A Graphical Specification of Model Transformations with Triple Graph Grammars

Lars Grunske[1], Leif Geiger[2], and Michael Lawley[3]

[1] School of Information Technology and Electrical Engineering,
University of Queensland, Brisbane, QLD 4072, Room 72-458 IT Building
grunske@itee.uq.edu.au
http://www.itee.uq.edu.au/

[2] University of Kassel, Software Engineering Research Group,
Department of Computer Science and Electrical Engineering,
Wilhelmshöher Allee 73, 34121 Kassel, Germany
leif.geiger@uni-kassel.de
http://www.se.eecs.uni-kassel.de/se/

[3] CRC for Enterprise Distributed Systems Technology (DSTC)**,
University of Queensland,
Brisbane, QLD 4072, Australia
michael@lawley.id.au
http://www.dstc.edu.au/

Abstract. Models and model transformations are the core concepts of OMG's MDA^{TM} approach. Within this approach, most models are derived from the MOF and have a graph-based nature. In contrast, most of the current model transformations are specified textually. To enable a graphical specification of model transformation rules, this paper proposes to use triple graph grammars as declarative specification formalism. These triple graph grammars can be specified within the FUJABA tool and we argue that these rules can be more easily specified and they become more understandable and maintainable. To show the practicability of our approach, we present how to generate Tefkat rules from triple graph grammar rules, which helps to integrate triple graph grammars with a state of a art model transformation tool and shows the expressiveness of the concept.

1 Introduction

Model Driven Engineering (MDE) is a software engineering principle that promotes the use of models and transformations as primary development artifacts. To practically apply this principle, the Object Management Group (OMG) has proposed the MDA$^{\text{TM}}$ [1] as a set of standards for integrating MDE tools. The MDA$^{\text{TM}}$ approach separates the specification of systems from the implementation of these systems. For this reason two basic model types are introduced,

** The work reported in this paper has been funded in part by the Co-operative Research Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme.

Platform Independent Models (PIM's) and Platform Specific Models (PSM's) using specific implementation platforms. PIM's can be specified in an abstract style without thinking about platform specific details. If all necessary PIM's are specified, they should be automatically mapped to a platform specific model by adding the platform specific details. To allow this mapping, model transformations are necessary.

The need for standardization of model transformations as well as the generation of views and the definition of queries lead to the MOF 2.0 Query/ Views/ Transformations Request for Proposals (RFP) [2] from the OMG. For this RFP the OMG initially received eight proposals of varying degrees of completeness which are reviewed and assessed in [3]. A final revised, merged submission supported by all original submitters is expected to be voted for adoption at the June 2005 OMG meeting in Boston. This submission supports several flavours of transformation specification allowing for both declarative and procedural specifications.

One important aspect we noticed when reviewing the revised submissions for the RFP is that most transformation languages are specified textually. This conflicts with the graph-based nature of most of the current MOF 2.0 models (e.g. UML 2.0 models). A graph-based transformation language would be more appropriate for specifying and applying model transformations. Having made this observation, we propose to use graph transformations rules (specifically triple graph transformation rules) and graph transformation systems as an extension to the current model transformation languages. These graph transformation rules are a straightforward extension of string or term rewriting rules, which were introduced in the seventies [4] and are currently applied in various domains [5] to transform or rewrite graph-based structures. However, normal graph transformation rules and systems are only suitable to operate on one particular graph. Thus, they are only suited to describe intra-model transformation, as they are needed to specify quality-improving refactorings [6–8]. To operate on two different graphs with two different graph schemata an extension called triple graph grammars [9] is suitable. These triple graph grammars and their rules are the theoretical foundation of this paper and we want to show their suitability for describing complex model-to-model transformations. Especially, we argue that triple graph grammars provide the following benefits, which are useful for the specification and application of model-to-model transformations [10]:

- Triple graph grammars allow incremental change propagations between two models A and B, if a model transformation system with triple graph grammar rules is applied once to transform a model A to a model B and the correspondence graph is generated. This means you can create tools that update a model if the other has been changed. [11, 9] This change propagation is also bidirectional and is especially important for iterative software engineering processes where a model evolves continuously.
- Triple graph grammars can be used to check the consistency between two models.

– Triple graph grammars can be applied to all graph-based data structures and models, not only to tree-based ones. This also includes hierarchical graph-based data structures [12–14], which we think is imported to model transformations between two MOF 2.0-compliant models.

The rest of the paper is structured as follows: Section 2 introduces the basic concepts of graph-based structures and graph transformations in general. Thereafter, Section 3 reviews the concepts of triple graph grammars and shows how triple graph grammar rules can be used to specify model-to-model transformations. In Section 4, the practical applicability of triple graph grammars is presented with the well-known example of the transformation from an object-oriented class diagram model into a relational database model. Section 5 presents an implementation of triple graph grammars within the FUJABA tool and describes how FUJABA could be used to generate textually specified model transformation rules (e.g. Tefkat rules). Before concluding, Section 6 discusses the limitation of triple graph grammars and sets up directions for future work to extend the current model-to-model transformation languages successfully.

