Efficient Model Synchronization 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 and incremental model transformation. In practical scenarios, unidirectional rules for incremental forward and backward transformation are automatically derived from the TGG rules in the specification, and the overall transformation process is governed by a control algorithm. Current incremental implementations either have a runtime complexity that depends on the size of related models and not on the number of changes and their affected elements, or do not pursue formalization to give reliable predictions regarding the expected results. In this paper, a novel incremental model synchronization algorithm for TGGs is introduced, which employs a static analysis of TGG specifications to efficiently determine the range of influence of model changes, while retaining all formal properties.

Keywords: triple graph grammars, model synchronization, control algorithm of incremental transformations, node precedence analysis

1 Introduction

Model-Driven Engineering (MDE) 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. As industrial applications require reliability and efficiency, the need for formal frameworks that guarantee useful properties of model transformation arises. Especially for *bidirectional* model transformation, it is challenging to define precise semantics for the manipulation and synchronization of models with efficient tool support. The *Triple Graph Grammar (TGG)* [13] approach has not only solid formal foundations [3,11] but also various tool implementations [1,6,10]. TGGs provide a declarative, rule-based means of specifying the consistency of two models in their respective domains, and tracking inter-domain relationships between elements explicitly by using a correspondence model. Although TGGs describe how *triples* consisting

* 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.

of source, correspondence, and target models are derived in parallel, most practical scenarios involve existing models and require unidirectional transformation. Consequently, TGG tools support model transformation based on unidirectional forward and backward operational rules, automatically derived from a single TGG specification, as basic transformation steps, and use an algorithm to control which rule is to be applied on which part of the input graph. Such a *batch transformation* is the standard scenario for model transformation, where existing models are transformed (completely) from scratch.

In contrast, *incremental* model transformation supports changing already related models and propagating deltas appropriately. The challenge is to perform the update in an *efficient* manner and to avoid information loss by retaining unaffected elements of the models. Determining such an update sequence is a difficult task because transformations of deleted elements and their dependencies, as well as transformations of potential dependencies of newly added elements must be revoked [9]. The challenge is to identify such dependent elements in the model and to undo their previous transformation taking all changes into account.

Current incremental TGG approaches guarantee either the formal properties of *correctness* meaning that only consistent graph triples are produced, and *completeness* meaning that all possible consistent triples, which can be derived from a source or a target graph, can actually be produced, but are inefficient (scale with the size of the overall models) [9], or are efficient (scale with the number of changes and affected elements), but do not consider formal aspects [5,7].

In this paper, we introduce a novel incremental TGG control algorithm for model synchronization and prove its correctness, completeness, and efficiency. Based on our *precedence*-driven TGG batch algorithm presented in [12], a static *precedence analysis* is used to retrieve information, which allows for deciding which elements may be affected by deletions and additions of elements.

Section 2 introduces fundamentals and our running example. Section 3 presents our node precedence analysis, used by the incremental TGG algorithm presented in Sect. 4. Section 5 discusses related approaches, while Sect. 6 concludes with a summary and future work.

2 Fundamentals and Running Example

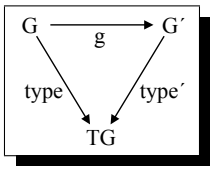
In this section, all concepts required to formalize and present our contribution are introduced and explained using our running example.

2.1 Type Graphs, Typed Graphs and Triples

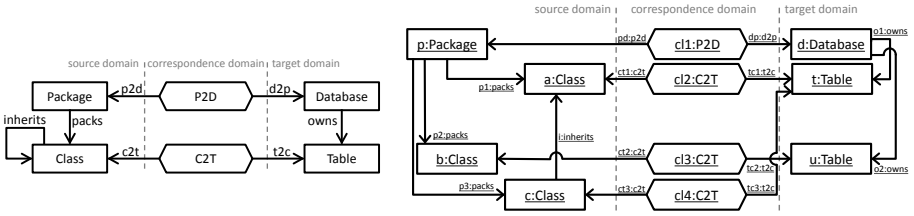
We introduce the concept of a *graph*, and formalize *models* as *typed graphs*.

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 depicted 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 [4,11]. In the following, we work with *typed graph triples* and corresponding morphisms.
Example: Our running example specifies the integration of *class diagrams* and corresponding *database schemata*. The *TGG schema* depicted in Fig. 1(a) is the type graph triple for our running example. In the *source domain*, class diagrams consist of `Packages`, `Classes`, and inheritance between `Classes`. In the *target domain*, a database schema consists of `Databases` and `Tables`. The *correspondence domain* specifies links between elements in the different domains, in this case `P2D` relating packages with databases, and `C2T` relating classes with tables. In Fig. 1(b), a schema conform (typed graph) triple is depicted: a package `p`



(a) TGG schema (b) A TGG schema conform triple

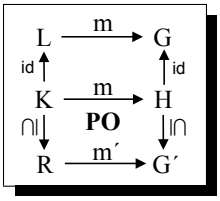
Fig. 1. TGG schema for the running example and a conform triple

consists of three classes `a`, `b`, and `c`, while the corresponding database schema `d` contains two tables `t` and `u`.

