Specification of Triple Graph Grammar Rules using Textual Concrete Syntax

Mirko Seifert
Lehrstuhl Softwaretechnologie
Fakultät Informatik
Technische Universität Dresden
Dresden, Germany
mirko.seifert@tu-dresden.de

Christian Werner
Lehrstuhl Softwaretechnologie
Fakultät Informatik
Technische Universität Dresden
Dresden, Germany
s7436532@inf.tu-dresden.de

ABSTRACT

Triple Graph Grammars provide a powerful mechanism to specify bidirectional model transformations. Complex transformations or synchronisation scenarios can be described using declarative rules. However, the specification of these rules is often hard, because rules are specified at the abstract syntax level of the involved models. Furthermore, rules are mostly visualised and edited graphically. This is very feasible for modelling languages that have a graphical syntax, but for textual modelling languages a gap between the languages and the rules is introduced. In addition, large rules written in graphical syntax can easily become confusing and hard to read.
To tackle these problems, we propose to automatically extend the textual syntax for the involved models and use it to specify rules. We explore the benefits and drawbacks of rules that are based on concrete textual syntax.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General

General Terms

Design

Keywords

triple graph grammars, model transformation, textual syntax

1. INTRODUCTION

Triple Graph Grammars (TGGs) [14] have been first introduced by Andy Schürr in 1994 as an extension to pair grammars. Various research papers have shown their applicability in different domains. For example, TGGs have been employed to perform model transformations [11], tool integration [12], and incremental model synchronisation [4]. TGGs were not only successful from an academic perspective, but have also started to gain more widespread use in the industry. This is mainly reflected by the Query View Transformations (QVT) [13] standard which shares concepts with TGGs (see [5] for a detailed comparison).

Current implementations (e.g., Fujaba[1] or MOFLON[2]) provide facilities to specify TGG rules and execute them. Besides debugging, the specification of these rules is the most complicated part when setting up a new TGG transformation. Given that a rule designer is familiar with the concepts of TGGs and in particular the semantics of the different elements of a rule (e.g., *required* and *create* nodes and links), the specification is still not straight forward.

From our perspective, one of the main issues is, that rules are defined at the level of abstract syntax. Instead of using the concrete syntax of the models that are transformed or synchronised, rule designers must refer to the abstract elements of the models (i.e., the meta classes). While this may not sound so complicated at first glance it can turn out to be quite cumbersome. Tiny bits of concrete syntax do often correspond to a large graph if represented in abstract syntax. A lively example are abstract representations of expressions, where simple terms turn into large expression trees.

In addition, rules are usually expressed using graphical syntax. While this has clear advantages in some cases it can imply problems in others. If TGGs are used to transform models that have a textual syntax on their own (e.g., textual Domain-Specific Languages (DSLs)), the graphical rule syntax creates a gap between the subject and the specification of the transformation. Transformation designers are intrinsically familiar with the concrete syntax of their languages. Thus, rule specification gets easier the closer it is to the subject languages.

Being at the heart of TGG-based model transformation and synchronisation, the rule specification is an important issue to pursue the adoption of the TGG formalism. In this paper, we will therefore present how the concrete textual syntax of modelling languages can be used to automatically obtain a TGG rule specification language having a syntax that is close to the original one. The approach will be presented based on a running example (Sect. 2). After explaining the syntax extension (Sect. 3) an explanation of the rule extraction follows (Sect. 4). We compare our work with related publications in Sect. 5 and draw conclusions in Sect. 6.

2. RUNNING EXAMPLE

Within this paper we will use an example from [10] to illustrate our ideas. The example involves a transformation between petri nets and toy train models. The petri nets consist of nodes, arcs and tokens. Nodes can be either transitions or places. Tokens are assigned to a place. The corresponding meta model is shown in Fig. 1.

