A Catalogue of Optimization Techniques for Triple Graph Grammars

Erhan Leblebici*[1], Anthony Anjorin*[1], Andy Schürr[2]

[1]Technische Universität Darmstadt
Graduate School of Computational Engineering, Germany
{leblebici, anjorin}@gsc.tu-darmstadt.de

[2]Technische Universität Darmstadt
Real-Time Systems Lab, Germany
andy.schuerr@es.tu-darmstadt.de

Abstract: Bidirectional model transformation languages are typically *declarative*, being able to provide unidirectional *operationalizations* from a common specification automatically. Declarative languages have numerous advantages, but ensuring *runtime efficiency*, especially without any knowledge of the underlying transformation engine, is often quite challenging.

Triple Graph Grammars (TGGs) are a prominent example for a completely declarative, bidirectional language and have been successfully used in various application scenarios. Although an optimization phase based on profiling results is often a necessity to meet runtime requirements, there currently exists no systematic classification and evaluation of optimization strategies for TGGs, i.e., the optimization process is typically an ad-hoc process.

In this paper, we investigate the runtime scalability of an exemplary bidirectional model-to-text transformation. While systematically optimizing the implementation, we introduce, classify and apply a series of optimization strategies. We provide in each case a *quantitative* measurement and *qualitative* discussion, establishing a catalogue of current and future optimization techniques for TGGs in particular and declarative rule-based model transformation languages in general.

1 Introduction and Motivation

In a Model Driven Engineering (MDE) context, the *bidirectionality* of model transformations is a crucial requirement for important tasks such as refactoring, evolution, and supporting the co-existence of different engineering artifacts [CFH+09]. Bidirectional model transformation languages are typically *declarative* and enable a high-level specification, which is then suitably *operationalized* for various application scenarios including forward/backward transformations, and propagation of incremental updates.

*Supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt.

Triple Graph Grammars (TGGs) [KLKS10] are a prominent example for a *rule-based* bidirectional language and have been successfully applied in real-world scenarios [GHN10, HGN+13]. Declarative languages such as TGGs have numerous advantages, but meeting *runtime efficiency* requirements can be quite challenging for both *TGG tool developers* and *transformation designers* working with TGGs. Although an optimization phase based on profiling results is, therefore, often a necessity, there currently exists no systematic classification and description of optimization techniques for TGGs in particular and declarative rule-based model transformation languages in general.

In this paper, we investigate the runtime scalability of an exemplary model-to-text round-trip implemented with TGGs. We take our example from the domain of textual *Domain Specific Language* (DSL) development and establish a DSL for describing the persistency layer of a mobile application.

Our contribution is to establish a catalogue for TGG optimization techniques, useful for both TGG tool developers and transformation designers, by:

- Identifying the TGG as a bottleneck in our transformation chain. This is an important result and motivation for our optimization techniques as a typical model-to-text transformation consists of several complex components and it is *a priori* unclear what exactly must be optimized. We use standard, mature parser and unparser technology (ANTLR [Par07] and StringTemplate [Par04], respectively) to provide a realistic comparison. This is presented together with our running example in Sect. 2.
- Suggesting a generic format for presenting current and future optimization techniques for TGGs (Sect. 3).
- Systematically applying a series of diverse optimization techniques to our TGG implementation with a quantitative measurement of improvement in runtime and a qualitative analysis in each case. The optimization techniques are demonstrated in Sect. 4 – 7 using our suggested presentation format.

We conclude with a brief review of related work in Sect. 8 and future work in Sect. 9.

2 Running Example

Our application scenario is taken from the domain of textual DSL development, and requires a model-to-text round-trip which is implemented with TGGs. The goal is to establish a compact textual DSL with which an end-user can describe the persistency layer of a mobile application. The DSL is used to generate Java and Objective-C code for Android and iOS platforms, respectively, from a *common* specification increasing productivity and maintainability. Our round-trip scenario is part of an industrial cooperation, simplified for presentation purposes, and comprises besides bidirectional transformations with TGGs, also unidirectional transformations with *Story Driven Modelling*[1] (SDMs) [FNTZ00]. Using our framework[2], the transformation chain depicted in Fig. 1 can

[1] A unidirectional model transformation language via programmed graph transformation.
[2] http://www.emoflon.org/

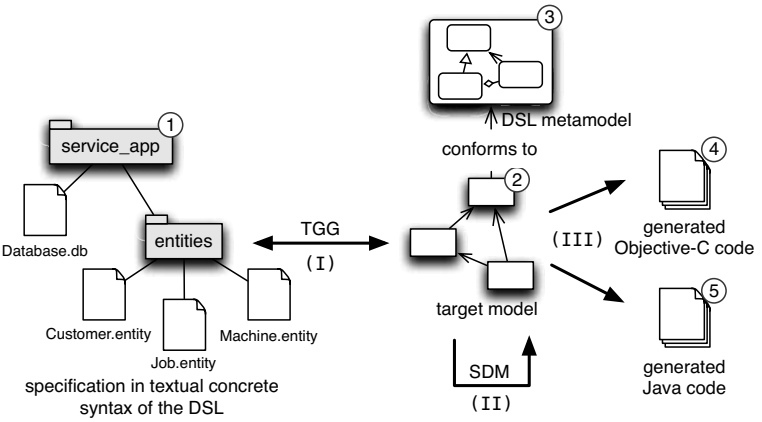

Figure 1: Detailed view of the application scenario with required transformations

