Graph Grammar Induction as a Parser-Controlled Heuristic Search Process

Luka Fürst[1], Marjan Mernik[2], and Viljan Mahnič[1]

[1] University of Ljubljana, Faculty of Computer and Information Science, Tržaška cesta 25, SI-1000 Ljubljana, Slovenia
{luka.fuerst,viljan.mahnic}@fri.uni-lj.si
[2] University of Maribor, Faculty of Electrical Engineering and Computer Science, Smetanova ulica 17, SI-2000 Maribor, Slovenia
marjan.mernik@uni-mb.si

Abstract. A graph grammar is a generative description of a graph language (a possibly infinite set of graphs). In this paper, we present a novel algorithm for inducing a graph grammar from a given set of 'positive' and 'negative' graphs. The algorithm is guaranteed to produce a grammar that can generate all of the positive and none of the negative input graphs. Driven by a heuristic specific-to-general search process, the algorithm tries to find a small grammar that generalizes beyond the positive input set. During the search, the algorithm employs a graph grammar parser to eliminate the candidate grammars that can generate at least one negative input graph. We validate our method by inducing grammars for chemical structural formulas and flowcharts and thereby show its potential applicability to chemical engineering and visual programming.

Keywords: graph grammars, graph grammar induction, graph grammar parsing, heuristic search

1 Introduction

Despite a large variety of applications [1, 5, 17], graph grammars have seldom been used for classifying, compressing, or characterizing graph sets. However, these potential roles would become far more important if grammars could be automatically induced from graphs. For example, by inducing a graph grammar from a set of chemical structural formulas, one could acquire a classifier to distinguish, e.g., biologically active substances from others, a way to compress large chemical databases, or a set of rules characterizing a given group of chemicals.

In this paper, we present a novel approach to inducing graph grammars from positive and (optionally) negative graph examples. Our algorithm is guaranteed to produce a grammar that can generate all of the positive and none of the negative examples. By formulating grammar induction as a best-first search process biased towards small grammars, the algorithm may be expected to induce a grammar that generalizes beyond the observed examples. The search proceeds in the specific-to-general direction, starting with a trivial grammar that can generate exactly the positive input graph set. As the algorithm progresses, it produces increasingly smaller and more general candidate grammars. To prevent over-generalization, the algorithm is coupled with a graph grammar parser, which is used to check whether a given candidate grammar can generate any negative input graph. If it can, it is immediately discarded.

In the domain of graph grammar induction, only few approaches have been formulated as a parser-controlled search process. However, the main contribution of this paper is the generalization operator in the search process, i.e., the way of proceeding from more specific to more general grammars.

The grammars induced by our algorithm constitute a subclass of the Layered Graph Grammars (LGG) formalism [18] and can therefore be parsed using the Rekers-Schürr parser [18, 8]. The parsability of the target formalism makes it possible to induce a grammar from both positive and negative examples, which results in a grammar suitable for classification purposes.

In this paper, we present two nontrivial applications of our algorithm. First, we induce a grammar of flowcharts comprising atomic, sequential, conditional, and iterative statements. As a second application, we induce a grammar of the structural formulas of a subset of hydrocarbons (chemical compounds comprising carbon and hydrogen atoms). The potential applications of inducing grammars from chemical formulas have already been mentioned. Induction of flowchart grammars (and diagram grammars in general) may find its uses in visual programming tools. Graph grammars are often difficult to create 'by hand'. Using our approach, a tool could automatically induce a parsable graph grammar from a few user-provided sample graphs.

The rest of this paper is structured as follows: In Sect. 2, we give a review of related work. Section 3 defines the basic concepts. Our approach is described in Sect. 4 and experimentally validated in Sect. 5. Section 6 concludes the paper.

2 Related Work

The work on graph grammar induction has been fairly scarce. This fact can be attributed partly to the complexity of the problem itself and partly to the lack of efficient general parsers, which stems from the NP-hardness of the parsing problem for many classes of graph grammars.

One of the first graph grammar induction approaches was proposed by Jeltsch and Kreowski [10]. Their algorithm induces a hyperedge replacement (HR) grammar [19, Chap. 2] from a set of positive graphs by successive generalizations of the trivial initial grammar. Our approach is based on a similar idea, but we employ a fairly different generalization operator and embed the generalization scheme into a search algorithm.

Jonyer et al. [11] also induce grammars from positive graphs in a specific-to-general direction. In each generalization step, their algorithm determines the 'best' (according to the Minimum Description Length principle) subgraph S in the input set, replaces it with a single nonterminal vertex v, and adds the production $v ::= S$ to the grammar. The generated productions are not equipped with any embedding rules and can therefore only represent chains of similar graphs connected with single edges. An improved version of this approach, proposed by Kukluk et al. [13], induces grammars that can represent sequences of graphs sharing common edges. Recently, Brijder and Blockeel [4] presented a method to induce a node-label controlled (NLC) grammar [19, Chap. 1] from a single graph containing a set of isomorphic subgraphs.

None of the approaches mentioned above makes use of a parser. Therefore, they cannot accept negative graphs, and the induced grammars are not suitable for classification purposes. An approach that does employ a parser, although only to validate the final grammar produced by the induction algorithm, was proposed by Ates et al. [2]. They induce grammars from the Spatial Graph Grammar formalism [12], which is parsable in polynomial time but fairly restricted.

Our approach induces grammars that are both parsable and fairly powerful, at least in comparison to those of Jonyer et al. and Ates et al. Another advantage of our method is that the parser actively participates in the induction process. Unfortunately, the combination of the power and parsability of the target formalism results in the exponential worst-case complexity of the parser and hence of the entire algorithm.

