

# Developing Model Transformations with Fujaba

Robert Wagner  
Software Engineering Group  
Department of Computer Science  
University of Paderborn  
Warburger Str. 100  
D-33098 Paderborn, Germany  
wagner@uni-paderborn.de

## ABSTRACT

In this paper, we present FUJABA's tool support for the specification and execution of model transformations based on the bidirectional transformation technique of triple graph grammars. The graphical though formal specification is executed incrementally and enables live model synchronization that helps to propagate changes between related models.

## 1. INTRODUCTION

Today's software engineering tools offer different formal and semi-formal notations for the construction of models. In addition, most tools support some kind of model-to-model transformation technology. Up to now, desired support for model transformations is missing in FUJABA and has to be realized by handcrafted code which is a quite annoying and error prone task.

The requirements and expectations regarding model transformation techniques are very demanding. A practicable solution should allow a visual specification of the transformation rules [4] with an underlying formal foundation.

In practice, the development of a software system is a quite iterative process with frequent modifications of the involved models. For example, after a model transformation often the target model has to be refined towards the final design. These modifications can have some impact on the source model and have to be propagated backward to keep the overall specification consistent. Therefore, a bidirectional model transformation technique is needed.

Moreover, modifications in the source model have to be propagated to the target model in a non-destructive manner. If a model transformation is applied from scratch, the modifications on a more detailed target model are lost. Therefore, we need an incremental transformation technique which is able to synchronize different models.

In order to meet these requirements we employ the visual, formal, and bidirectional transformation technique of triple graph grammars [9]. In this paper, we focus on the provided tool support for the development and execution of model transformations which was realized as a set of plug-ins for the FUJABA TOOL SUITE<sup>1</sup>. For a detailed description of our incremental algorithm and its evaluation we refer to [3].

The remainder of this paper is organized as follows. In the next section we introduce a simple example which will be used throughout this paper in order to explain the realized tool support. In Section 3, we give a brief and informal

<sup>1</sup><http://www.fujaba.de>

introduction to the concepts of triple graph grammars. Section 4 sketches our incremental transformation algorithm. The realized tool support with necessary steps for the development of a model transformation is described in Section 5. The paper closes with a conclusion and an outlook on future work in Section 6.

## 2. EXAMPLE

In order to explain the specification and execution of a model transformation, we introduce a simple example which will be used throughout this paper. The example stems from an earlier project in the domain of production control system. In this project, we combined a subset of the Specification and Description Language (SDL) [5] and the Unified Modeling Language (UML) [7] to an executable graphical language [8]. In this language, a block diagram is used to specify the overall static communication structure where processes and blocks are connected to each other by channels and signal routes. For implementation purposes, the block diagram is transformed to an initial class diagram. This class diagram can be refined and extended, e.g. using state charts, to an executable specification. Figure 1 shows a simple block diagram and the class diagram which results from a correct transformation.

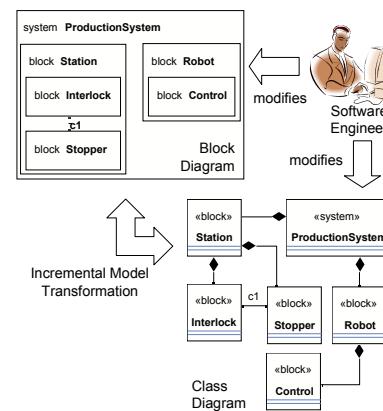


Figure 1: Application example

Basically, systems, blocks, and processes of a block diagram are transformed to classes with corresponding stereotypes. For example, the block *Station* is represented by the class *Station* with a stereotype `<<block>>`, the system *Pro-*

*ductionSystem* as a class with the stereotype «system». The hierarchical structure of the block diagram is expressed by composition relations between the respective classes in the class diagram. The channels and signal routes of the block diagram are mapped to associations between the derived classes. In addition, each signal received by a process in the block diagram is mapped to a method of the corresponding class in the class diagram (not shown in this example).

In order to support an iterative development process without any restrictions on the order of design steps, we allow the engineer to move freely between the block and class diagram to refine and adapt both models towards the final design. In this scenario, we have to ensure that the overlapping parts and mappings between these interrelated models stay consistent to each other. For this purpose, we use the bidirectional and incremental model transformation technique of triple graph grammars. This declarative specification technique is described in the following section.

### 3. TRIPLE GRAPH GRAMMARS

A triple graph grammar [9] is a declarative definition of a bidirectional model transformation and consists of a graphical though formal specification of transformation rules. In Figure 2, a triple graph grammar rule in the FUJABA-notation is shown.

