Bidirectional Model Transformation with Precedence Triple Graph Grammars

Marius Lauder*, Anthony Anjorin*, Gergely Varró**, and Andy Schürr

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

Abstract. Triple Graph Grammars (TGGs) are a rule-based technique with a formal background for specifying bidirectional model transformation. In practical scenarios, the unidirectional rules needed for the forward and backward transformations are automatically derived from the TGG rules in the specification, and the overall transformation process is governed by a control algorithm. Current implementations either have a worst case exponential runtime complexity, based on the number of elements to be processed, or pose such strong restrictions on the class of supported TGGs that practical real-world applications become infeasible. This paper, therefore, introduces a new class of TGGs together with a control algorithm that drops a number of practice-relevant restrictions on TGG rules and still has a polynomial runtime complexity.

Keywords: triple graph grammars, control algorithm of unidirectional transformations, node precedence analysis, rule dependency analysis

1 Introduction

The paradigm of Model-Driven Engineering (MDE) has established itself as a promising means of coping with the increasing complexity of modern software systems and, in this context, *model transformation* plays a central role [3]. As industrial applications require reliability and efficiency, the need for formal frameworks that guarantee useful properties of model transformation arises. This is especially the case for *bidirectional* model transformation, where defining a precise semantics for the automatic manipulation and synchronization of models with a corresponding efficient tool support is quite challenging [4]. Amongst the numerous bidirectional model transformation approaches surveyed in [18], the concept of *Triple Graph Grammars (TGGs)* features not only solid formal foundations [5,12] but also various tool implementations [7,11,12].

TGGs [16] provide a declarative, rule-based means of specifying the consistency of source and target models in their respective domains, and tracking

* Supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt.
** Supported by the Postdoctoral Fellowship of the Alexander von Humboldt Foundation and associated with the Center for Advanced Security Research Darmstadt.

inter-domain relationships between model elements explicitly by automatically maintaining a correspondence model. Although TGGs describe how *triples* consisting of source, correspondence, and target models are simultaneously derived, most practical software engineering scenarios require that source or target models already exist and that the models in the correspondence and the opposite domain be consistently constructed by a unidirectional forward or backward transformation. As a consequence, TGG tools that support bidirectional model transformation (i) rely on unidirectional forward and backward operational rules, automatically derived from a single TGG specification, as basic transformation steps, and (ii) use an algorithm that controls which rule is to be applied on which part of the input graph. As a TGG rule in the specification might require *context elements* created by another TGG rule, the control algorithm must consider these *precedences/dependencies* at runtime when (a) determining the order in which graph nodes can be processed, and (b) selecting the rule to be applied.

In this paper, we introduce a *node precedence analysis* to provide a global view on the dependencies in the source graph and to guide the transformation process. Additionally, we combine the node precedence analysis with *a rule dependency analysis* to support the control algorithm in determining the node processing order and selecting the next applicable rule. This approach can now exploit global dependency information, and perform an iterative, top-down resolution which is more expressive (can handle a larger class of TGGs) and fits better into future incremental scenarios. Finally, we prove that the improved control algorithm is still correct, complete, and polynomial.

Section 2 introduces fundamental definitions using our running example while Sect. 3 discusses existing TGG batch algorithms. Sect. 4 presents our rule dependency and node precedence analysis, used by the TGG batch algorithm presented in Sect. 5. Finally, Sect. 6 gives a broader overview of related *bidirectional* approaches and Sect. 7 concludes with a summary and future work.

2 Fundamentals and Running Example

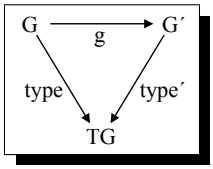
In this section, all concepts required to formalize and present our contribution are introduced and explained using our running example.

2.1 Type Graphs, Typed Graphs and Triples

We introduce the concept of a *graphs*, and formalize *models* as *typed graphs*.

Definition 1 (Graph and Graph Morphism). *A* graph $G = (V, E, s, t)$ *consists of finite sets* V *of nodes, and* E *of edges, and two functions* $s, t : E \to V$ *that assign each edge source and target nodes. A* graph morphism $h : G \to G'$*, with* $G' = (V', E', s', t')$*, is a pair of functions* $h := (h_V, h_E)$ *where* $h_V : V \to V'$*,* $h_E : E \to E'$ *and* $\forall e \in E : h_V(s(e)) = s'(h_E(e)) \ \wedge \ h_V(t(e)) = t'(h_E(e))$.

Definition 2 (Typed Graph and Typed Graph Morphisms).
A type graph *is a graph* $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$.
A typed graph $(G, type)$ *consists of a graph* G *together with a graph morphism* type*:* $G \to TG$.
Given typed graphs $(G, type)$ *and* $(G', type')$, $g : G \to G'$ *is a* typed graph morphism *iff the diagram commutes.*



These concepts can be lifted in a straightforward manner to *triples* of connected graphs denoted as $G = G_S \xleftarrow{h_S} G_C \xrightarrow{h_T} G_T$ as shown by [6,12]. In the following, we work with *typed graph triples* and corresponding morphisms.

Example. Our running example specifies the integration of *company structures* and corresponding *IT structures*. The *TGG schema* (Fig. 1) is the type graph triple for our running example. The *source domain* is described by a type graph for company structures: A `Company` consists of a `CEO`, `Employees` and `Admins`. In the *target domain*, an IT structure (`IT`) provides `PCs` and `Laptops` in `Networks` controlled by a `Router`. The *correspondence domain* specifies valid links between elements in the different domains.

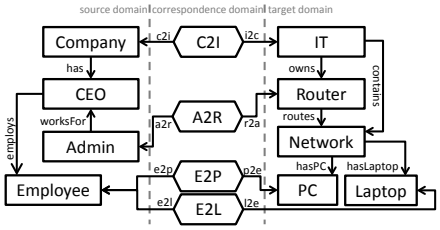


Fig. 1. TGG Schema for the integration of a company with its IT structure

A schema conform (typed graph) triple is depicted in Fig. 2. The company `ES` has a CEO named `Andy` for whom administrator `Ingo` works. Additionally, `Andy` employs `Tony` and `Marius`. The corresponding IT structure `ES-IT` consists of a router `WP53` for the network `ES-LAN` with a PC `PC65` and a laptop `X200`.

