



Extended Triple Graph Grammars with Efficient and Compatible Graph Translators

Felix Klar, Marius Lauder, Alexander Königs, and Andy Schürr

Technische Universität Darmstadt, Real-Time Systems Lab,
Merckstr. 25, 64283 Darmstadt, Germany
{felix.klar, marius.lauder, alexander.koenigs,
andy.schuerr}@es.tu-darmstadt.de
<http://www.es.tu-darmstadt.de>

Abstract. Model-based software development processes often force their users to translate instances of one modeling language into related instances of another modeling language and vice-versa. The underlying data structure of such languages usually are some sort of graphs. Triple graph grammars (TGGs) are a formally founded language for describing correspondence relationships between two graph languages in a declarative way. Bidirectional graph language translators can be derived from a TGG, which maps pairs of related graph instances onto each other. These translators must fulfill certain compatibility properties with respect to the correspondence relationships established by their TGG. These properties are guaranteed for the original TGG approach as published 15 years ago. However, its expressiveness is pushed to the limit in most real world scenarios. Furthermore, the original approach relies on a parsing algorithm with exponential runtime complexity. In this contribution, we study a more expressive class of TGGs with *negative application conditions* and show for the first time that derived translators with a polynomial runtime complexity still preserve the above mentioned compatibility properties. For this purpose, we introduce a new characterization of well-formed TGGs together with a new translation rule scheduling algorithm that considers *dangling edges* of input graphs.

Key words: triple graph grammars, bidirectional model transformation, negative application conditions, dangling edge condition

1 Introduction

Languages in general and computer languages in particular are used to describe artifacts and processes of the real world. These descriptions can be thought of as models of the real world or in a very general term as *data*. Languages have specific rules to form these models and model elements are mostly typed to assign semantics to a category of elements. Specific domains of the world require specific languages which are nowadays often called *domain specific languages* (DSL). These DSLs are tailored to the specific needs of a group of people which should benefit from using these DSLs. But, two languages that refer to the same

domain and share the same artifacts of the real world, may though have different representations. There are various reasons for the concurrent usage of different languages in engineering projects. Domain experts (humans or computers) do not “speak” one language but different ones. Or, one language might be more adequate to express domain specific facts than the other. So, it is a common scenario that an engineer manually translates an instance of one language into an instance of another language and spends a lot of time to keep these language instances then in a consistent state. But, translation between languages is time-consuming and requires experts in both languages. Therefore, it is reasonable to facilitate this process. Accordingly, (semi-)automated translators are needed that assist their users in keeping related instances of languages in a consistent state. Due to the reasons listed above, it seems to be quite natural to develop a language for the language translation domain, too. *Triple graph grammars (TGGs)* are a formally founded language that is used to build bidirectional translators easily.

In the following section, we briefly introduce the reader to the world of TGGs and state the challenges the designers of TGG-based languages have to face: (1) TGGs must be expressive enough for the specification of complex relationships between pairs of languages and (2) they must allow for the derivation of efficiently working translators that are compatible with the related TGG specification. Subsequently, we show in Sect. 3 that we can improve the expressiveness of TGGs by introducing *negative application conditions (NACs)* and nevertheless still derive compatible translators for a certain family of TGGs. In Sect. 4 we add the concept of checking *dangling edge conditions (DECs)* to our graph translation algorithms. DEC-checks resolve rule application conflicts, prevent the construction of illegal graphs, and thus guarantee a polynomial runtime behavior of the derived translators. Based on these achievements we present an efficient translation algorithm in Sect. 5 that is still compatible with TGGs of the formerly defined family of TGGs. Section 6 analyzes how related approaches deal with the challenges of bidirectional model transformation languages in general and TGG-based languages in particular. Finally, we conclude this contribution in Sect. 7 and state open challenges to be solved in future work.

2 TGGs with Negative Application Conditions

In 1994 the first publication appeared that introduced the concept of triple graph grammars [1]. It allows for the specification of correspondence relationships between two languages of graphs. TGGs are grammars that generate graph triples by applying productions of the grammar to an input graph triple (axiom). The three graphs are often called *source* and *target graph* representing the elements of the related languages, and *correspondence graph* containing correspondence links. The correspondence links are elements of a language, too—the correspondence language that relates elements of the source and target language. TGGs have been invented to support translation of documents based on related graph-like data structures. Related documents are, e.g., design documents of a piece of software (e.g., class diagrams) and design documents of a data management

environment (e.g., relational database schema) that persists the data which is processed by the software. Class diagrams and database schemata are closely related as we will see later on. TGGs enable to explicitly establish mappings between documents by means of traceability links that contain additional information about the transformation process. TGGs are used to build bidirectional-working formal language translators. In addition, they allow to check consistency of related documents and to efficiently propagate changes in one document to restore consistency in the corresponding document.

In the context of the model driven architecture (MDA) [2, 3] TGGs are used for the declarative specification of bidirectional model transformations. One TGG serves as input for the derivation of a pair of model translators that transform a model of one language into a corresponding model of the other language and vice versa. The correspondence relations are needed for traceability purposes and they encode additional information about the translation process itself. This allows for the realization of incremental updates that are required if changes occur in the involved models. In this contribution we will use the terms “graph”, “node”, etc. of the graph grammar world, but a translation of all definitions and theorems to the MDA world simply requires the replacement of the introduced terms by their MDA counterparts like “model”, “object”, ... [4].

Now, we will explain TGGs by example and afterwards describe how translators are derived from a TGG. Finally, we will summarize the fundamental properties of TGGs and derived translators as stated in [5]. The example used throughout this contribution is the well-known mapping between class diagrams, also referred to as “source domain”, and relational database schemata, also referred to as “target domain”. This example is discussed in numerous publications of the model and graph transformation domain [6, 3]. We have reduced the example to a core part that is used to explain the ideas presented in this contribution. We will now start building the triple graph grammar TGG_{CDDS} that specifies the mapping between class diagrams and database schemata.

2.1 TGG schema

TGGs consist of a *TGG schema* which describes the structural dependencies between the elements of the two related languages and the correspondence language. Figure 1 shows the TGG schema of the triple graph grammar TGG_{CDDS} . The schema defines the structural correspondences of the source and target domain. The domain of class diagrams defines classes and attributes. Classes may have subclasses and contain zero to many attributes. Attributes are ordered by the successor/predecessor relationship “Precedes”. Multiple inheritance is not supported in this example, so each subclass may have only one superclass. The domain of relational database schemata defines tables and columns. Each table may contain a number of columns. Columns are ordered by the successor/predecessor relationship “Precedes”.

The elements of both domains are related via so called *correspondence link types* which are located in the *link domain*. Link types are denoted as hexagonal elements. Throughout this contribution we call instances of link types (*TGG*)

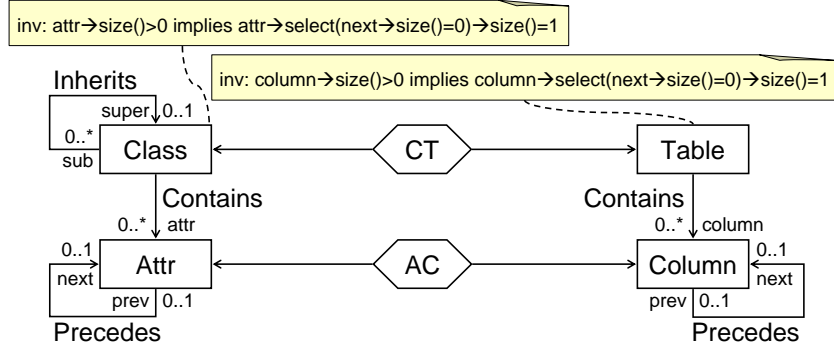


Fig. 1. TGG schema of TGG_{CDDS} that relates class diagrams and database schemata.

links. The link type CT (*class-table-relation*) maps classes to tables, whereas the link type AC (*attribute-column-relation*) maps attributes to columns. Let us have a closer look at the graph constraints of the given schema. Graph triples must fulfill these constraints in order to be valid. Successors of attributes and columns are realized via the “Precedes” edge type. The multiplicity “0..1” of the edge type’s end “next” denotes that each attribute and column respectively may have a successor, but need not. These are multiplicity constraints that might be expressed by OCL invariants $inv_{CD:P:n:mult}$ and $inv_{DS:P:n:mult}$. In addition, the OCL invariants shown in Fig. 1 constrain the number of elements without successor ($inv_{CD:C}$). Similarly, tables and columns are constrained by $inv_{DS:T}$.

2.2 TGG productions

In addition to the TGG schema, a set of *TGG productions* is specified. TGG productions are often called *TGG rules*. However, we stick to the term *TGG productions* to avoid clashes with derived *translation rules*. Productions define a language of consistent graph triples, i.e., they describe how related triples of graphs may evolve simultaneously. Each production consists of a graph pattern—its left-hand side L —that looks for a corresponding match (redex) in a graph triple. Applied to a redex a TGG production adds a copy of the elements of its right-hand side R that are not already part of L to the regarded graph triple.

The TGG productions of TGG_{CDDS} are depicted in Fig. 2 using a shorthand notation for graph productions. Instead of showing both left-hand and right-hand side as two separate parts of a production, both sides are merged [7]. The elements contained in the left-hand and in the right-hand side of the production $L \cap R$ are denoted as black elements without any additional markup. These elements are *context elements* that define a pattern that matches the redex in a host graph triple to which the production is then applied. The elements contained in the right-hand side only $R \setminus L$ are denoted as green elements with an additional ++ markup. These elements are created during the application of a production

to its redex. Nodes of a TGG production that are created by the production and attached to a TGG link that is created by the production are called *primary nodes*. In general, TGG productions may create additional *secondary nodes* (i.e., *non-primary nodes*) that are directly or transitively connected with the primary node. As the TGG formalism does not allow for the deletion of elements, elements that are contained in the left-hand side only $L \setminus R$ need not to be visualized in TGG productions. Groups of elements that form a NAC are crossed-out. No match for these elements must be found in the host graph; otherwise the matching NAC blocks the application of its production.

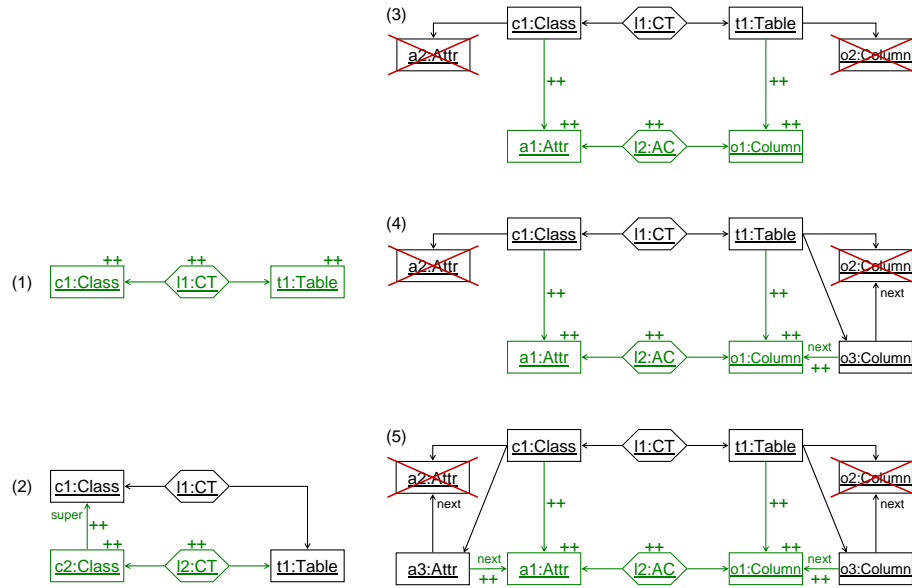


Fig. 2. TGG productions (1) and (2) that relate classes and tables. TGG productions (3), (4), and (5) that relate attributes and columns.

Productions (1) and (2) produce classes and tables, whereas productions (3), (4), and (5) produce attributes and columns. Production (1) is applicable in any situation, as it has no required context elements. It creates a new class in the source domain, a new table in the target domain, and a new CT link in the link domain. Furthermore, the new CT link relates the just created elements of source and target domain. Production (2) creates a new class $c1$ and a new CT link $l2$ if a class $c1$ and a table $t1$ exist that are already related via a CT link $l1$. The new class is added to the inheritance structure of $c1$ and the new CT link relates $c2$ and $t1$. The following productions are more interesting as they make use of NACs, which guarantee that no invalid graph triples are produced by the productions. An invalid graph triple would either violate a multiplicity constraint or an OCL invariant of the TGG schema (cf. Fig. 1).

Production (3) is used to create the first attribute in a class of an inheritance structure and the corresponding column in a table. Due to the NACs present in both domains, this production is applicable only if the matched class and table do not already contain an attribute or a column. So, the NACs ensure that at most one attribute and column are created per class and table respectively that has no successor and, therefore, prohibit a violation of $inv_{CD:C}$ and $inv_{DS:T}$.

