Model Transformation with Triple Graph Grammars and Non-terminal Symbols

vorgelegt von
William Bombardelli da Silva

Technische Universität Berlin
Fakultät Elektrotechnik und Informatik

zur Erlangung des akademischen Grades
Master of Science
M.Sc.

Betreuer: Dipl.-Inform. Max Bureck
Berichterin: Prof. Dr.-Ing. Ina Schieferdecker
Berichter: Prof. Dr. Dr. h.c. Sahin Albayrak

Tag der wissenschaftlichen Aussprache: 5. Dezember 2018

Berlin 2018
D 83

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Die selbständige und eigenständige Anfertigung versichert an Eiden statt:

Berlin, den

____

Unterschrift

Abstract

Zusammenfassung

Contents

List of Figures

List of Tables

List of Acronyms

ATL	Atlas Transformation Language
BNCE	Boundary Neighborhood-controlled Embedding
CYK	Cocke-Young-Kassami
EMF	Eclipse Modelling Framework
MDSD	Model-driven Software Development
NCE	Neighborhood-controlled Embedding
NLC	Node label controlled
NP	Neighborhood Preserving
NTC	Non-terminal Consistent
PAC	Positive Application Conditions
QVT-O	Query/View/Transformation Operational
QVT-R	Query/View/Transformation Relational
TGG	Triple Graph Grammar

1. Introduction

One of the biggest challenges for the construction of software is to construct high-quality software artifacts. The success of a software project is very often decided by the quality aspects of the produced outcome. These aspects include, among others, correctness, reliability, security, usability, and performance. Despite its importance, quality is frequently not achieved in software products. To overcome that, software engineering techniques of several kinds have been created. These techniques range from people and process management, test and verification methods up to construction standards, like coding guidelines, model checking and the model-driven software development (MDSD) approach.

The MDSD approach for software construction places the use of software models at the center of the building process. The term "model" is understood as an artifact that represents some part of a software system, that is, it encodes, possibly more abstractly, some aspects of this systems. One example of a model for an object-oriented software system is a class diagram, which encodes the classes that compose the system in a more abstract manner than the actual system does. Notably, abstraction plays an important role by the MDSD approach, for it shall allow better reasonings about aspects of interest. In other words, engineers discoursing about a model that holds only the information in which they are interested, should be less prone to be distracted by noise and, thus, less prone to make mistakes.

But as the use of models grows, also grows the need for tools that support engineers in tasks like storage and management of models, model checking, model verification or model transformation. The latter is a special problem of the realm of MDSD and consists, basically, on creating models automatically out of other models. In practice, the possibility of transforming models comes from the fact that different models may represent intersecting parts of the system under construction. One example of such a situation is the transformation of a class diagram into source-code or the compilation of source-code into machine-code.

Numerous solutions for the model transformation problem have been proposed by academy and industry. One branch of such solutions is the so-called operational approach, which focuses on the description of transformations either through imperative general-purpose languages, like Java, or domain-specific languages, like QVT-O. Another branch is the so-called relational approach, which focuses on the specification of transformations by means of a declarative language or formalism that embodies the relations between the elements of the different models to be transformed. Examples for this approach include QVT-R, ATL, and the graph grammar approach, which is grounded in the theories of graphs and formal languages to formalize models and describe relations between them. A specialization of the graph grammar formalism is the triple graph grammar (TGG) [Sch94], that consists of specifying transformations by means of context-sensitive grammars of, so-called, triple graphs.

Triple graphs are composed of three graphs, the source and the target graphs, representing two models, and the correspondence graph that connects the source and the target through arrows. A triple graph can be used to express the relationship between two graphs through the arrows between their vertices. More formally, a triple graph $G_s \overset{ms}{\leftarrow} G_c \overset{mt}{\rightarrow} G_t$ consists of three disjunct labeled graphs G_s, G_c, G_t, called source, correspondence, and target graphs, respectively, where the G_s and G_t contain elements of the two models of interest, and G_c contains elements that connect G_s and G_t via two partial mappings $ms : G_c \nrightarrow G_s$ and $mt : G_c \nrightarrow G_t$ [Sch94].

A TGG consists of a set of rules of the form $L \rightarrow R$, where L and R are triple graphs. The application of a rule $L \rightarrow R$ on a triple graph G can be informally understood as the replacement of the occurrence of L in G by R. By this means, a TGG, analogously to common string grammars, characterizes a language of triple graphs, that consists of all triple graphs generated by consecutive rule applications starting from a special initial triple graph Z_G. In this sense, a TGG describes a language of pairs of graphs whose vertices have a certain relationship. For the context of model transformation, in which one is interested in defining a translator from a source model to a target model, a TGG can be used to describe the set of all correctly translated source models and its correspondent target models, in form of a language of triple graphs.

Despite the various positive aspects of TGG, like a well-founded theory and a reasonable tool support [ALS16], we have identified, that for some scenarios, a transformation described with TGG results in a grammar that is too big and difficult to comprehend. We judge, this downside stems from the absence of the concept of non-terminal symbols in the TGG formalism. This concept allows, in the theory of formal languages, for a very effective representation of abstract entities in string grammars, what in turn makes grammars more comprehensible and easier to build. Moreover, it enables the hierarchical classification of grammars, known as the Chomsky hierarchy, that assigns different theoretical characteristics (e.g. generative power, parsing complexity) to different classes. Such characteristics have paved the way for the implementation of efficient parsers for specific classes of grammars, as well as, efficient compiler generators for programming languages.

We expect that a TGG formalism that describes model transformations in a more compact manner and that makes it possible to encode models' abstract concepts efficiently through non-terminal symbols be more comprehensible and make model transformations easier to be constructed, verified and validated. Therewith, such a TGG would lead to an enhancement of the quality of the software being constructed.

Hence, motivated by this benefit, the main objective of this thesis is to provide a novel formalism that redefines the standard triple graph grammars and introduces the notion of non-terminal symbols to create a context-free triple graph grammar formalism. In particular, we also aim at reviewing key aspects of the current state-of-the-art, like the parsing of graph grammars; at discussing some theoretical and practical aspects of our new TGG formalism; and at demonstrating how it can be used for solving the model transformation problem.

In order to build our new TGG formalism, we mix an already established graph grammar technique that supports non-terminal symbols called *graph grammar with neighborhood-controlled embedding* (NCE graph grammar) [JR82] and the standard definition of TGG [Sch94] and name it NCE TGG. A NCE TGG consists of a set of rules of the form $A \to R$, where A is a symbol and R is a triple graph. The application of a rule $A \to R$ on a triple graph G can be informally understood as the replacement of the occurrence of a vertex labeled with A in G by R. As in the standard TGG, a NCE TGG also characterizes a language of triple graphs holding all correctly transformed source and target graphs.

In order to demonstrate the application of our approach, we formalize the model transformation problem and show, supported by theoretical results, that a NCE TGG can be used to specify model transformations, which are then executed by our transformation algorithm with polynomial time complexity. To enhance the generative power of our basic version of NCE TGG, we extend it with a mechanism of application conditions, called PAC NCE TGG, that allows us to study our proposal in more practical scenarios and for which we also develop a parsing and a transformation algorithm. The challenges and problems of a concrete implementation of such algorithms are also discussed and a case study containing an analysis in depth of two instances of the model transformation problem specified with NCE TGG and PAC NCE TGG is thereupon put forward.

Lastly, for the purpose of evaluating the usability of our proposal, we compare the size of 5 model transformations specified with PAC NCE TGG and with the standard TGG. To assess the performance of our implemented transformer, we execute it on the same model transformation specifications for several models with different sizes and report the results.

In summary, our proposed PAC NCE TGG formalism outperforms the standard TGG in one of the 5 evaluated cases with a specification almost one order of magnitude smaller, and is able to describe one model transformation that we could not do with the standard TGG. Negatively, PAC NCE TGG is outperformed by standard TGG in the other 3 evaluated cases and our implementation is considerably slower than the state-of-the-art for TGG transformer eMoflon. Nonetheless, we believe that the outcomes of this thesis are of relevance for the current state of the research in the field and we judge that it contributes positively for the state-of-the-art.

The remainder of this thesis is as follows, in Chapter 2 we present a literary review of research works related to this thesis with a special focus on the topic of graph grammars with non-terminal symbols; in Chapter 3 we provide the theoretical background necessary for the definition of the NCE TGG formalism; in Chapter 4 we present our argumentation of how NCE TGG solves the model transformation problem and a transformation algorithm; in Chapter 5 we extend NCE TGG by adding application conditions and demonstrate how the parsing and transformation works for this extension; in Chapter 6 we discuss the details and open challenges of our implemented transformer; in Chapter 7 two representative examples for specifications of model transformations with NCE TGG and PAC NCE TGG are analyzed in depth aiming for the practical application of our proposal; in Chapter 8 an experimental empirical evaluation of usability and performance is exposed; and, finally, in Chapter 9 we close our exposition with a summary and an outlook about this work.

2. Related Works

In this section, we offer a literary review on the topics of graph grammars and triple graph grammars as well as we indicate published works that are related with our approach. Here, we focus on the node label and the hyperedge replacement approach for graph grammars. Nevertheless, the field does not restrict to this topic, instead, there is a myriad of different approaches to it, for example, the algebraic approach [ERKM99]. We refer to context-free and context-sensitive grammars, inspired by the use of such classification for string grammars, in a relaxed way without any compromise to the correct definition of context-freeness for graph grammars.

Hyperedge replacement graph grammars (HRG) are context-free grammars with semantics based on the replacement of hyperedges by hypergraphs [DKH97] governed by morphisms. Prominent polynomial-time top-down and shift-reduce parsing techniques for classes of such grammars can be found in [DHM15, DHM17, BDE16, CAB$^+$13] and applications for syntax definition of a visual language can be found in [Min06, EM98].

We divide the node label replacement approaches into context-sensitive and context-free approaches, we refer to context-sensitive and context-free grammars, inspired by the use of such classification for string grammars, in a relaxed way without any compromise to any definition of context-freeness for graph grammars. The context-sensitive field includes the *layered graph grammar*, whose semantics consists of the replacement of graphs by other graphs governed by morphisms [RS97] and for which exponential-time bottom-up parsing algorithms have been proposed [RS95, BTS00, FMM11]. Another context-sensitive formalism is the *reserved graph grammar*, that is based on the replacement of directed graphs by necessarily greater directed graphs governed by simple embedding rules [ZZC01] and for which exponential and polynomial-time bottom-up algorithms have been proposed in [ZZKS05, ZZLL17].

In the node label replacement context-free formalisms stand out the *node label controlled graph grammar* (NLC) and its successor *graph grammar with neighborhood-controlled embedding* (NCE). NLC is based on the replacement of one vertex by a graph, governed by embedding rules written in terms of the vertex's label [RW86]. For various classes of these grammars, there exists polynomial-time top-down and bottom-up parsing algorithms [Fla93, FF14, RW86, Wan91]. The recognition complexity and generation power of such grammars have also been analyzed [Fla98, Kim12]. NCE occurs in several formulations, including a context-sensitive one, but here we focus on the context-free formulation, where one vertex is replaced by a graph, and the embedding rules are written in terms of the vertex's neighbors [JR82, SW98]. For some classes of these grammars, polynomial-time bottom-up parsing algorithms and automaton formalisms were proposed and analyzed [Kim01, BS05]. In special, one of these classes is the *boundary graph grammar with neighborhood-controlled embedding* (BNCE), that is used to construct our own formalism. Moreover, it is worth mentioning that, according to [ER90], BNCE and HRG have the same generative power.

Beyond the approaches presented above, there is a myriad of alternative proposals for graph grammars, including a context-sensitive NCE [AKTY99], an edge-based grammar [SZH$^+$15], a grammar that replaces star graphs by other graphs [DHJM10], a coordinate system-based grammar [KZZ06] and a regular graph grammar [GLM17].

Regarding TGG [Sch94], a 20 years review of the realm is put forward by Anjorin et al. [ALS16]. In special, advances are made in the direction of expressiveness with the introduction of application conditions [KLKS10] and of modularization [ASLS14]. Furthermore, in the algebraic approach for graph grammars, we have found proposals that introduce inheritance [BEDLT04, HET08] and variables [Hof05] to the formalisms. Nevertheless, we do not know any approach that introduces non-terminal symbols to TGG with the purpose of gaining expressiveness or usability. In this sense our proposal brings something new to the current state-of-the-art.

3. Theoretical Review

In this section, we introduce the theoretical concepts used along this thesis. The definitions below are taken from the works of ...We first go on to define graphs and graph grammars and then, building upon it, we construct the so-called triple graph grammars.

3.1 Graph Grammars

We start presenting our notation for graphs and grammars, accompanied by examples, then we introduce the dynamic aspects of the graph grammar formalism that is, how graph grammars are to be interpreted.

Definition 3.1. A *directed labeled graph* G over the finite set of symbols Σ, $G = (V, E, \phi)$ consists of a finite set of vertices V, a set of labeled directed edges $E \subseteq V \times \Sigma \times V$ and a total vertex labeling function $\phi : V \to \Sigma$. Directed labeled graphs are often referred to simply as graphs. For a fixed graph G we refer to its components as V_G, E_G and ϕ_G. Moreover, we denote the set of all graphs over Σ by $\mathcal{G}_\Sigma$. In special, we do not allow loops (vertices of the form (v, l, v)), but multi-edges with different labels are allowed.

If $\phi_G(v) = a$ we say v is labeled by a. Two vertices v and w are neighbors (also adjacent) if, and only if,there is one or more edges between them, that is, $(v, \_, w) \in E_G \lor (w, \_, v) \in E_G$. Two graphs G and H are disjoint if, and only if,$V_G \cap V_H = \emptyset$. For two graphs G and H, we write $G \subseteq H$ if, and only if, $V_G \subseteq V_H, E_G \subseteq E_H$ and $\phi_G \subseteq \phi_H$

We define also de function $\text{neigh}_G : 2^{V_G} \to 2^{V_G}$, that applied to U gives the set of neighbors of vertices in U minus U. That is $\text{neigh}_G(U) = \{v \in V_G \setminus U \mid \text{ exists a } (v, l, u) \in E_G \text{ or a } (u, l, v) \in E_G \text{ with } u \in U\}$

Definition 3.2. A *morphism* of graphs G and H is a mapping $m : V_G \to V_H$.

Definition 3.3. An *isomorphism* of directed labeled graphs G and H is a bijective mapping $m : V_G \to V_H$ that maintains the connections between vertices and their labels, that is, $(v, l, w) \in E_G$ if, and only if, $(m(v), l, m(w)) \in E_H$ and

$\phi_G(v) = \phi_H(m(v))$. In this case, G and H are said to be isomorphic, we write $G \cong H$, and we denote the equivalence class of all graphs isomorphic to G by $[G]$. Notice that, contrary to isomorphisms, morphism do not require bijectivity nor label or edge-preserving properties.

Definition 3.4. A *Γ-boundary* graph G is such that vertices labeled with any symbol from Γ are not neighbors. That is, the graph G is Γ-boundary if, and only if, $\nexists(v, \_, w) \in E_G.\ \phi_G(v) \in \Gamma \wedge \phi_G(w) \in \Gamma$.

We use graphs to represent models, first because of the extensive theory behind them and, second, because their very abstract structure suits the description of a large spectrum of practical models. In the following we introduce graph grammars, which also suit our needs very well, because they serve as a very effective tool to characterize (possibly infinite) sets of graphs using very few notation.

Definition 3.5. A *graph grammar with neighborhood-controlled embedding* (NCE graph grammar) $GG = (\Sigma, \Delta \subseteq \Sigma, S \in \Sigma, P)$ consists of a finite set of symbols Σ that is the alphabet, a subset of the alphabet $\Delta \subseteq \Sigma$ that holds the terminal symbols (we define the complementary set of non-terminal symbols as $\Gamma := \Sigma \setminus \Delta$), a special symbol of the alphabet $S \in \Sigma$ that is the start symbol, and a finite set of production rules P of the form $(A \rightarrow R, \omega)$ where $A \in \Gamma$ is the so-called left-hand side, $R \in \mathcal{G}_\Sigma$ is the right-hand side and $\omega : V_R \nrightarrow 2^{\Sigma\times\Sigma}$ is the partial embedding function from the R's vertices to pairs of edge and vertex labels. NCE graph grammars are often referred to as graph grammars or simply as grammars.

For convenience, define the start graph of GG as $Z_{GG} := (\{v_s\}, \emptyset, \{v_s \mapsto S\})$

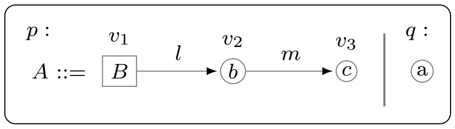
Vertices from the right-hand sides of rules labeled by non-terminal (terminal) symbols are said to be non-terminal (terminal) vertices.

Notice that, in the original definition of NCE grammars [JR82], the left-hand side of the productions were allowed to contain any connected graph. So, strictly speaking, the definition above characterizes actually a 1-edNCE graph grammar, that contains only one element in the left-hand side and a directed edge-labeled graph in the right-hand side. Nevertheless, for simplicity, we use the denomination NCE to mean a 1-edNCE grammar.

Definition 3.6. A *boundary graph grammar with neighborhood-controlled embedding* (BNCE graph grammar) GG is such that non-terminal vertices of the right-hand sides of rules are not neighbors. That is, the graph grammar GG is boundary if, and only if,all its rules' right-hand sides are Γ-boundary graphs.

In the following, we present our concrete syntax inspired by the well-known Backus-naur form to denote NCE graph grammar rules. Let $GG = (\{A, B, a, b,\ c, l, m\}, \{a, b, c, l, m\}, A, \{p, q\})$ be a graph grammar with production rules $p = (A \rightarrow G, \omega)$

and $q = (A \to H, \zeta)$ where $G = (\{v_1, v_2, v_3\}, \{(v_1, l, v_2), (v_2, m, v_3)\}, \{v_1 \mapsto B, v_2 \mapsto b, v_3 \mapsto c\})$, and $H = (\{u_1\}, \emptyset, \{u_1 \mapsto a\})$, we denote p and q together as



Observe that, we use squares for non-terminal vertices, circles for terminal vertices, position the respective label inside the shape and the (possibly omitted) identifier near it. Near each edge its respective label is positioned. The embedding function is not included in the notation, so it is expressed separately, if necessary.

In the sequel, we introduce the dynamic aspects of NCE graph grammars by means of the concepts of derivation step, derivation and language.

Definition 3.7. Let $GG = (\Sigma, \Delta, S, P)$ be a graph grammar and G and H be two graphs over Σ that are disjoint to all right-hand sides from P, G *concretely derives in one step into* H with rule r and vertex v, we write $G \overset{r,v}{\Rrightarrow}_{GG} H$ and call it a *concrete derivation step*, if, and only if, the following holds:

$$\begin{aligned}
r &= (A \to R, \omega) \in P \text{ and } A = \phi_G(v) \text{ and} \\
V_H &= (V_G \setminus \{v\}) \cup V_R \text{ and} \\
E_H &= (E_G \setminus (\{(v,l,w) \mid (v,l,w) \in E_G\} \cup \{(w,l,v) \mid (w,l,v) \in E_G\})) \\
&\quad \cup E_R \\
&\quad \cup \{(w,l,t) \mid (w,l,v) \in E_G \wedge (l, \phi_G(w)) \in \omega(t)\} \\
&\quad \cup \{(t,l,w) \mid (v,l,w) \in E_G \wedge (l, \phi_G(w)) \in \omega(t)\} \text{ and} \\
\phi_H &= (\phi_G \setminus \{(v,x) \mid x \in \Sigma\}) \cup \phi_R
\end{aligned}$$

Notice that, without loss of generality, we set $\omega(t) = \emptyset$ for all vertices t without an image defined in ω.

If G concretely derives in one step into any graph H' isomorphic to H, we say it *derives in one step into* H' and write $G \overset{r,v}{\Rightarrow}_{GG} H'$.

When GG, r or v are clear in the context or irrelevant we might omit them and simply write $G \Rrightarrow H$ or $G \Rightarrow H$. Moreover, we denote the reflexive transitive closure of $\Rightarrow$ by $\Rightarrow^*$ and, for $G \Rightarrow^* H'$, we say G derives into H'.

