Leveraging Incremental Pattern Matching Techniques for Model Synchronisation

Erhan Leblebici[1(✉)], Anthony Anjorin[2], Lars Fritsche[1], Gergely Varró[1], and Andy Schürr[1]

[1] Technische Universität Darmstadt, Darmstadt, Germany
{erhan.leblebici,lars.fritsche,
gergely.varro,andy.schurr}@es.tu-darmstadt.de
[2] Universität Paderborn, Paderborn, Germany
anthony.anjorin@uni-paderborn.de

Abstract. Triple Graph Grammars (TGGs) are a declarative, rule-based approach to model synchronisation with numerous implementations. TGG-based approaches derive typically a set of *operational* graph transformations from direction-agnostic TGG rules to realise *model synchronisation*. In addition to these derived graph transformations, however, further runtime analyses are required to calculate the consequences of model changes in a synchronisation run. This part of TGG-based synchronisation is currently manually implemented, which not only increases implementation and tool maintenance effort, but also requires tool or at least approach-specific proofs for correctness. In this paper, therefore, we discuss how *incremental* graph pattern matchers can be leveraged to simplify the runtime steps of TGG-based synchronisation. We propose to outsource the task of calculating the consequences of model changes to an underlying incremental pattern matcher. As a result, a TGG-based synchroniser is reduced to a component reacting solely to appearing and disappearing matches. This abstracts high-level synchronisation goals from low-level details of handling model changes, providing a viable and unifying foundation for a new generation of TGG tools.

1 Introduction and Motivation

Bidirectional model synchronisation is a current challenge that is becoming increasingly relevant in numerous domains [4]. In our context, bidirectional model synchronisation refers to the task of keeping two models (called *source* and *target*) consistent by propagating changes (called *deltas*) applied to one of the models, i.e., by executing a forward or backward transformation to restore consistency. The task of implementing an *incremental* synchroniser with clear and precise semantics is non-trivial. In this paper, an incremental forward[1] synchroniser takes the old target model into account when propagating source deltas

[1] In the entire paper, symmetric statements that hold analogously in both forward and backward directions are only formulated in the forward direction for brevity.

© Springer International Publishing AG 2017
J. de Lara and D. Plump (Eds.): ICGT 2017, LNCS 10373, pp. 179–195, 2017.
DOI: 10.1007/978-3-319-61470-0_11

(and does not create the target model from scratch). Bidirectional transformation (*bx*) languages address this task via diverse techniques [4].

Triple Graph Grammars (TGGs) [16] as a *bx* language, represent a declarative, rule-based approach to model synchronisation based on the mature field of graph transformations [5]. TGG rules are direction-agnostic, describing how consistent pairs of source and target models can be created simultaneously. TGG-based model synchronisation typically involves compile time and runtime subtasks: At compile time, operational (forward and backward) graph transformation rules are derived from TGG rules. At runtime, consequences of deltas with respect to applying these operational rules are calculated, and consistency is restored by revoking invalidated rule applications from former runs and performing new ones. This step ideally guarantees *correctness*, i.e., that the resulting pair of a source and target model can be created by applying the TGG rules.

All state-of-the-art TGG-based synchronisation frameworks we are aware of [7,8,10,11,13,15] address the runtime step (i) in a simple but non-scalable manner, starting each time from scratch and considering the entire models [10], or (ii) by providing auxiliary dependency analyses over the source (target) model [13,15] or correspondences [7], or (iii) by applying practically useful but as yet informal heuristics without proofs of correctness for all possible cases [8,11]. Our observation is that the complexity in addressing the runtime steps of a TGG-based synchronisation is accidental and is caused by entangling high-level incremental propagation strategies with low-level details of how deltas and their transitive consequences must be handled efficiently and correctly.

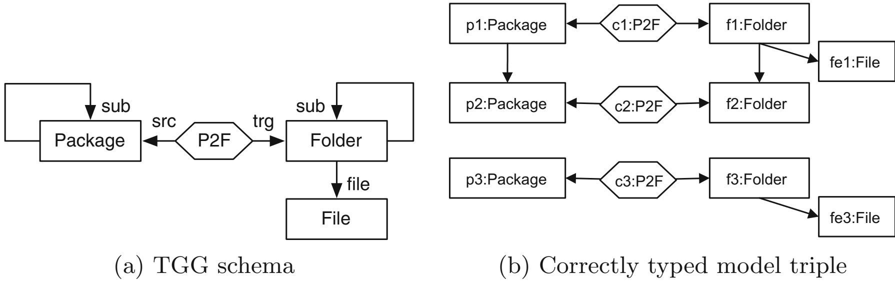
Incremental pattern matching techniques (e.g., [6]) provide a viable means of monitoring all matches of a given set of patterns in a host graph and thus can observe and report consequences of deltas. Although this naturally addresses the runtime requirements of TGG-based model synchronisation, incorporating incremental pattern matchers into TGGs has not yet been analysed up until now. Our contribution is, therefore, to integrate incremental pattern matching and TGG-based model synchronisation. Our aim is to provide a formal foundation for a new generation of TGG tools that can now leverage available incremental graph pattern matching tools [17,18]. We are able to reduce a TGG-based synchroniser to a relatively simple component that reacts to invalidated or available rule applications reported by its underlying incremental pattern matcher.

The paper is structured as follows: We present in Sect. 2 a compact but non-trivial synchronisation scenario and discuss the diverse delta propagation strategies. A novel concept for TGG-based synchronisers making use of incremental pattern matching techniques is presented intuitively in Sect. 3, and formalised in Sect. 4 with correctness arguments, shaping our main contribution. Related approaches and future work are discussed in Sects. 5 and 6, respectively.

2 Running Example and Preliminaries

A TGG specification consists of a *schema* and a set of *rules*. A schema is a triple of metamodels representing the abstract syntax of source, target, and correspondence models used for mappings between source and target elements.