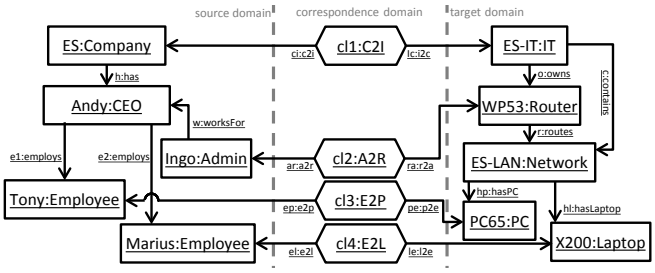
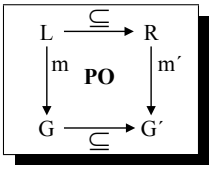


Fig. 2. A TGG schema conform triple

2.2 Triple Graph Grammars and Rules

The simultaneous evolution of typed graph triples such as our example triple (Fig. 2) can be described by a *triple graph grammar* consisting of *transformation rules*. This is formalized in the following definitions.

Definition 3 (Graph Triple Rewriting for Monotonic Creating Rules). *A* monotonic creating rule $r := (L, R)$, *is a pair of typed graph triples such that* $L \subseteq R$. *A rule* r rewrites *(via adding elements) a graph triple* G *into a graph triple* G' *via a match* $m : L \to G$, *denoted as* $G \overset{r@m}{\rightsquigarrow} G'$, *iff* $m' : R \to G'$ *is defined by building the* pushout G' *as denoted in the diagram.*



Elements in L denote the precondition of a rule and are referred to as *context elements*, while elements in $R \setminus L$ are referred to as *created elements*.

Definition 4 (Triple Graph Grammar). *A* triple graph grammar $TGG := (TG, \mathcal{R})$ *consists of a type graph triple* TG *and a finite set* $\mathcal{R}$ *of monotonic creating rules. The generated language (*$G_\emptyset$ *denotes the empty graph triple) is* $\mathcal{L}(TGG) := \{G \mid \exists\, r_1, r_2, \ldots, r_n \in \mathcal{R} :\ G_\emptyset \overset{r_1@m_1}{\rightsquigarrow} G_1 \overset{r_2@m_2}{\rightsquigarrow} \ldots \overset{r_n@m_n}{\rightsquigarrow} G_n = G\}$.

Example. The rules depicted in Fig. 3 build up an integrated company and IT structure simultaneously. Rule (a) creates the root elements of the models (a `Company` with a `CEO` and a corresponding `IT`), while Rule (b) appends additional elements (an `Admin` and a corresponding `Router` with the controlled `Network`). Rules (c) and (d) extend the models with an `Employee`, who can choose a `PC` or a `Laptop`. We use a concise notation by merging L and R of a rule, depicting context elements in black without any markup, and created elements in green with a "++" markup.

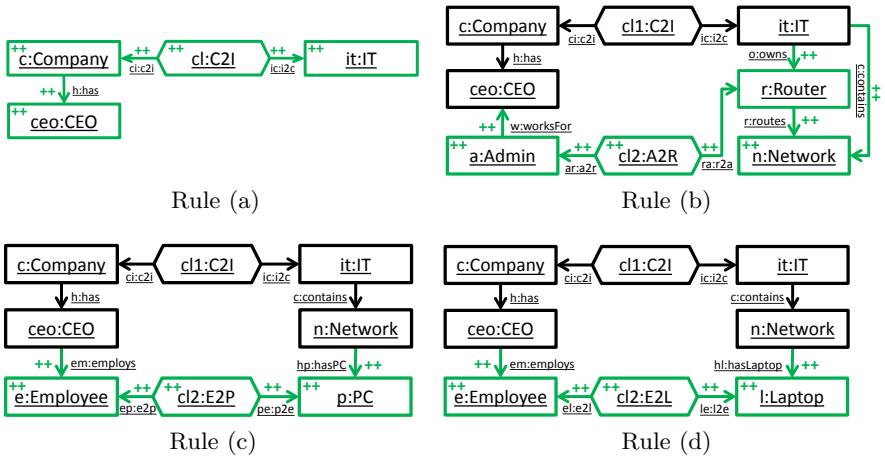


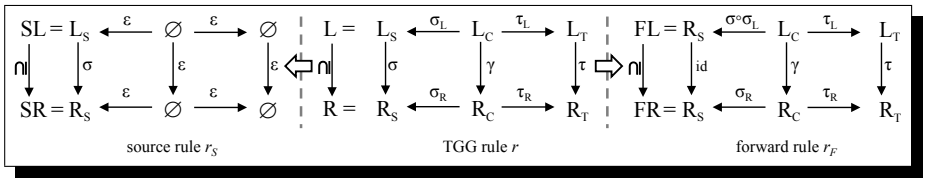
Fig. 3. Rules (a)–(d) for the integration

2.3 Derived Operational Rules

The real potential of TGGs as a bidirectional transformation language lies in the automatic derivation of *operational rules*. Such operational rules can be used to transform a given source domain model to produce a corresponding target domain model and vice versa. Although we focus in the following sections only on a forward transformation, all concepts and arguments are symmetric and can be applied analogously for the case of a backward transformation.

It has been proven by [5,16] that a sequence of TGG rules, which describes a simultaneous evolution, can be uniquely decomposed into (and conversely composed from) a sequence of *source rules* that only evolve the source model and *forward rules* that retain the source model and evolve the correspondence and target models. These operational rules serve as the building blocks used by a control algorithm for unidirectional forward and backward transformation.

