14] TFS-Group, Technical University of Berlin (2014): *AGG – Version 2.0.6*. Available at `http://user.cs.tu-berlin.de/~gragra/agg/`.