Example: Our running example is *Ecore2HTML*.[2] The scenario addressed is the general usage of models for *editable* report generation; we only focus here on packages and their corresponding folders and files. The TGG schema for our running example is depicted in Fig. 1a. Source models are hierarchies of packages and target models (representing reports) are hierarchies of folders where folders may contain files. The single correspondence type (`P2F`) connects packages to folders. With respect to this TGG schema, a correctly typed model triple is depicted in Fig. 1b consisting of three packages `p1`, `p2`, and `p3` as well as three corresponding folders `f1`, `f2`, and `f3`. The outermost folders `f1` and `f3` additionally contain a file `fe1` and `fe3`, respectively.



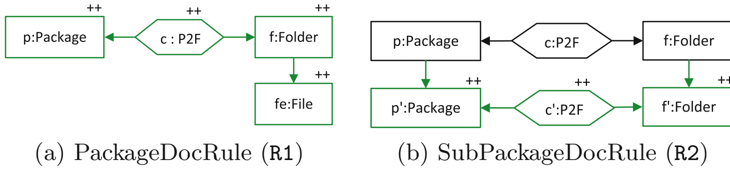
(a) TGG schema (b) Correctly typed model triple

Fig. 1. Schema and typed model triple

We use a compact syntax to represent TGG rules, merging both the precondition L and the postcondition R together in a single diagram. The elements in L (also referred to as context elements of the rule) are black, while elements in $R \setminus L$ (also referred to as created elements of the rule) are green with a `++` markup. Edges that are incident to a created node do not have explicit `++` markup as they must also obviously be created. A model triple is *consistent* with respect to a TGG, if it can be created by applying the rules of the TGG.

Example: Intuitively, what we want to specify is that packages correspond to folders and, additionally, that outermost packages require an extra file containing project-level documentation in their corresponding folder. To achieve this with TGGs, we need two rules: `PackageDocRule` (we shall also refer to this as `R1`) depicted in Fig. 2a, creates a package `p`, together with a corresponding folder `f` with a file `fe`. The created package and folder are also connected with a correspondence link `c`. `R1` has no context elements and can thus be applied to the empty triple. `SubPackageDocRule` (`R2`) depicted in Fig. 2b, requires a package `p` and a corresponding folder `f` (note how the correspondence link `c` is used to enforce this), and extends the package and folder hierarchies by creating a new sub-package `p'` and subfolder `f'`, connected via the correspondence link `c'`. In contrast to `R1`, rule `R2` has context elements and can only be applied to

[2] The entire synchronisation scenario including our excerpt is documented in the bx example repository at http://bx-community.wikidot.com/examples:ecore2html.



(a) PackageDocRule (`R1`) (b) SubPackageDocRule (`R2`)

Fig. 2. TGG rules for the running example

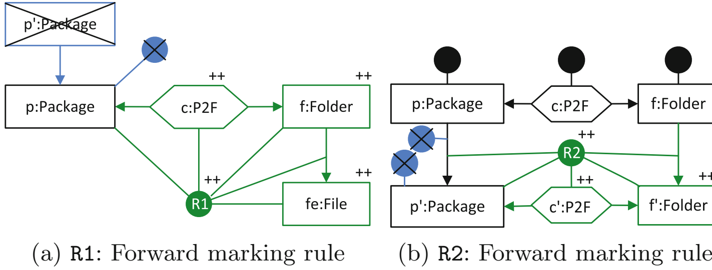
extend an existing model triple. In this sense, the triple depicted in Fig. 1b can be created with these rules and is thus consistent with respect to the TGG.

2.1 From TGG Rules to Operational Rules

When propagating a *source delta* in an existing model triple, an important step is handling newly created source elements by creating the corresponding structure in the target model. This is referred to as *marking* and captured as *forward marking rules*, which are derived from an original TGG rule. Explicit markers are used to keep track of which elements are processed/unprocessed in a model synchronisation run. Note that the correspondence model does not necessarily provide this information as correspondences in general do not have to exist for each element or multiple correspondences might exist for the same element.

Intuitively, the forward marking rule of a TGG rule does not create any source element but requires them, marks all created elements of its respective TGG rule, and requires that all context elements of the TGG rule be marked (e.g., by former applications of forward marking rules). Additionally, *Negative Application Conditions* (NACs) are used to ensure that source elements (the input elements in case of a forward synchronisation) are marked only once as they would be created once by the original TGG rules. Such NACs are referred to as *marker NACs*. Finally, an optional set of *filter NACs* is used to avoid invalid rule applications that would lead to a state where, e.g., certain edges can no longer be marked (such NACs are constructed automatically via static analysis techniques [9]).

Example: The forward marking rules for our running example are depicted in Fig. 3. All elements belonging to a NAC are depicted blue and crossed-out. For presentation purposes, markers are denoted by circles that are connected to elements (nodes and edges). The forward marking rule derived from `R1` (Fig. 3a) creates an `R1` marker connected to the package `p` and all created elements `c`, `f` and `fe`. The package `p` is demanded as context that must not already be marked (via the blue, crossed-out marker connected to it). If this forward marking rule were ever used to mark a sub-package `p`, it would be impossible to ever mark the incoming `sub` edge to `p`, as there does not exist a rule that creates an incoming `sub` edge to an existing package. Hence, a filter NAC is used to forbid the presence of such edges, i.e., the forward marking rule derived from `R1` can only mark outermost packages (as the original TGG rule `R1` can only create outermost packages). In the forward marking rule derived from `R2` (Fig. 3b), the context



(a) `R1`: Forward marking rule (b) `R2`: Forward marking rule

Fig. 3. Derived marking rules for the running example

elements of the original TGG rule are additionally required as already marked (no matter whether by the same or different markers).

2.2 Delta Propagation via Operational Rules

Given a consistent model triple and a source delta, the main task of forward synchronisation is to detect invalidated and available applications of forward marking rules. Invalidated applications (e.g., due to deleted context elements) must be *revoked* by deleting their created correspondence and target elements as well as obsolete markers. Conversely, available applications of forward marking rules lead to new correspondence and target elements with new markers.