be established. The *textual* specification (Fig. 1::1)[3] in form of a file and folder structure is (un)parsed (from) to a single *syntax tree* using industrial standard (un)parsing technology. The specification consists of a root folder (`service_app` in the concrete example), which contains a file `Database.db`, describing the properties of the database (e.g., remote or local) and a subfolder `entities`, containing a file describing each entity to be stored as a table in the database. The target model (Fig. 1::2) is an abstraction of this tree, which is chosen to be maximally suitable for the tasks of validation, refactoring and code generation. It is conform to the target metamodel (Fig. 1::3), which represents the *abstract syntax* of our DSL. After validation and refactorings, the target model is used to generate platform-specific code (Fig. 1::4,5). The whole transformation chain, therefore, consists of three different transformations between languages:

(I) The concrete syntax of the textual DSL is transformed to the target model with a TGG. This transformation is bidirectional, i.e., the target model is obtained from the textual specification via a *forward transformation* and, conversely, can be used to generate the textual specification via a *backward transformation*.

(II) The target model is transformed via SDM. This unidirectional transformation constitutes an *improvement* of the model, e.g., validation, refactorings, and creation of derived attributes and links, e.g., to simplify the task of code generation.

(III) Finally, the platform-specific code artifacts are generated from the (optimized) target model using StringTemplate.

In the following, we focus on the forward and backward transformation in (I), i.e., the TGG specification. The forward transformation is crucial for transforming text to model for platform-specific code generation, whereas the aim of the backward transformation is to keep the text consistent with the model after applying refactorings in (II).

[3]Fig. X::Y refers to label Y in Fig. X

2.1 Implementation with TGGs

TGGs are a rule-based, declarative technique for specifying the simultaneous evolution of two models, together with an additional correspondence model for traceability. The set of rules thus describes a language of triples of related models (graphs) in a generative manner, hence the term *triple graph grammar*. A TGG can be best viewed as a consistency relation on triples of source, correspondence and target models in the following manner: a triple of connected source, correspondence and target models is consistent, if and only if it can be generated with a sequence of rules of the TGG. The advantage of specifying such a high-level consistency relation is that numerous operational transformations can be automatically derived: A *forward transformation* parses a source model and creates a correspondence and target model such that the resulting triple is consistent, while a *backward transformation* parses a target model and creates a correspondence and source model. As a single specification is used for the derivation, the forward and backward transformations are always consistent with each other.

Specifying a TGG starts with declaring the semantic equivalence between the different concepts of the source and target languages. In practice, this is accomplished with a *TGG schema*, a metamodel triple of the source, correspondence and target domains as depicted in Fig. 2. The source metamodel on the left is a simple tree with concepts for `Folders`, `Files` and `Nodes`. The target metamodel consists of an abstract `Database` type, with `Local` and `Remote` as concrete subtypes with additional attributes (not shown explicitly). `Databases` contain arbitrary many `Entities` (e.g., customer, job, machine), which in turn contain `Properties` (e.g., name, address, id). The `extends` relation between two `Entities` is used to enable reuse of entity specifications, i.e., an entity can extend existing entities and, by doing so, combine and extend their properties. The correspondence metamodel in the middle, with types depicted as hexagons to differentiate them visually from source and target types, specifies which source and target elements are related, e.g., that a `File` is semantically equivalent to either a `Database` or an `Entity` (both concepts are specified with individual files in the textual DSL, cf. Fig. 1).

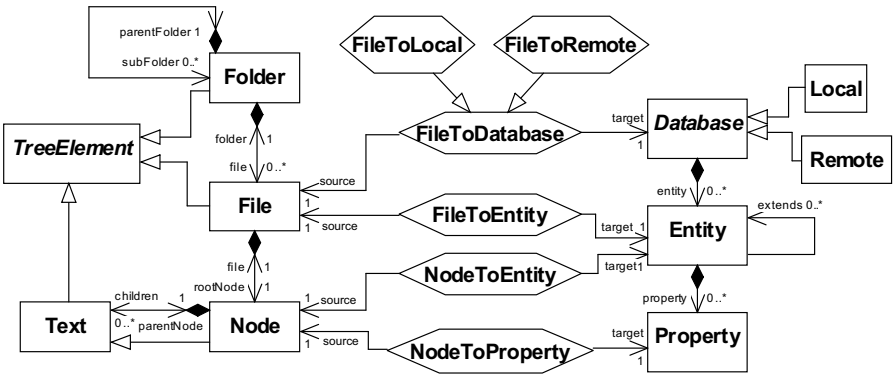

Figure 2: TGG schema for the model-to-text scenario

Declarative *TGG rules* are used to define the actual language of the TGG, i.e., the set of *model triples* consisting of connected source, correspondence and target models, which can be created by using the specified rules. A TGG rule consists of *context elements* (black without any markup) stating the *pre-condition*, which must hold in order to apply the rule, and *created elements* (green with a "++" markup) stating the *post-condition*, which must hold after the rule is applied. TGGs are declarative as the user does not specify *how* this should be achieved (no explicit order of rule application is given).