Production (4) creates the first attribute of a class of an inheritance structure, when another class of the inheritance structure already has an attribute. In this case the corresponding table already contains at least one column and, as columns are ordered, the new column $o1$ must be the successor of one of the existing columns. The NAC in the target domain ensures that the created column $o1$ is the successor of a column that does not have a successor yet. So, it ensures that the multiplicity constraint $inv_{DS:P:n:mult}$ of the endpoint “next” in the target domain holds after production application. The NAC in the source domain has the same effect as the NAC in production (3). The new column $o1$ has no successor and so enables the production to be applied for other classes that are part of the inheritance structure that have no attribute yet.

Production (5) is applicable in situations where a class and a table have at least one attribute and column respectively, i.e., productions (3) or (4) have been applied earlier on. The effect of both NACs is similar to the NAC of the target domain in production (4). They ensure that the multiplicity constraints $inv_{CD:P:n:mult}$ and $inv_{DS:P:n:mult}$ hold, by assigning the just created elements $a1$ and $o1$ as next element of elements that do not have successors yet.

2.3 Simultaneous Evolution of Graph Triples

We will now discuss the simultaneous evolution of graph triples by applying the TGG productions of TGG_{CDDS} to an empty graph triple. The resulting graph triple GT_6 (cf. Fig. 3 (a)) is an element of the language of the just introduced TGG. The graph triple is produced by applying production (1) to the empty graph triple and afterwards production (2) to the resulting graph triple. Finally, the following productions are applied: (3), (4), (5), and (5) again. Thus, GT_6 is produced by sequence $SEQ_6 = (p^{(1)}, p^{(2)}, p^{(3)}, p^{(4)}, p^{(5)}, p^{(5)})$.

Production (1) simultaneously creates a class $c1$ and a table $t1$ and relates them via link $l1$, whereas production (2) creates the subclass $c2$ and relates it with the already existing table $t1$ via link $l2$. Note that the context in which productions are applied is important (cf. Sect. 5). The context in which production (3) can be applied is either $(c1, l1, t1)$ or $(c2, l2, t1)$. In our example, we first choose subclass $c2$ as context. Production (3) is applicable in the context of $c2$ because neither class $c2$ nor table $t1$ contain elements at this moment. As a consequence attribute $a3$ and column $o3$ together with their link $l3$ are created. From now on, $p^{(3)}$ is neither applicable in the context of class $c1$ nor $c2$ due to the NAC in the target domain that blocks because table $t1$ already contains column $o3$. In addition, its NAC in the source domain blocks in the context of class $c2$ because $c2$ contains attribute $a3$. In the next step we apply production (4) in the context of class $c1$. It is applicable because no attribute

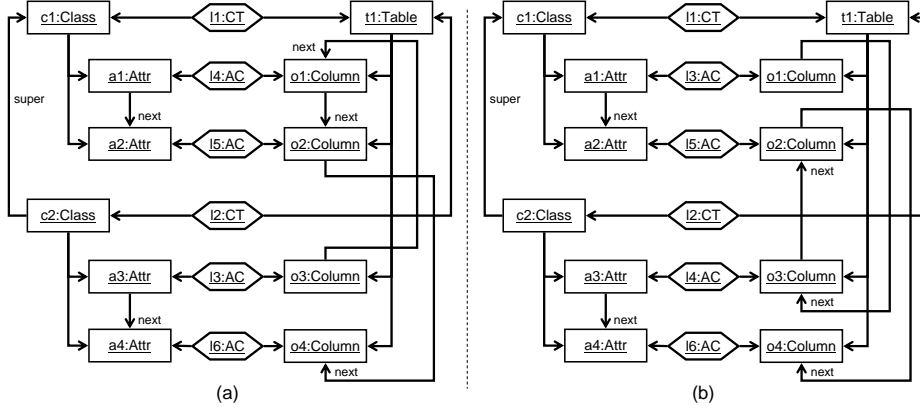


Fig. 3. Schema compliant graph triples GT_6 and GT_6^* produced by TGG_{CDDS} .

is currently present in class $c1$ and one column $o3$ that has no successor yet, is present in table $t1$. So, attribute $a1$ and column $o1$ are created and related via link $l4$ and column $o1$ is set as successor of column $o3$. The next two applications of production (5) are again possible in the context of class $c1$ and $c2$. We decide to first apply $p^{(5)}$ in the context of $c1$ and afterwards in the context of $c2$. This leads to sequence $SEQ_6 = (p^{(1)}@∅, p^{(2)}@c1, p^{(3)}@c2, p^{(4)}@c1, p^{(5)}@c1, p^{(5)}@c2)$ and the final situation depicted in Fig. 3 (a).

The order of columns located in the target domain of GT_6 is $o3$, $o1$, $o2$, and $o4$. This order is determined by the sequence of production applications in a particular context as described above. If the example above is changed such that production (3) is applied in the context of class $c1$ and production (4) in the context of class $c2$ then $SEQ_6^* = (p^{(1)}, p^{(2)}, p^{(3)}@c1, p^{(4)}@c2, p^{(5)}@c1, p^{(5)}@c2)$. The order of the columns would be $o1$, $o3$, $o2$, and $o4$ leading to graph triple GT_6^* (cf. Fig. 3 (b)). We consider these graph triples semantically equivalent according to TGG_{CDDS} because the relative order of attributes in the inheritance structure of classes $c1$ and $c2$ is not destroyed and the relative order of columns of a relational database does not matter in a “pure” relational calculus.

2.4 Language Translators based on TGGs

A TGG can be compiled into a pair of *forward and backward graph translators* ($FGTs/BGTs$). The generated translators take a graph of the *input domain*, either source or target, and produce a graph triple that consists of the given input graph, the corresponding graph of the *output domain*, either target or source, and the correspondence graph which connects the related source and target graph elements. A translator mainly consists of a set of *graph translation rules* and an algorithm that controls the stepwise translation of a given input graph into the related output graph. Each forward/backward graph translation rule (FGT/BGT rule), often called *operational rule*, is directly derived from a

single TGG production¹. Therefore, TGG productions are split into sets of *local rules* and the aforementioned *translation rules* [1]. Local rules generate graphs of the input domain ensuring that only valid graphs are produced. Hence, local rules are applicable only if NACs are not violated.

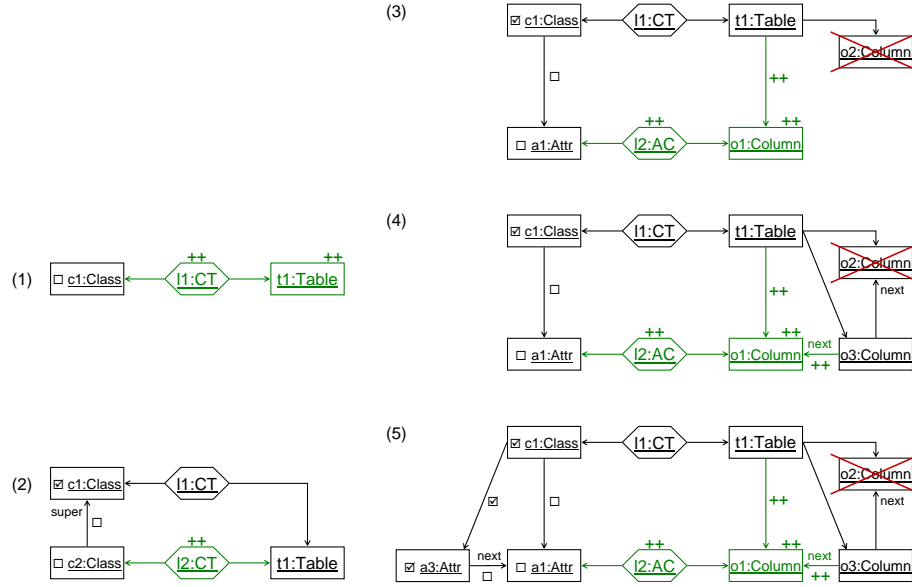


Fig. 4. Forward translation rules derived from TGG_{CDDS} .

Figure 4 shows the forward translation rules derived from the productions of TGG_{CDDS} . Translation rules contain elements of source, correspondence, and target domain. The elements of the input domain are read-only as they have been created earlier by a corresponding local rule. Consequently, translation rules only produce elements of the output and correspondence domain. Empty checkboxes denote that the elements next to them are not yet translated, i.e., no corresponding elements in the output domain have been created by another translation beforehand. A translator will mark all elements of the input domain as “translated”, right after a translation rule has been applied successfully. Elements that were context elements in a TGG production must be translated before a rule is applicable. This is denoted as enabled checkboxes placed next to or inside these elements. NACs of the input domain may be omitted under certain conditions, as we will learn in Sect. 3.2, whereas NACs of the output domain are retained.

A translation algorithm applies the operational rules to the input graph such that it simulates the simultaneous evolution of the computed graph triple with

¹ For a detailed description of the derivation process we refer to [8].

respect to the given set of TGG productions. Therefore, a translator must be able to determine the order in which elements of the input graph would have been created by a sequence of TGG productions. Guessing the proper choice is one of the difficulties that arise when the simultaneous evolution of graph triples is simulated by a translator. In general, this computation of an appropriate sequence of rules requires a graph grammar parsing algorithm with exponential runtime behavior [9]. Interleaved with the stepwise computation of a sequence of TGG productions and the resulting derivation of the input graph the corresponding sequence of operational rules is executed to generate the related output and correspondence graph instances. For further details concerning a formalization of this process the reader is referred to [1, 10]. An example that shows how an FGT translates an input graph is presented in Sect. 5.

2.5 Fundamental Properties of TGGs and Translators

As stated in previous work TGGs should not violate certain design principles in order to be “useful” in practice [5]. Derived translators should be *efficient* and they must be *compatible* with their TGG which in addition must be *expressive* enough. According to [5], efficient translators have polynomial space and time complexity $O(m \times n^k)$ with m = number of rules, n = size of input graph, and k = maximum number of elements² of a rule. This requirement is based on two worst case assumptions: (1) n^k is the worst-case complexity of the pattern matching step of a graph translation rule with k elements. (2) Without starting the pattern matching process for a selected rule we cannot determine whether this rule can be used to translate a just regarded element. As a consequence we require that derived translators do somehow process the elements of an input graph in a given order such that no element has to be regarded and translated more than once. Selecting always somehow the “right” translation rules we do not have to explore multiple translation alternatives using, e.g., a depth-first backtracking algorithm for that purpose. Compatible translators are *consistent* and *complete* with respect to their TGG. Consistency is guaranteed if a translator translates an input graph into a graph triple GT that is always an element of the language $\mathcal{L}(TGG)$ defined by the TGG. Completeness demands that for every graph triple GT that is an element of $\mathcal{L}(TGG)$, a translator is able to produce this graph triple (or an equivalent one) given the graph of the graph triple, which belongs to the translator’s input domain. Expressiveness finally requires that the TGG formalism is able to capture all important consistency relationships between studied pairs of graph languages.

As a consequence the original TGG approach is continuously extended such that it supports, e.g., handling of attributed typed graphs as well as the definition of productions with NACs. We have learned that NACs are additional preconditions that must be satisfied so a production is applicable. They are, e.g., used to prohibit the construction of graph triples that violate constraints defined in the schema of the source and target domain. But, after more than 15 years

² In [5] it is *nodes*. We expand this to *elements* which means nodes and edges.

of TGG research activities we still have problems to handle TGGs with NACs appropriately, i.e., to find the right compromise between expressiveness of TGG productions on the one hand and the introduced consistency, completeness, and efficiency properties of derived translators on the other hand. This contribution introduces, therefore, for the first time a subclass of TGGs with NACs in Sect. 3 that allow for the derivation of efficiently working compatible graph translators. Essentially, the definition and application of TGG productions is restricted in such a way that the here introduced rule application control algorithm (cf. Sect. 5) never has to resolve rule application conflicts by making an arbitrary choice. For this purpose we first replace NACs on the input graph side of a translation by graph constraints, thereby avoiding positive/negative rule application conflicts³. Positive/positive rule application conflicts are then eliminated by inspecting the context of those nodes more closely that are just translated by a given rule (cf. Sect. 4)⁴. Inspired by the definition of the double-pushout (DPO) graph grammar approach [11] a new kind of “dangling edge condition” is introduced that blocks the translation of nodes with afterwards still untranslated incident edges under certain conditions.