A concrete derivation can be informally understood as the replacement of a non-terminal vertex v and all its adjacent edges in G by a graph R plus edges e from former neighbors w of v to some vertices t of R, provided e's label and w's label are in the embedding specification $\omega(t)$. That is, the embedding function ω of a rule specifies which neighbors of v are to be connected with which vertices of R, according to their labels and the adjacent edges' labels. The process that governs the creation of these edges is called embedding and can occur in various forms in different graph grammar formalisms. We opted for a rather simple approach, in which the edges' directions and labels are maintained and cannot be used to define embedding. As an additional note, it is worth mentioning, that string grammars have no embedding because a replaced symbol in a string has "connections" only with its left and right neighbors, so the replacement is always "connected" with both sides.

Definition 3.8. A *derivation* D in the grammar GG is a non-empty sequence of derivation steps and is written as

$$D = (G_0 \overset{r_0,v_0}{\Rightarrow} G_1 \overset{r_1,v_1}{\Rightarrow} G_2 \overset{r_2,v_2}{\Rightarrow} \dots \overset{r_{n-1},v_{n-1}}{\Rightarrow} G_n)$$

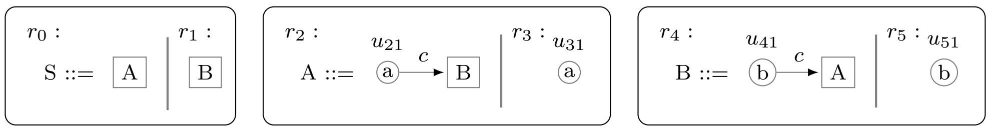
Definition 3.9. The *language* $L(GG)$ generated by the grammar GG is the set of all graphs containing only terminal vertices derived from the start graph Z_{GG}, that is

$$L(GG) = \{H \text{ is a graph over } \Delta \text{ and } Z_{GG} \Rightarrow^* H\}$$

It is clear that, for every graph $G \in L(GG)$, there is at least one finite derivation $(Z_{GG} \overset{r_0,v_0}{\Rightarrow} \dots \overset{r_{n-1},v_{n-1}}{\Rightarrow} G)$ with $n \geq 1$, but it is not guaranteed that this derivation be unique. In the case that there is more than one derivation for a G, we say that the grammar GG is ambiguous.

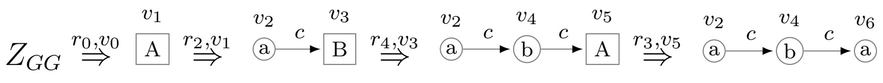
Below we give one example of a grammar whose language consists of all chains of one or more vertices with interleaved vertices labeled with a and b.

Example 3.1. Chains of a's and b's. $GG = (\{S, A, B, a, b, c\}, \{a, b, c\}, S, P)$, where $P = \{r_0, r_1, r_2, r_3, r_4, r_5\}$ is denoted by



with $\omega_0 = \omega_1 = \emptyset$, $\omega_2(u_{21}) = \omega_3(u_{31}) = \{(c, b)\}$ and $\omega_4(u_{41}) = \omega_5(u_{51}) = \{(c, a)\}$ being the complete definition of the embedding functions of the rules, $r_0, r_1, r_2, r_3, r_4, r_5$ respectively.

The graph $G = \begin{tikzcd} ⓐ \arrow[r, "c"] & ⓑ \arrow[r, "c"] & ⓐ \end{tikzcd}$ belongs to $L(GG)$ because it contains only terminal vertices and Z_{GG} derives into it using the following derivation:



3.2 Triple Graph Grammars

Building upon the concepts of graphs and graph grammars, we present, in the following, our understanding over triple graphs and triple graph grammars (TGG), supported by the TGG specification from [Sch94].

Definition 3.10. A *directed labeled triple graph* $TG = G_s \overset{ms}{\leftarrow} G_c \overset{mt}{\rightarrow} G_t$ over Σ consists of three disjoint directed labeled graphs over Σ (see 3.1), respectively, the source graph G_s, the correspondence graph G_c and the target graph G_t, together with two bijective partial morphisms (see 3.2) $ms : V_{G_c} \nrightarrow V_{G_s}$ and $mt : V_{G_c} \nrightarrow G_{G_t}$, called source and target morphisms, respectively. Directed labeled triple graphs are often referred to simply as triple graphs and we might omit the morphisms' names in the notation. Moreover, we denote the set of all triple graphs over Σ as $\mathcal{TG}_\Sigma$. We might refer to all vertices of TG by $V_{TG} := V_s \cup V_c \cup V_t$, all edges by $E_{TG} := E_s \cup E_c \cup E_t$ and the complete labeling function by $\phi_{TG} := \phi_{G_s} \cup \phi_{G_c} \cup \phi_{G_t}$. Moreover, we define the special empty triple graph as $\varepsilon := E \overset{ms}{\leftarrow} E \overset{mt}{\rightarrow} E$ with $E = (\emptyset, \emptyset, \emptyset)$ and $ms = mt = \emptyset$.

Definition 3.11. A *triple isomorphism* of directed labeled triple graphs $G = (G_s \overset{gs}{\leftarrow} G_c \overset{gt}{\rightarrow} G_t)$ and $H = (H_s \overset{hs}{\leftarrow} H_c \overset{ht}{\rightarrow} H_t)$ is a bijective mapping $m : V_G \rightarrow V_H$ that maintains the connections between vertices as well as their labels and the source and target morphisms, that is, $(v, l, w) \in E_G$ if, and only if, $(m(v), l, m(w)) \in E_H$ and $\phi_G(v) = \phi_H(m(v))$ and $v \in G_c$ if, and only if, $v \in \operatorname{dom} gs \rightarrow m(gs(v)) = hs(m(v))$ and $v \in \operatorname{dom} gt \rightarrow m(gt(v)) = ht(m(v))$. In this case, we write $G \cong H$, and we denote the equivalence class of all triple graphs isomorphic to G also by $[G]$.

Definition 3.12. A *Γ-boundary* triple graph $TG = G_s \leftarrow G_c \rightarrow G_t$ is such that G_s, G_c and G_t are Γ-boundary graphs.

As stated before, triple graphs are for us a good tool to express relations between the vertices of two graphs. In the context of model transformation, where graphs represent models, a triple graph holds, for example, a source model and a target model generated from the source, together with the relationship between their vertices. We also advise that in literature, TGG are often modeled as typed graphs, but we judge that for our circumstance labeled graphs fit better and we are convinced that such divergence does not threat the validity of our approach.

Below we start introducing the standard definition of TGG of the current research's literature. As the reader should notice, this definition of TGG does not fit our needs optimally, because it defines a context-sensitive graph grammar whilst we wish a context-free graph grammar to use together with the NCE graph grammar formalism. Hence, after presenting the conventional TGG definition, we refine it to create a NCE TGG, that fits our context best.

Definition 3.13. A *triple graph grammar* $TGG = (\Sigma, \Delta \subseteq \Sigma, S \in \Sigma, P)$ consists of, analogously to graph grammars (see Definition 3.5), an alphabet Σ, a set of terminal symbols Δ, a start symbol S and a set of production rules P of the form $L \to R$ with $L = L_s \leftarrow L_c \to L_t$ and $R = R_s \leftarrow R_c \to R_t$ and $L_s \subseteq R_s, L_c \subseteq R_c, L_t \subseteq R_t, \sigma_l \subseteq \sigma_r$ and $\tau_l \subseteq \tau_r$.

Definition 3.14. A *triple graph grammar with neighborhood-controlled embedding* (NCE TGG) $TGG = (\Sigma, \Delta \subseteq \Sigma, S \in \Sigma, P)$ consists of, an alphabet Σ, a set of terminal symbols Δ (also define $\Gamma := \Sigma \setminus \Delta$), a start symbol S and a set of production rules P of the form $(A \to (R_s \leftarrow R_c \to R_t), \omega_s, \omega_t)$ with $A \in \Gamma$ being the left-hand side, $(R_s \leftarrow R_c \to R_t) \in \mathcal{TG}_\Sigma$ the right-hand side and $\omega_s : V_{R_s} \nrightarrow 2^{\Sigma \times \Sigma}$ and $\omega_t : V_{R_t} \nrightarrow 2^{\Sigma \times \Sigma}$ the partial embedding functions from the right-hand side's vertices to pairs of edge and vertex labels. We might refer to the complete embedding function by $\omega := \omega_s \cup \omega_t$.

For convenience, define the start triple graph of TGG as $Z_{TGG} := Z_s \overset{ms}{\leftarrow} Z_c \overset{mt}{\rightarrow} Z_t$ where $Z_s = (\{s_0\}, \emptyset, \{s_0 \mapsto S\})$, $Z_c = (\{c_0\}, \emptyset, \{c_0 \mapsto S\})$, $Z_t = (\{t_0\}, \emptyset, \{t_0 \mapsto S\})$, $ms = \{c_0 \mapsto s_0\}$ and $mt = \{c_0 \mapsto t_0\}$. Production rules of triple graph grammars are also called triple rules.

Definition 3.15. A *boundary triple graph grammar with neighborhood-controlled embedding* (BNCE TGG) is such that non-terminal vertices of the right-hand sides of rules are not neighbors. That is, the triple graph grammar TGG is boundary if, and only if,all its rules' right-hand sides are Γ-boundary triple graphs.

The most important difference between the traditional TGG and the NCE TGG, is that the former allows any triple graph to occur in the left-hand sides, whereas the latter only one symbol. In addition to that, traditional TGG requires that the whole left hand side occur also in the right-hand side, that is to say, the rules are monotonic crescent. Therewith, embedding is not an issue, because an occurrence of the left-hand side is not effectively replaced by the right-hand side, instead, only new vertices are added. On the other hand, NCE TGG has to deal with embedding through the embedding function.

In the following, the semantics for NCE TGG is presented analogously to the semantics for NCE graph grammars.

Definition 3.16. Let $TGG = (\Sigma, \Delta, S, P)$ be a NCE TGG and $G = G_s \overset{gs}{\leftarrow} G_c \overset{gt}{\rightarrow} G_t$ and $H = H_s \overset{hs}{\leftarrow} H_c \overset{ht}{\rightarrow} H_t$ be two triple graphs over Σ disjoint from any right-hand side from P, G *concretely derives in one step into* H with rule r and distinct vertices

v_s, v_c, v_t, we write $G \overset{r,v_s,v_c,v_t}{\Rrightarrow}_{TGG} H$ if, and only if, the following holds:

$$\begin{aligned}
r &= (A \to (R_s \overset{rs}{\leftarrow} R_c \overset{rt}{\to} R_t), \omega_s, \omega_t) \in P \text{ and}\\
A &= \phi_{G_s}(v_s) = \phi_{G_c}(v_c) = \phi_{G_t}(v_t) \text{ and}\\
V_{H_s} &= (V_{G_s} \setminus \{v_s\}) \cup V_{R_s} \text{ and}\\
V_{H_c} &= (V_{G_c} \setminus \{v_c\}) \cup V_{R_c} \text{ and}\\
V_{H_t} &= (V_{G_t} \setminus \{v_t\}) \cup V_{R_t} \text{ and}\\
E_{H_s} &= (E_{G_s} \setminus (\{(v_s, l, w) \mid (v_s, l, w) \in E_{G_s}\} \cup \{(w, l, v_s) \mid (w, l, v_s) \in E_{G_s}\}))\\
&\quad \cup E_{R_s}\\
&\quad \cup \{(w, l, t) \mid (w, l, v_s) \in E_{G_s} \wedge (l, \phi_{G_s}(w)) \in \omega_s(t)\}\\
&\quad \cup \{(t, l, w) \mid (v_s, l, w) \in E_{G_s} \wedge (l, \phi_{G_s}(w)) \in \omega_s(t)\} \text{ and}\\
E_{H_c} &= (E_{G_c} \setminus (\{(v_c, l, w) \mid (v_c, l, w) \in E_{G_c}\} \cup \{(w, l, v_c) \mid (w, l, v_c) \in E_{G_c}\}))\\
&\quad \cup E_{R_c} \text{ and}\\
E_{H_t} &= (E_{G_t} \setminus (\{(v_t, l, w) \mid (v_t, l, w) \in E_{G_t}\} \cup \{(w, l, v_t) \mid (w, l, v_t) \in E_{G_t}\}))\\
&\quad \cup E_{R_t}\\
&\quad \cup \{(w, l, t) \mid (w, l, v_t) \in E_{G_t} \wedge (l, \phi_{G_t}(w)) \in \omega_t(t)\}\\
&\quad \cup \{(t, l, w) \mid (v_t, l, w) \in E_{G_t} \wedge (l, \phi_{G_t}(w)) \in \omega_t(t)\} \text{ and}\\
hs &= (gs \setminus \{(v_c, x) \mid x \in V_{G_s}\}) \cup rs\\
ht &= (gt \setminus \{(v_c, x) \mid x \in V_{G_t}\}) \cup rt\\
\phi_{H_s} &= (\phi_{G_s} \setminus \{(v_s, x) \mid x \in \Sigma\}) \cup \phi_{R_s} \text{ and}\\
\phi_{H_c} &= (\phi_{G_c} \setminus \{(v_c, x) \mid x \in \Sigma\}) \cup \phi_{R_c} \text{ and}\\
\phi_{H_t} &= (\phi_{G_t} \setminus \{(v_t, x) \mid x \in \Sigma\}) \cup \phi_{R_t}
\end{aligned}$$

Notice that, without loss of generality, we set $\omega(t) = \emptyset$ for all vertices t without an image defined in ω.

Analogously to graph grammars, if $G \overset{r,v_s,v_c,v_t}{\Rrightarrow}_{TGG} H$ and $H' \in [H]$, then $G \overset{r,v_s,v_c,v_t}{\Rrightarrow}_{TGG} H'$, moreover the reflexive transitive closure of $\Rrightarrow$ is denoted by $\Rrightarrow^*$ and we call these relations by the same names as before, namely, *derivation in one step* and *derivation*. We might also omit identifiers.

A concrete derivation of a triple graph $G = G_s \overset{gs}{\leftarrow} G_c \overset{gt}{\to} G_t$ can de informally understood as concrete derivations (see 3.7) of G_s, G_c and G_t according to the right-hand sides R_s, R_c and R_t. The only remark is the absence of an embedding mechanism for the correspondence graph, which edges are not important for our application. Nevertheless, the addition of such a mechanism for the correspondence graph should not be a problem if it is desired.

Definition 3.17. A *derivation* D in the triple graph grammar TGG is a non-empty sequence of derivation steps

$$D = (G_0 \overset{r_0,s_0,c_0,t_0}{\Rightarrow} G_1 \overset{r_1,s_1,c_1,t_1}{\Rightarrow} G_2 \overset{r_2,s_2,c_2,t_2}{\Rightarrow} \ldots \overset{r_{n-1},s_{n-1},c_{n-1},t_{n-1}}{\Rightarrow} G_n)$$

Definition 3.18. The *language* $L(TGG)$ generated by the triple grammar TGG is the set of all triple graphs containing only terminal vertices derived from the start triple graph Z_{TGG}, that is

$$L(TGG) = \{H \text{ is a triple graph over } \Delta \text{ and } Z_{TGG} \Rightarrow^{*} H\}$$

Our concrete syntax for NCE TGG is similar to the one for NCE graph grammars and is presented below by means of the Example 3.2. The only difference is at the right-hand sides, that include the morphisms between the correspondence graph and source and target graphs depicted with dashed lines.

Example 3.2. *Pseudocode* to *Controlflow*. This example illustrates the definition of a BNCE TGG that characterizes the language of all *Pseudocode* graphs together with their respective *Controlflow* graphs. A *Pseudocode* graph is an abstract representation of a program written in a pseudo-code where vertices refer to *actions*, *ifs* or *whiles* and edges connect these items together according to how they appear in the program. A *Controlflow* graph is a more abstract representation of a program, where vertices can only be either a *command* or a *branch*.

Consider, for instance, the program *main*, written in a pseudo-code, and the triple graph TG in Figure 3.1. The triple graph TG consists of the *Pseudocode* graph of *main* connected to the *Controlflow* graph of the same program through the correspondence graph in the middle of them. In such graph, the vertex labels of the *Pseudocode* graph p, i, a, w correspond to the concepts of *program*, *if*, *action* and *while*, respectively. The edge label f is given to the edge from the vertex p to the program's first statement, x stands for *next* and indicates that a statement is followed by another statement, p and n stand for *positive* and *negative* and indicate which assignments correspond to the positive of negative case of the *if*'s evaluation, finally l stands for *last* and indicates the last action of a loop. In the *Controlflow* graph, the vertex labels g, b, c stand for the concepts of *graph*, *branch* and *command*, respectively. The edge label r is given to the edge from the vertex g to the first program's statement, x, p and n mean, analogous to the former graph, *next*, *positive* and *negative*. In the correspondence graph, the labels pg, ib, ac, wb serve to indicate which labels in the source and target graphs are being connected through the triple graph's morphism.

The main difference between the two graphs is the absence of the w label in the *Controlflow* graph, what makes it encode loops through the combination of b-labeled vertices and x-labeled edges.

The TGG that specifies the relation between these two types of graphs is $TGG = (\{S, A, p, a, i, w, g, b, c, f, x, n, l, r, pg, ac, ib, wb\}, \{p, a, i, w, g, b, c, f, x, n, l, r, pg, ac, ib, wb\}, S, P)$, where $P = \{r_i \mid 0 \leq i \leq 5\}$ is denoted by

```
program $main(n)$
if $n < 0$ then
    return Nothing
else
    $f \leftarrow 1$
    while $n > 0$ do
        $f \leftarrow f * n$
        $n \leftarrow n - 1$
    end while
    return Just $f$
end if
```

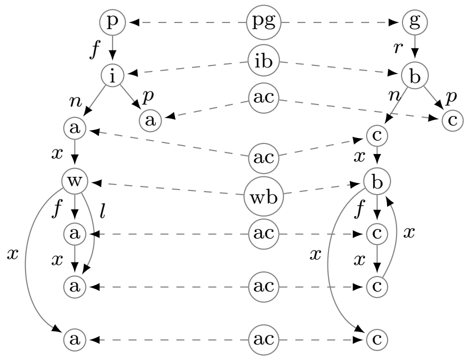
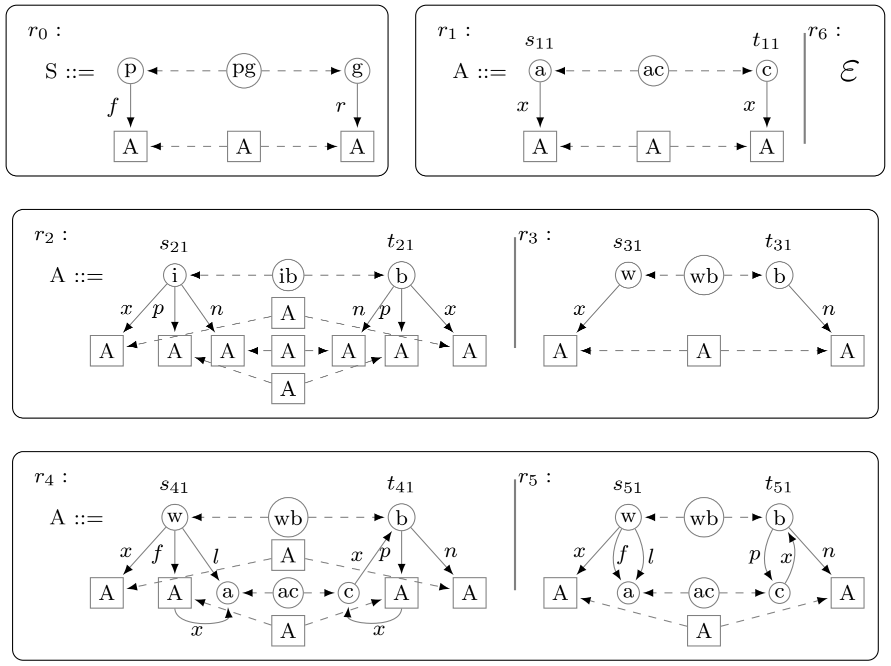


Figure 3.1: A program written in pseudo-code on the left and its correspondent triple graph with the *PseudoCode* and the *ControlFlow* graphs on the right



with $\sigma_0 = \emptyset$, $\sigma_1(s_{11}) = \sigma_2(s_{21}) = \sigma_3(s_{31}) = \sigma_4(s_{41}) = \sigma_5(s_{51}) = \{(f,p),(x,a),(x,i),(x,w),(p,i),(n,i),(l,w),(f,w)\}$ and $\tau_1(t_{11}) = \tau_2(t_{21}) = \tau_3(t_{31}) = \tau_4(t_{41}) = \tau_5(t_{51}) = \{(r,g),(x,c),(x,b),(p,b),(n,b)\}$ being the complete definition of the source and target embedding functions of the rules r_0 to r_5, respectively.