2 Preliminaries

This section introduces the basic concepts of graph-based structures and the fundamental graph transformation theory in an informal and intuitive way. In addition, this section presents an overview about useful graph transformation techniques and extensions that are needed to specify model transformations within the MDA^{TM} approach.

2.1 Directed Typed Graphs and Graph Morphisms

We choose directed typed graphs as the basic structure, because they are well suited to specifying different types of models, especially MOF-based models [15]. These directed typed graphs contain nodes and edges that are instances of node and edge types. The instance relation between the nodes and edges and their types is similar to the relation between objects and classes in object-oriented software engineering. Due to this, a node or edge type can contain a set of application specific attributes and operations. To model the graph-based structure each edge is associated with a source and a target node. Formally, a typed graph can be defined as follows:

Definition 1: (Directed Typed Graphs) Let L_V be a set of node types and L_E be a set of edge types; then a directed typed graph G from the possible set of graphs $\mathcal{G}$ over L_V and L_E is characterized by the tuple $\langle V, E, \mathit{source}, \mathit{target}, \mathit{type} \rangle$, with two finite sets V and E of nodes (or vertices) and edges, a function type composed of the two functions $\mathit{type}_V : V \to L_V$ and $\mathit{type}_E : E \to L_E$ which assigns a type to each edge and node and two functions $\mathit{source} : E \to V$ and $\mathit{target} : E \to V$ that assign to each edge a source and a target node.

Another preliminary for the definition of graph transformation systems are graph morphisms. These graph morphisms are structure and type-preserving mappings between two graphs that can be defined as follows:

Definition 2: (Graph Morphism) Let $G = \langle V, E, \mathit{source}, \mathit{target}, \mathit{type} \rangle$ and $G' = \langle V', E', \mathit{source}', \mathit{target}', \mathit{type}' \rangle$ be two graphs; then a graph morphism $m : G \rightarrow G'$ consists of a pair of mappings $\langle m_V, m_E \rangle$, with $m_V : V \rightarrow V'$ and $m_E : E \rightarrow E'$, which satisfy the following conditions (type and structure-preserving):

- $\forall\ e \in E : \mathit{type}'(m_E(e)) = \mathit{type}(e)$
- $\forall\ v \in V : \mathit{type}'(m_V(v)) = \mathit{type}(v)$
- $\forall\ e \in E : \mathit{source}'(m_E(e)) = m_V(\mathit{source}(e))$
- $\forall\ e \in E : \mathit{target}'(m_E(e)) = m_V(\mathit{target}(e))$

If both mappings $m_V : V \rightarrow V'$ and $m_E : E \rightarrow E'$ are injective (surjective, bijective) then the mapping $m : G \rightarrow G'$ is injective (surjective, bijective).

Graph Variants Besides the introduced directed typed graphs, several other variants and extensions gain attention in the graph transformation community. One basic variant uses undirected edges. These undirected edges can be modeled in a directed graph with two contrary edges for each undirected edge. Another variant are hypergraphs[16], where each (hyper) edge is associated with a sequence of source and target node. That means, these edges can have an arbitrary number of source and target nodes. For the construction of hierarchical models, hierarchical graphs are important. These hierarchical graphs model the hierarchical structure either by (hyper)edge [12] or node refinement [17].

2.2 Graph Transformation and Graph Transformation Systems

Basic Principles Graph transformation systems make use of graph rewriting techniques to manipulate graphs. A graph transformation system is defined with a set of graph production rules, where a production rule consists of a left-hand side (LHS) graph and a right-hand side (RHS) graph. Such rules are the graph equivalent of term rewriting rules, i.e., intuitively, if the LHS graph is matched in the source graph, it is replaced by the RHS graph. Intuitively, a graph transformation rule can be defined as follows:

Definition 3: (Graph Transformation Rule) A graph transformation rule $p = \langle G_{LHS}, G_I, G_{RHS}, m_l, m_r \rangle$ consists of three directed typed graphs G_{LHS}, G_I and G_{RHS}, which are called left-hand side graph, interface graph and right-hand side graph. The interface graph G_I is just an auxiliary graph. The morphisms $m_l : G_I \rightarrow G_{LHS}$ and $m_r : G_I \rightarrow G_{RHS}$ are used to describe the correspondence between these graphs and map the elements of the interface graph to either the left-hand side or the right-hand side graph.