Figure 3 depicts one of the TGG rules required to implement our scenario. This rule maps the `extends` reference between two `Entities` to a `Node` (named `SUPER_TOKEN`) with a child `Node` representing the name of the extended `Entity`. This rule requires the two `Entities` and their related `Files` as context. The attribute constraint `eq(extendedName.name, entity2.name)` requires that the name of the created node `extendendedName` in the tree be equal to the name of the extended entity `entity2` in the model. The complete set of TGG rules consists of 5 rules.

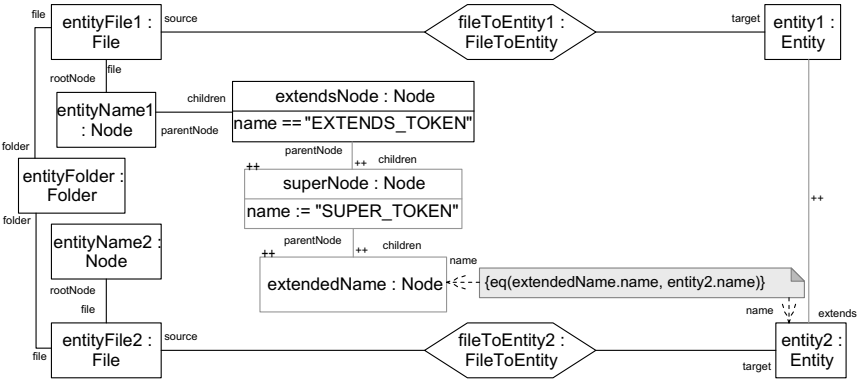

Figure 3: TGG rule creating an `extends` relationship and corresponding tree structure

2.2 Runtime Measurements

Runtime results for this initial implementation are depicted in Fig. 4. All measurements were repeated 10 times (the median is shown in the plot) and executed on Windows 7 x64 with an Intel i5-3550 (3.30 GHz) processor and 8 GB memory. The x-axis shows the number of elements in the target model and ranges from 1000 to 20000. The black vertical dashed line is used here and in the rest of the paper to indicate a change in step size (a change from 1000 to 10000 in Fig. 4). The y-axis shows the time required for each single transformation in seconds using a logarithmic scale. Confidence intervals are not depicted as there was practically no significant difference between measurements.

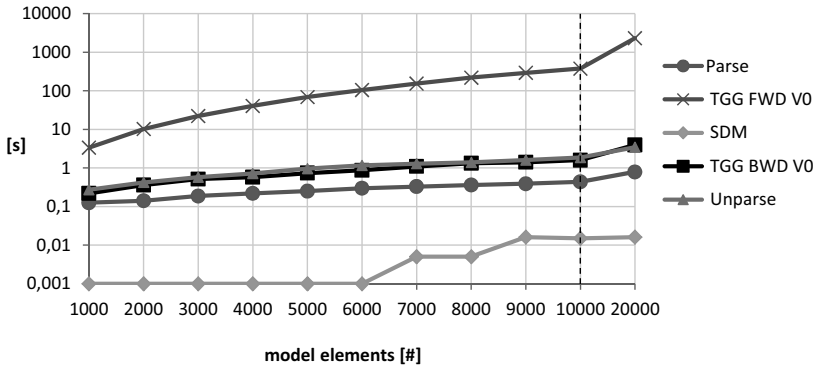

Figure 4: Runtime measurement for the inital implementation with TGGs

Note that the tree structure in the source domain, i.e., the input created by the parser (`Parse`) for the forward transformation (`TGG FWD V0`), contains approximately six times as many elements as the target model in our round-trip scenario. Furthermore, the implemented refactoring (`SDM`) ensures that all entities have a property, which is marked as being unique in the textual syntax (this influences the generated search methods). For example, if the entity `Job` only has a property `name`, which is not necessarily unique, the refactoring creates a new unique property `_id` for `Jobs`. We generate input models randomly ensuring that a fifth of all model elements are `Entities` and that from these `Entities`, a fourth do not have a unique property and are then corrected by the refactoring. This means that 5% of all model elements are manipulated by the SDM transformation. These changes are transformed back to text by the backward transformation (`TGG BWD V0`), recreating the tree from scratch, which is then unparsed to text (`Unparse`).

These results clearly identify the forward transformation as the bottleneck in the transformation chain requiring almost 40 minutes for a model with 20000 elements, compared to less than a second for parsing! Perhaps even more crucial – the forward transformation runs out of memory for models with more than 20000 elements.

Using a Java profiler, the TGG rule in Fig. 3 is identified as the main cause for runtime and memory problems in the forward direction, as all pairwise combinations of `Files` are collected (memory consumption) and checked for cross references (runtime). This is especially problematic due to the higher number of elements in the source domain.

Although the TGG rule is certainly "correct" and appropriate from a declarative point of view, it severly limits the scalability of the derived transformations for large models. In the following, we systematically apply a series of optimization techniques to the TGG to reduce (i) the runtime of mainly the forward transformation (and in some cases also the backward transformation) and (ii) the memory usage to enable a round-trip with models larger than 20000 elements.