2.2 Triple Graph Grammars and Rules

The simultaneous evolution of typed graph triples can be described by a *triple graph grammar* consisting of *transformation rules*. In general, transformation rules can be formalized via a double-pushout to allow for creating and deleting elements in a graph [4]. As TGG rules are restricted to the creation of elements, we simplify the definition in the following:

Definition 3 (Graph Triple Rewriting for Monotonic Creating Rules). *A* monotonic creating rule $r := (L = K, R)$, *is a pair of typed graph triples s.t.* $L \subseteq R$. *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* $m' : R \to G'$ *can be defined by building the* pushout G' *as denoted in the diagram to the right.*



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

Example: In Fig. 2, Rules (a)–(c) declare how an integrated class diagram and a database schema are created simultaneously. Rule (a) creates the root elements (a `Package` and a corresponding `Database`), while Rule (b) appends a `Class` and a `Table`, and Rule (c) extends the models with an inheriting `Class`, which is related to the same `Table`. We use a concise notation (merging L and R) depicting context elements in black without any markup, and created elements in green with a "++" markup.

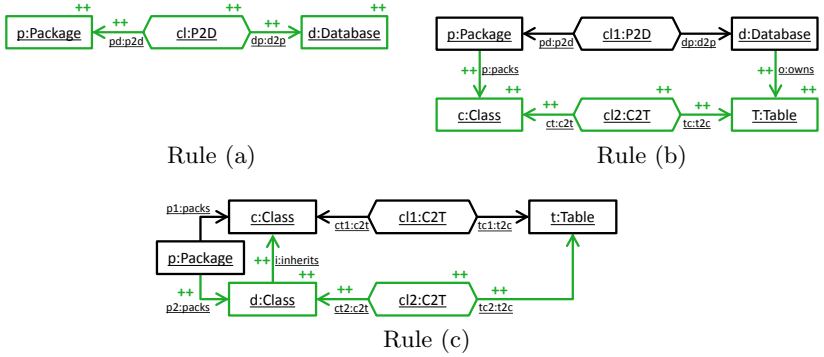


Fig. 2. TGG Rules (a)–(c) for the integration

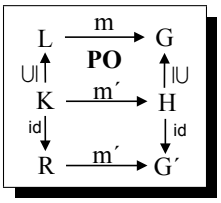
2.3 Derived Operational Rules

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 a corresponding target domain model, and vice versa. Although we focus in the following sections only on forward transformations, all concepts and arguments are symmetric and can be applied analogously for the case of backward transformations.

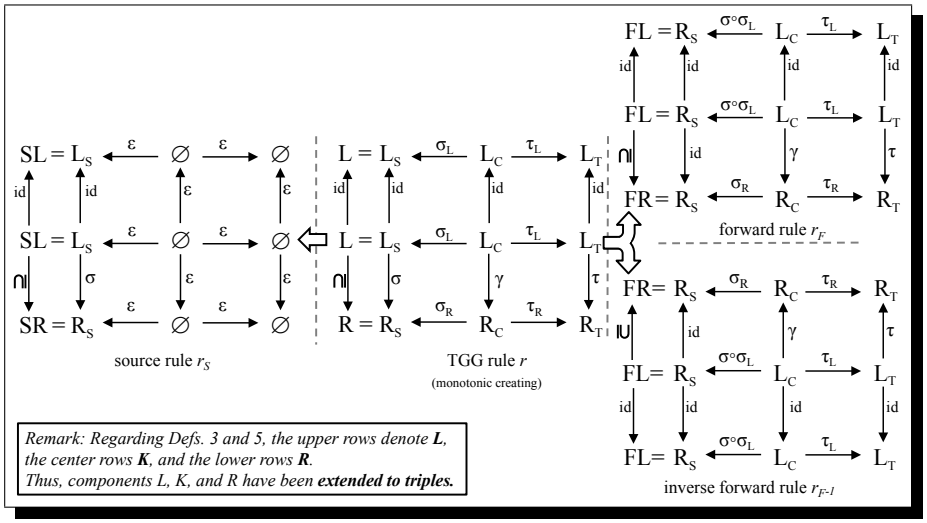
As shown in [3,13], a sequence of TGG rules, which describes a simultaneous evolution, can be uniquely decomposed into (and conversely composed from) a sequence of *source rules* that only evolve the source model and *forward rules* that retain the source model and evolve the correspondence and target models. In addition, *inverse forward rules* revoke the effects of forward rules. These operational rules serve as the building blocks used by our control algorithm. As inverse forward rules only delete elements, we define monotonic deleting rules:

Definition 5 (Graph Triple Rewriting for Monotonic Deleting Rules). *A* monotonic deleting rule $r := (L, K = R)$*, is a pair of typed graph triples s.t.* $L \supseteq R$*. A rule* r rewrites *(via deleting 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* $m' : R \to G'$ *can be defined by building the* pushout complement $H = G'$ *as denoted in the diagram to the right.*



The elements in $L \setminus R$ of a monotonic deleting rule are referred to as *deleted elements*. Using this definition, operational rule derivation is formalized as follows:

Definition 6 (Derived Operational Rules). *Given a* $TGG = (TG, \mathcal{R})$ *and a rule* $r = (L, R) \in \mathcal{R}$*, a* source rule $r_S = (SL, SR)$*, a* forward rule $r_F = (FL, FR)$ *and an* inverse forward *rule* $r_{F^{-1}} = (FR, FL)$ *are derived as follows:*



The forward rule r_F *can be applied according to Def. 3, i.e., this involves building a pushout to* create *the required elements, while the inverse forward rule* $r_{F^{-1}}$ *involves building a pushout complement to* delete *the required elements according*

to Def. 5. Given a forward rule r_F, the existence of rule $r_{F^{-1}}$, which reverses an application of r_F up to isomorphism, can be shown according to Fact 3.3 in [4].

Although forward and inverse forward rules retain all source elements, the control algorithm keeps track of which source elements are transformed by a rule application. This can be done by introducing marking attributes [9], or maintaining a bookkeeping data structure in the control algorithm [6]. In concrete syntax, we equip every *transformed* element with a *checked box*, and every *untransformed* elements with an *unchecked box* (cf. Fig. 4) as introduced in [11].
Example: From Rule (b) (Fig. 2), the operational rules r_S, r_F, and r_F^{-1} depicted in Fig. 3 are derived. The source rule extends the source graph by adding a `Class` to an existing `Package`, while the forward rule r_F *transforms* (denoted as $\square \to \boxtimes$) an existing `Class` by *creating* a new `C2T` link and `Table` in the corresponding `Database`. The inverse forward rule *untransforms* (denoted as $\boxtimes \to \square$) a `Class` in a `Package` by *deleting* the corresponding link and `Table`, i.e., revoking the modifications of the forward rule. In addition to the already introduced merged representation of L and R of a rule, we further indicate deleted elements by a "−−" markup and red color. Forward and inverse forward rules match the same context element and retain the checked box (denoted as $\boxtimes \to \boxtimes$).

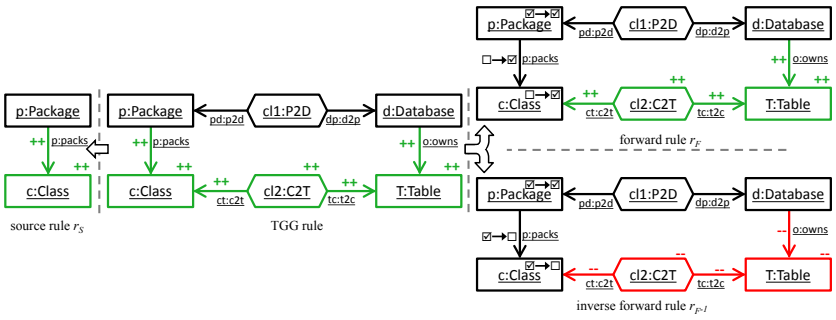


Fig. 3. Source and forward rules derived from Rule (b)

3 Precedence Analysis for TGGs

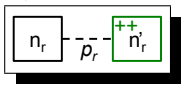
In the following, we introduce a path-based precedence analysis, which is used to partially sort the nodes in a source graph and thus control the transformation process. We formalize the concepts only for the source domain and a corresponding forward transformation, but, as before, all concepts can be directly transferred to the target domain and backward transformation, respectively.

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 $p := n_1 \cdot e_1^{\alpha_1} \cdot n_2 \cdot$*

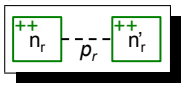
$\ldots \cdot n_{k-1} \cdot e_{k-1}^{\alpha_{k-1}} \cdot n_k$, *where* $\alpha_i \in \{+, -\}$ *specifies if an edge* e_i *is traversed from source* $s(e_i) = n_i$ *to target* $t(e_i) = n_{i+1}$ *(+), or in a reverse direction (–). A* type path *is a path between node types and edge types in* V_{TG} *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 *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:*
$\boldsymbol{context}_S(p_r) := \exists\ r \in \mathcal{R}$ *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.*



$\boldsymbol{create}_S(p_r) := \exists\ r \in \mathcal{R}$ *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 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.* $TPaths_S := \{tp \mid \exists\ p \in Paths_S$ *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*, reflecting that one node could be used as context for 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:*

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

Example: For our running example, $\mathcal{PF}_S$ consists of the following entries:
Rule (a): $\emptyset$. Rule (b): $\mathcal{PF}_S(\texttt{Package} \cdot \texttt{packs}^+ \cdot \texttt{Class}) = \lessdot$.
Rule (c): $\mathcal{PF}_S(\texttt{Package} \cdot \texttt{packs}^+ \cdot \texttt{Class}) = \lessdot$, $\mathcal{PF}_S(\texttt{Class} \cdot \texttt{inherits}^- \cdot \texttt{Class}) = \lessdot$, $\mathcal{PF}_S(\texttt{Class} \cdot \texttt{packs}^- \cdot \texttt{Package} \cdot \texttt{packs}^+ \cdot \texttt{Class}) = \lessdot$.