The problem of graph grammar induction has been inspired by that of string grammar induction [16], where many approaches are based on similar ideas as our method, i.e., specific-to-general search, parser-based validation, etc. [7, 15].

Graph grammar induction is also related to the problem of metamodel inference [9], where the goal is to induce a metamodel from a given set of models, and to that of model transformation by example [3], where the goal is to infer model transformation rules from a set of known transformation pairs. Since model transformation rules can be represented as graph grammars in the Triple Graph Grammar (TGG) formalism [20], the model-transformation-by-example problem can be formulated as a TGG induction problem.

3 Definitions

A *directed graph* G is a tuple (V_G, E_G, $\mathit{VLabels}_G$, $\mathit{ELabels}_G$, conn_G, vlabel_G, elabel_G), where V_G, E_G, $\mathit{VLabels}_G$, and $\mathit{ELabels}_G$ are the sets of *vertices*, *edges*, *vertex labels*, and *edge labels*, respectively, $\mathit{conn}_G \colon E_G \to V_G \times V_G$ is the function defining the source and the target vertex for each edge, and $\mathit{vlabel}_G \colon V_G \to \mathit{VLabels}_G$ and $\mathit{elabel}_G \colon E_G \to \mathit{ELabels}_G$ are the functions defining the *labels* of individual graph elements. For convenience, let $\mathit{label}_G(x) \equiv \mathit{vlabel}_G(x)$ if $x \in V_G$ and $\mathit{label}_G(x) \equiv \mathit{elabel}_G(x)$ if $x \in E_G$. Unlabeled vertices and edges will be treated as if they were labeled with a special label ϕ_V and ϕ_E, respectively. Let $|G| = |V_G| + |E_G|$ be the *size* of the graph G. *Undirected graphs* are defined in the same way as directed ones, except that for each edge e, $\mathit{conn}(e)$ is a two-element set rather than an ordered pair. The subscripts in V_G, E_G, conn_G, etc., will be omitted when the associated graph is clear from context.

Graphs G and H are *isomorphic* (denoted $G \approx H$) if there exists a bijective vertex-to-vertex and edge-to-edge mapping (called *isomorphism*) $h \colon G \to$

H that preserves labels and adjacencies, i.e., $\mathit{label}_H(h(x)) = \mathit{label}_G(x)$ and $\mathit{conn}_H(h(e)) = h(\mathit{conn}_G(e))$ for all $x \in V_G \cup E_G$ and $e \in E_G$.[3] A graph H is a *subgraph* of a graph G (denoted $H \preceq G$) if $V_H \subseteq V_G$ and $E_H \subseteq E_G$. An *occurrence* of a graph H in a graph G is a subgraph $H' \preceq G$ such that $H' \approx H$. Let us define the *neighborhood* of a subgraph $H \preceq G$ in G (denoted $\mathit{Nh}_G(H)$) as the set of all vertices in $V_G \setminus V_H$ connected to at least one vertex in V_H, i.e., $\mathit{Nh}_G(H) = \{v \in V_G \setminus V_H \mid \exists w \in V_H, e \in E_G \colon \mathit{conn}_G(e) = (v, w) \vee \mathit{conn}_G(e) = (w, v) \vee \mathit{conn}_G(e) = \{v, w\}\}$.

Let $[u\colon \mathsf{A} \xrightarrow{e\colon t} v\colon \mathsf{B}]$ (or $[u\colon \mathsf{A} \overset{e\colon t}{\text{——}} v\colon \mathsf{B}]$) denote a graph comprising a vertex u labeled A, a vertex v labeled B, and an edge e labeled t with $\mathit{conn}(e) = (u, v)$ (or $\mathit{conn}(e) = \{u, v\}$). Let $[u(S)v]$ denote a graph comprising vertices u and v, a subgraph S such that $\mathit{Nh}_{[u(S)v]}(S) = \{u, v\}$, and an arbitrary number of edges connecting the vertices of S to the vertices u and v.

Let us now define the graph grammar formalism induced by our method. A *graph grammar* is the quadruple $GG = (\mathcal{T}^V, \mathcal{T}^E, \mathcal{N}^E, \mathcal{P})$, where $\mathcal{T}^V$, $\mathcal{T}^E$, and $\mathcal{N}^E = \{\#1, \#2, \ldots\}$ are pairwise disjoint sets of *terminal vertex labels*, *terminal edge labels*, and *nonterminal edge labels*, respectively, and $\mathcal{P}$ is a set of *productions* of the form $p\colon L ::= R$, where $L = \mathit{LHS}(p)$ (the left-hand side or LHS) and $R = \mathit{RHS}(p)$ (the right-hand side or RHS) are connected graphs such that $\mathit{VLabels}_L \subseteq \mathit{VLabels}_R \subseteq \mathcal{T}^V$, $\mathit{ELabels}_L \subseteq \mathcal{N}^E$, and $\mathit{ELabels}_R \subseteq \mathcal{T}^E \cup \mathcal{N}^E$. Additionally, each production has to belong to one of the following types:

Type I: Productions of this type take the form $\lambda ::= R$, where λ denotes the graph with no elements (the null graph).