Definition 5 (Derived Operational Rules). *Given a* $TGG = (TG, \mathcal{R})$ *and a rule* $r = (L, R) \in \mathcal{R}$*, a* source rule $r_S = (SL, SR)$ *and a* forward rule $r_F = (FL, FR)$ *can be derived according to the following diagram:*



Example. From Rule (c) of our running example (Fig. 3), the operational rules r_S and r_F depicted in Fig. 4 can be derived. The source rule extends the source graph by adding an `Employee` to an existing `CEO` in a `Company`, while the forward rule r_F *transforms* an existing `Employee` of a `CEO` by *creating* a new `E2P` link and a `PC` in the corresponding `Network`.

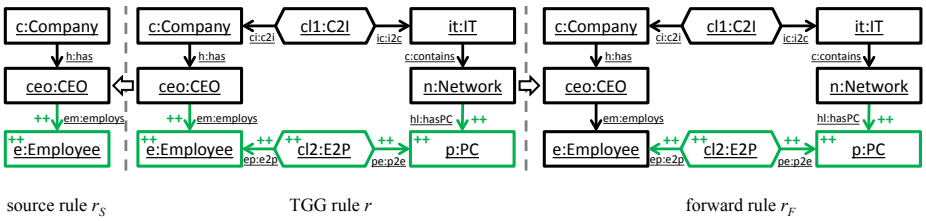


Fig. 4. Source and forward rules derived from Rule (c)

3 Related Work on TGG Control Algorithms

Constructing forward (and conversely backward) transformations from operational rules requires a *control algorithm* that is able to determine a sequence of forward rules to be applied to a given source graph. The challenge is to specify a control algorithm that is correct (only consistent graph triples are produced), complete (all consistent triples, which can be derived from a source or a target graph, can actually be produced), efficient (runtime complexity scales polynomially with the number of nodes to be processed), and still expressive enough for real-world applications. To better understand this challenge, we discuss how existing algorithms handle the source graph of our example triple (Fig. 2).

(I) Bottom-Up, Context-Driven and Recursive: An established strategy is to transform elements in a bottom-up context-driven manner, i.e., to start with a random node and check if all context nodes (dependencies) are already transformed *before* the selected initial node can be transformed. If a context node is not yet transformed, the algorithm transforms it, by recursively checking and transforming its context. Context-driven algorithms always start their transformation process with an arbitrarily selected node, without "knowing" if this was a good choice, i.e., if the node can be transformed immediately or if the input model as a whole is even valid. Such algorithms are correct, but, in general, have problems with completeness due to wrong *local* decisions.

(I.a) Backtracking: A simple backtracking strategy could be employed to cope with wrong local decisions. For our example, a first iteration over all nodes would determine that only `ES` together with `Andy` can be transformed by applying Rule (a). In a second iteration the algorithm would determine again in a trial and error manner that only `Ingo` can be transformed next with Rule (b), as neither `Tony` nor `Marius` can be transformed using Rule (c) or (d) (a `Network` is missing in the opposite domain). Finally, `Tony` and `Marius` can be transformed. This algorithm is correct and complete as shown in [5,16] but has exponential runtime and is, therefore, impractical for real-world applications.

It is, however, possible to guarantee polynomial runtime of the context-driven recursion strategy by restricting the class of supported TGGs appropriately as in case of the following approaches.

(I.b) Functional Behavior: Demanding *functional behavior* [7,9] guarantees that the algorithm can choose freely between applicable rules at every decision point and will always get the *same result* without backtracking. Although functional behavior might be suitable for fully automatic integrations, our experience with industrial partners [14,15] shows that user interaction or similar guidance (e.g., configuration files) of the integration process is required and leads naturally to non-functional sets of rules with certain degrees of freedom [13,14,15]. Please note that our running example is clearly non-functional due to Rules (c) and (d), which can be applied to the same elements on the source side, but create different elements on the target side. Therefore, depending on the choice of rule applications, *different* target graphs are possible with our running example. Demanding functional behavior is a strong restriction that reduces the expressiveness and suitability of TGGs for real-world applications [12,17]. Nev-

ertheless, such a strategy has polynomial runtime and its applicability can be enforced statically via critical pair analysis [6].

(I.c) Local Completeness: Algorithms that allow a non-functional set of rules to handle a larger set of scenarios exploit the explicit traceability to cope with non-determinism and non-bijectivity [19], while still guaranteeing completeness for a certain class of TGGs. Hence, [12] demands *local completeness*, i.e., that a local decision between rules that can transform the current node *cannot* lead to a dead-end. This means that a local choice (which can be influenced by the user or some other means) might actually result in *different* output graphs, which are, however, always consistent, i.e., in the defined language of the TGG ($\mathcal{L}(TGG)$). For our running example, we could start with an arbitrary node, e.g., `Ingo`. According to Rule (b), a `CEO` and a `Company` are required as context and Rule (a) will thus be applied to `ES` and `Andy`. After processing `Ingo`, `Tony` and `Marius` can be transformed in an arbitrary order, each time making a local choice if a `PC` (Rule (c)) or `Laptop` (Rule (d)) is to be created. Furthermore, a *dangling edge check* is introduced in [12] to further enlarge the class of supported TGGs via a look-ahead to prevent wrong local decisions that would lead to "dangling" edges that can no longer be transformed. Note that our running example is *not* local complete, as it cannot be decided whether an `Admin` or an `Employee` should be transformed first (Rules (c) and (d) demand an element on the *target* side that can only be created by Rule (b)). For this reason, the algorithm might fail if it decides to start with one of the `Employees`. In this case, Rules (c) and (d) would state that `ES` and `Andy` are required as context and have to be transformed first. This is, however, insufficient as a `Network` must be present in the target domain as well. This context-driven approach fails here as transforming `ES` and `Andy` with Rule (a) *does not* guarantee that the employees `Marius` and `Tony` can be transformed. The problem here is that context-driven algorithms only regard the given input graph for controlling the rule application and do not consider *cross-domain context dependencies* such as `Network` in this case.

(II) Top-Down and Iterative: In contrast to context-driven recursive strategies, which lack a *global view* on the overall dependencies and seem to be unsuitable for an *incremental* synchronization scenario, algorithms can operate in a top-down iterative manner exploiting a certain global view on the whole input graph instead of arbitrarily choosing a node to be transformed.

