Electronic Communications of the EASST
Volume 67 (2014)

Proceedings of the
13th International Workshop on Graph Transformation
and Visual Modeling Techniques
(GTVMT 2014)

A Comparison of Incremental Triple Graph Grammar Tools

Erhan Leblebici, Anthony Anjorin, Andy Schürr,
Stephan Hildebrandt, Jan Rieke, and Joel Greenyer

14 pages

Guest Editors: Frank Hermann, Stefan Sauer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Comparison of Incremental Triple Graph Grammar Tools

Erhan Leblebici[1*], Anthony Anjorin[1†], Andy Schürr[2], Stephan Hildebrandt[3], Jan Rieke[4], and Joel Greenyer[5]

[1]Graduate School of Computational Engineering, Technische Universität Darmstadt, {leblebici, anjorin}@gsc.tu-darmstadt.de
[2]Real-Time Systems Lab., Technische Universität Darmstadt, andy.schuerr@es.tu-darmstadt.de
[3]Hasso-Plattner-Institut, Universität Potsdam, stephan.hildebrandt@hpi.uni-potsdam.de
[4]Heinz Nixdorf Institut, Universität Paderborn, jrieke@uni-paderborn.de
[5]Software Engineering Group, Leibniz Universität Hannover, greenyer@inf.uni-hannover.de

Abstract: *Triple Graph Grammars* (TGGs) are a graph-based and visual technique for specifying bidirectional model transformation. TGGs can be used to transform models from scratch (in the *batch* mode), but the real potential of TGGs lies in propagating updates *incrementally*. Existing TGG tools differ considerably in their incremental mode concerning underlying algorithms, user-oriented aspects, incremental update capabilities, and formal properties. Indeed, the different foci, strengths, and weaknesses of current TGG tools in the incremental mode are difficult to discern, especially for non-developers. In this paper, we close this gap by (i) identifying a set of criteria for a *qualitative* comparison of TGG tools in the incremental mode, (ii) comparing three prominent incremental TGG tools with regard to these criteria, and (iii) conducting a *quantitative* comparison by means of runtime measurements.

Keywords: Triple Graph Grammars, incremental update, comparison

1 Introduction and Motivation

Bidirectional model transformation is a viable means of ensuring consistency in a Model-Driven Engineering (MDE) context and Triple Graph Grammars (TGGs) [Sch95] have been shown to be a promising, graph-based, and visual bidirectional language with solid formal foundations. A TGG specifies a consistency relation between source, target, and correspondence models in a declarative and rule-based manner. Using a TGG specification, operational scenarios such as a forward/backward transformation can be supported, e.g., to transform a source/target model to a target/source model, which can either be created from scratch (referred to as batch mode) or, more importantly, be *incrementally updated* to restore consistency (incremental mode).

There are currently multiple TGG tools that support incremental updates. The incremental mode of these tools differ in *user-oriented aspects*, *update capabilities*, and the *formal properties* of the underlying incremental algorithm. Existing publications on TGG approaches, however, either explicitly exclude the incremental case [HLG+13], or only focus on a single TGG tool

* Supported by the ‘Excellence Initiative’ of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt.
† The project on which this paper is based was funded by the German Federal Ministry of Education and Research, funding code 01IS12054. The authors are responsible for all contents.

[GH09, LAVS12]. It is thus a challenging task for users (non-developers) to understand the different foci, strengths, and weaknesses of the TGG tools that support incremental updates.

In this paper, we complement the existing survey of TGG tools [HLG$^+$13], which only handled the batch mode, by: (i) presenting a set of criteria for comparing TGG tools in incremental mode and (ii) extending the qualitative and quantitative comparison of the three TGG tools *MoTE*, *eMoflon*, and the *TGG Interpreter* based on these criteria. In contrast to the existing batch comparison, our criteria are focused strictly on the incremental case, which is the real strength of TGGs and truly exploits the explicit support for traceability links. A discussion of formal properties and restrictions regarding incremental updates is also provided, as each incremental TGG tool focuses on different aspects. Our results, therefore, serve as a guideline for choosing the appropriate TGG tool for a specific update scenario.

To the best of our knowledge, the chosen tools for the comparison were the only incremental TGG tools at the time when this paper was written. We encourage TGG researchers to extend our comparison with new incremental TGG tools in the future. This work is a further step towards a benchmark for testing and evaluating existing and new (incremental) TGG tools.

The paper is structured as follows: Section 2 compares our contribution to existing surveys, while Sect. 3 introduces TGG fundamentals and our running example. Section 4 provides our criteria for a qualitative assessment of incremental TGG tools. Subsequently, Sect. 5 presents three incremental TGG tools and compares them qualitatively. Section 6 shows runtime measurements for a quantitative comparison, while Sect. 7 concludes with a brief discussion of results.