For the application of a graph transformation rule to an application graph G_{APP} the following simplified algorithm can be used:

1. Identify the left-hand side G_{LHS} within the application graph G_{APP}. For this, it is necessary to find a total graph morphism $m : G_{LHS} \rightarrow G_{APP}$ that matches the left-hand side G_{LHS} in the application graph G_{APP}.
2. Delete all corresponding graph elements, w.r.t. m, in the application graph G_{APP} that are part of the left-hand side G_{LHS} and are not part of the interface graph G_I.
3. Create a graph element in the application graph G_{APP} for each graph element that is part of the right-hand side G_{RHS} and is not part of the interface graph G_I. Connect or glue these added graph elements to the rest of the application graph G_{APP}.

For a formal description of the rule application formalisms, we refer to [18, 19], where the formal foundations of the single pushout (SPO) and double pushout (DPO) approach are reviewed. Currently these approaches have the most impact in the graph transformation community [5, 15, 8].

Application Conditions In complex graph transformation systems it is often necessary to restrict the application of single rules. Therefore, in [20] the concept of positive and negative application conditions (PACs and NACs) is introduced. These application conditions are formally graphs that define a required context (PACs; e.g. the presence of nodes or edges) or a forbidden context (NACs; e.g the absence of nodes or edges). The fulfillment of these application conditions must be checked before the rule is applied. Consequently, the algorithm in the previous Section must be extended by adding another step after the first one, which checks the application conditions. In this paper, we use crossed out nodes to visualize nodes belonging to the negative application condition.

Specification of Graph Transformation Rules In traditional approaches for specification of graph transformation rules the right hand side and the left hand side of a rule are drawn separately [19, 18]. Throughout this paper the approach of the Fujaba Tool [21], that combines both sides, is used. Fujaba uses UML collaboration diagrams to model graph transformations. Nodes become objects and edges become links between objects. Objects and links that have no stereotypes appear on both sides of the graph transformation rule. Objects and links marked with the ≪*destroy*≫ stereotype appear only on the left hand side, i.e. they are deleted. The stereotype ≪*create*≫ marks elements only used on the right hand side, i.e. such elements are created. Fujaba uses programmed graph transformation rules. This means a control structure can be specified that manages the order of the execution of transformation rules. Such control structure is modeled using UML activity diagrams. The transformation rules are then embedded into the activities. Fujaba makes use of typed graphs. In the graph transformations, the type is specified after the object's name, and separated by a colon. More elaborate elements of graph transformations like negative application conditions, multi objects and non-injective matching are also supported by the Fujaba tool. All of these features will be needed for the effective specification of model transformation rules.

Figure 1 shows a graph transformation rule in Fujaba. The rule consists only of one transformation that deletes every column in a table that has the same name as another column. To achieve this, the transformation tries to match an object *c1* of type *Column* which has a *col* link to a *Table* object *t*. This object must itself have a *col* link to another object of class *Column*. If this *column* has the same attribute value for its attribute *name* as the object *c1*, the matching can be applied. If a matching is found, the column *c2* and its *col* link will be destroyed. Note, that the activity has a doubled border. Such an activity, a so-called for-each activity, is applied as long as a matching is found. Thus, the rule deletes every duplicated column in every table.

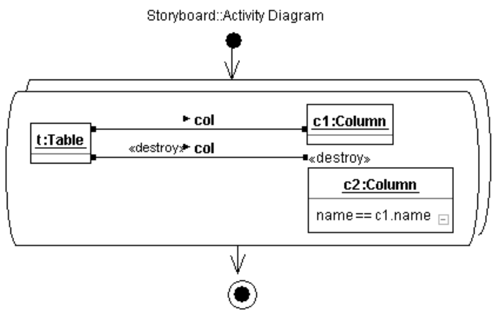

Fig. 1: Graph transformation "Fujaba-style"

3 Triple Graph Grammars

In this section we introduce the concept of triple graph grammars (TGG) and describe their suitability to extend current model transformation systems. Thereafter, we show how triple graph grammars can be specified within the Fujaba tool and how forward and backward transformations can be derived from these triple graph grammar rules.

3.1 Introduction

Triple graph grammars are a straightforward extension of pair grammars and pair grammar rules that were introduced by Pratt [17] in the early seventies. These pair grammars are used to specify graph-to-string translations. By this means, a pair grammar rule rewrites two models: a source graph and a target string. Thus, it contains a pair of production rules (a graph and a string production rule), which modify simultaneously the two participating models. Because of this, pair grammars are well suited to specify transformations between graphs and strings. If the string production rule is substituted by a graph production rule, these pair grammars can be also used for graph-to-graph translations.