3 Generic Structure of an Optimization Technique

In this section, we propose a structure for presenting (TGG) optimization techniques, inspired by [GHJV95]. This is then used consequently in ensuing Sect. 4 – 7 to present a series of concrete optimization techniques. As our catalogue is far from complete, this structure is meant to be used for presenting future techniques and to be extended as necessary, possibly also for specific (TGG) engines.

Each optimization technique is presented in 8 parts:

Name: A descriptive name for the optimization technique used to identify and refer to it. In the following Sect. 4 - 7, the title of the section states the name of the respective optimization technique.

Intent: A brief description of the main idea and goal of the technique.

Motivation (forces): Reasons and arguments why a specific optimization technique is advantageous and particularly effective in the context of TGGs.

Target User: We consider the target user to be either: (i) *A TGG tool developer* who understands and has access to the underlying TGG engine, and is able to implement generic, problem-independent optimizations, or (ii) *A transformation designer*, who is a domain expert in the relevant application field and can implement problem-specific optimizations.

Mechanics: A schematic description of how the technique is to be applied.

Example: An exemplary transformation showing how the technique is applied and giving quantitative measurement results with a qualitative discussion.

Consequences: A discussion of applicability, limitations and scope of the optimization technique. In what cases is the technique particularly effective and when not.

Extensions: A discussion of possible variations and generalizations of the optimization technique, and a comparison with other related techniques that are either alternatives or which can be successfully combined with the presented technique.

4 Progressive, Domain-Driven Determination of Rule Applicability

Intent: The pre-condition of a TGG rule must hold, i.e., all context elements must be present and all attribute constraints must hold, for the rule to be applied. This is also the case for the operational rules (forward, backward) derived automatically from the TGG rule. In case of a forward[4] rule, however, the context in the source domain is extended to cover *all* source elements in the TGG rule. This is because a forward rule *parses* a given source model and *creates* new elements only in the correspondence and target models.

At runtime, a *forward transformation* is realized by determining an appropriate sequence of forward rule applications. This involves the main task of checking if a rule can be applied to translate a certain element in the source model. To improve efficiency, this *rule*

[4] Arguments for backward rules are analogous.

applicability check can be performed *progressively* by checking applicability first in the source domain, and then, only for successful cases, extending the check to all domains.

Motivation (forces): Navigating from input model elements to correspondence and output model elements involves larger patterns and navigating inter-model references, possibly in an inverse direction to their actual navigability. This is a costly operation and is performed unnecessarily if input model elements already violate the source domain-specific pre-conditions of the rule. Filtering out rules as early as possible is, therefore, an important means of reducing runtime.

Target User: *TGG tool developers* who should implement a progressive, domain-driven sequence of rule applicability checks, and *transformation designers* who should support this optimization by preferring domain-specific pre-conditions over cross-domain pre-conditions wherever possible.

Mechanics: According to [KLKS10], an operational rule is said to be *appropriate* if its precondition is satisfied with respect to the input domain and *applicable*, if its complete precondition (over all domains) is satisfied. In our framework, operational rules derived from a TGG specification are decomposed into an *appropriateness check*, an *applicability check*, and a *perform transformation*. In this manner, appropriateness checks are used to filter all rules before applying applicability checks and finally applying the chosen rule with the perform transformation.

In the process of this rule decomposition, the attribute constraints of a TGG rule are analyzed and decomposed analogously, depending on if they can be solved completely in the input domain, or not. Due to local variables used to link constraints, this is a non-trivial analysis and must be conservative in some cases.

Example: The sole attribute constraint in the TGG rule depicted in Fig. 3 requires that the name of the created `Node` (`extendedName`) be equal to the name of the extended `Entity` (`entity2`). In case of a forward transformation, this cross-domain constraint defeats the optimization as *all* possible pairs of `Files` (`entityFile1` and `entityFile2`) fulfill the precondition in the source domain and only the applicability check can choose the correct match via the attribute constraint. In this case, however, the attribute constraint can be reformulated and restricted to the source domain without changing the semantics of the rule. Requiring the name equality already within the tree structure, i.e., with `eq(extendedName.name, entityName2.name)`, would filter all pairs of `Files` that do not reference each other and eliminate invalid matches already with the appropriateness check *before* performing further pattern matching in the applicability check.

This is an example for a small change that has a *considerable* impact on runtime and memory consumption as can be seen from our measurements depicted in Fig. 5. With identical axes and experiment setup as for Fig. 4, the runtime for the initial version of the TGG forward transformation is displayed in the plot as `TGG FWD V0`. The improved version with the reformulated constraint, which makes the TGG rule conducive for the mentioned optimization via decomposition of the rule, is displayed as `TGG FWD V1`. All other curves are optimizations discussed in ensuing sections. The results show that, firstly, the runtime of the transformation has been considerably reduced from about 40 minutes to 37 seconds, and secondly, that the transformation now runs in about 4 minutes for 50000 model

