Electronic Communications of the EASST
Volume 57 (2013)

Proceedings of the Second International Workshop on Bidirectional Transformations (BX 2013)

A Survey of Triple Graph Grammar Tools

Stephan Hildebrandt[1], Leen Lambers[1], Holger Giese[1]
Jan Rieke[2], Joel Greenyer[3], Wilhelm Schäfer[2]
Marius Lauder[4], Anthony Anjorin[4] and Andy Schürr[4]

17 pages

Guest Editors: Perdita Stevens, James F. Terwilliger
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Survey of Triple Graph Grammar Tools

Stephan Hildebrandt[1], Leen Lambers[1], Holger Giese[1]
Jan Rieke[2], Joel Greenyer[3], Wilhelm Schäfer[2]
Marius Lauder[4], Anthony Anjorin[4] and Andy Schürr[4]

[1] Hasso-Plattner-Institut
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
firstname.surname@hpi.uni-potsdam.de

[2] Universität Paderborn, Heinz Nixdorf Institut
Zukunftsmeile 1
33102 Paderborn, Germany
firstname.surname@uni-paderborn.de

[3] Politecnico di Milano
Dipartimento di Elettronica e Informazione
via Golgi, 42
20133 Milano, Italy
surname@elet.polimi.it

[4] Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany
firstname.surname@es.tu-darmstadt.de

Abstract: Model transformation plays a central role in Model-Driven Engineering (MDE) and supporting *bidirectionality* is a current challenge with important applications. Triple Graph Grammars (TGGs) are a formally founded, bidirectional model transformation language shown by numerous case studies to be promising and useful in practice. TGGs have been researched for more than 15 years and multiple TGG tools are under active development. Although a common theoretical foundation is shared, TGG tools differ considerably concerning expressiveness, applicability, efficiency, and the underlying translation algorithm. There currently exists neither a quantitative nor a qualitative overview and comparison of TGG tools and it is quite difficult to understand the different foci and corresponding strengths and weaknesses. Our contribution in this paper is to develop a set of criteria for comparing TGG tools and to provide a concrete quantitative and qualitative comparison of three TGG tools.

Keywords: bidirectionality, triple graph grammars, MoTE, TGG Interpreter, eMoflon

1 Motivation

Triple Graph Grammars (TGGs) [KLKS10] are a formally founded, rule-based and declarative bidirectional model transformation language, shown by numerous case studies [GNH10, GR12, LSRS10] to be promising and useful in practice. TGGs have been researched for more than 15 years and there are currently multiple implementations, all being actively developed. Although a common theoretical foundation is shared, these implementations differ in expressiveness (what features are supported), applicability (what limitations are imposed), efficiency (strategies to ensure polynomial runtime), and the underlying translation algorithm (choice and sequence of rule applications). As neither a quantitative nor a qualitative overview and comparison of TGG tools exists, it is quite difficult to understand their different strengths and weaknesses.

Our contribution in this paper is to provide (1) a set of criteria for comparing TGG tools, and (2), based on these criteria, a concrete *quantitative* and *qualitative* comparison of three TGG tools with fundamentally different approaches: MoTE,[1] the TGG Interpreter,[2] and eMoflon.[3] We also provide (3) a discussion of formal guarantees and restrictions, and (4) virtual machines via *Share* [VM11] with a complete installation of each implementation to complement this paper. Our work is a first step towards a benchmark for TGG tools that can be extended to test and evaluate other existing tools or new ones. Our results can be used as a guideline for choosing the appropriate TGG tool for a specific task.

Section 2 compares our contribution to existing surveys, while Sect. 3 introduces TGG fundamentals and gives an overview of a schematic TGG control algorithm. Our criteria for the comparison are discussed in Sect. 4 and used in Sects. 5–7 to present each implementation with a qualitative comparison. This comparison is summarized in Sect. 8. Section 9 complements this with runtime measurements, while areas of future work are discussed briefly in Sect. 10.

2 Related Work

The *feature-based survey* of model transformation approaches by Czarnecki and Helsen [CH06] regards directionality, incrementality, and the way transformation rules are specified. With a similar goal, Mens and van Gorp [MV06] propose a *taxonomy* of model transformations, but focus more on the transformation *scenario* than on the model transformation approach. Their classification includes not only the *number* of source/target models and different *kinds* of transformations, but also quality aspects of the language/tool, like *usability* or *performance*. Another survey by Stevens [Ste08] focuses on *bidirectional* transformations and investigates more formal aspects of different approaches. In contrast to these broad surveys, we identify relevant criteria and conduct a quantitative and qualitative comparison of three *concrete TGG tools*.

The Model Transformations in Practice Workshop (MTIP 2005) [BRST06] sought to establish a benchmark for (bidirectional) model transformations with the *Class Diagram to Database* (CDDS) transformation as an example. The solutions submitted, however, often modified the example to fit the respective tool, leading to multiple variants and simplifications that do not allow an objective comparison of the tools. In particular, no *quantitative*, i.e., performance comparison was provided by the different solutions. Other papers [GH09, HEGO10] do present a performance evaluation, but only of a single tool in each case. In this paper, we formalize the CDDS example for TGG tools by providing not only metamodels *but also* a TGG to test and compare the TGG tools. Although the example had to be simplified so that *all* three tools could be used, we are able to provide, for the first time, a runtime comparison of three TGG tools.

We provide virtual machines via *Share* [VM11] with a complete installation for each tool: [Onl12]. The interested reader is welcome to try out the tools. Our single example is by no means a complete benchmark, but it is a first step in the right direction and must be extended in the future to cover further tools and, more importantly, a series of examples and different transformation scenarios such as in Varró et al. [VSV05] for graph pattern matching.

[1] www.mdelab.de/mote/
[2] www.cs.upb.de/index.php?id=tgg-interpreter
[3] www.emoflon.org

3 Foundations and Running Example

Our running example is inspired by Bézivin et al. [BRST06] and is a variant of the well-known bidirectional transformation from class diagrams to database schemata. Figure 1 depicts a concrete class diagram on the left, and the corresponding database schema on the right. A class diagram consists of *classes* such as `Bank`, `Client` and `Account`, and of *associations* between these classes such as `clients` between `Bank` and `Client`. Furthermore, each element in the class diagram has a unique ID: 01 for `Bank`, 02 for `clients`, etc. In the domain of database schemata, *tables* such as `Bank_01`, `Client_03` and `Account_05` have *columns*. In each table, exactly one column is designated as the *primary key* of the table (named "PK"), while all other columns (e.g., the column "clients::02" in `Bank_01`) are *foreign keys* that reference other tables (indicated with arrows). A class obviously corresponds to a table with a primary key, while associations correspond to foreign keys. Please note the correspondences between the names and IDs of classes and the IDs of tables (e.g., `Bank`, `01` and `Bank_01`), and the names and IDs of associations and the names of columns (e.g., `clients`, `02` and `clients::02`).

