The Incremental Advantage: Evaluating the Performance of a TGG-based Visualisation Framework

Roland Kluge[1(✉)] and Anthony Anjorin[2]

[1] Technische Universität Darmstadt, Darmstadt, Germany
roland.kluge@es.tu-darmstadt.de
[2] University of Paderborn, Paderborn, Germany
anthony.anjorin@upb.de

Abstract. Triple Graph Grammars (TGGs) are best known as a *bidirectional* model transformation language, which might give the misleading impression that they are wholly unsuitable for unidirectional application scenarios. We believe that it is more useful to regard TGGs as just graph grammars with "batteries included", meaning that TGG-based tools provide simple, default execution strategies, together with algorithms for incremental change propagation. Especially in cases where the provided execution strategies suffice, a TGG-based infrastructure may be advantageous, even for unidirectional transformations.

In this paper, we demonstrate these advantages by presenting a TGG-based, read-only visualisation framework, which is an integral part of the metamodelling and model transformation tool eMoflon. We argue the advantages of using TGGs for this visualisation application scenario, and provide a quantitative analysis of the runtime complexity and scalability of the realised incremental, unidirectional transformation.

Keywords: Graph transformation · Triple graph grammars · Incremental model transformation

1 Introduction

Triple Graph Grammars (TGGs) [23] provide a declarative, rule-based means of specifying how two modelling languages are related. This is done in a direction-agnostic manner using rules that describe how related models can be simultaneously generated. TGGs are best known for their application to bidirectional model transformation, as both forward and backward transformations can be derived automatically from a TGG. When choosing the right model transformation for a certain task, one might thus assume that TGGs, being "bidirectional", are somehow inherently unsuitable for unidirectional tasks. This is perhaps due to the assumption that there must be some "overhead" involved in specifying both directions at once. While this might be conceptually true, we believe it is more helpful to regard TGGs as just graph grammars, but with "batteries

R. Echahed and M. Minas (Eds.): ICGT 2016, LNCS 9761, pp. 189–205, 2016.
DOI: 10.1007/978-3-319-40530-8_12

included". This means that TGG-based tools provide a set of default, out-of-the-box execution strategies including a forward transformation, a backward transformation, simultaneous model generation [22], incremental forward and backward change propagation [2], and consistency checking [15]. We suggest to base the decision to use TGGs less on the question of bidirectionality and more on the following:

Is the transformation task simple enough to be handled by one of the default execution strategies? The forward and backward transformations that can be derived automatically from a TGG are rather simple, performing only a single pass over the input model (each element is visited and marked exactly once). A transformation that is inherently complex, requiring rules with advanced application conditions that create or delete auxiliary elements to trigger the application of other rules, most probably cannot be expressed elegantly (or at all) as a TGG. The same argument applies to deeply nested, recursive control flow structures. Simple, straightforward transformations are a much better fit for TGGs.

Is incremental change propagation required? The true potential of TGGs lies in the formally founded infrastructure for incremental change propagation. Without any additional specification effort, the automatically derived forward and backward transformations can be executed in an incremental mode, updating existing related models appropriately. Incrementality is crucial when the output model cannot be recreated from scratch without losing information [7]. In many situations (large models and small changes), incrementality can also speed-up the transformation process [7]. Even for "simple, straightforward" unidirectional transformations, providing support for incremental updates is non-trivial, especially concerning a choice of sensible semantics.

Contribution. In this paper, we present a case study for using TGGs (i. e., graph transformations) to visualise various models used in the specification and execution of graph transformations. The generated visualisations are rendered in a read-only view, meaning that the transformation is currently unidirectional. We argue that using a TGG-based tool for this task is nonetheless advantageous, as the provided infrastructure can be suitably leveraged to enable declarative, compact specifications that can be executed incrementally.

To address justified concerns of scalability, we perform a detailed quantitative analysis of the transformation, which has been implemented as a general visualisation framework and is currently an integral part of the metamodelling and model transformation tool eMoflon [16]. To enable a realistic evaluation, we make use of a substantial set of real-world models collected over five years of using eMoflon in diverse applications including industrial case studies, bootstrapping eMoflon as much as possible (this includes the TGGs for the TGG-based visualisation framework itself!), and a substantial collection of unit and system tests (see [6] for a repository containing some of these examples).

Structure. Section 2 presents our TGG-based visualisation framework and is complemented by a quantitative evaluation of the realised incremental transformation provided in Sect. 3. Section 4 gives a brief overview of related case studies. Section 5 concludes the paper with a summary and a brief outlook on future work.

2 A TGG-Based Visualisation Framework

A schematic overview of the TGG-based visualisation framework realised in eMoflon is depicted in Fig. 1. Further examples can be found in the appendix.

The top part of the diagram (❶ and ❷) represents what is seen by the *end-user*: ❶ is a tree-view representation of a source model. ❷ is a generated visualisation of the currently selected model element, which is a TGG rule (red box). The visualisation of the TGG rule is in concrete syntax, as defined by an underlying transformation (specified as a TGG). Changes made to the source model inside the tree-view editor are propagated *incrementally* to the visualisation as soon as the editor content is saved.

The bottom part of the diagram (between ❸ and ❹) depicts the chain of transformations used to generate the visualisation from the source model. In the most general case, the end-user makes a change Δ_S (referred to in the following as *delta*) to the source model G_S and triggers an update by saving the editor (represented by ❸). In this context, a "batch" transformation is just a special case with an empty source model G_S. Note that the source model is a typed graph G_S with type graph TG_S. As TGG-based synchronisers operate on *triple graphs*, a correspondence graph G_C and visualisation model G_{vis} are maintained in the background by the tool as a consistent typed triple graph $G_S \leftarrow G_C \rightarrow G_{vis}$ with type triple graph $TG_S \leftarrow TG_C \rightarrow TG_{vis}$. In the context of the visualisation framework, the type graph TG_{vis} is fixed, representing the visualisation capabilities that the framework currently supports.