Example: A simple source delta in our example is given by creating (deleting) a package, leading to an available (invalidated) application of the forward marking rule of `R1` or `R2` (depending on whether the package is an outermost package or not). A non-trivial source delta is given by creating a `sub` edge such that a former outermost package becomes a child package, e.g., creating a `sub` edge from `p2` to `p3` in Fig. 1b. In this case, an application of the forward marking rule of `R1` becomes invalid (the filter NAC is violated as the outermost package `p3` now becomes a child package). After deleting the obsolete marker of `p3` (and the corresponding target elements), an application of the forward marking rule of `R2` becomes available, i.e., `p3` can now be re-marked as a subpackage.

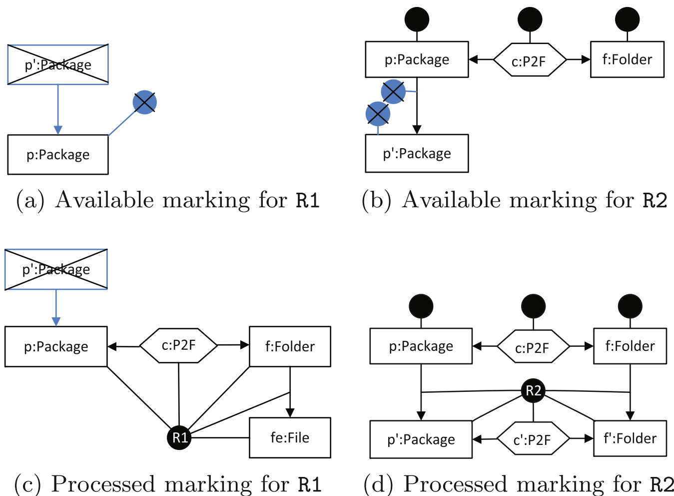
Existing TGG approaches differ from each other mainly concerning how invalid or available applications of operational rules are detected. In *precedence-driven* approaches [13,15], an auxiliary precedence analysis between model elements is performed (and maintained) to determine which model elements are potentially affected by deletions or creations of others. Alternatively, this analysis is performed between correspondences [7] (affected correspondences are calculated for a given source delta). Such hand-crafted analyses, however, either overestimate the actual dependencies as dependencies are retrieved at the type level [13], or underestimate them relying on additional information via user-interaction [15] or special correspondences [7]. A completely different strategy is to re-mark an existing triple from scratch and to complement missing markers in a final step [10]. This, however, makes the synchronisation process dependent

on the entire model size even if a small change is to be propagated. We argue in the following section that incremental pattern matchers naturally address the same tasks and can thus be exploited to simplify and to unify TGG approaches.

3 Using Incremental Pattern Matching Techniques for TGG-Based Model Synchronisation

Graph transformation applications depend highly on the discovery of occurrences of patterns in a host graph (called pattern matching). When an application operates on relatively large models where individual model changes usually concern only a small part, it is impractical to restart the pattern matching process each time from scratch. While auxiliary data structures (such as precedences, look-up tables, or rule application protocols) strive to avoid this in an application-specific manner, incremental pattern matching techniques (e.g., [6]) with recently developed practical solutions (e.g., [17,18]) address the same challenges in a generic and reusable manner. An incremental pattern matcher is capable of maintaining partial and complete matches of a given set of patterns found in a possibly changing host graph. Consequently, appearing or disappearing matches for a given set of patterns can be determined between two points in time (e.g., before and after changing the models). This enables client applications to focus on their business logic and high-level goals by reacting to appearance or disappearance of these matches (without searching and maintaining them manually). We discuss in the following how this vision can be realised for TGG-based model synchronisation.

For forward synchronisation, we propose to monitor two types of patterns in a model triple: *available* and *processed* markings with forward marking rules. In Fig. 4, these patterns are depicted for our running example. Basically, the pattern



(a) Available marking for R1 (b) Available marking for R2

(c) Processed marking for R1 (d) Processed marking for R2

Fig. 4. Patterns to be monitored by an incremental pattern matcher

for an available marking with a forward marking rule is the precondition (L) of the forward marking rule together with its filter NACs as well as marker NACs. The pattern for a processed marking comprises the postcondition (R) of the forward marking rule together with its filter NACs.

Assuming that matches for available and processed markings are monitored in a model triple by an incremental pattern matcher, Algorithm 1 represents our proposed concept for TGG-based model synchronisation in pseudo code. The procedure PropagateSourceDelta takes the following inputs: (i) a consistent model triple $G = G_S \leftarrow G_C \rightarrow G_T$ which is fully marked (from former runs), (ii) a source delta δ_S that changes G_S to G'_S, and (iii) an incremental pattern matcher *pm* that is initialised with G and monitors patterns for available and processed markings. The outcome of the procedure is a new model triple $G' = G'_S \leftarrow G'_C \rightarrow G'_T$ which reflects the source delta and is again consistent.

Algorithm 1. Model Synchronisation

```
 1: procedure PropagateSourceDelta(G, δ_S, pm)
 2:
 3:     G′ ← change G via δ_S                                        ▷ Phase 1
 4:     pm.update(δ_S)
 5:
 6:     while pm has a disappearing match for processed markings do  ▷ Phase 2
 7:         m⁻ ← choose a disappearing match for processed markings
 8:         (G′, δ⁻) ← revoke the fwd marking rule for m⁻ in G′
 9:         pm.update(δ⁻)
10:     end while
11:
12:     while pm has an appearing match for available markings do    ▷ Phase 3
13:         m⁺ ← choose a match for available markings
14:         (G′, δ⁺) ← apply the fwd marking rule for m⁺ in G′
15:         pm.update(δ⁺)
16:     end while
17:
18:     return G′
19:
20: end procedure
```

Overall, PropagateSourceDelta consists of three main phases:

Phase 1 (Line 3–4): The source delta is applied to the model triple and the incremental pattern matcher updates its matches for available and processed markings in the model triple.