(II.a) Correspondence-Driven: The algorithm presented by [11] requires that all TGG rules demand and create at least one correspondence link, i.e., a hierarchy of correspondence links must be built up during the transformation. The correspondence model can be used to store dependencies between links in this case and is interpreted as a directed acyclic graph, which is used to drive and control the transformation. This algorithm is both batch *and* incremental but it is unclear from [11] for which class of TGGs completeness can be ensured.

(II.b) Precedence-Driven: A precedence-driven strategy defines and uses a partial order of nodes in the source graph according to their *precedence*, i.e., the sorting guarantees that the nodes can only be transformed in a sequence that is compatible with the partial order.

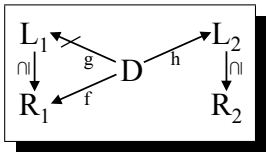
4 Rule Dependency and Precedence Analysis for TGGs

In this section, we present a node precedence analysis that provides a partial order required for a precedence-driven strategy, together with a rule dependency analysis that partially solves the problem of cross-domain context dependencies caused by context elements in the domain under construction.

4.1 Rule Dependency Analysis

To handle cross-domain context dependencies, we utilize the concept of *sequential independence* as introduced by [6], to statically determine which rules depend on other rules. The intuition is that a rule r_2 depends on another rule r_1, if r_1 creates elements that r_2 requires as context.

Definition 6 (Rule Dependency Relation $\lessdot_R$). *Given rules $r_1 = (L_1, R_1)$ and $r_2 = (L_2, R_2)$, r_2 is sequentially dependent on r_1 iff a graph D and morphisms f, h exist, such that there exists no morphism g as depicted to the right, i.e., at least one element required by r_2 (an element in L_2), is created by r_1 (this element is in R_1 but not in L_1).*



The precedence relation $\lessdot_R \subseteq \mathcal{R} \times \mathcal{R}$ is defined for a given TGG as follows:
$r_1 \lessdot_R r_2 \Leftrightarrow r_2$ *is sequentially dependent on* r_1.

In practice, $\lessdot_R$ can be calculated statically by determining all possible intersections of R_1 and L_2. If at least one element in an intersection is not in L_1 then r_2 is sequentially dependent on r_1 (i.e., $r_1 \lessdot_R r_2$).

Example. For the TGG rules of our running example (Fig. 3), the following pairs of rules constitute $\lessdot_R$: *Rule* $(a) \lessdot_R$ *Rule* (b), *Rule* $(a) \lessdot_R$ *Rule* (c), *Rule* $(a) \lessdot_R$ *Rule* (d), *Rule* $(b) \lessdot_R$ *Rule* (c), and *Rule* $(b) \lessdot_R$ *Rule* (d).

4.2 Precedence Analysis

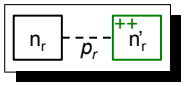
The following definitions present our path-based node precedence analysis which is used to topologically sort the nodes in a source graph and thus control the iterative transformation process:

Definition 7 (Paths and Type Paths). *Let G be a typed graph with type graph TG. A* path *p between two nodes $n_1, n_k \in V_G$ is an alternating sequence of nodes and edges in V_G and E_G, respectively, denoted as $p := n_1 \cdot e_1^{\alpha_1} \cdot n_2 \cdot \ldots \cdot n_{k-1} \cdot e_{k-1}^{\alpha_{k-1}} \cdot n_k$, where $\alpha_i \in \{+, -\}$ specifies if an edge e_i is traversed from source $s(e_i) = n_i$ to target $t(e_i) = n_{i+1}$ (+), or in a reverse direction (–). A* type path *is a path between node types and edge types in V_{TG} and E_{TG}, respectively. Given a path p, its* type (path) *is defined as $type_p(p) := type_V(n_1) \cdot type_E(e_1)^{\alpha_1} \cdot type_V(n_2) \cdot type_E(e_2)^{\alpha_2} \cdot \ldots \cdot type_V(n_{k-1}) \cdot type_E(e_{k-1})^{\alpha_{k-1}} \cdot type_V(n_k)$.*

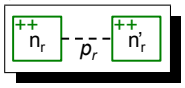
For our analysis we are only interested in paths that are induced by certain *certain patterns* present in the *TGG rules*.

Definition 8 (Relevant Node Creation Patterns). *For a* $TGG = (TG, \mathcal{R})$ *and all rules* $r \in \mathcal{R}$*, where* $r = (L, R) = (L_S \leftarrow L_C \rightarrow L_T, R_S \leftarrow R_C \rightarrow R_T)$*. The set* Paths_S *denotes all paths in* R_S *(note that* $L_S \subseteq R_S$*).*
The predicates $context_S : Paths_S \rightarrow \{true, false\}$ *and* $create_S : Paths_S \rightarrow \{true, false\}$ *in the source domain are defined as follows:*

context$_S(p_r) := \exists\ r \in \mathcal{R}$ *s.t.* p_r *is a path between two nodes* $n_r, n'_r \in R_S$ *:*
$(n_r \in L_S) \wedge (n'_r \in R_S \setminus L_S)$*, i.e., a rule* r *in* $\mathcal{R}$ *contains a path* p_r *which is isomorphic to the node creation pattern depicted in the diagram to the right.*



create$_S(p_r) := \exists\ r \in \mathcal{R}$ *s.t.* p_r *is a path between two nodes* $n_r, n'_r \in R_S$ *:*
$(n_r \in R_S \setminus L_S) \wedge (n'_r \in R_S \setminus L_S)$*, i.e., a rule* r *in* $\mathcal{R}$ *contains a path* p_r *which is isomorphic to the node creation pattern depicted in the diagram to the right.*



We can now define the set of interesting type paths, relevant for our analysis.