Note that regarding our running example, path $\texttt{Class} \cdot \texttt{packs}^- \cdot \texttt{Package}$ is not in $\mathcal{PF}_S$ as this path is neither in TP_S^{create} nor in $TP_S^{context}$.

Restriction: As our precedence analysis depends on paths in rules of a given TGG, the presented approach only works for 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, n'.

Based on the precedence function $\mathcal{PF}_S$, we now analyze typed graphs with two relations $\lessdot_S$ and $\doteq^*_S$. These are used to topologically sort a given source input graph and determine the sets of affected elements due to changes.

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. 1(b)), the following pairs constitute $\lessdot_S$: $(\texttt{p} \lessdot_S \texttt{a}), (\texttt{p} \lessdot_S \texttt{b}), (\texttt{p} \lessdot_S \texttt{c}), (\texttt{a} \lessdot_S \texttt{c})$.

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. 1(b)), relation $\doteq^*_S$ partitions the nodes of the source graph into the following equivalence classes: {p}, {a}, {b}, and {c}. For a more complex example with non-trivial equivalence classes we refer to [12].

We now define the concept of a *precedence graph* based on our relations $\doteq^*_S$, $\lessdot_S$ to sort a given graph according to its precedences, which is used by the incremental algorithm to determine if an element is available for transformation.

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. 4.
Remark: $\mathcal{PG}_S$ defines a partial order over equivalence classes. This is a direct consequence of Def. 15.

Finally, we define the class of typed graph triples that do not introduce contradicting precedence relations between connected source and target domain elements. This is important as the synchronization control algorithm presented in Sect. 4 relies *only* on the source domain when applying appropriate changes to the correspondence and target domain.

Definition 16 (Forward Precedence Preserving Graph Triples). *Given a graph triple* $G = G_S \xleftarrow{h_S} G_C \xrightarrow{h_T} G_T$ *and two corresponding precedence graphs* $\mathcal{PG}_S$ *and* $\mathcal{PG}_T$*. For* $EQ_S \in V_{\mathcal{PG}_S}$ *and* $EQ_T \in V_{\mathcal{PG}_T}$*, the predicate* cross-domain-connected *on pairs of equivalence classes in precedence graphs of different domains is defined as follows: cross-domain-connected*$(EQ_S, EQ_T) := \exists\, n_C \in V_{G_C}$ *s.t.* $h_S(n_C) \in EQ_S \wedge h_T(n_C) \in EQ_T$*.*
Given $EQ_S, EQ'_S \in V_{\mathcal{PG}_S}, EQ_S \neq EQ'_S$ *and* $EQ_T, EQ'_T \in V_{\mathcal{PG}_T}, EQ_T \neq EQ'_T$ *s.t. cross-domain-connected*$(EQ_S, EQ_T) \wedge$ *cross-domain-connected*(EQ'_S, EQ'_T)*.*
The graph triple G *is* forward precedence preserving *iff*
$\exists$ *path* $p_T(EQ_T, EQ'_T) = EQ_T \cdot e_{T_1}^{\alpha_{T_1}} \cdot \ldots \cdot e_{T_n}^{\alpha_{T_n}} \cdot EQ'_T$ *s.t.* $\alpha_{T_i} = + \;\forall\, i \in \{1, \ldots, n\}$
$\Rightarrow$
$\exists$ *path* $p_S(EQ_S, EQ'_S) = EQ_S \cdot e_{S_1}^{\alpha_{S_1}} \cdot \ldots \cdot e_{S_n}^{\alpha_{S_n}} \cdot EQ'_S$ *s.t.* $\alpha_{S_i} = + \;\forall\, i \in \{1, \ldots, n\}$

Example: The running example (Fig. 1(b)) satisfies this property.