The rule specifies a consistent correspondence mapping between the objects of the source and target model. In particular, the presented rule defines a mapping between a block and a corresponding class. The objects of the block diagram are drawn on the left and the objects of the class diagram are drawn on the right. They are marked with the «left» and «right» stereotypes respectively. The correspondence objects in the middle of the rule are tagged with the «map» stereotype.

The rule is separated into a triple of productions (source production, correspondence production, and target production), where each production is regarded as a context-sensitive graph grammar rule. A graph grammar rule consists - as all grammars - of a left-hand side and a right-hand side. All objects which are not marked with the «create» stereotype belong to the left-hand side and to the right-hand side; the objects which are tagged with the «create» stereotype occur on the right-hand side only. In fact, these tags make up a production in FUJABA's graph grammar notation.

The production on the left shows the generation of a new sub block and linking it to an existing parent block. The production on the right shows the addition of a new class and stereotype and its linking to the class diagram. Moreover, to reflect the containment of the sub block, a composition association is created between the classes representing the parent block and the sub block. For this purpose, the rule contains additional objects representing the roles and cardinalities of the association. The correspondence production shows the relations between a block and a class and an additional constraint  $\{block.name == clazz.name\}$  specifies that the block and the class have to be named uniquely.

Up to this point, the assignments and constraints to the object attributes have not been considered yet. Since triple graph grammars can be executed in both directions, the attribute constraints help to identify the objects to be matched, whereas the attribute assignments are applied only to created objects. However, since the computations of the attribute values may be more complicated than in our simple

example, the assignments cannot be always derived from the constraints and have to be specified explicitly.

A graph grammar rule is applied by substituting the left-hand side with the right-hand side if the pattern of the left-hand side can be matched to a graph, i.e., if the left-hand side is matched all objects tagged with the «create» stereotype will be created. Hence, our example rule, in combination with additional rules covering other diagram elements, can generate a set of blocks along with the corresponding classes and associations in a class diagram. Though the transformation will not be executed this way, conceptually, we can assume that whenever a block is added to the block diagram, a corresponding class with an appropriate association will be generated in the class diagram. This way, the triple graph grammar rules define a transformation between block diagrams and class diagrams.

The correspondence production in the middle of the rule enables a clear distinction between the source and target model and holds additional traceability information. This information can be used to realize bidirectional and incremental model transformations that helps to propagate changes between related models, i.e., it helps to realize live model synchronization between different but related models.

### 4. TRANSFORMATION ALGORITHM

Before we explain our incremental transformation algorithm we have to take a closer look at the correspondence metamodel shown in Figure 3. The metamodel defines the mapping between a source and a target metamodel by the classes *TGGNode* and *Object* and its associations *sources* and *targets*. Since all classes inherit implicitly from the *Object* class (not shown here), the correspondence model stores the traceability information needed to preserve the consistency between two models. In addition, the class *TGGNode* has a self-association *next* which connects the correspondence nodes with their successor correspondence nodes. This extra link is used by our transformation algorithm.

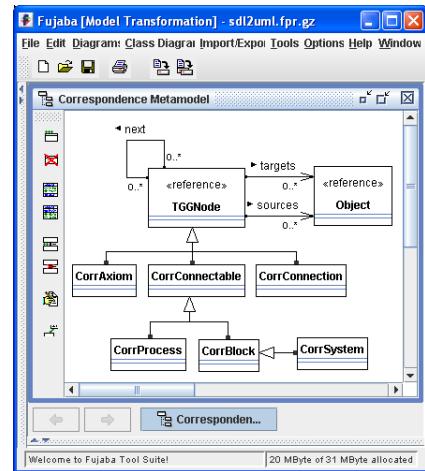
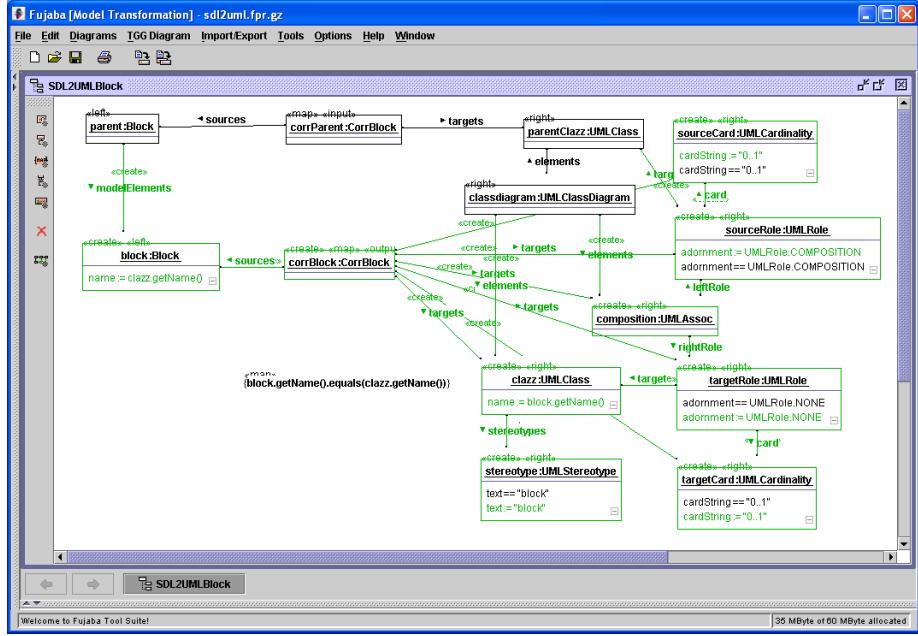