3 Formalization of Constrained TGGs with NACs

In the preceding section we have informally introduced TGGs with NACs. Furthermore, we already mentioned that [5] does already guarantee consistency but not completeness for the derived translators without introducing a backtracking algorithm, i.e., without trading efficiency for completeness. We will now identify a sort of TGG productions with NACs which do not lead to positive/negative FGT/BGT rule application conflicts for any input graph. This is a first step towards our goal to eliminate all kinds of FGT/BGT rule application conflicts and thereby to guarantee completeness of derived translation functions. For this purpose we extend the TGG formalism as introduced in [1] by NACs that are used to preserve the integrity of graph triples (i.e., resulting graph triples never violate constraints—neither temporary) without destroying the fundamental propositions proved in [1]. Therefore, TGGs will operate on typed constrained graphs and support NACs in a way that derived translators do not violate the mentioned compatibility properties. Permitted NACs will be ignored on the input graph of translation rules assuming that integrity violations of input graphs are captured before a translation process starts.

3.1 Constrained and Typed Graph Grammars with NACs

We start with the basic definitions of constrained, typed graphs on the basis of [11], which are then used for the definition of TGGs that generate triples of typed and constrained graphs in Sect. 3.2.

³ A positive/negative rule application conflict of two operational rules r and r' w.r.t. specific redexes exists if r creates a graph element that is forbidden by a NAC of r' .

⁴ Two operational rules w.r.t. to specific redexes constitute a positive/positive rule application conflict if both rules compete to translate the same graph element.

Definition 1. *Graphs, Graph Morphisms, and Graph Operators.*

A quadruple $G := (V, E, s, t)$ is a graph with $\text{elements}(G) := V \cup E$, where

- (1) V is a finite set of nodes (or vertices), E is a finite set of edges, and
- (2) $s, t : E \rightarrow V$ are functions assigning sources and targets to edges.

Let $G := (V, E, s, t), G' := (V', E', s', t')$ be two graphs.

A pair of functions $h := (h_V, h_E)$ with $h_V : V \rightarrow V'$ and $h_E : E \rightarrow E'$

is a graph morphism from G to G' , i.e., $h : G \rightarrow G'$, iff

- (3) $\forall e \in E : h_V(s(e)) = s'(h_E(e)) \wedge h_V(t(e)) = t'(h_E(e))$

Furthermore, the operators \subseteq for subgraph, \cup for union of graphs with gluing of nodes and edges (with same identifiers), and \setminus for the deletion of the removal of graph elements, are defined as usual, and with $h : G \rightarrow G'$ being a morphism, $h(G) \subseteq G'$ denotes that subgraph in G' which is the image of h .

Definition 2. *Typed Graph and Type Preserving Graph Morphisms.*

A type graph is a distinguished graph $TG := (V_{TG}, E_{TG}, s_{TG}, t_{TG})$.

V_{TG} and E_{TG} are called the node and the edge type alphabets, respectively.

A tuple (G, type) of a graph G together with a graph morphism $\text{type} : G \rightarrow TG$ is called typed graph. G is called instance of TG and TG is called type of G .

Given typed graphs G, G' a typed graph morphism $g : G \rightarrow G'$ is type preserving iff the diagram shown in Fig. 5 (a) commutes.

$\mathcal{L}(TG)$ is the set of all graphs of type TG .

In the following we assume that all graphs with suffix ‘‘S’’, ‘‘C’’, and ‘‘T’’ have type graphs TG_S, TG_C , and TG_T , respectively. Furthermore, we assume that all morphisms between graphs of the same type are type preserving.

Definition 3 introduces constrained graphs. The regarded constraints are typed graph constraints (e.g., OCL invariants) in the spirit of [11], i.e., Boolean formulae over atomic typed graph constraints. A typed graph G fulfills a typed graph constraint c , e.g., if c is evaluated to *true*.

Definition 3. *Constrained Typed Graph.*

A type graph TG with a set of constraints \mathcal{C} defines a subset $\mathcal{L}(TG, \mathcal{C}) \subseteq \mathcal{L}(TG)$ of the set of all graphs of type TG that fulfill the given set of constraints \mathcal{C} . The empty graph $G_\emptyset \in \mathcal{L}(TG, \mathcal{C})$. Furthermore, $\bar{\mathcal{L}}(TG, \mathcal{C}) := \mathcal{L}(TG) \setminus \mathcal{L}(TG, \mathcal{C})$ denotes the set of all graphs of type TG that violate a constraint in \mathcal{C} .

Based on this definition of constraints we will now define *graph productions with NACs* and graph rewriting. An important property of these productions is that they do not delete any graph elements, i.e., left-hand side L is a subset of right-hand side R . Therefore, they are called monotonic productions.

Definition 4. *Monotonic Graph Productions with NACs.*

The set of all monotonic productions $\mathcal{P}(TG, \mathcal{C})$ with negative application conditions \mathcal{N} for a type graph TG with a set of constraints \mathcal{C} is defined as follows:

- (L, R, \mathcal{N}) $\in \mathcal{P}(TG, \mathcal{C})$ iff
- (1) $L, R \in \mathcal{L}(TG, \mathcal{C}) \wedge L \subseteq R$
- (2) $\mathcal{N} \subseteq \mathcal{L}(TG) \wedge \forall N \in \mathcal{N} : N \supseteq L$

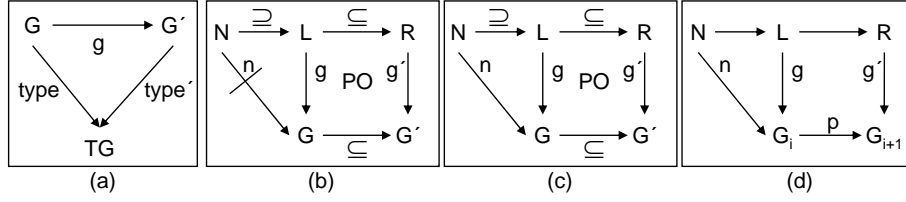


Fig. 5. Diagrams used in Def. 2, Def. 5, Def. 6, and in proof of Corollary 3.

Definition 5. *Graph Rewriting for Monotonic Productions with NACs.*

A production $p := (L, R, \mathcal{N}) \in \mathcal{P}(TG, \mathcal{C})$ rewrites a graph $G \in \mathcal{L}(TG)$ into a graph $G' \in \mathcal{L}(TG)$ with a redex (match) $g : L \rightarrow G$, i.e., $G \overset{p@g}{\rightsquigarrow} G'$ iff

- (1) $g' : R \rightarrow G'$ is defined by building the pushout diagram presented in Fig. 5 (b)
- (2) $\neg(\exists N \in \mathcal{N}, n : N \rightarrow G : n|_L = g)$, i.e., there exists no N such that mapping n is identical to g w.r.t. the left-hand side graph L
- (3) all morphisms are type preserving

We will limit productions with NACs to so-called *integrity-preserving productions* in Def. 6 such that NACs are only used to prevent the creation of graphs which violate the set of constraints \mathcal{C} . These productions have the important properties that (1) given a valid input graph, a valid output graph is produced, (2) if productions where NACs are eliminated produce a valid graph then the input graph is also valid, and (3) a production that would block due to a NAC otherwise would always produce an invalid graph. Due to contraposition of (1) all invalid output graphs are derived from invalid input graphs. So, integrity-preserving productions produce only invalid output graphs if the input graph was already invalid. Moreover, contraposition of (2) (i.e., (2*)) states that invalid input graphs result in invalid output graphs even if NACs are eliminated from a production; i.e., productions with or without NACs do not repair invalid graphs.

Definition 6. *Integrity-Preserving Productions.*

Let p be a production $(L, R, \mathcal{N}) \in \mathcal{P}(TG, \mathcal{C})$ and $p^- := (L, R, \emptyset)$ being the corresponding production of p where all negative application conditions have been eliminated. Then, p is integrity-preserving iff

- (1) $\forall G, G' \in \mathcal{L}(TG) \wedge G \overset{p}{\rightsquigarrow} G' : G \in \mathcal{L}(TG, \mathcal{C}) \Rightarrow G' \in \mathcal{L}(TG, \mathcal{C})$
- (2) $\forall G, G' \in \mathcal{L}(TG) \wedge G \overset{p^-}{\rightsquigarrow} G' : G' \in \mathcal{L}(TG, \mathcal{C}) \Rightarrow G \in \mathcal{L}(TG, \mathcal{C})$
- (3) $\forall N \in \mathcal{N} : \text{the existence of the diagram depicted in Fig. 5 (c) with type preserving morphisms } n|_L = g = g'|_L \text{ implies } G' \in \overline{\mathcal{L}}(TG, \mathcal{C})$

Now, we show for TGG_{CDDS} that the source and target production components satisfy the conditions of Def. 6. We will limit the discussion to the source domain as situations in the target domain are almost identical. All productions satisfy condition (1), i.e., given a valid graph, productions with NACs produce only valid graphs (discussed already in Sect. 2.2). If productions (3), (4), and (5)

without NACs produce valid output graphs under certain conditions then the input graph was also valid due to the fact that none of the productions is able to repair invalid graphs even if their NACs are ignored. Productions (3) and (4) would produce even more (invalid) attributes without successor. Production (5) does not increase the number of attributes without successors that violate multiplicity constraint $inv_{CD:P:n:mult}$, but preserves the number of attributes that violate $inv_{CD:P:n:mult}$. Therefore, condition (2) is satisfied. Condition (3) of Def. 6 is satisfied because a blocking NAC of productions (3) and (4) prevents the production of an additional attribute without successor. Therefore, violation of $inv_{CD:C}$ is prevented. In addition, a blocking NAC in production (5) prevents violation of the multiplicity constraint of a “Precedes” edge $inv_{CD:P:n:mult}$. Therefore, the set of productions of TGG_{CDDS} is *integrity-preserving*.

Finally, Def. 7 states how graph grammars produce constrained typed graphs.

Definition 7. *Language of Typed and Constrained Graph Grammars.*

A graph grammar $GG := (TG, \mathcal{C}, \mathcal{P})$ over a type graph TG , a set of constraints \mathcal{C} , and a finite set of integrity-preserving productions $\mathcal{P} \subseteq \mathcal{P}(TG, \mathcal{C})$, with G_\emptyset being the empty graph, generates the following language of graphs

$$\mathcal{L}(GG) := \{G \in \mathcal{L}(TG, \mathcal{C}) \mid G_\emptyset \xrightarrow{p_1} G_1 \xrightarrow{p_2} \dots \xrightarrow{p_n} G_n = G \text{ with } p_1, \dots, p_n \in \mathcal{P}\}$$

The language that is generated by a graph grammar GG as defined in Def. 7 (i.e., the graphs that are producible by the grammar) is a subset of the set of graphs of type TG that fulfill the given set of constraints \mathcal{C} .

Corollary 1. $\mathcal{L}(GG) \subseteq \mathcal{L}(TG, \mathcal{C})$

Proof. Follows from Def. 6 and directly from Def. 7. □

Furthermore, the language $\mathcal{L}(GG^-)$ generated by a graph grammar where NACs of productions have been eliminated contains at least the same graphs as the language $\mathcal{L}(GG)$ generated by this graph grammar with NACs.

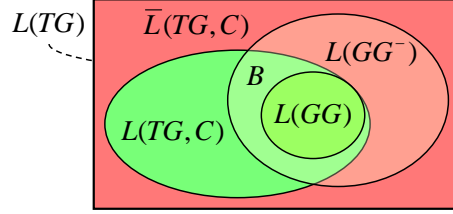
Corollary 2. *Let GG^- be a graph grammar derived from a graph grammar GG , where all negative application conditions of productions have been eliminated. Then $\mathcal{L}(GG^-) \supseteq \mathcal{L}(GG)$.*

Proof. Due to the fact that a valid application of a production p with NACs is also a valid application of the production p^- where NACs are ignored, $\mathcal{L}(GG^-)$ is at least as large as $\mathcal{L}(GG)$. □

Moreover, $\mathcal{L}(GG)$ is the intersection of the graphs producible by $\mathcal{L}(GG^-)$ and the set of graphs of type TG that fulfill the given set of constraints \mathcal{C} .

Corollary 3. *Let GG^- be a graph grammar derived from a graph grammar GG as defined in Def. 7, where all negative application conditions of productions have been eliminated. Then $\mathcal{L}(GG) = \mathcal{L}(GG^-) \cap \mathcal{L}(TG, \mathcal{C})$.*

Proof. Due to Corollary 1 and Corollary 2 the intersection of sets of graphs defined by $\mathcal{L}(TG, \mathcal{C})$, $\bar{\mathcal{L}}(TG, \mathcal{C})$, $\mathcal{L}(GG)$, and $\mathcal{L}(GG^-)$ looks like depicted:



Therefore, we only have to show that $\mathcal{B} := \mathcal{L}(GG^-) \cap \mathcal{L}(TG, \mathcal{C}) \setminus \mathcal{L}(GG)$ is empty. Let $G \in \mathcal{B}$, i.e., G is generated by a sequence of production applications

$$G_\emptyset \rightsquigarrow \dots \rightsquigarrow G_i \xrightarrow{p^-} G_{i+1} \rightsquigarrow \dots \rightsquigarrow G$$

with $p = (L, R, \mathcal{N})$ being a production of GG and $p^- = (L, R, \emptyset)$ being the corresponding production of GG^- such that $\exists N \in \mathcal{N}$ so the diagram shown in Fig. 5 (d) commutes, i.e., p is blocked by N , but p^- rewrites G_i into G_{i+1} .