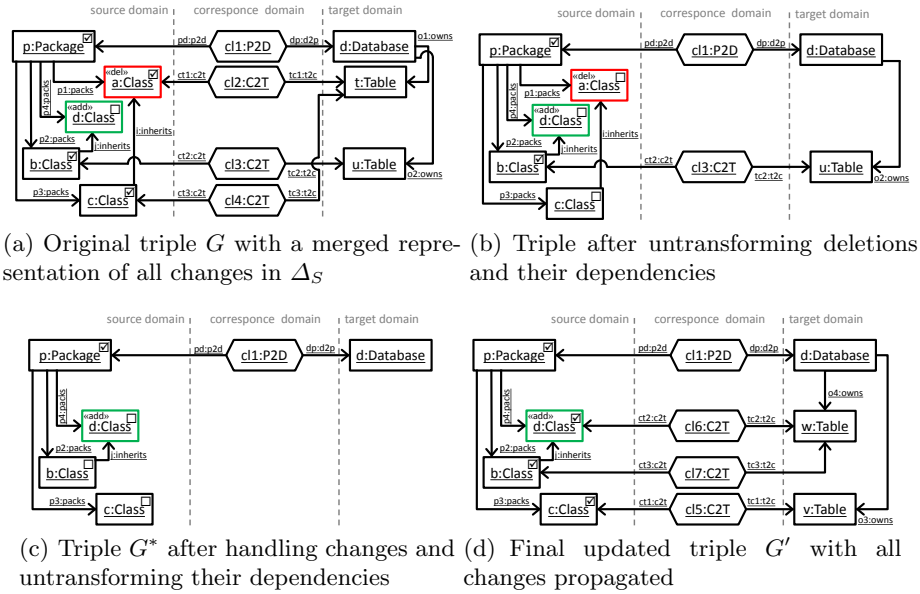
4 Incremental Precedence TGG Algorithm

To realize bidirectional incremental model synchronization with TGGs, a *control algorithm* is required that accepts a triple $G = G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)$, an update graph triple [9] for the source domain $\Delta_S = G_S \leftarrow D \rightarrow G'_S$, the pre-compiled precedence function for the source domain $\mathcal{PF}_S$, and precedence graph $\mathcal{PG}_S$ used in a previous batch or incremental transformation, and returns a consistent graph triple $G' = G'_S \leftarrow G'_C \rightarrow G'_T$ with all changes propagated to the correspondence and target domain. Therefore, this algorithm (i) untransforms deleted elements and their dependencies in a valid order, (ii) untransforms elements (potentially) dependent on additions in a valid order, and (iii) transforms all untransformed and newly created elements by using the precedence-driven batch algorithm of [12]. Regarding the valid order, the algorithm has to find a way to delete elements in the opposite domain without compromising the transformation of existing elements. As a (fomal) restriction, edges can only be added (deleted) together with adjacent nodes, hence we focus on nodes only. In practice, Ecore for example assigns all edges to nodes, which overcomes this restriction.
Example: Using our example, we describe the incremental forward propagation of the following changes in the source domain (Fig. 4(a)): class `a` is deleted (indicated by «`del`») and a new class `d` is added (indicated by «`add`»). Parameters passed to the algorithm (line 1) are the original graph triple G (Fig. 1(b)), its source domain precedence graph $\mathcal{PG}_S$ (Fig. 5(a)), update Δ_S with deleted nodes $\Delta^- := V_{G_S} \setminus V_D$ and added nodes $\Delta^+ := V_{G'_S} \setminus V_D$, and the pre-compiled source domain precedence function $\mathcal{PF}_S$ (cf. example for Def. 10). The algorithm returns a consistent graph triple with all changes propagated (Fig. 4(d)) on line 13.

Algorithm 1 Incremental Precedence TGG Algorithm

1: **procedure** PROPAGATECHANGES($G, \Delta_S, \mathcal{PF}_S, \mathcal{PG}_S$)
2: **for** (node $n^- \in \Delta^-$) **do**
3: UNTRANSFORM($n^-, \mathcal{PG}_S$)
4: **end for**
5: $(G_S^-,\ \mathcal{PG}_S^-) \leftarrow$ remove all n^- in Δ^- from G_S and $\mathcal{PG}_S$
6: $(G_S^+,\ \mathcal{PG}_S^+) \leftarrow$ insert all n^+ in Δ^+ to G_S^- and $\mathcal{PG}_S^-$
7: **if** $\mathcal{PG}_S^+$ is cyclic **then**
8: **terminate with error** ▷ Additions invalidated G'_S
9: **end if**
10: **for** (node $n^+ \in \Delta^+$) **do**
11: UNTRANSFORM($n^+, \mathcal{PG}_S^+$)
12: **end for** ▷ At this point G has changed to $G^* = G'_S \leftarrow G^*_C \rightarrow G^*_T$
13: **return** $(G'_S \leftarrow G'_C \rightarrow G'_T) \leftarrow$ TRANSFORM($G^*, \mathcal{PF}_S$) ▷ Call batch algo [12]
14: **end procedure**
15: **procedure** UNTRANSFORM($n, \mathcal{PG}_S$)
16: *deps* $\leftarrow$ all nodes in all equiv. classes in $\mathcal{PG}_S$ with incoming edges from $EQ(n)$
17: **for** node *dep* in *deps* **do**
18: **if** *dep* is *transformed* **then**
19: UNTRANSFORM($dep, \mathcal{PG}_S$)
20: **end if**
21: **end for**
22: *neighbors* $\leftarrow$ all nodes in $EQ(n)$
23: **for** node *neighbor* in *neighbors* **do**
24: **if** n is *transformed* **then**
25: APPLYINVERSERULE(n) ▷ Throw exception if Def. 16 is violated
26: **end if**
27: **end for**
28: **end procedure**