The rule r_0 relates programs to graphs, r_1 actions to commands, r_2 ifs to branches, r_3 empty whiles to simple branches, r_4 filled whiles to filled loops with branches, r_5 whiles with one action to loops with branches with one command and, finally, r_6 produces an empty graph from a symbol A, what allows any derivation in the grammar to finish.

The aforementioned triple graph TG is in $L(TGG)$, because the derivation $Z_{TGG} \overset{r_0}{\Rightarrow} G_1 \overset{r_3}{\Rightarrow} G_2 \overset{r_6}{\Rightarrow} G_3 \overset{r_1}{\Rightarrow} G_4 \overset{r_6}{\Rightarrow} G_5 \overset{r_1}{\Rightarrow} G_6 \overset{r_4}{\Rightarrow} G_7 \overset{r_1}{\Rightarrow} G_8 \overset{r_6}{\Rightarrow} G_9 \overset{r_1}{\Rightarrow} G_{10} \overset{r_6}{\Rightarrow} TG$ is a derivation in TGG with appropriate G_i for $1 \leq i \leq 10$.

3.3 Parsing of Graphs with Graph Grammars

In the last section we cleared how the concepts of graphs and languages fit together. In this section we are interested in the problem of deciding, given a BNCE graph grammar GG and a graph G, whether $G \in L(GG)$. This is sometimes called the *membership* problem and can be solved through a recognizer algorithm that always finishes answering yes if and only if $G \in L(GG)$ and no otherwise. A slight extension of this problem is the *parsing* problem, which consists of deciding if $G \in L(GG)$ and finding a derivation $Z_{GG} \Rightarrow^* G$.

The parsing algorithm posed in this section is an imperative view of the method proposed by (), which is basically a version fro graphs of the well-known CYK (Cocke-Young-Kassami) algorithm for parsing of strings with a context-free (string) grammar. Preliminarily to the actual algorithm's presentation, we introduce some necessary concepts that are used by it. The first of them is the neighborhood preserving normal form.

Definition 3.19. A BNCE graph grammar $GG = (\Sigma, \Delta, S, P)$ is *neighborhood preserving* (NP), if and only if, the embedding of each rule with left-hand side A is greater or equal than the context of each A-labeled vertex in the grammar. That is, let

$$\mathrm{cont}_{(A \to R, \omega)}(v) = \{(l, \phi_R(w)) \mid (v, l, w) \in E_R \text{ or } (w, l, v) \in E_R\} \cup \omega(v)$$

be the context of v in the rule $(A \to R, \omega)$ and

$$\eta_{GG}(A) = \bigcup_{(B \to Q, \zeta) \in P, v \in V_Q, \phi_Q(v) = A} \mathrm{cont}_{B \to Q, \zeta}(v)$$

be the context of the symbol A in the grammar GG, then GG is a NP BNCE graph grammar, if and only if,

$$\forall r = (A \to R, \omega) \in P.\ \eta_{GG}(A) \subseteq \bigcup_{v\ inV_R} \omega(v)$$

If this property holds for a rule r, we say r is NP. Otherwise it is non-NP.

The NP property is important to the correctness of the parsing algorithm. Furthermore, it is guaranteed that any BNCE graph grammar can be transformed in an equivalent NP BNCE graph grammar in polynomial time. More details in [RW86] and in [SW98].

The next paragraphs present zone vertices and zone graphs, that are our understanding of the concepts also from

Definition 3.20. A *zone vertex* h of a graph G over Σ is a pair $(\sigma \in \Sigma, U \subseteq V_G)$, that is, a symbol from Σ and a subset of the vertices of G.

A zone vertex can be understood as a contraction of a subgraph of G defined by the vertices U into one vertex with symbol σ.

Definition 3.21. Let $H = \{(\sigma_0, U_0), (\sigma_1, U_1), \dots, (\sigma_m, U_m)\}$ be a set of zone vertices of a graph G over Σ with disjoint vertices (i.e. $U_i \cap U_j = \emptyset$ for all $0 \leq i, j \leq m$ and $i \neq j$) and $V(H) = \bigcup_{0 \leq i \leq m} U_i$. A *zone graph* $Z(H)$ for H is $Z(H) = (V, E, \phi)$ with V being the zone vertices, $E \subseteq V \times \Sigma \times V$ the edges between zone vertices and $\phi : V \to \Sigma$ the labeling function, determined by

$$\begin{aligned}
&V = H \cup \{(\phi_G(x), \{x\}) \mid x \in \text{neigh}_G(V(H))\} \\
&E = \{((\sigma, U), l, (\eta, T)) \mid (\sigma, U), (\eta, T) \in V \text{ and } U \neq T \text{ and} \\
&\quad (u, l, t) \in E_G \text{ and } u \in U \text{ and } t \in T\} \\
&\phi = \{(\sigma, U) \mapsto \sigma \mid (\sigma, U, W) \in V\}
\end{aligned}$$

The zone graph $Z(H)$ can be intuitively understood as a subgraph of G, where each zone vertex in $V_{Z(H)}$ is either a (σ_i, U_i) of H, which is a contraction of the vertices U_i of G, or a $(\phi_G(x), \{x\})$, which stems from x being a neighbor of some vertex in V_i.

For convenience, define $Y(H)$ as the subgraph of $Z(H)$ induced by H.

Definition 3.22. Let h be a zone vertex, r a production rule and X a (potentially empty) set of parsing trees, $(h^r \rightrightarrows X)$ is a *parsing tree*, whereby h is called the root node and X the children and r is optional. $D(pt)$ gives a derivation for the parsing tree pt, which can be calculated by performing a depth-first walk on pt, starting from its root node, producing as result a sequence of derivation steps that correspond to each visited node and its respective rule. Additionally, a set of parsing trees is called a parsing forest.

Finally, the Algorithm 3.1 displays the parsing algorithm of graphs with a NP BNCE graph grammar. Informally, the procedure follows a bottom-up strategy that tries to find production rules in GG that generate zone graphs of G until it finds a rule that generates a zone graph containing all vertices of G and finishes answering yes and returning a valid derivation for G or it exhausts all the possibilities and finishes answering no.

The variable *bup* (*bup* stands for bottom-up parsing set, see ())is started with the trivial zone vertices of G, each containing only one vertex of V_G, and grows iteratively with bigger zone vertices that can be inferred using the grammar's rules and the elements of *bup*.

The variable h stands for handle and is any subset from *bup* chosen to be evaluated for the search of new zone vertices to insert in *bup*. The procedure **select** gives one

Algorithm 3.1 Parsing Algorithm for NP BNCE Graph Grammars

```
Require: $GG$ is a valid NP BNCE graph grammar
Require: $G$ is a valid graph over $\Delta$                    ▷ $G$ has terminal vertices only
 1: function $parse(GG = (\Sigma, \Delta, S, P), G = (V_G, E_G, \phi_G))$: $Derivation$
 2:     $bup \leftarrow \{(\phi_G(x), \{x\}) \mid x \in V_G\}$          ▷ start $bup$ with trivial zone vertices
 3:     $pf \leftarrow \{(b \rightrightarrows \emptyset) \mid b \in bup\}$                ▷ initialize parsing forest
 4:     repeat
 5:         $h \leftarrow$ select$\{X \subseteq bup \mid$ for all $U_i, U_j \in X$ with $i \neq j$. $U_i \cap U_j = \emptyset\}$
 6:         for all $d \in \Gamma$ do                        ▷ for each non-terminal symbol
 7:             $r \leftarrow$ any $\{(d \rightarrow R, \omega) \in P \mid R \cong Y(h)\}$
 8:             $l \leftarrow (d, V(h))$
 9:             if $Z(\{l\}) \overset{r,l}{\Rightarrow} Z(h)$ then
10:                 $bup \leftarrow bup \cup \{l\}$                    ▷ new zone vertex found
11:                 $pf \leftarrow pf \cup \{(l^r \rightrightarrows \{(z^y \rightrightarrows X) \mid (z^y \rightrightarrows X) \in pf, z \in h\})\}$
12:             end if
13:         end for
14:     until $(S, V_G) \in bup$                           ▷ if found the root, stop
15:     return $(S, V_G) \in bup$ ? Just $D(((S, V_G)^y \rightrightarrows X) \in pf)$ : Nothing
16: end function
Ensure: $return$ is either Nothing or of the form Just $Z_{GG} \Rightarrow^* G$
```

yet not chosen handle or an empty set and cares for the termination of the execution. Then, for the chosen h, rules r with left-hand side d and right-hand side isomorphic to $Y(h)$ that produce $Z(h)$ from $Z(\{l\})$ are searched. If any is found, then $l = (d, V(h))$ is inserted into *bup*. This basically means that it found a zone vertex that encompasses the vertices $V(h)$ (a possibly bigger subset than other elements in *bup*), from which, through the application of a sequence of rules, we can produce the subgraph of G induced by $V(h)$. This information is saved in the parsing forest pf in form of a parsing tree with node l and children $(z^y \rightrightarrows X)$, already in the parsing forest pf, for all $z \in h$.

If, in some iteration the zone vertex (S, V_G) is inferred, then it means that the whole graph G can be produced through the application of a derivation starting from the start graph Z_{GG} and thus $G \in L(GG)$. This derivation is, namely, the result of a depth-first walk in the parsing tree whose root is (S, V_G). If, otherwise, all possibilities for h were exhausted without inferring such zone vertex, then Nothing is returned, what means that G cannot de parsed with GG and therefore $G \notin L(GG)$.

This parsing algorithm supports ambiguous grammars, in which case, there exists more than one derivation for at least one graph in the grammar's language. The output derivation is, in this case, non-deterministic, because of the non-deterministic selection of the handles by **select** in Line 5.

4. Model Transformation with NCE Triple Graph Grammars

As already introduced, TGG can be used to characterize languages of triple graphs holding correctly transformed models. That is, one can interpret a TGG as the description of the correctly-transformed relation between two sets of models $\mathcal{S}$ and $\mathcal{T}$, where two models $G \in \mathcal{S}$ and $T \in \mathcal{T}$ are in the relation if and only if G and T are respectively, source and target graphs of any triple graph of the language $L(TGG)$. That being said, we are interested in this section on defining a model transformation algorithm that interprets a BNCE TGG TGG to transform a source model G into one of its correspondent target models T according to the correctly-transformed relation defined by TGG.

For that end, let $TGG = (\Sigma = \Sigma_s \cup \Sigma_t, \Delta, S, P)$ be a triple graph grammar defining the correctly-transformed relation between two arbitrary sets of graphs $\mathcal{S}$ over Σ_s and $\mathcal{T}$ over Σ_t. And let $G \in \mathcal{S}$ be a source graph. We want to find a target graph $T \in \mathcal{T}$ such that $G \leftarrow C \rightarrow T \in L(TGG)$. To put in words, we wish to find a triple graph holding G and T that is in the language of all correctly transformed models. Hence, the model transformation problem is reduced— according to the definition of triple graph language (see Definition 3.18)— to the problem of finding a derivation $Z_{TGG} \Rightarrow^*_{TGG} G \leftarrow C \rightarrow T$.

Our strategy to solve this problem is, first, to get a derivation for G with the source part of TGG and, then, construct the derivation $Z_{TGG} \Rightarrow^*_{TGG} G \leftarrow C \rightarrow T$. For this purpose, consider the definitions of the s and t functions, that extract the source and the target part of production rules.

Definition 4.1. Let $r = (A \rightarrow (G_s \leftarrow G_c \rightarrow G_t), \omega_s, \omega_t)$ be a production rule of a triple graph grammar, $\mathrm{s}(r) = (A \rightarrow G_s, \omega_s)$ gives the source part of r and $\mathrm{t}(r) = (A \rightarrow G_t, \omega_t)$ gives the target part. Moreover, $\mathrm{s}^{-1}((A \rightarrow G_s, \omega_s)) = r$ and $\mathrm{t}^{-1}((A \rightarrow G_t, \omega_t)) = r$ are the inverse of these functions.

In order for s^{-1} to be well defined, we require that all source parts $(A \rightarrow G_s, \omega_s)$ be unique. This does not affect the generality of the formalism, for right-hand side graphs G_s are still allowed to be isomorphic.

Definition 4.2. Let $TGG = (\Sigma, \Delta, S, P)$ be a triple graph grammar, $\mathrm{S}(TGG) = (\Sigma, \Delta, S, \mathrm{s}(P))$ gives the source grammar of TGG and $\mathrm{T}(TGG) = (\Sigma, \Delta, S, \mathrm{t}(P))$ gives the target grammar of TGG.

Furthermore, consider the definition of the non-terminal consistent (NTC) property of TGG, which assures that non-terminal vertices of the correspondent graph are connected to vertices with the same label in the source and target graphs.

Definition 4.3. A triple graph grammar $TGG = (\Sigma, \Delta, S, P)$ is *non-terminal consistent* (NTC) if and only if, for all rules $(A \to (G_s \overset{ms}{\leftarrow} G_c \overset{mt}{\rightarrow} G_t), \omega_s, \omega_t) \in P$, the following holds:

1. $\forall c \in V_{G_c}$. if $\phi_{G_c}(c) \in \Gamma$ then $\phi_{G_c}(c) = \phi_{G_s}(ms(c)) = \phi_{G_t}(mt(c))$ and
2. For the sets $N_s = \{v \mid \phi_{G_s}(v) \in \Gamma\}$ and $N_t = \{v \mid \phi_{G_t}(v) \in \Gamma\}$, the range-restricted functions $(ms \rhd N_s)$ and $(mt \rhd N_t)$ are bijective.

Finally, the following result gives us an equivalence between a derivation in TGG and a derivation in its source grammar $S(TGG)$, which allows us to construct our goal derivation of $G \leftarrow C \rightarrow T$ in TGG using the derivation of G in $S(TGG)$.

Theorem 4.1. *Let $TGG = (\Sigma, \Delta, S, P)$ be a NTC TGG and $k \geq 1$,*
$D = Z_{TGG} \overset{r_0,s_0,c_0,t_0}{\Rightarrow} G^1 \overset{r_1,s_1,c_1,t_1}{\Rightarrow} \dots \overset{r_{k-1},s_{k-1},c_{k-1},t_{k-1}}{\Rightarrow} G^k$ is a derivation in TGG if, and only if, $\overline{D} = Z_{S(TGG)} \overset{s(r_0),s_0}{\Rightarrow} G_s^1 \overset{s(r_1),s_1}{\Rightarrow} \dots \overset{s(r_{k-1}),s_{k-1}}{\Rightarrow} G_s^k$ is a derivation in $S(TGG)$.

Proof. We want to show that if D is a derivation in $TGG = (\Sigma, \Delta, S, P)$, then $\overline{D}$ is a derivation in $SG := S(TGG) = (\Sigma, \Delta, S, SP)$, and vice-versa. We prove it by induction in the following.

First, for the induction base, since, $Z_{TGG} \overset{r_0,s_0,c_0,t_0}{\Rightarrow}_{TGG} G^1$, then expanding Z_{TGG} and G^1, we have

$Z_s \leftarrow Z_c \rightarrow Z_t \overset{r_0,s_0,c_0,t_0}{\Rightarrow}_{TGG} G_s^1 \leftarrow G_c^1 \rightarrow G_t^1$, then, by Definition 3.16,
$r_0 = (S \to (R_s \leftarrow R_c \rightarrow R_t), \omega_s, \omega_t) \in P$ and, by Definition 4.1,
$\mathrm{s}(r_0) = (S \to R_s, \omega_s) \in SP$

Hence, using it plus the configuration of $\phi_{Z_s}(s_0)$, $V_{G_s^1}$, $E_{G_s^1}$ and $\phi_{G_s^1}$ and the equality $Z_s = Z_{SG}$, we have, by Definition 3.7, $Z_{SG} \overset{\mathrm{s}(r_0),s_0}{\Rightarrow}_{SG} G_s^1$.

In the other direction, we choose c_0, t_0 from the definition of Z_{TGG}, with $\phi_{Z_c}(c_0) = S$ and $\phi_{Z_t}(t_0) = S$. In this case, since,

$Z_{SG} \overset{s(r_0),s_0}{\Rightarrow}_{SG} G_s^1$, then by Definition 3.7,
$\mathrm{s}(r_0) = (S \to R_s, \omega_s) \in SP$ and, using the bijectivity of s, we get
$r_0 = \mathrm{s}^{-1}(s(r_0)) = (S \to (R_s \leftarrow R_c \rightarrow R_t), \omega_s, \omega_t) \in P$

Hence, using it plus the configuration of $\phi_{Z_{SG}}(s_0)$, $V_{G_s^1}$, $E_{G_s^1}$ and $\phi_{G_s^1}$, the equality $Z_s = Z_{SG}$ and constructing $V_{G_c^1}$, $V_{G_t^1}$, $E_{G_c^1}$, $E_{G_t^1}$, $\phi_{G_c^1}$, $\phi_{G_t^1}$ from Z_c and Z_t according to the Definition 3.16 $Z_{TGG} \overset{r_0,s_0,c_0,t_0}{\Rightarrow}_{TGG} G_s^1 \leftarrow G_c^1 \rightarrow G_t^1$.

Now, for the induction step, we want to show that if $Z_{TGG} \Rightarrow^*_{TGG} G^i \overset{r_i,s_i,c_i,t_i}{\Rightarrow}_{TGG} G^{i+1}$ is a derivation in TGG, then $Z_{SG} \Rightarrow^*_{SG} G_s^i \overset{s(r_i),s_i}{\Rightarrow}_{SG} G_s^{i+1}$ is a derivation in SG and vice-versa, provided that the equivalence holds for the first i steps, so we just have to show it for the step $i+1$.

So, since, $G^i \overset{r_i,s_i,c_i,t_i}{\Rightarrow}_{TGG} G^{i+1}$, that is

$G_s^i \overset{ms_i}{\leftarrow} G_c^i \overset{mt_i}{\rightarrow} G_t^i \overset{r_i,s_i,c_i,t_i}{\Rightarrow}_{TGG} G_s^{i+1} \leftarrow G_c^{i+1} \rightarrow G_t^{i+1}$, then, by Definition 3.16,
$r_i = (S \rightarrow (R_s \leftarrow R_c \rightarrow R_t), \omega_s, \omega_t) \in P$, and by Definition 4.1,
$\mathrm{s}(r_i) = (S \rightarrow R_s, \omega_s) \in SP$

Hence, using it plus the configuration of $\phi_{G_s^i}(s_i)$, $V_{G_s^{i+1}}$, $E_{G_s^{i+1}}$ and $\phi_{G_s^{i+1}}$, we have, by Definition 3.7, $G_s^i \overset{\mathrm{s}(r_i),s_i}{\Rightarrow}_{SG} G_s^{i+1}$.

In the other direction, we choose, using the bijectivity from the range restricted function s, stemming from the NTC property, $c_i = ms_i^{-1}(s_i), t_i = mt_i(c_i)$. Moreover, since TGG is NTC, and because, by induction hypothesis, $Z_{TGG} \Rightarrow^*_{TGG} G^i$ is a derivation in TGG and $\phi_{G_s^i}(s_i) \in \Gamma$, it is clear that $\phi_{G_s^i}(s_i) = \phi_{G_c^i}(c_i) = \phi_{G_t^i}(t_i)$.