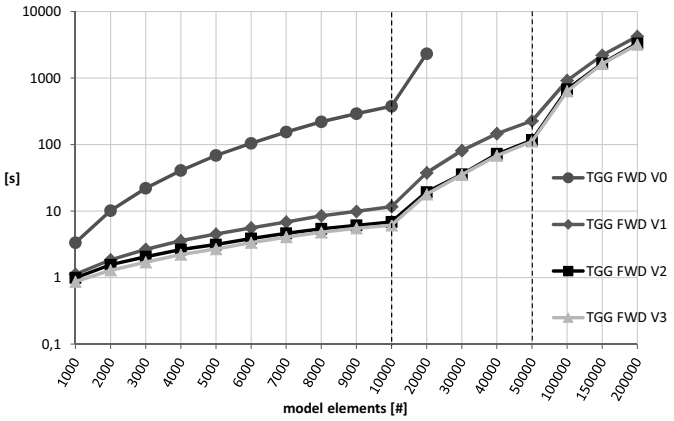

Figure 5: Runtime measurements for the first three optimizations

elements and a bit more than an hour for 200000 model elements. The main reason for this improvement is that far fewer partial matches have to be stored, which reduces memory consumption and avoids memory trashing. As a final remark, note that this change is *problem-specific* and requires domain knowledge of the transformation designer to decide how to reformulate constraints in a semantics-preserving manner.

Consequences: Our results show that it is advantageous to eliminate inappropriate matches in an early phase of rule application. The overall complexity of the underlying TGG algorithm increases, however, due to the intermediate steps and additional rule decomposition. This is unnecessary when only cross-domain dependencies play a role in rule applicability, i.e., it is impossible to reformulate constraints as in our running example. Furthermore, the optimization technique implies that runtime can actually be *improved* if the transformation designer provides additional context information, which is actually redundant from a declarative point of view. This behaviour is neither intuitive nor expected for users without prior experience with constraint solvers.

Extensions: As the pattern matching process is separated into two sequential steps, partial matches from the appropriateness check can be reused in the applicability check to further improve efficiency (tradeoff of memory for runtime performance). Furthermore, user-defined *costs* for attribute constraints together with a unified handling of attribute and graph constraints can be used to determine an optimal search plan for pattern matching.

5 Caching and Indexing of Derived Graph Properties

Intent: The amount of pattern matching required for performing appropriateness checks can be substantially reduced by caching derived graph properties in the input model, e.g., attribute values or relations between nodes.

Motivation (forces): Caching of derived information in a model is, in general, highly non-trivial as certain but not all changes to the model require updating the cache. As a

TGG forward/backward transformation, however, does not change the input model, cached information is valid during the entire transformation and does not require complex and potentially costly bookkeeping.

Target User: The TGG tool developer who must provide a suitable caching/indexing mechanism in the tool, and the transformation designer who has the required domain knowledge about which properties to cache/index and how this should be accomplished.

Mechanics: Our framework supports virtual links between nodes in TGG rules, referred to as *binding expressions*. Binding expressions represent auxiliary methods that possibly access a global cache and return candidates for the required model elements. Stubs for these methods are automatically generated by the tool and must be implemented by the transformation designer with SDMs or plain Java.

Example: The TGG rule from our previous optimization (`TGG FWD V1` in Fig. 5) still collects all pairs of `Files` and filters them using the attribute constraint. This can be made more efficient by caching all root `Nodes` of all `Files` in an initial iteration and using this cache (e.g., a hashtable mapping names of root `Nodes` to the actual `Nodes`) as an index when searching for a root `Node` with a particular name.

Fig. 6 depicts the adjusted TGG rule, which now uses a binding expression (dashed arrow) from `extendedName` to `entityName2`. The latter is depicted with a bold frame to indicate that it is now *bound* via an auxiliary method that takes `extendedName` as parameter (recall that `extendedName` and `entityName2` must have the same name). Cross-references are now found in constant time with the help of a global cache. The forward transformation with this new version is displayed in Fig. 5 as `TGG FWD V2`. The results show that a moderate speed-up with factor of up to 2 can be achieved as compared to our first optimization `TGG FWD V1`. Note that, although we apply the optimizations in the order we present them, `TGG FWD V2` does not profit from the first optimization `TGG FWD V1` as the attribute constraint is no longer used for filtering in the forward direction.

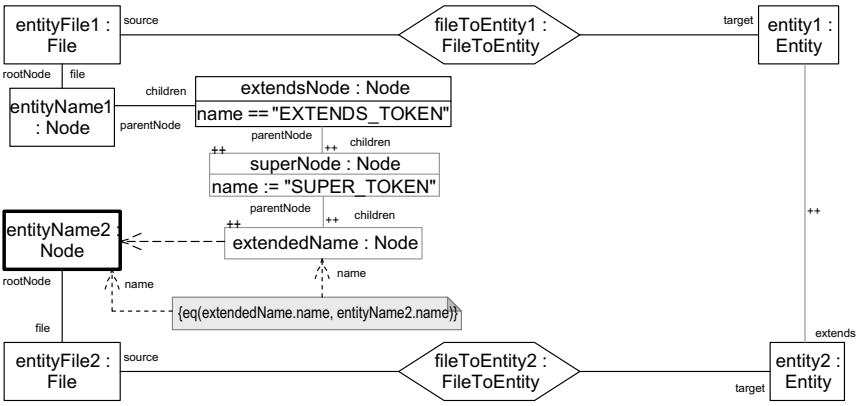

Figure 6: The new version of the TGG rule using binding expressions

Compared to `TGG FWD V1`, the runtime improvement is especially noticeable for mid-sized models (10000 to 50000 model elements). Apparently, the pairwise matching of `Files` is not the critical factor for smaller models. In case of larger models, other issues such as pattern matching, more specifically navigating to other domains, become more time consuming and outweigh the positive effects of the optimization.