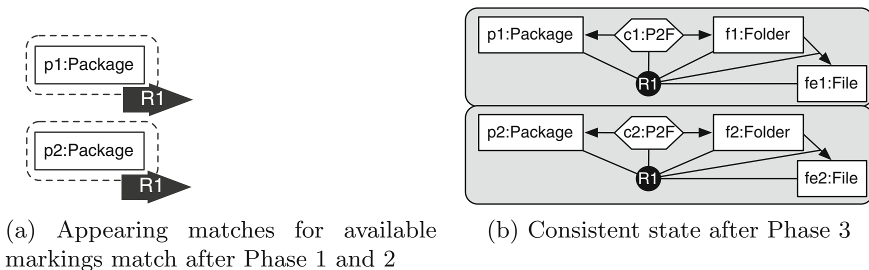
Phase 2 (Line 6–10): Disappearing matches of processed markings indicate that the respective applications of the forward marking rules from former runs are invalidated due to the source delta. Such invalidated rule applications must be revoked by deleting their created markers, correspondences, and target elements.

The incremental pattern matcher updates its matches after these deletions. Note that this can trigger further disappearances of processed marking patterns which again must be handled in the same manner until the pattern matcher does not report any further disappearing match for processed markings.

Phase 3 (Line 12–16): Appearing matches of available markings indicate that the respective forward marking rules are applicable. An arbitrary match is chosen and the forward marking rule is applied by creating a marker, correspondences, and target elements. The incremental pattern matcher updates its matches after these creations. Some matches for available markings disappear (at least the chosen match itself disappears) as some elements are now marked and violate marker NACs. Note that disappearing matches for available markings in this phase indicate progress in the synchronisation process (not to be confused with disappearing matches due to the source delta in Phase 2). Further matches for available markings can also appear due to the creation of new elements and must be handled in the same manner until the pattern matcher does not report any further appearing match for available markings.

In the following, we exemplify the intermediate and end results of Algorithm 1 based on two synchronisation runs with our running example.

Example (initial transformation): We first discuss an initial forward transformation of a source model to a target model. This is a special case of forward synchronisation where the entire source model is a delta applied to an empty triple. We assume that the incremental pattern matcher is initialised with an empty triple and that the source delta is the creation of two outermost packages. Applying this delta in Phase 1, two matches occur as available markings for the forward marking rule of `R1`, depicted in Fig. 5a via an `R1`-labeled arrow at the bottom-right corner of each match. No matches for processed markings disappear in this example as the model triple was initially empty, i.e., no rule application is to be revoked in Phase 2. Finally, applying forward marking rules for available markings in Phase 3, two matches occur for processed markings, depicted in Fig. 5b. The model triple is again consistent and fully marked.

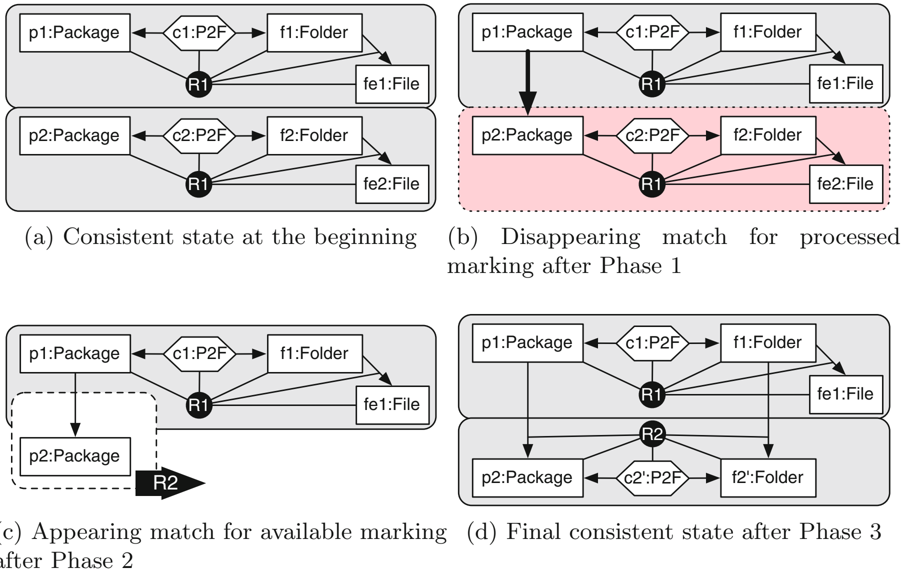


(a) Appearing matches for available markings match after Phase 1 and 2

(b) Consistent state after Phase 3

Fig. 5. Intermediate results of propagating two outermost packages

Example (creating a `sub` edge): We now assume that the incremental pattern matcher is initialised with the result of the previous example (Fig. 6a) and create a `sub` edge between the two packages making one of them, namely `p2`, a child package. After applying this delta, a match for a processed marking disappears as `p2`, being no longer an outermost package, violates the filter NAC. The `sub` edge violating the filter NAC is depicted bold in Fig. 6b while the rectangle with red filling and dashed border represents the disappearing match. Revoking the respective forward marking rule application of this match (i.e., deleting the marker of `p2` as well as its corresponding target elements) in Phase 2, a new marking becomes available for the forward marking rule of `R2`, depicted in Fig. 6c. Applying the forward marking rule for the available marking in Phase 3, the model triple is consistent and fully-marked again (Fig. 6d).



(a) Consistent state at the beginning

(b) Disappearing match for processed marking after Phase 1

(c) Appearing match for available marking after Phase 2

(d) Final consistent state after Phase 3

Fig. 6. Intermediate results of propagating a `sub` edge

Finally, it should be mentioned that the phases of Algorithm 1 represent a straightforward approach without any heuristics to improve the quality of model synchronisation, especially with regard to information preservation capabilities. While Phase 2 revokes rule applications until no more disappearing match is reported, another possible reaction to disappearing matches would be to *repair* them (instead of revoking) as discussed in [7]. Alternatively, target elements deleted in Phase 2 can be *reused* in Phase 3 for new rule applications as proposed in [8]. In both cases, the goal is to preserve as much as possible from the older version of the target model. These extensions are orthogonal to our contribution

and can analogously be supported via an incremental pattern matcher. Basically, new types of reactions to appearing/disappearing matches are required for this but the idea of a reactive synchroniser concept remains the same.