In this case, since

$G_s^i \overset{s(r_i),s_i}{\Rightarrow}_{SG} G_s^{i+1}$, then, by Definition 3.7,
$\mathrm{s}(r_i) = (A \rightarrow R_s, \omega_s) \in SP$ and, using the bijectivity of s, we get
$r_i = \mathrm{s}^{-1}(s(r_i)) = (A \rightarrow (R_s \leftarrow R_c \rightarrow R_t), \omega_s, \omega_t) \in P$

Hence, using, additionally, the configuration of $\phi_{G_s^i}(s_i)$, $\phi_{G_c^i}(c_i)$, $\phi_{G_t^i}(t_i)$, $V_{G_s^{i+1}}$, $E_{G_s^{i+1}}$ and $\phi_{G_s^{i+1}}$ and constructing $V_{G_c^{i+1}}$, $V_{G_t^{i+1}}$, $E_{G_c^{i+1}}$, $E_{G_t^{i+1}}$, $\phi_{G_c^{i+1}}$, $\phi_{G_t^{i+1}}$ from G_c^i and G_t^i according to the Definition 3.16, we have

$$G_s^i \leftarrow G_c^i \rightarrow G_t^i \overset{r_i,s_i,c_i,t_i}{\Rightarrow}_{TGG} G_s^{i+1} \leftarrow G_c^{i+1} \rightarrow G_t^{i+1}$$

This finishes the proof. □ □

Therefore, by Theorem 4.1, the problem of finding a derivation $D = Z_{TGG} \Rightarrow^*_{TGG} G \leftarrow C \rightarrow T$ is reduced to finding a derivation $\overline{D} = Z_{S(TGG)} \Rightarrow_{S(TGG)} G$, what can be done with the already presented parsing algorithm 3.1. The final construction of the triple graph $G \leftarrow C \rightarrow T$ becomes then just a matter of creating D out of $\overline{D}$.

The complete transformation procedure is presented in the Algorithm 4.2. Thereby it is required that the TGG be neighborhood preserving (NP), what poses no problem to our procedure, once any TGG can be transformed into the neighborhood preserving normal form. Notice that this algorithm always terminates. A more elaborated discussion on its complexity is given in Section 8.2.

Algorithm 4.2 Transformation Algorithm for NP NTC BNCE TGG

```
Require: $TGG$ is a valid NP NTC BNCE triple graph grammar
Require: $G$ is a valid graph over $\Sigma$
    function $transform(TGG = (\Sigma, \Delta, S, P), G = (V_G, E_G, \phi_G))$: $Graph$
        $SG \leftarrow S(TGG)$                                          ▷ see 4.1
        $\overline{D} \leftarrow parse(SG, G)$                          ▷ use algorithm 3.1
        if $\overline{D} = Z_{SG} \Rightarrow^*_{SG} G$ then            ▷ if parsed successfully
            from $\overline{D}$ construct $D = Z_{TGG} \Rightarrow^*_{TGG} G \leftarrow C \rightarrow T$
            return Just $T$
        else
            return Nothing              ▷ no $T$ satisfies $(G \leftarrow C \rightarrow T) \in L(TGG)$
        end if
    end function
Ensure: $return$ is either Nothing or Just $T$, such that $(G \leftarrow C \rightarrow T) \in$
    $L(TGG)$
```

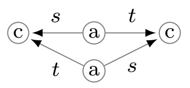
Our transformation method is robust enough to support both ambiguous grammars, because the parser supports it, and non-functional transformations. A transformation specified by the triple graph grammar TGG is non-functional if, and only if, for at least one source graph G there is more than one target graph T, such that $G \leftarrow C \rightarrow T \in L(TGG)$. This can happen when the source part's right-hand side of two rules are isomorphic but their target parts not. That is, a evidence for the transformation to be non-functional is the existence of two rules r and q in the set of rules of TGG with $\mathrm{s}(r) = (A \rightarrow R_r, \omega_r)$, $\mathrm{s}(q) = (B \rightarrow R_q, \omega_q)$ and $R_r \cong R_q$. In this case, the output for the graph G is non-deterministic, because any of the rules r or q may be chosen by the parser at the construction of the parsing tree for the G.

We claim, moreover, that our transformation method is sound and complete in the TGG sense. The former shall hold, because all the set of triple graphs generated by our algorithm is contained in $L(TGG)$ for every NP NTC BNCE TGG. The latter shall hold, because $L(TGG)$ is contained by the set of all triple graphs generated by our algorithm for every NP NTC BNCE TGG. Hence, we claim that our transformation method is correct in the TGG sense.

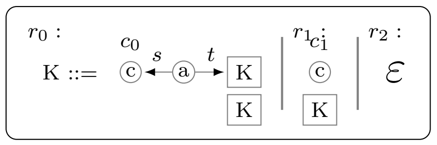
5. An Extension of NCE Triple Graph Grammars with Application Conditions

The NCE graph grammar formalism from [JR82], presented in the previous sections, can define with very few rules the languages of several classes of labeled graphs, including trees, path graphs, star graphs, control-flow graphs, edgeless graphs, complete graphs, and others. However, it is at least difficult to define the languages of other classes, like the class-diagram graphs, with NCE graph grammars. In this Section, we approach the problem of defining a NCE graph grammar for these classes of graphs and propose a solution for that by means of an extension of NCE that includes application conditions.

Class diagrams are commonly used to model object-oriented software artifact that are composed of several classes related by associations. For the sake of demonstrating the problem of NCE with class diagrams, consider a simplified view of the class-diagrams graphs, in which a vertex has either label c or a, respectively representing a class or an association, and an edge between an association and a class with label s (t) signalizes that the class is the source (target) of the association. In Figure 5.1a, a class-diagram graph with two classes connected by two associations is depicted. An attempt for a NCE graph grammar that would describe the language of all class-diagram graphs is $GG = (\{K, a, c, s, t\}, \{a, c, s, t\}, K, \{r_0, r_1, r_2\})$, with r_0, r_1, and r_2 depicted in Figure 5.1b and $\omega_0(c_0) = \omega_1(c_1) = \{(t, a)\}$ and $\omega_0 = \emptyset$ being the complete embedding definition of the rules r_0, r_1, and r_2, respectively.



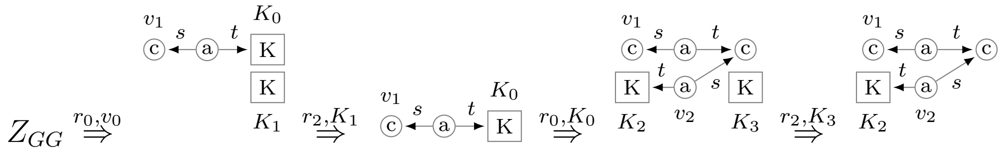
(a) A class-diagram graph



(b) The depiction of the rules of GG

Figure 5.1: An example for a class-diagram graph with two classes connected by associations in (a) and the rules r_0, r_1, and r_2 of the graph grammar GG in (b)

The problem of the graph grammar GG is that it does not define the complete language of the class-diagram graph. In fact, the graph in Figure 5.1a is not in $L(GG)$. To see this, consider the following derivation in GG



This is the closest we get to deriving the graph in Figure 5.1a using GG. Thereby, we would like to connect the association v_2 to the class v_1 but it is not possible, because v_1 was not a neighbor of the vertex K_0 that preceded v_2. Notice that a vertex in any sentential form can only be either connected to vertices that stem from the same rule application or to neighbors of its precedent vertex. In fact, this seems to be a general characteristic for context-free grammars, where the information about elements in the context of the precedents are not available for descendant elements. In order to overcome it, one could potentially elaborate an alternative grammar that defines the desired language completely and concisely, but we believe that such ad-hoc solution would include a bigger number of rules and add complexity to the grammar. With that in mind, we propose in the sequel an extension of the NCE grammar formalism with positive application conditions (PAC) that solves this issue.

In NCE graph grammars with PAC, rules' right-hand sides are equipped with application conditions in form of special vertices that are produced by derivation steps and removed by so-called resolution steps. A resolution step is responsible for removing such special vertices and moving their adjacent edges to other vertices. This resolution mechanism allows that the vertex v_2 from the previous example be connected to v_1.

In order to define the PAC mechanism in detail, the definitions of rule and derivation step are augmented as follows.

Definition 5.1. A *rule with PAC* is of the form $(A \to R, \omega, U)$ with A, R and ω as described in 3.5 and $U \subseteq \{v \in V_R \mid \phi_R(v) \in \Delta\}$, the set of special vertices, called PAC vertices.

If a graph grammar has at least one rule with PAC, then we say it is a graph grammar with PAC.

Definition 5.2. A *concrete derivation step with PAC* in the graph grammar GG is of the form $G \overset{r,v,U}{\Rightarrow}_{GG} H$ with G, H, v being as described in Definition 3.7, and $r = (A \to R, \omega, U)$ being a production rule with PAC. Given that, a *derivation step with PAC* is, analogously, of the form $G \overset{r,v,W}{\Rightarrow}_{GG} H'$ with $W = m(U)$ where m is the isomorphism from H and H'.

So far, PAC vertices do not change anything in the behavior of a derivation step and the set U in a derivation step serves just to tag which vertices are PAC in a sentential form. Nevertheless, PAC vertices play an important role on a resolution step, defined below. If W is empty, we might omit it from the notation.

Definition 5.3. Let $GG = (\Sigma, \Delta, S, P)$ be a graph grammar and G a graph over Δ, G *resolves into* H with the resolution partial function $\rho : V_G \nrightarrow V_G$, we write $G \overset{\rho}{\rightarrowtail} H$ and call it a *resolution step*, if, and only if, the following holds:

$$\begin{aligned}
&\forall v \in \operatorname{dom}\rho.\ \rho(v) \notin \operatorname{dom}\rho \text{ and } \phi_G(\rho(v)) = \phi_G(v) \text{ and}\\
V_H &= V_G \setminus \operatorname{dom}\rho \text{ and}\\
E_H &= (E_G \setminus (\{(u,l,t) \mid u \in \operatorname{dom}\rho, (u,l,t) \in E_G\}\\
&\qquad\qquad \cup \{(t,l,u) \mid u \in \operatorname{dom}\rho, (t,l,u) \in E_G\}))\\
&\quad \cup \{(\rho(u),l,t) \mid u \in \operatorname{dom}\rho, (u,l,t) \in E_G\}\\
&\quad \cup \{(t,l,\rho(u)) \mid u \in \operatorname{dom}\rho, (t,l,u) \in E_G\}
\end{aligned}$$

A resolution step can be informally understood as the removal of the PAC vertices of G— that are in the domain of the resolution function ρ —followed by the redirection of the edges adjacent to the PAC vertices to other vertices of H.

For the PAC mechanism to work, it is still necessary to combine derivation and resolution steps to define the language of a grammar with PAC, what we do in the following.

Definition 5.4. A *production* Q in a graph grammar with PAC is a sequence of n derivation steps followed by n resolution steps with $n > 0$, as follows:

$$Q = (G_0 \overset{r_0,v_0,W_0}{\Rightarrow} G_1 \overset{r_1,v_1,W_1}{\Rightarrow} \dots \overset{r_{n-1},v_{n-1},W_{n-1}}{\Rightarrow} G_n^0 \overset{\rho_0}{\rightarrowtail} G_n^1 \overset{\rho_1}{\rightarrowtail} \dots \overset{\rho_{n-1}}{\rightarrowtail} G_n^n)$$

with ρ_i being a resolution total function $\rho_i : m_i(W_i) \rightarrow V_{G_n^i}$ and $m_i : W_i \rightarrow V_{G_n^i}$ the mapping from the PAC vertices generated on the derivation step i to their correspondent vertices in G_n^i, for all $0 \leq i < n$.

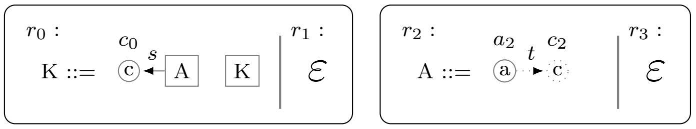
It is clear that, the mapping m of the previous definition exists and is bijective because all PAC vertices are, by definition, terminal and, therefore, are not deleted by derivation steps and, moreover, the images of all m_i are pair-wise disjunct.

Definition 5.5. The *language* $L(GG)$ generated by the grammar GG with PAC is

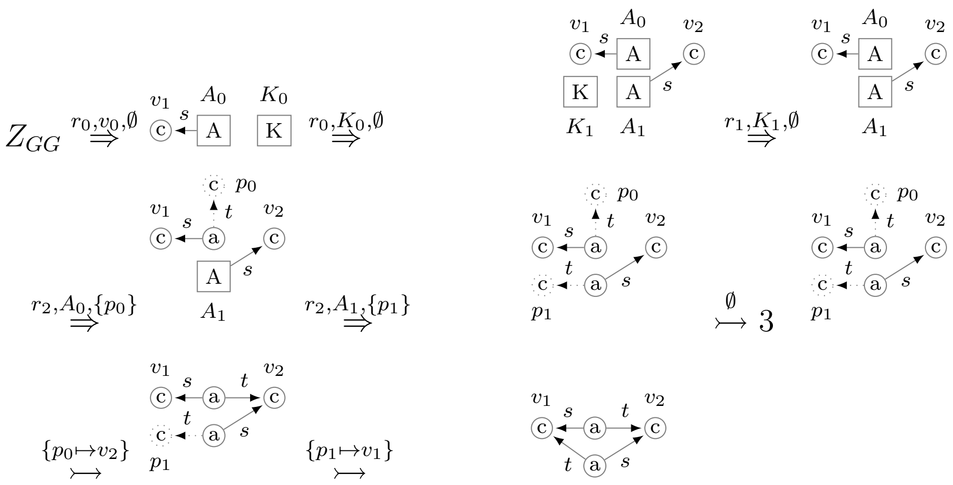
$$L(GG) = \{H \text{ is a graph over } \Delta \text{ and } Z_{GG} \Rightarrow^n H' \rightarrowtail^n H\}$$

where $\Rightarrow^n$ and $\rightarrowtail^n$ denote a sequence of n derivation steps and n resolution steps, respectively.

Ultimately, we put forward a NCE graph grammar with PAC whose language is the set of all class-diagram graphs. This grammar is $GG = (\{K, A, a, c, s, t\}, \{a, c, s, t\}, K, \{r_0, r_1, r_2, r_3\})$ with $\omega_0(c_0) = \{(t, a)\}$, $\omega_2(a_2) = \{(s, c)\}$, $\omega_2(c_2) = \{(s, a), (t, a)\}$, $\omega_1 = \omega_3 = \emptyset$ being the complete characterization of the embedding functions of the respective rules, and the the rules being denoted as below. We advise that PAC vertices and their adjacent edges are depicted with dotted lines.



Below, we demonstrate that the graph from Figure 5.1a is in $L(GG)$, by means of a production in GG.



In this production, the rule r_2 creates, through two applications, two PAC vertices p_0 and p_1, which are then removed and have their adjacent vertices moved to the vertices v_2 and v_1, respectively, through the last two resolution steps. The resolution steps have thus the power of connecting vertices that could not be connected otherwise.

Remark 5.1. If, for all rules $(A \to R, \omega, U)$ in a grammar GG have $U = \emptyset$, then the GG degrades to a normal NCE grammar without PAC and the resolution steps have no effect in $L(GG)$.

Remark 5.2. Given a graph grammar GG with PAC, if the graph g is in $L(GG)$, then g has no PAC vertices. That is, the resolution steps remove all PAC vertices, because every resolution function ρ_i is required to map to vertices that are not in its domain. This guarantees that the number of PAC vertices reduces at each resolution step with $\rho_i \neq \emptyset$.

5.1 Parsing of Graphs with Application Conditions

Regarding the parsing procedure, the Algorithm 3.1 can be slightly modified to support PAC, by augmenting the zone vertices with PAC vertices, changing the way how zone graphs are constructed and how zone vertices are added to *bup* and to the parsing forest. The details of these changes are described in the sequel.

Definition 5.6. A *zone vertex with PAC* of a graph G is a triple (σ, U, W), with σ and U being as explained in Definition 3.20 and $W \in V_G$ being the set of PAC vertices disjunct from U.

Definition 5.7. Let $H = \{(\sigma_o, U_0, W_0), \ldots, (\sigma_m, U_m, W_m)\})$ be a set of zone vertices with PAC of a graph G, as given in Definition 3.21, and $W(H) = \bigcup_{0 \le i \le m} W_i$. A *zone graph with PAC* $Z(H)$ for H is (V, E, ψ), with

$$\begin{aligned}
&V = H \cup \{(\psi_G(x), \{x\}, \emptyset) \mid x \in \text{neigh}_G(V(H)) \setminus W(H)\} \\
&E = \{((\sigma, U, W), l, (\eta, T, X)) \mid (\sigma, U, W), (\eta, T, X) \in V \text{ and } U \neq T \text{ and} \\
&\quad (u, l, t) \in E_G \text{ and } u \in U \setminus X \text{ and } t \in T \setminus W\} \\
&\phi = \{(\sigma, U, W) \mapsto \sigma \mid (\sigma, U, W) \in V\}
\end{aligned}$$

The Algorithm 5.3 is a parsing method that returns a valid derivation if, and only if, the input graph G is in the language of the graph grammar GG with PAC. Notice that, this procedure does not return a derivation, but a production, that is built by the function Q that performs, analogously to D in Algorithm 3.1, a depth-first walk in the parsing tree.

The most important difference between Algorithm 3.1 and 5.3 are, first, in the use of a derivation with PAC $Z(\{l\}) \overset{r,l,W}{\Rightarrow} Z(h)$ in line 9, where W is the set of PAC vertices U from the rule r mapped to the zone graph $Z(h)$, and, second, in the construction of the zone vertex l, that is augmented with the PAC vertices $W(h)$ which are, in turn, removed from the normal vertices of l, in line 8. In practice, this allows, on the one hand, that PAC vertices participate in the search for rules that produce the desired zone graphs, and, on the other hand, that they be not included in the set of normal vertices of zone vertices so they can be effectively added to the zone vertices that effectively produce them.

5.2 Model Transformation with Application Condition

In regard to the application of NCE graph grammars with PAC to the problem of model transformation, the extension is also possible, as shown in the next argumentation. First, consider the extension of NCE TGG to support PAC.

Algorithm 5.3 Parsing Algorithm for NP BNCE Graph Grammars with PAC

```
Require: $GG$ is a valid NP BNCE graph grammar with PAC
Require: $G$ is a valid graph over $\Delta$
 1: function $parse(GG = (\Sigma, \Delta, S, P), G = (V_G, E_G, \phi_G))$: $Derivation$
 2:     $bup \leftarrow \{(\phi_G(x), \{x\}, \emptyset) \mid x \in V_G\}$
 3:     $pf \leftarrow \{(b \rightrightarrows \emptyset) \mid b \in bup\}$
 4:     repeat
 5:         $h \leftarrow$ select$\{X \subseteq bup \mid$ for all $U_i, U_j \in X$ with $i \neq j.\ U_i \cap U_j = \emptyset\}$
 6:         for all $d \in \Gamma$ do
 7:             $r \leftarrow$ any $\{(d \rightarrow R, \omega, U) \in P \mid R \cong Y(h)\}$
 8:             $l \leftarrow (d, V(h) \setminus W(h), W(h))$          ▷ $l$ is augmented with PAC $W(h)$
 9:             if $Z(\{l\}) \overset{r,l,W}{\Rrightarrow} Z(h)$ then          ▷ derivation with PAC is possible
10:                 $bup \leftarrow bup \cup \{l\}$
11:                 $pf \leftarrow pf \cup \{(l^r \rightrightarrows \{(z^y \rightrightarrows X) \mid (z^y \rightrightarrows X) \in pf, z \in h\})\}$
12:             end if
13:         end for
14:     until $(S, V_G, \_) \in bup$          ▷ if found the root, no matter which PAC
15:     return $(S, V_G, \_) \in bup$ ? Just $Q(((S, V_G, \_)^y \rightrightarrows X) \in pf)$ : Nothing
16: end function
Ensure: $return$ is either Nothing or of the form Just $Z_{GG} \Rightarrow^* G \rightarrowtail^*$
```

Definition 5.8. A *triple rule with PAC* is of the form $(A \rightarrow (R_s \leftarrow R_c \rightarrow R_t), \omega_s, \omega_t, U_s, U_t)$ with $A \rightarrow (R_s \leftarrow R_c \rightarrow R_t)$, ω_s, and ω_t as defined in Definition 3.14 and $U_s \subseteq \{v \in V_{R_s} \mid \phi_{R_s}(v) \in \Delta\}$ the set of PAC vertices of R_s and $U_t \subseteq \{v \in V_{R_t} \mid \phi_{R_t}(v) \in \Delta\}$ the set of PAC vertices of R_t.

Analogously, the concepts of *concrete derivation step* and *derivation step* for TGG are extended to be as follows.