[1] http://www.fujaba.de
[2] http://www.moflon.org

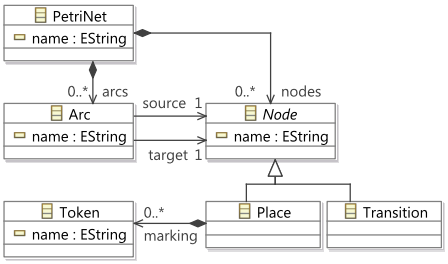

Figure 1: Meta Model for Petri Nets

The toy train models (cf. Fig. 2) are called projects and contain components. Components can be either switches or tracks. Connections can be drawn between ports, where each component can have multiple in and out ports. Trains are assigned to a component denoting that the train is currently located on this track or switch.

Even though the graphical notation of petri nets is used more widely, we will express instances in textual syntax. For the toy train models, being a typical example for DSLs, the textual syntax may be a good choice, because it is easy to define and tool support can be generated automatically. Other transformations that involve textual languages (e.g., programming languages such as Java) are most suitable for the textual rule specification. Nonetheless, we will use the simple transformation between petri nets and toy trains to sketch our general ideas.

Note that textual syntax should not be considered as a contradiction to graphical syntax here. Rather, it is an alternative one. Both types can be used in combination, for example to version control or exchange models in textual form (compare [1]) and edit them in graphical syntax.

Figure 3 shows two example models. To define these syntaxes and to generate tool support we used EMFText [8], but any other textual concrete syntax mapping tool would suffice too. The two example languages are available from the EMFText Syntax Zoo[3].

The transformation between these two models will map the

[3]http://www.emftext.org/zoo

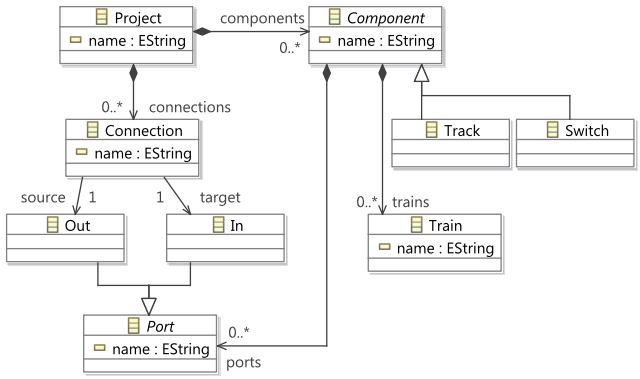

Figure 2: Meta Model for Toy Trains

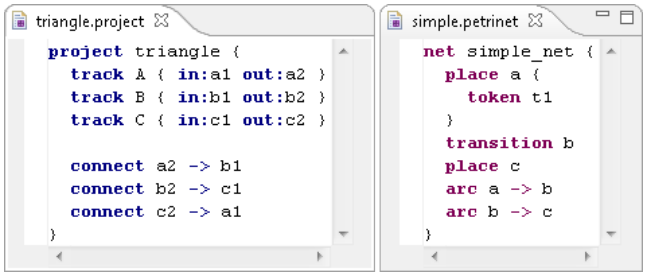

Figure 3: Example Toy Train Project and Petri Net in Concrete Syntax

toy train models to their respective dynamic semantics expressed in terms of a petri net. The rules that specify this transformation have been taken from [10]. Before we show what the textual variants of these rules look like, we shortly sketch how the textual syntaxes of the involved languages can be automatically extended to obtain the rule syntaxes.

3. LANGUAGE EXTENSION

One of the problems mentioned before is that TGG rules are usually specified at the level of abstract syntax. To allow for a more intuitive specification, we want to use the concrete textual syntax of the languages involved in the transformation. More specifically, we want to extend the involved languages with annotation facilities to enable the specification of rules. The general approach used to achieve this is shown in Fig. 4.