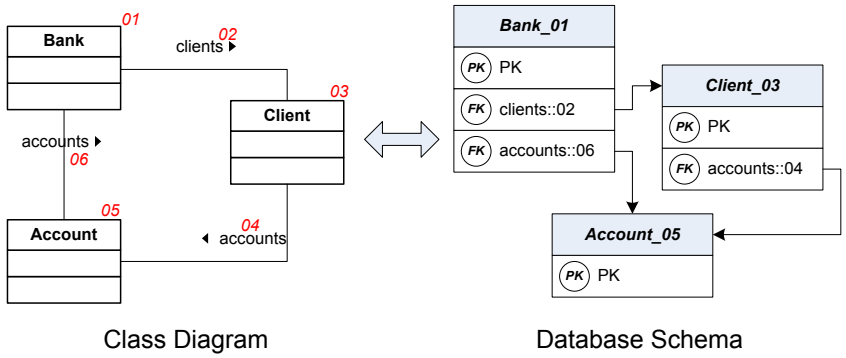


Figure 1: A class diagram and corresponding database schemata

Triple Graph Grammars (TGGs) provide a declarative, rule-based means for specifying bidirectional transformations. A TGG consists of a set of rules that describe how related models from a source domain (*class diagrams*) and from a target domain (*database schemata*) can be produced simultaneously. A third model (hence *triple* graph grammars) is created in the process and can be viewed as a set of traceability links between corresponding model elements from the source and target domains. A TGG can be used to generate a set or *language* of triples of source, correspondence and target models and can be viewed as a *consistency relation*; a source (M_S) and target (M_T) model are *consistent* with respect to a given TGG, if a triple of models M_S, M_C, M_T, denoted as $M_S \leftarrow M_C \rightarrow M_T$, can be generated with the TGG, where M_C is the correspondence model. Our goal in this paper is to impart a clear intuition for the core concepts of TGGs and we refer to [EEE+07, KW07, GHL10, KLKS10] for further details.

Fig. 2 depicts a TGG rule for our running example. A rule $r = (L, R)$ consists of a *precondition* L and a *postcondition* R, both *typed graphs* (structures consisting of nodes and links with types) representing model fragments that conform to the specified triple of metamodels. The rule can be interpreted as follows: if an occurrence or *match* can be found for the precondition L in a given

model M, then the rule r can be *applied* to the model M to yield a new model M' by replacing the determined match of L with R. The rule r, depicted in Fig. 2, requires an already related triple of class diagram, correspondence, and database schema (created using a different rule) and extends this triple by adding a new class to the class diagram and a corresponding new table with its primary key column. Elements in L and $R \setminus L$ are referred to as *context* and *created* elements, respectively. TGG rules have been extended formally to handle attributes in the related models [AVS12]. In Fig. 2, the *attribute formula* specifies that the table's `UID` must correspond to the concatenation of the `name`, an underscore, and the `id` of the corresponding class.

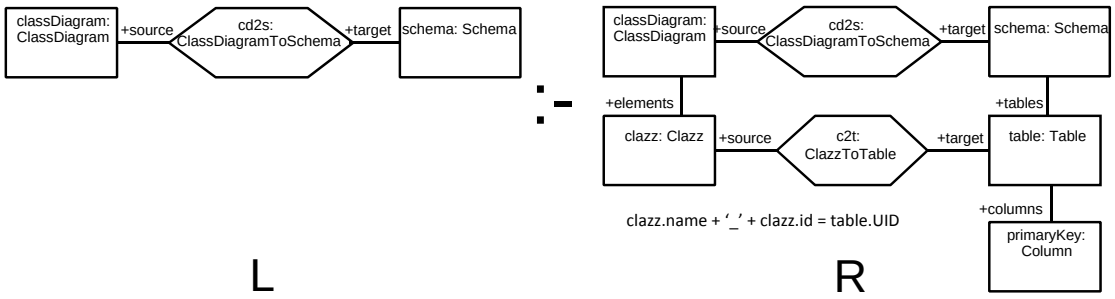


Figure 2: TGG rule *ClassToTable* in formal syntax

Although TGG rules can be used to produce source and target models simultaneously, the real potential of TGGs lies in the automatic derivation of *operational* forward and backward transformation rules. From the TGG rule r (Fig. 2), an operational *forward* rule r_f can be derived, which differs from r in that it *translates* all create elements in the source domain and only creates new elements in the correspondence and target domains. Deriving operational rules is straightforward: the precondition of a rule is extended to include all elements created in the source domain. To create valid triple models with these rules, however, it must be ensured for the existing source model that (i) context nodes in rules are only matched to translated host graph nodes, and (ii) created nodes in rules are only matched to host graph nodes that have not yet been translated. A *backward* rule is derived analogously. Attribute values are assigned based on attribute formulae, which are either automatically computed from the attribute formula or provided explicitly by the user. Although attributes can be regarded formally as separate nodes, most implementations treat attributes as primitives and only allow assignment of attributes for created nodes in rules, i.e., attributes of context nodes cannot be created or deleted.

Given a source model M_S, a *forward transformation* can be executed by applying forward rules derived from the TGG to translate M_S and yield a triple $M_S \leftarrow M_C \rightarrow M_T$, which could have been generated by using TGG rules. An important question is under which conditions this is always possible, and formal results from [EEE+07] prove existence and uniqueness provided that TGG rules are *non-deleting*, namely, $L \subseteq R$, which is the case for the three considered tools. The proof in [EEE+07] is however not constructive and the task of determining the *correct sequence* of forward rule applications remains a challenge and a point where the different TGG tools diverge. The derived operational rules can be used in various scenarios, e.g., to create a consistent triple of models $M_S \leftarrow M_C \rightarrow M_T$ from a given input model M_S (a *batch* transformation), or to

incrementally update a triple $M_S \leftarrow M_C \rightarrow M_T$ to result in a consistent triple $M'_S \leftarrow M'_C \rightarrow M'_T$ (an *incremental* transformation) given the changes that produced M'_S from M_S.

A TGG tool, therefore, requires a *control algorithm* to (i) determine a traversal order through the input model and (ii) choose the right rule to process each model element. To commence the transformation, a starting node in the input model and an operational rule to translate this node must be determined. Such a valid starting rule without any context elements, i.e., no precondition, is referred to as an *axiom*. After the first node is translated, a strategy is required to systematically cover all elements in the input model *in an appropriate order*. In general, *more than one rule* can be applicable for the current node to be translated and making the right choice requires backtracking to undo wrong decisions. This is, however, inefficient (exponential runtime) and most TGG tools restrict the class of supported TGGs to avoid backtracking.

4 Criteria for Comparison

Figure 3 depicts a schematic architecture of a TGG implementation. The *transformation designer* (Fig. 3::1, i.e., label 1 in Fig.3) specifies the bidirectional transformation as a TGG. The *end user* (Fig. 3::2) is mainly interested in the implemented transformation and uses this as a black box. In many scenarios, both users are actually the same person; the transformation designer, for instance, needs to test and refine the TGG, thus temporarily taking on the role of the end user.