Definition 5.9. A *concrete derivation step with PAC* in the triple graph grammar TGG is of the form $G \overset{r,v_s,v_c,v_t,U_s,U_t}{\Rrightarrow}_{TGG} H$, where G, H, v_s, v_c, v_t being as described in Definition 3.16, $r = (A \rightarrow (R_s \leftarrow R_c \rightarrow R_t), \omega_s, \omega_t, U_s, U_t)$ being a triple rule with PAC. Given that, a *derivation step with PAC* is, analogously, of the form $G \overset{r,v_s,v_c,v_t,W_s,W_t}{\Rightarrow}_{TGG} H'$ with $W_s = m(U_s)$ and $W_t = m(U_t)$ where m is the triple isomorphism from H to H'.

If W_s and W_t are empty, we might omit them from the notation. The *resolution step* for TGG is as follows.

Definition 5.10. Let $TGG = (\Sigma, \Delta, S, P)$ be a triple graph grammar and $G = (G_s \overset{gs}{\leftarrow} G_c \overset{gt}{\rightarrow} G_t)$ a triple graph over Δ, G *resolves into* $H = (H_s \overset{hs}{\leftarrow} H_c \overset{ht}{\rightarrow} H_t)$ with the resolution partial functions $\rho_s : V_{G_s} \nrightarrow V_{G_s}$ and $\rho_t : V_{G_t} \nrightarrow V_{G_t}$, we write

$G \overset{\rho_s,\rho_t}{\rightarrowtail} H$ and call it a *resolution step*, if, and only if, the following holds:

$$
\begin{aligned}
&\quad \forall v \in \operatorname{dom}\rho_s.\ \rho_s(v) \notin \operatorname{dom}\rho_s \text{ and } \phi_{G_s}(\rho_s(v)) = \phi_{G_s}(v) \text{ and}\\
&\quad \forall v \in \operatorname{dom}\rho_t.\ \rho_t(v) \notin \operatorname{dom}\rho_t \text{ and } \phi_{G_t}(\rho_t(v)) = \phi_{G_t}(v) \text{ and}\\
V_{H_s} &= V_{G_s} \setminus \operatorname{dom}\rho_s \text{ and}\\
V_{H_c} &= V_{G_c} \setminus gs^{-1}(\operatorname{dom}\rho_s) \text{ and}\\
V_{H_t} &= V_{G_t} \setminus \operatorname{dom}\rho_t \text{ and}\\
E_{H_s} &= (E_{G_s} \setminus (\{(u,l,t) \mid u \in \operatorname{dom}\rho_s, (u,l,t) \in E_{G_s}\}\\
&\qquad\qquad \cup \{(t,l,u) \mid u \in \operatorname{dom}\rho_s, (t,l,u) \in E_{G_s}\}))\\
&\quad \cup \{(\rho_s(u),l,t) \mid u \in \operatorname{dom}\rho_s, (u,l,t) \in E_{G_s}\}\\
&\quad \cup \{(t,l,\rho_s(u)) \mid u \in \operatorname{dom}\rho_s, (t,l,u) \in E_{G_s}\}\\
E_{H_c} &= E_{G_c} \setminus (\{(u,l,t) \mid u \in gs^{-1}(\operatorname{dom}\rho_s), (u,l,t) \in E_{G_c}\}\\
&\qquad\qquad \cup \{(t,l,u) \mid u \in gs^{-1}(\operatorname{dom}\rho_s), (t,l,u) \in E_{G_c}\})\\
E_{H_t} &= (E_{G_t} \setminus (\{(u,l,t) \mid u \in \operatorname{dom}\rho_t, (u,l,t) \in E_{G_t}\}\\
&\qquad\qquad \cup \{(t,l,u) \mid u \in \operatorname{dom}\rho_t, (t,l,u) \in E_{G_t}\}))\\
&\quad \cup \{(\rho_t(u),l,t) \mid u \in \operatorname{dom}\rho_t, (u,l,t) \in E_{G_t}\}\\
&\quad \cup \{(t,l,\rho_t(u)) \mid u \in \operatorname{dom}\rho_t, (t,l,u) \in E_{G_t}\}
\end{aligned}
$$

Finally, production and language for TGG are extended as follows.

Definition 5.11. A *production* Q in a TGG with PAC is a sequence of n derivation steps followed by n resolution steps with $n > 0$, as follows:

$$Q = (G_0 \overset{r_0,s_0,c_0,t_0,W_0,T_0}{\Rightarrow} G_1 \overset{r_1,s_1,c_1,t_1,W_1,T_1}{\Rightarrow} \dots \overset{r_{n-1},s_{n-1},c_{n-1},t_{n-1},W_{n-1},T_{n-1}}{\Rightarrow} G_n^0 \overset{\rho_0,\tau_0}{\rightarrowtail} G_n^1 \overset{\rho_1,\tau_1}{\rightarrowtail} \dots \overset{\rho_{n-1},\tau_{n-1}}{\rightarrowtail} G_n^n)$$

with $\rho_i : m_i(W_i) \to V_{G^i_{s,n}}$ and $\tau_i : n_i(T_i) \to V_{G^i_{t,n}}$ being the resolution total functions and $m_i : W_i \to V_{G^i_{s,n}}$ and $n_i : T_i \to V_{G^i_{t,n}}$ the mappings from the PAC vertices generated on the derivation step i to their correspondent vertices in the source and target graphs of the triple graph G^i_n, for all $0 \leq i < n$.

Definition 5.12. The *language* $L(TGG)$ generated by the triple grammar TGG with PAC is

$$L(TGG) = \{H \text{ is a triple graph over } \Delta \text{ and } Z_{TGG} \Rightarrow^n H' \rightarrowtail^n H\}$$

where $\Rightarrow^n$ and $\rightarrowtail^n$ denote a sequence of n derivation steps and n resolution steps, respectively.

To define the model transformation procedure, consider the redefinition of the s function.

Definition 5.13. Let $r = (A \to (G_s \leftarrow G_c \to G_t), \omega_s, \omega_t, U_s, U_t)$ be a production rule of a triple graph grammar, redefine $\mathrm{s}(r)$ as $\mathrm{s}(r) = (A \to G_s, \omega_s, U_s)$ and $\mathrm{s}^{-1}((A \to G_s, \omega_s, U_s)) = r$.

Furthermore, consider the definition of the PAC consistent (PC) property of TGG, which assures that a PAC vertex of the source graph is connected with a PAC vertex in the correspondence and in the target graphs.

Definition 5.14. Let $TGG = (\Sigma, \Delta, S, P)$ be a triple graph grammar and $\Pi = \bigcup_{r \in P} \phi_{G_s}(U_s)$ be the set of all PAC labels, TGG is *PAC consistent* (PC) if, and only if, for all rules $(A \to (G_s \overset{ms}{\leftarrow} G_c \overset{mt}{\rightarrow} G_t), \omega_s, \omega_t, U_s, U_t) \in P$, the following holds

1. $\forall v \in U_s.\ mt(ms^{-1}(v)) \in U_t$
2. $\forall v \in G_s.$ if $\phi_{G_s}(v) \in \Pi$ then $mt(ms^{-1}(v)) \in G_t$

Finally, Theorem 5.1 is, analogously to Theorem 4.1, allows us to construct a production in TGG out of a production in $S(TGG)$, for a triple graph grammar TGG.

Theorem 5.1. *Let $TGG = (\Sigma, \Delta, S, P)$ be a NTC PC TGG and $k \geq 1$,*

$$Q = Z_{TGG} \overset{r_0,s_0,c_0,t_0,W_0,Y_0}{\Rightarrow} G^1 \overset{r_1,s_1,c_1,t_1,W_1,Y_1}{\Rightarrow} \dots \overset{r_{k-1},s_{k-1},c_{k-1},t_{k-1},W_{k-1},Y_{k-1}}{\Rightarrow} G^k \overset{\rho_0,\tau_0}{\rightarrowtail} H^1 \overset{\rho_1,\tau_1}{\rightarrowtail} \dots \overset{\rho_{k-1},\tau_{k-1}}{\rightarrowtail} H^k$$

is a production in TGG if, and only if,

$$\overline{Q} = Z_{S(TGG)} \overset{s(r_0),s_0,W_0}{\Rightarrow} G_s^1 \overset{s(r_1),s_1,W_1}{\Rightarrow} \dots \overset{s(r_{k-1}),s_{k-1},W_{k-1}}{\Rightarrow} G_s^k \overset{\rho_0}{\rightarrowtail} H_s^1 \overset{\rho_1}{\rightarrowtail} \dots \overset{\rho_{k-1}}{\rightarrowtail} H_s^k$$

is a production in $S(TGG)$.

Proof. We want to show that if Q is a production in $TGG = (\Sigma, \Delta, S, P)$, then $\overline{Q}$ is a production in $SG := S(TGG) = (\Sigma, \Delta, S, SP)$, and vice-versa.

For the first half of the production, that is, for the k derivations steps the equivalence holds trivially because the PAC vertices W_i and Y_i do not harm the result of Theorem 4.1. So we just have to show it for the second half, that is, the k resolutions, what we do by induction in the following.

The induction base is trivial, because the start graph Z_{TGG} has no PAC vertices, hence, $W_0 = Y_0 = \emptyset$ and thus $\rho_0 = \tau_0 = \emptyset$. Therefore, if $G^k \overset{\rho_0,\tau_0}{\rightarrowtail} H^1$, then $G_s^k \overset{\rho_0}{\rightarrowtail} H_s^1$, and vice-versa.

For the induction step, we want to show that if $G^k \rightarrowtail^* H^i \overset{\rho_i,\tau_i}{\rightarrowtail} H^{i+1}$ is a resolution, then $G_s^k \rightarrowtail^* H_s^i \overset{\rho_i}{\rightarrowtail} H_s^{i+1}$ is also a resolution and vice-versa, provided that the equivalence holds for the first i steps, so we just have to show it for the step $i+1$.

So, in the one direction, if $H^i \overset{\rho_i,\tau_i}{\rightarrowtail} H^{i+1}$, then, trivially, $H_s^i \overset{\rho_i}{\rightarrowtail} H_s^{i+1}$, by Definition 5.3 and 5.10.

In the other direction, we want to show that if $H_s^i \overset{\rho_i}{\rightarrowtail} H_s^{i+1}$ then $(H_s^i \overset{ms}{\leftarrow} H_c^i \overset{mt}{\rightarrow} H_t^i) \overset{\rho_i,\tau_i}{\rightarrowtail} (H_s^{i+1} \leftarrow H_c^{i+1} \rightarrow H_t^{i+1})$. For that regard, we set

$$\tau_i = \{mt(ms^{-1}(v)) \mapsto mt(ms^{-1}(\rho_i(v))) \mid v \in \operatorname{dom} \rho_i\}$$

Because TGG is PC, it holds that $mt(ms^{-1}(v)) \in H_t^i$ and $mt(ms^{-1}(\rho_i(v))) \in H_t^i$ for all v in $\mathrm{dom}\,\rho_i$, thus τ_i is well defined. Moreover, the induction hypothesis supports that

$$\forall v \in \mathrm{dom}\,\rho_i.\ \rho_i(v) \notin \mathrm{dom}\,\rho_i \text{ and } \phi_{H_s^i}(\rho_i(v)) = \phi_{H_s^i}(v)$$

Hence, by mt and ms bijectivity and by the PC property, it is clear that

$$\forall v \in \mathrm{dom}\,\tau_i.\ \tau_i(v) \notin \mathrm{dom}\,\tau_i \text{ and } \phi_{H_t^i}(\tau_i(v)) = \phi_{H_t^i}(v)$$

Thus, choosing V_{H_t} and E_{H_t} according to Definition 5.10 and τ_i, we have that if $H_s^i \overset{\rho_i}{\rightarrowtail} H_s^{i+1}$ then $(H_s^i \overset{ms}{\leftarrow} H_c^i \overset{mt}{\rightarrow} H_t^i) \overset{\rho_i,\tau_i}{\rightarrowtail} (H_s^{i+1} \leftarrow H_c^{i+1} \rightarrow H_t^{i+1})$.

This finishes the proof. $\square$

The effective transformation procedure for TGG with PAC is essentially the same as the one in Algorithm 4.2, with the additional requirement of TGG being PC and the use of Theorem 5.1 to derive and resolve PAC vertices for the produced triple graph.

6. Implementation

In this chapter, we present in details our implementation for the model transformer that we exposed in the previous chapters. As programming language and runtime platform we use Java. As modeling and code generation tool we use Eclipse Modeling Framework (EMF).

The model transformer procedure is depicted in Figure 6.1. The input for the procedure consists of a source model, which is an instance of the source metamodel, and a TGG model, which is an instance of the TGG metamodel. The source model represents the model to be transformed into a target model and the TGG model holds the BNCE triple graph grammar that describes the transformation between the source metamodel and a target metamodel. The output model is an instance of the TG metamodel and holds the triple graph generated by the transformation procedure.

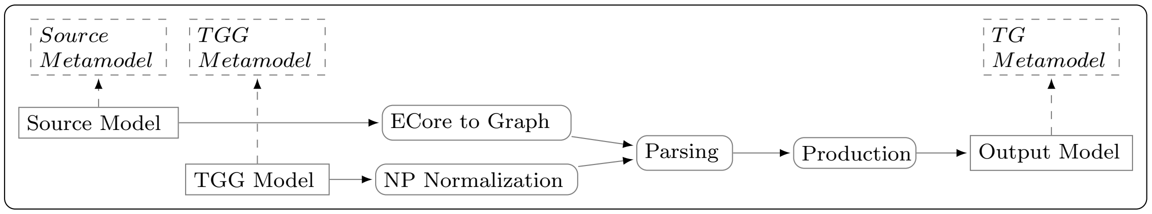


Figure 6.1: Implementation scheme for the model transformer presented by this thesis

Ecore to Graph. The source model is transformed into a graph by the step *Ecore to Graph*. This can be done trivially, for it is a one-to-one transformation, where each element from the model is transformed into a vertex of the graph. It suffices, thus, to trespass the source model element by element, starting from the roots up to the elements whose all children have already been trespassed. At each visited element, a vertex and an edge to each of its children is created.

NP Normalization. The TGG model is normalized to fit the neighborhood preserving (NP) normal form by the step *NP Normalization*. This normalization consists of creating an triple graph grammar TGG' equivalent to TGG, i.e. $L(TGG') = L(TGG)$, for which the NP property (see Definition 3.19) holds. An NP Normalization algorithm for NLC graph grammars can be found in [RW86]. We adapt this algorithm for NCE graph grammars and use it to normalize the source part of the TGG model (see Definition 4.1).

The NP Normalization starts by looking for non-NP rules. For each of these rules, it modifies its left-hand side so that it becomes NP. Moreover, it also adds new rules, that are produced by replacing each occurrence of the old left-hand side by the new one. This procedure is then repeated until the grammar is not modified anymore. It is guaranteed that it always stops producing a NP NCE graph grammar. After stopping, the NP Normalization also modifies the TGG model adding and modifying the correspondent triple rules whose source parts were modified in the process.

Parsing. The result of the steps *Ecore to Graph* and *NP Normalization*, that are the source graph to be transformed and a normalized TGG, are used by the step *Parsing* to produce a valid derivation for the source graph, in case it can be transformed following the rules in the TGG. Section 3.3 already offers an abstract presentation of the parsing algorithm for BNCE graph grammar. Thus, in the following paragraphs, we explore more concrete issues that come along with the implementation of the parser.

The parsing procedure can be seen as a search algorithm that explores systematically the search space of all parsing trees for the TGG until it finds the parsing tree for the input graph. It is easy to see that such search space is huge (and potentially infinite) for any practical TGG. The parser starts from the trivial parsing trees, each one containing only one zone vertex of the source graph (see Line 3 in Algorithm 5.3). Then, at every time that it finds a new derivation (see Line 9), it augments its search space with a new parsing tree for the just found derivation (see Line 11). Additionally, the parser also holds the so-called *bup* set with the zone vertices found by derivations assembled from other zone vertices in *bup*, which also grows at each time a derivation is found (see Line 10).

Notice, thus, that the direction to where the search space grows depends on the choice of which subset of zone vertices are picked from *bup* (see Line 5) as a handle to find new derivations and that the number of possibilities for such choice explodes as the size of *bup* grows. Indeed, the function that describes this growth, despite its importance in the complexity analysis of the parsing algorithm, is not known, as far as we know, but it is a polynomial on the size of the source graph [RW86, p. 160].

Therefore, we implemented three different strategies to query the *bup* set: The *naive*, the *greedy* and the *greedy aware*. In the *naive* strategy, the selection of subsets from *bup* are performed from the smaller to the bigger ones, without any other criterion. In the *greedy* strategy, subsets containing zone vertices added after the initialization get higher priority. Moreover, from those subsets, the ones with the greater amount of vertices (i.e. the greatest) get an even higher priority and are served first. Finally, the *greedy aware* strategy extends the *greedy* strategy, by using information about the graph being parsed and the grammar being utilized.

In this strategy, beyond the size criterion of the zone vertices, subsets that contain more zone vertices closer to the biggest zone vertex in it are ranked better. More specifically, the selector takes the biggest zone vertex and sums the approximate distance from the other zone vertices to it. The lesser this sum is, the better the subset is ranked to be queried. The approximate distance between two zone vertices z and y is k, if the least undirected path from any vertex of z to any vertex of y is k and $k \leq 2$; otherwise it is 4.

Beyond the distance criterion, the *greedy aware* strategy also uses the depth information of each vertex to decide on the priority of subsets that tied in the previous criteria. That is, subsets with zone vertices that contain deeper vertices are prioritized. The depth of a vertex v, in this case, is the length of the first path that got to v in the *Ecore to Graph* transformation. And, finally, this strategy does not had over subsets with k zone vertices, where the source grammar has no rule with a $k-$sized right-hand side.

In general, the use of a *greedy* strategy entails the growth of the search space in the direction of greater parsing trees because of the size criterion, what potentially makes it get to the final parsing tree in fewer steps. A *greedy aware* strategy, moreover, explores locality through the assumption that a derivation is more likely to be found with a handle containing nearer zone vertices, reducing the amount of effort with subsets that do not generate new zone vertices. Such assumption is supported by the fact that rules' right-hand sides tend to be connected in practical TGG. Lastly, it prioritizes deeper zone vertices following the observation that often grammars are built in such a way that deeper vertices (i.e. vertices that are further from the root vertex) tend to occur deeper in parsing tress too. In such case, specially for the beginning of the parsing, a deeper zone vertex may entail the generation of parsing trees that are more likely to be the correct ones and end up reducing the search effort considerably.

Our belief is that the *greedy aware* strategy outperforms the other two alternatives in average, although we also suppose that some strategy may suit better the parsing of some classes of graphs or grammars. The former expectation is supported by one of our brief experimental evaluations but we cannot affirm that firmly, for a more detailed study would be necessary for that end.

More strategies besides the three ones presented here could be created, including, for example, the implementation of meta-heuristics, like the simulated annealing, to guide the parsing tree search in a more robust manner.

Regarding the parallelization of the parsing procedure, it is possible to parallelize it with as many threads as wished. We do it by having a central manager for the *bup* set, that retrieves subsets and adds zone vertices to the *bup* upon requests and according to the strategy. This central manager receives such requests from the concurrent threads, that effectively evaluate a subset with zone vertices in search of new derivations.

In this parallel architecture, enhancements can be done to decrease the synchronization time required by the central *bup* manager at the addition of new zone vertices. This operation is specially costly because the addition of a new zone vertex implies the creation of new subsets and the insertion of them in an ordered queue. Such addition has a worst-case time complexity greater than constant in our implementation.