For every source type graph TG_S that is to be visualised, a *transformation designer* must provide a TGG that specifies how triples $G_S \leftarrow G_C \rightarrow G_{vis}$ are to be constructed. This TGG entails decisions on how source model elements are to be mapped to visualisation elements such as rectangles, arrows, labels, colours, and other available shapes. A forward synchroniser *fwd* is derived automatically from this TGG and is used to forward propagate the applied source model delta Δ_S to yield correspondence and visualisation deltas Δ_C and Δ_{vis}, respectively. As depicted in Fig. 1, applying these computed deltas results in a new consistent triple graph $G'_S \leftarrow G'_C \rightarrow G'_{vis}$, with consistently updated correspondence graph G'_C and visualisation G'_{vis}. Note that this resulting triple graph is also well-typed even though this is not shown explicitly in Fig. 1.

In a final step, G'_{vis} is used to regenerate the visualisation presented to the end-user via a model-to-text transformation *m2t* to produce a file in the `.dot` format, which is then converted to an image (e. g., `.jpeg`) via the Graphviz[1] command line tool *dot*. This final image ❹ is what the end-user can observe in

[1] http://www.graphviz.org.

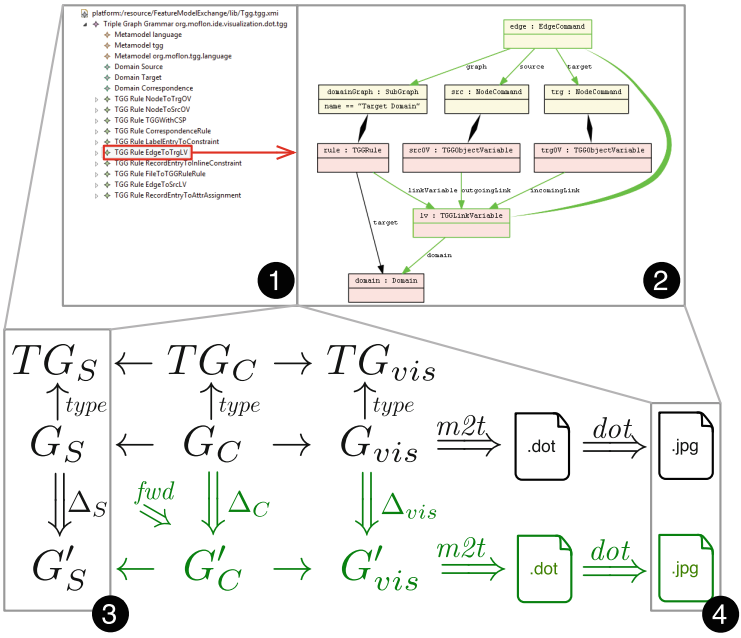


Fig. 1. Overview of TGG-based visualisation framework (Color figure online)

a corresponding view ❷. In contrast to *fwd*, note that *m2t* and *dot* are both currently non-incremental (indicated in Fig. 1 by omitting the deltas).

3 Evaluation

In this section, we present and discuss a quantitative analysis of the forward transformation (*fwd* in Fig. 1) used in our visualisation framework. In the following, we briefly describe the five types of source models that can be visualised, currently. For each transformation, we provide the following statistics to give a rough impression of the complexity of the transformation: (1) the total number of TGG rules n_{tot}, (2) the number of abstract TGG rules n_{abs},[2] and (3) the average number of (object and link) variables per TGG rule $\overline{n_{\text{var}}}$.

TGG [n_{tot}=14, n_{abs}=4, $\overline{n_{\text{var}}}$=22.1]: A *TGG rule*, such as depicted in Fig. 1, is a monotonic triple rules (triples of story patterns without deletion).

SDM [n_{tot}=11, n_{abs}=3, $\overline{n_{\text{var}}}$=15]: *Story diagrams*, a dialect of programmed graph transformations that is similar to simplified UML activity diagrams, are used to specify control flow structures in eMoflon.

[2] An *abstract TGG rule* serves to, e. g., extract commonalities of multiple TGG rules, but cannot itself be applied. TGG rules may refine other (abstract or non-abstract) rules to reuse common elements. Refinement is roughly comparable to the purpose of inheritance in object-oriented programming languages. See [3] for more details.

SP [n_{tot}=9, n_{abs}=1, $\overline{n_{\text{var}}}$=16.5]: A *story pattern* represents a regular graph transformation rule that is embedded in an activity node of a story diagram. A story pattern is a graph with annotated nodes and edges, formally representing a graph transformation rule $r : L \rightarrow R$ in the SPO approach.

TM [n_{tot}=3, n_{abs}=0, $\overline{n_{\text{var}}}$=16]: A *triple match* represents the match[3] of a TGG rule in an input model and is similar to a story pattern.

PG [n_{tot}=3, n_{abs}=0, $\overline{n_{\text{var}}}$=11.7]: A *precedence graph* is a—predominantly acyclic—intermediate data structure representing all possible triple matches of all TGG rules in an input model, together with all resulting dependencies between these triple matches. Precedence graphs are used to control the TGG-based synchronisation process in eMoflon [2].

Our first two research questions to be investigated with this analysis focus on the performance of *fwd* when executed in batch mode, i. e., the first time a user opens a source model in an editor and chooses to visualise it.

RQ 1a: Does *fwd* scale? More precisely does the runtime of *fwd* grow non-exponentially with source model size when executed in batch mode?

RQ 1b: Is the batch runtime of *fwd* acceptable for realistic source models?