$\Rightarrow G_{i+1} \in \bar{\mathcal{L}}(TG, \mathcal{C})$. This is a direct consequence of Def. 6 (3), which requires that the application of p^- produces a graph G_{i+1} , which violates at least one constraint if the application of p is blocked by its NAC N .

$\Rightarrow G \in \bar{\mathcal{L}}(TG, \mathcal{C})$. This is a direct consequence of Def. 6 (2*) because all graphs on the derivation path from G_{i+1} to G (including G) are invalid due to the fact that productions of GG^- preserve the property of a graph to violate some constraint. This leads to contradiction. \square

As a consequence of Def. 6 and due to Corollary 3, we can either check NACs during the execution of a (TGG) production to prohibit the violation of graph constraints immediately or check potentially violated graph constraints after a sequence of graph rewriting steps that simply ignore NACs; for a more detailed discussion of the relationship of (positive) pre- and postconditions of graph transformation rules and graph constraints we refer to [12].

3.2 Constrained and Typed Triple Graph Grammars with NACs

Having introduced definitions and properties of graph grammars with NACs for languages of typed constrained graphs we now present the corresponding definitions of TGGs with NACs for typed constrained graph triples. For this purpose we have to replace the definitions of simple graphs and graph grammars in [1] by the more elaborate definitions given in Sect 3.1.

Definition 8 describes the conditions typed graph triples must satisfy. A graph triple consists of three graphs. Each graph is in the set of graphs of a particular language, i.e., conforms to a graph schema defined by a certain type graph. In addition, two morphisms h_S and h_T relate elements of the correspondence graph with elements of the source and target graph. Constraints for correspondence graphs are disregarded in Def. 8, but can be added easily if needed.

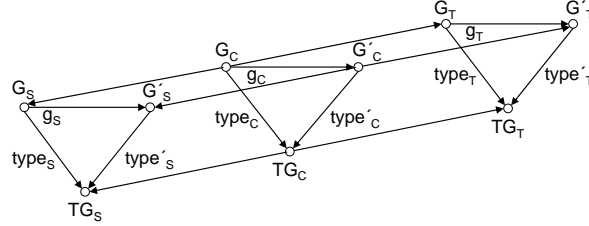
Definition 8. *Constrained Typed Graph Triple.*

Let $\mathcal{L}(TG_S, \mathcal{C}_S)$ and $\mathcal{L}(TG_T, \mathcal{C}_T)$ be languages of source and target graphs with constraints, whereas $\mathcal{L}(TG_C)$ defines a language of correspondence graphs that relate pairs of source and target graphs.

$GT := (G_S \xleftarrow{h_S} G_C \xrightarrow{h_T} G_T) \in \mathcal{L}(TGG)$ is a properly typed graph triple iff
(1) $G_S \in \mathcal{L}(TG_S, \mathcal{C}_S)$, (2) $G_C \in \mathcal{L}(TG_C)$, (3) $G_T \in \mathcal{L}(TG_T, \mathcal{C}_T)$
(4) $h_S : G_C \rightarrow G_S$, (5) $h_T : G_C \rightarrow G_T$

Definition 9. *Type Preserving Graph Triple Morphisms.*

A type graph triple $TGT := (TG_S \xleftarrow{type_S} TG_C \xrightarrow{type_T} TG_T)$ is a distinguished graph triple. TGT together with morphisms $type_S : G_S \rightarrow TG_S$, $type_C : G_C \rightarrow TG_C$, $type_T : G_T \rightarrow TG_T$ is called type of GT . A graph triple morphism (g_S, g_C, g_T) with $g_S : G_S \rightarrow G'_S$, $g_C : G_C \rightarrow G'_C$, $g_T : G_T \rightarrow G'_T$ is type preserving iff the so-called “toberone” diagram



commutes. $\mathcal{L}(TGT)$ is the set of all graphs of type TGT .

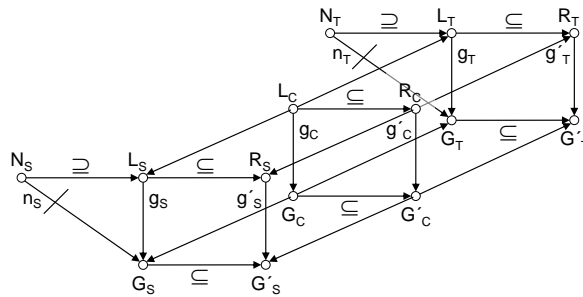
Now, we are ready to lift graph rewriting (cf. Def. 5) based on monotonic productions (cf. Def. 4) and integrity-preserving productions (cf. Def. 6) to graph triple rewriting in Def. 10.

Definition 10. *Integrity-Preserving Graph Triple Rewriting.*

Let $p := (p_S \xleftarrow{h_S} p_C \xrightarrow{h_T} p_T)$ be a production triple with NACs and

- (1) $p_S := (L_S, R_S, \mathcal{N}_S) \in \mathcal{P}(TG_S, \mathcal{C}_S)$ be an integrity-preserving production
- (2) $p_C := (L_C, R_C, \emptyset) \in \mathcal{P}(TG_C, \emptyset)$ be a simple production
- (3) $p_T := (L_T, R_T, \mathcal{N}_T) \in \mathcal{P}(TG_T, \mathcal{C}_T)$ be an integrity-preserving production
- (4) $h_S : R_C \rightarrow R_S$, $h_S|_{L_C} : L_C \rightarrow L_S$ and (5) $h_T : R_C \rightarrow R_T$, $h_T|_{L_C} : L_C \rightarrow L_T$

The application of such a production triple to a graph triple GT produces another graph triple GT' , i.e., $GT \xrightarrow{p} GT'$, which is uniquely defined (up to isomorphism) by the existence of the following “pair of cubes” diagram:



This diagram consists of commuting square-like subdiagrams only and contains a pushout subdiagram for each application of a production component (i.e., p_S , p_C , and p_T) to its corresponding graph component.

For the details of the definition and the proof that production triples applied to graph triples at a given redex always produce another graph triple uniquely defined up to isomorphism, cf. [1]. NACs introduced here do not destroy the constructions and proofs introduced in [1] due to the fact that they do not (further) influence the application of a production to a given graph (triple) after all NAC applicability checks have been executed. Based on the presented definitions we introduce *typed triple graph grammars* and their languages. For reasons of readability we omit the prefix “typed” throughout the rest of this contribution.

Definition 11. *Triple Graph Grammar and Triple Graph Grammar Language.* A triple graph grammar TGG over a triple of type graphs (TG_S, TG_C, TG_T) is a tuple (P, GT_\emptyset) , where P is the set of its TGG productions and GT_\emptyset is the empty graph triple. The language $\mathcal{L}(TGG)$ is the set of all graph triples that can be derived from $GT_\emptyset := (G_\emptyset \xleftarrow{\varepsilon} G_\emptyset \xrightarrow{\varepsilon} G_\emptyset)$ using a finite number of TGG production rewriting steps.

We can now show that a triple graph grammar TGG^- , where all NACs (that prevent the creation of graph triples that violate graph constraints) are removed from TGG productions, produces the same set of constrained graph triples that is produced by the unmodified triple graph grammar TGG .

Theorem 1. *With $\mathcal{L}(TGG)$ being the language of graph triples generated by a triple graph grammar TGG over (TG_S, TG_C, TG_T) we can show:*

- (1) for all $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$:
 $G_S \in \mathcal{L}(TG_S, \mathcal{C}_S)$, $G_C \in \mathcal{L}(TG_C)$, $G_T \in \mathcal{L}(TG_T, \mathcal{C}_T)$
- (2) with TGG^- being the triple graph grammar derived from TGG where all NACs of productions have been removed:
 $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG) \Leftrightarrow (G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG^-) \wedge (G_S, G_C, G_T) \in \mathcal{L}(TG_S, \mathcal{C}_S) \times \mathcal{L}(TG_C) \times \mathcal{L}(TG_T, \mathcal{C}_T)$

Proof. Follows from Def. 10 (which lifts graph to graph triple rewriting) and Corollaries 1 and 2. The proof is analogous to the proof of Corollary 3. \square

It is a direct consequence of Theorem 1 that checking of NACs can be replaced by checking integrity of generated graphs with respect to their sets of constraints and vice versa. This observation directly affects translators derived from a given TGG as follows: According to [1], a production triple p may be split into pairs of production triples (r_I, r_{IO}) , where r_I is an *(input-) local rule* and r_{IO} its corresponding *(input-to-output domain) translation rule*, with $GT \xrightarrow{p} GT' \Leftrightarrow GT \xrightarrow{r_I} GT_I \xrightarrow{r_{IO}} GT'$. Forward translation is based on (r_S, r_{ST}) , whereas (r_T, r_{TS}) is used in the reverse direction. To rewrite the source graph only, the *source-local production triple*, i.e., *source-local rule* $r_S := (p_S \xleftarrow{\varepsilon} (\emptyset, \emptyset, \emptyset) \xrightarrow{\varepsilon} (\emptyset, \emptyset, \emptyset))$ is applied. The *source-to-target domain translating production triple*, i.e., *forward*

graph translation rule r_{ST} keeps the source graph unmodified but adjusts the correspondence and target graph as follows: the effect of applying first r_S and then r_{ST} to a given graph triple is the same as applying p itself if (and only if) we keep the source domain redex, i.e., the morphism g'_S , fixed. Thanks to Theorem 1 the source component of r_{ST} does not have to check any NACs on the source graph as long as any regarded source graph does not violate any graph constraints, i.e., as long as it has been constructed by means of integrity-preserving productions only. As a consequence, we need no longer care about positive/negative rule application conflicts on the source side when translating a source graph into a related target graph.

Definition 12. *Forward Graph Translation Rules.*

With p being constructed as listed above in Def. 11 the derived forward graph translation rule (FGT rule) is $r_{ST} := (p_{S,id}^- \xleftarrow{h_S} p_C \xrightarrow{h_T} p_T)$ with components:

- (1) $p_{S,id}^- := (R_S, R_S, \emptyset)$, i.e., the source component p_S of p without any NACs that matches and preserves the required subgraph of the source graph only
- (2) $p_C := (L_C, R_C, \emptyset)$, i.e., the unmodified correspondence component of p
- (3) $p_T := (L_T, R_T, N_T)$, i.e., the unmodified target component of p

For a detailed definition of r_{ST} that includes the morphisms between its rule components as well as for the definition of r_S/r_T and the definition of a *backward graph translation rule* (BGT rule) r_{TS} the reader is referred to [1]. The definitions presented there can be adapted easily to the scenario of integrity-preserving graph triple rewriting as done here for the case of FGT rules r_{ST} .

Definition 13 introduces the so-called *local completeness criterion* of the source domain which must be satisfied by the productions of a TGG. Essentially the definition requires that any sequence $SEQ_{i=1}^n(r_{S,i})$ can be completed to a sequence $SEQ_{i=1}^n(r_{ST,i})$ of derivation steps of a graph triple GT that exactly mimics the derivation of its source graph G_S . This criterion will be used later on in Sect. 5 to prove the completeness of the introduced algorithm that translates a given source graph G_S into a compatible target graph G_T together with a graph G_C that connects G_S and G_T appropriately.

A similar criterion of the target domain can be defined accordingly. The productions of TGG_{CDDS} satisfy both source and target criteria.

Definition 13. *Source-Local Completeness Criterion.*

A triple graph grammar TGG fulfills the source-local completeness criterion iff for all $GT_i := (G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$ and $p := (p_S \leftarrow p_C \rightarrow p_T) \in P$ with $G_S \xrightarrow{p_S @ g_S} G'_S$ exists $p^* := (p_S^* \leftarrow p_C^* \rightarrow p_T^*) \in P$, $g^* := (g_S^*, g_C^*, g_T^*)$, and $GT_{i+1}^* := (G'_S \leftarrow G'_C \rightarrow G'_T) \in \mathcal{L}(TGG)$ such that $GT_i \xrightarrow{p^* @ g^*} GT_{i+1}^*$

The local completeness criteria demand that for each local graph (G_S or G_T) of all graph triples $GT \in \mathcal{L}(TGG)$, which is rewritten by the local component of a production p , there must be at least one production p^* (p^* may equal p) which rewrites GT . Therefore, each match $g'_I(R_I \setminus L_I)$ of an input component $p_{I,id}^-$ of a translation rule r_{IO} that identifies not yet translated elements in an

input graph can be completed to a full match on the correspondence and output graphs. This is due to the fact that at least one local rule r_I (derived from a production p) exists that has created the matched yet untranslated elements in the input graph. According to the local completeness criterion a production p^* exists from which a local rule r_I^* is derived that creates the same elements as r_I . Hence, a translation rule r_{IO}^* exists that has an equivalent input component to r_{IO} which is able to translate the matched not yet translated elements. As a consequence, derived translation rules are complete, i.e., they can be used to translate any given input graph of a TGG language into a properly related graph. Furthermore, Theorem 1 guarantees the consistency of derived translation rules even if NACs are omitted. Consequently, derived rules never translate input graphs of a TGG language into output graphs such that the resulting graph triple is not an element of the just regarded TGG language.