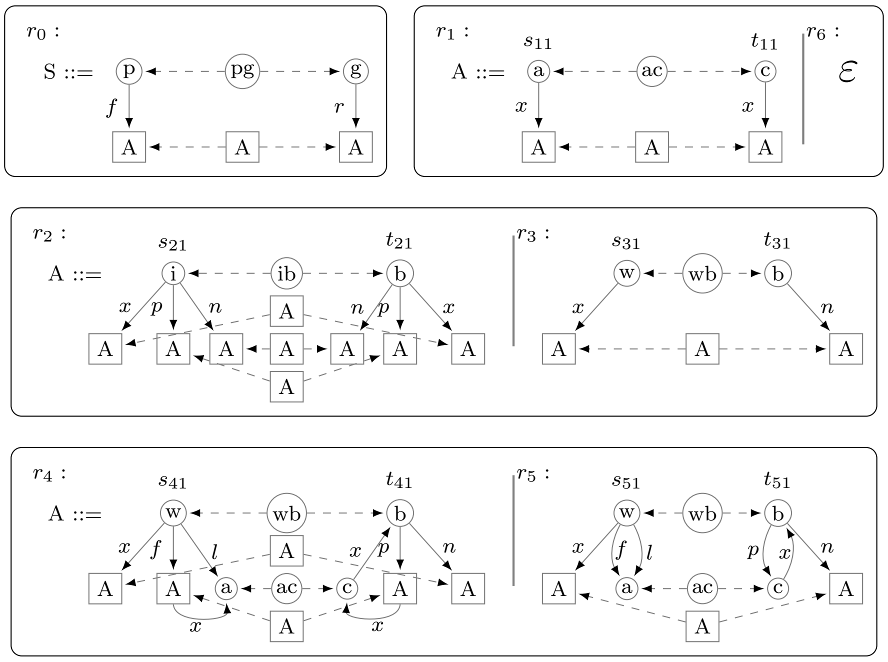
In general our parser implementation could become more efficient also through the reduction of heavy memory operations executed by the copy of zone vertices throughout the parsing process, that are not essentially necessary. Furthermore, experimental profile analysis indicate that the isomorphism checks consume a considerable amount of time. This isomorphism checks are necessary to verify that a handle can be generated by a derivation step (Line 7 and 9 of Algorithm 5.3) and, although its time complexity is a function on the maximal size of the handle— i.e. the maximal size of the right-hand sides of the grammar's rules— which tends to be much smaller than the size of the source graph, the overhead produced by it is still considerable.

Production. To finalize the whole transformation procedure, the step *Production* takes the derivation of the source graph found by the parser to create a triple graph whose source part is identical to the source graph (up to isomorphism) and the target part holds the desired transformed graph, as exposed in Chapter 4. This can be done, practically, by a two-pass method. First, a triple graph with unresolved PAC vertices is created step-by-step by iterating in the derivation, applying at each derivation step the respective triple rule on the respective vertices and creating at each derivation step a resolution step that maps the just created PAC vertices to their respective vertices in the source graph. At the second pass, these resolution steps are iterated in such a way that, at each step, a resolution step is applied to solve the respective PAC vertices.

7. Case Study

In this chapter, we take two examples of transformations specified with BNCE TGG and study them in depth, namely the already briefly introduced *Pseudocode2Controlflow* transformation and the *Class2Database* transformation. These two transformations are specially relevant for the practical application of our approach and embody well the main aspects of our presented theoretical framework. For each transformation specification, we try to convey the intuition behind each grammar rule and to make it clearer how the parsing and the transformation algorithms work by means of example derivations.

For the first case study, consider the *Pseudocode2Controlflow* transformation specification from Example 3.2. This transformation is encoded through the BNCE TGG $TGG = (\{S, A, p, a, i, w, g, b, c, f, x, n, l, r, pg, ac, ib, wb\}, \{p, a, i, w, g, b, c, f, x, n, l, r, pg, ac, ib, wb\}, S, P)$, where $P = \{r_i \mid 0 \leq i \leq 5\}$ is denoted by



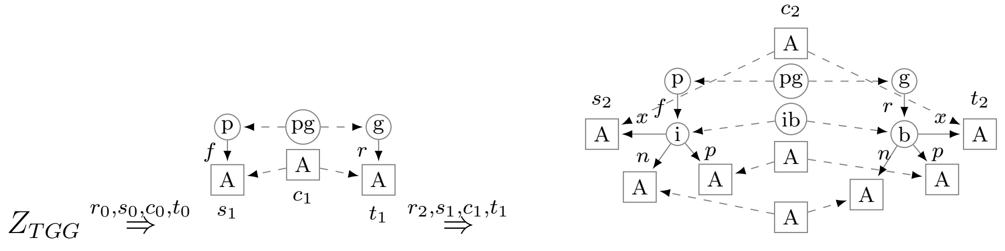
with $\sigma_0 = \emptyset$, $\sigma_1(s_{11}) = \sigma_2(s_{21}) = \sigma_3(s_{31}) = \sigma_4(s_{41}) = \sigma_5(s_{51}) = \{(f, p), (x, a), (x, i), (x, w), (p, i), (n, i), (l, w), (f, w)\}$ and $\tau_1(t_{11}) = \tau_2(t_{21}) = \tau_3(t_{31}) = \tau_4(t_{41}) =$

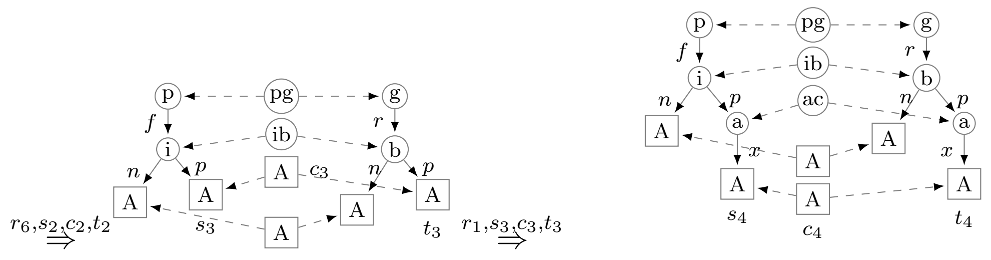
$\tau_5(t_{51}) = \{(r, g), (x, c), (x, b), (p, b), (n, b)\}$ being the complete definition of the source and target embedding functions of the rules r_0 to r_5, respectively.

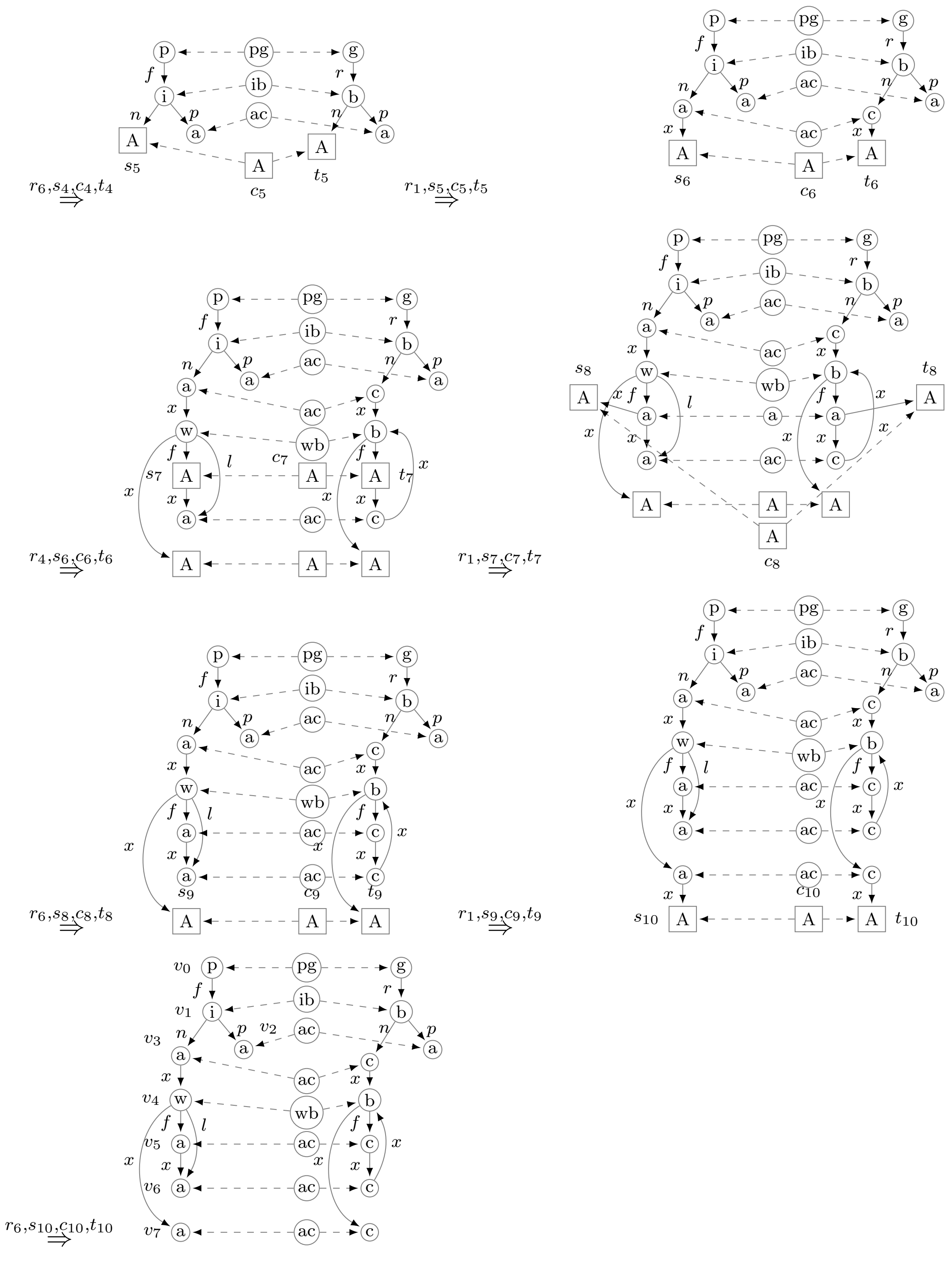
Rules. The rule r_0 is responsible for creating the only p and g vertices of each triple graph in $L(TGG)$. This is the only rule that can transform the start symbol S into something else and thus the rule that is always applied first in any derivation for triple graphs in $L(TGG)$. One could say that r_0 encodes the fact that any *Pseudocode* graph consists of a p (i.e. a program) containing an A (i.e. an statement) and a *Controlflow* graph consists of a g (i.e. a graph) containing an A (i.e. a basic block). The different possibilities of what an A can be is in turn encoded by the different rules r_1, r_2, r_3, r_4, r_5, and r_6.

Through r_1 can an A be transformed into an a (i.e. an action) in the *Pseudocode* graph and into an c (i.e. a command) in the *Controlflow* graph, both followed by another A. Analogously, through r_2 can A become a i (i.e. an if) and a b (i.e. a branch) both with positive and negative branches, which are also As, followed by another A. r_3 transforms an A into an w (i.e. a while) and a b without further follow-up vertices except by an A through the edge x, that is, r_3 can produce empty loops. r_5 and r_4, on the contrary, produce, respectively, a loop with one internal action/ command or with more than one internal statement/ basic block, represented by an A that must be followed by an a/ c. Thereby, the f-labeled edge indicates the first statement in a loop and the l-labeled edge indicates the last action in the loop. We require the last element to be an action so that we can assure that it has no follow-ups. Lastly, r_6 allows an A to be transformed into an empty graph, which has the effect of removing A and makes it possible for a derivation to stop.

Derivation. In order to obtain a more concrete understanding on how the rules in P work, we provide in the following the only derivation in TGG for the triple graph TG from Example 3.2. By investigating this derivation, it should be clear how the BNCE TGG mechanism works. It starts by the start triple graph Z_{TGG} and apply consecutively rules from P to produce finally the goal triple graph.




r6,s2,c2,t2
p
pg
g
f
r
i
ib
b
n
p
A
c3
n
p
A
A
A
A
s3
A
t3
r1,s3,c3,t3
p
pg
g
f
i
ib
r
b
n
p
ac
n
p
A
a
A
a
x
A
x
A
A
A
s4
c4
t4


$\overset{r_6,s_4,c_4,t_4}{\Longrightarrow}$
s_5
c_5
t_5
$\overset{r_1,s_5,c_5,t_5}{\Longrightarrow}$
s_6
c_6
t_6
$\overset{r_4,s_6,c_6,t_6}{\Longrightarrow}$
s_7
c_7
t_7
$\overset{r_1,s_7,c_7,t_7}{\Longrightarrow}$
s_8
c_8
t_8
$\overset{r_6,s_8,c_8,t_8}{\Longrightarrow}$
s_9
c_9
t_9
$\overset{r_1,s_9,c_9,t_9}{\Longrightarrow}$
s_{10}
c_{10}
t_{10}
$\overset{r_6,s_{10},c_{10},t_{10}}{\Longrightarrow}$
v_0
v_1
v_2
v_3
v_4
v_5
v_6
v_7

Transformation. The transformation procedure consists, as already expound in Chapters 4, 5 and 6, of the parsing of the input graph and the production of the triple graph. The goal of the parsing is to find the parsing tree correspondent to a derivation D for the input as fast as possible. This search is efficiently performed when the set of found derivation steps contains only the ones in D. In Algorithm 3.1 this is achieved when the set *bup* grows minimally, that is, it is enlarged only with the zone vertices corresponding to the derivation steps in D.

To illustrate this optimal growth for Example 3.2, we construct the minimal final value for *bup*, by adding new subsets to its initial value $bup = \{(p, \{v_0\}), (i, \{v_1\}), (a, \{v_2\}), (a, \{v_3\}), (w, \{v_4\}), (a, \{v_5\}), (a, \{v_6\}), (a, \{v_7\})\}$. This construction is as follows,

$$\begin{aligned} bup \leftarrow bup &\cup \{(A, \{\})\} && (7.1)\\ &\cup \{(A, \{v_7\})\} \cup \{(A, \{v_5\})\} \cup \{(A, \{v_4, v_5, v_6, v_7\})\} \cup \{(A, \{v_2\})\} && (7.2)\\ &\cup \{(A, \{v_3, v_4, v_5, v_6, v_7\})\} \cup \{(A, \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\})\} && (7.3)\\ &\cup \{(S, \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\})\} && (7.4) \end{aligned}$$

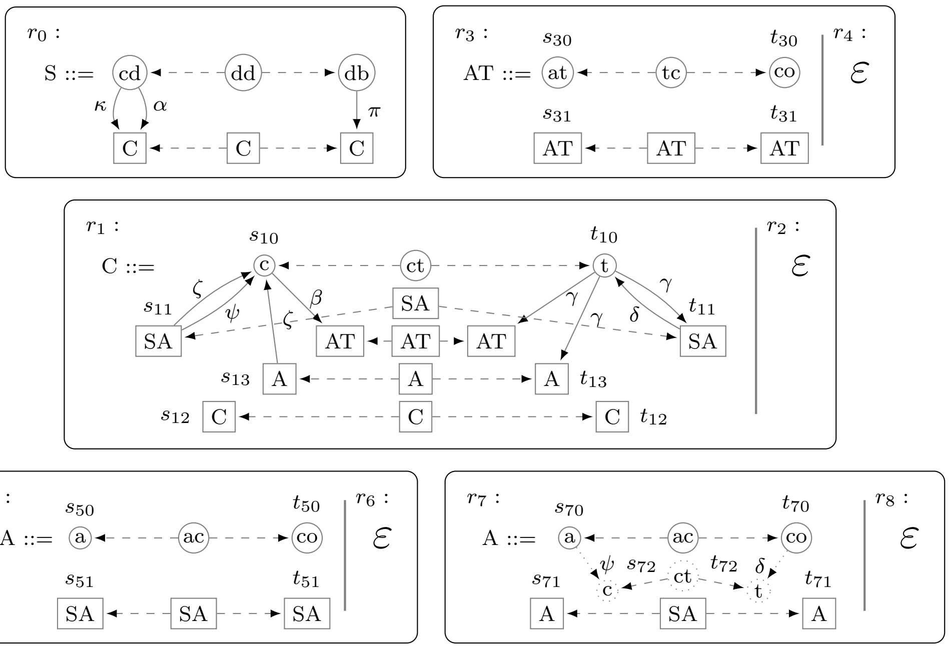
Through Equation 7.1 *bup* receives the zone vertex $(A, \{\})$ because A produces an empty graph through rule r_6. Through Equation 7.2 *bup* is enlarged with those zone vertices stemming from derivations steps $\overset{r_1,s_9,c_9,t_9}{\Rrightarrow}$, $\overset{r_1,s_7,c_7,t_7}{\Rrightarrow}$, $\overset{r_4,s_6,c_6,t_6}{\Rrightarrow}$, $\overset{r_1,s_3,c_3,t_3}{\Rrightarrow}$. Lastly, Equation 7.2 enlarges *bup* with those zone vertices stemming from derivations steps $\overset{r_1,s_5,c_5,t_5}{\Rrightarrow}$, $\overset{r_2,s_1,c_1,t_1}{\Rrightarrow}$ and Equation 7.4 adds the final zone vertex $(S, V_{S_{TG}})$ because of derivation step $\overset{r_0,s_0,c_0,t_0}{\Rrightarrow}$.

Although the optimal search course with a minimal final value for *bup* is desired, it is not always achieved by our implementation, despite the several heuristics presented in Chapter 6. For instance, a naive implementation could end up adding the zone vertices $(A, \{v_6\})$ or $(A, \{v_4, v_5, v_6\})$ to *bup*, because rules r_1 and r_4 allow it, even though they do not contribute to the final parsing tree. In other words, not all found derivations and added zone vertices are useful for finding the solution of the parsing problem, instead they even lead to useless exploration of the search space. Indeed, this is a negative aspect of our implementation, for determinism is not guaranteed and such useless computations may affect considerably the runtime of the parser. We believe that a better handling of how empty productions are used and a formalism where the direction of the edges played a bigger role by the embedding could improve our mechanism.

For the second case study, consider the *Class2Database* transformation specification, which specifies all triple graphs holding correctly transformed class diagrams and database diagrams. Class diagrams are very often used for modelling of object-oriented information systems and database diagrams are used to depict database schemes. For simplicity purposes, we simplify considerably such diagrams. Here, class diagrams are graphs containing exactly one *cd*-labeled vertex that represents the *class diagram* and that is connected with all *c*-labeled and *a*-labeled vertices that represent *classes* and *associations* through edges κ and α, respectively. An *association* is necessarily connected to a source and a target *class* through the edges

ζ and ψ, respectively. Additionally, a *class* may have zero ore more *attributes* represented by *at*-labeled vertices connected to its *class* through a β-labeled edge. A database diagram is a graph containing exactly one *db*-labeled vertex that represents the *database* and is connected through π-labeled edges to all *t*-labeled vertices, which represent *tables*. A *table* may have one or more *columns* connected to its tables through γ edges. A *column* can additionally reference an extra *table* through a δ edge.

The *Class2Database* transformation is encoded through the BNCE TGG $CD = (\{S, C, AT, SA, A, cd, c, at, a, db, t, co, dd, tc, ct, ac, \kappa, \alpha, \zeta, \psi, \beta, \pi, \gamma, \delta\}, \{cd, c, at, a, db, t, co, dd, tc, ct, ac, \kappa, \alpha, \zeta, \psi, \beta, \pi, \gamma, \delta\}, S, P)$, where $P = \{r_i \mid 0 \leq i \leq 8\}$ is denoted by



with $\sigma_0 = \sigma_2 = \sigma_4 = \sigma_6 = \sigma_8 = \emptyset$, $\sigma_1(s_{10}) = \{(\kappa, cd), (\psi, a)\}$, $\sigma_1(s_{12}) = \{(\kappa, cd), (\alpha, cd)\}$, $\sigma_1(s_{11}) = \sigma_1(s_{13}) = \{(\alpha, cd)\}$, $\sigma_3(s_{30}) = \sigma_3(s_{31}) = \{(\beta, c)\}$, $\sigma_5(s_{50}) = \sigma_5(s_{51}) = \{(\zeta, c), (\psi, c), (\alpha, cd)\}$, $\sigma_7(s_{70}) = \sigma_7(s_{71}) = \{(\zeta, c), (\alpha, cd)\}$, $\sigma_7(s_{72}) = \{(\kappa, cd), (\zeta, a), (\psi, a), (\beta, at)\}$ being the complete definition of the source embedding functions and $\tau_0 = \tau_2 = \tau_4 = \tau_6 = \tau_8 = \emptyset$, $\tau_1(t_{10}) = \{(\pi, db), (\delta, co)\}$, $\tau_1(t_{12}) = \{(\pi, db)\}$, $\tau_3(t_{30}) = \tau_3(t_{31}) = \{(\gamma, t)\}$, $\tau_5(t_{50}) = \tau_5(t_{51}) = \{(\gamma, t), (\delta, t)\}$, $\tau_7(t_{70}) = \tau_7(t_{71}) = \{(\gamma, t)\}$, $\tau_7(t_{72}) = \{(\pi, db), (\delta, co), (\gamma, co)\}$ being the complete definition of the target embedding functions of the rules r_0 to r_8, respectively.