Figure 3: Correspondence metamodel

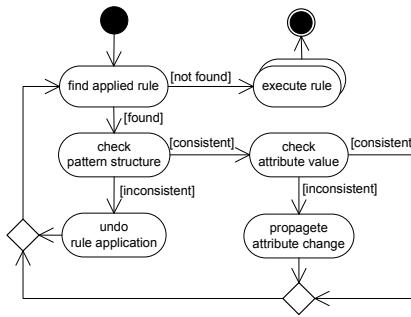
The two described classes and their associations are essential for our transformation algorithm. However, further correspondence nodes and refined associations can be added. In our example, we have added six additional correspondence nodes, including the correspondence node *CorrBlock* used in



**Figure 2:** A triple graph grammar rule

our example rule (cf. Figure 2). The additional correspondence classes increase the performance of our transformation algorithm since for a given correspondence node type only those rules have to be checked that have the same correspondence node type on their left-hand side.

The transformation algorithm exploits the extra information from the correspondence metamodel for the incremental model synchronization. In Figure 4 an overview of the realized algorithm is shown. The incremental transformation and update algorithm traverses the correspondence nodes along the *next* links of the *TGGNode* objects using breadth-first search. First, for each correspondence node the algorithm checks whether an inconsistent situation has occurred. This is done by retrieving an applied rule (*find applied rule*) and checking whether it still matches to the pattern structure (*check pattern structure*).



**Figure 4:** Transformation and update algorithm

If the rule cannot be matched anymore, e.g., due to the deletion of a model element, we have found indeed an inconsistency. In that case, the algorithm has to undo the applied transformation rule (*undo rule application*). This is achieved by deleting the correspondence node and all created ele-

ments. Note that by deleting the correspondence node the precondition for all successors of the deleted correspondence node will not hold anymore. As a consequence, this leads both to the deletion of the succeeding correspondence nodes and the elements in the class diagram referenced by the deleted correspondence nodes.

In the case that the structure of the applied rule still holds and only an attribute constraint evaluates to false (*check attribute value*), it is sufficient to propagate the attribute value change in the current transformation direction (*propagate attribute change*).

If all old rule applications have been checked, the algorithm searches for new model elements and transforms those elements according to the triple graph grammar specification (*execute rule*). Note that the transformation is executed as long as rules are applicable which is indicated by a *for-each* activity (*execute rule*).

The briefly described algorithm was implemented as a plug-in for FUJABA and is an essential part of the realized tool support which is described in the following section.

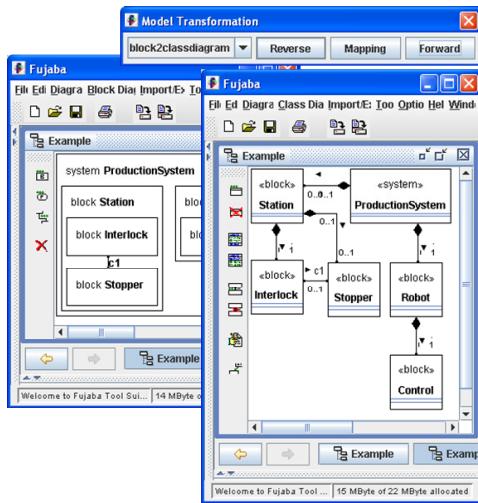
## 5. TOOL SUPPORT

For the visual specification of a triple graph grammar rule we use the TGGEDITOR (cf. Figure 2). This editor is a realized as a FUJABA plug-in and ensures conformance to the source, the correspondence, and the target metamodels. For this purpose, the required metamodels have to be specified in FUJABA as class diagrams.