Definition 9 (Type Path Sets). *The set* TPaths_S *denotes all type paths of paths in* $Paths_S$ *(cf. Def. 8), i.e.,* $TPaths_S := \{tp \mid \exists\ p \in Paths_S \text{ s.t. } type_p(p) = tp\}$*. Thus, we define the restricted* create type path set *for the source domain as* $TP_S^{create} := \{tp \in TPaths_S \mid \exists\ p \in Paths_S \wedge type_p(p) = tp \wedge create_S(p)\}$*, and the restricted* context type path set *for the source domain as* $TP_S^{context} := \{tp \in TPaths_S \mid \exists\ p \in Paths_S \wedge type_p(p) = tp \wedge context_S(p)\}$*.*

In the following, we formalize the concept of *precedence between nodes*, indicating that one node could be used as context when transforming another node.

Definition 10 (Precedence Function $\mathcal{PF}_S$). *Let* $\mathcal{P} := \{\lessdot, \doteq, \cdot\lightning\cdot\}$ *be the set of precedence relation symbols. Given a* $TGG = (TG, \mathcal{R})$ *and the restricted type path sets for the source domain* $TP_S^{create}, TP_S^{context}$*. The precedence function for the source domain* $\mathcal{PF}_S : \{TP_S^{create} \cup TP_S^{context}\} \rightarrow \mathcal{P}$ *is computed as follows:*

$$\mathcal{PF}_S(tp) := \begin{cases} \lessdot & \text{iff } tp \in \{TP_S^{context} \setminus TP_S^{create}\} \\ \doteq & \text{iff } tp \in \{TP_S^{create} \setminus TP_S^{context}\} \\ \cdot\lightning\cdot & \text{otherwise} \end{cases}$$

Example. $\mathcal{PF}_S$ for our running example consists of the following entries:
Rule (a): $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO}) = \doteq$ and $\mathcal{PF}_S(\texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}) = \doteq$
Rule (b): $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}) = \lessdot$ and
$\mathcal{PF}_S(\texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}) = \lessdot$
Rules (c) and (d): $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{employs}^- \cdot \texttt{Employee}) = \lessdot$ and
$\mathcal{PF}_S(\texttt{CEO} \cdot \texttt{employs}^- \cdot \texttt{Employee}) = \lessdot$

Restriction. As our precedence analysis depends on paths in rules of a given TGG, the presented approach requires TGG rules that are (weakly) connected in each domain. Hence, considering the source domain, the following must hold: $\forall\ r \in \mathcal{R}, \forall\ n, n' \in R_S : \exists\ p \in \text{Paths}_S$ between n and n'.

Based on the precedence function $\mathcal{PF}_S$, relations $\lessdot_S$ and $\doteq_S^*$ can now be defined and used to topologically sort a given input graph and determine the sets of elements that can be transformed at each step in the algorithm.

Definition 11 (Source Path Set). *For a given typed source graph G_S, the* source path set *for the source domain is defined as follows:*
$P_S := \{p \mid p \text{ is a path between } n, n' \in V_{G_S} \wedge type_p(p) \in \{TP_S^{create} \cup TP_S^{context}\}\}$.

Definition 12 (Precedence Relation $\lessdot_S$). *Given $\mathcal{PF}_S$, the precedence function for a given TGG, and a typed source graph G_S. The precedence relation $\lessdot_S \subseteq V_{G_S} \times V_{G_S}$ for the source domain is defined as follows: $n \lessdot_S n'$ if there exists a path $p \in P_S$ between nodes n and n' such that $\mathcal{PF}_S(type_p(p)) = \lessdot$.*

Example. For our example triple (Fig. 2), the following pairs constitute $\lessdot_S$: $(\texttt{ES} \lessdot_S \texttt{Ingo}), (\texttt{ES} \lessdot_S \texttt{Tony}), (\texttt{ES} \lessdot_S \texttt{Marius}), (\texttt{Andy} \lessdot_S \texttt{Ingo}), (\texttt{Andy} \lessdot_S \texttt{Tony})$, and $(\texttt{Andy} \lessdot_S \texttt{Marius})$.

Definition 13 (Relation $\doteq_S$). *Given $\mathcal{PF}_S$, the precedence function for a given TGG, and a typed source graph G_S. The symmetric relation $\doteq_S \subseteq V_{G_S} \times V_{G_S}$ for the source domain is defined as follows: $n \doteq_S n'$ if there exists a path $p \in P_S$ between nodes n and n' such that $\mathcal{PF}_S(type_p(p)) = \doteq$.*

Definition 14 (Equivalence Relation $\doteq_S^*$). *The equivalence relation $\doteq_S^*$ is the transitive and reflexive closure of the symmetric relation $\doteq_S$.*

Example. For our example triple (Fig. 2), the following equivalence classes constitute $\doteq_S^*$: $\{\texttt{Andy}, \texttt{ES}\}$, $\{\texttt{Ingo}\}$, $\{\texttt{Tony}\}$, and $\{\texttt{Marius}\}$.

Definition 15 (Precedence Graph $\mathcal{PG}_S$). *The precedence graph for a given source graph G_S is a graph $\mathcal{PG}_S$ constructed as follows:*
(i) The equivalence relation $\doteq_S^*$ is used to partition V_{G_S} into equivalence classes $EQ_1, \ldots EQ_n$ which serve as the nodes of $\mathcal{PG}_S$, i.e., $V_{\mathcal{PG}_S} := \{EQ_1, \ldots, EQ_n\}$.
(ii) The edges in $\mathcal{PG}_S$ are defined as follows:
$E_{\mathcal{PG}_S} := \{e \mid s(e) = EQ_i,\ t(e) = EQ_j : \exists\ n_i \in EQ_i, n_j \in EQ_j \text{ with } n_i \lessdot_S n_j\}$.

Example. The corresponding $\mathcal{PG}_S$ constructed from our example triple is depicted in Fig. 5(a) in Sect. 5.