Rules. Rule r_0 is responsible for guaranteeing that every triple graph in $L(CD)$ has exactly one *cd*-labeled and one *db*-labeled vertex in the source and target graphs, respectively. Rule r_1 is responsible for creating each $c/\ t$ vertex of any triple graph, which is connected to three nonterminal vertices AT(representing its attributes/

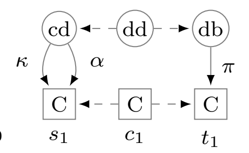
columns), A (representing its associations/ references) and SA (representing its self associations/ references). The C-labeled vertices allow for a triple graph to have more than one class/ table. Rule r_3 produces all *at*/ *co* vertices of a class. The AT vertex in this rule works for allowing a class/ table to have more than one attribute/ column. Rule r_5 produces, analogously, a-labeled vertices representing associations whose source and target are the same class in the source graph and *co*-labeled vertices representing columns that reference the same table to which they belong in the target graph.

Rule r_7 create all associations between different classes in the source graph and all columns that reference not its owner table in the target graph. Thereby, it is important to notice that the c and t-labeled vertices are PAC, what means that they are not produced by r_7, but they work as prerequisites for r_7 to be applied. That is, an association can only be created if it has a target class—represented by the c-labeled PAC vertex, which is resolved to a concrete vertex by a resolution step. Notice also that the existence of a source association is guaranteed by the ζ-labeled edge from A to c in r_1. In this rule, the c-labeled vertex could not be a common (i.e. a non-PAC) vertex, because it would imply in the creation of a new class for each association, what in turn would not allow a class to have more than one incoming association. One could think of the PAC mechanism as a way to make a rule refer to vertices created by another rules before or after its application in a derivation. Insofar, we judge that this feature is a very powerful tool for the enhancement of the BNCE formalism.

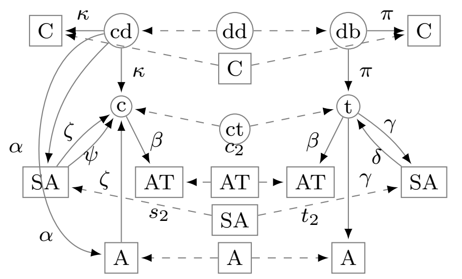
Lastly, rules r_2, r_4, r_6, and r_8 allow the vertices labeled with the symbols C, AT, SA, and A to be removed from a triple graph. Here, we highlight the practicality of such empty productions, for they allow to denote not only the optionality of a concept in a triple graph but also the termination of a recursive definition.

Derivation. In the following, we present an example of a derivation with PAC of a triple graph that belongs to the language of the triple graph grammar CD, introduced above. Notice that this is not the only possible derivation for the given triple graph, for the derivation steps could be rearranged in other order. Therefore, the grammar CD is ambiguous. Notice also that, for a matter of space, we do not draw all the graphs in the derivation, instead we write G_0, G_1, G_2, G_3, and G_4 for some. The derivation should still be clear for the careful reader.

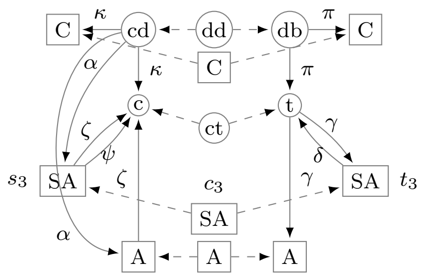
$Z_{TGG} \overset{r_0,s_0,c_0,t_0}{\Longrightarrow}$



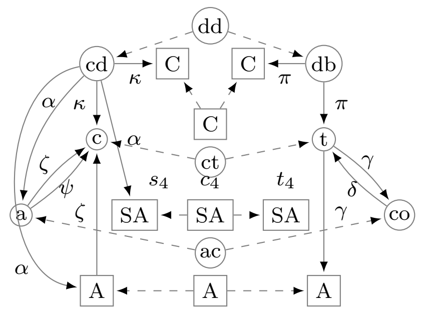
$\overset{r_1,s_1,c_1,t_1}{\Longrightarrow}$



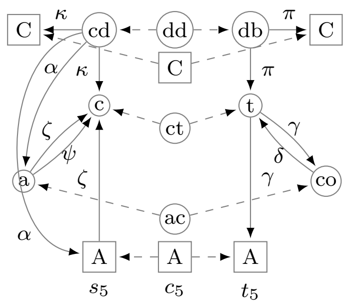
$\overset{r_4,s_2,c_2,t_2}{\Longrightarrow}$



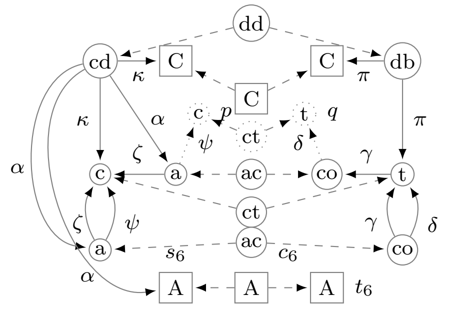
$\overset{r_5,s_3,c_3,t_3}{\Longrightarrow}$



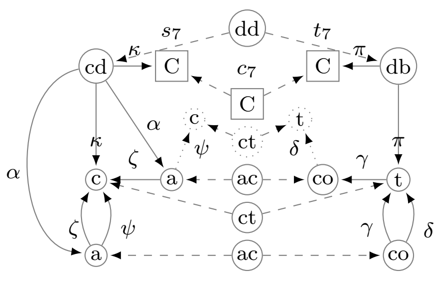
$\overset{r_6,s_4,c_4,t_4}{\Longrightarrow}$



$\overset{r_7,s_5,c_5,t_5,\{p\},\{q\}}{\Longrightarrow}$

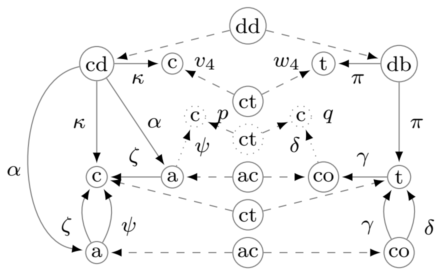


$\overset{r_8,s_6,c_6,t_6}{\Longrightarrow}$

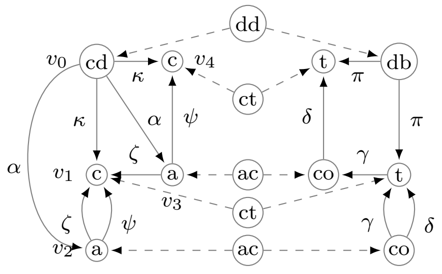


$\overset{r_1,s_7,c_7,t_7}{\Longrightarrow} G_0 \overset{r_2,s_8,c_8,t_8}{\Longrightarrow} G_1 \overset{r_6,s_9,c_9,t_9}{\Longrightarrow} G_2$

$\overset{r_4,s_{10},c_{10},t_{10}}{\Rightarrow} G_3 \overset{r_8,s_{11},c_{11},t_{11}}{\Rightarrow}$



$\overset{\emptyset}{\rightarrowtail}^{5}\ G_3 \overset{\{p\mapsto v_4\},\{q\mapsto w_4\}}{\rightarrowtail}$



$\overset{\emptyset}{\rightarrowtail}^{6}\ G_4$

Transformation. As already pointed out, the parsing phase of the transformation consists of growing the *bup* until the final zone vertex containing all vertices of the input graph is obtained. A minimal *bup* set at the end of the parsing of the input graph derived previously is constructed in the sequel from the initial value $bup = \{(cd, \{v_0\}), (c, \{v_1\}), (a, \{v_2\}), (a, \{v_3\}), (c, \{v_4\})\}$

$$bup \leftarrow bup \cup \{(C, \{\}, \{\}), (AT, \{\}, \{\}), (SA, \{\}, \{\}), (A, \{\}, \{\})\} \tag{7.5}$$
$$\cup \{(C, \{v_4\}, \{\})\} \cup \{(SA, \{v_2\}, \{\})\} \tag{7.6}$$
$$\cup \{(A, \{v_3\}, \{v_4\})\} \tag{7.7}$$
$$\cup \{(C, \{v_1, v_2, v_3, v_4\}, \{v_4\})\} \cup \{(S, \{v_0, v_1, v_2, v_3, v_4\}, \{v_4\})\} \tag{7.8}$$

Through Equation 7.5 *bup* receives the zone vertex correspondent to the rules that generate an empty graph r_2, r_4, r_6, and r_8, respectively. Through Equation 7.6 *bup* is enlarged with those zone vertices stemming from derivations steps $\overset{r_1,s_7,c_7,t_7}{\Rightarrow}$, $\overset{r_5,s_3,c_3,t_3}{\Rightarrow}$. Then, Equation 7.7 enlarges *bup* with the zone vertex stemming from derivation step $\overset{r_7,s_5,c_5,t_5,\{p\},\{q\}}{\Rightarrow}$. Notice that this zone vertex holds the vertex v_4 as a PAC vertex, which is used then later by the production phase of the transformation to resolve this PAC. Finally, Equation 7.8 adds the last zone vertices because of derivation steps $\overset{r_1,s_1,c_1,t_1}{\Rightarrow}$ and $\overset{r_0,s_0,c_0,t_0}{\Rightarrow}$.

Differently from the first case study, this case study makes use of the PAC mechanism. Thus, its production phase in the transformation procedure occurs in a two-pass fashion, as expounded in Chapter 6. In the first pass, the rule of each derivation step is applied on the triple graph being constructed and PAC vertices are created (in the example, vertices p and q). Because the zone vertices contain the information about the actual vertex in which the PAC vertices should be resolved, resolution steps according to this informations are created correctly, such that, in the second pass, the PAC vertices p and q are resolved into v_4 and w_4.

8. Evaluation

In this chapter, we present the results of the experimental evaluation of our work. We assess the usability of the BNCE TGG formalism, by comparing the size of the BNCE grammars for five example transformations with their equivalent grammars written in standard TGG, and the performance of our implemented transformer, by measuring the average runtime took to transform some model instances for our example transformations.

8.1 Usability

In order to evaluate the usability of the proposed BNCE TGG formalism, we compare the amount of rules and elements (vertices, edges and mappings) we needed to describe some typical model transformations in BNCE TGG and in standard TGG without application conditions. Table 8.1 presents these results. Each line displays the results for one different transformation, the first and second columns provide the amount of rules and elements of the standard TGG specifications and the third and forth columns the amount of rules and elements of the BNCE TGG specifications, respectively. The size of the smaller specification for each transformation is printed in bold. If a transformation could not be specified with a formalism, the respective cells are marked with a dash (-). The three last lines indicate the sum, the arithmetic average and the median of the amount of rules and elements of each formalism for the compared transformations, respectively. The transformations *Pseudocode2Controlflow*, *BTree2XBTree* and *Star2Wheel* were specified with BNCE TGG without PAC, whereas the transformations *Class2Database* and *Statemachine2Petrinet* were specified with PAC BNCE TGG.

In general, we judge that the smaller a grammar is, the better its usability is. In this sense, our approach outperforms the baseline in one transformation case and can specify another case, that we could not specify with standard TGG at all. In addition, judging by the measurements of total and average, BNCE TGG perform significantly better than the considered baseline. This observation is though not conclusive, because the negative result of the standard TGG is strongly influenced by one studied case in which it performs specially worse, this is made clear by the median of the grammar sizes. Insofar, we cannot claim that our evaluation has a strong statistical validity, for the studied transformations are not very representative in general and we cannot guarantee that these are the smallest grammars that describe the desired transformations. Nonetheless, this results should demonstrate the potential of our approach.

Table 8.1: Results of the usability evaluation of the BNCE TGG formalism in comparison with the standard TGG for the model transformation problem

	Standard TGG		BNCE TGG	
Transformation	Rules	Elements	Rules	Elements
Pseudocode2Controlflow	47	1085	**7**	**185**
BTree2XBTree	**4**	**50**	5	80
Star2Wheel	-	-	**6**	**89**
Class2Database	**5**	**80**	9	117
Statemachine2Petrinet	**5**	**114**	7	131
Total	61	1329	**28**	**513**
Average	15.25	332.25	**7**	**128.25**
Median	**5**	**97**	7	124

In the case of *Pseudocode2Controlflow*, our proposed approach shows a clear advantage against the standard TGG formalism. We judge that, similarly to what happens to programming languages, this advantage stems from the very nested structure of *Pseudocode* and *Controlflow* graphs. That is, for instance, in rule the r_2 of this TGG (see Example 3.2), a node in a positive branch of an if-labeled vertex is never connected with a node in the negative branch. This disjunctive aspect allows every branch to be defined in the rule (as well as effectively parsed) independently of the other branch. This characteristic makes it possible for BNCE TGG rules to be defined in a very straightforward manner and reduces the total amount of elements necessary.

In addition to that, the use of non-terminal symbols gives BNCE TGG the power to represent abstract concepts very easily. For example, whereas the rule r_1 encodes, using only few elements, that after each *action* comes any statement A, which can be another *action*, an *if*, a *while* or nothing (an empty graph), in the standard TGG without application condition or any special inheritance treatment, we need to write a different rule for each of these cases. For the whole grammar, we need to consider all combinations of *actions*, *ifs* and *whiles* in all rules, what causes the great amount of rules and elements.

The *BTree2XBTree* transformation consists of lifting binary trees to graphs by adding edges between siblings. In this scenario, our approach performed slightly worse than TGG. The *Star2Wheel* transformation consists of transforming star graphs, which are complete bipartite graphs $K_{1,k}$, with the partitions named center and border, to wheel graphs, that can be constructed from star graphs by adding edges between border vertices to form a minimal cycle. We could not write this transformation in standard TGG, specially because of the rules' monotonicity (see Definition 3.13). That is, we missed the possibility to erase edges in a rule, feature that we do have in the semantics of BNCE TGG through the embedding mechanism.

In summary, our experimental usability evaluation does not allow the drawing of definitive conclusions concerning the usability of the compared formalisms, although it provides strong positive evidences about the BNCE TGG's potential. In addition, we cannot affirm which of the two formalisms is more expressive, for that would require a deeper theoretical analysis that include the characterization of an order of grammars. That is, in order to say that a family of grammars is more expressive than other, we would need to find a containment relation between the former and the latter, what does not seem to be trivial.

Positively, we highlight the capacity of BNCE TGG to specify very compactly transformations between graphs for whose label sets there exists some kind of inheritance relation. In other words, models in which element types' undergo a hierarchical structure fit BNCE TGG well. That is particularly common for behavioral models of software systems, therefore, we claim that our approach suits well, for example, the automatic generation of source-code out of graphic models.

Regarding the *Star2Wheel* transformation, we believe that it could be written with standard TGG augmented with application conditions (AC). In which case, BNCE TGG can be seen as an alternative to TGG with AC, what makes it, in circumstances where AC are not wished, specially applicable.

Negatively, the lack of context imposed by the pure BNCE TGG demand the use of PAC for some transformations. Although we prove in Theorem 5.1 that the PAC mechanism can be used safely for the solving of the model transformation problem, we suppose that its application can be perceived as cumbersome in some situations, especially with the occurrence of multiple PAC, that we did not include in our evaluation.

8.2 Performance

For the purpose of evaluating the performance of our implemented transformation algorithm, we measure the runtime taken by it to forward-transform various input graphs of various sizes for the same five example transformations from the previous section. We provide the comparison between the average runtime of our implementation and of the *eMoflon*[1] standard TGG transformer [LAS14a] discriminated by the whether the input graph generates a deeper or a shallower parsing tree. The results are depicted in Figures 8.1 to 8.5, whereby the x-axis represents the size of the inputs, the y-axis represents the runtime in seconds and the lines display

[1]emoflon.org

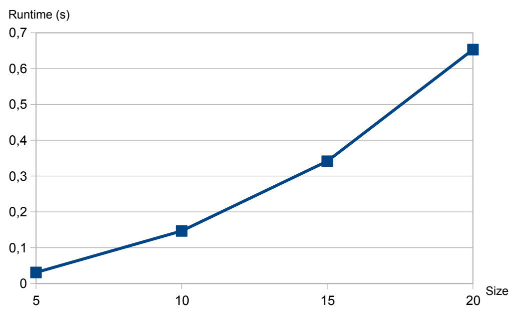


Figure 8.1: The arithmetic average of the execution times of several runs of *Star2Wheel* transformations with example input graphs whose sizes range from 5 to 20 on the BNCE transformer

the arithmetic average of the runtimes for the two classes of inputs—*Deep* and *Shallow*—on the two transformer implementations—our *BNCE* and the *eMoflon* implementations. We advise that the scale on the y-axis vary for each chart.

We execute the transformations on a Intel Core i7 2.3GHz 4x 64bit with 8GB RAM. The runtime measurement consists of the difference between the system time after and before the transformation of a each input graph, whereby both input graph and grammar are lazily load in memory beforehand. We transform, in total, 16 graphs, separated in four different sizes (graphs with 5, 10, 15, and 20 vertices) and two different classes (graphs whose parsing trees are either deep or shallow), for the four comparable transformations *Pseudocode2Controlflow*, *BTree2XBTree*, *Class2Database*, and *Statemachine2Petrinet* on both evaluated implementations and more 4 graphs for the *Star2Wheel* transformation on our BNCE implementation, one after another, and repeat it for 5 times, what culminates in 680 runs. For our BNCE TGG transformer, we set a runtime limit of 120s and configure the parsing procedure with 4 threads and the greedy aware strategy.

eMoflon is a very efficient tool that supports model transformation with TGG [LAS14a]. Differently from our approach, eMoflon does not interpret an input grammar to perform transformation. Instead, it first compiles the input grammar to Java, generating an application that is able to transform input models for that specific grammar. We picked eMoflon as baseline implementation because latest reports evinces its advantage over other TGG tools, like *MoTE*[2] or *TGG Interpreter*[3], when the task to be solved is batch forward transformation [HLG+13, LAS+14b].

Figure 8.1 displays the result of the performance evaluation for the transformation *Star2Wheel*. For this transformation, we cannot discriminate between star graphs with a certain size n that generate a deep or a shallow parsing tree, because there exists only one such star graph with size n (up to isomorphism). Furthermore, as

[2]www.hpi.uni-potsdam.de/giese/public/mdelab/mdelab-projects/mote-a-tgg-based-model-transformation-engine

[3]jgreen.de/tools/tgg-interprete

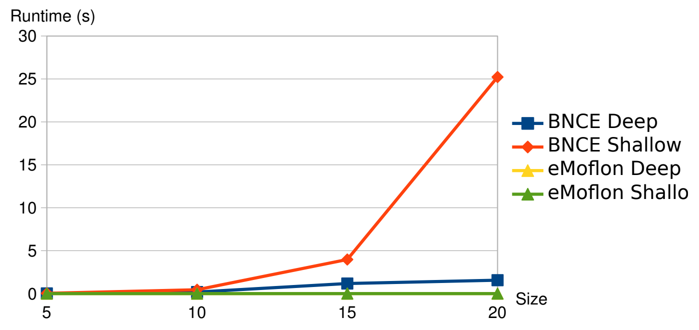
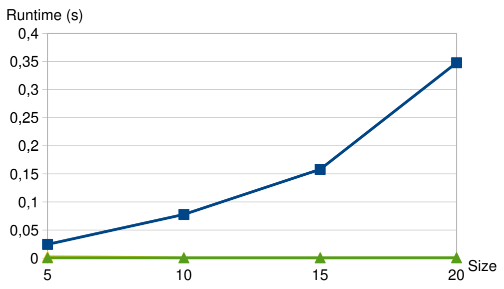
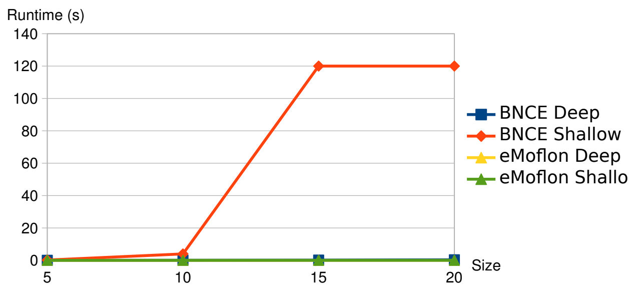


Figure 8.2: The arithmetic average of the execution times of several runs of *Pseudocode2Controlflow* transformations with example input graphs whose sizes range from 5 to 20 and that are discriminated by their parsing trees (*Deep* or *Shallow*) on both implementations (*BNCE* and *eMoflon*)



(a) Runtimes only for the class *Deep*



(b) Runtimes for all four classes

Figure 8.3: The arithmetic average of the execution times of several runs of *BTree2XBTree* transformations with example input graphs whose sizes range from 5 to 20 and that are discriminated by their parsing trees (*Deep* or *Shallow*) on both implementations (*BNCE* and *eMoflon*)

stated in Section 8.1, we could not model this transformation in standard TGG. Thus, the only line in Figure 8.1 refers to the BNCE implementation and evinces the already expected polynomial behavior of the algorithm.