Triple graph grammars, as introduced in the early nineties [22] are used for graph-to-graph translations and data integration. Each triple graph grammar rule contains three graph productions; one operates on a source graph, one on the target graph and one on a correspondence graph. The correspondence graph describes a graph-to-graph mapping that relates elements of the source graph to elements of the target graph. Based on this mapping, incremental change propagations, that update the target graph if an element in the source graph is changed, are possible. Formally, triple graph grammar rules can be defined as follows [22]:

Definition 4: (Triple Graph Transformation Rule) A triple graph transformation rule $tgg = \langle p_{left}, p_{right}, p_{map} \rangle$ consists of three graph transformation rules p_{left}, p_{right} and p_{map} where p_{left} transforms the source model, p_{right} transforms the target model and p_{map} transforms a relation model that maps source to target elements. All three graph production are applied simultaneously.

3.2 Specification

To specify a complex triple graph grammar rule all three graph grammar rules should be specified in one rule diagram. For the specification of each single rule we use the Fujaba-style and separate the three rules within the rule diagram. Due to this separation a user can identify to which side the element belongs.

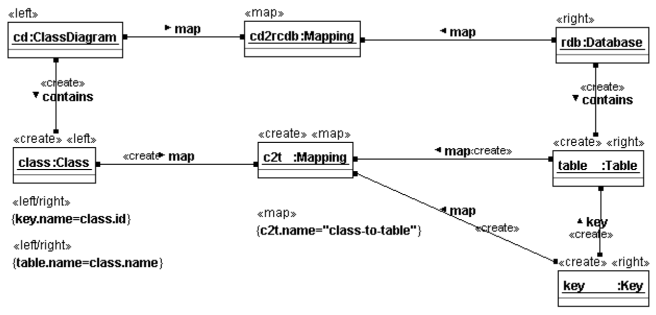

Fig. 2. Example of a TGG rule class-to-table in FUJABA

Figure 2 shows a transformation rule that contains seven objects; two source model objects, three target model objects and two correspondence model objects. The objects from the source model are drawn left, the objects of the target model are drawn right and the objects of the correspondence model are drawn in the middle of the rule diagram. Additionally, they are marked with the stereotypes ≪*left*≫, ≪*map*≫ or ≪*right*≫. The rule shown in the figure demonstrates a mapping between classes in a class diagram and tables in a relational database. The precondition of this rule is drawn in the top of the diagram. This means, to apply this rule there must already exist a class diagram which is mapped via a *Mapping* node to a relational database. The elements which have to be related are drawn green with a ≪*create*≫ stereotype. This means, an object of the type *Class* is mapped to an object of the type *Table* which has a link to a key object and vice versa. Attribute conditions are modeled as constraints. For example, the *id* of a class is stored as *name* of the table's key.

A triple graph grammar rule enable the generation of three transformation rules [9]: the forward rule, the reverse rule and a relation rule that checks the consistency of both models. The forward rule is created by removing the ≪*create*≫ stereotype from all elements which belongs to the source (≪*left*≫) model. The reverse rule is created by removing the ≪*create*≫ stereotype from all elements which belongs to the target (≪*right*≫) model. The last rule, the relation rule, is

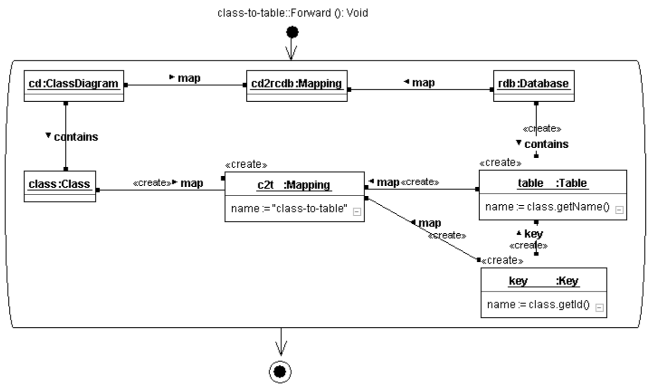

Fig. 3. Forward rule derived from the TGG-rule class-to-table

derived by removing the ≪*create*≫ stereotype from all elements that does not belong to the correspondence (≪*map*≫) model.

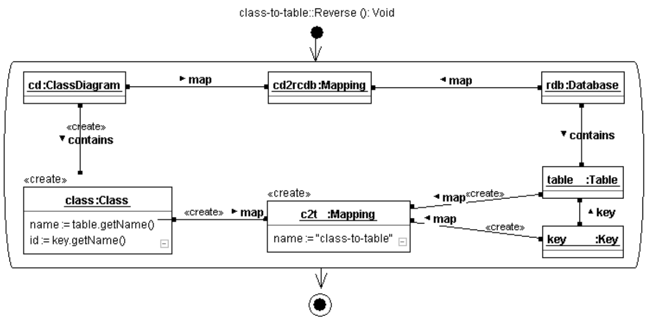

Fig. 4. Reverse rule derived from the TGG-rule class-to-table

Figure 3 shows the derived forward rule. If a mapping from a class diagram to a relational database exists and if the class diagram contains a class, a new table and a new key are created and its attributes are set accordingly to the TGG rule. These newly created objects are marked as being mapped to the class using a new mapping node.