Consequences: Determining what to cache requires problem-specific knowledge in general. In our example for instance, the speed-up provided by the cache depends on the number of `extends` relations between entities. In the worst case scenario, using the cache for very large models can actually be *slower* if exactly one extends relations is used, i.e., the time is spent for lazily filling the cache, which is only used once. It is also important to remember that a cache represents a trade-off of memory for runtime.

A further issue is that TGG rules get more abstract the more binding expressions are used. Details are hidden behind virtual links that do not really describe how the search is performed. This can reduce the overall readability of the rules if used excessively.

Extensions: Similar to attribute constraints as discussed in the previous section, binding expressions can be weighted with a user-defined cost function and uniformly handled as constraints by the search plan generator. For straightforward cases, the TGG tool could determine if an index should be built over a certain attribute value (e.g., name in our running example), creating and implementing an appropriate binding expression automatically.

6 Static Analysis of TGG Rules

Intent: A *static analysis* of the TGG rules can be performed to extract information about the dependencies and structure of the rules. This can be exploited to improve performance.

Motivation (forces): Results and techniques from the mature field of graph transformations can be used to analyze the structure of and dependencies between TGG rules.

Target User: TGG tool developers who must provide a static analysis at *compile time* to generate additional artifacts to support the transformation process.

Mechanics: A concrete technique is to obtain a *global view of dependencies* between rules and types via a precedence analysis [LAVS12a], which can be used to optimize the order in which elements of the source model are processed. The goal is to avoid arbitrary choices by the algorithm and a consequent recursion stack.

Another strategy is to construct *rule filter tables* using the types of created elements in TGG rules and attribute constraints that compare attributes of nodes to constant values. These tables can be used at runtime to filter *rule candidates* in constant time *before* running the appropriateness checks.

Example: Applying a rule filter table for our running example, we were able to speed-up the translation by about 10% as depicted in Fig. 5 (`TGG FWD V3`). Although this is moderate compared to the previous optimizations, note that this optimization is "free-of-charge" at transformation time and is applicable for all TGGs, i.e., is a generic optimization

and is not problem-specific. Furthermore, depending on the concrete example, the speed-up obtained by using filter tables can be much more, especially for rules with large patterns and models with few types and many attribute constraints.

Consequences: Generating additional information from a TGG specification prolongs the compilation process, hindering an agile development/test/debug workflow especially for large TGGs. A more critical issue is that the table lookup must be much faster than the saved effort of pattern matching. Especially if the filters are not able to exclude any rules, this might actually slow down the translation process if too much computation is involved.

Extensions: The goal of reducing the effort of checking rule applicability for a large number of rules is closely related to incrementally updating matches in the context of *incremental pattern matching*. Existing results and techniques [VD13] show that a *Rete network* can be constructed from all rules and used to avoid redundant pattern matching completely. This can be seen as a generalization of our rule filter tables and is future work.

A related approach is using *model sensitive search plans* [VDWS13] to exploit information collected from the models (especially the input model) during the transformation process.

A final approach is using static analysis techniques from graph transformations to parallelize (i) the rule applicability checks, and (ii) independent transformation steps [IM12].

7 Incremental Updates

Intent: In case of an existing triple of relatively large source, correspondence and target models, a "small" update to the source model (target analogously) typically affects only a "small" subset of the correspondence and target models. Propagating these changes by incrementally adapting the existing models can be potentially much more efficient than recreating the models from scratch.

Motivation (forces): The consistency relation described by a TGG can be used not only to derive forward and backward transformations, which create output models from scratch, but also to realize *incremental updates*. Changes to the source model can be propagated incrementally to an existing target model in this case. Only the relevant parts of the triple are traversed and manipulated as required to restore consistency.

Target User: TGG tool developers who must implement an appropriate incremental TGG algorithm, and the transformation designer who must decide if an incremental propagation of changes is necessary and feasible in a certain application scenario.

Mechanics: Given the changes applied to the input model, a TGG incremental algorithm has to (i) compute the complete set of input model elements S that must be re-transformed as they depend on, e.g., deleted elements, (ii) revoke the rule applications used to transform S, i.e., delete related elements in the correspondence and output domains, and (iii) re-transform all elements in S by invoking the forward transformation on the existing triple.

This is typically achieved by exploiting additional information such as dependencies between the correspondence links, or transformation protocols recorded during the transfor-

mation. The crucial point is that all steps must be independent of the total size of the models involved and only depend on the size of the change and transitively affected elements. The latter is typically a much smaller set than the total number of elements.

In practice, a critical component for realizing incremental updates is a change recognition mechanism, which can either be a *model diff* that compares two versions of a model, or an *online change detector* that records all changes in a notification-based environment.