Due to these achievements we are able to build translators that are consistent and complete with respect to their TGG. During the translation process a translator parses a given input graph in order to find a valid sequence of translation rules that mimics the derivation of the input graph. Although the TGG productions contain NACs these can be safely ignored in the parsing process in the case of integrity-preserving productions. Therefore, positive/negative rule application conflicts are prevented on the input graph. Positive/negative conflicts on the output graph will not lead to dead-ends (i.e., wrong translation alternatives which require backtracking) during parsing because the local completeness criterion guarantees that for each remaining untranslated element in the input graph, created by a local rule, a translation rule exists that is able to translate these elements. Unfortunately, we still have to solve one problem: in general we are only able to guarantee the completeness of a derived graph translator if we explore an exponential number of derivation paths (w.r.t. the size of a given input graph) due to the remaining positive/positive rule application conflicts. The following section will solve this efficiency problem for a sufficiently large class of TGGs (from a practical point of view) by introducing a new application condition for translation rules. This condition rules out any situation, where more than one rule can be used to translate a just regarded node of the input domain in a related subgraph of the output domain.

4 Dangling Edge Condition (DEC)

Translators derived from a TGG face certain difficulties concerning the selection of an appropriate sequence of translation rules in the presence of positive/positive rule application conflicts. Reconsider our triple graph grammar TGG_{CDDS} from Sect. 2. An FGT derived from TGG_{CDDS} translates class diagrams to database schemata. Figure 6 (a) depicts a graph that consists of two classes $c1$ and $c2$. The empty checkboxes⁵ denote that the elements next to them are not yet translated. The graph is valid, as it is derivable by applying productions (1) and afterwards (2) of TGG_{CDDS} to the empty graph triple. The graph is given

⁵ Fig. 6 uses an alternative representation of (not) translated nodes.

as input graph to the FGT. First, the translator applies the FGT rule derived from production (1), which translates class $c1$. Next, both rules (1) and (2) are applicable in the context of class $c2$. If the translator chooses to translate class $c2$ via rule (1) the source graph would contain two translated classes with an untranslated edge between them (cf. Fig. 6 (b)). Unfortunately, no rule exists that is able to translate the remaining untranslated edge $e1$. So, the translator produced a so-called *dangling edge* in the source graph. Consequently, the translator states at the end of the translation process that it is not able to translate the (valid) input graph completely due to this *dangling edge*.



Fig. 6. (a) Input graph given to the FGT, (b) Input graph with two translated nodes

Whenever constellations in the input graph appear, where two or more rules are applicable that translate overlapping sets of input graph elements, translation algorithms are demanding for help to select the appropriate rule. We propose an extension that is inspired by building parsers for compilers and related techniques for parsing words that are passed to the compiler. Typically, top-down and bottom-up parsers decide on more information than just the recent input: they take a *look-ahead* into account. In the following subsection we introduce a so-called *dangling edge condition* (DEC) that prevents the application of a rule if the rule would produce a dangling edge. TGG translators produce dangling edges if an edge is still untranslated at the end of the translation process. So, translators must ensure that before applying a rule another translation rule exists that is able to translate this currently “dangling” edge later on. This DEC is inspired by an analogous condition in DPO approaches, which explicitly prohibits deleting a node without deleting all incident context edges as part of the same rule application step. This way, our DEC eliminates positive/positive rule application conflicts. We restrict our focus to forward translators in the sequel, but all concepts and ideas can be transferred to backward translators as well.

The core idea of the DEC is that several productions may be applicable such that their matches overlap in some node. If the production with the smaller match is applied, incident edges cannot be translated later on. The DEC resolves conflicts where context-sensitive productions create one primary node that is connected via new edges to at least one context node. It does not offer a solution for those cases where the created nodes are not connected. In the following, we regard TGG productions that create only one primary node on each side and do not contain additional secondary elements⁶. Primary nodes of context-sensitive productions must be connected to at least one context node. The graphs

⁶ Allowing additional secondary elements would require in depth discussions and special handling in Algorithm 1 which we had to omit due to lack of space.

that result by applying such productions are either graph structures that are not connected to other structures (in case of applying initial context-free productions like production (1) of TGG_{CDDS}) or connected graph structures (in case of applying context-sensitive productions (2) to (5) of TGG_{CDDS}).

4.1 Formal introduction to LNCC and DEC

As shown at the beginning of Sect. 4, application of certain translation rules may lead to invalid graph triples since some edges in the graph of the input domain remain untranslated. Based on this observation we define for the source graph of a TGG the so-called *Legal Node Creation Context* relation with a look-ahead of one $LNCC_S(1)$ ⁷ that will be used to control the selection and application of FGT rules. A relation $LNCC_T(1)$ used by BGTs is constructed similarly. TGG productions can be broken down to certain fragments, where at most two nodes make up a part of the production. Elements of $LNCC_S(1)$ are *4-tuples* that represent certain kinds of source graph production fragments. The first and third component of a tuple represent the type of the node that is the source and target of an edge e created by a production, respectively. The type of this edge e is used as second component. The fourth component denotes whether the source node, target node, or both nodes are used as context in the production fragment. Tuples of $LNCC_S(1)$ are derived from a given TGG as follows:

Definition 14. *Legal Node Creation Context with a look-ahead of 1.*

$LNCC_S(1) \subseteq V_{TG_S} \times E_{TG_S} \times V_{TG_S} \times \{s, t, st\}$ is the smallest legal node creation context relation for the source graph of a given TGG such that

$(vt_s, et, vt_t, c) \in LNCC_S(1)$ iff

(1) \exists TGG production $((L_S, R_S, \mathcal{N}_S) \xleftarrow{h_S} p_C \xrightarrow{h_T} p_T)$ that creates edge $e \in R_S \setminus L_S$ with at least one incident already existing context node $s(e)$ or $t(e) \in L_S$

(2) $vt_s = \text{type}(s(e))$, (3) $et = \text{type}(e)$, (4) $vt_t = \text{type}(t(e))$

(5) $c \in \{s, t, st\}$, with the following semantics:

(5.1) $s: s(e) \in L_S, t(e) \in R_S \setminus L_S$

(5.2) $t: t(e) \in L_S, s(e) \in R_S \setminus L_S$ (5.3) $st: s(e), t(e) \in L_S$

Figures 7 (a), (b), and (c) identify all possible node and edge constellations that contribute tuples to $LNCC_S(1)$. In addition, Figs. 7 (d), (e), and (f) depict those production fragments that do not contribute any tuples to $LNCC_S(1)$.

The motivation behind the definition of $LNCC_S(1)$ is to block a translation of a node of the source graph that has incident edges that are not translated in the same step and that cannot be translated later on (i.e., to avoid dangling edges). This situation occurs if a TGG contains overlapping productions (e.g., productions $p^{(1)}$ and $p^{(2)}$ of TGG_{CDDS}). These productions are applicable in the same context and create a node of the same type (both $p^{(1)}$ and $p^{(2)}$ create nodes of type ‘‘Class’’) but at least one production creates an edge that relates the new

⁷ We plan to introduce $LNCC_S(n)$ with a look-ahead of $n > 1$ that also takes indirectly referenced nodes into account in future work.

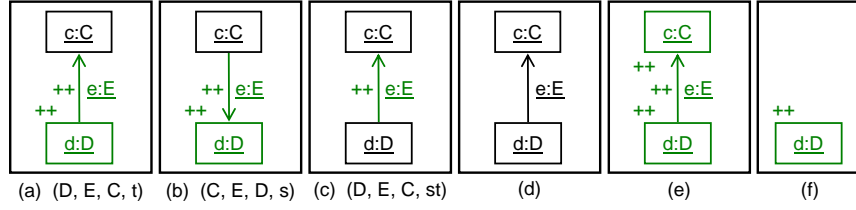


Fig. 7. TGG production fragments relevant and irrelevant for $LNCC_S(1)$.

node to an already existing node ($p^{(2)}$ creates an edge from the new subclass to its superclass). Therefore, a translator that applies one of the rules derived from these productions would destroy the match of the other rule and potentially leave an untranslatable edge. In order to identify such dangling edge situations, TGG production fragments must be inspected which create edges where the source or the target of the edge already exists, i.e., is used as context (cf. Figs. 7 (a), (b), and (c)). Translation rules derived from TGG productions containing these fragments have the potential to translate edges of the input graph using one or two already translated incident nodes as context. As patterns (d) to (f) do not translate such edges they can be neglected. Pattern (a) depicts a production fragment in which node c is the already existing context for the new node d and d is the source of the new edge e ($s(e) = d$). In production fragment (b) the direction of the edge is changed: $s(e) = c$. Pattern (c) depicts a situation where a new edge between nodes c and d is created, i.e., both nodes are used as context.

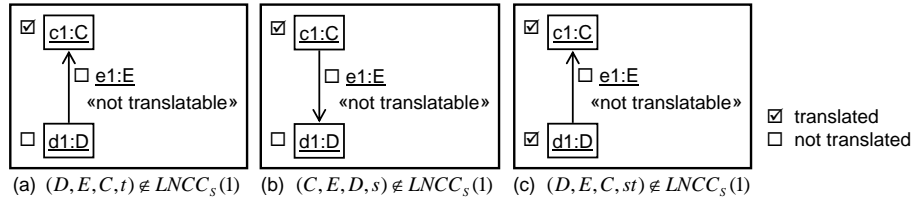


Fig. 8. Patterns in input graph that violate $DEC(1)$.

Whenever we encounter a not translated edge with an already translated incident node, we will use the relation $LNCC_S(1)$ to check whether an FGT rule exists that can be used later on to translate the regarded edge. If $LNCC_S(1)$ does not contain an appropriate tuple then the just regarded edge cannot be translated. On the other hand, the existence of an appropriate tuple does not guarantee that the edge is translatable. This is due to the fact that FGT rules r_{ST} containing a tuple are only applicable if a match of a rule's complete left-hand side ($R_S \leftarrow L_C \rightarrow L_T$) is found in the host graph triple and no NAC in the target domain blocks. In general we have to restrict the application of translation rules such that the situations depicted in Fig. 8 are avoided:

- (a) Node $d1$ is not translated yet but $c1$ (the target of $e1$) is and there exists no rule with production fragment (D, E, C, t) that may translate $e1$ later on.
 - (b) Node $d1$ is not translated yet but $c1$ (the source of $e1$) is and there exists no rule with production fragment (C, E, D, s) that may translate $e1$ later on.
 - (c) Nodes $d1$ (source) and $c1$ (target) are both translated and there exists no rule with production fragment (D, E, C, st) that may translate $e1$ later on.
- Therefore, the application of a translation rule must satisfy certain application conditions given in Def. 15 including the *Dangling Edge Condition* ($DEC(1)$).

Definition 15. *Rule application conditions for FGTs with a look-ahead of 1.*
 Let TX be the set of already translated elements of the source graph G_S , $e \in E_S$, and p be a TGG production $((L_S, R_S, \mathcal{N}_S) \xrightarrow{h_S} p_C \xrightarrow{h_T} p_T)$. Thus, for each match g'_S of translation rule r_{ST} in G_S the rule application conditions (1) to (3) must hold including the dangling edge condition $DEC(1)$ that consists of the subconditions $DEC_1(1)$, $DEC_2(1)$, and $DEC_3(1)$ in order to apply r_{ST} to $(G_S \leftarrow \dots \rightarrow \dots)$:

- (1) $g'_S(L_S) \subseteq TX$ (context elements are already translated)
 - (2) $\forall x \in g'_S(R_S \setminus L_S) : x \notin TX$ (no element x shall be translated twice)
 - (3) $TX' := TX \cup g'_S(R_S \setminus L_S)$ (TX is extended with translated elements)
- ($DEC_1(1)$) $\forall e \notin TX'$ where $s(e) \in TX', t(e) \notin TX'$:
 $(type(s(e)), type(e), type(t(e)), s) \in LNCC_S(1)$
- ($DEC_2(1)$) $\forall e \notin TX'$ where $t(e) \in TX', s(e) \notin TX'$:
 $(type(s(e)), type(e), type(t(e)), t) \in LNCC_S(1)$
- ($DEC_3(1)$) $\forall e \notin TX'$ where $s(e), t(e) \in TX'$:
 $(type(s(e)), type(e), type(t(e)), st) \in LNCC_S(1)$

Def. 15 thus introduces a rather straightforward way to decide if a translation rule shall be applied or not just by looking at the 1-context of a to-be-translated node. By adding this condition to the translation algorithm defined in [5] (cf. Algorithm 1 in Sect. 5), we are able to reduce the number of situations significantly, where we were forced to choose one of the applicable rules nondeterministically and run into dead-ends due to the wrong choice. In general, Def. 15 is not able to resolve all positive/positive conflicts, i.e., there may be multiple rules that are able to translate a node using different matches, i.e., matches containing different to-be-translated elements. Therefore, Algorithm 1 will abort in this case. Alternatively, the user could be asked which of these elements should be translated or rule priorities [8] can be used to reduce the number of different matches if more than one rule is applicable by filtering matches of rules with low priority.