Figure 4 shows the reverse rule, which will create a class for every table in the relational model. The matching and creation of objects is done in the same manner as is done for the forward rule.

The last rule created is the relation rule shown in Figure 5. This rule needs a class diagram and a database and tries to relate them. This will result in a consistency check between these two models. Therefore, the only objects created in this rule are the mapping nodes.

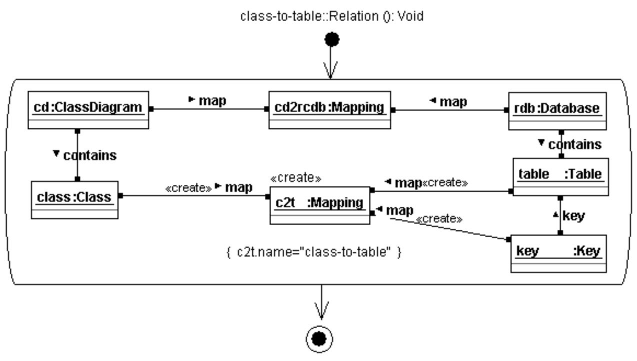

Fig. 5. Relation rule derived from the TGG-rule class-to-table

The rule in Figure 5 searches for a class diagram that has already been mapped to a relational database. If the class diagram contains a class which can be related to a table as specified in the rule, a new mapping is created. If a mapping can be created for all classes and for all tables the two models are consistent.

4 Example

To show the practical applicability of triple graph grammars in the context of model transformations we use the well-known example of the transformation from an object-oriented class model to a relational database model. This transformation is required if an application needs to store a set of objects persistently in a database. A text-based realisation of this example can be found in several QVT-proposals [23, 24]. A graphical specification of the transformation rules of this example can be found in [25].

The basic meta-models, the object-oriented class model and the relational database model, for this transformation are presented in Figure 6. To keep the transformations simple the object-oriented model is cut down in the following aspects:

– A class can contain only attributes and no methods, because methods don't need to be stored persistently.
– Only 1:1 and 1:n associations are considered. These associations are represented by attributes that have classes as types. For a modelling of m:n associations a new meta class *Association* need to be introduced, that has a source and a target association to the meta-class *Class*.
– In some examples, the meta-class *Classifier* or *Class* has a Boolean attribute *ispersistent* that is used to mark all objects that need to be stored in the database. For simplicity in our transformation we transform the complete object-oriented model into a relational model.

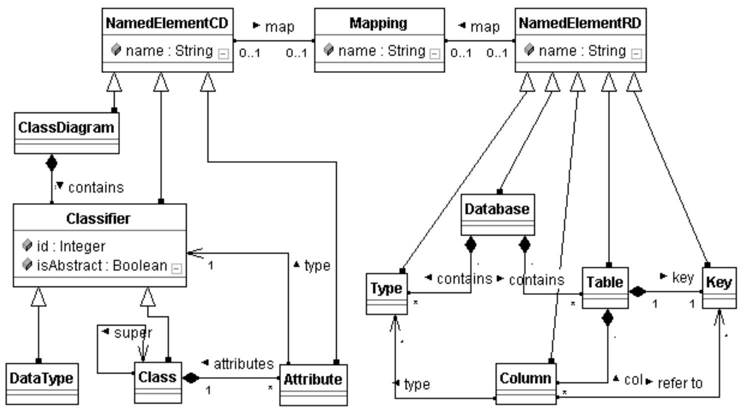

Fig. 6. Meta-models including the mapping relation

The effect of these simplifications will be described and discussed in the following, when we describe the transformations in detail.

To transform an instance of the object-oriented meta-model into the relational model the following natural language rules(requirements/laws) are used:

- *Classes* correspond to *Tables* that have a unique *Key*. This *Key* is identical to the *id* of the *Class*.
- *Types* in the relational model correspond to simple *Datatypes* in the object-oriented model.
- *Attributes* are stored in *Columns*, where each *Column* is owned by the *Table* of the corresponding *Class*.

To implement the transformations and consistency checks between the object-oriented class model and the relational database model a set of triple graph grammar rules must be created for each law. The triple graph grammar rule for the first law, the mapping from classes to tables, has already been discussed in Section 3.2. The rule for the second law is very similar. This rule relates each *Datatype* in the object-oriented model and each *Type* in the relational model the same way as it is done for the first law.