The next three research questions concern the incremental execution of *fwd*:

RQ 2a: How large is the speed-up in runtime obtained via incremental change propagation? More precisely, how large is the ratio of runtime of *fwd* in incremental mode compared to batch mode?

RQ 2b: Is this speed-up in runtime perceivable for realistic source models? Would an end-user notice the difference in runtime for re-translating the whole source model as compared to incrementally propagating changes?

RQ 2c: Is it better (wrt. attained speed-up) to synchronise frequently, i. e., after every small change, or to accumulate changes before synchronising?

Finally, the last research question investigates the optimality of *fwd*:

RQ 3: To what extent is incremental change propagation coupled to model size? Optimal would be no coupling at all, i. e., constant time for propagating the same change independent of model size.

Evaluation Setup. The dataset of the evaluation comprises two subsets:

D1: To provide for "realistic" models, required for RQs 1b and 2b, we collected instances of all five metamodels from the current eMoflon developer workspace and all test suite workspaces. These models have been used and collected for over five years from various industrial case studies, the development of eMoflon itself, and numerous examples and tests. To obtain realistic (or even pessimistic) results, all runtime data for D1 was acquired on a typical business notebook (i7-4600U with 2×3.3 GHz, 12 GB RAM) running Windows 8.1 (64bit).

[3] A match of a rule $r : L \rightarrow R$ in a graph G is an occurrence $m : L \rightarrow G$ of the left-hand side L of the rule in G.

Table 1. Characteristics of the evaluation datasets D1 and D2

Property	TGG(D1)	TGG(D2)	SDM(D1)	SP(D1)	TM(D1)	PG(D1)
Model count	3,660	12	4,395	11,191	11,310	293
Mean model size	173	206,945	342	45	69	2,991
Median model size	124	182,436	269	28	56	272

D2: To evaluate scalability for large models, required for RQs 1a, 2a, and 3, we derived a TGG-based model generator [22] from the TGG for visualising TGG rules and used it to synthesise large TGG rules. We chose TGG rules for this complementary synthetic data generation as the corresponding visualisation is currently the most often used one in eMoflon and is thus the richest (uses most visual elements). To be able to run the evaluation in a reasonable amount of runtime, the data for D2 were acquired on a workstation (i7-2600, 4×3.4 GHz, 8 GB RAM) with Windows 7 Professional (64bit).

On both machines, we used Eclipse Mars 4.5 (`-Xmx4G`), eMoflon 2.12.0[4] and version 1.0.0 of our evaluation application[5]. Table 1 summarises the core characteristics of the datasets. The size of a model is the number of its contained nodes (*EObject*s) and edges (*EReference*s). Both the mean and median of all model sizes are provided to indicate the presence of outliers, e. g., in **PG** (D1).

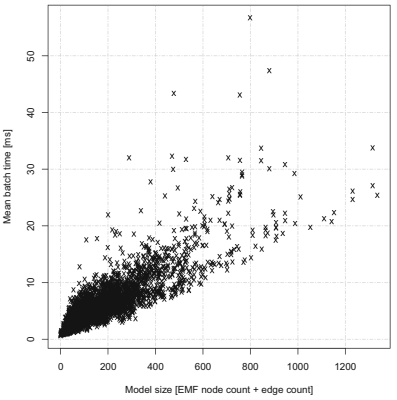
3.1 RQ1: Scalability of Batch Transformation

Figure 2 depicts the runtime for batch transformation in milliseconds plotted over model size for all models. The caption of each subplot shows the dataset and number n of models. Each data point is the median execution time of 5 runs. When comparing the plots, it is important to note that the x- and y-axes of all plots are of vastly different scale. The characteristic runtime values are additionally summarised in Table 2.

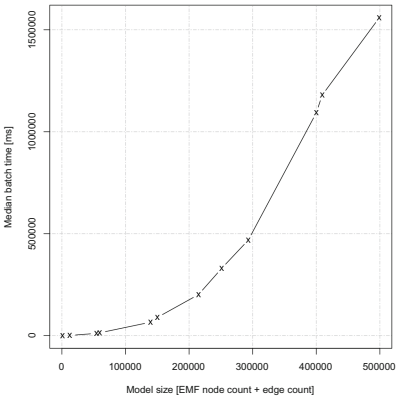
Table 2. Characteristic batch runtime values

Property		TGG(D1)	TGG(D2)	SDM(D1)	SP(D1)	TM(D1)	PG(D1)
Maximum	[ms]	56.7	1,558,500.9	55.4	122.9	38.6	9,019.6
Mean	[ms]	6.0	417,717.3	6.4	3.7	7.0	112.9
Median	[ms]	5.0	145,397.2	6.3	2.1	6.3	3.9

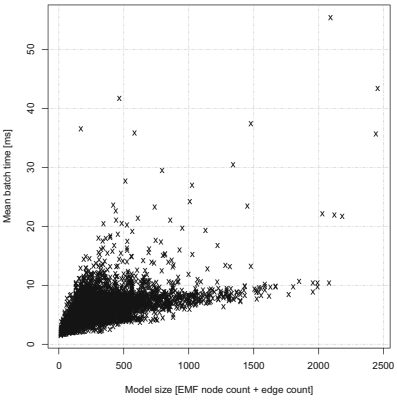
[4] http://www.emoflon.org.
[5] https://github.com/eMoflon/paper-icgt2016/releases/tag/icgt2016-v1.0.0.