4 Correctness of Delta Propagation

We formalise in the following triple graphs and forward marking rules via construction techniques over *functor* categories [5], and prove the correctness of Algorithm 1 under sufficient conditions. Our correctness proof is in line with that of [10,13,15] but eliminates the need of an entire marking from scratch as in [10], or an additional dependency analysis as in [13,15]. The added value of our formalisation thus lies in its simplified form.

A functor category $[\mathcal{S}, \mathcal{C}]$ consists of structure preserving arrows between objects of shape $\mathcal{S}$, constructed from objects and arrows in the host category $\mathcal{C}$. This is used to construct graphs from sets, marked graphs from graphs, and triple graphs from marked graphs. A marked graph is of the form $G \leftarrow \overline{G} \rightarrow M$ where the intermediate graph $\overline{G}$ indicates the part of G that is mapped to a marker graph M. A triple graph is then of the form $G_S \leftarrow G_C \rightarrow G_T$ where each of G_S, G_C, and G_T are marked graphs (the suffixes S, C, and T refer to the source, correspondence, and target domain, respectively). We provide our formalisation without attribute and type information in graphs for brevity. The formalisation, however, can compatibly be extended to attributed graphs [5] where typing can be captured as a slice category of triple graphs over a distinguished type object.

Definition 1 (Triple Graphs). *Let* **Sets** *be the category of sets and total functions. The category* **Graphs** *of graphs and graph morphisms is the functor category* $[\, E \rightrightarrows V \,, \mathbf{Sets}]$. *The category* **MGraphs** *of marked graphs is the functor category* $\left[G \leftarrow \overline{G} \rightarrow M, \mathbf{Graphs}\right]$. *The category* **Triples** *of triple graphs and triple graph morphisms is the functor category* $[G_S \leftarrow G_C \rightarrow G_T, \mathbf{MGraphs}]$.

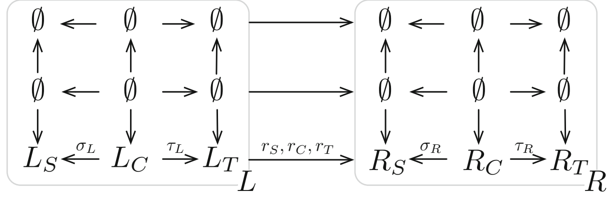
Definition 2 (Triple Rule and Derivation). *A* triple rule *is a morphism* $r : L \rightarrow R$ *in* **Triples**. *A* Negative Application Condition *(NAC) for a triple rule* $r : L \rightarrow R$ *is a morphism* $n : L \rightarrow N$ *in* **Triples**.

Given a triple rule r *with a set* $\mathcal{N}$ *of NACs, a* direct derivation $G \overset{r@m}{\Longrightarrow} G'$ *(or just* $G \overset{r}{\Longrightarrow} G'$*) is given by the pushout* $(r' : G \rightarrow G', m' : R \rightarrow G')$ *of* $r : L \rightarrow R$, *and* $m : L \rightarrow G$ *in* **Triples** *if* $\nexists\, n : L \rightarrow N \in \mathcal{N}, \exists\, n' : N \rightarrow G, m = n' \circ n$.

A derivation $G \overset{*}{\Longrightarrow} G'$ *with a set* $\mathcal{R}$ *of triple rules is a sequence of* k *direct derivations* $G \overset{r_1}{\Longrightarrow} G_1 \overset{r_2}{\Longrightarrow} \cdots \overset{r_k}{\Longrightarrow} G'$, $r_1, r_2, \cdots, r_k \in \mathcal{R}$ *(*$G' = G$ *for* $k = 0$*).*

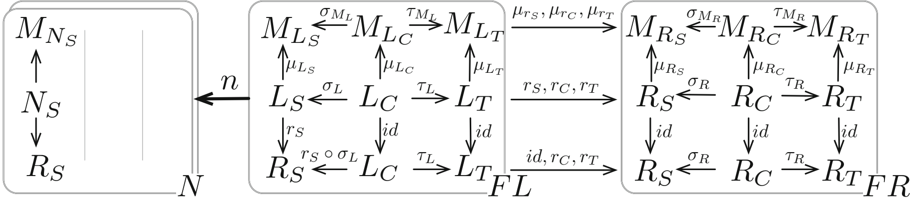
In a TGG, the original rules do not create or require markers (e.g., Fig. 2) but forward marking rules do (e.g., Fig. 3).

Definition 3 (Triple Graph Grammar). *A* triple graph grammar *(TGG) is a finite set $\mathcal{R}$ of triple rules, each of the form depicted to the right. A TGG is* source progressive *if $r_s \neq id$. The language of a TGG is given by $\mathcal{L}(TGG) := \{G \mid \exists\ G_\emptyset \stackrel{*}{\Longrightarrow} G$ with $\mathcal{R}\}$ where $G_\emptyset$ is the empty triple graph.*



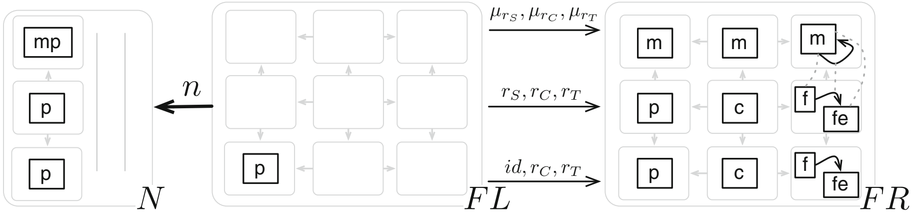
A forward marking rule fr for a TGG rule r as introduced in Definition 3 (i) does not create the elements in $R_S \backslash L_S$ but requires them, (ii) creates all elements in $R_C \backslash L_C$ and $R_T \backslash L_T$, (iii) requires markers for all elements in L_S, L_C, and L_T, (iv) forbids markers for elements in $R_S \backslash L_S$, and finally (v) creates new markers for all elements in $R_S \backslash L_S$, $R_C \backslash L_C$, and $R_T \backslash L_T$ (cf. Fig. 3).