To store the attributes in columns and vice versa, as requested in the third law, we need to distinguish between attributes of a simple type and attributes that are classes. Due to this reason, we need to specify two different TGG-rules (cp. Figure 7). To get an impression what these rules do, we now have a look at the forward rules, which can be generated from the two TGG-rules. The first rule searches for all *Attributes* of a *Class* that are typed by a DataType. It then creates a *Column* for each *Attribute* and assigns the *Column* to the *Table* of the *Class* and to the *Type* of the corresponding *Attribute*. The second rule tries to match all *Attributes* of a *Class* that refer to another *Class*. If this rule finds such a match, it creates a new *Column* and assigns it to the *Table* of the *Class* to which the *Attribute* belongs. To set the correct *Type* of the *Attribute* the *Table*

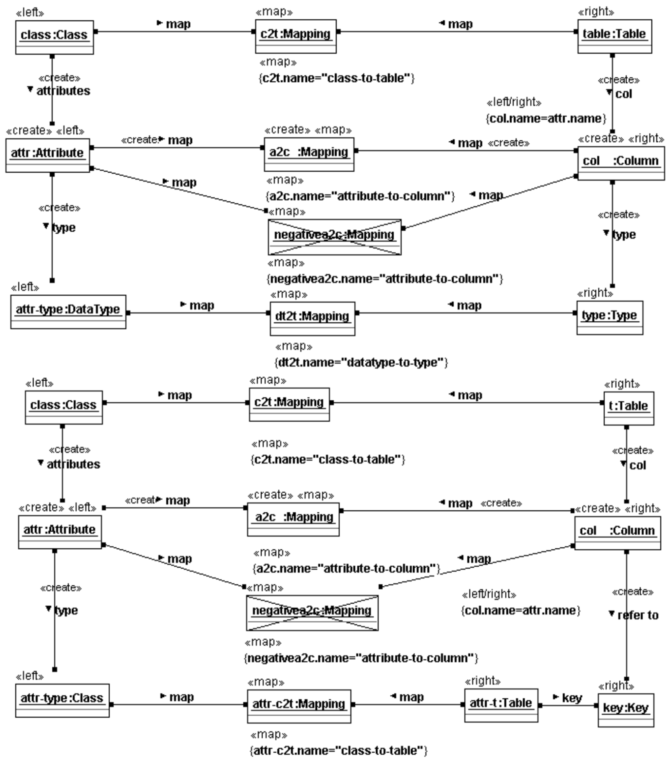
«left» class:Class
map
«map» c2t:Mapping
map
«right» table:Table
«map» {c2t.name="class-to-table"}
«create» attributes
«create» col
«left/right» {col.name=attr.name}
«create» «left» attr:Attribute
«create» map
«create» «map» a2c :Mapping
map «create»
«create» «right» col :Column
map
«map» {a2c.name="attribute-to-column"}
map
«map» negativea2c:Mapping
«create» type
«create» type
«map» {negativea2c.name="attribute-to-column"}
«left» attr-type:DataType
map
«map» dt2t:Mapping
map
«right» type:Type
«map» {dt2t.name="datatype-to-type"}
«left» class:Class
map
«map» c2t:Mapping
map
«right» t:Table
«map» {c2t.name="class-to-table"}
«create» attributes
«create» col
«create» «left» attr:Attribute
«creat» map
«create» «map» a2c :Mapping
map «create»
«create» «right» col :Column
map
«map» {a2c.name="attribute-to-column"}
map
«map» negativea2c:Mapping
«create» type
«left/right» {col.name=attr.name}
«create» refer to
«map» {negativea2c.name="attribute-to-column"}
«left» attr-type:Class
map
«map» attr-c2t:Mapping
map
«right» attr-t:Table
key
«right» key:Key
«map» {attr-c2t.name="class-to-table"}

Fig. 7. TGG-rules for attributes and columns

of the *Class* is identified with the mapping relation and the association *refer to* is set to the *Key* of this *Table.* [4] [5]

5 Implementation

For specifying triple graph grammar rules we use FUJABA's [21] TGG Plug-in, which provides all necessary triple graph grammar concepts for specifying model transformations. Additional, the TGG-Plug-in is open source and easily extendable.

[4] Note that before these rules can be applied it is necessary to apply the rules that transform *Classes* and *Datatypes*. If these rules are not previously applied, the mapping relations are missing and the LHS of the TGG-rule can't be matched. This leads to an implicit specification of the ordering of the rules.

[5] With the described transformations of attributes that are typed by classes, all 1:n and 1:1 associations can be transformed. To transform the m:n associations an additional rule is necessary. This rule creates a new table for all m:n associations and stores the links to the corresponding classes as foreign keys in this table.

To execute model transformations, that are specified as TGG rules, two options are provided by the FUJABA tool. As a first option, story diagrams could be generated from TGG rules. These story diagrams can be used to generate Java code, which enables a so called in-memory model transformation [26].

Based on this approach we have implemented a second alternative, which generates rules that can be used in the Tefkat tool [27]. Tefkat is an implementation of the transformation language proposed by DSTC et al [23] in their response to the OMG's QVT RFP. It is a declarative, logic-based language with a fixpoint semantics. It supports single-direction transformation specifications from one or more source models to one or more target models. The transformation specifications are constructive meaning that they specify the construction of the target model(s). There is currently no support for in-place update of models.