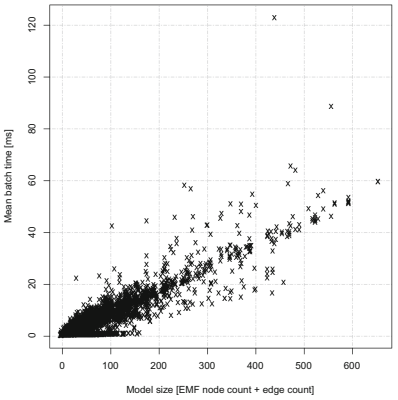
(a) TGG (D1, n=3,660)



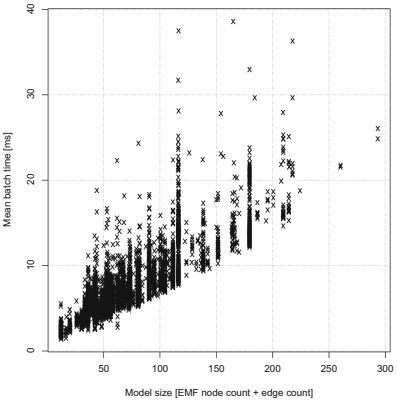
(b) TGG (D2, n=12)



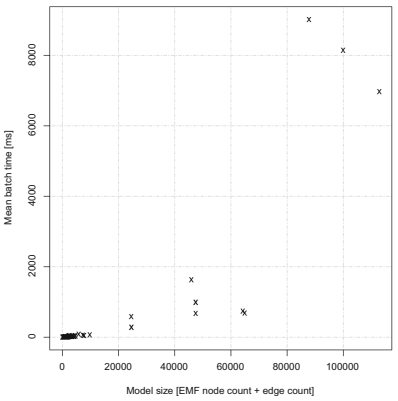
(c) SDM (D1, n=4,395)



(d) SP (D1, n=11,191)



(e) TM (D1, n=11,310)



(f) PG (D1, n=293)

Fig. 2. Runtime of batch transformation over model size (n: model count)

Discussion. While the size of the largest story pattern (SP) in D1 is below 700, the size of the largest precedence graph (PG) is above 100,000. This wide range of real-world model sizes shows that the visualisation of large models is *not* an unrealistic requirement, justifying our complementary dataset (D2) of synthetically generated models (up to 500,000 in size).

The plots in Fig. 2a, c–e indicate that the runtime of the batch transformation is generally linear for model sizes of up to about 1,000 elements. This is a positive result, as our dataset D1 shows that most realistic models for visualisation are in this range.

For larger models, however, Fig. 2b and f indicate non-linear behaviour. This is to be expected, as the complexity class for TGG-based transformation is polynomial [14]. The absolute values are still arguably reasonable for a visualisation task: about 8 (25) min for a model of size 300,000 (500,000). For small models, Table 2 shows that the mean and median runtimes for models in D1 are less than 10 ms. The large gap between mean and median execution time for PG can easily be explained by the three extreme outliers in D1.

In summary, our results suggest the following answers to RQ 1: (1a) *fwd* appears to scale satisfactorily even up to model sizes of over 500,000, and (1b) batch runtime for realistic source models in D1 is certainly acceptable for visualisation purposes (being less than 10 ms).

3.2 RQ2: Synchronisation Behaviour

To analyse synchronisation behaviour, we focused on TGG models. For each data point, we first performed a batch forward transformation and then applied the following changes to the source model (a TGG rule), to mimic typical modifications applied by an end-user: **(C1**) addition of three object variables, **(C2)** renaming of one object variable, and **(C3)** removal of two random object variables. To investigate RQ 2c, we consider the following two situations: synchronisation after every change (II), and synchronisation only after performing all changes (III). In both cases, we compare the required time with (I), the duration of a batch forward transformation after all changes.

Table 3 summarises the results of this experiment for both datasets, D1 and D2. For each model, the runtimes for each situation (I)–(III) is the median of five runs. The last two rows show the runtime of the synchronisation as a percentage of the batch transformation (the lower the value, the greater the speed-up). The given maximum, mean, and median values have been calculated for the metric of each row, i. e., the maximum value for (II)/(I) is *not* equal to the ratio of the maximum (II) and the maximum (I) values.

Discussion. The benefit of synchronising changes incrementally instead of re-transforming the entire model is particularly evident for D2. The synchronisation only takes between 0.2 % and 1.5 % of the batch transformation time in the mean

Table 3. Comparison of batch transformation `b` in I and synchronisation `s` in II, III.

			D1				D2			
			Max	Mean	Median	Min	Max	Mean	Median	Min
I	`C1+C2+C3+b`	[ms]	101.0	11.4	7.8	0.4	1,827,980	504,472	186,756	233.6
II	`C1+s+C2+s+C3+s`	[ms]	17.8	1.8	1.6	0.3	2,387.2	990.8	802.4	29.9
III	`C1+C2+C3+s`	[ms]	24.6	2.2	1.1	0.1	822.7	330.8	277.4	2.1
II/I	Rem. runtime	[%]	166.7	32.5	22.2	2.1	12.8	1.5	0.5	0.1
III/I	Rem. runtime	[%]	90.9	27.5	20.5	0.8	0.9	0.2	0.2	0.0