A for-loop (line 2) untransforms every deleted node in Δ^- (in our case class `a`) by calling method UNTRANSFORM. Line 16 places `c` in `deps` as this is dependent on $EQ(\texttt{a})$ ($EQ(x)$ returns the appropriate equivalence class of node x) and calls UNTRANSFORM recursively on line 19. The equivalence class of `c` has no dependent elements in $\mathcal{PG}_S$ and on line 25, calling APPLYINVERSERULE untransforms `c` by applying the inverse forward rule of Rule (c) (Fig. 2). Note that with an appropriate bookkeeping data structure (not explained here) this method is aware of all previous rule applications and applies the correct inverse forward rule to the same match used previously by the forward transformation. The rule application can only fail if building the pushout complement was not possible due to dependencies in G_T which would violate the forward precedence preserving property for graph triples (Def. 16). In this case, an appropriate exception is thrown. After returning from the recursive call, `a` is untransformed by using the inverse forward rule of Rule (b). The resulting graph triple is depicted in Fig. 4(b). Next, all changes in Δ_S are used to update G_S and $\mathcal{PG}_S$ on lines 5 and 6. Adding elements may result in a cyclic precedence graph indicating cyclic context de-



(a) Original triple G with a merged representation of all changes in Δ_S

(b) Triple after untransforming deletions and their dependencies

(c) Triple G^* after handling changes and untransforming their dependencies

(d) Final updated triple G' with all changes propagated

Fig. 4. Consistent change propagation from source to target domain

pendencies and the algorithm would terminate with an error on line 8. For our running example, the updated precedence graph $\mathcal{PG}_S$ is acyclic (Fig. 5(b)), so the algorithm continues untransforming all elements that potentially depend on newly added elements as context. The only dependent element of `d`, which is `b`, is untransformed by calling UNTRANSFORM on line 11 which results in the triple G^* (Fig. 4(c)). Finally, on line 13 the intermediate triple G^* is passed to the TGG batch transformation algorithm of [12], which transforms all untransformed elements (with empty checkboxes) and returns the integrated and updated graph triple $G'_S \leftarrow G'_C \rightarrow G'_T$ depicted in Fig. 4(d).

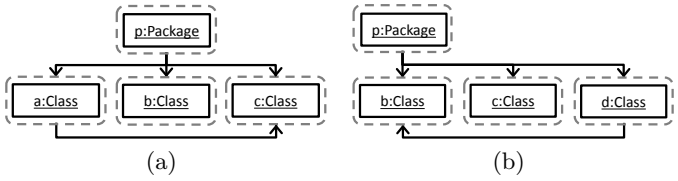


Fig. 5. $\mathcal{PG}_S$ for the original (left) and $\mathcal{PG}_S^{\perp}$ for the updated source graph (right)

Formal Properties of the Incremental Precedence TGG Algorithm

In this section, we prove that our algorithm retains all formal properties proposed in [14] and proved for the precedence-driven TGG batch algorithm of [12].

Definition 17 (Correctness, Completeness and Efficiency).
Correctness: *Given an input graph triple $G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)$ and an update $\Delta_S = G_S \leftarrow D \rightarrow G'_S$, the transformation algorithm either terminates with an error or produces a consistent graph triple $G'_S \leftarrow G'_C \rightarrow G'_T \in \mathcal{L}(TGG)$.*
Completeness: *$\forall\ G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG), G'_S \leftarrow G'_C \rightarrow G'_T \in \mathcal{L}(TGG)$ and a corresponding update $\Delta_S = G_S \leftarrow D \rightarrow G'_S$, the transformation algorithm produces a consistent triple $G'_S \leftarrow G^*_C \rightarrow G^*_T \in \mathcal{L}(TGG)$.*
Efficiency: *According to [14], a TGG batch transformation algorithm is efficient (polynomial runtime) if its runtime complexity class is $O(n_S^k)$, where n_S 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. In the incremental case, the algorithm is efficient if the synchronization runtime effort scales with the number of changes ($|\Delta^-| + |\Delta^+|$) and (potentially) dependent elements n_δ and not with the size of the updated graph triple, i.e., the incremental algorithm runs in the order of $O(n_\delta^k)$.*

All properties are defined analogously for backward transformations.