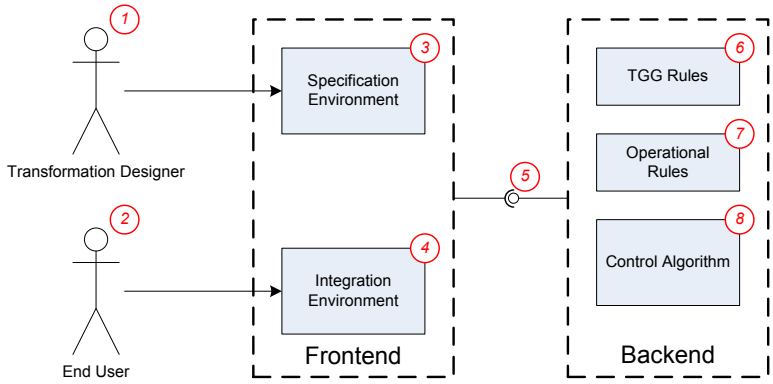


Figure 3: Schematic Architecture of a TGG Implementation

There is normally a clear separation between a *front end* and a *back end*. The front end can be divided into two parts: a *specification environment* (Fig. 3::3) used by the transformation designer to specify the involved metamodels and rules, and an *integration environment* (Fig. 3::4), with which the end user can run the transformation, view the results, and understand the transformation process. Depending on the concrete TGG tool, these might be clearly separated or form a single front end component. The front end and back end must be connected (Fig. 3::5) either via an import/export persistence format or an API. This again depends on whether the front end and back end are realized in completely separate tools or not.

The back end consists of data structures for the metamodels and the TGG rules (Fig. 3::6).

Depending on the concrete TGG tool, there might also be explicit data structures for the operational rules (Fig. 3::7) used by a control algorithm (Fig. 3::8). Operational rules are, however, optional as an interpreter might use the TGG rules directly.

We consider the following groups of criteria to be most important when comparing TGG implementations from a practical point of view:

Usability (U): The first hurdle when using a new tool is getting it *installed and properly configured (U1)*. This task should be supported with an installer, a sensible default configuration, tutorials, interactive help in the tool, and examples. The transformation designer requires a *suitable concrete syntax (U2)* (visual or textual) with which TGG rules can be specified. Equally important is a thorough *static analysis (U3)* and numerous sanity checks to identify modeling errors early in the process and offer possible fixes. Tool support should also be provided to increase *productivity (U4)*, such as refactorings, or automatic rule derivation from other rules.

At this point, a clear *workflow (U5)*, stating which steps have to be taken in what sequence, becomes crucial, especially if there is an explicit switch between specification and integration environments. Furthermore, the end user requires support to configure and *invoke the transformation (U6)* for an input model, and a means of *visualizing the resulting triple (U7)* in such a way that the translation process can be understood.

Expressiveness (E): A central question is if a chosen TGG tool is expressive enough to describe a required transformation. Here, we limit ourselves to important features of the considered tools, meaning that this list is to be extended when comparing other tools.

(Negative) application conditions (E1) restrict the applicability of a rule to certain cases. To translate attributes, the tools use different types of *attribute constraints (E2)*. Attribute constraints may be *unidirectional*, i.e., there must be two constraints (forward and backward). With *bidirectional* attribute constraints, there is only a single constraint that is used for both forward and backward transformations. Some tools allow the *transformation of single edges (E3)*, i.e., rules where only edges are created in a triple rule. This requires explicit bookkeeping of the transformed edges. Tools may impose restrictions on the *connectivity of the rules (E4)*, e.g., require them to be strongly connected or to be just weakly connected. Without any restrictions, rules may even consist of disjunct components.

TGG tools usually have to find a compromise between expressiveness on the one hand and efficiency and formal properties on the other. Therefore, each tool appropriately restricts the class of supported TGGs (cf. properties F1 and F2).

Formal Properties (F): Every TGG tool is expected to be *correct (F1)*, *complete with respect to the class of supported TGGs (F2)* and *efficient* for *batch (F3)* and *incremental (F4)* transformations. According to [KLKS10], these properties are defined as follows:

Correctness: Given an input model M_i, the resulting triple $M_i \leftarrow M_C \rightarrow M_o$ created by a forward/backward transformation must be a member of the language generated by the TGG.

Completeness: Every input model that can (in theory) be extended to a consistent triple must be extended to a consistent triple by the forward/backward transformation.

Efficiency: For a batch transformation, the transformation must have polynomial runtime complexity (n^k) w.r.t. the number of model elements (n) and the maximal number of elements (k) in a single rule of the TGG. In the incremental case, the runtime must scale w.r.t. the number of changes to be propagated incrementally and not with the size of the involved models.

Besides correctness, which ensures that results of a forward/backward transformation never contradict the given TGG, it is also useful to guarantee completeness at least for a certain class of TGGs. If a transformation fails with a complete tool, the user knows for sure that either (i) the input model is invalid, i.e., cannot be completed to a valid triple, or (ii) the given TGG is not in the class of TGGs supported by the tool, which must be clearly defined when proving completeness. In practice, to ensure *efficiency*, most TGG tools are correct and complete *only for a certain subset* of all possible TGGs, i.e., each implementation poses a *different* set of limitations on valid TGG rules. This is one of the factors that make it difficult to use different TGG tools and is to be addressed in this paper. Nevertheless, these properties can be guaranteed by formalizing the core algorithm of an approach and arguing that they hold at least at this level of abstraction. Tests and actual measurements are of course necessary to support such claims, and a series of benchmarks in a controlled testing framework are some of the steps taken in this paper to achieve this goal.

In the following, we present three TGG tools and enable a qualitative and quantitative comparison using the criteria defined in this section.

5 MoTE (Hasso-Plattner-Institute)

MoTE (Model Transformation Engine) is an EMF-based model transformation tool that supports bidirectional model transformation and synchronization (or incremental updates).

Usability: MoTE can be installed via the Eclipse Update Manager from the MDELab update site. An example transformation can be installed and user documentation is available via the Eclipse help system *(U1)*. A GMF-based editor is provided to specify TGG rules using the common TGG visual concrete syntax, where LHS and RHS are presented in a combined notation using colors to differentiate context (black) from created (green and marked with the $<<create>>$ stereotype) elements (Fig. 4) *(U2)*. In MoTE, TGG rules can have rule parameters, which specify values to be assigned to created elements when the TGG rules are applied directly to create both models as used in the automatic conformance testing framework [HLG+12]. The derived operational rules calculate the values of rule parameters using the provided *forward* and *backward* expressions (denoted with f and b). The editor also provides a comprehensive validation of TGG rules *(U3)*. A wizard is provided to create a new TGG rule project with an initial TGG axiom and a first incomplete TGG rule *(U4)*. After completing the specification, the transformation developer can derive operational rules by executing a designated *workflow*[4] file, which additionally creates configuration files and Java code to invoke the operational rules *(U5)*. Finally, the TGG rule project has to be deployed as an Eclipse plugin, which is discovered by MoTE via the extension mechanism in Eclipse. A model transformation or synchronization can be invoked either via its Java API, with the appropriate wizard from the *Model Transformations* menu, or as part of a workflow *(U6)*. In addition, a testing framework is provided [HLG+12], which can be used to generate random test models and test the TGG. This allows the transformation developer to validate whether his TGG behaves as expected.