Type II: These productions take the form $[u\colon \mathsf{A} \xrightarrow{e\colon \#i} v\colon \mathsf{B}] ::= [u(S)v]$ or $[u\colon \mathsf{A} \overset{e\colon \#i}{\text{——}} v\colon \mathsf{B}] ::= [u(S)v]$, where $\{\mathsf{A}, \mathsf{B}\} \subseteq \mathcal{T}^V$ and $\#i \in \mathcal{N}^E$. The subgraph S will be called the *core*, and the vertices u and v will be called the *guards*. Productions of this type will often be written as $[\mathsf{A} \xrightarrow{\#i} \mathsf{B}] ::= [\mathsf{A}(S)\mathsf{B}]$ or $[\mathsf{A} \overset{\#i}{\text{——}} \mathsf{B}] ::= [\mathsf{A}(S)\mathsf{B}]$.

Type III: These productions take the form $[u\colon \mathsf{A} \xrightarrow{e\colon \#i} v\colon \mathsf{B}] ::= [u \xrightarrow{\mathsf{r}} v]$ or $[u\colon \mathsf{A} \overset{e\colon \#i}{\text{——}} v\colon \mathsf{B}] ::= [u \overset{\mathsf{r}}{\text{—}} v]$, where $\{\mathsf{A}, \mathsf{B}\} \subseteq \mathcal{T}^V$, $\#i \in \mathcal{N}^E$, and $\mathsf{r} \in \mathcal{T}^E \cup \mathcal{N}^E$. Productions of this type will often be written as $[\mathsf{A} \xrightarrow{\#i} \mathsf{B}] ::= [\mathsf{A} \xrightarrow{\mathsf{r}} \mathsf{B}]$ or $[\mathsf{A} \overset{\#i}{\text{——}} \mathsf{B}] ::= [\mathsf{A} \overset{\mathsf{r}}{\text{—}} \mathsf{B}]$.

For example, the grammar GG_7 in Fig. 2 contains one production of each type ($p_{7,1}$ belongs to type I, $p_{7,2}$ to type II, and $p_{7,3}$ to type III). The guard vertices on production RHSs are marked with small black circles. In the case of directed grammars (e.g., in Fig. 1), the RHS guards are marked with 'S' and 'T'. The letter 'S' marks the vertex that coincides with the source vertex on the LHS.

To *apply* a type-II production $p\colon [u\colon \mathsf{A} \xrightarrow{e\colon \#i} v\colon \mathsf{B}] ::= [u(S)v]$ to a graph G, one has to (1) find an occurrence L' of the graph $[u\colon \mathsf{A} \xrightarrow{e\colon \#i} v\colon \mathsf{B}]$ in G, (2) replace in L' the edge that corresponds to e with a copy S' of the graph S, and

[3] For any function $f\colon A \to B$, let $f((x, y)) = (f(x), f(y))$ and $f(\{x, y\}) = \{f(x), f(y)\}$.

(3) connect the vertices of S' to the vertices corresponding to u and v in the same way as the vertices of S are connected to the vertices u and v in $RHS(p)$. To apply a type-III production $[u\colon \mathsf{A} \xrightarrow{e\colon \#i} v\colon \mathsf{B}] ::= [u \xrightarrow{r} v]$ to a subgraph $[u'\colon \mathsf{A} \xrightarrow{e'\colon \#i} v'\colon \mathsf{B}]$, the edge e' has to be replaced with an edge labeled r. To *reverse-apply* a production, this procedure is simply reversed. Undirected productions are applied and reverse-applied in the same way as directed ones. A sample grammar and a series of production applications is shown in Fig. 1. The notation $L ::= R_1 \mid \ldots \mid R_s$ is an abbreviation for $L ::= R_1, \ldots, L ::= R_s$.

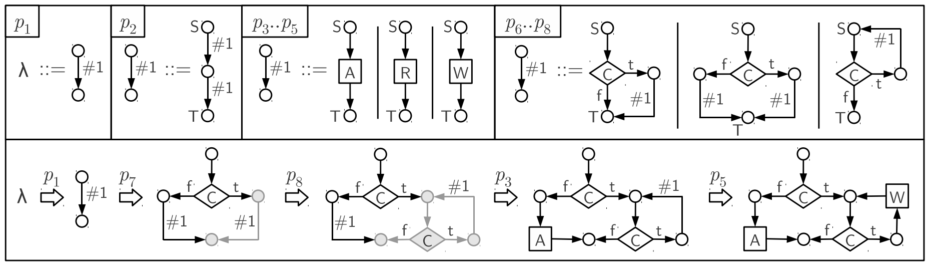

Fig. 1. *Top row*: The reference grammar for the flowcharts language. *Bottom row*: A derivation of a sample flowchart. The application of the production p_8 is highlighted.

A grammar GG *covers* a graph G if GG can generate G, i.e., if G can be derived from the null graph using the productions of GG. The *language* of a grammar GG is the set of all terminal-labeled graphs covered by GG. A *parser* is an algorithm that determines whether a given graph G belongs to the language of a given grammar GG.

The RHSs of the type-I productions of a grammar GG will be collectively called the *base graphs* of GG and denoted $\mathcal{B}(GG)$. The *size* of a grammar will be defined as $|GG| = \sum_{p \in \mathcal{P}(GG)} |p|$, where $|p| = |RHS(p)|$ if p is a type-I production, and $|p| = |RHS(p)| + |LHS(p)| - 2 = |RHS(p)| + 1$ otherwise. (The two guards are common to the LHS and RHS, hence '-2'.) A grammar GG_1 is *larger* (or *smaller*) than a grammar GG_2 if $|GG_1| > |GG_2|$ (or $|GG_1| < |GG_2|$).