2 Related Work

Czarnecki et al. [CH06] propose a feature model for model transformation handling aspects such as transformation rules, algorithms, directionality and, relevant for this paper, *incrementality*. Stevens [Ste08] gives a comprehensive overview of *bidirectional* transformation languages and gives a brief qualitative discussion. A comparative case study of model transformation by *graph transformation* is provided by Taentzer et al. [TEG$^+$05], comparing four different approaches including a TGG-based tool. Finally, Kusel et al. [KEK$^+$13] provide a survey of incremental approaches including, amongst others, two TGG representatives.

Of the surveys mentioned so far, [CH06] covers bidirectionality and incrementality briefly, but do not focus on these aspects. Although [Ste08, TEG$^+$05] identify TGGs as being suitable for incremental transformations, the main focus is not on incremental updates. While [KEK$^+$13] focuses on incrementality and distinguishes TGGs as being bidirectional, individual strengths and weaknesses of TGG *tools* are not handled due to the broad scope of the survey. Finally, [KEK$^+$13, Ste08, TEG$^+$05] discuss and compare different model transformation approaches *qualitatively*, but none of the surveys conducts a *quantitative* tool comparison.

Lauder et. al [LAVS12] present a TGG-based incremental algorithm, which is discussed and compared qualitatively to other approaches. User-oriented criteria are, however, not considered and a performance analysis of different tools is not given. Although a quantitative analysis *is* provided in [GH09], the results cannot be used directly for a comparison as only a single TGG-based tool is considered in each case. The survey of TGG tools in [HLG$^+$13] compares TGG tool qualitatively and quantitatively in the batch mode and excludes the incremental mode explicitly.

An extension of this survey to incremental TGG approaches is identified as future work. In this paper, we now fill this gap by handling the incremental case.

3 Foundations and Running Example

In this paper, we discuss incremental updates with different TGG tools using a consistency relation between *class diagrams* and *database schemata*. This application scenario, abbreviated as CDDS, is popular in the model transformation community [BRST06] and is used in [HLG+13] to compare TGG tools in batch mode.

TGGs [Sch95] describe how consistent pairs of source and target models are built up simultaneously together with a correspondence model between them. In an MDE context, all source and target models conform to a given source and target *metamodel*, respectively. Analogously, the correspondence model conforms to a correspondence metamodel which connects the source and target metamodel. The triple of metamodels is referred to as a *TGG schema*.

The upper part of Fig. 1 depicts the TGG schema for our *CDDS* example. The structure of class diagrams, consisting of classes and associations between them, is described by the source metamodel on the left. The source and target class of an association is represented by the *source* and *target* reference, respectively. Moreover, an inheritance relationship between two classes is represented by the *super* reference. The target metamodel on the right describes the structure of database schemata, consisting of tables and columns. A column can reference other columns to represent cross-references or self-references in database schemata. Note that class diagrams and database schemata in our example have a similar structure consisting of entities (classes and tables) and their features (associations and columns). An interesting difference, however, is the *super* reference between two classes that does not have any counterpart in a database schema. Finally, the correspondence metamodel in the middle defines three correspondence types *ClassDiagramToSchema*, *ClazzToTable*, and *AssociationToColumn*, which relate elements of a class diagram with those of a database schema in a straightforward manner.

Together with a TGG schema, a set of *TGG rules* describes how triples of source, correspondence, and target models, denoted as $M_S \leftarrow M_C \rightarrow M_T$, can be produced simultaneously. A rule $r = (L,R)$ consists of *context* elements representing the precondition L, and *created* elements representing the completion of the postcondition R. Based on a *pattern matching* process, a TGG rule is applied by extending an occurrence (*match*) of L to R in a model triple. Figure 1 depicts the six TGG rules required for our CDDS scenario. In concrete syntax, we use green and a ++ markup to distinguish created elements from black context elements.

Rule $r1$ does not require any precondition and creates a class diagram and a database schema as well as a correspondence between them. Rule $r2$ requires the created elements in $r1$ as precondition and creates a class and a related table with a primary key column. Rule $r3$ creates a subclass of an existing class and relates this subclass to the same table as its super class. Note that $r3$ does not create any tables but relates existing ones to new classes. As a consequence, all classes in the same inheritance hierarchy are related to the same table.

The remaining rules $r4-r6$ handle the creation of associations in class diagrams and columns in database schemata: Rule $r4$ creates an association between two classes that are related to different tables, and a column that represents the corresponding cross-reference between the

