An Adaptable TGG Interpreter for In-Memory Model Transformations *

Ekkart Kindler, Vladimir Rubin, Robert Wagner
Software Engineering Group, Department of Computer Science, University of Paderborn
[kindler|vroubine|wagner]@upb.de

Abstract

Triple graph grammars are a technique for translating and for maintaining consistency among different models. Implementations of such triple graph grammar transformations work only for automatically generated models. Therefore, the transformations cannot be applied to third-party models. In this paper, we discuss this problem and ideas for its solution.

1. Introduction

With the advent of *Model Driven Architecture* (MDA) [5], the generation and transformation of models has become more and more important. There are many different techniques for defining and implementing such transformation; for a good overview and a discussion of the different approaches see [2].

Triple graph grammars (TGGs) [9], an extension of graph rewriting [7], is one of these techniques. TGGs are particularly useful for graph based models such as diagrams. A TGG defines a translation on a relatively high level of abstraction based on the syntactic structure of the underlying models. This way, it is possible to prove the correctness of the defined translation. In addition, TGGs do not only define a translation from one model to another, but also capture the correspondence between the source and the target model. Therefore, they can be used to translate back to the source model after some changes at the target model, and they can be used to check and maintain the consistency between two models.

TGGs have been used in different projects, and there are different implementations of TGGs. PROGRES [8] basically stores the complete model in a database and then applies the TGG rules to the stored model. FUJABA [10] applies Story Charts and pattern matching for this transformation. This results in a simple and quite natural implementation. This implementation, however, requires that there is a meta model on top of which the TGG rules are formulated. Moreover, it is necessary that the models for which the rules are to be applied are generated from this meta model according to the rules of FUJABA. When dealing with models of third parties, where the mapping from the meta model to a Java implementation differs from FUJABA's implementation, this implementation does not work anymore.

In this paper, we present an idea for applying TGG transformations to models that have not been generated from the meta model underlying the TGG rules. Rather, we would like to use any (Java) implementation of the model. We call these models *in-memory models*. In order to apply a TGG translation to such in-memory models, there must be a mapping which defines how the constructs of the meta model, its classes, its attributes, and its associations are implemented. Though it would be a worthwhile task to develop a framework for defining such mappings in the most general way, we propose a simple technique to start with, which can be extended in the future. The idea is quite simple: The mapping is implemented by a class with a particular interface. This interface requires methods, which map arbitrary objects from the implementation to the corresponding class of the meta model. And it requires methods that, for a given object, provides all links corresponding to an association of the meta model. Moreover, this class must provide methods for generating objects and links in the implementation. With this additional *mapper class* – in fact, it is two mapper classes for the source and the target meta model – it is easy to translate models by a TGG interpreter.

The ideas of this paper were inspired by a project and tool called *Component Tools*. For understanding this background, we will briefly discuss this project in Sect. 2. Then, we will rephrase the concept of TGGs in Sect. 3. The core ideas and implementation techniques for an in-memory TGG transformation will be discussed in Sect. 4.

2. Tool Support for System Engineering

In this section we give a brief overview on the concepts for a tool called COMPONENTTOOLS (see [3] for a more detailed description). Parts of this tool have been implemented as a prototype already. The tool will support building a system from components, transforming these models, and for exporting them for analysis purposes as well as for importing analysis results back to the component view.

COMPONENTTOOLS was originally inspired by the case study within the ISILEIT[1] project. The ISILEIT project aims at the development of a seamless methodology for the integrated design, analysis, and validation of distributed production control systems. Its particular emphasis lies on reusing existing techniques, which are used by engineers in industry, and on improving them with respect to formal analysis, simulation, and automatic code generation.

The specification of such systems is done in close cooperation with mechanical and electrical engineers. It turned out

*This work has partly been supported by the German Research Foundation (DFG) grant GA 456/7 ISILEIT as part of the SPP 1064.

[1] ISILEIT is the German acronym for "Integrative Specification of Distributed Production Control Systems for the Flexible Automated Manufacturing".

that system engineers prefer to construct systems from some components in a way that is independent from a particular modelling technique. However, they still would like to use the power of different techniques, once they have constructed their system.

Faced with this requirements, we have started to build a tool solving these problems. In the following, we will use a simplified toy train example representing a material flow system within our case study for explaining the main ideas.

Figure 1 shows such a simple toy train system built from components. There are basically four different components: straight tracks, curved tracks, tracks with a stop signal and switches. The components are equipped with some ports, which are graphically represented as small boxes or circles at the border of the component. The ports are used to connect the components to each other. In our example, there are ports representing the physical connections of tracks, and there are ports which allow to attach controller components, e.g. for the switches or the light signals. Note that, for simplicity, in our example we did not connect the system to controller components and only the physical connections of tracks are presented.