Our target grammar formalism can be regarded as a subset of both LGG and HR formalisms. Hyperedge replacement grammars consist of productions for replacing individual hyperedges with hypergraphs. A *hyperedge* is an edge that connects an arbitrary sequence or multiset of vertices. (An ordinary edge is therefore a special case of a hyperedge.) A *hypergraph* is a graph composed of vertices and hyperedges. Type-I productions can be thus viewed as HR productions with zero-arity hyperedges on their LHSs. Type-II and type-III productions also specify HR rules, since the guard vertices do not participate in the replacement process itself; rather, they only determine the context of replacement. The guards correspond to *external vertices* in the HR terminology.

4 The Proposed Graph Grammar Induction Algorithm

In this section, we will often refer to Fig. 2, which shows the induction of a grammar from a single positive graph, namely the structural formula of butane. In this example, the final result is the grammar GG_7.

4.1 Overview

The pseudocode of the induction algorithm is shown in Fig. 3. The induction algorithm induces a graph grammar from a set of positive graphs ($\mathcal{G}^+$) and (optionally) a set of negative graphs ($\mathcal{G}^-$) such that $\mathcal{G}^+ \cap \mathcal{G}^- = \emptyset$. The algorithm accepts two additional parameters (positive integers): *beamWidth* specifies the beam width in the search process, and *maxVertexCount* determines the maximum vertex count in the search for production cores (explained later).

The induction algorithm operates as a specific-to-general beam search process. Its search space can be visualized as a graph in which the vertices represent individual *candidate grammars* and the edges represent possible *elementary generalizations* of candidate grammars. A candidate grammar is a grammar that covers all of the positive input graphs and none of the negative ones. An elementary generalization step transforms a given candidate grammar GG into a new candidate grammar that is at least as general as GG.

The goal of the algorithm is to find a candidate grammar of the minimum size. By restricting its search to the space of candidate grammars, the algorithm is guaranteed to produce a correct grammar in terms of the coverage of the input graphs. Its preference for small grammars is likely to result in a grammar that generalizes beyond the observed examples.

The algorithm starts with the most specific candidate grammar. This grammar, denoted GG_1, consists of productions $\{\lambda ::= G \,|\, G \in \mathcal{G}^+\}$ and thus covers precisely the positive input set. During its execution, the algorithm maintains a priority queue of all candidate grammars that have been generated but not yet generalized. In each step, the algorithm removes the smallest grammar from the queue and applies to it all possible elementary generalizations, producing a new set of candidate grammars. Each resulting grammar is verified by our improved version of the Rekers-Schürr parser [8]. If a grammar covers at least one negative input graph, it is immediately discarded; otherwise, it is placed into the queue. To reduce the computational effort, only the *beamWidth* smallest grammars are kept in the queue. When the queue becomes empty, the algorithm outputs the smallest grammar created during the search process.

4.2 Elementary Generalizations

To perform a single step forward in our specific-to-general search, a given candidate grammar is ‘slightly’ generalized or merely restructured without changing its generative power. This is achieved by elementary generalizations of two types, called ‘type A’ and ‘type B’.