Basically the meta models of both languages involved in a transformation are extended with annotation concepts defined in an annotation meta model. Instances of the enriched meta models can contain annotations, which are evaluated by a rule derivation algorithm. Then, a TGG rule is created from each pair of models. As the model instances shall be written in concrete textual syntax, the syntax for both languages must be also extended (not shown in Fig. 4). One may also consider to use annotation mechanisms that are already available in the involved languages. However, this restricts the rule specification to languages, which do have such mechanisms, whereas our approach is applicable for every language.

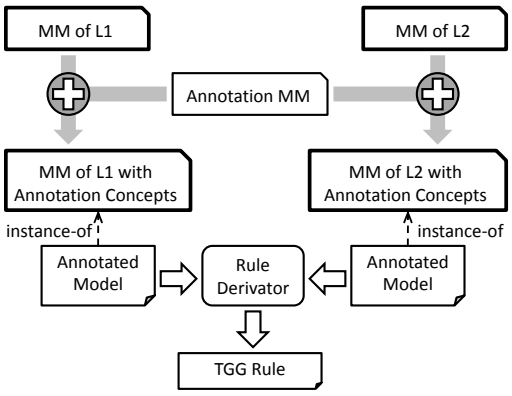

Figure 4: Rule Specification based on Language Extension

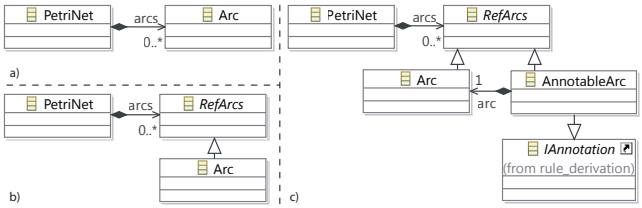

Figure 5: Excepts from the original (a), the extensible (b) and extended Meta Model (c) for Petri Nets

The annotations added to the meta models, are specific to rule specification. For example, annotations tag elements that correspond to each other. Elements must also be typed as being required or created. Furthermore, constraints that control the rule application need to be expressed.

3.1 Meta Model Extension

To extend a meta model it needs to be either designed for extensibility or, if this is not the case, refactorings can be applied. In [9] details about the requirements for extensible meta models and the respective refactorings can be found. The paper refers to Ecore models, but the principles can be applied to other metamodelling languages as well.

Briefly said, the extensibility is established by using dedicated abstract classes to type each reference. By doing so, new types can be attached to the reference by extension of these abstract classes. In addition, all attributes having primitive types are replaced by type wrappers. This wrapping is needed to raise primitive types to the level of complex types. Thus, the same mechanism (references with abstract type) can be used to allow for arbitrary extensions.

As an example, consider the (non-extensible) meta model shown in Fig. 1. Here, the extensibility can be established by replacing the reference `arcs` between `PetriNet` and `Arc` with a reference to a new abstract class `RefArcs`, being a superclass of `Arc`. By doing so, new subclasses of `RefArcs` can be added, which allows arbitrary extensions.

Figure 5 shows an excerpt from the meta model before and after this refactoring (Note: attributes were omitted). In the original meta model (a) the reference `arcs` directly refers to class `Arc`, whereas in the extensible version (b) the reference refers the the abstract class `RefArcs`. In (c) an new subclass `AnnotableArc` is introduced, which contains an ordinary arc and inherits from the annotation concept `IAnnotation`. This class is imported from our TGG annotation meta model.

We will use these annotations to add information to models that allow the derivation of rules. Annotations do have identifiers and a type, which is shown in Fig. 6.

When specifying rules, every meta class of the involved languages can potentially be annotated. Thus, all meta classes need to be extended to support our rule annotations. Therefore, for each existing meta class the extension sketched in Fig. 5 is performed. A new subclass is created that inherits both from the reference type and `IAnnotation`. Having done this for both meta models, we obtain support for annotations. Note that this extension can easily be automated.