5 Precedence TGG Batch Algorithm

In this section, we present our batch algorithm (cf. Algorithm 1) and explain how the introduced rule dependency and node precedence analyses are used to efficiently transform a given source graph. For a forward transformation (a backward transformation works analogously), the input for the algorithm is a graph G_S, the statically derived rule dependency relation $\lessdot_R$, and the precedence function for the source domain $\mathcal{PF}_S$.
Procedure TRANSFORM determines a graph triple $G_S \leftarrow G_C \rightarrow G_T$ as output. The first step (line (2)) of the algorithm is to build the precedence graph $\mathcal{PG}_S$ according to Def. 15. Note that the procedure BUILDPRECEDENCEGRAPH will terminate with an error if there is a cycle in the precedence graph and it is thus

Algorithm 1 Precedence TGG Batch Algorithm

```
 1: procedure TRANSFORM(G_S, ≺_R, PF_S)
 2:     PG_S ← BUILDPRECEDENCEGRAPH(G_S, PF_S)
 3:     while (PG_S contains equivalence classes) do
 4:         readyNodes ← all nodes in equiv. classes in PG_S without incoming edges
 5:         readyNodes ← sort readyNodes utilizing ≺_R
 6:         for (node n in readyNodes) do
 7:             transformedNodes ← CHOOSEANDAPPLYRULE(n)
 8:             if transformedNodes ≠ ∅ then
 9:                 PG_S ← remove all nodes in transformedNodes from PG_S
10:                 break
11:             end if
12:         end for
13:         if transformedNodes = ∅ then
14:             terminate with error        ▷ Local Completeness Criterion violated
15:         end if
16:     end while
17:     return G_S ← G_C → G_T
18: end procedure
```

impossible to sort the elements of the source graph according to their dependencies. Starting on line (3), a while-loop iterates over equivalence classes in $\mathcal{PG}_S$ until there are none left. In the while-loop, the set *readyNodes* contains all nodes that can be transformed next, i.e., whose context elements have already been transformed (line (4)). This set is determined by taking all nodes in the equivalence classes of $\mathcal{PG}_S$, which do not have incoming edges (dependencies). On line (5), *readyNodes* is sorted according to the partially ordered relation $\prec_R$, i.e., the rules that can be used to transform nodes in *readyNodes* are determined, sorted with $\prec_R$ and reflected in *readyNodes*. This could be achieved by assigning an integer to each rule according to the partial order of $\prec_R$ and then selecting the largest number of all rules that translate $n \in$ *readyNodes* for n.[1] Next, a for-loop iterates over the sorted *readyNodes* (line (6)). On line (7) the procedure CHOOSEANDAPPLYRULE is used to determine and filter the rules as presented in [12], allowing for user input or choosing arbitrarily from the final applicable rules. If a rule could be successfully chosen and applied to transform n on line (7), a non-empty set of *transformedNodes* is returned that is used to update $\mathcal{PG}_S$ on line (9). In this case, the for-loop is terminated and the while-loop is repeated with the updated and thus "smaller" $\mathcal{PG}_S$. If *transformedNodes* is empty, the for-loop is repeated for the next node in $readyNodes$. If *transformedNodes*, however, *remains* empty on line (13), we know that no node in *readyNodes* has been transformed and that the algorithm has hit a dead-end. This can only happen for TGGs that violate the *Local Completeness Criterion* (cf. algorithm strategy I.c in Sect. 3) and are *not* in the class of supported TGGs.

[1] If it is not possible to sort *readyNodes* due to cycles in $\prec_R$, this additional analysis supplies no further information and *readyNodes* remains unchanged.

Example. To demonstrate the presented algorithm, we apply a forward transformation for the source graph of our example triple depicted in Fig. 2. Given as input is G_S, the rule dependency relation $\lessdot_R$ (depicted as a graph in Fig. 5(b)), and the precedence function $\mathcal{PF}_S$ (cf. example for Def. 10). On line (2), the precedence graph $\mathcal{PG}_S$ for G_S, depicted in Fig. 5(a), is built. $\mathcal{PG}_S$ is acyclic, hence the transformation can continue.

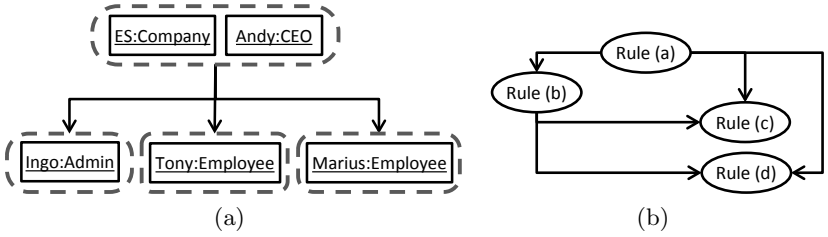


Fig. 5. $\mathcal{PG}_S$ for the input graph (left) and relation $\lessdot_R$ for all rules (a)–(d) (right)

On line (4), the set *readyNodes* is determined, consisting in this case of the nodes `ES` and `Andy` from a single equivalence class of $\mathcal{PG}_S$. On line (5), only one rule can be used to transform both nodes and, therefore, the sorting is trivial. On line (6) `ES` or `Andy` is chosen randomly, and in either case, the only candidate rule is Rule (a) (Fig. 3), which can be directly applied on line (7). Again in either case, *transformedNodes* contains both nodes as Rule (a) transforms `ES` and `Andy` simultaneously. $\mathcal{PG}_S$ is updated on line (9) to consist of three unconnected equivalence classes `Ingo`, `Tony`, and `Marius`, and the for-loop terminates. In the second iteration through the while-loop, *readyNodes* now contains all these three elements and will be sorted according to $\lessdot_R$ on line (5). This time, the sorting reveals that `Ingo` must be transformed before `Tony` and `Marius` as Rules (c) and (d) both require a `Network` as context in the target domain, which can only be created by applying Rule (b) first, i.e., *Rule* $(b) \lessdot_R$ *Rule* (c), *Rule* $(b) \lessdot_R$ *Rule* (d) (Fig. 5(b)). The for-loop in line (6), therefore, starts with `Ingo`. Applying Rule (b) (line (7)) puts `Ingo` in *transformedNodes*, $\mathcal{PG}_S$ is updated on line (9) to now contain only `Tony` and `Marius` and the for-loop is terminated with the break on line (10). In the third iteration, *readyNodes* contains `Tony` and `Marius`, and no sorting is needed as Rules (c) and (d) do not depend on each other. On line (6) `Tony` could be randomly selected first and (arbitrarily or via user input) Rule (c) could be chosen to be applied on line (7). After updating $\mathcal{PG}_S$ again and breaking out of the for-loop, only `Marius` remains untransformed. Similar to the penultimate iteration, Rule (d) could be selected and applied this time. Updating $\mathcal{PG}_S$ on line (9) empties the precedence graph, which terminates the while-loop on line (3). The created graph triple depicted in Fig. 2 is returned on line (17).