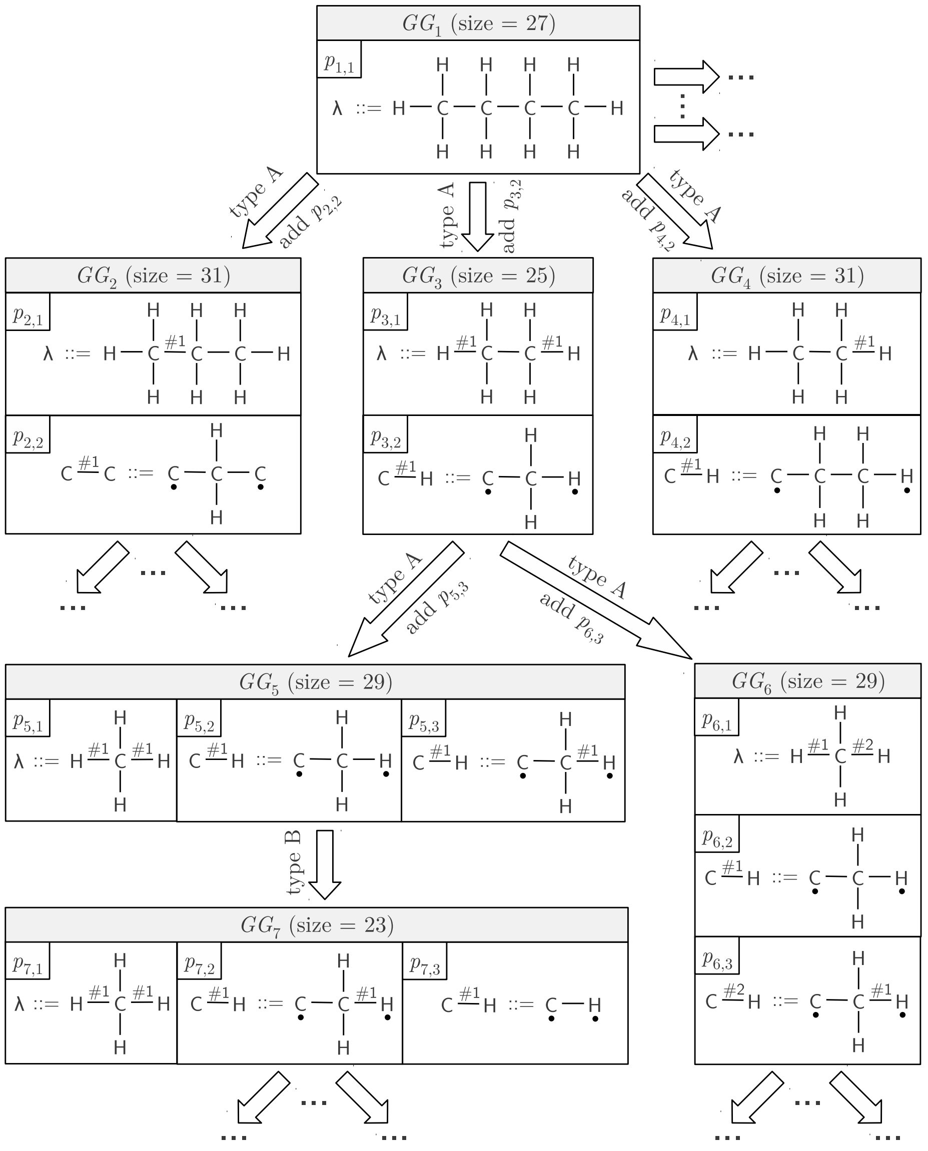

Fig. 2. Inducing a grammar from the structural formula of butane

```
procedure induceGrammar($\mathcal{G}^+$, $\mathcal{G}^-$, $beamWidth$, $maxVertexCount$)
    $GG_1$ := the grammar with productions $\lambda ::= G$ for each $G \in \mathcal{G}^+$;
    $GG_{\min} := GG_1$;
    $Queue := \{GG_1\}$;
    while $Queue \neq \emptyset$ do
        $GG$ := the smallest grammar from $Queue$;
        if $|GG| < |GG_{\min}|$ then $GG_{\min} := GG$ end;
        $Queue := Queue \setminus \{GG\}$;
        $NewProductions$ := findProductions($GG$, $maxVertexCount$);
        foreach $p \in NewProductions$ do
            $GG'$ := type-A generalization of $GG$ via the production $p$;
            if $GG'$ does not cover any graph from $\mathcal{G}^-$ then
                $GG''$ := type-B generalization of $GG'$ (if it exists);
                if $GG''$ exists and does not cover any graph from $\mathcal{G}^-$ then
                    $Queue := Queue \cup \{GG''\}$
                else $Queue := Queue \cup \{GG'\}$ end
            end
        end;
        retain the $beamWidth$ smallest grammars in $Queue$ and discard the others
    end;
    return $GG_{\min}$
end
```

Fig. 3. The induction algorithm

Type-A Generalization. A type-A generalization of a candidate grammar GG adds a new type-II production p to GG and reverse-applies it to all occurrences of $RHS(p)$ in the base graphs of GG, resulting in a new grammar GG'. In Fig. 2, the addition of the production $p_{3,2}$ to the grammar GG_1 results in the grammar GG_3. The grammar GG_3 is thus a type-A generalization of GG_1 via the production $p_{3,2}$.

How can we find a type-II production p to transform a grammar GG into GG'? Since the base graphs of GG' are obtained by reverse-applying p to the base graphs of GG, the base graphs of GG must contain at least one occurrence of $RHS(p)$, i.e., at least one subgraph of the form $[u(S)v]$. We thus search the base graphs of GG for all possible subgraphs of the form $[u(S)v]$. Each such subgraph is an occurrence of the RHS of some type-II production, and each such production is eligible to enrich the grammar GG. Since we cannot determine the 'best' type-II production in advance, we have to consider all such productions, and thus we obtain many possible type-A generalizations of GG.

To simplify the explanation, let us first focus on undirected graphs and grammars. The process of finding type-II productions to generalize an undirected grammar GG is outlined in Fig. 4. The auxiliary procedure findSubgraphs (omitted for lack of space) finds all subgraphs comprising up to $maxVertexCount$ vertices in the set of base graphs of GG. More precisely, the procedure creates a set of pairs ($Subgraph$, $Occurrences$), where $Subgraph$ is a graph and $Occurrences$

is a set of pairs (*Match*, *Host*) such that *Match* is an occurrence of the graph *Subgraph* in the graph $Host \in \mathcal{B}(GG)$. The procedure findSubgraphs first finds all single-vertex subgraphs and then iteratively produces larger subgraphs as single-vertex extensions of individual occurrences of smaller subgraphs. This approach was inspired by a large selection of algorithms sharing similar goals [6], especially by the VSiGraM approach [14], which could be used in place of it.

```
1  procedure findProductions($GG$, $maxVertexCount$)
2      $SubsAndOccs := \text{findSubgraphs}(\mathcal{B}(GG), maxVertexCount)$;
3      $Productions := \emptyset$;
4      foreach $(Subgraph, Occurrences) \in SubsAndOccs$ do
5          foreach $(Match, Host) \in Occurrences$ do
6              if $|Nh_{Host}(Match)| = 2$ then
7                  call the two vertices in $Nh_{Host}(Match)$ $u$ and $v$;
8                  $\mathsf{A} := label(u)$; $\mathsf{B} := label(v)$;
9                  $\mathcal{I} := \{i \mid [\mathsf{A} \overset{\#i}{-\!\!-\!\!-} \mathsf{B}]$ is a subgraph in $GG\}$;
10                 if $\mathcal{I} = \emptyset$ then $k := 1$ else $k := \max(\mathcal{I}) + 1$ end;
11                 foreach $i \in \mathcal{I} \cup \{k\}$ do
12                     $Productions := Productions \cup \{[\mathsf{A} \overset{\#i}{-\!\!-\!\!-} \mathsf{B}] ::= [\mathsf{A}(Subgraph)\mathsf{B}]\}$
13                 end
14             end
15         end
16     end;
17     return $Productions$
18 end;
```