Example: With the same setup and environment as for the measurement in Fig. 5, the runtime measurement results for our incremental TGG algorithm [LAVS12b] are depicted in Fig. 7. In the plot on the left, the refactoring described in Sect. 2 (recall that 5% of all model elements are manipulated) was performed directly in the textual syntax and propagated with our incremental forward TGG transformation `TGG FWD V4`. To provide a comparison, the optimized *batch* (non-incremental) forward transformation is displayed in the plot as `TGG FWD V3`. To simulate a situation where offline change recognition is necessary, the textual syntax was changed offline and re-parsed to yield a new tree with the refactoring applied. This is then compared to the old version of the tree via a *diff* mechanism, to identify the differences which are then propagated incrementally to the existing correspondence and target models. To show the actual cost of offline change recognition, the sum of diff *and* incremental algorithm is displayed in the plot as `Diff + TGG FWD V4`. Note that we had to implement a specialized tree diff as generic model diffs such as *EMF compare* were much too inefficient. Our results show that the incremental algorithm is considerably faster than the optimized batch transformation with less than half a second compared to almost three seconds for 5000 model elements.[5] A speed-up factor of about 7 - 13 remains constant for all model sizes. For a model size of up to 10000, the total cost of diff plus synchronization is less than for the batch transformation with 1.3s as compared to 2.6s for 5000 model elements. This, however, reduces progressively and the advantage of the incremental algorithm is defeated by the cost of calculating the changes with the diff algorithm. This is to be expected as the tree diff is not incremental and does not exploit information from previous runs.

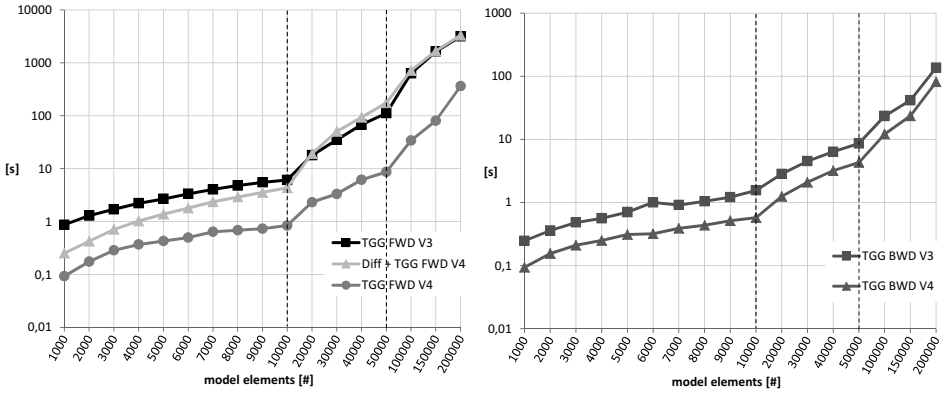

Figure 7: Runtime measurements for the incremental TGG implementation

[5] Recall that the elements in the tree are roughly 6 times as many as in the model.

The plot on the right compares the batch and incremental modes for the backward transformation. In this case, the refactoring is applied with SDMs on the model and propagated back to the tree and textual syntax using our incremental TGG algorithm `TGG BWD V4`. To simulate a situation where an online change recognition is possible, the SDM refactoring transformation additionally creates the appropriate *deltas* for the incremental propagation, i.e., there is no need for a model diff. The backward batch transformation is displayed in the plot as `TGG BWD V3`. The results show that both modes are much more efficient in the backward direction. This is to be expected as the model is smaller, better typed, and suitably connected, i.e., it is easier to create the larger, weakly typed tree structure than to parse it. Nonetheless, the incremental algorithm enables a moderate speed-up with 0,5s compared to 1,5s (a factor of 3) for 10000 model elements and slightly less for larger models with 81s compared to 137s (a factor of 1.7) for 200000 model elements.

Consequences: Although online change recognition is more efficient and less error-prone, an application scenario might require offline changes to be handled.[6] In such a case, naïve diff algorithms might not scale and it can be quite challenging to deal with change recognition correctly and efficiently. We have focussed with our experiments on efficiency arguments for incremental updates. A further, possibly even more important argument is *infomation loss*. In many application scenarios, there is irrelevant information in one or both domains, which *cannot* be reconstructed from the model in the other domain. In such cases, updates *must* be handled incrementally to ensure correct results.

Extensions: Handling a set of *concurrent* changes, i.e., changes to both source and target models requires conflict resolution and is currently not supported by our incremental TGG algorithm. This is, however, often the case in practice and is important future work. Finally, our current incremental TGG algorithm is rather conservative and can be improved by further analyses to accurately determine the exact set of transitively affected elements.

8 Related Work

There exist various bidirectional transformation languages. For a survey and a detailed discussion, we refer to [Ste08, CFH+09]. In this context, efficiency has, however, not yet received as much attention as, e.g., formal properties, although it is crucial for the practical feasibility of a bidirectional language as certain minimal runtime requirements often must be met. Depending on the application scenario, this might even outweigh all other advantages that a bidirectional, declarative language has to offer. With our contribution we try to close this gap for TGGs and demonstrate that it is indeed possible to optimize the derived transformations as required. Our results are TGG specific but the core ideas and techniques can be transferred to other (especially rule-based) languages.

Giese et. al [GH09] and Lauder et. al [LAVS12b] both lay emphasis on efficiency as the main argument for an incremental TGG algorithm. In practice, however, especially when changes have transitive effects (e.g., a root element is changed), or change recognition is particularly difficult, supporting incrementality is not the sole way of improving efficiency.

[6] This is currently the case in an ongoing industrial application.

In such cases, other optimization techniques as presented in this paper are necessary.