Figure 6: Meta Model for Annotations

3.2 Syntax Extension

After extending the meta models to capture rule annotations, the concrete syntax must be extended along the same path. For each new meta class appropriate syntax must be defined. As most other textual syntax tools, EMFText allows to define syntax per meta class. Thus, to obtain a syntax definition for the meta model with annotation support, the original syntax is imported and new rules, one for each new meta class, are added. The new syntax rules basically prefixes the original syntax (e.g., the one defined for class `Arc`) with syntax elements for the attributes of class `IAnnotation` as shown in Listing 1.

```
Arc           ::= "arc" (name[IDENT])?
                  source[IDENT] "->" target[IDENT];
AnnotableArc  ::= (identifier[IDENT])+
                  (type[TYPE])? arc;
```

Listing 1: Syntax Rules for Arc and AnnotableArc

The syntax extension is identical for all new meta classes and can thus be also performed automatically. Equipped with two extended meta models and two matching syntax definition, EMFText can be used to generated tool support (e.g., parsers and editors) for the two new languages. To summarise, we can automatically extend both the meta model and the syntax of the languages involved in a transformation and generate tool support. Thus, one can start specifying rules in concrete textual syntax without any manual effort.

4. RULE DERIVATION

Up to now, we have shown how to automatically derive languages with annotation support. We have also sketched, what these annotations look like, but the more interesting issue is how to derive TGG rules from annotated models. Before giving details of the rule derivation lets shortly recapitulate the elements of TGG rules.

TGG rules have a left-hand and a right-hand side. Both sides are graphs and therefore consist of nodes and links. Nodes can be either correspondence nodes or model element nodes (i.e., they refer to a meta class of one of the involved languages). Links connect nodes within rules. Since TGG rules are usually non-deleting, the two sides are often merged into one graph by tagging the new nodes (i.e., the ones that are present at the right-hand side only) as *create* nodes, which is often denoted by `++`. All other nodes are marked as *required* nodes. In addition TGG rules can contain constraints, assignments or *forbidden* nodes. However, for the scope of this paper, we will not deal with these concepts.

We derive one rule from each pair of annotated models. One could also consider, the rule derivation as abstraction, because some parts of the pair are not included in the rule. To perform rule derivation, we execute the following steps:

- For each annotated model element, create a rule node
- For each set of model elements that are annotated with the same identifier, create a correspondence node and create links connecting the new correspondence node with the respective rule nodes
- Mark all rule nodes as *create* where the corresponding model element is annotated as *create* element
- For each pair of model elements that is connected by exactly one reference create a link between the respective rule nodes
- For each pair of model elements that is connected by multiple references use the references specified in the annotation and create links between the respective rule nodes

To give an example for the rule derivation process, consider Fig. 7 (taken from [10]). The rule specifies how to map tracks to petri net components. Each track is mapped to an arc, its `in` port is mapped to a place, while its `out` port is mapped to a transition in the petri net. The prerequisite for applying this mapping is an existing correspondence between the project and the petri net.

The very same rule can be specified in textual syntax using the pair of models shown in Fig. 8. The correspondence nodes are highlighted in bold font face. Black font indicates required correspondence node, whereas create nodes a shown in green font and followed by `++`.

By looking at Fig. 8 it is easy to see, how the nodes in the textual syntax and the nodes in the graphical representation relate to each other. Basically each box (in the graphical syntax) can be found as an annotated element (in the textual syntax). For example, `@Pr2PN project` is the annotated node that triggers the creation of the `:Project` node in the upper left corner of the TGG rule in Fig. 7. The same applies to `@Pr2PN net`, which causes the creation of the `:PetriNet` node. The annotation `Pr2PN` itself is transformed to the correspondence node `:Pr2PN` and the two links connecting `:Pr2PN` with `:Project` and `:PetriNet`.