[4] MDELab workflows (http://www.mdelab.de/workflow/) can be used to automate certain modeling tasks by plugging together workflow components, e.g., a component to read a model, a component to transform this model to a target model, and another component to save the target model to disk.

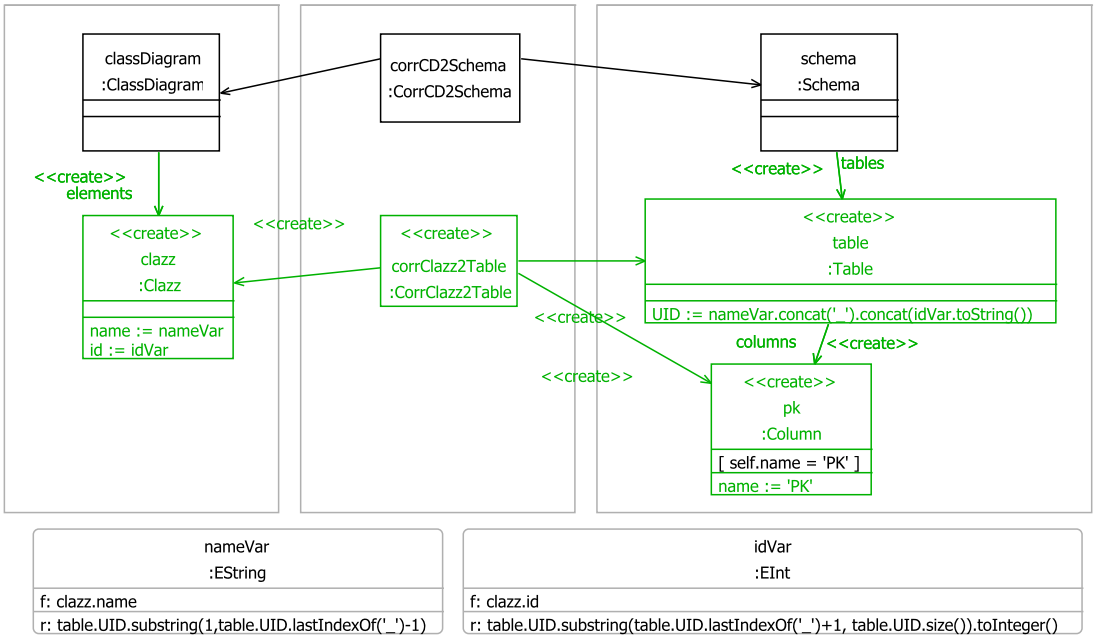


Figure 4: The Clazz2Table rule in MoTE's TGG editor

Expressiveness: MoTE has been used successfully in an industrial case study [GNH10] to integrate a SysML and an AUTOSAR modeling tool. TGG rules are compiled to *Story Diagrams*,[5] which are *interpreted* by a *Story Diagram Interpreter* [GHS09]. MoTE is EMF-based and supports constraints and attribute assignments in OCL or Java. Attribute assignments, however, have to be specified for each direction separately *(E2)*. Currently, MoTE does not support (negative) application conditions *(E1)* and rules that create edges between context nodes *(E3)*. Furthermore, some restrictions are imposed on the structure of valid TGG rules to be able to formalize the transformation algorithm and ensure comparably high transformation efficiency *(E4)*.

Formal Properties: The batch transformation algorithm in MoTE is formalized in [GHL10], defining the exact restrictions imposed on valid TGG rules and proving correctness *(F1)*, and completeness *(F2)* for TGGs that exhibit *functional behavior* [HEGO10]. Functional behavior means that the result of a transformation is unique up to isomorphism and can be viewed, depending on the application scenario, either as a limitation, or as a useful property. A TGG has functional behavior if it is terminating, which is ensured by checking the structure of TGG rules, and if its rules do not conflict, which can be checked via a *critical pair analysis*. However, this is not yet supported in the current MoTE release. MoTE supports batch *(F3)* and incremental *(F4)* transformations, which have been shown to be efficient (see Sect. 9 and [GH09]).

Regarding the underlying control algorithm; MoTE requires a designated starting node (typically the root node of an EMF model) and only allows exactly one axiom in the TGG. To drive the transformation process, MoTE takes the following approach: Each rule creates exactly one

[5] Programmed graph transformations for specifying unidirectional model transformation.

correspondence node and requires a set of correspondence nodes as context. The derived operational rules create a link between context and the created correspondence node. This results in a directed acyclic correspondence graph, which also represents dependencies between applied rules. After the axiom is applied to create a single correspondence node c, all rules that require c as context are applied with c as the entry point for pattern matching to translate and create further elements in the input and output models respectively. Each rule that transforms an element also creates a new correspondence node and the process is repeated until it terminates (a correspondence node is created which is not required by any other rule). This allows for efficient pattern matching and also reduces the number of rules that must be checked for applicability in each step. MoTE does not support backtracking, which is one reason why a TGG must have functional behavior.

Current and Future Focus: Due to the widespread use of model transformations in MDE, the *quality* of model transformations has to be ensured. *Conformance testing* of model transformations is, therefore, still an important and open issue. Current development focusses on improving and extending the automatic conformance testing framework presented in [HLG+12], and generalizing it to support and test transformations specified with other transformation languages, not only TGGs.

6 The TGG Interpreter (University of Paderborn)

The TGG Interpreter is a TGG model transformation and incremental update tool that was developed as a result of comparing TGGs and the OMG standard for (bidirectional) model transformations, *Query/View/Transformation* QVT [GK07, GK10].

Usability: The TGG Interpreter is Eclipse-based and can be installed via the Eclipse Update Manager *(U1)*. Figure 5 depicts a screenshot of the TGG rule editor showing a visual concrete syntax similar to all other tools *(U2)*.

Several sanity checks (e.g., node/edge type conformance, rule inheritance validity, OCL syntax checks, etc.) are performed statically to prevent modeling errors *(U3)*. However, there is no support for critical-pair analysis in order to identify possibly conflicting rules. A GMF editor is provided for editing TGG rules along with convenience functionality, e.g., creating new rules based on patterns in other rules, and creating correspondence types on-the-fly for correspondence nodes *(U4)*. The TGG Interpreter directly executes TGG specifications without further processing *(U5)*. It integrates itself into the Eclipse GUI, allowing performing transformations by right-clicking on a model file or via an Eclipse Run Configuration *(U6)*. Transformations can also be executed programmatically via an API call.