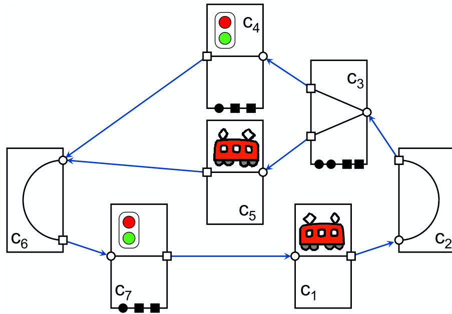

Figure 1: A toy train build from components

In order to build such a system, we need to provide a component library, which contains these four components. The component library defines all the available ports, their graphical appearance, and how ports may be connected. Moreover, the component allows us to provide a model for each component that define its behaviour.

Figure 2 shows the Petri net models for two of the components. In fact, we can provide even more models for each component. For example, there could be abstract models as shown in Figure 2, or there could me more concrete ones. Or there could be additional models in different notation such as State Charts or other notations. From these models, and the system built by the user, ComponentTools generates several overall models of the system, each in one particular notation, which can then be used by appropriate tools supporting this formalism, e.g. for analysis, verification or code generation.

The presented tool is implemented in some parts as a prototype [4], whereas the model generation and transformations will be implemented in a course called “project group” at the University of Paderborn.

In the next section, we give a short introduction to triple graph grammars, which we are using for the specification and execution of model transformations between the component model and the underlying models of each component to the overall models in a particular notation.

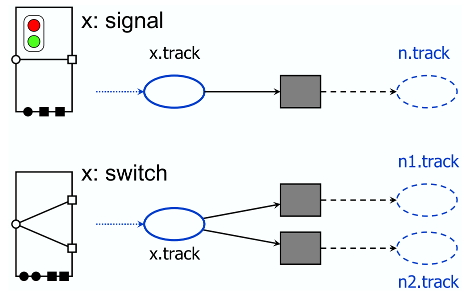

Figure 2: Two models for components

3. Triple Graph Grammars

In his original work, Schürr [9] extended pair grammars [6] to triple graph grammars. In contrast to pair graph grammars, triple graph grammars support context-sensitive productions with rather complex left-hand and right-hand sides. Generally, the separation of correspondence objects enables the modeling of m-to-n relationships between related sides.

The triple graph grammar approach makes a clear distinction between source and target models; it also keeps the extra links needed for specifying the transformations as a separate specification. Using this approach, separate grammars are produced for each of the graphs involved.

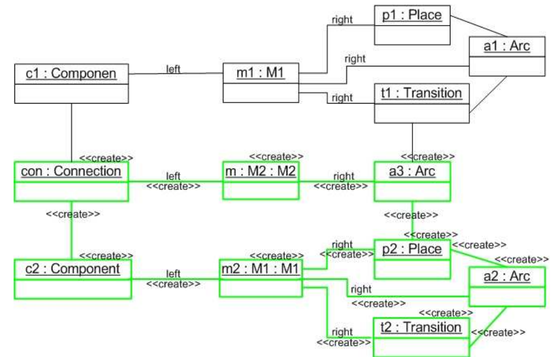

Figure 3: TGG Rule Example

A triple graph grammar specification is a declarative definition of a mapping between two meta models. In Fig. 3 a triple graph grammar rule is shown, defining the correspondence between a component and a Petri net model. It consists of a triple of productions (left production, correspondence production, right production), where each production is regarded as a context-sensitive graph grammar rule. The left production shows the generation of a new component and linking it to the existing one. The right part shows the addition of a new place, a transition and two arcs to the existing Petri Net. The correspondence production shows the relations between the left-hand and right-hand sides.

This declarative specification can be translated into simple graph rewriting rules which are used for the transformation in both directions. In Fig. 4 the forward transformation rule is presented.

The forward transformation rule is applied to the model, if the left production of the triple graph grammar is detected, i.e. if a component was added to the project. In this case, the graph rewriting system will search for all objects contained in the left-hand side of the forward transformation rule. If a match is found, the correspondence object, the objects

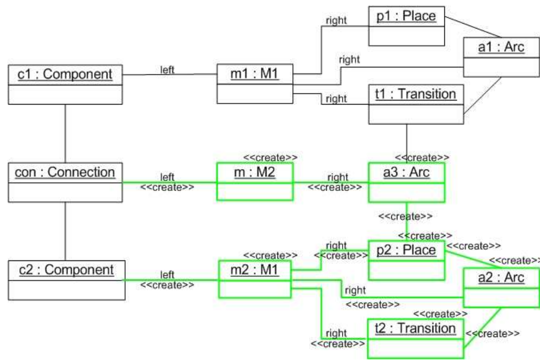

Figure 4: Forward graph rewriting rule

representing places, transitions, arcs, and all links between the objects are created.