This part of the rule derivation is performed by steps 1 and 2. Applying both steps to all other annotated elements (i.e., the ones tagged with `Cp2PN`, `IP2Pl`, and `OP2Tr`) yields all boxes contained in the rule shown in Fig. 7. In addition, links between the correspondence nodes and the domain nodes have been created. However, the nodes are not yet typed

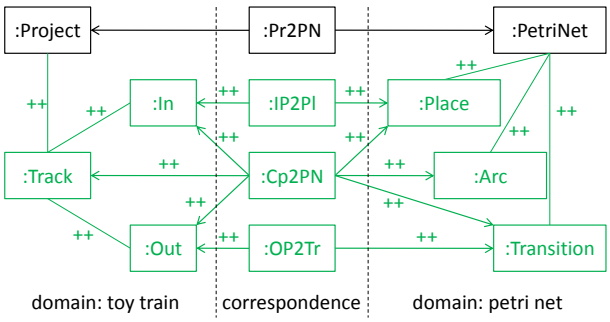

Figure 7: Visual Rule for Mapping Tracks

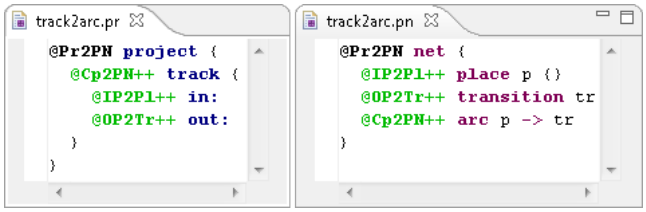

Figure 8: Textual Rule for Mapping Tracks

as required or create. This is done in step 3 of the rule derivation procedure using the type of the annotation.

Now, step 4 takes care of the links that connect elements in a single domain. For example, the link between `:Project` and `:Track` has not yet been established. To obtain these links, the rule derivation must analyse the relations between the elements in each textual model. If model elements are connected by references, links need to be created in the TGG rule. If elements are connected by a single reference only, one rule link is obtained. If elements are connected by multiple references the rule must be refined to specify which references are important for rule derivation. This may be either all references or only selected ones (step 5).

As mentioned in the introduction, we do not handle constraints on attributes yet. Possibly, one could automatically create equality constraints if attribute values (e.g., the names) of elements match. However, this not feasible for all types of attributes. For string-typed attributes many possible values exist, which makes it easy to choose two equals values (e.g., names). For boolean attributes the case is different, because there are only two possible values. Thus, wrong constraints may be derived, because values match even though this is not meant to express equality. Evaluating this is subject to future work.

5. RELATED WORK

Visual rules were already used to specify transformations in the early papers that introduced TGGs [14]. Later, different notations based on object-diagrams were presented that integrated more smoothly with UML modelling [6]. However, the specification was still based on abstract syntax. While applying TGGs to more and more problems, the difficulty of specifying TGG rules (complexity, readability, reuse) has been discovered and studied. In [2], *Triple Patterns* a compact and visual notation has been proposed. The authors share the idea of a more compact representation of TGG rules, but use a visual specification which is still based on abstract syntax. This is in contrast to our work, which focuses on textual concrete syntax.

Other works, that are based on concrete syntax [15, 3] target specification by example. Here, the syntax of the involved languages can be used to specify pairs of models. Based on these pairs, the goal is to derive TGG rules that match the transformation semantics stated by the two models. In other words, rules are derived, which produce one of the models if using the other one as input. This approach is similar to ours regarding the use of concrete syntax, but differs in the way rules are defined. While we explicitly state the correspondence part of the rules using annotations, specification

by example approaches try to derive this part automatically. While this may in principle yield the same result, more example pairs may be needed in the latter case. In addition, transformation designers need to check the derived rules to make sure their expectations are met, while our approach allows to state the expected correspondences beforehand.

6. CONCLUSIONS

In this paper we have shown how to use textual syntax to specify TGG rules. Based on the concrete syntax of the involved languages, we derived extended languages providing annotation support. This extension can be performed fully automatic, which enables the application of our approach to arbitrary textual modelling languages. The gathered annotation support was then employed to add additional information to textual models. Namely, the correspondence between two models was expressed. In contrast to specification by example, which does also use pairs of models to derive rules, we do explicitly state how models correspond instead of deriving this relation.