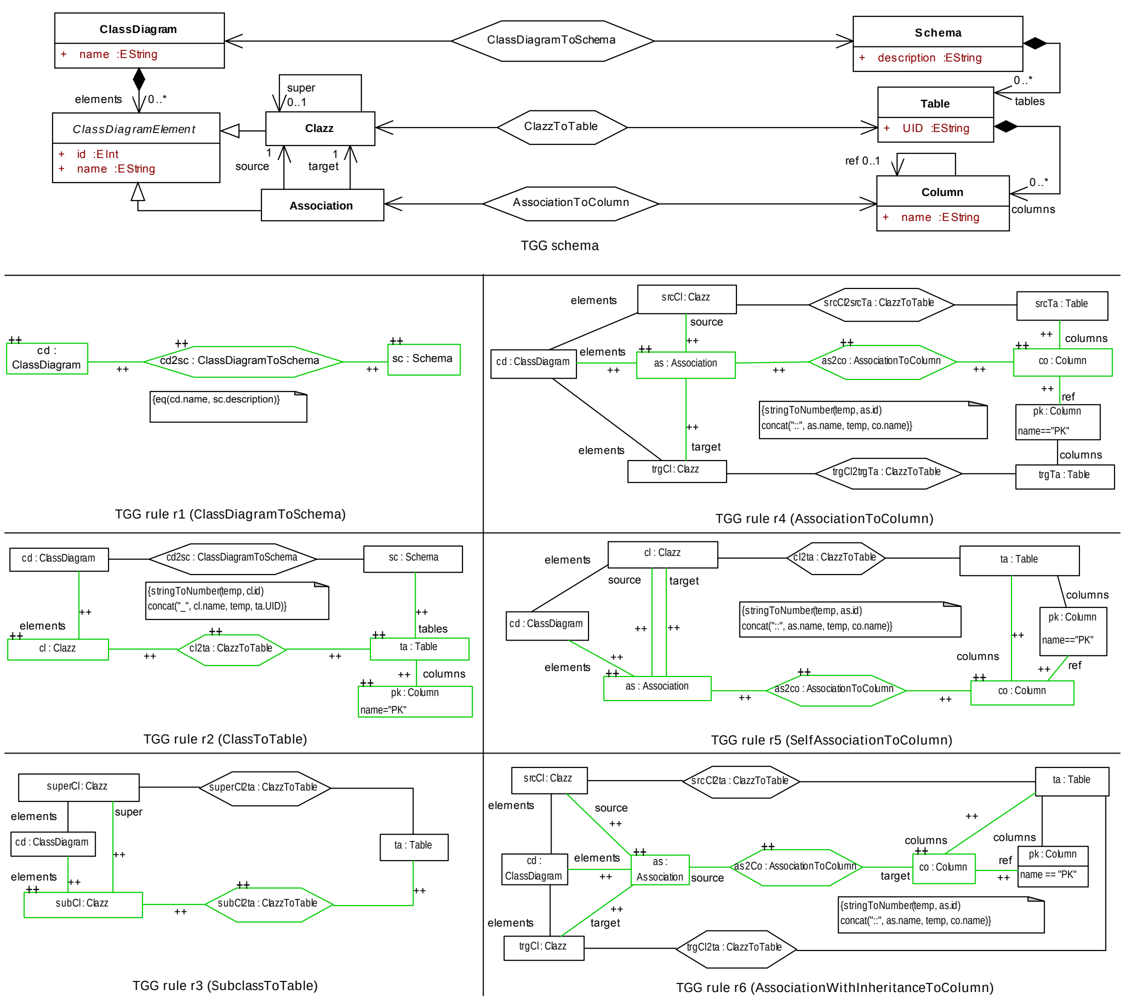

Figure 1: TGG schema and TGG rules for the running example

tables. An association from a class to itself and a self-reference column in the related table are created by rule $r5$. Finally, rule $r6$ creates an association between two classes that are related to the same table (due to inheritance), together with a self-reference column in the table. Note that the *attribute constraints* in each rule define attribute relationships between class diagram and database schema elements. In our simplified variant of *CDDS*, associations are restricted to having only a single target class (0/1 multiplicity), and only single inheritance is supported.

As TGG rules define the simultaneous production of model triples, *operational forward* and *backward rules* have to be derived to allow for forward and backward transformations. A forward rule does not create any elements in the source model, but *translates* existing ones (e.g., the forward rule of $r2$ translates a class by creating a new table with a primary key as well as a correspondence in between). This applies analogously to backward rules. An interesting backward rule in our running example results from $r3$, which does not create any target elements. In this case, the backward rule does not translate any elements and is ignored in a backward transformation as it would be unclear how often such rules should be applied. As a result, the backward transformation does not create multiple classes related to the same table. In other words, *super* references in a class diagram are not reproducible from the related database schema.

The operational rules are applied by a governing *control algorithm* to either translate an input model to a related output model from scratch, or to propagate modifications in one model to another already existing and related model. Our tool comparison focuses on the latter case: Given a consistent model triple $M_S \leftarrow M_C \rightarrow M_T$ and a model *delta* Δ_S that changes M_S to M'_S, an incremental update in the forward direction revokes (former) invalidated forward rule applications and performs new ones so that a consistent triple $M'_S \leftarrow M'_C \rightarrow M'_T$ is produced. Incremental updates in the backward direction operate analogously. We focus on updates only in one direction as *concurrent* changes in both domains are currently not supported by any incremental TGG tool.

4 Comparison Criteria

In the following, three groups of criteria are defined and used for the ensuing qualitative comparison of incremental updates with different TGG tools. Our comparison criteria, in contrast to those in existing surveys mentioned in Sect. 2, focus strictly on incremental updates. Although the discussion is provided from a TGG point of view, our criteria reflect the general requirements and challenges for incremental updates in MDE, regardless of the chosen approach.