Formal Properties of the Precedence TGG Batch Algorithm

In the following we argue that the presented algorithm retains all formal properties stipulated in [17] and proved for the context-driven algorithm of [12].

Definition 16 (Correctness, Completeness and Efficiency).
Correctness: *Given a source graph G_S, the transformation algorithm either terminates with an error or produces a graph triple $G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)$.*
Completeness: *For all triples $G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)$, the transformation algorithm produces a consistent triple $G_S \leftarrow G'_C \rightarrow G'_T \in \mathcal{L}(TGG)$ for the input source graph G_S.*
Efficiency: *According to [17], a TGG batch transformation algorithm is efficient if its runtime complexity class is $O(n^k)$, where n is the number of nodes in the source graph to be transformed and k is the largest number of elements to be matched by any rule r of the given TGG.*

All properties are defined analogously for backward transformations.

Theorem. *Algorithm 1 is correct, complete and efficient for any source-local complete TGG [12].*

Proof.
Correctness: If the algorithm returns a graph triple, i.e., does not terminate with an error, it was able to determine a sequence of source rules $r_{1_S}, r_{2_S}, \ldots, r_{n_S}$ that would build the given source graph G_S and, thus, the corresponding sequence of forward rules $r_{1_F}, r_{2_F}, \ldots, r_{n_F}$ that transform the given source graph (Def. 5). The *Decomposition and Composition Theorem* of [5] guarantees that it is possible to compose the sequence $r_{1_S}, r_{2_S}, \ldots, r_{n_S}, r_{1_F}, r_{2_F}, \ldots, r_{n_F}$ to the sequence of TGG rules $r_1, r_2, \ldots, r_n$ which proves that the resulting graph triple is consistent, i.e., $G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)$. □
Completeness: Showing completeness is done in two steps: First of all, we consider the algorithm without the additional concept of rule dependencies via the relation $\lessdot_R$.

The remaining algorithm transforms nodes with the same concepts (e.g., dangling edge check) as the previous algorithm in [12], but iteratively *in a fixed sequence*, for which we guarantee, by definition of the precedence graph (cf. 15), that the context of every node is always transformed first. As the context-driven strategy taken by the algorithm in [12] is able to transform a model by arbitrarily choosing an element and transforming its context elements in a bottom-up manner (cf. Sect. 3), the fixed sequence taken by our algorithm must be a possible sequence that could be chosen by the algorithm in [12]. Algorithm 1 can, therefore, be seen as forcing the context-driven algorithm to transform elements in one of the possible sequences, from which it can arbitrarily choose. This shows that all completeness arguments from [12] can be transferred to the new algorithm, i.e., Algorithm 1 is complete for the class of local complete TGGs.

In a second step, we now consider the algorithm with the additional relation $\lessdot_R$ and, therefore, the capability of handling specifications with cross-domain

context dependencies as in our running example. We have shown in Sect. 3 that the algorithm presented in [12] cannot cope with such specifications as they violate the local-completeness criterion. We can, hence, conclude that Algorithm 1 is more expressive than the previous context-driven algorithm as it can handle certain TGGs that are not local complete. We leave the precise categorization of this new class of TGGs to future work. □

Efficiency: Building the precedence graph $\mathcal{PG}_S$ on line (2), essentially a topological sorting, is realizable in $O(n^l)$, where l is the maximum length of relevant paths according to $\mathcal{PF}_S$. Note that l can be at most of size k (the largest number of elements to be matched by any rule r of the given TGG), thus we can estimate this with $O(n^k)$. The while-loop starting on line (3) iterates through $\mathcal{PG}_S$, which will be decreased every time by at least one node from an equivalence class. The while-loop is, thus, run in the worst-case (equivalence classes in $\mathcal{PG}_S$ all consisting of exactly one node) n times. In the while-loop, we select equivalence classes without incoming edges in line (4). This can be achieved in $O(n)$ by iterating through $\mathcal{PG}_S$. Building the topological order on line (5) requires inspecting all nodes in *readyNodes* and their appropriate rules in $O(n)$. The for-loop starting on line (6) iterates in the worst-case over all nodes in *readyNodes* where updating $\mathcal{PG}_S$ on line (9), requires traversing all successor nodes which is at most $n-1$ (i.e., $O(n)$). As argued in [12], transforming a node, i.e., checking all conditions and performing pattern matching (line (7)), is assumed to run in $O(n^k)$ (cf. Def. 16). Summarizing, we obtain: $n^k + n \cdot (n + n + n \cdot (n^k + n)) \in O(n^k)$. □

As TGGs are symmetric [8], all arguments can be transferred analogously to backward transformations.