Definition 4 (Forward Marking Rule). *Given a $TGG = \mathcal{R}$, the forward marking rule $fr : FL \rightarrow FR$ for each $r \in \mathcal{R}$ has the structure as follows:*

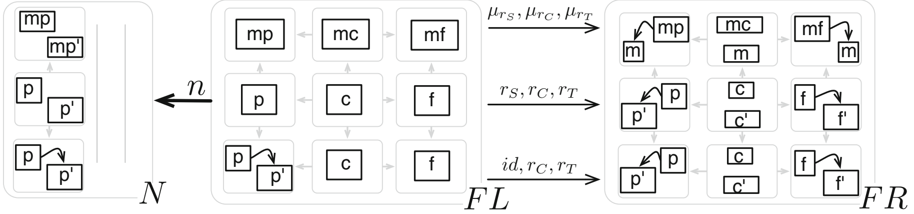


For $X \in \{S, C, T\}$, the graphs M_{L_X} are isomorphic to L_X. The graphs M_{R_X} extend M_{L_X} by an extra node m. All nodes in $R_X \backslash L_X$ are mapped by μ_{R_X} to m. Every edge in $R_X \backslash L_X$ is mapped to an edge added to M_{R_X} so that μ_{R_X} is structure preserving. Furthermore, the forward rule fr is equipped with a set $\mathcal{N}$ of marker NACs. Marker NACs $n : FL \rightarrow N$ extend the source components of FL (all other components remain the same and are not depicted explicitly) to forbid the presence of markers for any element in $R_S \backslash L_S$.

Example: The diagram below depicts the forward marking rule for `R1` (Fig. 3a) formally. In FL a node `p` is required in the source component and the presence of markers for `p` is forbidden by the marker NAC N. In FR, the correspondence and target elements (`c`, `f`, `fe`, and an edge between `f` and `fe`) are created together with a marker `m` in each component. Note that all nodes that are created in the original TGG rule ($R_X \backslash L_X$) are mapped to one marker `m` in each component (shown explicitly via dashed lines for `f` and `fe`) where edges are mapped to a self-edge of markers (e.g., the edge between `f` and `fe`).



Furthermore, the diagram below depicts the forward marking rule for `R2` (Fig. 3b) formally. In this case, markers (`mp`, `mc`, `mf`) are required in FL for context elements (`p`, `c`, `f`, respectively) while created elements as well as created/forbidden markers are analogous to the previous diagram.



Next, we define the source language of a TGG, i.e., source graphs G_S for which a triple $G_S \leftarrow G_C \rightarrow G_T$ exists in $\mathcal{L}(TGG)$. Accordingly, *the forward marked language* of a TGG is given by derivations via forward marking rules beginning with triples of the form $G_S \leftarrow \emptyset \rightarrow \emptyset$.

Definition 5 (Source Language, Forward Marked Language). *Given a* $TGG = \mathcal{R}$*, the* source language *of* TGG *is defined as*
$\mathcal{L}(TGG)_S = \{G_S \mid \exists\, G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)\}$.

The forward marking grammar *fwd(TGG) for TGG consists of the set fwd($\mathcal{R}$) of forward marking rules for $\mathcal{R}$.*

The forward marked language *of fwd(TGG) for* $G_S \in \mathcal{L}(TGG)_S$ *is defined as*
$\mathcal{L}(fwd(TGG), G_S) = \{G \mid \exists\; (G_S \leftarrow \emptyset \rightarrow \emptyset) \overset{*}{\Longrightarrow} G \text{ with } fwd(\mathcal{R})\}$.

A triple graph is fully marked if every node/edge in its source, correspondence, and target components are mapped to a marker node/edge. Fully marked triples are of interest as every derivation with triple rules from a TGG can be traced back to a unique derivation with the respective forward marking rules where the result is fully marked. We furthermore introduce an operator Φ to extract an unmarked triple graph from a marked one.

Definition 6 (Fully Marked and Unmarked Triple Graphs). *Let* $G = G_S \leftarrow G_C \rightarrow G_T$ *be a triple graph.* G *is* fully marked *if each of its marked graphs* G_X*,* $X \in \{S, C, T\}$*, is of the form* $X \overset{id}{\leftarrow} X \rightarrow M_X$*. A triple graph is* unmarked *if each of its marked graphs is of the form* $X \leftarrow \emptyset \rightarrow \emptyset$ *and* $\Phi(G)$ *denotes the unmarked triple graph obtained from* G *by removing its markers.*

Fact 1 (Bijection Between TGG and Forward Marking Grammar). *Given a TGG with a forward marking grammar fwd(TGG),* $\exists\; G = G_S \leftarrow G_C \rightarrow$

$G_T \in \mathcal{L}(TGG) \iff \exists\ \overline{G} \in \mathcal{L}(fwd(TGG), G_S)$ *where* $\overline{G}$ *is fully marked and* $G = \Phi(\overline{G})$.

Proof (Sketch). This is a standard operationalisation result for TGGs [10], applied to marked graphs and marking rules. Basically, a forward marking rule fr for a TGG rule r maps exactly those source elements to a marker that are created by r. Both r and fr create the same correspondence and target elements whereas fr maps each of them additionally to markers (cf. Definitions 3 and 4). □

We now introduce the notion of *confluence*, a well-known property of graph grammars that can be checked statically [5]. Every partial derivation of a confluent graph grammar can be completed to a derivation that produces the same result, and this ensures (together with Fact 1) that applying forward marking rules (e.g., as in Algorithm 1) results always in a fully marked triple graph.

Definition 7 (Confluence). *A pair* $G_1 \overset{*}{\Longleftarrow} G \overset{*}{\Longrightarrow} G_2$ *of derivations with a set* $\mathcal{R}$ *of triple rules is* confluent *if there exists a* G' *together with derivations* $G_1 \overset{*}{\Longrightarrow} G'$ *and* $G_2 \overset{*}{\Longrightarrow} G'$.