4.1 User-oriented Aspects (U)

The first step when propagating changes in a model M'_S to a related model M'_T is *change detection (U1)*, i.e., determining Δ_S that changes M_S to M'_S. Model changes can be detected "online" meaning that any modification is recorded at runtime or "offline" via a model diff that compares two versions of a model and derives the modifications retrospectively. An incremental approach should support both change detection mechanisms so the user can make a case-specific decision.

Another major aspect is *plausibility (U2)* regarding the results of the incremental update. In addition to explicit correspondences between related models supported inherently by TGGs, an incremental tool should provide suitable means to understand the actions performed during the update. A debug mode and/or (persistent) information such as an update protocol might be imaginable to address this issue. Although this introduces additional components to be managed by the user, it is advantageous to analyse, if possible in advance, the effects of an update.

A tool might be forced to make certain decisions to resolve non-determinism or to apply heuristics during incremental updates. Considering incremental updates in the backward direction of our running example, a self-reference column of a table can be translated with either $r5$ or $r6$, giving rise to a degree of freedom: $r5$ would create an association from a related class to itself, whereas $r6$ would create an association between two classes related to this same table. In the latter case, a further decision point arises from determining the direction of the association, i.e., which class is the source (or target) of the created association. More interestingly, these decision points in the backward direction are only relevant in the incremental mode as only a previous forward transformation can lead to these circumstances (recall from Sect. 3 that a backward transformation does not create multiple classes related to the same table). Therefore, *user* *involvement (U3)* is a desirable feature to qualitatively improve the update result. The preferred choice during a non-deterministic update process can be determined by incorporating user interaction, configuration files, or similar guidance. Finally, user involvement should be avoided if it does not impact the result of the incremental update.

4.2 Incremental Update Capabilities (C)

The class of update scenarios that a TGG tool can handle is strongly affected by its *supported change types (C1)*. In an MDE context, four different types of change operations are distinguished: creation, deletion, movement, and attribute modification. A fully-fledged tool should be able to propagate all of these four change types. Nevertheless, although this is not ideal, a moved or modified element can be considered as deleted and re-created. Thus, a tool must at least support creations and deletions to fulfill the minimum requirements for incremental updates.

Finally, *backtracking or look-ahead capabilities (C2)* of the underlying algorithm are crucial in many incremental update scenarios. Some TGG tools are able to make a choice between more than one applicable rule to translate a model element. In general, this choice might lead to a wrong decision, meaning that a translation step results in untranslated elements at the end of the whole process. TGG tools that do not support backtracking, a look-ahead, or decision making, cannot cope with update scenarios in which model changes require previous rule applications to be revoked, even though the applied rule still matches. Such a case can be constructed using our running example: Assume two classes have been translated by $r2$ in a forward transformation. Now a new *super* reference is added, and is to be propagated incrementally, i.e., a class *becomes* the subclass of another. Although $r2$ still matches for both classes, the subclass must now be translated differently (with $r3$ instead) as the TGG specification relates all classes in an inheritance hierarchy to the same table. The application of $r2$ must be revoked, and backtracking or a look-ahead is required to ensure that $r3$, and not $r2$, is applied to re-translate the subclass. It is important to emphasize at this point that TGG tools usually make a compromise between backtracking/look-ahead capabilities, performance, and guaranteeing formal properties.

4.3 Formal Properties (F)

Given a consistent model triple $M_S \leftarrow M_C \rightarrow M_T$ and a model delta Δ_S that changes M_S to M'_S, there are certain properties every TGG tool should guarantee when repairing consistency by changing M_T to M'_T, i.e., computing a valid Δ_T in a forward update. The following definitions are formulated only for the forward direction for presentation purposes, but apply analogously for the backward direction

The three major properties for TGGs [KLKS10] are revised as follows for incremental updates:

Correctness (F1): The result of the incremental update, i.e., $M'_S \leftarrow M'_C \rightarrow M'_T$, must be consistent, i.e., a member of the language generated by the TGG.

Completeness (F2): If there exists a consistent triple $M'_S \leftarrow M'_C \rightarrow M'_T$, the update must find it.

Efficiency (F3): The execution time of the incremental update must be polynomial in the size of Δ_S and its affected elements, and not in the size of the model triple. In this context, the *affected* elements of Δ_S are all elements that must be either deleted or re-translated to restore consistency.

Finally, we introduce a new property to require a minimal update in the following sense:

Least change (F4): An incremental update must choose a Δ_T to restore consistency such that there is no subset of Δ_T that would also restore consistency, i.e., the computed Δ_T does not contain redundant modifications.

Least change is required to preserve information, which should not be affected by propagated changes. This property subsumes *hippocraticness*[1] [Ste10] but does not handle the case where totally different sets of modifications are possible. Formalizing a least change property for this more general case is ongoing research and is left to future work.

5 Qualitative Assessment

In the following, an introduction and qualitative assessment of three TGG tools are given in accordance to the identified comparison criteria. It should be mentioned at this point that all three TGG tools are based on the Eclipse Modeling Framework (EMF) enabling a direct comparison.