An additional plugin, the *Correspondence View*, can be used to visualize the results of a transformation. It shows the concrete syntax of both models side-by-side, allowing the user to visualize corresponding elements in the models. A transformation designer can use the debug mode (Fig. 5) to inspect the transformation and pattern matching process *(U7)*.

Expressiveness: Application conditions *(E1)* and attribute constraints *(E2)* can be formulated in OCL, but bidirectional relations on attribute values have to be expressed as assignments in the forward and backward direction. No strong restrictions are imposed on the structure of TGG rules, i.e., patterns need only be weakly connected *(E4)*, edges can be created between context

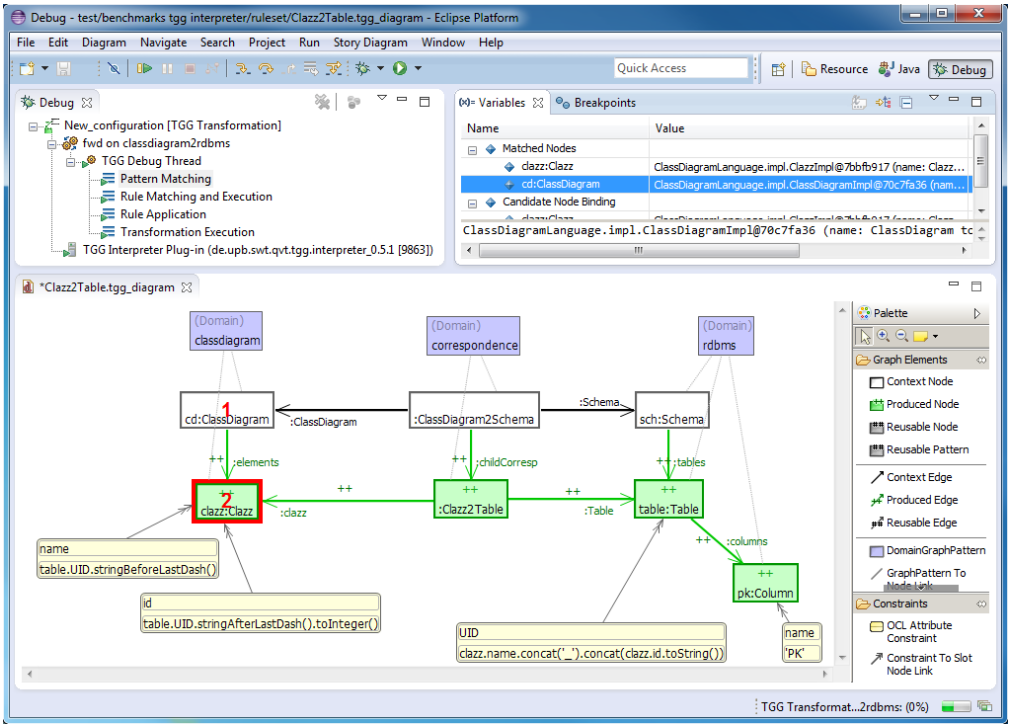


Figure 5: TGG Rule Editor of the TGG Interpreter during Debugging

nodes *(E3)*, and correspondence nodes can be connected to source and target elements as the user wishes. The TGG Interpreter supports advanced TGG concepts such as a form of rule inheritance, stereotypes in UML domains, and reusable nodes and patterns to retain information in the incremental case [GR12].

Formal Properties: The TGG Interpreter is correct (validated but not formally verified) *(F1)*, but not complete *(F2)* in general, due to a lack of backtracking/look-ahead functionality. Thus, similar to MoTE, it may fail to find a correct sequence of rule applications to match a given source model when more than one rule is applicable. A static analysis to check these conditions is currently not provided. As the TGG Interpreter directly interprets the TGG and, moreover, supports advanced features such as explicit edge bindings, it is outperformed by both other tools (cf. Sect. 9), but is still efficient enough for most practical cases *(F3)*. Furthermore, as the TGG is interpreted to perform the transformation, it can be updated on-the-fly during a transformation without needing to recompile or compute anything. Like MoTE, the TGG Interpreter also supports incremental updates *(F4)*. Regarding the control algorithm; the TGG Interpreter requires a designated starting node, only allows exactly one axiom in the TGG, and requires functional behavior to avoid backtracking. The input model is traversed by driving a *front* consisting of the current set of translated nodes, which is extended by examining all elements that can be reached from the front via a single edge. Only these front-extension elements are considered in the next step limiting the search for applicable rules to only those that require a front element and translate a front-extension element. The process terminates when the front can no longer be extended.

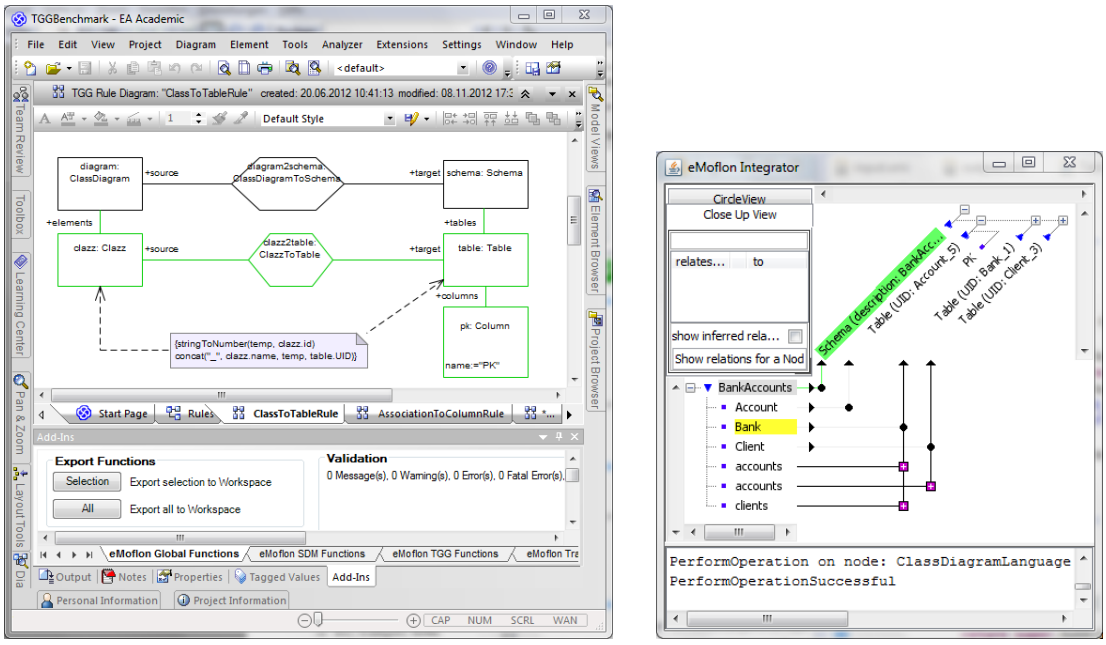
Current and Future Focus: Current development on the TGG Interpreter is in the direction of (i) support for improved algorithms for incremental updates [GPR11] and conflict resolution, (ii) a static analysis to identify conflicting rules *(F2)*, and (iii) support for non-functional transformations [RS12]. Furthermore, performance optimization is planned as future work.