Fig. 4. Finding eligible type-II productions to generalize an undirected grammar GG

After receiving a set of subgraph-occurrence pairs, the procedure findProductions searches this set for all subgraphs that can serve as possible production *cores* (not entire RHSs!). For a subgraph S to serve as the core of a production, S must have exactly two neighbors in the base graph in which it occurs. Such a subgraph S gives rise to a type-II production p: $[\mathsf{A} \overset{\#i}{-\!\!-\!\!-} \mathsf{B}] ::= [\mathsf{A}(S)\mathsf{B}]$, where $\#i$ could stand for any nonterminal label. If the graph $[\mathsf{A} \overset{\#i}{-\!\!-\!\!-} \mathsf{B}]$ already occurs as a subgraph somewhere in the grammar, then the production p actually generalizes the grammar GG; otherwise, GG is merely restructured. To take both possibilities into account, we create a separate production $[\mathsf{A} \overset{\#i}{-\!\!-\!\!-} \mathsf{B}] ::= [\mathsf{A}(S)\mathsf{B}]$ for each i such that $[\mathsf{A} \overset{\#i}{-\!\!-\!\!-} \mathsf{B}]$ occurs as a subgraph in GG and for a single value of i that does not meet this condition (lines 9–13 in Fig. 4). For example, the grammar GG_3 in Fig. 2 is extended by the productions $p_{5,3}$ (giving the grammar GG_5) and $p_{6,3}$ (giving the grammar GG_6), which differ only in their LHS edge labels.

In the case of directed graphs and grammars, a production core $[\mathsf{A}(S)\mathsf{B}]$ can serve as the RHS in two distinct production families, namely $[\mathsf{A} \xrightarrow{\#i} \mathsf{B}] ::= [\mathsf{A}(S)\mathsf{B}]$ and $[\mathsf{B} \xrightarrow{\#i} \mathsf{A}] ::= [\mathsf{B}(S)\mathsf{A}]$. Since neither of these families can be considered preferable in advance, both should be added to the resulting production set. The family $[\mathsf{A} \xrightarrow{\#i} \mathsf{B}] ::= [\mathsf{A}(S)\mathsf{B}]$ contains a production for each i such that $[\mathsf{A} \xrightarrow{\#i} \mathsf{B}]$ occurs as a subgraph in GG and for a single value of i that does not meet this condition. The other family is defined in an analogous fashion.

After each type-A generalization step, the resulting grammar is simplified by reverse-applying its type-II productions to its base graphs wherever possible and as many times as possible. This procedure is not essential for the induction process, but can make the grammar considerably smaller.

Type-B Generalization. A type-B generalization of a candidate grammar GG replaces two 'similar' productions of GG with a set of new productions, giving a grammar GG' that is at least as general as GG. In Fig. 2, the grammar GG_5 is generalized to GG_7 by replacing the productions $p_{5,2}$ and $p_{5,3}$ with $p_{7,2}$ and $p_{7,3}$. The notion of 'similar' productions is based on the concept of *unifiability*.

Edge labels l and m are *unifiable* (denoted $l \cong m$) if $(l = m) \vee (l \in \mathcal{N}^E) \vee (m \in \mathcal{N}^E)$. The *unification* of unifiable edge labels l and m (denoted $\mathit{unif}(l, m)$) is the label l if $l \in \mathcal{N}^E$; otherwise, $\mathit{unif}(l,\, m) = m$. Graphs G and H are *unifiable* if there exists a *unifying isomorphism* $h\colon G \to H$, i.e., a bijective vertex-to-vertex and edge-to-edge mapping such that $\mathit{label}(h(v)) = \mathit{label}(v)$, $\mathit{conn}(h(e)) = h(\mathit{conn}(e))$, and $\mathit{label}(h(e)) \cong \mathit{label}(e)$ for all $v \in V_G$ and $e \in E_G$. The *unification* of such graphs G and H (denoted $\mathit{unif}(G, H)$) is a graph obtained from G by setting $\mathit{label}(e) := \mathit{unif}(\mathit{label}(e), \mathit{label}(h(e)))$ for all edges $e \in G$.

Type-B generalization can be applied to a pair of directed productions p and q if they take the form $p\colon [u\colon \mathsf{A} \xrightarrow{e\colon \#i} v\colon \mathsf{B}] ::= [u(S)v]$ and $q\colon [u'\colon \mathsf{A} \xrightarrow{e'\colon \#i} v'\colon \mathsf{B}] ::= [u'(S')v']$ and if there exists a unifying isomorphism $h\colon \mathit{RHS}(p) \to \mathit{RHS}(q)$ such that $h(u) = u'$ and $h(v) = v'$. A type-B generalization step replaces the productions p and q with a set that comprises: (1) a production $[\mathsf{A} \xrightarrow{\#i} \mathsf{B}] ::= [\mathsf{A}(S'')\mathsf{B}]$, where $S'' = \mathit{unif}(S, S')$ (let $g\colon S'' \to S$ and $g'\colon S'' \to S'$ denote the corresponding unifying isomorphisms); (2) a production $[\mathsf{P} \xrightarrow{\#j} \mathsf{Q}] ::= [\mathsf{P} \xrightarrow{\mathsf{r}} \mathsf{Q}]$ for each edge e of S'' such that $\mathit{label}(\mathit{conn}(e)) = (\mathsf{P}, \mathsf{Q})$, $\mathit{label}(e) = \#j$, and $\mathit{label}(g(e)) = \mathsf{r} \vee \mathit{label}(g'(e)) = \mathsf{r}$. Undirected productions are treated in an analogous manner.