5.1 MoTE

Each TGG rule in MoTE (www.mdelab.de/mote) creates a single new correspondence, which depends on all context correspondences of the rule. All created elements in the rule are connected to this new correspondence. The incremental algorithm [GH09] of MoTE exploits the induced dependency tree of correspondences to determine elements affected by a modification. Correspondences connected to modified elements are sorted in a queue and processed by *synchronization operations*. A synchronization operation checks if the consistency of an affected correspondence can be re-established solely by updating attribute values. In case of success, dependent correspondence nodes are put into the queue as their consistency might now be affected by the attribute changes. Such a direct attribute propagation is not applicable if the modification deletes elements connected to the correspondence. In this case, *repair operations* are used to replace missing source elements by newly created ones, adjusting correspondence and target elements instead of applying a complete rule. A successful repair operation does not necessarily belong to the same rule that created the correspondence in a former run, i.e., consistency can be repaired by using another rule after modifications. If the repair process also fails, the algorithm deletes all obsolete correspondence and target elements (revoking the previous rule application), and re-translates all modified elements.

User-oriented Aspects: MoTE utilizes the notification mechanism of EMF to recognize model modifications at runtime *(U1)*. Offline derivation of modifications via a model diff is not supported. An overview of the actions performed by an incremental update is neither provided in advance nor a-posteriori *(U2)*. The TGG implementation of MoTE is designed for fully automatized scenarios and requires *functional* behaviour, which guarantees that the algorithm will always produce the same result. As a consequence, rule application and consistency restoration does not involve any decision making *(U3)*.

Incremental Update Capabilities: MoTE supports all four different change types *(C1)*, propagating attribute changes with synchronization operations, and moved elements with repair operations. TGG specifications in MoTE are required to be *conflict-free*, meaning that the algorithm

[1] In a TGG-context this means doing nothing when the delta to be propagated does not induce any inconsistencies.

expects at most one rule for translating an element at any given time. This limitation implies that the algorithm does not have to choose between applicable rules, and thus eliminates the need for backtracking or a look-ahead for the batch transformation and incremental updates *(C2)*.

Formal Properties: Currently, the correctness *(F1)* and completeness *(F2)* of the incremental algorithm in MoTE has not yet been proven formally. Nevertheless, experience from numerous case studies indicates that these properties hold for the incremental algorithm by analogy with the MoTE batch algorithm, formalized and proven to be correct and complete in [GHL10]. In general, the runtime complexity of the algorithm depends on the size of modifications and their effects *(F3)* as only the affected correspondences are processed. The increase in runtime in case of large-scale EMF data structures is negligible. The concept of synchronize and repair operations directly addresses hippocraticness and least change requirements *(F4)*, even though the repair process has its limitations (e.g., missing source elements can only be replaced by newly created elements and not already existing ones, even though this might be possible).

Current and Future Focus: Important future work for MoTE is the formalization of its incremental algorithm to guarantee the basic TGG formal properties (*F1, F2,* and *F3*), and perhaps, at least to a certain extent, even *(F4)*.

5.2 TGG Interpreter

The TGG Interpreter[2] directly interprets TGG rules without first deriving operational forward or backward rules from them. The incremental algorithm [GPR11] takes a modified graph triple, iterates over rule applications in the order in which they were applied and checks whether the respective rule application is still valid. In case of an invalidated rule application, a repair is attempted by updating attribute values of elements created by the rule. If the rule application is still invalid, e.g., due to structural modifications, correspondence and target elements associated with the rule application are *marked as deleted*. Subsequently, the affected source elements are re-translated by applying the same or another rule. While applying a rule to repair consistency, correspondence and target elements marked for deletion are *reused* when possible. Such a rule application (i) removes the deletion marks of reused elements, (ii) creates the remaining elements, and (iii) enforces attribute constraints. In a final iteration, correspondence and target elements that are still marked for deletion are finally destroyed.

User-oriented Aspects: As the TGG Interpreter processes a modified model triple directly, it does not require any intermediate delta structure. This eliminates the need for a change detector as all modifications, more precisely, all *induced inconsistencies* are handled *(U1)*. A debugger enables the transformation designer to examine the rule applications visually *(U2)*. This feature is, however, not yet provided in the official release. Users can influence the process of reusing elements via configuration files *(U3)*. However, user interaction is not supported for choosing between multiple applicable rules.

Incremental Update Capabilities: The concept of reusing deleted elements enables the handling of movements and attribute value changes along with additions and deletions (*C1*). In general, the TGG Interpreter does not impose strong restrictions such as conflict-freeness or functional behaviour, but also does not provide any guarantees that wrong decisions can be avoided

[2] http://www.cs.uni-paderborn.de/en/research-group/software-engineering/research/projects/tgg-interpreter.html

in every case, i.e., neither backtracking nor a look-ahead is implemented (*C2*). A crucial consequence for incremental updates is that the algorithm does not reconsider decision points from former runs to revoke invalidated rule applications (as required for our running example).