Theorem. *Algorithm 1 is correct, complete, and efficient for any source-local complete TGG (due to space restrictions we refer to Def. 13 in [11]) and forward precedence preserving graph triples (Def. 16).*

Proof.
Correctness: Lines 2 – 12 of the algorithm only invert previous rule applications. The order of rule applications is directed by the precedence graph (Def. 15), which represents potential dependencies between nodes, i.e., a node `x` has as dependencies all other nodes `y`, which may be transformed by applying a rule that matches `x` as context. These dependencies are potential dependencies as actual rule applications may select other nodes in place of `x`. Nevertheless, `y` potentially depends on `x`. The algorithm traverses to the very last dependency of every deleted/added node and applies the inverse of the rule used in a previous transformation. Demanding precedence preserving graph triples (Def. 16) guarantees that $\mathcal{PG}_S$ is sufficient to correctly revoke forward rules in a valid order. If an element on the target side is deleted by applying an inverse forward rule, although this element is still in use as context for another element, we know that the forward precedence preserving property is violated. This also guarantees that deleting elements via building a pushout complement (Def. 5) is always possible and cannot be blocked due to "dangling" edges. In combination with bookkeeping of previously used matches, it is guaranteed (Def. 6) that the resulting triple is in the state it was before transforming the untransformed node.
It directly follows that if the triple G was consistent, the remaining integrated part of G remains consistent. Since UNTRANSFORM inverts rule applications of a previous transformation, we know that the graph triple after line 12 is a valid intermediate graph triple produced by the batch transformation algorithm. As shown in [12], the precedence-driven TGG batch algorithm is correct (produces only correct graph triples or terminates with an error if no correct graph triple can be produced), so it directly follows that Algorithm 1 is also correct. □

Completeness: The correctness proof shows that the incremental update produces a triple via a sequence of rule applications that the batch algorithm could have chosen for a forward transformation of G'_S. Completeness arguments from [12] for the batch algorithm can, hence, be transferred to this algorithm.□
Efficiency: Efficiency is influenced mainly by the cost of (i) untransforming dependent elements of a deleted or added node (lines 2–4 and 10–12), (ii) updating the precedence graph and graph triple itself (lines 5 and 6), and (iii) transforming all untransformed elements via our precedence-driven TGG batch algorithm (line 13). The number of deleted/added nodes ($|\Delta^-|+|\Delta^+|$) and their dependencies is denoted by n_δ. Regarding UNTRANSFORM, a recursive depth-first search on the precedence graph $\mathcal{PG}_S$ is invoked starting at a certain node. Depth-first search has a worst-case complexity of $O(|V_{\mathcal{PG}_S}| + |E_{\mathcal{PG}_S}|)$ if the changed node was an (indirect) dependency of all other equivalence classes in $\mathcal{PG}_S$. If the algorithm encounters an already untransformed element on line 18, we know for sure that all subsequent elements are already untransformed and, therefore, can safely terminate recursion. Independent of the position of the changed element, UNTRANSFORM traverses every dependent element exactly once. Finally, applying the inverse operational rule (line 25) is (at most) of the same complexity as the appropriate previous rule application since the rule and match are already known. Considering both untransformation runs together, we know that n_δ elements are untransformed, and that every element is treated exactly once. Updating G_S on line 5 (6) involves deleting (inserting) $|\Delta^-|$ ($|\Delta^+|$) elements $m \in \Delta^-(\Delta^+)$ and updating, each time, a number of adjacent nodes (degree(m)). Updating $\mathcal{PG}_S$ has similar costs since elements have to be deleted (added) and updating the edge set of $\mathcal{PG}_S$ means to traverse all adjacent nodes of a deletion or addition in G_S and retrieve appropriate entries from $\mathcal{PF}_S$. Thus, the complexity of line 5 and 6 can be estimated with $O(|\Delta_S|)$, as Δ_S contains all nodes and edges that have been changed and, therefore, need to be revised. Finally, transforming the rest of the prepared graph (line 13) has $O(n_\delta^k)$ complexity [12]. Because only added elements, their dependencies, and the dependencies of removed elements have been untransformed, n_δ refers to these elements only, and not to all elements in G_S. The algorithm, therefore, scales with the number of changes and their dependencies and not with the size of the graph triple: $n_\delta \leq n$. □

5 Related Work

This section complements the discussion from Sect. 1 on related incremental synchronization approaches grouped according to their strengths.

Formality: Providing formal aspects for incremental updates that guarantee well-behavedness according to a set of laws or properties is challenging. Algebraic approaches such as lenses [2] and the framework introduced by Stevens [15] provide a solid basis for formalizing concrete implementations that support incremental model synchronization. Inspired by [2], a TGG model synchronization framework was presented in [9] that is correct and complete. The proposed algorithm, however, requires a complete remarking of the entire graph triple and

depends, therefore, on the size of the related graphs and not on the size of the update and affected elements. This is infeasible for an *efficient* implementation and the need for an improved strategy is stated as future work in [9].