The Tefkat implementation is based on the Eclipse Modeling Framework (EMF) [28] and supports transforming native Ecore models as well as those based on MOF2, UML2, and XMLSchema. It is usable in both standalone form and as an Eclipse plugin with a source-level debugger.

To generate Tefkat rules the TGG plug-in identifies for each object, link and constraint its position (e.g. ≪*left*≫, ≪*map*≫ and ≪*right*≫) and its modifier (e.g. ≪*create*≫ and ≪*delete*≫) and fills based on this information a template for the forward, backward and consistency checking rule. The complete algorithm follows the basic concepts given in Section 3. As an example from the TGG rule 2 the Tefkat rule presented in Fig 8 will be generated.

```
RULE class-to-tableForward(class, cd, rdb, key, table)
 FORALL Class class, ClassDiagram cd, Database rdb
  WHERE cd2rcdb LINKS cd=cd, db=rdb
        AND cd.contains=class
   MAKE Key key, Table table
    SET key.key=table,
        rdb.contains=table,
        key.name=class.id,
        table.name=class.name
LINKING c2t
   WITH table=table, class=class, key=key;
```

Fig. 8. Forward Rule in Tefkat

6 Discussion

With the transformation example given in Section 4 we have shown that triple graph grammars and triple graph grammar rules are suitable to specify simple inter-model-transformations. However, we still see some problems in the application of these triple graph transformation rules to real world transformation problems. First, the success of all transformation languages within the MDA depends mostly on the performance of the application of the transformation rules.

It is unacceptable to wait several hours until all rules are applied. The most time consuming task within the application of triple graph grammar rules, is to find all matches of the left hand side (LHS) in an application graph. Consequently, it is necessary to optimise the rule matching algorithms. This can be done e.g. by specifying or identifying an optimal order in which the objects should be matched.

Another important point is an optimal support for an easy specification of transformation rules. This includes a tool that guides the user within the specification without restricting them too much. At this point Fujaba helps a lot. However, it can still be improved. The other aspects that comes to mind when talking about the easy specification of a rule, is a support to reuse and to adapt existing rules. This includes extending and superseding rules as described in [23]. To our current knowledge, there is no theoretical concept for applying inheritance to graph grammar rules.

Finally, current triple graph grammar rules are only suited to model transformations between one source and one target model. To solve this problem in [29], an extension, called MDI-rules, is presented, which allows transformation between N source models and M target models. To provide this possibility, for each additional source or target model an additional graph production rule must be specified. This means, that a 1-to-2 transformation must be specified with quadruple graph grammar rules. However, in most cases these N-to-M transformations can be also specified with a set of 1-to-1 transformation rules, except from the case of model merging (N-to-1), where not only tree-based structures are involved [30].

7 Conclusion and Future Work

This paper describes a possible extension for the current transformation language within the MDA^{TM} approach. This extension is based on triple graph grammars and triple graph grammar rules, which provide a deep theoretical concept for data integration [9] between different graph-based structures. Thus, they can easily be adapted for model-to-model transformations [26]. An important feature of triple graph grammar rules is the implicit creation of a correspondence graph between the two models. This allows incremental change propagation in case one model evolves. The practical applicability of triple graph grammars for model-to-model transformations is presented with the well-known example of the transformation from an object-oriented class model to a relational database model.

The main benefit of triple graph grammars is the ability to graphically specify transformation rules. However, it needs to be proven that these graphical transformation rules are really easier to specify and to maintain. One possibility to prove this would be with empirical studies. We are currently planning such study with student teams that need to specify and maintain (implement new requirements) a complex model-to-model transformation system.

Besides the benefits of triple graph grammars, we have also discussed (cp. Section 6) the existing problems with applying TGG-rules in real-world model transformation systems. Thereby, especially the extension of the theoretical concepts to allow inheritance and M-to-N transformations with TGG-rules seem to be interesting research topics. With these extensions and optimized algorithms for matching the right hand side of a rule, we think that triple graph grammars can become a useful concept for specifying and applying model-to-model transformations within the model driven engineering paradigm.