Formal Properties: Although the TGG Interpreter has been successfully used in various case studies, the correctness of the incremental algorithm is yet to be formally verified (*F1*). Due to a lack of backtracking or look-ahead functionality, the incremental algorithm is not complete (*F2*), i.e., it may fail to determine a correct sequence of rule applications to be revoked and re-applied. As the complete sequence of rule applications is checked for inconsistencies induced by modifications, an efficient incremental update is infeasible (*F3*). The runtime complexity of the algorithm is linear in the size of the model triple and exponential in the size of the modified and affected elements. Reuse of correspondence and target elements has, together with appropriate user configuration, the potential to retain as much information as possible. This addresses hippocraticness and least change requirements adequately (*F4*).

Current and Future Focus: An integration of the debugger in the official release, as well as an additional visualization plugin to highlight the results of an incremental update, are planned to improve plausibility *(U2)* during and after an incremental update. Backtracking functionality *(C2)* is also planned to avoid dead-ends when applying rules (*F2*).

5.3 eMoflon

eMoflon (www.emoflon.org) performs a *precedence analysis* [LAVS12] to extract translation dependencies in a model. The incremental algorithm calculates a *precedence graph* that determines (i) which elements are translated together in a rule application, and (ii) which elements serve as context for a certain translation. This auxiliary information is extracted automatically by means of a pattern matching process prior to the actual translations. The incremental algorithm traverses all dependencies of every deleted/added element and computes the set of transitively affected elements that must be re-translated as a consequence. Affected elements are either (potentially) created together with deleted/created elements or (potentially) require deleted/created elements as context. Using a *translation protocol* persisted from the previous transformation, all affected elements in the correspondence and output domain are deleted, while affected elements in the input domain are marked as to be re-translated. Starting from this intermediate graph triple, which represents a valid intermediate step of the batch mode, the normal batch algorithm [KLKS10] is used to re-translate all affected input elements. Finally, all auxiliary data, i.e., precedence graphs and translation protocols must be updated to complete the process.

User-oriented Aspects: eMoflon provides a data structure to express model modifications as deltas. This decouples change detection mechanisms from the actual incremental algorithm. Both online notifications, as well as offline model diff algorithms can, therefore, be employed to create the delta data structure required by the algorithm *(U1)*. Integrated support is, however, only provided for online change detection via EMF notifications as part of eMoflon in the official release. Translation protocols are recorded and can be used to “replay” the update in a visual integration environment *(U2)*. Moreover, the computed precedence graphs provide a suitable means for foreseeing the consequences of the modifications. User involvement is enabled in cases where more than one rule is applicable *(U3)*.

Incremental Update Capabilities: Currently, eMoflon supports only deletions and additions *(C1)*, i.e., attribute value changes and movements are interpreted as a combination of these two basic change types. Precedence graphs correctly indicate which rule applications must be revoked after modifications, irrespective of rule applicability *(C2)*, as they provide a global view of dependencies compiled from all TGG rules. Moreover, a *Dangling Edge Condition* (DEC) [KLKS10] is used as a look-ahead to determine if an edge of a translated node would remain untranslated after a rule application (this is sufficient to avoid the translation of subclasses with *r*2 in our running example). The restriction on TGGs is, therefore, relaxed from *conflict-freeness* to a *local completeness* criterion [KLKS10], which basically requires that DEC, i.e., a (local) look-ahead of one step on edges, is sufficient to avoid wrong decisions for the given TGG.

Formal Properties: The incremental algorithm has been shown in [LAVS12] to be correct *(F1)* and complete *(F2)*. Although the computation and maintenance of auxiliary information (precedence graphs and translation protocols) incurs a certain overhead, the core incremental algorithm has been shown, again in [LAVS12], to be efficient *(F3)*. The least change property is, however, clearly not guaranteed as the algorithm deletes all correspondence and output elements affected by modifications, without attempting to reuse them *(F4)*. In general, hippocraticness is also not guaranteed due to a missing consistency check at the beginning of the translation.

Current and Future Focus: An integrated change detection mechanism is currently under development *(U1)*. Support for attribute changes is planned to extend the incremental update capabilities *(C1)*. Finally, an optimization of the precedence analysis is in progress to further reduce the overhead and improve the efficiency of the complete incremental update process *(F3)*.

5.4 Comparative Discussion

Table 1 summarizes the qualitative comparison of MoTE, the TGG Interpreter, and eMoflon based on the introduced criteria, where ● denotes "sufficient/good", ◐ "can be improved", and ○ "missing/inadequate". A more detailed discussion of each criterion is provided in the following to reveal differences between the three tools.