We refer to $\mathcal{R}$ *as* confluent *if all pairs of its derivations are confluent.*

Fact 2 (Confluence of Forward Marking Rules). *Given a* $TGG = \mathcal{R}$ *with forward marking rules fwd(*$\mathcal{R}$*), source language* $\mathcal{L}(TGG)_S$ *and* $G_S \in \mathcal{L}(TGG)_S$*, if fwd(*$\mathcal{R}$*) is confluent, then every derivation* $\tilde{d} = G_S \leftarrow \emptyset \rightarrow \emptyset \overset{*}{\Longrightarrow} \widetilde{G}$ *with fwd(*$\mathcal{R}$*) can be extended to a derivation* $\overline{d} = G_S \leftarrow \emptyset \rightarrow \emptyset \overset{*}{\Longrightarrow} \widetilde{G} \overset{*}{\Longrightarrow} \overline{G}$*, where* $\overline{G}$ *is fully marked.*

Proof. $G_S \in \mathcal{L}(TGG)_S \overset{Definition\,5}{\Longrightarrow} \exists\, G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG) \overset{Fact\,1}{\Longrightarrow} \exists$ fully marked $\overline{G} \in \mathcal{L}(fwd(TGG), G_S)$. Extension of $\tilde{d}$ to $\overline{d}$ follows from Definition 7 with $G = G_S \leftarrow \emptyset \rightarrow \emptyset, G_1 = \widetilde{G}, G_2 = G' = \overline{G}$. □

To ensure practical applicability of TGGs, forward marking rules are often enriched with *filter NACs* (cf., Fig. 3). The goal of filter NACs is to make a non-confluent forward marking grammar confluent [9]. That is, filter NACs block only those derivations that do not lead to a fully marked graph.

Definition 8 (Filter NACs). *Given a* $TGG = \mathcal{R}$ *and its forward marking rules fwd(*$\mathcal{R}$*), each forward marking rule* $fr : FL \rightarrow FR$ *in fwd(*$\mathcal{R}$*) can be equipped with a set* $\mathcal{N}'$ *of* filter NACs *that do not block any derivation* $G_S \leftarrow \emptyset \rightarrow \emptyset \overset{*}{\Longrightarrow} \overline{G}$ *in fwd(*$\mathcal{R}$*) without filter NACs where* $\overline{G}$ *is fully marked. Filter NACs are, therefore, only used to ensure that fwd(*$\mathcal{R}$*) is confluent. We refer to, e.g., [9] for a construction technique.*

Next, we define matches that are to be monitored in a forward synchronisation process as discussed in Sect. 3 for Algorithm 1. Matches for available markings are simply matches for direct derivations via forward marking rules, while matches for processed markings appear after a direct derivation via forward marking rules and forbid the violation of filter NACs.

Definition 9 (Matches for Available and Processed Markings). *Let $fr : FL \to FR$ be a forward marking rule and $\mathcal{N}'$ the set of filter NACs of fr. Given a triple graph G, for each possible direct derivation $G \overset{fr@m}{\Longrightarrow} G'$, we refer to m as a* match for available marking *with fr. A match* for processed marking *with fr is given by the comatch $m' : FR \to G'$. A match m' for processed marking is valid if it does not violate any filter NAC, i.e., $\forall n : FL \to N \in \mathcal{N}', \nexists n' : N \to G$ such that $n' \circ n = m' \circ fr$.*

Deltas represent changes to graphs (to a source graph in case of a forward synchronisation) and lead to appearance or disappearance of matches for available or processed markings.

Definition 10 (Delta). *A* delta *is a span of graphs and graph morphisms $\delta_X = G_X \leftarrow \Delta_X \to G'_X$. Elements in $G_X \backslash \Delta_X$ and $G'_X \backslash \Delta_X$ are referred to as* deleted *and* created*, respectively, by δ_X.*

Finally, the following Theorem states the correctness of Algorithm 1, i.e., that the result G' is a fully marked triple where $\Phi(G') \in \mathcal{L}(TGG)$. We require a source progressive TGG (Definition 3) with a confluent forward marking grammar (Definition 7) as sufficient conditions.

Theorem 1 (Correctness of Delta Propagation). *Given a source progressive $TGG = \mathcal{R}$, let $G \in \mathcal{L}(fwd(TGG), G_S)$ be a fully marked triple, and $\delta_S = G_S \leftarrow \Delta_S \to G'_S$ a delta. We assume a pattern matcher pm monitoring matches in G for available and processed markings with forward marking rules in fwd($\mathcal{R}$). If fwd($\mathcal{R}$) is confluent and $G'_S \in \mathcal{L}(TGG)_S$, the result of* propagateSourceDelta$G$, δ_S, pm *(Algorithm 1) is a triple graph G' such that $\Phi(G') \in \mathcal{L}(TGG)$, i.e.,* propagateSourceDelta$G$, δ_S, pm *is correct.*

Proof. We use in the following the intermediate results of Phase 1, 2, and 3 in propagateSourceDelta$G$, δ_S, pm for the proof.

Phase 1: $G \in \mathcal{L}(fwd(TGG), G_S)$ is fully marked. Hence, there exists a derivation $d_1 = (G_S \leftarrow \emptyset \to \emptyset) \overset{*}{\Longrightarrow} G$ with fwd($\mathcal{R}$). The comatch of each direct derivation in d_1 leads to a match for processed marking (Definition 9). When changing G_S according to δ_S, such matches can disappear while new matches for available markings can appear.

Phase 2: For each disappearing match for processed markings, direct derivations with fwd($\mathcal{R}$) in d_1 are revoked. This step does not create any match for processed markings and thus terminates when no more matches disappear (d_1 has finitely many direct derivations). With the remaining direct derivations from d_1 (i.e., direct derivations that have not been revoked), we get a derivation $d_2 = G'_S \leftarrow \emptyset \to \emptyset \overset{*}{\Longrightarrow} \widetilde{G}$ with fwd($\mathcal{R}$).