In contrast to the forward rule, the rule which handles the translation from the right-hand model to the left-hand model, i.e. from the Petri net model to the component model, is called backward rule. It is created from the triple graph grammar in the very same way as the forward rule: we just exchange the left and right side. For more details, we refer to [9].

The advantage of triple graph grammars over the other approaches lies within the definition of inter-graph relationships, which provide the flexibility to use productions for both forward and backward transformation and correspondence analysis. A triple graph grammar, as a declarative definition of the mapping between the two graphs, can be used for the implementation of a translator in either direction. Such an translator will be presented in the next section.

4. Interpreter

In this section, we present the core ideas and implementation techniques for an in-memory model transformation based on triple graph grammars. We start with a brief description of FUJABA's approach for graph rewriting and discuss its limitations.

The problem. In order to execute triple graph grammar rules in FUJABA, the specified rules are transformed into simple graph rewriting rules. These graph rewriting rules are translated to a Java implementation which performs the desired graph pattern matching and graph rewriting. However, this implementation is based on the meta models of the source, the target, and the correspondence graphs and requires that both meta models are implemented in a predefined way.

In fact, FUJABA requires the implementation to be automatically generated from the meta models by FUJABA. For example, each attribute of a meta model class must be implemented as a private variable with appropriate get and set methods. Associations must be implemented as bidirectional references with well-defined access methods following some naming conventions. These access methods allow navigation between in-memory objects, accessing and modifying in-memory objects, and creating new objects. The mapping between the conceptual model and its implementation is implicitly given by the code generator, which is fundamental for FUJABA's graph rewriting algorithm.

In the COMPONENTTOOLS project, we deal with already existing third-party models. In some cases, the source code of the model implementation is given. In other cases only some kind of an Application Programming Interface (API), which typically differ from FUJABA's model implementation. Hence, the graph pattern matching and graph rewriting algorithms of FUJABA will not work for these models.

Figure 5 shows a part of an example of a *conceptual model* underlying the rules of a TGG and a typical *implementation*, which, for simplicity, is represented also in UML. This example shows that we cannot be even sure that the names

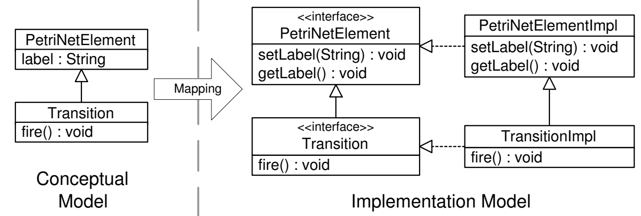

Figure 5: Simple mapping example

of elements in the conceptual model have the same names in the implementation. In order to generate new objects, the TGG interpreter needs to know the names of the classes implementing the interfaces and how associations are implemented.

Architecture. In general, there is no way to map some objects and references of an implementation to the corresponding classes and associations of the conceptual model fully automatically without providing additional information. Therefore, we need a mechanism that defines this mapping such that the TGG interpreter can understand the implementation model. To this end, we propose a simple architecture which is shown in Fig. 6.

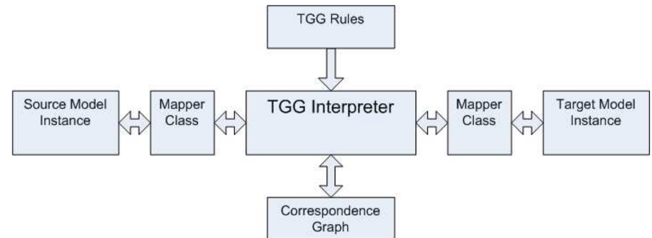

Figure 6: Architecture with Mapper Class

Between the TGG interpreter and the source and target model, there are *mapper classes*, which define the mapping between the conceptual model and the implementation. These classes provide methods that, for a given object of the implementation, return the corresponding class in the conceptual model. Moreover, they provide methods that return all links in the conceptual model for a given object of the implementation. On the other hand, the mapper classes provide methods for generating new objects in the implementation model for a given class of the conceptual model and they provide methods for generating links in the implementation.