Though, the algorithm permits multiple *locally-applicable rules*, i.e., rules that translate the same elements. A locally-applicable rule is either applicable also on the whole graph triple or its application is prevented, e.g., due to NACs in the output component. The set of productions of TGG_{CDDS} contains multiple locally-applicable rules. FGT rules (3) and (4) are both applicable in the context of the first attribute of a class. Likewise, BGT rules (4) and (5) are both applicable in the context of the non-first column of a table. These rules are *disjoint applicable*, i.e., only one of the locally-applicable rules is applicable on the whole graph triple (cf. forward translation example in Sect. 5). In general, multiple locally-applicable rules need not to be disjoint applicable because

they translate the same elements. Executing one of the locally-applicable rules nondeterministically does not lead into dead-ends due to the local completeness criterion and the same reason why positive/negative conflicts on the target side do not lead into dead-ends (cf. Def. 13 and subsequent discussion).

4.2 Dangling Edge Condition by Example

Now, we show by example that checking for dangling edges helps deciding which rule should be applied by translators derived from a TGG if multiple rules are applicable at overlapping matches. Therefore, we consider again the FGT derived from TGG_{CDDS} and the input graph depicted in Fig. 6 (a) already discussed at the beginning of Sect. 4. As we have already shown, both translation rules (1) and (2) are applicable after applying rule (1) to this input graph. Based on the classification scheme of Fig. 7 and Def. 14 we construct the set of tuples from the TGG productions of TGG_{CDDS} which results in $LNCC_S(1) =$

$$\{(Class, Inherits, Class, t), (Class, Contains, Attr, s), (Attr, Precedes, Attr, s)\}.$$

Next, we pretend to apply rule (1) in the context of class $c2$. Then, we calculate the set $inc(c2) = \{e1\}$ which contains incident edges of $c2$ that are not yet translated. We must check whether all edges in $inc(c2)$ are translatable by further rewriting steps, i.e., whether $DEC(1)$ is satisfied. As both source and target of $e1$ are already translated, a tuple must exist in $LNCC_S(1)$ that satisfies subcondition $DEC_3(1)$. Therefore, the tuple $(Class, Inherits, Class, st)$ must be in $LNCC_S(1)$ which is not the case. As a consequence, we do not apply FGT rule (1), because this would result in a dangling edge (cf. Fig. 6 (b)) and proceed pretending to apply rule (2). In this case $inc(c2) = \emptyset$. So, the rule application conditions given in Def. 15 are satisfied, i.e., there are no dangling edges. Concluding, we were able to translate the input graph completely due to the fact that the DEC prohibited selecting a wrong translation rule match.

5 Extended Translation Algorithm with DEC

In this section we extend the algorithm presented in [5] so it handles NACs as presented in Sect. 3 and checks the dangling edge condition (cf. Sect. 4). We discuss the extended algorithm (cf. Algorithm 1) by translating the graph triple shown in Fig. 10 (a) with the FGT derived from TGG_{CDDS} . Each translator implements procedure $evolve : GT_{in} \rightsquigarrow^* GT_{out}$ which simulates the simultaneous evolution of a given graph triple GT_{in} . The input graph triple GT_{in} is either $(G_S \leftarrow G_\emptyset \rightarrow G_\emptyset)$ in case of an FGT or $(G_\emptyset \leftarrow G_\emptyset \rightarrow G_T)$ in case of a BGT⁸. The input graph G_{input} is either G_S or G_T depending on the type of translator (FGT/BGT), whereas the output graph G_{output} is either G_T or G_S . Procedure $evolve$ assumes that the underlying TGG is integrity-preserving and that the input graph GT_{in} was produced by a sequence of input-local rules r_I ,

⁸ In the general case, when incremental updates are performed with a translator, the correspondence graph and the graph of the opposite domain need not to be empty.

i.e., $G_{input} \in \mathcal{L}(GG_I)$. *Evolve* is able to cope with situations where the underlying TGG or the input is invalid. It throws errors if it detects an invalid TGG specification and exceptions in case of invalid inputs. A valid translation produces an output graph triple $GT_{out} = (G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$. Therefore, *evolve* calls subroutine $translate(GraphTriple)$ which in turn calls procedure $translate(Node)$ for all nodes in the input graph G_{input} . Resulting graph triples of invalid translations are undefined. Algorithm 1 uses so-called *core rules* (cf. [5]) to determine matches of translation rules in the input graph. A core rule is closely related to the input component $p_{I,id}^-$ (either $p_{S,id}^-$ or $p_{T,id}^-$) of a translation rule (cf. Def. 12). Fig. 9 shows the core rules derived from TGG_{CDDS} which are used by the forward translator. A core rule looks up the context elements of a given primary element in G_{input} , which may or may not be translated already but must be translated so the primary element is translatable (cf. Def. 15 (1)). Core rules only contain elements of the input graph. NACs are not contained in a core rule. The primary element and additional incident edges must not be translated yet. This is indicated by the empty checkboxes next to these elements.

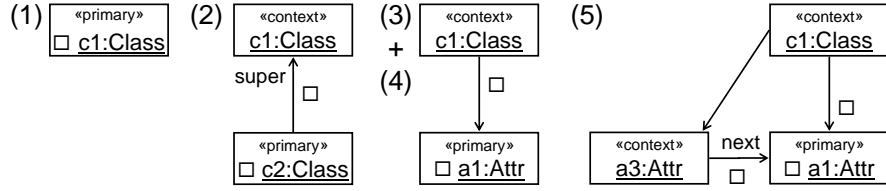


Fig. 9. Core rules of the FGT derived from TGG_{CDDS} .

The algorithm starts translating G_{input} (cf. Fig. 10 (a)) with node $c1$. It may start with any other node as well because it recursively translates the context of the current node before it translates the node itself. First, the algorithm determines *appropriate rules* from the set of *candidate rules*. A rule is a candidate if its primary node in the input domain is type compatible with the to-be-translated primary node. An appropriate rule has at least one *core match* which contains the primary node. A core match satisfies Def. 15 (2), i.e., all to-be-translated elements are not translated yet. If multiple matches of one rule in the input graph exist⁹, the algorithm checks for every match if the rule is appropriate. In order to be appropriate, every context node required by the primary node is recursively translated. But, the context is only translated if the dangling edge condition would be satisfied afterwards (cf. Def. 15 DEC(1)). Next, the algorithm determines *applicable rules* from the set of appropriate rules. The application condition Def. 15 DEC(1) has to be reassured because it might have been invalidated due to potential competing recursive context translations. In addition, the core match of an applicable rule must be completed in the rule's

⁹ In TGG_{CDDS} at most one core match exists for any rule in a valid input graph.

left-hand side (i.e., input, link, and output domain) and the NACs in the output domain must not block. If no applicable rule is determined then either a match exists in the input domain but it may not be completed or the to-be-translated node is not even locally translatable. In the first case the set of TGG productions violates the local completeness criterion (cf. Def. 13), in the latter case the input graph is invalid. If multiple rules are applicable at some completed match, the algorithm ensures that their to-be-translated elements in the core match are identical. Otherwise it aborts with an error as this might lead into dead-ends (cf. Sect. 4.1). It is up to the developer of a set of TGG productions to guarantee that this will never happen in practice. Finally, the algorithm translates the primary node. It selects one entry from the set of applicable rules, applies the rule at its match, and extends the set of translated elements (cf. Def. 15 (3)).

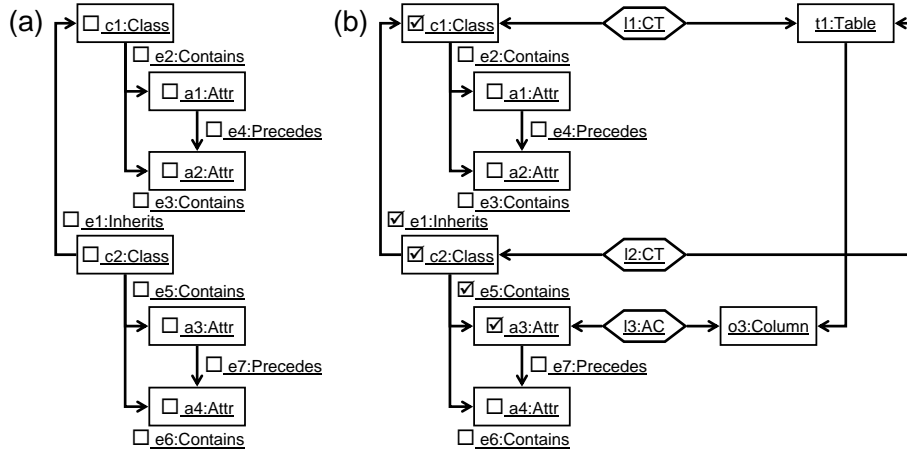


Fig. 10. Snapshots of the translation process during forward translation.

FGT rule (1) (i.e., $r_{ST}^{(1)}$) is the only candidate that has the capability to translate $c1$ because no match exists for the other candidate rule (2). The algorithm checks whether application of rule (1) satisfies DEC(1). Therefore, it determines the elements in G_{input} created by the corresponding local rule $r_S^{(1)}$ (i.e., $\{c1\}$) and joins these elements, the required context elements (i.e., \emptyset), and the set of currently translated elements (i.e., \emptyset) which results in set $TX' := \{c1\}$. Then, it checks whether condition DEC(1) is satisfied for all not translated incident edges of node $c1$, i.e., $inc(c1) := \{e1, e2, e3\}$. According to Def. 15, the required tuples for $inc(c1)$ are $(Class, Inherits, Class, t)$ and $(Class, Contains, Attr, s)$. So, DEC(1) is satisfied because $LNCC_S(1)$ contains these tuples (cf. Sect. 4.2). Since $c1$ does not have any context, no additional elements need to be translated and rule (1) is marked *appropriate*. Moreover, its core match can be completed to a full match. Therefore, $r_{ST}^{(1)}$ is an applicable rule. As it is the only applicable

```

1  procedure GraphTriple evolve(inputGraphTriple: GraphTriple) { //  $GT_{in}$ 
2    global inputGraph: Graph = Translator.getInputGraph(inputGraphTriple); //  $G_{input}$ 
3    global translatedElements: ElementSet = inputGraph.getTranslatedElements(); //  $TX$ 
4    global justRegardedElements: ElementSet =  $\emptyset$ ;
5    inputValid: boolean = inputGraph.verifyConstraints(); // Def.8(1)/(3) satisfied?
6    outputGraphTriple: GraphTriple = translate(inputGraphTriple); // produce  $GT_{out}$ 
7    outputValid: boolean = Translator.getOutputGraph(outputGraphTriple).verifyConstraints();
8    translated: boolean = inputGraph.isCompletelyTranslated();
9    if (inputValid && outputValid && translated)
10     return outputGraphTriple; // successfully produced  $GT_{out}$ 
11   else if (inputValid && !outputValid) // Def.10(3) violated!
12     throw TGGContainsIntegrityDestroyingProductionsError(outputGraphTriple, translated);
13   else throw InputGraphNotPartOfDerivableGraphTripleException( // user-error: ...
14     outputGraphTriple, inputValid, outputValid, translated); // ... $G_{input} \notin \mathcal{L}(GG_I)$ 
15 }
16 procedure GraphTriple translate(graphTriple: GraphTriple) {
17   forall (node  $\in$  inputGraph) { translate(node); }
18   return graphTriple;
19 }
20 procedure translate(n: Node) {
21   if (n  $\in$  translatedElements) return;
22   else if (n  $\in$  justRegardedElements)
23     throw CycleInRecursiveContextTranslationError(n, justRegardedElements);
24   else { justRegardedElements.add(n);
25     nodeLocallyTranslatable: boolean = false;
26     appropriateRules, applicableRules: PairSet<Rule, Match> =  $\emptyset$ ;
27     candidateRules: RuleSet = select rules where r.primaryInputNode.type equals n.type;
28     forall (rule  $\in$  candidateRules) { // collect appropriate rules and core matches
29       compute core matches of rule in inputGraph with n as primary node;
30       forall (cm  $\in$  core matches) { // Def.15(2):  $g'_I(R_I \setminus L_I) \cap TX = \emptyset$  satisfied!
31         if (not isDECSatisfied(n, join(cm.toBe, cm.context))) //  $g'_I(R_I \setminus L_I) \cup g'_I(L_I)$ 
32           { continue; } // do not translate context if Def.15(DEC(1)) would be violated
33         forall (contextNode  $\in$  context elements of core match)
34           { translate(contextNode); } // recursively translate required context
35         if (all context elements of core match are translated)
36           { appropriateRules.add(rule, cm); } // Def.15(1):  $g'_I(L_I) \subseteq TX$  satisfied!
37       } // end of appropriate rule at core match with n as primary node calculation
38       forall ((rule, cm)  $\in$  appropriateRules) { // collect rules applicable at full match
39         // reassure Def.15(DEC(1)): may be violated due to competing context translation
40         if (not isDECSatisfied(n, cm.toBe)) { continue; }
41         nodeLocallyTranslatable = true; // now n must be translatable due to Def.13
42         if (cm can be completed in other domains and NACs in output domain don't block)
43           { applicableRules.add(rule, full match); }
44       } // end of applicable rule at full match with n as primary node calculation
45       if (applicableRules.isEmpty()) { // node is not translatable
46         if (nodeLocallyTranslatable) // match could not be completed in other domain(s)
47           throw LocalCompletenessCriterionError(n, appropriateRules); // Def.13 violated!
48         else
49           throw InputGraphNotPartOfDerivableGraphTripleException(n); //  $G_{input} \notin \mathcal{L}(GG_I)$ 
50       }
51       if (not applicableRules.matches->forall(m1, m2 | m1 <> m2 implies
52         m1.cm.toBe = m2.cm.toBe)) throw CompetingCoreMatchesError(n, applicableRules);
53       select one rule/match pair from applicableRules;
54       apply rule at match; // evolve  $GT \rightsquigarrow GT'$  with  $r_{IO}@g_{IO}$ 
55       translatedElements.add(elements of inputGraph translated by rule); //  $g'_I(R_I \setminus L_I)$ 
56       justRegardedElements.remove(n);
57     } }
58 procedure boolean isDECSatisfied(node: Node, toBeTranslated: ElementSet) {
59   translatedElements' = translatedElements  $\cup$  toBeTranslated; //  $TX'$  (cf. Def.15(3))
60   select all incident edges e of node where (e  $\notin$  translatedElements')
61     and (s(e) or t(e)  $\in$  translatedElements')
62   forall (e  $\in$  selected incident edges)
63     { if (not (DEC(1) satisfied for e)) { return false; } } // ensure Def.15(DEC(1))
64   return true; // all edges translatable
65 }