Figure 8.2 reports on the average runtimes for the *Pseudocode2Controlflow* transformation. For input graphs that generate a deep parsing tree, we perform only slightly worse than eMoflon. However, for the case of shallow parsing trees, the computation time for our implementation grows fast with the inputs' sizes. We think that the good performance of the former case is due to our greedy heuristic that prioritizes the exploration of deeper trees, what enhances the probability of finding the complete parsing tree for these graphs first.

Figure 8.3a depicts the runtimes for the *BTree2XBTree* transformation for the inputs that generate deep parsing trees on both implementations. Thereby, it is made clear how the BNCE's curve grows faster than the eMoflon's with greater inputs.

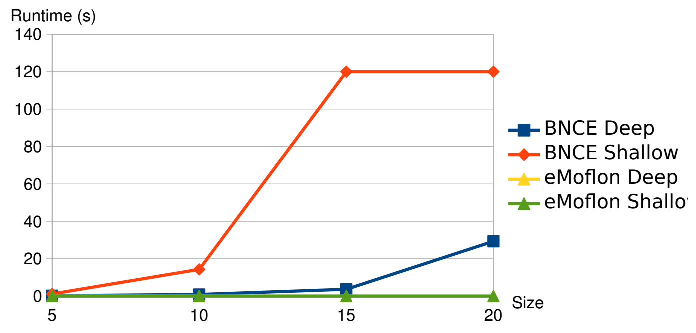


Figure 8.4: The arithmetic average of the execution times of several runs of *Statemachine2Petrinet* transformations with example input graphs whose sizes range from 5 to 20 and that are discriminated by their parsing trees (*Deep* or *Shallow*) on both implementations (*BNCE* and *eMoflon*)

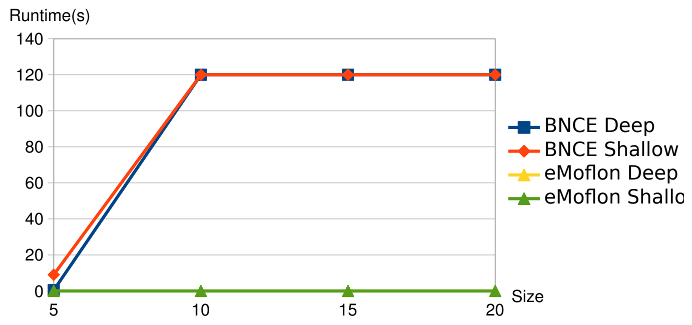


Figure 8.5: The arithmetic average of the execution times of several runs of *Class2Database* transformations with example input graphs whose sizes range from 5 to 20 and that are discriminated by their parsing trees (*Deep* or *Shallow*) on both implementations (*BNCE* and *eMoflon*)

Indeed, eMoflon's curve seems to be characterized by a polynomial with a much smaller degree than the BNCE's.

Figure 8.3b depicts the runtimes also for the *BTree2XBTree* case where the two classes of inputs on both implementations are evaluated. This result demonstrates how our implementation performs bad for the class of shallow parsing tree. In particular, the runtime function grows vertiginously for input sizes greater than 10. Notice that, the runtimes are limited by 120s, as we set this timeout in our configuration. Therefore, it is fair to assume, that the real runtime for this class is even worse.

Figure 8.4 reports on the runtime evaluation of the *Statemachine2Petrinet* transformation for the two classes of inputs on both evaluated implementations and shows again the disadvantage of our BNCE TGG implementation against eMoflon, specially for the shallow parsing tree case.

Lastly, Figure 8.5 presents the worst performance result of our approach, which is the one for the *Class2Database* transformation. For this case, our solution is impracticable even for inputs of size 10 that generate deep parsing trees, whereas eMoflon presents a very good performance for both classes of inputs. We cannot explain this phenomenon exactly, but the following reasoning about the time complexity of the BNCE TGG transformer may shed a light on it.

The worst-case time complexity of the BNCE TGG transformer is polynomial in the size of the input graphs, for degree-bounded connected graphs, but the degree of this polynomial is unknown, as far as our knowledge goes. For the general case, the problem of transforming an input graph with a BNCE TGG is NP-complete. This diagnose is valid because the BNCE TGG transformer's complexity is dominated by the parser's, which is proven to be polynomial for the connected case with bounded degree and NP-complete for the general case, according to Rozenberg et al. [RW86, p. 160]. That the total algorithm's complexity is dominated by the parser is easy to see in Figure 6.1, because the step *Ecore to Graph*'s computational effort is linear on the size of the input model and the step *NP Normalization* is dependent only on the grammar's size (what is considered to be constant) and, finally, the step *Production*'s effort is is linear on the length of the derivation, which in turn is also linear on the size of the input graph.

More detailed, the parser's complexity can be roughly described as the multiplication of two factors: the number of loop iterations executed until the desired final zone vertex is found (Lines 4 to 14 in Algorithm 5.3) and the number of operations necessary to find the derivations for a handle (Lines 5 to 13 in Algorithm 5.3). The latter is clearly a function on the size of the grammar, that is defined through the number of rules and the sizes of right-hand sides and are considered to be constant. The former corresponds to the size of the *bup* set, which, in turn, is polynomial in the size of the source graph, for a degree-bounded connected graph [RW86, p. 161], but not polynomial in the general case (unless $P = NP$).

Concerning worst-case space complexity, the BNCE parsing problem is NLOG for degree-bound connected graphs [Kim01] and PSPACE-complete for the general case [RW86], what means that the parsing algorithm runs in log-space for the former case and in polynomial-space for the latter. It is also clear that the space complexity of the BNCE transformer is dominated by the parser. In practice, our implementation consumes more space than the necessary, since we keep in memory copies of input's subgraphs that correspond to parsing trees found through the exploration of the parsing tree space. In this sense, our memory consumption could be lessened.

In summary, our approach's performance is not sufficiently good for the general case and is clearly outperformed by eMoflon, specially when the parser generates a shallow parsing tree for the input. One could argue that our transformer is still applicable for the case of deep parsing trees, but, in the practice, this would imply in a too strong restriction. As a solution for this issue, we believe that a top-down parser could enhance our transformer's performance considerably, in spite of the probably necessary backtracking. Indeed, eMoflon applies a top-down parser that do not need to backtrack, what guarantees it the good performance.

9. Conclusion

In this thesis, a novel triple graph grammar formalism that includes non-terminal symbols is proposed. This formalism is the product of the combination of an already existent graph grammar formalism with non-terminal symbols, called *graph grammar with neighborhood controlled embedding* (NCE), with the traditional triple graph grammar (TGG). We name this new fromalism NCE TGG and propose in this thesis a syntactical and semantical characterization of it. Furthermore, we present an extension of the NCE TGG that supports application conditions and increases its generative power in such an extand that allows us to apply it for real world situations, we name this extension PAC NCE TGG. For the purpose of illustrating the application of our formalism, we demonstrate how it can be used to solve the model transformation problem, which is a very relevant problem for the model-driven software development realm. Ultimately, we provide an evaluation of the usability and performance of the NCE TGG for this problem and discuss these results.

In summary, a NCE TGG is a grammar that generates a language of triple graphs. This grammar consists, basically, of an alphabet (partitioned into terminal and non-terminal symbols) and a set of rules of the form $(A \to R, \omega_s, \omega_t)$, where A is a non-terminal symbol of the alphabet and is called left-hand side, R is any triple graph over the alphabet and is called right-hand side, and ω_s and ω_t are the embedding functions that determine how the right-hand side is to be used in a rule application.

Given a triple graph G over the alphabet, a rule application in this grammar that generates a new triple graph H (we write $G \Rrightarrow H$) is done by (1) removing any triple of vertices with label A from G, (2) inserting R in G and adding edges between the removed vertices neighbors and the R's vertices according to the functions ω_s and ω_t. A triple graph containing only terminal symbols generated by a sequence of rule applications starting from a special start symbol is in the language of the grammar.

We believe that the key on understanding how a NCE TGG works lies on how the embedding functions ω_s and ω_t work. These functions map from a vertex of R and an edge label to a vertex label. In this regard, if v is a vertex of R and $\omega_s(v, \alpha) = \beta$, then on any application of this rule, v will be connected to every $\beta-$labeled vertex through an $\alpha-$labeled edge of the graph being modified, if the removed vertex had such an adjacent vertex with an $\alpha-$labeled edge.

We are aware that these embedding functions make the semantics of the rule application seem quite complicated for the unused reader. This difficulty stems from the fact that it is hard for a human to imagine on creation-time how a rule will behave when applied to vertices with different neighbors. That is, a priori, it is out of the control of the rule, which and how many neighbors will a removed vertex have. Therefore, we call for more investigations on how to make the NCE TGG's semantics more comprehensible for humans. It is possible that formalisms in which the context of the removed vertex is made more clear lead to a better understanding of the rule application. One of such formalisms is the *hyperedge replacement grammars* (HRG), where the left-hand side of a rule consists of a hyperedge connected to a fixed set of vertices. Another solution could be the explicit addition of context to the left-hand side of the rules, although it would imply the necessity for a totally new parsing and transformation algorithm.

The PAC NCE TGG extension does not solve the issue with the embeddings but does add generative power to the grammar by means of special vertices in R that are generated by rule applications and removed by so-called resolutions, we name these vertices PAC vertices. Therewith, a rule application may end up connecting vertices generated by other rule applications that could not be added without PAC vertices. We are also aware about the complication that such extension adds to the comprehension of the semantics and of the parsing algorithm for PAC NCE TGG.

Nevertheless, we are convinced that the discussions driven throughout this thesis shed a light at the current state-of-the-art of NCE graph grammars and triple graph grammars. In especial regarding the imperative characterization of the parsing and transformation algorithms exposed here. Such algorithms are often described in a less accessible way in academic literature.

As stated above, we demonstrate the application of the NCE TGG and PAC NCE TGG on the solving of the model transformation problem. When two correctly transformed models are packed in a triple graph, then the set of all correctly transformed models is the language of a TGG. By being so, the problem of model transformation reduces in polynomial time to the problem of parsing a graph with a graph grammar. We expose therefore a parsing algorithm for NCE TGG [RW86] and PAC NCE TGG that has polynomial worst-case time complexity. This algorithm is based on the CYK parser for string grammars and applies dynamic programming techniques to construct bottom-up a parsing tree for the input graph. The algorithm starts by constructing parsing trees for subgraphs of the input graph using the grammar rules and tries to increase the size of such parsing trees until one of them cover the whole input graph. Hence, this process can be seen as a systematic exploration of the search space of parsing trees for the grammar, which is big. And as our parser cannot impede unproductive explorations of it, it ends up often performing unnecessary computations which affects negatively the method's performance.

Therefore, we would wish to have a more efficient parsing algorithm, that implemented, for example, better search heuristics. Another alternative would be the design of a top-down parser, which can construct parsing trees more eagerly. The problem of a top-down parser is that it needs to backtrack a priori, what implies a exponential complexity.

The transformation algorithm for NCE TGG consists basically of applying each rule of the derivation returned by the parser to produce the final triple graph holding the input graph and the transformed graph, this method is proved to be correct by Theorem 4.1. The transformer for PAC NCE TGG differs from the former in which it produces the output triple graph in a two-pass manner. First, it produces all vertices and edges according to the derivation given by the parser and, second, it resolves the PAC vertices, removing them and connecting their edges with correspondent normal vertices. Theorem 5.1 also proves this method's correctness.

Bibliography

[AKTY99] Yoshihiro Adachi, Suguru Kobayashi, Kensei Tsuchida, and Takeo Yaku. An NCE context-sensitive graph grammar for visual design languages. In *Visual Languages, 1999. Proceedings. 1999 IEEE Symposium on*, pages 228–235. IEEE, 1999.

[ALS16] Anthony Anjorin, Erhan Leblebici, and Andy Schürr. 20 years of triple graph grammars: A roadmap for future research. *Electronic Communications of the EASST*, 73, 2016.

[ASLS14] Anthony Anjorin, Karsten Saller, Malte Lochau, and Andy Schürr. Modularizing triple graph grammars using rule refinement. In *International Conference on Fundamental Approaches to Software Engineering*, pages 340–354. Springer, 2014.

[BDE16] Henrik Björklund, Frank Drewes, and Petter Ericson. Between a rock and a hard place–uniform parsing for hyperedge replacement DAG grammars. In *International Conference on Language and Automata Theory and Applications*, pages 521–532. Springer, 2016.

[BEDLT04] Roswitha Bardohl, Hartmut Ehrig, Juan De Lara, and Gabriele Taentzer. Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 214–228. Springer, 2004.

[BS05] Franz-Josef Brandenburg and Konstantin Skodinis. Finite graph automata for linear and boundary graph languages. *Theoretical Computer Science*, 332(1-3):199–232, 2005.

[BTS00] Paolo Bottoni, Gabriele Taentzer, and A Schurr. Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In *Proceeding 2000 IEEE International Symposium on Visual Languages*, pages 59–60. IEEE, 2000.

[CAB+13] David Chiang, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones, and Kevin Knight. Parsing graphs with hyperedge replacement grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 924–932, 2013.

[DHJM10] Frank Drewes, Berthold Hoffmann, Dirk Janssens, and Mark Minas. Adaptive star grammars and their languages. *Theoretical Computer Science*, 411(34-36):3090–3109, 2010.

[DHM15] Frank Drewes, Berthold Hoffmann, and Mark Minas. Predictive top-down parsing for hyperedge replacement grammars. In *International Conference on Graph Transformation*, pages 19–34. Springer, 2015.

[DHM17] Frank Drewes, Berthold Hoffmann, and Mark Minas. Predictive shift-reduce parsing for hyperedge replacement grammars. In *International Conference on Graph Transformation*, pages 106–122. Springer, 2017.

[DKH97] Frank Drewes, H-J Kreowski, and Annegret Habel. Hyperedge replacement graph grammars. In *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 1: Foundations*, pages 95–162. World Scientific, 1997.

[EM98] Joost Engelfriet and Sebastian Maneth. Tree languages generated by context-free graph grammars. In *International Workshop on Theory and Application of Graph Transformations*, pages 15–29. Springer, 1998.

[ER90] Joost Engelfiet and Grzegorz Rozenberg. A comparison of boundary graph grammars and context-free hypergraph grammars. *Information and Computation*, 84(2):163–206, 1990.

[ERKM99] Hartmut Ehrig, Grzegorz Rozenberg, Hans-J Kreowski, and Ugo Montanari. *Handbook of graph grammars and computing by graph transformation*, volume 3. World Scientific, 1999.

[FF14] Mariusz Flasiński and Zofia Flasińska. Characteristics of bottom-up parsable edNLC graph languages for syntactic pattern recognition. In *International Conference on Computer Vision and Graphics*, pages 195–202. Springer, 2014.

[Fla93] Mariusz Flasiński. On the parsing of deterministic graph languages for syntactic pattern recognition. *Pattern Recognition*, 26(1):1–16, 1993.

[Fla98] M Flasiński. Power properties of NLC graph grammars with a polynomial membership problem. *Theoretical Computer Science*, 201(1-2):189–231, 1998.

[FMM11] Luka Fürst, Marjan Mernik, and Viljan Mahnič. Improving the graph grammar parser of rekers and schürr. *IET software*, 5(2):246–261, 2011.

[GLM17] Sorcha Gilroy, Adam Lopez, and Sebastian Maneth. Parsing graphs with regular graph grammars. In *Proceedings of the 6th Joint Conference on Lexical and Computational Semantics (* SEM 2017)*, pages 199–208, 2017.

[HET08] Frank Hermann, Hartmut Ehrig, and Gabriele Taentzer. A typed attributed graph grammar with inheritance for the abstract syntax of UML class and sequence diagrams. *Electronic Notes in Theoretical Computer Science*, 211:261–269, 2008.

[HLG$^+$13] Stephan Hildebrandt, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin, and Andy Schürr. A survey of triple graph grammar tools. *Electronic Communications of the EASST*, 57, 2013.

[Hof05] Berthold Hoffmann. Graph transformation with variables. In *Formal Methods in Software and Systems Modeling*, pages 101–115. Springer, 2005.

[JR82] Dirk Janssens and Grzegorz Rozenberg. Graph grammars with neighbourhood-controlled embedding. *Theoretical Computer Science*, 21(1):55–74, 1982.

[Kim01] Changwook Kim. Efficient recognition algorithms for boundary and linear eNCE graph languages. *Acta informatica*, 37(9):619–632, 2001.

[Kim12] Changwook Kim. On the structure of linear apex NLC graph grammars. *Theoretical Computer Science*, 438:28–33, 2012.

[KLKS10] Felix Klar, Marius Lauder, Alexander Königs, and Andy Schürr. Extended triple graph grammars with efficient and compatible graph translators. In *Graph transformations and model-driven engineering*, pages 141–174. Springer, 2010.

[KZZ06] Jun Kong, Kang Zhang, and Xiaoqin Zeng. Spatial graph grammars for graphical user interfaces. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 13(2):268–307, 2006.

[LAS14a] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. Developing emoflon with emoflon. In *International Conference on Theory and Practice of Model Transformations*, pages 138–145. Springer, 2014.

[LAS$^+$14b] Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke, and Joel Greenyer. A comparison of incremental triple graph grammar tools. *Electronic Communications of the EASST*, 67, 2014.

[Min06] Mark Minas. Syntax definition with graphs. *Electronic Notes in Theoretical Computer Science*, 148(1):19–40, 2006.

[RS95] Jan Rekers and A Schurr. A graph grammar approach to graphical parsing. In *Proceedings of Symposium on Visual Languages*, pages 195–202. IEEE, 1995.

[RS97] Jan Rekers and Andy Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages & Computing*, 8(1):27–55, 1997.

[RW86] Grzegorz Rozenberg and Emo Welzl. Boundary NLC graph grammars—basic definitions, normal forms, and complexity. *Information and Control*, 69(1-3):136–167, 1986.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1994.

[SW98] Konstantin Skodinis and Egon Wanke. Neighborhood-preserving node replacements. In *International Workshop on Theory and Application of Graph Transformations*, pages 45–58. Springer, 1998.

[SZH$^+$15] Zhan Shi, Xiaoqin Zeng, Song Huang, Zekun Qi, Hui Li, Bin Hu, Sainan Zhang, Yanyun Liu, and Cailing Wang. A method to simplify description and implementation of graph grammars. In *Computing, Communication and Networking Technologies (ICCCNT), 2015 6th International Conference on*, pages 1–6. IEEE, 2015.

[Wan91] Egon Wanke. Algorithms for graph problems on BNLC structured graphs. *Information and Computation*, 94(1):93–122, 1991.

[ZZC01] Da-Qian Zhang, Kang Zhang, and Jiannong Cao. A context-sensitive graph grammar formalism for the specification of visual languages. *The Computer Journal*, 44(3):186–200, 2001.

[ZZKS05] Xiaoqin Zeng, Kang Zhang, Jun Kong, and Guang-Lei Song. RGG+: An enhancement to the reserved graph grammar formalism. In *005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 272–274. IEEE, 2005.

[ZZLL17] Yang Zou, Xiaoqin Zeng, Yufeng Liu, and Huiyi Liu. Partial precedence of context-sensitive graph grammars. In *Proceedings of the 10th International Symposium on Visual Information Communication and Interaction*, pages 16–23. ACM, 2017.