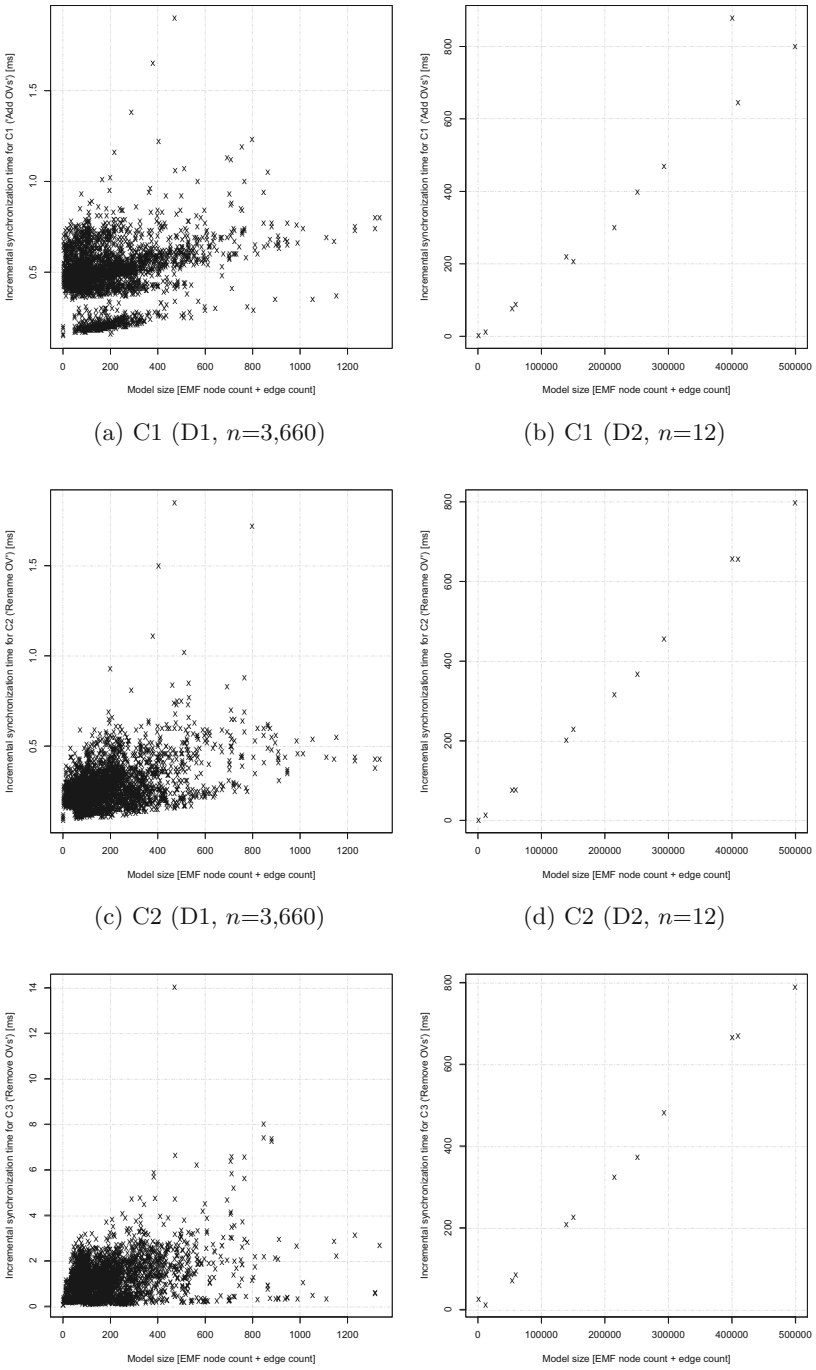
and median cases. In the worst case, when the ratio of synchronisation time and batch re-transformation time is maximal, synchronising still takes only 12.8 % to 0.9 % of batch runtime. For D1, i. e., real-world models, the speed-up is less impressive but still remarkable: In the mean and median cases between about 70 % and 80 % of the runtime is saved.

Our results thus suggest the following answers to our research questions: the speed-up enabled by incrementality is substantial and, as can be expected, increases with model size (RQ 2a), even though the speed-up is still substantial for D1, for most realistic models, an end-user probably will not notice a difference of only a few milliseconds in our visualisation scenario (RQ 2b), finally, incremental propagation appears to perform somewhat better if changes are collected (RQ 2c). Although this is not so clear for small- and medium-sized models in D1, the difference is evident for the larger models in D2. This is because (1) every synchronisation run has a certain overhead that increases with model size due to technical reasons, and (2) certain optimisations can be performed by the algorithm, propagating multiple changes at the same time.

3.3 RQ3: Coupling of Incremental Change Propagation to Model Size

The plots in Fig. 3 show the runtime of synchronisation over model size for D1 (Fig. 3a, c and e) and D2 (Fig. 3b, d and f). In each row of Fig. 3, the left figure shows the runtime behaviour for (realistic) model sizes of up to about 1,200, while the right figure shows asymptotic runtime behaviour for large synthetic models. When comparing plots, note that the x- and y-axes of left and right plots have vastly different scales.

Discussion. For an optimal synchroniser, incremental propagation time would be constant, i. e., independent of model size. Figure 3 shows that this is not really the case in practise (for eMoflon). Due to technical reasons and challenges involved with using EMF collections for large models, there is a certain coupling with model size. The results are, nonetheless, reasonably positive: for real-world models (left plots) synchronisation time is almost constant with only a slight linear increase (less than 0.5 ms). For large models, the linear increase is evident but with a very small gradient: for all changes, it only takes about half a second longer to visualise a model with 300,000 more elements.



(a) C1 (D1, n=3,660) (b) C1 (D2, n=12)

(c) C2 (D1, n=3,660) (d) C2 (D2, n=12)

(e) C3 (D1, n=3,660) (f) C3 (D2, n=12)

Fig. 3. Synchronisation time over model size (TGG, n: model count)

Our answer to RQ 3 is that a certain coupling of incremental propagation to model size is indeed still present in practise, but it is linear and reasonably small for our application scenario.