```

Algorithm 1. Algorithm that handles NACs and checks for dangling edges.

rule it is applied at the complete match which translates node $c1$ and creates a corresponding table in the target domain. The algorithm proceeds translating by selecting node $a3$ from the set of remaining nodes $\{c2, a1, a2, a3, a4\}$. FGT rules (3), (4), and (5) are candidate rules for translating $a3$. But a core match, which requires $c2$ as context, exists only for rules (3) and (4). First, the core match of rule $r_{ST}^{(3)}$ is examined. DEC(1) is satisfied for $inc(a3) = \{e7\}$ as $(Attr, Precedes, Attr, s) \in LNCC_S(1)$. Now, the context $c2$ is translated by a recursive call to $translate(Node)$. The candidate rules for translating $c2$ are $r_{ST}^{(1)}$ and $r_{ST}^{(2)}$. The algorithm randomly selects $r_{ST}^{(2)}$ to be checked first. DEC(1) is not violated (cf. Sect. 4.2) and the context of $c2$ (i.e., $c1$) is already translated. The algorithm does not translate $c1$ again because it notices that $c1$ is already translated. After marking rule (2) as appropriate, candidate rule $r_{ST}^{(1)}$ is checked. It would violate DEC(1) and therefore it is not added to the set of applicable rules. As the match of $r_{ST}^{(2)}$ can be completed it is the only applicable rule and used to translate $c2$. The algorithm returns from the recursion and resumes in the context of $a3$. It marks $r_{ST}^{(3)}$ as appropriate and proceeds with $r_{ST}^{(4)}$ which is also appropriate. Though, the only applicable rule is $r_{ST}^{(3)}$ because, contrary to $r_{ST}^{(4)}$, its match can be completed. Consequently, both locally-applicable rules $r_{ST}^{(3)}$ and $r_{ST}^{(4)}$ are *disjoint applicable* in this case (cf. discussion in Sect. 4.1). Therefore, $r_{ST}^{(3)}$ is used to translate $a3$. Right now, the remaining nodes are $\{a1, a2, a4\}$ and the current graph triple looks like depicted in Fig. 10 (b). The following translation steps are rather similar to the preceding steps so we will abbreviate the explanation. Next, attribute $a1$ is translated. Its context $\{c1\}$ is already translated. Rule candidates with a core match are $r_{ST}^{(3)}$ and $r_{ST}^{(4)}$, which both satisfy DEC(1), but $r_{ST}^{(3)}$ is blocked due to the NAC in the output domain. FGT rule (4) is applicable as a match of the LHS is found and the NAC does not block. Finally, attributes $a2$ and $a4$ are translated in this order. Their context is already translated and the rule candidates are $r_{ST}^{(3)}$, $r_{ST}^{(4)}$, and $r_{ST}^{(5)}$. FGT rules (3) and (4) are neglected as their application both would violate DEC(1) for $inc(a2) = \{e4\}$ and $inc(a4) = \{e7\}$ because $(Attr, Precedes, Attr, st) \notin LNCC_S(1)$. Hence, application of $r_{ST}^{(5)}$ would not violate DEC(1). Therefore, both attributes $a2$ and $a4$ are translated by $r_{ST}^{(5)}$.

So, the algorithm has successfully translated all nodes and edges of the input graph to a corresponding output graph. The sequence of applied FGT rules $SEQ(r_{ST}) = (r_{ST}^{(1)}@{\emptyset}, r_{ST}^{(2)}@c1, r_{ST}^{(3)}@c2, r_{ST}^{(4)}@c1, r_{ST}^{(5)}@c1, r_{ST}^{(5)}@c2)$ constructed in this example translates the primary nodes in this order: $(c1, c2, a3, a1, a2, a4)$. In conjunction with the also constructed correspondence graph a graph triple GT_{out} was produced which is equivalent to graph triple GT_6 (cf. Fig. 3 (a)) that was derived from TGG production sequence SEQ_6 (cf. Sect. 2.3) starting with the empty graph triple. Therefore, the FGT sequence exactly mimics SEQ_6 .

The next theorems state that translators based on Algorithm 1 are efficient as well as consistent and complete with respect to their TGG if the algorithm never aborts for any given valid input graph. If the algorithm aborts then either

$G_{input} \notin \mathcal{L}(GG_I)$ or the TGG specification is erroneous, i.e., does not satisfy the conditions stated throughout this contribution.

Theorem 2. *Efficiency of Graph Translation.*

Algorithm 1 has worst case runtime complexity of $O(n^k)$ with n being the number of nodes of G_{input} and k being a constant that depends on the regarded TGG.

Proof.

Sketch:

- (1) The algorithm just loops through the set of all n nodes of the input graph; the implicit reordering of the translation of input graph elements in the loop for not yet translated context elements of a just regarded graph element does not affect its runtime complexity.
- (2) The book keeping overhead of the algorithm is neglectible and the execution time for basic graph operations like traversing an edge or creating a new graph element is bounded by a constant (otherwise we should add a logarithmic or linear term depending on the implementation of the underlying graph data structure).
- (3) The worst case execution time of all needed rules applied to a given (primary) input graph node is $(n+n')^{k-1}$, where n' is the number of nodes of the output graph, and k is the maximum number of elements of any applicable rule. In the worst case the match of the primary node is extended by testing all possible $(n+n')^{k-1}$ permutations of source/target graph elements.
- (4) Furthermore, $n' \leq c * n$ for a given constant c that is the maximum number of new nodes of the output component of any TGG production. \square

In the case of TGG_{CDDS} , the complexity of a forward translation is $O(n^2)$ for the following reasons: The worst case execution time of its rules (cf. Fig. 4) is $O(n') \leq O(n)$ due to the fact that rules (1), (2), and (3) have a constant execution time, whereas rules (4) and (5) have to determine the last column node of a table node. Assuming that all nodes of the output graph are columns of the just regarded table $n' \leq n$ nodes have to be inspected in the worst case.

Theorem 3. *Consistency of Graph Translation.*

Let $G_I \in \mathcal{L}(TG_I, \mathcal{C}_I)$ be an input graph (either G_S or G_T) and G_O be an output graph (either G_T or G_S). If

$$FGT(G_S \leftarrow G_\emptyset \rightarrow G_\emptyset) = (G_S \leftarrow G_C \rightarrow G_T) \text{ and}$$

$$BGT(G_\emptyset \leftarrow G_\emptyset \rightarrow G_T) = (G_S \leftarrow G_C \rightarrow G_T), \text{ respectively,}$$

is a not aborting complete translation of G_I with Algorithm 1 then:

- (1) $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$ and (2) $G_O \in \mathcal{L}(TG_O, \mathcal{C}_O)$.

Proof.

Sketch:

- (1) $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$ is a direct consequence of Theorem 1 and the fact that $G_I \in \mathcal{L}(TG_I, \mathcal{C}_I)$. As a consequence the simulated application of TGG productions without NACs in the input domain does not have any effect concerning the applicability of translation rules.

- (2) The behavior of translation rules on the output side is identical with the behavior of the related TGG production: i.e., a rule finds a match on the output side iff the related TGG production has the same match.
 $G_O \in \mathcal{L}(TGO, \mathcal{C}_O)$ is then a direct consequence of (1). \square

Theorem 4. *Completeness of Graph Translation.*

Let $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$ and let us assume that the execution of Algorithm 1 does not abort with any error. Then, we can guarantee that graphs G_C^* , G_T^* and G_S^* , G_C^* , respectively, exist such that:

$$FGT(G_S \leftarrow G_\emptyset \rightarrow G_\emptyset) = (G_S \leftarrow G_C^* \rightarrow G_T^*) \in \mathcal{L}(TGG) \text{ and}$$

$$BGT(G_\emptyset \leftarrow G_\emptyset \rightarrow G_T) = (G_S^* \leftarrow G_C^* \rightarrow G_T) \in \mathcal{L}(TGG), \text{ respectively;}$$

i.e., the algorithm terminates without throwing any exception.

Proof.

(by induction) Sketch:

Let $GT_{out} \in \mathcal{L}(TGG)$ be a graph triple that has been derived using a sequence of derivation steps $SEQ_{i=1}^n(p_i) = ((p@g)_1, \dots, (p@g)_n)$ of length n and let $SEQ_{i=1}^n(r_{I,i}) = ((r_I@g_I)_1, \dots, (r_I@g_I)_n)$ be the projection of the regarded sequence of graph triple derivation steps on its input graph. Furthermore, let $SEQ_{i=0}^m(r_{IO,i})$ with $0 \leq m \leq n$ be the sequence of the first m translation rule applications $((r_{IO}@g_{IO})_1, \dots, (r_{IO}@g_{IO})_m)$ generated by the algorithm that exactly mimics the derivation of GT_{out} .

Case 1, $m = 0$: A translation rule sequence of length 0 trivially mimics the derivation of the empty graph triple GT_\emptyset .

Case 2, $0 < m < n$: We have to show that the algorithm extends the given sequence of rule applications of length m to a sequence of length $m+1$ such that it simulates either the original sequence of TGG productions $SEQ(p)$ or a slightly modified sequence $SEQ(p^*)$ that still generates the same input graph. Let $(v_I)_i$ be the primary node of the input graph of each rule application $(r_{IO}@g_{IO})_i$ and TGG production application $(p@g)_i$ with $1 \leq i \leq m$. Let v be the next to-be-translated primary node which is selected by the algorithm. Furthermore, we assume that the algorithm has already translated successfully the context nodes of all rules that might be able to translate node v .

Case 2.1, $v = (v_I)_{m+1}$: Due to the fact that the algorithm does not throw a `CompetingCoreMatchesError` we can safely assume that there exists at most one set of translation rules with the same to-be-translated elements in their core match including node v . Furthermore, we know that there exists at least one rule with $p_{I,id}^-$ (that is the input component of the translation rule derived from p_{m+1}) that matches node $v = (v_I)_{m+1}$. The local completeness criterion (cf. Def. 13) guarantees that the algorithm finds a TGG production application $p^*@g^*$ that corresponds to one of the translation rules r_{IO}^* that is able to handle the translation of the selected node v . Applying Def. 13 multiple times we can generate a new sequence of TGG production applications $SEQ_{i=1}^n(p_i^*)$ such that:

$$1 \leq i \leq m: (p^*@g^*)_i = (p@g)_i$$

$$i = m + 1: (p^*@g^*)_i = p^*@g^*$$

$$m + 1 < i \leq n: (p^*@g^*)_i \text{ is a new production application that mimics } (p@g)_i$$

As a consequence the algorithm is able to create a sequence of translation steps $SEQ(r_{IO}^*)$ of length $m + 1$ that has the same properties as the given sequence of translation steps $SEQ(r_{IO})$ of length m w.r.t. the new sequence $SEQ_{i=1}^n(p_i^*)$ that replaces $SEQ_{i=1}^n(p_i)$.