The execution of a model transformation is done by the new FUJABA plug-ins MOTE and MORTEN. MOTE is the abbreviation for Model Transformation Engine. It is the core library for the execution of triple graph grammars and can be also used without FUJABA. MORTEN is the abbreviation for Model Round Trip Engineering. MORTEN integrates the MOTE library into FUJABA and provides a graphical user interface to setup and control several transformation tasks.

In order to execute the algorithm presented in Section 4, we derive from each rule story diagrams implementing the activities shown in Figure 4. From the automatically derived story diagrams, we generate Java code using FUJABA's code generation facilities. This code is compiled to executable transformation rules which are bundled into a single JAR archive file. In addition, the JAR file has to provide a configuration file with listed transformation rules. The archive represents the catalog of transformation rules defining the model transformation specified by a triple graph grammar. Once the catalog is available, model transformations can be carried out.

The first step to transform our block diagram is to setup an appropriate transformation task. For this purpose, we have to name the transformation task and select the catalog containing the rules for the transformation. Thereafter, we have to select the source and/or the target diagram for the transformation. For example, we can select a block diagram and transform it initially to a class diagram. Or, we can select a class diagram and transform it backward to a block diagram. If we select two diagrams, i.e., a class and a block diagram, both diagrams are checked for corresponding parts and the related parts are connected by correspondence nodes. After an initial transformation, the block and class diagram can be modified and further refined. For synchronization purposes, the incremental transformation can be re-executed each time a diagram changes (cf. Figure 5).



**Figure 5: Block to class diagram transformation**

MORTEN is intended to be used to test a specified model transformation during development. After a transformation is specified and tested, it can be integrated into any Java-based software tool with FUJABA-compliant metamodels or appropriate model adapters using the MoTE library. An example for such an integration is presented in [2] where Matlab/Simulink models are automatically transformed to pattern specifications by utilizing the MoTE plug-in.

## 6. CONCLUSION AND FUTURE WORK

In this paper, we have presented the realized tool support for the model-based development of model transformation rules based on the technique of triple graph grammars. The available tool support includes a FUJABA plug-in for the

visual specification of triple graph grammar rules, the automatic derivation of graph rewriting rules in the form of story diagrams, and a plug-in for the incremental execution of these rules.

We are currently working on an approach which eases the specification of the triple graph grammar rules and enables the specification of rules in the concrete syntax of the appropriate models. At the same time, we are extending the model synchronization support by some auto-completion capabilities to allow a synchronization between already existing models.

As future work we plan to provide a QVT [6] compatible front-end in order to make the technology available to a broader audience. In addition, we plan to port our tools to the Eclipse platform to enable the transformation of EMF-compliant models [1].

## 7. REFERENCES

- [1] The Eclipse Foundation. *Eclipse Modeling Framework (EMF)*, available at <http://www.eclipse.org/emf>, 2006.
- [2] H. Giese, M. Meyer, and R. Wagner. A Prototype for Guideline Checking and Model Transformation in Matlab/Simulink. In H. Giese and B. Westfechtel, editors, *Proc. of the 4<sup>th</sup> International Fujaba Days 2006, Bayreuth, Germany*, volume tr-ri-06-275 of *Technical Report*. University of Paderborn, September 2006.
- [3] H. Giese and R. Wagner. Incremental Model Synchronization with Triple Graph Grammars. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proc. of the 9<sup>th</sup> International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genoa, Italy*, volume 4199 of *Lecture Notes in Computer Science*, pages 543–557. Springer Verlag, October 2006.
- [4] L. Grunske, L. Geiger, and M. Lawley. A graphical specification of model transformations with triple graph grammars. In *2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, 2005.
- [5] International Telecommunication Union (ITU), Geneva. *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*, 1994 + Addendum 1996.
- [6] OMG. *MOF QVT Final Adopted Specification*, *OMG Document ptc/05-11-01*. <http://www.omg.org/>.
- [7] OMG, 250 First Avenue, Needham, MA 02494, USA. *Unified Modeling Language Specification Version 1.5*.
- [8] W. Schäfer, R. Wagner, J. Gausemeier, and R. Eckes. An Engineer's Workstation to support Integrated Development of Flexible Production Control Systems. In H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*. Springer Verlag, September 2004.
- [9] A. Schürr. Specification of graph translators with triple graph grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science, 20<sup>th</sup> International Workshop, WG '94*, volume 903 of *LNCS*, pages 151–163, Herrsching, Germany, June 1994.