3.4 Threats to Validity

Our primary concern is *external validity*, i. e., can our results be generalised beyond our specific case study. This is a justified concern and has two orthogonal dimensions: (1) Do our results hold for other TGG-based tools or are they specific for eMoflon? (2) Do our results hold for other possibly more "complex" application scenarios that require, e. g., hundreds of TGG rules. Concerning (1), TGG comparison papers [12,18] have shown that TGG-based tools are quite diverse, especially concerning their underlying synchronisation algorithms. It is thus difficult to argue that our results hold in any way for "TGGs in general". More evidence should be provided with a new comparison paper, comparing current TGG-based tools using, e. g., our data from this case study. To mitigate (2), we have mined all our workspaces and collected a substantial number of real-world metamodels for the measurements (our dataset D1). This ensures that at least the input data for the transformation is somewhat realistic and not completely synthesised. The task of visualisation is also quite varied, ranging from story diagrams that are deeply nested tree-like structures, to flat, highly connected, more graph-like patterns (TGG rules, story patterns, triple matches). The primary limitation of our case study is more the "complexity" of the TGGs used for the visualisation, the largest TGG having only 14 rules. Although we have 5 TGGs, so in total 40 TGG rules, this is still not comparable to other application scenarios requiring hundreds of TGG rules. We argue, however, that equating complexity with number of rules is naïve: we have encountered cases that are essentially trivial 1–1 bijections, but still require hundreds of rules as the source and target models simply have many types.

Finally, using two different machines to perform the runtime measurements on D1 and D2, respectively, may be considered a threat to *construct validity*. We emphasise here, however, that the objectives of performing the experiments on the two datasets were rather different: While the measurements on D1 focused on applicability in terms of acceptable runtime for realistic models, the measurements on D2 served to observe the behaviour of the TGG-based visualisation for large, synthesised models.

4 Related Work

This paper builds on our previous work in [16,17] and shares the common goal of bootstrapping eMoflon. In [16], Leblebici et al. present the various model transformations used in eMoflon and describe how we have progressed from

an initial implementation of our import/export in C#, to a bootstrapped version with story diagrams (a unidirectional programmed graph transformation dialect), to a final bootstrapped TGG-based version that is still being optimised and extended up until today. A runtime comparison of story diagrams and TGGs is provided, showing on the positive side that TGGs are expressive enough to derive both forward (export) and backward (import) transformations from the same specification. Noteworthy is also that both directions perform comparably well, exhibiting almost linear behaviour for up to 10,000 elements, and then polynomial until running out of memory for about 300,000 elements. This indicates that TGGs are inherently symmetric and do not favour any direction. On the negative side, however, the measurements show that TGGs are still 10–15 times slower than story diagrams with lots of room for improvement in this regard. In comparison to this paper, the TGG-based transformations in [16] were *not* executed incrementally and the provided measurements thus give no indication of how feasible or useful this might be. The measurements are also for a single TGG and a single pair of source and target metamodels, while we provide evidence for various research questions using 5 TGGs and a substantial number of diverse source metamodels collected over five years in our test and development workspaces (the visualisation, i. e., target metamodel is fixed in all cases).

In [17], Leblebici et al. compare TGGs implemented in eMoflon with Medini QVT[6], showing that TGGs outperform Medini QVT up to a factor of 20 for model sizes of about 1,000–200,000 elements. In comparison to this paper, the focus of [17] is on showcasing multi-amalgamation, a new language feature of TGGs. Only a single "toy" TGG and a fixed pair of source and target metamodels are used for the comparison. Finally, just as with [16], the transformation is *not* executed in an incremental mode.

There has also been comparable work in the TGG community such as [12,18], which provide a comparison of various TGG tools, including runtime measurements. In contrast to this paper, the focus of [12,18] is on comparing the different tools and not on providing evidence for the performance or advantages of TGGs in general. To ensure that all tools could be used for the exact same TGG, a very simple toy example is used, and only synthetic data is generated for the measurements. The results indicate, however, that there are considerable differences between TGG tools regarding runtime efficiency and expressiveness. This means that our results are primarily valid for eMoflon and cannot be directly generalised to all other TGG tools (see the discussion in Sect. 3.4).

A further source for TGG runtime measurements and comparison with other tools is the annual transformation tool contest (TTC). For example, [11,13, 20] present TGG-based solutions to various contests. Although these results provide evidence for the expressiveness and applicability of TGGs, it is difficult to compare solutions in many cases: for example, [13] and [11] provide solutions using different TGG tools, but the degree of freedom of the contest (the choice of the source metamodel) makes it impossible to compare absolute runtime values. In many cases, it is also impossible to discern the runtime complexity of the

[6] http://projects.ikv.de/qvt.

solutions, as the provided test cases are more often used to ensure correctness. Nonetheless, both [20] and [11], for example, provide encouraging evidence that the involved TGG transformation is *not* necessarily the bottleneck in practical model transformation chains. Our experience corroborates this as `dot` dominates our transformation chain for large diagrams, especially as it is not incremental.

Finally, there is some evidence indicating that TGGs can be used successfully for industrial scale applications. In [10], Hermann et al. report on using TGGs for the translation of satellite procedures. Their results show that a pragmatic mix of programmed graph transformation and TGGs can be made to be "more efficient than what is needed for practical use" by applying advanced optimisation techniques. Such powerful domain- and even task-specific optimisations are feasible mainly due to the formal and declarative nature of graph transformations. The application scenario of [10] provides an interesting contrast to this paper as it is also unidirectional but *not* incremental. The main motivation for using TGGs in [10] is the formal guarantee of correctness, while in the case of our visualisation, correctness is important but not crucial; incrementality, conciseness, and readability are arguably more useful for our application scenario.

An inherently incremental industrial application scenario is presented in [4]. Blouin et al. demonstrate how a synchronisation layer between textual and graphical editors can be established using TGGs. As explained in [4], incrementality, expressiveness, and scalability are crucial for the application scenario. Unfortunately, no evaluation and measurement results are provided by Blouin et al., making it hard to conclude more than that the TGG-based solution was "fast enough" for practical usage. Other (industrial) case studies include work from Giese et al., e. g., in [8] for a TGG-based synchronisation between SysML and AUTOSAR models, and from Greenyer et al., e. g., in [9] for a TGG-based transformation of sequence diagram specifications to timed game automata.