7 eMoflon (Technische Universität Darmstadt)

eMoflon [ALPS11] is a *tool suite* for metamodeling and model transformation. Hence, not only bidirectional model transformation with TGGs is supported, but also unidirectional model transformation with Story Diagrams, and metamodeling with Ecore/EMF. eMoflon aspires to provide a complete environment for all required activities (cf. the online demo via Share [Onl12]).

Usability: A detailed tutorial (available at www.emoflon.org) describes the few steps required to get eMoflon up and running. The specification and integration environment are provided as plugins of *separate* tools: an extension for Enterprise Architect (EA) and an Eclipse plugin. EA is a professional UML tool and has proven to be advantageous for visual modeling as opposed to, e.g., GMF [ALPS11].

Installing eMoflon (U1) only requires clicking through a Windows Installer for the EA extension, and installing an Eclipse plugin. Figure 6(a) depicts the *Clazz2Table* TGG rule in EA. The eMoflon *concrete syntax (U2)* is visual and uses a compact (merged) representation. Currently, the eMoflon frontend only runs on Windows, but an alternative specification environment in Eclipse with a *textual* concrete syntax is under development.



(a) eMoflon Specification Environment

(b) eMoflon Integration Environment

Figure 6: TGG rule specification in EA (left) and runtime integration environment (right)

Specifying TGGs in EA (Fig. 6(a)) is supported by a substantial *static analysis (U3)*. Editor features support *productivity (U4)*, e.g., via the automatic derivation of TGG rules from other rules or from a choice of metamodel elements. The *workflow (U5)* is completed by exporting the specification from EA (simple context menu) and switching to the corresponding Eclipse workspace. The generated transformation code can be *invoked (U6)* by executing an automatically generated Java main method. The resulting triple created by the transformation can be *visualized (U7)* and stepped through using an *integrator* (Fig. 6(b)). This GUI component visualizes a triple of models in a matrix, i.e., the correspondence graph is visualized as links between two trees. A *protocol* containing a trace of the whole transformation process can be used to step through the transformation and understand which rules were applied when and why to which elements (colors indicate different states of elements during the transformation). As regions can be collapsed in the integrator (Fig. 6(b)) and programmable *breakpoints* can be provided to navigate directly to interesting situations, models with up to about 800 - 1000 nodes can be analyzed successfully, depending of course on how interconnected the nodes are.

Expressiveness: eMoflon uses *CodeGen2* from the Fujaba tool suite as its underlying graph pattern matcher. TGGs are compiled to Story Diagrams[6] with Story Diagrams, which are then compiled with CodeGen2 to plain EMF code. In addition to basic graph patterns, *negative application conditions* [AST12] *(E1)* and flexible, bidirectional attribute manipulation [AVS12] (E2) are supported in TGG rules. The latter is achieved via *in node* attribute assignments, such as for `pk:Column` (cf. Fig. 6(a)), and via a bidirectional and extensible constraint language, such as for concatenating class names and table IDs. Additionally, TGG rules are allowed to only transform edges (E3). Regarding the connectivity of the rules (E4), the underlying graph pattern matcher requires to have weakly connected patterns in the source and target domain of a rule.

Formal Properties: The eMoflon TGG tool is based on the algorithm of [KLKS10], which has been shown to be *correct (F1)*, *complete* for a certain class of TGGs *(F2)*, and *efficient (F3)*. eMoflon is not yet *incremental (F4)* but the incremental TGG control algorithm presented in [LAVS12] is currently being implemented.

The control algorithm of eMoflon can start with any node in the input model and handle arbitrary many axioms. TGGs are *not* required to be *functional* and a *look-ahead* is used to resolve local rule choices by inspecting edges that can no longer be translated if a wrong choice is made (*Dangling Edge Check* (DEC)). For efficiency, eMoflon only supports a look-ahead of *one edge* (DEC 1) from the current node to be translated, i.e., the class of supported TGGs is limited to *local complete* TGGs where DEC 1 is sufficient to resolve conflicts. In cases where multiple choices are correct, i.e., there is a true degree of freedom in the translation, eMoflon asks a component (the user, a configuration file, algorithm, etc.) to decide. Note that the set of local complete TGGs subsumes the set of functional TGGs, which only require "DEC 0".

Regarding the traversal order; this order is determined on-the-fly in a *context-driven* manner. All context nodes required by each potential rule are *recursively translated*, which automatically induces a feasible order in which all elements of the input model can be translated successfully. Please note that this recursive, eager and context-driven transformation strategy works for the supported class of TGGs, which is precisely defined in [KLKS10] with a description of all runtime exceptions that are thrown when a required restriction is violated.

[6] Story Diagrams are a combination of UML Activity Diagrams and graph transformations.

Current and Future Focus: Current and future development on eMoflon is in the direction of (i) implementing the incremental algorithm presented in [LAVS12] and investigating/evaluating other strategies for supporting incremental transformations with TGGs, (ii) providing a rich static analysis based on first ideas presented in [AST12], (iii) investigating and implementing advanced modularity concepts for TGGs, (iv) providing a textual concrete syntax for TGGs (and eMoflon in general), and (v) extending the general framework for bidirectional model-to-text transformations with TGGs presented in [ASRS12].

8 Summary of Qualitative Comparison

Table 1 summarizes the comparison based on criteria from Sect. 4 (● denotes "sufficient/good", ◐ "can be improved", and ○ "missing/inadequate"). In general, there is no "single best" tool; all three tools have their own strengths and weaknesses.

	U1	U2	U3	U4	U5	U6	U7	E1	E2	E3	E4	F1	F2	F3	F4
MoTE	●	◐	◐	◐	●	●	○	◐	◐	○	◐	●	●	◐	●
Interp.	●	◐	◐	●	●	●	◐	◐	◐	●	●	●	◐	◐	●
eMoflon	◐	◐	◐	●	◐	●	◐	●	●	●	◐	●	◐	●	○

Table 1: Summary of the Qualitative Comparison of the three TGG Implementations

Some observations are: eMoflon's installation process is a bit complex as two separate tools are used (U1), an alternative textual concrete syntax for TGGs is currently missing in all tools (U2), all tools should be improved regarding static analyses (U3), and MoTE can be improved regarding the visualization of results and the transformation process (U7).