Case 2.2, $v \neq (v_I)_{m+1}$: Due to the fact that the selected node v is not yet translated and that $(v_I)_1, \dots, (v_I)_n$ is the complete set of all primary nodes of the given input graph (generated by the given sequence of TGG production applications) there exists an index $k > m + 1$ such that $v = (v_I)_k$. Let $(p_{I,id}^-)_k$ be the input component of the translation rule derived from $(p@g)_k$. We know that all context nodes potentially required by $(p_{I,id}^-)_k$ are already translated. Again relying on the fact that the algorithm does not throw any error and on Def. 13 we know that a rule r_{IO}^* exists, derived from a production p^* , which is able to translate the given primary node v . Using the same line of arguments as in case 2.1 we can construct a new sequence of TGG productions p^* of length n with the same properties as listed above. As a consequence the algorithm is again able to create a sequence of translation steps $SEQ(r_{IO}^*)$ of length $m + 1$ that has the same properties as the given sequence of translation steps of length m .

Case 3, $m = n$: The translation rule sequence mimics the complete derivation of the input graph, i.e., generates a valid translation into a graph triple GT_{out}^* that has the same input graph as GT_{out} but may have different correspondence and output graphs then GT_{out} . \square

The consequence of the proof sketches is as follows. If we are able to show for a given TGG that derived translators never abort with an error then:

- (1) The presented algorithm can be executed efficiently (polynomial complexity) as long as the matches of all translation rules can be computed efficiently.
- (2) Forward and backward translation results are consistent, i.e., do only produce graph triples that belong to the language of the regarded TGG.
- (3) Forward and backward translations will always produce a result for a given input graph if the language of the regarded TGG contains a graph triple that has this input graph as a component.

Finally, our running example shows that in the general case the result of a graph translation is not uniquely determined up to isomorphism, i.e., sets of TGG productions needed in practice often do not satisfy any (local) confluence criteria. Therefore, it was of importance to develop an efficiently working graph translation algorithm that does not rely on (local) confluence criteria of TGG productions or translation rules, but nevertheless fulfills the initially presented expressiveness, consistency, and completeness properties, too!

6 Related Work

Based on the characterization of “useful” TGGs in Sect. 2 we proposed extensions to triple graph grammars in Sects. 3, 4, and 5. Now, we are prepared to evaluate and assess various forms of declarative bidirectional model or graph transformations that have been published in the past.

The first TGG publication [1] introduced a rather straight-forward construction of translators. It relied on the existence of graph grammar parsing algorithms with exponential worst-case space and time complexity. As a consequence a first generation of follow-up publications [13, 14] all made the assumption that the regarded graphs have a dominant tree structure and that the components of a TGG production possess one and only one primary node. Based on these assumptions an algorithm is used that simply traverses the tree skeleton of an input graph node by node and selects an arbitrary matching FGT/BGT rule for a regarded node that has a node of this type as its primary node. This algorithm defines translation functions that are neither consistent nor complete in the general case. Both properties are endangered by the fact that the selected tree traversal order does not guarantee that rules are applied in the appropriate order. It may happen that the application of a rule fails because one of its context nodes has not been processed yet or that a rule is applied despite of the fact that one of its context nodes has not been matched by another rule beforehand.

As a consequence, [8] introduces an algorithm that still relies on a tree traversal, but keeps track of the set of already processed nodes and uses a waiting queue to delay the application of rules if needed. This algorithm defines consistent translators, but has an exponential worst-case behavior concerning the number of re-applications of delayed rule instances. Another class of TGG approaches (cf. [15]) attacked the rule ordering problem in a rather different way. These approaches introduce a kind of controlled TGGs, where each rule explicitly creates a number of child rule instances that must be processed afterwards. Thus, one of the main advantages of a rule-based approach is in danger that basic rules can be added and removed independently of each other and that it is not necessary to encode a proper graph traversal algorithm explicitly.

All publications mentioned so far refrained from the usage of NACs that were introduced in [16] in the context of model transformation approaches based on graph transformation. Some of them even argued that NACs cannot be added to TGGs without destroying their fundamental properties! But, rather recently some application-oriented TGG publications simply introduced NACs without explaining how derived translation rules and their rule application strategies have to be adapted precisely. The publications even give the reader the impression that NACs can be evaluated faithfully on a given input graph without regarding the derivation history of this graph with respect to its related TGG. [17], e.g., explicitly makes the proposal to handle complex graph constraints in this way, whereas [18] and [4] ignore the problems associated with the usage of NACs completely. Despite these TGG approaches that already introduced NACs to TGGs without a guarantee for consistency and completeness, we proposed translators in [5] that guarantee consistency. Nevertheless, we could not guarantee completeness of these translators.

We have to reference [19] as the first publication that studied useful properties of translators including “invertibility” from a formal point of view. The authors of this paper are interested in pairs of translation relations that are inverse to each other. As a consequence they have to impose hard restrictions

on TGGs in order to be able to construct their proofs. Furthermore, [19] has a main focus on consistency, whereas efficiency, expressiveness, and completeness are out-of-scope. In addition, [19] extends the concept of triple graphs based on simple graphs to triple graphs based on typed, attributed graphs. Follow-up publications (e.g., [20] and [10]) then introduced NACs in an appropriate way and proved that translators may be derived from a TGG with NACs that are compatible with their TGG. Unfortunately, both [20] and [10] trade efficiency for completeness. That is, neither [20] nor [10] present an algorithm that is able to find an appropriate sequence of translation rules in polynomial time which is necessary to create efficiently working translators. Compared to both approaches, we showed here that we are able to derive compatible translators—from a precisely defined subset of TGGs with NACs—which are still efficient.

Other bidirectional model/graph translation approaches either suffer from similar deficiencies or circumvent the efficiency versus completeness tradeoff problem as follows: QVT Relational [3] as a representative of this sort of model transformation approaches simply applies all matches of all translation rules to a given input model in parallel and merges afterwards elements of the generated output model based on key attributes. This approach is rather error-prone and requires a deep insight of the QVT tool developer as well as its users how rules match and interact with each other. As a consequence, [21] shows that today existing QVT Relational tools may produce rather different results when processing the same input. For a more comprehensive survey of bidirectional transformation approaches the reader is referred to [22].

7 Conclusion and Future Work

In this contribution we presented a “useful” class of triple graph grammars together with translators that comply to the four design principles stated in the “*Grand Research Challenge of the Triple Graph Grammar Community*” introduced in [5]: the development of a consistent, complete, and efficient graph translation algorithm for a hopefully still sufficiently expressive class of triple graph grammars (TGGs). For this purpose we combined (a) restrictions for negative application conditions of TGG productions with (b) a dangling edge condition for graph translation rules that was inspired by “look ahead” concepts of parsing algorithms. As a consequence, graph translators derived from the thus restricted class of TGGs no longer have to take care of rule application conflicts by either using a depth-first backtracking parsing algorithm or a breadth-first computation of all possible derivations of a given input graph. Therefore, the presented new graph translation algorithm has a polynomial runtime complexity of $O(n^k)$ for a rather small k in practice that is determined by the worst-case complexity of computing matches for all needed graph translation rules.

Promising directions for future work are, e.g., the adaptation of the confluence checking algorithms of AGG [23], which are based on critical pair analysis, as well as the model checking approach for graph grammars of GROOVE [24] to the world of TGGs. Confluence checking techniques should offer the right

means for the detection and classification of potential rule application conflicts at compile time. In this way we would be able to guarantee already at compile time that a graph translator derived from a specific class of TGGs will not stop its execution with an error instead of generating an existing output graph for a given input graph. Furthermore, constraint verification techniques of GROOVE should allow to check the here introduced requirements already at compile time: (a) TGG productions never create graph triples that violate graph constraints of the related schema, (b) NACs are only used to block graph modifications that would violate a graph constraint, (c) TGG productions never repair constraint violations by rewriting an invalid graph into a valid graph, and (d) TGGs fulfill the local completeness criterion. Until then, TGG developers have to design and test their TGGs carefully such that TGG productions do not violate the presented conditions of integrity-preserving productions. Moreover, the presented algorithm has to be extended to cope with secondary elements and to perform incremental updates in order to also synchronize changes in source and target domain models. In addition, the limited class of TGGs with NACs presented in this contribution has to be enlarged but compatibility and efficiency properties of derived translators have to be ensured. We are currently evaluating the here presented class of TGGs with NACs in research cooperations with industrial partners, where TGGs are used to ensure consistency of design artifacts. Time will show whether our claim is true that the here introduced new class of TGGs is still expressive enough for the specification of a sufficiently large class of bidirectional model/graph translations that are needed in practice.

References

1. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In Mayr, Schmidt, eds.: Proc. WG'94 Workshop on Graph-Theoretic Concepts in Computer Science. (1994) 151–163
2. Kleppe, Warmer, Bast: MDA Explained. Addison-Wesley (2003)
3. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, v1.0. (April 2008) <http://www.omg.org/spec/QVT/1.0/>.
4. Taentzer, G., et al.: Model Transformation by Graph Transformation. In: Model Transformation in Practice (MTiP'05), workshop at MODELS'05. (2005)
5. Schürr, A., Klar, F.: 15 Years of Triple Graph Grammars - Research Challenges, New Contributions, Open Problems. In: 4th International Conference on Graph Transformation. Volume 5214 of LNCS., Heidelberg, Springer (2008) 411–425
6. Bruel, J.M., ed.: Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers. In Bruel, J.M., ed.: MoDELS Satellite Events. Volume 3844 of LNCS., Springer (2006)
7. Zündorf, A.: Rigorous Object Oriented Software Development. University of Paderborn (2001) Habilitation Thesis.
8. Königs, A.: Model Integration and Transformation - A Triple Graph Grammar-based QVT Implementation. PhD thesis, TU Darmstadt (2009)
9. Rekers, J., Schürr, A.: Defining and Parsing Visual Languages with Layered Graph Grammars. Journal of Visual Languages and Computing **8**(1) (1997) 27–55

10. Ehrig, H., Ermel, C., Hermann, F., Prange, U.: On-the-fly construction, correctness and completeness of model transformations based on triple graph grammars. In Schürr, A., Selic, B., eds.: *Model Driven Engineering Languages and Systems. Proceedings of MODELS 2009*. Volume 5795 of LNCS., Springer (2009) 241–255
11. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: *Fundamentals of Algebraic Graph Transformation*. EATCS Series. Springer-Verlag (2006)
12. Heckel, R., Cherchago, A.: Structural and behavioural compatibility of graphical service specifications. *J. Log. Algebr. Program.* **70**(1) (2007) 15–33
13. Lefering, M.: Software document integration using graph grammar specifications. In: *6th International Conference on Computing and Information*. Volume 1 of *Journal of Computing and Information*. (1994) 1222–1243
14. Jahnke, J., Schäfer, W., Zündorf, A.: A design environment for migrating relational to object oriented database systems. In: *12th International Conference on Software Maintenance (ICSM'96)*. (1996) 163–170
15. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. Volume 4199 of LNCS., Springer Verlag (2006) 543–557
16. Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae* **26**(3-4) (1996) 287–313
17. Kindler, E., Wagner, R.: *Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios*. Technical Report tr-ri-07-284, Department of Computer Science, University of Paderborn, Germany (2007)
18. Grunske, L., Geiger, L., Lawley, M.: A Graphical Specification of Model Transformations with Triple Graph Grammars. In Hartman, A., Kreische, D., eds.: *First European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA*. Volume 3748 of LNCS., Springer (2005) 284–298
19. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In: *Fundamental Approaches to Software Engineering (FASE)*. Volume 4422 of LNCS., Springer (2007)
20. Ehrig, H., Hermann, F., Sartorius, C.: Completeness and Correctness of Model Transformations based on Triple Graph Grammars with Negative Application Conditions. *ECEASST* **18** (2009)
21. Greenyer, J., Kindler, E.: Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars. *Software and Systems Modeling* (2009)
22. Czarnecki, K., Foster, J., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.: Bidirectional Transformations: A Cross-Discipline Perspective. In: *Theory and Practice of Model Transformations: Second International Conference, ICMT2009*. Volume 32 of LNCS., Springer Verlag (2009) 260–283
23. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. In: *Proceedings of the First International Conference on Graph Transformation*. Volume 2505 of LNCS., Springer (2002) 161–176
24. Rensink, A.: Explicit State Model Checking for Graph Grammars. In Nicola, R.D., Degano, P., Meseguer, J., eds.: *Concurrency, Graphs and Models*. Volume 5065 of LNCS. Springer Verlag, Berlin (2008) 114–132