5 Conclusion and Future Work

In this paper, we presented a TGG-based visualisation framework, which is currently being used as an integral part of the metamodelling and model transformation tool eMoflon. This is an example of a real-world, unidirectional, and incremental application scenario for TGGs. With a detailed quantitative analysis, we have shown that the realised transformation scales with model size, and that incrementality provides a substantial speed-up. The case study highlights the major advantage of TGGs: due to their declarative nature, multiple default execution strategies for the same TGG can be provided by a TGG tool. Specifically, we investigated the derived forward incremental mode, and made use of the simultaneous mode for generating large models for our scalability analysis.

A current limitation of the visualisation framework is that some steps in the tool chain are not incremental (e. g., *dot* used to render diagrams), and become the bottleneck of the framework. Future work includes, therefore, use cases that further realise the potential of TGGs such as: (1) allowing manual adjustments of the layout in the visualisation (i. e., coping with information loss), and

(2) implementing a completely incremental tool chain such as a TGG-based code generator for EMF. Case studies considering the correspondence between the concrete and abstract syntax (or semantics) of a specification, as discussed, e. g., in [1,5], would also be interesting for further investigating the potential of graph transformation in general and TGGs in particular. Finally, a comparison of our TGG-based model generator to other model generators such as [19,21] would be illuminating: it is, for instance, currently impossible to enforce certain statistical properties using our model generator, but the models are at least guaranteed to be translatable with the underlying TGG. This would not be the case with general purpose (random) model generators and would require a potentially large set of additional constraints to adequately control the generation.

Acknowledgements. This work has been funded by the German Research Foundation (DFG) as part of projects A01 within the Collaborative Research Centre (CRC) 1053 – MAKI.

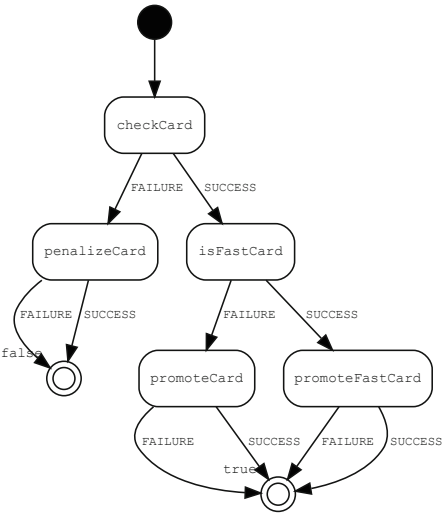
Appendix: Examples from the eMoflon Handbook

We show concrete examples of visualised source models taken from the eMoflon handbook,[7] whose illustrative example is Leitner's learning box, a system, e. g., for language learning. This system works by creating cards, sorted into sequential partitions, with a front face showing the known word (e. g., "hello" in English) and a back face showing the to-be-learnt word (e. g., "Hallo" in German). While exercising, the learner takes a card from a partition, tries to guess the back-face word based on the front-face word, and, if successful, may move the card to the next partition. A so-called fast card contains easy-to-learn words and may be moved to the last partition upon success, immediately.

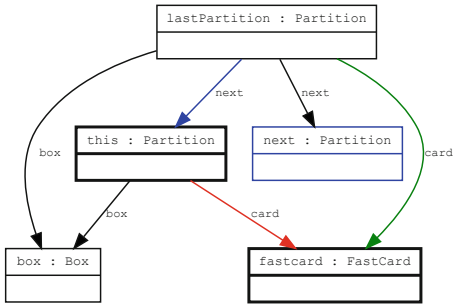
The story diagram in Fig. 4a shows the logic of checking a card: If the answer is correct (story pattern `checkCard`) and if the card is a so-called fast card (story pattern `isFastCard`), then this card is promoted to the last partition, as shown in the story pattern depicted in Fig. 4b.

Another task in the eMoflon handbook is to synchronise (using TGGs) a learning box with a dictionary, whose entries can be thought of as simple key-value pairs. Figure 4c shows the precedence graph resulting from translating the sample box in the handbook into a dictionary. The root node `BoxToDictionaryRule 0` indicates that the box is first of all translated into an empty dictionary, before translating all cards to dictionary entries. Finally, Fig. 4d depicts the triple match that corresponds to `CardToEntryRule 5` in Fig. 4c. This match shows that the card containing "Question One" is mapped to the entry with content "One : Eins".

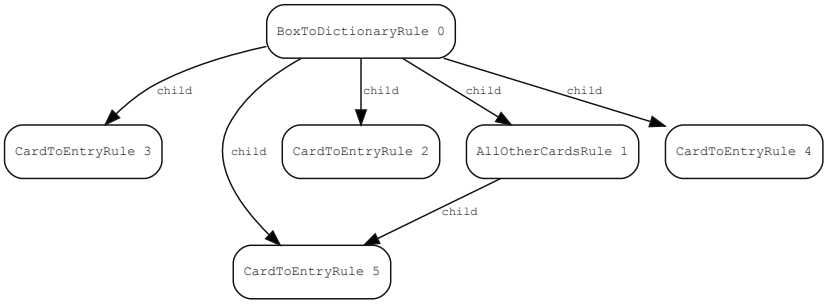
[7] http://www.emoflon.org/.