MoTE has the strongest restrictions on TGG rules (E1 to E4), the TGG Interpreter allows the most flexible patterns (E4), and only eMoflon supports true bidirectional attribute constraints (E2). Although bidirectional constraints guarantee that a TGG works bidirectionally, note that unidirectional constraints are more expressive in some cases.

All tools are correct (F1), but a formal proof for completeness (F2) only exists for MoTE's and eMoflon's transformation algorithm. eMoflon supports a larger class of TGGs than both MoTE and the TGG Interpreter, is fastest (F3) on the example scenario (cf. Sect. 9), but does not yet support incremental transformations (F4).

9 Quantitative Comparison (Runtime Measurements)

To allow a first estimation of the batch transformation performance of the tools, we conducted a benchmark, in which each tool was used to translate the same set of models using the transformation presented in Sect. 3. This quantitative assessment was not conducted for incremental transformations, as this is not yet fully supported by all tools (cf. Sect. 8). Using the model generator of the TGG test framework [HLG+12] (part of MoTE), random test models were created. In total, 19 random class diagram test models were generated with 100, 200, ..., 1000 elements and 2000, 3000, ..., 10000 elements. The average execution times were calculated after each

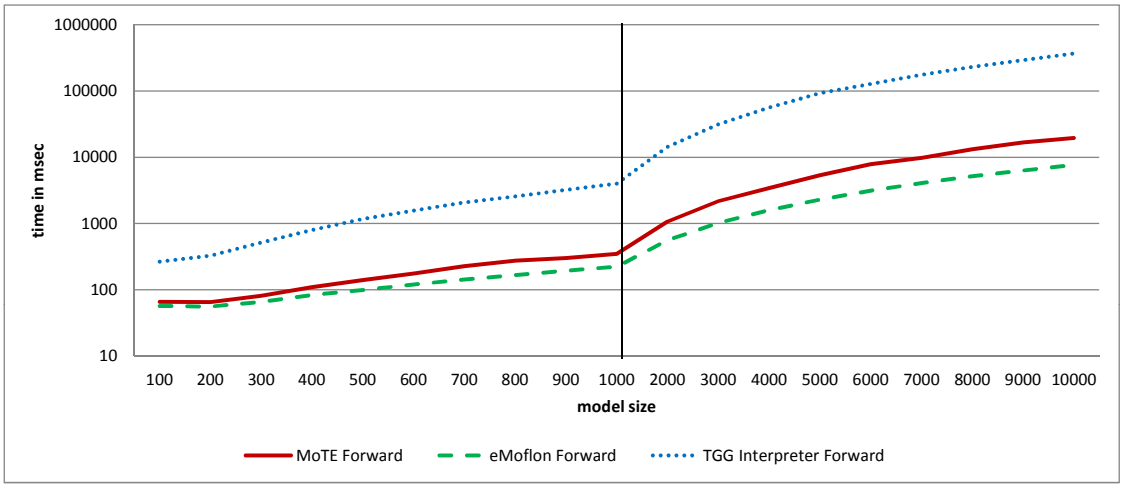


Figure 7: Runtime Measurement Results for Forward Transformation

TGG tool transformed each class diagram 20 times. The measurements were conducted on a Windows 7 x64 PC with an Intel i5 750 (2.66 GHz) processor, Oracle JDK 1.7, and Eclipse 4.2. The class diagram and database schema metamodels and all test models are available for download.[7]

The average execution times for all tools and model sizes for the forward and the backward transformation are depicted in Fig. 7 and 8. Note that a logarithmic scale is used for the Y-axis. A vertical line indicates where the X-axis scaling changes.

For the forward transformation, the results show that eMoflon (green solid curve), the only tool that generates Java code from TGG rules, is faster than the TGG Interpreter (red dotted curve), which interprets TGG rules directly, and MoTE (blue dashed curve), which interprets operational rules as Story Diagrams. The difference between eMoflon and MoTE (8s/20s for 10000 elements) is, however, considerably smaller than that between eMoflon and the TGG Interpreter (8s/367s for 10000 elements). This is probably due to the dynamic pattern matching strategy of the story diagram interpreter [GHS09], and MoTE's algorithm (see Sect. 5), in which the application of the next TGG rule is guided by dependencies in the correspondence model; both perform well in this scenario.

The TGG Interpreter is faster when transforming backwards than when transforming forward in this scenario. MoTE is also faster for models with more than 700 elements. Interestingly, eMoflon is slower in the backward direction than in the forward direction for models with more than 300 elements, infact, MoTE outperforms eMoflon when transforming backwards for model with more than 2000 elements.

The different results for forward and backward transformations indicate that the matching and transformation strategies implemented in the tools are quite different and have a significant impact on the overall performance. To make any general conclusions, however, other transformation scenarios have to be considered. This is planned as future work.

[7] http://emoflon.org/fileadmin/download/eMoflon/bx13_TGG_Survey.zip

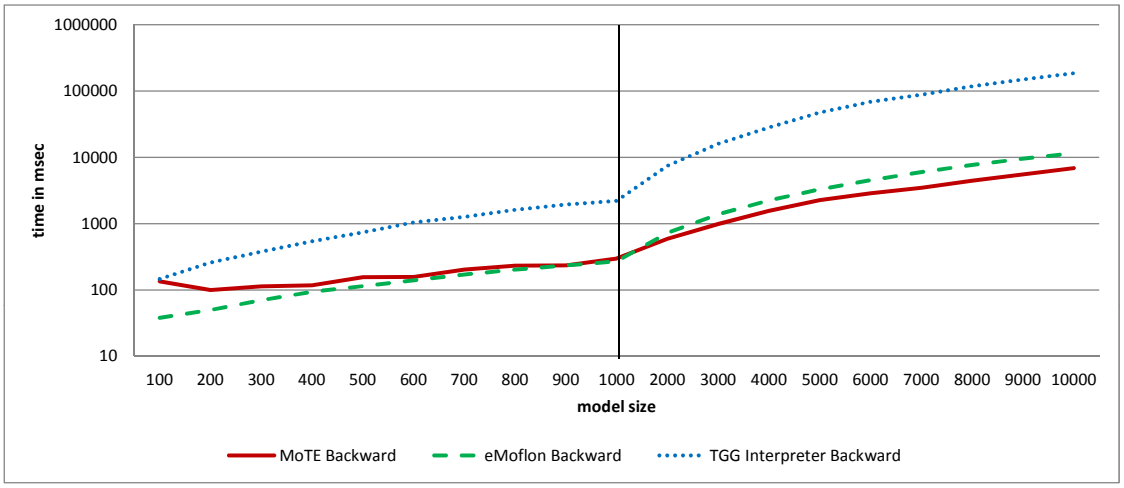


Figure 8: Runtime Measurement Results for Backward Transformation

A polynomial runtime (in the size of the input model) is formally proven for eMoflon [KLKS10], but not for MoTE and the TGG Interpreter. Nonetheless, from our experience, the runtime of the two latter tools is also polynomial in the size of the input model.