Efficiency: In contrast to this formal framework, an incremental TGG transformation algorithm has been presented in [5], which exploits the correspondence model to determine an efficient update strategy. Although the batch mode of this algorithm has been formally presented in [6], the incremental version has not been fully formalized and it is unclear how the update propagation order is determined correctly for changes to elements that are not linked via the correspondence model to other elements. The authors describe an event-handling mechanism and so it can be assumed that model changes are instantly propagated. This allows for reduced complexity regarding dependencies between changes, but forbids the option of collecting a set of changes before propagating. This is, however, a requirement for scenarios in which changes are applied to models offline (i.e., without access to the related model) and the actual model synchronization must be performed later. The TGG interpreter described in [7] employs basically the same approach as [5], but additionally attempts to *reuse* elements instead of deleting and creating them. This is important as it prevents a loss of information that cannot be recovered by (re-)creating an element (user added contents). Unfortunately, this approach has also not been formalized and it is unclear whether the algorithm guarantees correctness and completeness. Nonetheless, this concept of reuse is crucial for industrial relevance and should be further investigated.

Concurrency: The challenge of dealing with *concurrent* changes to both domains has been discussed and investigated in [8,16]. A cascade of *propagate*, *calculate diff*, and *merge* steps is proposed that finally results in a consistent model. Extending our TGG algorithm based upon these ideas but retaining efficiency is also an important task of future research.

6 Conclusion and Future Work

A novel incremental algorithm for TGG has been presented that employs a precedence analysis to determine the effects of model changes. This involves not only determining which elements rely on deletions and, hence, must be untransformed, but also includes finding all elements that may rely on additions and also have to be untransformed. This must be achieved without compromising formal properties (i.e., correctness and completeness) while scaling efficiently with the size of the changes and their dependencies and not with the size of the overall graph.

Current restrictions include the lack of support for concurrent change propagation, which we plan to handle according to [16], and the formal requirement that edges can only be deleted or added together with adjacent nodes. Last but not least, we shall implement the presented incremental algorithm as an extension of our current implementation in our metamodelling tool eMoflon[1][1] and perform empirical performance assessments and comparisons with other implementations.

[1] `http:\\www.moflon.org`

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.) Informatik 2011. LNI, vol. 192, p. 281. GI, Bonn (2011)
2. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 304–318. Springer, Berlin (2011)
3. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In: Dwyer, M., Lopes, A. (eds.) FASE 2007, LNCS, vol. 4422, pp. 72–86. Springer, Berlin (2007)
4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Berlin (2006)
5. Giese, H., Hildebrandt, S.: Efficient Model Synchronization of Large-Scale Models. Tech. Rep. 28, Universitätsverlag Potsdam (2009)
6. Giese, H., Hildebrandt, S., Lambers, L.: Toward Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars. In: MoDeVVA 2010. pp. 19–24. IEEE, New York (2010)
7. Greenyer, J., Pook, S., Rieke, J.: Preventing Information Loss in Incremental Model Synchronization by Reusing Elements. In: France, R., Kuester, J., Bordbar, B., Paige, R. (eds.) ECMFA 2007. LNCS, vol. 6698, pp. 144–159. Springer, Berlin (2011)
8. Hermann, F., Ehrig, H., Ermel, C., Orejas, F.: Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 178–193. Springer, Berlin (2012)
9. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y.: Correctness of Model Synchronization Based on Triple Graph Grammars. In: France, R., Kuester, J., Bordbar, B., Paige, R. (eds.) MODELS 2011, LNCS, vol. 6981, pp. 668–682. Springer, Berlin (2011)
10. Kindler, E., Rubin, V., Wagner, R.: An Adaptable TGG Interpreter for In-Memory Model Transformations. In: Schürr, A., Zündorf, A. (eds.) Fujaba Days 2004. pp. 35–38. Paderborn (2004)
11. Klar, F., Lauder, M., Königs, A., Schürr, A.: Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift, LNCS, vol. 5765, pp. 141–174. Springer, Berlin (2010)
12. Lauder, M., Anjorin, A., Varró, G., Schürr, A.: Bidirectional Model Transformation with Precedence Triple Graph Grammars. In: Tolvanen, J.P., Vallecillo, A. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 287–302. Springer, Berlin (2012)
13. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Tinhofer, G. (ed.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Berlin (1994)
14. Schürr, A., Klar, F.: 15 Years of Triple Graph Grammars. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008, LNCS, vol. 5214, pp. 411–425. Springer, Berlin (2008)
15. Stevens, P.: Towards an Algebraic Theory of Bidirectional Transformations. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 1–17. Springer, Berlin (2008)
16. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Synchronizing Concurrent Model Updates Based on Bidirectional Transformation. SoSyM pp. 1–16 (2011), Online First^{TM}, January 4^{th} 2011