The numerous optimization techniques in the mature field of graph transformations mostly concern the pattern matching process [VAS04, VDWS13, GSR05], and discuss diverse strategies for optimal search plan generation. These results have served as a major source of inspiration, and we have already adapted ideas that are especially effective for TGGs.

Finally, the parallel execution of independent transformation steps as proposed in [IM12] is an optimization technique that is currently rarely used in practice, but can be potentially very powerful for TGGs when combined with a suitable dependency analysis.

9 Conclusion and Future Work

In this paper, we have proposed a format for presenting current and future TGG optimization techniques. We have used this format to discuss four concrete optimization techniques, demonstrating each of them on our running example, which is part of an industrial application where a textual DSL requiring a model-to-text round-trip is used to specify the persistency layer of mobile applications for different target platforms. Our runtime measurements show that an optimization factor of about 300 (complete transformation) to 500 (incremental with 5% change size) can be achieved in the bottleneck of our scenario, namely the forward, i.e., text-to-model, transformation with TGGs. More crucially, the optimizations have enabled round-trips with larger model sizes by reducing memory consumption. Our proposed catalogue of optimization techniques should serve as a guideline for both TGG tool developers and transformation designers when efficiency issues threaten to outweigh the advantages of TGGs.

As future work, we plan to implement and investigate the various extensions to each technique as already discussed in the paper. To improve the validity of our measurement results, we plan to establish a *transformation zoo* of diverse TGG examples, which can serve as a benchmark that covers important aspects of various application scenarios.

References

[CFH+09] Krzysztof Czarnecki, John Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In Richard F. Paige, editor, *Proc. of ICMT 2009*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. of TAGT 1998*, volume 1764 of *LNCS*, pages 157–167. Springer, 2000.

[GH09] Holger Giese and Stephan Hildebrandt. Efficient Model Synchronization of Large-Scale Models. Technical report, Hasso-Plattner Institute, University of Potsdam, 2009.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1995.

[GHN10] Holger Giese, Stephan Hildebrandt, and Stefan Neumann. Model Synchronization at Work : Keeping SysML and AUTOSAR Models Consistent. In Andy Schürr, Claus Lewerentz, Gregor Engels, Wilhelm Schäfer, and Bernhard Westfechtel, editors, *Festschrift Nagl*, pages 555–579. Springer, 2010.

[GSR05] Leif Geiger, Christian Schneider, and Carsten Reckord. Template- and Modelbased Code Generation for MDA-Tools. In *Proc. of the 3rd International Fujaba Days*, pages 57–62, 2005.

[HGN+13] Frank Hermann, Susann Gottmann, Nico Nachtigall, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, and Thomas Engel. On an Automated Translation of Satellite Procedures Using Triple Graph Grammars. In Keith Duddy and Gerti Kappel, editors, *Proc. of ICMT 2013*, volume 7909 of *LNCS*, pages 50–51. Springer, 2013.

[IM12] Gábor Imre and Gergely Mezei. Parallel Graph Transformations on Multicore Systems. In Victor Pankratius and Michael Philippsen, editors, *Proc. of MSEPT 2012*, volume 7303 of *LNCS*, pages 86–89. Springer, 2012.

[KLKS10] Felix Klar, Marius Lauder, Alexander Königs, and Andy Schürr. Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In Andy Schürr, Claus Lewerentz, Gregor Engels, Wilhelm Schäfer, and Bernhard Westfechtel, editors, *Festschrift Nagl*, volume 5765 of *LNCS*, pages 141–174. Springer, 2010.

[LAVS12a] Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. Bidirectional Model Transformation with Precedence Triple Graph Grammars. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitris Kolovos, editors, *Proc. of ECMFA 2012*, volume 7349 of *LNCS*, pages 287–302. Springer, 2012.

[LAVS12b] Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. Efficient Model Synchronization with Precedence Triple Graph Grammars. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. of ICGT 2012*, volume 7562 of *LNCS*, pages 401–415. Springer, 2012.

[Par04] Terence John Parr. Enforcing Strict Model-View Separation in Template Engines. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proc. of WWW 2004*, pages 224 – 233. ACM, 2004.

[Par07] Terence John Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007.

[Ste08] Perdita Stevens. A Landscape of Bidirectional Model Transformations. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Proc. of GTTSE 2008*, volume 5235 of *LNCS*, pages 408–424. Springer, 2008.

[VAS04] Attila Vizhanyo, Aditya Agrawal, and Feng Shi. Towards Generation of Efficient Transformations. In Gabor Karsai and Eelco Visser, editors, *Proc. of GPCE 2004*, volume 3286 of *LNCS*, pages 298–316. Springer, 2004.

[VD13] Gergely Varró and Frederik Deckwerth. A Rete Network Construction Algorithm for Incremental Pattern Matching. In Keith Duddy and Gerti Kappel, editors, *Proc. of ICMT 2013*, volume 7909 of *LNCS*, pages 125–140. Springer, 2013.

[VDWS13] Gergely Varró, Frederik Deckwerth, Martin Wieber, and Andy Schürr. An Algorithm for Generating Model-Sensitive Search Plans for EMF Models. In Zhenjiang Hu and Juan de Lara, editors, *Proc. of ICMT 2012*, volume 7307 of *LNCS*, pages 224–239. Springer, 2013.