5 Experimental Results

5.1 Application to Flowcharts

In our first series of experiments, we applied the induction algorithm to various sets of valid flowchart graphs. Our goal was to induce a grammar that generates (a superset of) the language generated by the reference grammar in Fig. 1. We experimented with different sets of randomly generated flowchart graphs and different input parameters. Each input set comprised between 10 and 50 graphs with up to 25 vertices. Different sets gave rise to different grammars, but in many cases, the algorithm induced the grammar shown in Fig. 5 or some variation thereof. The productions p_1 through p_8 in Fig. 5 are equivalent to the productions of the reference grammar despite the fact that all nonterminal edges are reversed. (Note the positions of the markers ‘S’ and ‘T’; in every type-II production, the source vertex on the LHS coincides with the bottom vertex on the RHS.) The induced grammar can therefore generate any valid flowchart.

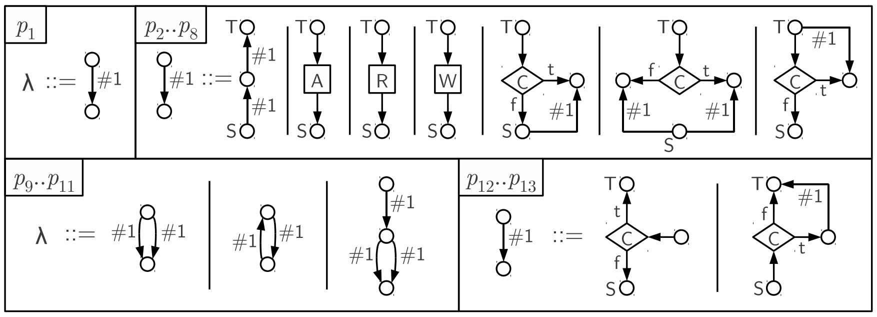

Fig. 5. A grammar induced from various sets of valid flowcharts

5.2 Application to Chemical Structural Formulas

In our second series of experiments, we tried to induce a grammar of linear hydrocarbons with single and double bonds (LHSDB). This graph language comprises the structural formulas of chemical compounds consisting of carbon atoms (vertices labeled C) and hydrogen atoms (vertices labeled H). The carbon atoms form a chain connected with single and double bonds (edges). The hydrogen atoms are connected to the carbon atoms by means of single bonds so that every carbon atom has exactly four incident bonds. Some positive and negative examples of the LHSDB language are shown in Fig. 8. Our reference grammar for this language is depicted in Fig. 6.

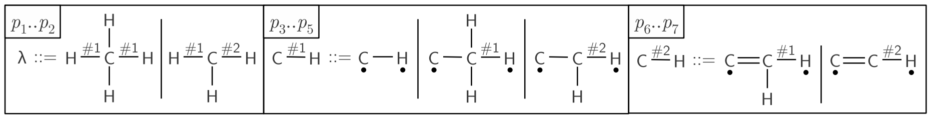

Fig. 6. The reference grammar for the LHSDB language

To make the induction problem more challenging, the induced grammar was required to cover *only* valid (though not necessarily linear) hydrocarbons. We thus demanded that the induced grammar cover all valid LHSDB graphs and that every graph covered by the induced grammar represent a valid hydrocarbon.

Experiments showed that such a grammar probably cannot be induced from positive graphs alone. Moreover, a correct grammar could not be induced from 'almost any' pair of input sets. A favorable combination of positive and negative input graphs had to be sought more systematically.

To determine whether a correct LHSDB grammar can be induced and from what set of examples this can be achieved, we prepared a set of 42 positive examples ($\mathcal{G}_0^+$) and a set of 200 negative examples ($\mathcal{G}_0^-$). The positive set comprised all correct LHSDB graphs with up to 6 carbon vertices. The negative set was obtained by randomly removing one or two hydrogen atoms in correct LHSDB graphs with up to four carbon vertices. We then tried to find such subsets $\mathcal{S}^+ \subseteq \mathcal{G}_0^+$ and $\mathcal{S}^- \subseteq \mathcal{G}_0^-$ that the grammar induced from them would cover all graphs from $\mathcal{G}_0^+$ and none from $\mathcal{G}_0^-$. The sets $\mathcal{S}^+$ and $\mathcal{S}^-$ were obtained by a simple procedure shown in Fig. 7. The resulting sets are displayed in Fig. 8, and the grammar induced from them (using $\mathit{beamWidth} = 10$ and $\mathit{maxVertexCount} = 5$) is shown in Fig. 9. The size of the induced grammar equals 59. For comparison, the size of the reference grammar (Fig. 6) amounts to 54.

```
procedure findInputExamples($\mathcal{G}_0^+$, $\mathcal{G}_0^-$, $beamWidth$, $maxVertexCount$)
    $\mathcal{S}^+ := \{$the smallest graph in $\mathcal{G}_0^+\}$;
    $\mathcal{S}^- := \emptyset$;
    $GG :=$ induceGrammar($\mathcal{S}^+$, $\mathcal{S}^-$, $beamWidth$, $maxVertexCount$);
    $Missed^+ := \{G \in \mathcal{G}_0^+ \mid GG$ does not cover $G\}$;
    $Missed^- := \{G \in \mathcal{G}_0^- \mid GG$ covers $G\}$;
    while $(Missed^+ \neq \emptyset) \vee (Missed^- \neq \emptyset)$ do
        if $Missed^- \neq \emptyset$ then $\mathcal{S}^- := \mathcal{S}^- \cup \{$the smallest graph from $Missed^-\}$
        else $\mathcal{S}^+ := \mathcal{S}^+ \cup \{$the smallest graph from $Missed^+\}$ end;
        $GG :=$ induceGrammar($\mathcal{S}^+$, $\mathcal{S}^-$, $beamWidth$, $maxVertexCount$);
        $Missed^+ := \{G \in \mathcal{G}_0^+ \mid GG$ does not cover $G\}$;
        $Missed^- := \{G \in \mathcal{G}_0^- \mid GG$ covers $G\}$
    end;
    return $(\mathcal{S}^+, \mathcal{S}^-)$
end
```