Phase 3: Given that $G'_S \in \mathcal{L}(TGG)_S$ and fwd($\mathcal{R}$) is confluent, d_2 can be extended to a derivation $d_3 = G'_S \leftarrow \emptyset \to \emptyset \overset{*}{\Longrightarrow} \widetilde{G} \overset{*}{\Longrightarrow} G'$ by applying available marking matches in any order where the result is a fully marked triple G' due to

Fact 2 and $\Phi(G') \in \mathcal{L}(TGG)$ due to Fact 1. Termination of this step is guaranteed as the TGG is source progressive (Definition 3), i.e., each direct derivation marks at least one source element and reduces the number of source elements for which available matches can appear (due to marker NACs in Definition 4). □

5 Related Work

Bx approaches can be classified mainly in three categories: relational (e.g., [3]), programming-based (e.g., [12]), and rule-based approaches such as TGGs. Our contribution exploits the rule-based characteristics of TGGs and governs a synchronisation process via appearing/disappearing rule matches. We believe, nevertheless, that at least relation-based approaches can be inspired by our contribution to monitor (un-)satisfied relations between two models.

We have already discussed in Sect. 2 different TGG approaches [7,8,10,11,13,15] to emphasise what our contribution exactly simplifies with regard to the runtime tasks of a TGG-based synchronisation. While our focus is the underlying technology of TGG-based synchronisation, practical extensions including repair rules [7] or reusing deleted elements [8] are useful to improve the quality of TGG-based synchronisation (by keeping as many elements as possible from the older versions of models). An incremental pattern matcher can facilitate such extensions by introducing new types of reactions to appearing/disappearing matches. Furthermore, the confluence requirement in our formalisation can be relaxed via static analysis techniques [1] such that model synchronisation can have more than one possible valid result. This is orthogonal to our contribution and non-confluence has not been considered due to space limitations.

Our work is inspired by model synchronisation applications operating with incremental pattern matching techniques. Most closely, Bergmann et al. [2] demonstrate how to transform a source delta to a target delta by using incremental pattern matchers. The transformation step, however, is a manually implemented forward transformation, while TGGs introduce a grammatical and declarative consistency notion and the forward transformation is automatically derived together with its backward counterpart.

6 Conclusion and Future Work

We have presented a novel concept for TGG-based model synchronisation based on an underlying incremental pattern matcher. A TGG-based synchroniser is reduced to a component that simply reacts to appearing or disappearing matches monitored by its underlying incremental pattern matcher. We have formalised our synchroniser concept and shown its correctness under sufficient conditions.

Future work has already started on, first and foremost, implementing a feature-complete TGG tool with our concept and evaluating its capabilities with comparisons to other TGG and bx approaches. Incremental pattern matchers strive for scalable computations of matches whose runtime depends on delta size

and not on model size. Our expectation, therefore, is that improved scalability will be a second advantage besides the simplified synchroniser concept when integrating incremental pattern matchers into TGGs. This is yet to be validated by extending the comparison of TGG-based model synchronisation tools [14].

We are also interested in (incremental) consistency checking and model integration (two-way model synchronisation potentially with conflict resolution) with TGGs. Such advanced use cases become tractable and can be handled uniformly after abstracting TGGs from low-level details of match maintenance.

Acknowledgement. This work has been funded by the German Federal Ministry of Education and Research within the Software Campus project GraTraM at TU Darmstadt, funding code 01IS12054.

References

1. Anjorin, A., Leblebici, E., Schürr, A., Taentzer, G.: A static analysis of non-confluent triple graph grammars for efficient model transformation. In: Giese, H., König, B. (eds.) ICGT 2014. LNCS, vol. 8571, pp. 130–145. Springer, Cham (2014). doi:10.1007/978-3-319-09108-2_9
2. Bergmann, G., Ráth, I., Varró, G., Varró, D.: Change-driven model transformations - change (in) the rule to rule the change. SoSym **11**(3), 431–461 (2012)
3. Cicchetti, A., Ruscio, D., Eramo, R., Pierantonio, A.: JTL: a bidirectional and change propagating transformation language. In: Malloy, B., Staab, S., Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 183–202. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19440-5_11
4. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: a cross-discipline perspective. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02408-5_19
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2006)
6. Forgy, C.: Rete: a fast algorithm for the many patterns/many objects match problem. Artif. Intell. **19**(1), 17–37 (1982)
7. Giese, H., Hildebrandt, S.: Efficient model synchronization of large-scale models. Technical report, HPI at the University of Potsdam (2009)
8. Greenyer, J., Pook, S., Rieke, J.: Preventing information loss in incremental model synchronization by reusing elements. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 144–159. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21470-7_11
9. Hermann, F., Ehrig, H., Golas, U., Orejas, F.: Efficient analysis and execution of correct and complete model transformations based on triple graph grammars. In: Bézivin, J., Soley, R.M., Vallecillo, A. (eds.) MDI 2010, pp. 22–31. ACM (2010)
10. 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.) MODELS 2011. LNCS, vol. 6981, pp. 668–682. Springer, Heidelberg (2011). doi:10.1007/978-3-642-24485-8_49
11. Klassen, L., Wagner, R.: EMorF - a tool for model transformations. ECEASST **54** (2012)

12. Ko, H.-S., Zan, T., Hu, Z.: BiGUL: a formally verified core language for putback-based bidirectional programming. In: PPEPM 2016, pp. 61–72. ACM, New York (2016)
13. Lauder, M., Anjorin, A., Varró, G., Schürr, A.: Efficient model synchronization with precedence triple graph grammars. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 401–415. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33654-6_27
14. Leblebici, E., Anjorin, A., Schürr, A., Hildebrandt, S., Rieke, J., Greenyer, J.: A comparison of incremental triple graph grammar tools. ECEASST **67** (2014)
15. Orejas, F., Pino, E.: Correctness of incremental model synchronization with triple graph grammars. In: Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 74–90. Springer, Cham (2014). doi:10.1007/978-3-319-08789-4_6
16. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995). doi:10.1007/3-540-59071-4_45
17. Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-IncQuery: an integrated development environment for live model queries. Sci. Comput. Program. **98**(1), 80–99 (2015)
18. Varró, G., Deckwerth, F.: A rete network construction algorithm for incremental pattern matching. In: Duddy, K., Kappel, G. (eds.) ICMT 2013. LNCS, vol. 7909, pp. 125–140. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38883-5_13