6 Related Work on Alternative Bidirectional Languages

Complementing our related work on TGG batch algorithms (cf. Sect. 3), we now focus on *alternative bidirectional languages* that share and address similar challenges as TGGs but take fundamentally different strategies. As bidirectionality is a challenge in various application domains and communities, there exists a substantial number of different approaches, formalizations and tools [18]. The lenses framework is of particular interest when compared to TGGs, as [8] has shown that incremental TGGs can be viewed as an implementation of a *delta-based* framework for *symmetric lenses*. Although we have presented a batch algorithm for TGGs, our ultimate goal is to provide a solid basis for an efficient incremental TGG implementation. As compared to existing lenses implementations for string data or trees such as *Boomerang* [2], TGGs are better suited for MDE where model transformations operate on complex *graph-like* structures. Similar to TGGs, *GRoundTram*, a bidirectional framework based on graph transformations [10], aims to support model transformations in the context of MDE. There are, however, a number of interesting differences: (i) While GRoundTram demands a forward transformation from the user and automatically generates a consistent backward transformation, TGGs (in this respect similar to lenses)

provide a language from which both forward and backward transformations are automatically derived. Both approaches face a different set of non-trivial challenges. (ii) GRoundTram uses UnQL+, which is based on the graph query algebra UnCAL, with a strong emphasis on compositionality, while TGGs are rule-based algebraic graph transformations. (iii) GRoundTram maintains traceability in an implicit manner while TGGs create explicit typed traceability links between integrated models, which can be used to store extra information for incremental model synchronization or manual reviews. In contrast to both Boomerang and GRoundTram, TGGs adhere to the fundamental *unification* principle in MDE (everything is a model) and as such, a bidirectional model transformation specified as a TGG is a model which is conform to a well defined TGG metamodel. Unification has wide-reaching consequences including enabling a natural *bootstrap* and *higher order transformations*. Finally, TGGs served as an inspiration and basis for the standard OMG bidirectional transformation language QVT and can be regarded as a valid implementation thereof [18].

7 Conclusion and Future Work

In this paper, an improvement of our previous TGG batch algorithm was presented. We introduced a novel *node precedence analysis* of TGG specifications combined with a *rule dependency analysis* to further support the batch transformation control algorithm in determining the node processing order. The result is an iterative batch transformation strategy in a top-down manner with increased expressiveness. We have shown that this algorithm runs in polynomial runtime and complies to the formal properties for TGG implementations according to [17], and, therefore, is well-suited for real-world applications where efficiency is almost as important as the reliability of the expected result.

As a next step, we shall implement the presented algorithm as an extension of our current batch implementation in our metamodeling tool eMoflon[2][1], and start working on an efficient incremental TGG algorithm based on our rule dependency and node precedence analyses. Finally, providing a *rule checker* that decides at compile time if a given TGG can be transformed by our algorithm is a crucial task to improve the usability of our tool.

References

1. Anjorin, A., Lauder, M., Patzina, S., Schürr, A.: eMoflon : Leveraging EMF and Professional CASE Tools. In: Heiß, H.U., Pepper, P., Schlingloff, H., Schneider, J. (eds.) Proc. of MEMWe '11. LNI, vol. 192. GI (2011)
2. Bohannon, A., Foster, J., Pierce, B., Pilkiewicz, A., Schmitt, A.: Boomerang : Resourceful Lenses for String Data. ACM SIGPLAN Notices 43(1), 407–419 (2008)
3. Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Systems Journal 45(3), 621–645 (2006)

[2] `http://www.moflon.org`

4. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.: Bidirectional Transformations: A Cross-Discipline Perspective. In: Paige, R.F. (ed.) Proc. of ICMT '09, LNCS, vol. 5563, pp. 260–283. Springer (2009)
5. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In: Dwyer, M., Lopes, A. (eds.) Proc. of FASE '07, LNCS, vol. 4422, pp. 72–86. Springer (2007)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. (Monographs in Theoretical Computer Science. An EATCS Series.). Springer (2006)
7. Giese, H., Hildebrandt, S., Lambers, L.: Toward Bridging the Gap between Formal Semantics and Implementation of Triple Graph Grammars. In: Lúcio, L., Vieira, E., Weißleder, S. (eds.) Proc. of MoDeVVA '10. pp. 19–24. IEEE (2010)
8. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y.: Correctness of Model Synchronization Based on Triple Graph Grammars. In: Whittle, J., Clark, T., Kühne, T. (eds.) Proc. of MODELS '11, LNCS, vol. 6981, pp. 668–682. Springer (2011)
9. Hermann, F., Golas, U., Orejas, F.: Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In: Bézivin, J., Soley, M.R., Vallecillo, A. (eds.) Proc. of MDI '10. ICPS, vol. 482, pp. 22–31. ACM (2010)
10. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Nakano, K.: GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations. In: Alexander, P., Pasareanu, C., Hosking, J. (eds.) Proc. of ASE '11. pp. 480–483. IEEE (2011)
11. Kindler, E., Rubin, V., Wagner, R.: An Adaptable TGG Interpreter for In-Memory Model Transformations. In: Schürr, A., Zündorf, A. (eds.) Proc. of Fujaba Days '04. pp. 35–38 (2004)
12. Klar, F., Lauder, M., Königs, A., Schürr, A.: Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In: Schürr, A., Lewerentz, C., Engels, G., Schäfer, W., Westfechtel, B. (eds.) Graph Transformations and Model Driven Enginering, LNCS, vol. 5765, pp. 141–174. Springer (2010)
13. Königs, A.: Model Transformation with Triple Graph Grammars. In: Proc. of MTIP '05 (2005)
14. Lauder, M., Schlereth, M., Rose, S., Schürr, A.: Model-Driven Systems Engineering: State-of-the-Art and Research Challenges. Bulletin of the Polish Academy of Sciences, Technical Sciences 58(3), 409–422 (2010)
15. Rose, S., Lauder, M., Schlereth, M., Schürr, A.: A Multidimensional Approach for Concurrent Model Driven Automation Engineering. In: Osis, J., Asnina, E. (eds.) Model-Driven Domain Analysis and Software Development, pp. 90–113. IGI (2011)
16. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Tinhofer, G. (ed.) Proc. of WG '94. LNCS, vol. 903, pp. 151–163. Springer (1994)
17. Schürr, A., Klar, F.: 15 Years of Triple Graph Grammars. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) Graph Transformations, LNCS, vol. 5214, pp. 411–425. Springer (2008)
18. Stevens, P.: A Landscape of Bidirectional Model Transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) Proc. of GTTSE '07, LNCS, vol. 5235, pp. 408–424. Springer (2008)
19. Stevens, P.: Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. SoSym 9(1), 7–20 (2008)