For each conceptual model underlying the TGG and each implementation of such a model, a user must implement such a mapper class. These classes are passed to the TGG interpreter as additional parameters. In order to pass these mapper classes to the TGG interpreter, the TGG interpreter defines a *mapper interface*, which must be implemented by

all mapper classes. The interface requires that there are all the methods which have been discussed above: methods for mapping objects of the implementation to the corresponding classes of the conceptual model, methods for getting all links corresponding to some particular association of the conceptual model of an object of the implementation, and methods for generating objects and links in the implementation models that correspond to some class or association of the meta model.

Discussion. Though the mapper class approach is quite simple, it is the most powerful one because, in principle, any mapping can be implemented as a mapper class. The disadvantage of this approach is that it requires programming the mapper classes for each new implementation of a model, which is tedious work. In particular, an inexperienced user might provide a flawed mapper class, which would result in flawed translations even if the TGG interpreter works correctly.

Therefore, it would be nice to define the mappings from the conceptual model to the implementation on a higher level of abstraction and in a notation particularly tailored for this purpose. A good notation for defining such mappings, however, needs more detailed investigation. Once such a notation is available, it is easy to implement a standard mapper class, which receives such a mapping definition as input and which uses the Java Reflection API for implementing the methods required by the mapper interface. With this generic mapper class, it will no longer be necessary for the user to implement a mapper class for each new implementation. Rather, it will be necessary to provide an abstract definition of the mapping in the new notation.

Likewise, the TGG interpreter could be easily used with implementations that are generated automatically from the conceptual models. In this case, the mapper classes could be generated automatically too. Then, it will not be necessary to implement mapper classes for generated models.

Another idea for implementing mapper classes would be to have a standard mapper class which is provided with some scripts for implementing the mapping. Then, it would not be necessary to implement a complete mapper class; rather it is necessary to provide some scripts for defining the mapping only. For example, we could use the scripting language BeanShell [1] for this purpose. On the one hand, this approach would avoid the compilation step for the mapper class, which might be an advantage for a stand-alone tool. On the other hand, using a scripting language will result in some performance loss in comparison to a programming language.

Anyway, all these extended mapping concepts can be built on top of our mapper class concept by implementing a generic mapper class.

Implementation. Currently, we are working on an implementation of the above ideas in the context of COMPONENT-TOOLS. But, the TGG interpreter itself will be completely independent from the graphical user interface, so that it can be easily used in other tools such as FUJABA or as a stand-alone tool.

Even more, our interpreter and mapping concept can also be used for graph rewriting because triple graph grammars are just a specialized sort of graph grammars. The mapping concept immediately carries over to graph rewriting.

5. Conclusion and Future Work

In this paper, we have presented the problem of applying TGG transformations and consistency algorithms to in-memory models that have not been automatically generated. We have presented some ideas for an interpreter for TGGs that solves this problem. This way, TGG techniques can be applied to legacy code and models that have not been generated from our own models.

We just started with a detailed design of the mapper interface and with an implementation of the in-memory TGG interpreter. But, we hope to have a first prototype soon.

6. Acknowledgments

We would like to thank all members of the project group Component Tools at Paderborn University for all their discussions, which help to clearly identify the problem and to come up with the first concepts of the in-memory TGG interpreter.

References

[1] BeanShell. *Leightweight Scripting for Java.* http://www.beanshell.org (last visited July 2003).

[2] K. Charnecki and S. Helsen. Classification of model transformation approaches. In *OOPSLA 2003 Workshop on Generative Techniques in the Context of Model-Driven Architecture, Anaheim, CA, USA*, October 2003.

[3] A. Gepting, J. Greenyer, E. Kindler, A. Maas, S. Munkelt, C. Pales, T. Pivl, O. Rohe, V. Rubin, M. Sanders, A. Scholand, C. Wagner, and R. Wagner. Component Tools: A vision for a tool. In preparation, July 2004.

[4] J. Greenyer. Maintaining and using component libraries for the design of material flow systems: Concept and prototypical implementation, October 2003.

[5] OMG. *Model Driven Architecture.* http://www.omg.org/mda/.

[6] T. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences 5*, pages 560–595, 1971.

[7] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, Singapore, 1999.

[8] A. Schürr. PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems. Technical Report AIB 94-11, RWTH Aachen, Germany, 1994.

[9] A. Schürr. Specification of graph translators with triple graph grammars. In *Proceedings of the* 20th *International Workshop on Graph-Theoretic Concepts in Computer Science*, Herrschin, Germany, June 1994. Spinger Verlag.

[10] University of Paderborn, Germany. *Fujaba Tool Suite. Online at http://www.fujaba.de/.*