	U1	U2	U3	C1	C2	F1	F2	F3	F4
MoTE	◐	○	○	●	○	◐	◐	●	●
TGG Interpreter	●	◐	◐	●	○	◐	○	○	●
eMoflon	◐	◐	●	◐	●	●	●	◐	○

Table 1: Summary of the Qualitative Comparison

Both online and offline changes are detected only by the TGG Interpreter *(U1)*, whereas MoTE and eMoflon detect only online changes via EMF notifications with an in-built change detection solution. Thus, MoTE and eMoflon can be improved by extending the change detection mechanism to the offline case. The TGG Interpreter and eMoflon provide additional components to understand the actions performed by the incremental update *(U2)*. These components are, however, currently used primarily by the tool developers who have a deep insight into the respective tool and must, therefore, be improved and simplified for end-users. eMoflon supports user interaction for choosing between different applicable rules and involves the user consequently in the decision points *(U3)*, whereas the TGG Interpreter uses user-specific configurations to control reusing deleted elements but does not provide support for user interaction when choosing between applicable rules. MoTE handles only updates without any decision making so user involvement is, therefore, completely missing.

MoTE and the TGG Interpreter propagate all four change types *(C1)*. eMoflon can propagate additions and deletions, only fulfilling the minimum requirements for incremental updates. Movements and attribute changes in eMoflon are expressed as a combination of deletions and additions, giving rise to a point for improvement. eMoflon reconsiders decision points from a former run (backtracking) or avoids wrong decisions (look-ahead) when propagating model changes *(C2)*, whereas MoTE and the TGG Interpreter do not have such a mechanism and handle only conflict-free TGGs with neither automatic nor manual decision making.

A formal proof of correctness *(F1)* and completeness *(F2)* in the incremental mode currently exists only for eMoflon. Correctness of the incremental algorithm in MoTE and the TGG Interpreter is examined in different case studies but must sill be proven on a formal basis. The same also applies to completeness of MoTE, whereas the TGG Interpreter does not guarantee completeness. Of all three tools, MoTE is most streamlined for guaranteed efficiency *(F3)*, which is also reflected by our runtime measurement results in Sect. 6. Efficiency of eMoflon suffers in practice from additional computation steps for auxiliary information and must be improved at this point, whereas the TGG Interpreter does not provide any guarantee for efficiency. MoTE and the TGG Interpreter attempt to fulfill least change requirements *(F4)* by repairing rule applications and reusing deleted elements. Reusing elements and repair operations do not necessarily guarantee least change or hippocraticness, but provide a sufficient means to attain these properties in practical scenarios. eMoflon does not strive for least change and hippocraticness in any way as it directly revokes all affected rule applications without any attempts at repair or reuse.

6 Quantitative Comparison

In order to have a common basis for runtime measurements that can be handled by all three TGG tools, we consider only additions and deletions (current limitation of eMoflon), and simplify our running example by removing the rules $r3$ and $r6$. The resulting TGG is conflict-free and can be handled without a look-ahead or backtracking (current limitation of MoTE and the TGG Interpreter). Starting with randomly generated, consistent pairs of class diagrams and database schemata with 1000, 5000, and 10000 elements, the following modifications to the source models were propagated incrementally to the corresponding target models: (i) adding 1, 10, 100, and 1000 elements, and (ii) deleting 1, 10, and 100 elements. Each update was executed 20 times on Intel Xeon E5420 machine with 2.5 GHz, 32GB RAM, running Debian Linux 5.0.3, Oracle JDK 1.7.0_25, and Eclipse 4.2., and the average values for each tool were measured. Note that exactly the same elements were modified in each benchmark run so that the execution times of all runs and all tools are directly comparable. The results for additions and deletions are depicted to the left and right of Fig. 2, respectively.

MoTE is currently the only tool that exhibits strictly efficient behaviour, depending only on the change sizes regardless of model sizes (e.g., approximately 11 ms for 1 added element *regardless of test model size*). Although eMoflon is in a few cases faster than MoTE (e.g., 1 or 10 deleted elements in 1000 elements), its incremental update time increases with the model sizes due to

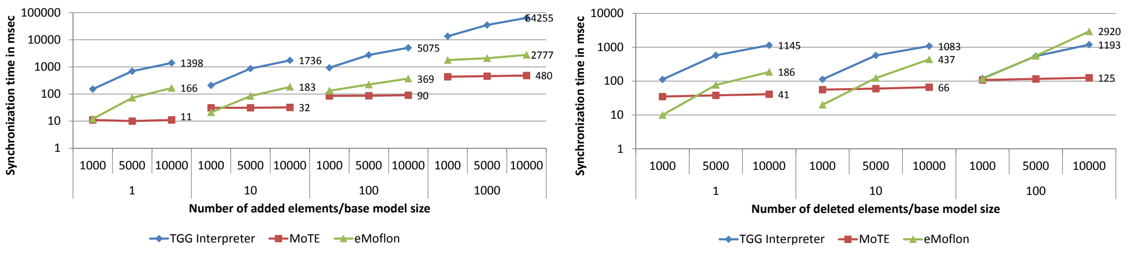

Figure 2: Runtime measurement results for incremental updates

the maintenance of auxiliary data structures (e.g., 0.1s to 3s for 100 deleted elements in different model sizes). This overhead is especially noticeable for deletions as this is a costly operation in EMF.[3] Finally, the update times of the TGG Interpreter is the longest in most cases (e.g., 64s when adding 1000 elements to 10000 elements, whereas eMoflon takes 2.7s, and MoTE only 0.4s) and depends on the model sizes as well. Similar observations can be made for runtime results in the backward direction.