10 Conclusion and Future Work

In this paper, we provide a set of criteria to be used for comparing TGG tools. Based on these criteria and the well-known class diagrams to database schemata transformation, we presented a quantitative and qualitative comparison of three actively developed tools: MoTE, the TGG Interpreter, and eMoflon.

Our results show that the tools vary considerably and have different strengths and weaknesses, depending on the application scenario and corresponding requirements: A purely interpretative approach (the TGG Interpreter) is probably the best choice if metamodels are to be used *without* demanding generated code, eMoflon is a viable choice if user interaction is to be integrated in the transformation (i.e., the TGG is non-functional), and MoTE/eMoflon can be used for large models and in cases where efficiency is paramount. MoTE/the TGG Interpreter are currently the best choice for incremental updates, and, finally, the TGG Interpreter and eMoflon support an interesting set of advanced TGG features that might be necessary if high expressiveness is required.

As future work, we plan to extend the comparison to include further TGG tools and cover a whole suite of transformations that test different aspects including various kinds of *incremental* updates. The vision is to establish a benchmark for TGG tools, which can be used to drive and measure improvements in the future.

Bibliography

[ALPS11] A. Anjorin, M. Lauder, S. Patzina, A. Schürr. eMoflon : Leveraging EMF and Professional CASE Tools. In *MEMWe '11*. LNI 192, p. 281. GI, 2011.

[ASRS12] A. Anjorin, K. Saller, S. Rose, A. Schürr. A Framework for Bidirectional Model-to-Platform Transformations. In *Proc. of SLE 2012*. LNCS. Springer Berlin / Heidelberg, 2012.

[AST12] A. Anjorin, A. Schürr, G. Taentzer. Construction of Integrity Preserving Triple Graph Grammars. In Ehrig et al. (eds.), *Proc. of ICGT '12*. LNCS 7562, pp. 356–370. Springer, 2012.

[AVS12] A. Anjorin, G. Varró, A. Schürr. Complex Attribute Manipulation in TGGs With Constraint-Based Programming Techniques. In Hermann and Voigtländer (eds.), *Proc. of bx '12*. EC-EASST 49, pp. 1–16. EASST, 2012.

[BRST06] J. Bézivin, B. Rumpe, A. Schürr, L. Tratt. Model Transformation in Practice. In Bruel (ed.), *MoDELS 2005 Workshops*. LNCS 3844, pp. 120–127. Springer, 2006.

[CH06] K. Czarnecki, S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal* 45(3):621–645, 2006.

[EEE+07] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, G. Taentzer. Information Preserving Bidirectional Model Transformations. In Dwyer and Lopes (eds.), *Proc. of FASE '07*. LNCS 4422, pp. 72–86. Springer, 2007.

[GH09] H. Giese, S. Hildebrandt. Efficient Model Synchronization of Large-Scale Models. Technical report 28, Universitätsverlag Potsdam, 2009.

[GHL10] H. Giese, S. Hildebrandt, L. Lambers. Toward Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars. In *Proc. of MoDeVVa '10*. Pp. 19–24. IEEE, 2010.

[GHS09] H. Giese, S. Hildebrandt, A. Seibel. Improved Flexibility and Scalability by Interpreting Story Diagrams. In Margaria et al. (eds.), *Proc. of GT-VMT '09*. EC-EASST 18. EASST, 2009.

[GK07] J. Greenyer, E. Kindler. Reconciling TGGs with QVT. In Engels et al. (eds.), *Proc. of MoDELS '07*. LNCS 4735, pp. 16–30. Springer, 2007.

[GK10] J. Greenyer, E. Kindler. Comparing Relational Model Transformation Technologies: Implementing Query/View/Transformation with Triple Graph Grammars. *SoSyM* 9(1):21–46, 2010.

[GNH10] H. Giese, S. Neumann, S. Hildebrandt. Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. In Engels et al. (eds.), *Festschrift Nagl*. LNCS 5765, pp. 555–579. Springer, 2010.

[GPR11] J. Greenyer, S. Pook, J. Rieke. Preventing Information Loss in Incremental Model Synchronization by Reusing Elements. In France et al. (eds.), *Proc. of ECMFA '11*. LNCS 6698, pp. 144–159. Springer, 2011.

[GR12] J. Greenyer, J. Rieke. Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata. In Schürr et al. (eds.), *Proc. of AGTIVE 2011*. LNCS 7233, pp. 222–237. Springer, 2012.

[HEGO10] F. Hermann, H. Ehrig, U. Golas, F. Orejas. Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In Bézivin et al. (eds.), *Proc. of MDI '10*. Pp. 22–31. ACM, 2010.

[HLG+12] S. Hildebrandt, L. Lambers, H. Giese, D. Petrick, I. Richter. Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations. In Schürr et al. (eds.), *Proc. of AGTIVE '11*. Volume 7233, pp. 238–253. Springer, 2012.

[KLKS10] F. Klar, M. Lauder, A. Königs, A. Schürr. Extended Triple Graph Grammars With Efficient and Compatible Graph Translators. In Engels et al. (eds.), *Festschrift Nagl*. LNCS 5765, pp. 141–174. Springer, 2010.

[KW07] E. Kindler, R. Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical report tr-ri-07-284, Department of Computer Science, University of Paderborn, 2007.

[LAVS12] M. Lauder, A. Anjorin, G. Varró, A. Schürr. Efficient Model Synchronization With Precedence Triple Graph Grammars. In Ehrig et al. (eds.), *Proc. of ICGT '12*. LNCS 7562, pp. 401–415. Springer, 2012.

[LSRS10] M. Lauder, M. Schlereth, S. Rose, A. Schürr. Model-Driven Systems Engineering: State-of-the-Art and Research Challenges. *Bulletin of the Polish Academy of Sciences, Technical Sciences* 58(3):409–422, 2010.

[MV06] T. Mens, P. Van Gorp. A Taxonomy of Model Transformation. *ENTCS* 152:125–142, 2006.

[Onl12] Online demos for all tools as SHARE images. http://www.moflon.org/emoflon/a-survey-of-tgg-tools/. 2012.

[RS12] J. Rieke, O. Sudmann. Specifying Refinement Relations in Vertical Model Transformations. In *Proc. of ECMFA 2012*. Pp. 210–225. Springer Berlin/Heidelberg, 2012.

[Ste08] P. Stevens. A Landscape of Bidirectional Model Transformations. In Lämmel et al. (eds.), *Proc. of GTTSE '07*. LNCS 5235, pp. 408–424. Springer, 2008.

[VM11] P. Van Gorp, S. Mazanek. SHARE: A Web Portal for Creating and Sharing Executable Research Papers. *Proc. of ICCS '11* 4:589–597, 2011.

[VSV05] G. Varró, A. Schürr, D. Varró. Benchmarking for Graph Transformation. In *Proc. of VLHCC '05*. Pp. 79–88. IEEE, 2005.