In the course of this work we came to the conclusion that rule specification based on annotations should be applied to graphical syntaxes as well. By doing so, one could annotate the models in their natural syntax. For example, rules that involve both a graphical language and a textual one (e.g., UML to Java transformations) could be specified using annotations in the graphical syntax (on the UML side) and in the textual syntax (on the Java side). This would allow to use the concrete syntax of the involved languages instead of operating at the abstract syntax level. However, the specification and extension of graphical syntax is not understood as well as its textual counterpart.

The work presented here, yields different open questions. First, the completeness of the rule derivation has to be shown. It is not yet clear, whether all meaningful TGG rules can be specified using the presented annotations. We managed to specify all the basic rules given in [10], but there may be other rules, that can not be specified yet. However, the example rules indicate that the annotation-based specification can be very compact and easy to read if textual languages are subject to TGG transformations. Second, more transformation scenarios need to be conducted to evaluate the practical applicability of our approach. In particular the integration into one of the tools supporting TGGs could give more insights about annotation-based rule derivation.

Acknowledgments

This research has been co-funded by the European Commission within the FP6 project MODELPLEX #034081 and by the German Ministry of Education and Research within the project SuReal. We are also very grateful to Jendrik Johannes who encouraged this work and contributed a lot of ideas in our discussions.

7. REFERENCES

[1] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In *Applications and Theory of Petri Nets 2003: 24th International Conference*, pages 1023–1024, Eindhoven, The Netherlands, June 2003.

[2] J. de Lara, E. Guerra, and P. Bottoni. Triple Patterns: Compact Specification for the Generation of Operational Triple Graph Grammar Rules. In K. Ehrig and H. Giese, editors, *GTVMT'07*, volume 6, 2007.

[3] I. García-Magariño, J. J. Gómez-Sanz, and R. Fuentes-Fernández. Model Transformation By-Example: An Algorithm for Generating Many-to-Many Transformation Rules in Several Model Transformation Languages. In R. F. Paige, editor, *ICMT*, volume 5563 of *LNCS*, pages 52–66, 2009.

[4] H. Giese and R. Wagner. Incremental Model Synchronization with Triple Graph Grammars. In Hartman and Kreische [7], pages 543–557.

[5] J. Greenyer and E. Kindler. Reconciling TGGs with QVT. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *MoDELS*, volume 4735 of *LNCS*, pages 16–30. Springer, 2007.

[6] L. Grunske, L. Geiger, and M. Lawley. A Graphical Specification of Model Transformations with Triple Graph Grammars. In Hartman and Kreische [7].

[7] A. Hartman and D. Kreische, editors. *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005, Proceedings*, volume 3748 of *LNCS*. Springer, 2005.

[8] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and Refinement of Textual Syntax for Models. In *Proceedings of the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA)*, 2009.

[9] F. Heidenreich, J. Johannes, M. Seifert, C. Wende, and M. Böhme. Generating Safe Template Languages. In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering (GPCE'09)*, 2009.

[10] E. Kindler and R. Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical Report tr-ri-07-284, University of Paderborn, D-33098 Paderborn, Germany, June 2007.

[11] A. Königs. Model Transformation with Triple Graph Grammars. In *Model Transformations in Practice Satellite Workshop of MODELS 2005, Montego Bay, Jamaica*, 2005.

[12] A. Königs and A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. *Electronic Notes in Theoretical Computer Science*, 148(1):113–150, 2006.

[13] OMG. MOF Query/View/Transformation Specification Version 1.0. Technical report, Object Management Group (OMG), 2008.

[14] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Proceedings of 20th International Workshop on Graph-Theoretic Concepts in Computer Science, Herrsching, Germany*, volume 903 of *LNCS*. Springer Verlag, 1994.

[15] D. Varró. Model Transformation by Example. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS*, volume 4199 of *LNCS*, pages 410–424. Springer, 2006.