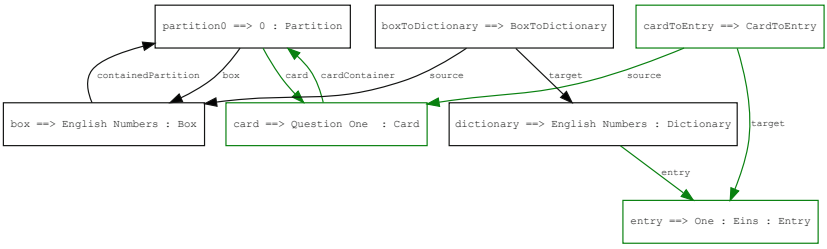
(a) SDM `checkCard`



(b) SP `checkCard::promoteFastCard`



(c) PG of box-to-dictionary synchronisation



(d) TM `BoxToDictionaryRule 0`

Fig. 4. Visualisations of sample models (a) SDM `checkCard` (b) SP `checkCard::promoteFastCard` (c) PG of box-to-dictionary synchronisation, (d) TM `BoxToDictionaryRule 0`

References

1. Pérez Andrés, F., de Lara, J., Guerra, E.: Domain specific languages with graphical and textual views. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 82–97. Springer, Heidelberg (2008)
2. Anjorin, A.: Synchronization of Models on Different Abstraction Levels using Triple Graph Grammars Phd thesis, Technische Universität Darmstadt (2014)
3. Anjorin, A., Saller, K., Lochau, M., Schürr, A.: Modularizing triple graph grammars using rule refinement. In: Gnesi, S., Rensink, A. (eds.) FASE 2014 (ETAPS). LNCS, vol. 8411, pp. 340–354. Springer, Heidelberg (2014)
4. Blouin, D., Plantec, A., Dissaux, P., Singhoff, F., Diguet, J.-P.: Synchronization of models of rich languages with triple graph grammars: an experience report. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 106–121. Springer, Heidelberg (2014)
5. Bottoni, P., Guerra, E., de Lara, J.: Enforced generative patterns for the specification of the syntax and semantics of visual languages. JVLC **19**(4), 429–455 (2008)
6. Cheney, J., McKinna, J., Stevens, P., Gibbons, J.: Towards a repository of Bx examples. In: Workshops of EDBT/ICDT 2014. CEUR Workshop Proceedings, vol. 1133, pp. 87–91. CEUR-WS.org (2014)
7. Diskin, Z., Wider, A., Gholizadeh, H., Czarnecki, K.: Towards a rational taxonomy for increasingly symmetric model synchronization. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 57–73. Springer, Heidelberg (2014)
8. Giese, H., Hildebrandt, S., Neumann, S.: Model synchronization at work: keeping SysML and AUTOSAR models consistent. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 555–579. Springer, Heidelberg (2010)
9. Greenyer, J., Rieke, J.: Applying advanced TGG concepts for a complex transformation of sequence diagram specifications to timed game automata. In: Schürr, A., Varró, D., Varró, G. (eds.) AGTIVE 2011. LNCS, vol. 7233, pp. 222–237. Springer, Heidelberg (2012)
10. Hermann, F., Gottmann, S., Nachtigall, N., Ehrig, H., Braatz, B., Morelli, G., Pierre, A., Engel, T., Ermel, C.: Triple graph grammars in the large for translating satellite procedures. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 122–137. Springer, Heidelberg (2014)
11. Hermann, F., Nachtigall, N., Braatz, B., Engel, T., Gottmann, S.: Solving the FIXML2Code-case study with HenshinTGG. In: TTC 2014. CEUR Workshop Proceedings, vol. 1305, pp. 32–46. CEUR-WS.org (2014)
12. Hildebrandt, S., Lambers, L., Giese, H., Rieke, J., Greenyer, J., Schäfer, W., Marius Lauder, A., Anjorin, A. Schürr : A survey of triple graph grammar tools. In: BX 2013. ECEASST, vol. 57. EASST (2013)
13. Kulcsár, G., Leblebici, E., Anjorin, A.: A solution to the FIXML case study using triple graph grammars and eMoflon. In: TTC 2014. CEUR Workshop Proceedings, vol. 1305, pp. 71–75. CEUR-WS.org (2014)
14. Lauder, M., Anjorin, A., Varró, G., Schürr, A.: Efficient model synchronization with precedence triple graph grammars. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 401–415. Springer, Heidelberg (2012)
15. E. Leblebici: Towards a graph grammar-based approach to inter-model consistency checks with traceability support. In: BX 2016. CEUR Workshop Proceedings, vol. 1571. CEUR-WS.org (2016)

16. Leblebici, E., Anjorin, A., Schürr, A.: Developing eMoflon with eMoflon. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 138–145. Springer, Heidelberg (2014)
17. Leblebici, E., Anjorin, A., Schürr, A., Taentzer, G.: Multi-amalgamated triple graph grammars. In: Parisi-Presicce, F., Westfechtel, B. (eds.) ICGT 2015. LNCS, vol. 9151, pp. 87–103. Springer, Heidelberg (2015)
18. Leblebici, E., Anjorin, A., Schürr, A., Hildebrandt, S., Rieke, J., Greenyer, J.: A comparison of incremental triple graph grammar tools. In: GT-VMT 2014. ECEASST, vol. 67. EASST (2014)
19. Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform random generation of huge metamodel instances. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 130–145. Springer, Heidelberg (2009)
20. Peldszus, S., Kulcsár, G., Lochau, M.: A Solution to the java refactoring case study using eMoflon. In: TTC 2015. CEUR Workshop Proceedings, vol. 1524, pp. 118–122. CEUR-WS.org (2015)
21. Scheidgen, M.: Generation of large random models for benchmarking. In: BigMDE 2015. CEUR Workshop Proceedings, vol. 1406, pp. 1–10. CEUR-WS.org (2015)
22. Schleich, A.: Skalierbare und effiziente Modellgenerierung mit Tripel-Graph-Grammatiken Master's thesis. TU Darmstadt, Germany (2015)
23. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)