References

1. OMG (The Object Managemant Group): MDA specifications, http:// www.omg.org/ mda/ specs.htm. (2002-2004)
2. OMG (The Object Managemant Group): OMG MOF 2.0 query, views, transformations request for proposals (QVT RFP), http://www.omg.org/ techprocess/ meetings/ schedule/ MOF 2.0 Query View Transf.RFP.html or http://www.omg.org/docs/ad/02-04-10.pdf (2002)
3. Gardner, T., Griffin, C., Koehler, J., Hauser, R.: A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard, http://www.omg.org/docs/ad/03-08-02.pdf (2003)
4. Ehrig, H., Pfender, M., Schneider, H.J.: Graph grammars: An algebraic approach. In Book, R.V., ed.: Proceedings of the 14th Annual Symposium on Switching and Automata Theory, University of Iowa, IEEE Computer Society Press (1973) 167–180
5. Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.J., Kuske, K., Plump, D., Schürr, A., Taentzer, T.: Graph transformation for specification and programming. Science of Computer Programming **34** (1999) 1–54
6. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison Wesley (1999)
7. Van Gorp, P., Van Eetvelde, N., Janssens, D.: Implementing refactorings as graph rewrite rules on a platform independent meta model. In: Proceedings of Fujaba Days 2003. (2003)
8. Mens, T., Demeyer, S., Janssens, D.: Formalising behaviour preserving program transformations. In: Graph Transformation. Volume 2505 of Lecture Notes in Computer Science., Springer-Verlag (2002) 286–301
9. Schürr, A., Winter, A., Zündorf, A.: Graph grammar engineering with PROGRES. In: Proceedings 5th European Software Engineering Conference ESEC. Volume LNCS 989., Springer (1995) 219–234
10. Grunske, L., Geiger, L., Zündorf, A., VanEetvelde, N., VanGorp, P., Varró, D.: Using graph transformation for practical model driven software engineering. In: Model-driven Software Development - Volume II of Research and Practice in Software Engineering, edited by Sami Beydeda and Volker Gruhn, ISBN: 3-540-25613-X. (2005) 91–119
11. Becker, S., Haase, T., Westfechtel, B., Wilhelms, J.: Integration tools supporting cooperative development processes in chemical engineering. In: Proceedings Integrated Design and Process Technology (IDPT-2002), Pasadena, California (2002)
12. Drewes, F., Hoffmann, B., Plump, D.: Hierarchical graph transformation. J. Comput. Syst. Sci. **64** (2002) 249–283

13. Grunske, L.: Automated software architecture evolution with hypergraph transformation. In: 7th International Conference Software Engineering and Application (SEA 03), Marina del Ray, CA, USA (2003) 613–621
14. Grunske, L.: Formalizing architectural refactorings as graph transformation systems. In: Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD05), Towson, Maryland, USA, IEEE Computer Society, IEEE Computer Society (2005) 324–329
15. Baresi, L., Heckel, R.: Tutorial introduction to graph transformation: A software engineering perspective. In: International Conference on Graph Transformation, ICGT, LNCS. Volume 2505 of Lecture Notes in Computer Science., Springer (2002) 402–439
16. Habel, A.: Hyperedge replacement: grammars and languages. Volume 643 of Lecture Notes in Computer Science. Springer-Verlag Inc., New York, NY, USA (1992)
17. Pratt, T.W.: Pair grammars, graph languages and string-to-graph translations. Journal of Computer and System Sciences **5** (1971) 560–595
18. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation I: Basic concepts and double pushout approach. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations. World Scientific (1997) 163–246
19. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach. In Rozenberg, G., ed.: The Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations. World Scientific (1997) 247–312
20. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26** (1996) 287–313
21. FUJABA: (Fujaba homepage http://www.fujaba.de/)
22. Schürr, A.: Specification of graph translators with triple graph grammars. In: Proceedings 20th Workshop on Graph-Theoretic Concepts in Computer Science. (1994) 151–163
23. DSTC/IBM/CBOP: Second revised submission for MOF 2.0 Query / Views / Transformations RFP, http://www.omg.org/ docs/ad/04-01-06.pdf (2004)
24. QVT-Partners: Revised submission for MOF 2.0 Query / Views / Transformations RFP, http://www.omg.org/docs/ ad/03-08-08.pdf (2003)
25. Jahnke, J.H.: Management of Uncertainty and Inconsistency in Database Reengineering Processes, Ph.D Thesis Uni Paderborn (2002)
26. Kindler, E., Rubin, V., Wagner, R.: An Adaptable TGG Interpreter for In-Memory Model Transformation. In Schürr, A., Zündorf, A., eds.: Proc. of the 2nd International Fujaba Days 2004, Darmstadt, Germany, University of Paderborn (2004) 35–38
27. DSTC: Tefkat: The EMF Transformation Engine, online documentation. (`http://www.dstc.edu.au/tefkat/`)
28. Merks, E., Eliersick, R., Grose, T., Budinsky, F., Steinberg, D.: The Eclipse Modeling Framework. Addison Wesley (2003)
29. Königs, A., Schürr, A.: Multi-domain integration with mof and extended triple graph grammars. In: in Proceedings of the Dagstuhl Seminar 04101, Language Engineering for Model-Driven Software Development J. Bzivin (Univ. Nantes, FR), R. Heckel (Univ. Paderborn, DE), Dagstuhl (2004)
30. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Software Eng. **28** (2002) 449–462