7 Conclusion and Future Work

In this paper, we have provided a set of criteria to compare current and future TGG tools in incremental mode. We have also given a qualitative assessment of three prominent TGG tools based on these criteria, and complemented this with a quantitative comparison using runtime measurements of the well-known class diagrams to database schemata transformation. Our results lead to the following conclusions: For scenarios that can be appropriately handled with conflict-free TGGs, MoTE enables fully automatic and efficient incremental updates. eMoflon is able to handle TGGs that require a look-ahead and decision making with the current price of an overhead and a certain coupling with model size due to auxiliary data structures. The strength and focus of the TGG Interpreter lies in its information preservation capabilities and not in efficiency.

Our quantitative comparison is conducted on one small example to have a common and simple measurement basis for the considered tools. Our measurements provide a first insight into the performance of these tools. But the relevance of our results in a real-world application scenario might still depend on the characteristics of the respective scenario. In future work, we therefore plan to support and participate in the establishment of a new benchmark [ACG+14] and an example repository [CMSG13] in the bidirectional transformations (BX) community. Our aim is to cover more qualitative and quantitative aspects of incremental updates, in particular with TGGs, and to gain more representative results when comparing the performance of different tools. Finally, we encourage TGG researchers to extend this qualitative and quantitative comparison with new incremental TGG tools such as EMorF (www.emorf.org) and Henshin-TGG (http://www.eclipse.org/henshin/projects.php).

[3] eMoflon is almost completely implemented with graph transformations (bootstrapped with itself), increasing its dependency on EMF.

Bibliography

[ACG$^+$14] A. Anjorin, A. Cunha, H. Giese, F. Hermann, A. Rensink, A. Schürr. BenchmarX. In Candan et al. (eds.), *BX 14*. CEUR Workshop Proceedings 1133, pp. 82–86. Sun SITE, 2014.

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

[BRST06] J. Bézivin, B. Rumpe, A. Schürr, L. Tratt. Model Transformations in Practice Workshop. In Bruel (ed.), *MoDELS 05*. Volume 3844(January), pp. 120–127. Springer, 2006.

[CH06] K. Czarnecki, S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal* 45(3):621–646, 2006.

[CMSG13] J. Cheney, J. McKinna, P. Stevens, J. Gibbons. Towards a Repository of Bx Examples. In Candan et al. (eds.), *BX 14*. CEUR Workshop Proceedings 1133, pp. 87–91. Sun SITE, 2013.

[CP08] B. Cédric, A. Pierantonio. Model Differences in the Eclipse Modelling Framework. *UPGRADE* IX(2):29–34, 2008.

[GH09] H. Giese, S. Hildebrandt. Efficient Model Synchronization of Large-Scale Models. Technical report, Hasso-Plattner Institute, 2009.

[GHL10] H. Giese, S. Hildebrandt, L. Lambers. Toward Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars. Technical report 37, Hasso-Plattner Institute, 2010.

[GHN10] H. Giese, S. Hildebrandt, S. Neumann. Model Synchronization at Work : Keeping SysML and AUTOSAR Models Consistent. In Schürr 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.), *ECMFA 11*. Volume 6698, pp. 144–159. Springer, 2011.

[GR11] 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.), *AGTIVE 11*. LNCS 7233, pp. 222–237. Springer, 2011.

[HGO10] F. Hermann, U. Golas, F. Orejas. Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In Bézivin et al. (eds.), *MDI 10*. MDI 2010, pp. 22–31. ACM, 2010.

[HLG+13] S. Hildebrandt, L. Lambers, H. Giese, J. Rieke, J. Greenyer, W. Schäfer, M. Lauder, A. Anjorin, A. Schürr. A Survey of Triple Graph Grammar Tools. In Stevens and Terwilliger (eds.), *BX 13*. ECEASST 57. EASST, 2013.

[KEK+13] A. Kusel, J. Etzlstorfer, E. Kapsammer, P. Langer, W. Retschitzegger, J. Schoenboeck, W. Schwinger, M. Wimmer. A Survey on Incremental Model Transformation Approaches. In *ME 13*. 2013.

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

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

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

[Obj11] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query / View / Transformation Specification. 2011.

[Sch95] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In Mayr et al. (eds.), *WG 94*. LNCS 903, pp. 151–163. Springer, 1995.

[Ste08] P. Stevens. A Landscape of Bidirectional Model Transformations. In Lämmel et al. (eds.), *GTTSE 08*. LNCS 5235, pp. 408–424. Springer, 2008.

[Ste10] P. Stevens. Bidirectional model transformations in QVT : semantic issues and open questions. *SoSyM* 9(1):7–20, 2010.

[TEG+05] G. Taentzer, K. Ehrig, E. Guerra, J. D. Lara, L. Lengyel, T. Levendovszky, U. Prange, D. Varro, S. Varro-Gyapay. Model Transformation by Graph Transformation : A Comparative Study. In *MTiP 05*. 2005.