Fig. 7. Extraction of a pair of small favorable input graph sets ($\mathcal{S}^+$ and $\mathcal{S}^-$) from a pair of larger disjoint graph sets ($\mathcal{G}_0^+$ and $\mathcal{G}_0^-$)

The grammar of Fig. 9 meets the requirements stated above. By mathematical induction on the length of carbon vertex chains, we could prove that every valid LHSDB graph can be generated by the induced grammar. To prove that the grammar generates only valid hydrocarbon graphs, we would have to show that every vertex introduced by the grammar eventually obtains the correct number of incident edges (four in the case of carbon vertices and one in the case of hydrogen vertices). To see this, consider that any subgraph $\mathsf{C} \overset{\#1}{\text{——}} \mathsf{H}$ expands into $\mathsf{C}-(\ldots)-\mathsf{H}$ and that any subgraph $\mathsf{C} \overset{\#2}{\text{——}} \mathsf{H}$ expands into $\mathsf{C}=(\ldots)-\mathsf{H}$.

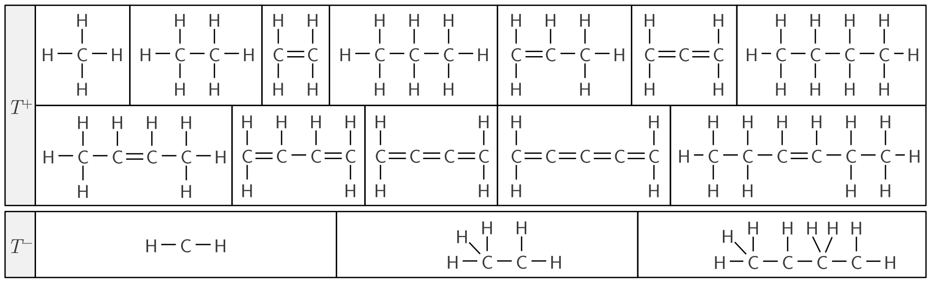

Fig. 8. The positive input set ($\mathcal{S}^+$) and the negative input set ($\mathcal{S}^-$) for the induction of an LHSDB grammar

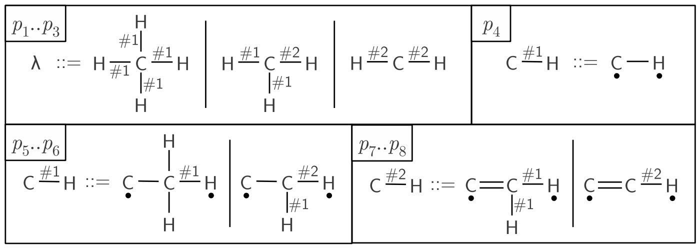

Fig. 9. The grammar induced from the input sets of Fig. 8

Given the input sets $\mathcal{S}^+$ and $\mathcal{S}^-$ of Fig. 8, the induction algorithm was shown to be robust to the parameters *beamWidth* and *maxVertexCount*, provided that *beamWidth* ≥ 1 and *maxVertexCount* ≥ 3. The values 1 and 2 for *maxVertexCount* cannot possibly produce any meaningful results, since the production p_5 in Fig. 9, which seems to be an indispensable part of any valid grammar, has three vertices in its core. We systematically varied both parameters and ran the induction algorithm for each pair of values. The resulting grammars were tested on a set of positive and negative graphs disjoint from $\mathcal{G}_0^+$ and $\mathcal{G}_0^-$.

5.3 Computational Complexity

Owing to the exhaustive subgraph enumeration procedure, which is the basis of type-A generalization, and to the Rekers-Schürr parser, our algorithm has exponential worst-case complexity in terms of time and memory consumption. The complexity of subgraph search could be reduced at the cost of missing some subgraphs. For example, the approaches of Jonyer et al. [11], Kukluk et al. [13], and Ates et al. [2] place an upper limit on the number of created subgraphs and hence run in a polynomial time and space at the cost of suboptimal results. The improved version of the Rekers-Schürr parser runs in polynomial time and space for many grammars [8], but its worst-case complexity is still exponential. This fact should not come as a surprise, since the problem of graph grammar parsing is NP-hard even for very restricted classes of grammars [19].

Table 1 shows the performance of our induction algorithm on the input graph set of Fig. 8 with respect to the parameters *beamWidth* and *maxVertexCount*. We measured the number of generated candidate grammars and the total execution time of the algorithm on an 1.86-GHz Intel Core 2 Duo machine. The results for different values of *beamWidth* (with *maxVertexCount* fixed at 5) are shown on the left side of the table, and those for different values of *maxVertexCount* (with *beamWidth* = 10) are displayed on the right side.

Figure 10 shows how the execution time depends on the number of input examples. To obtain the left chart, the number of negative examples was fixed at 200, and the number of positive examples was varied from 1 to 42 in the order of increasing graph size. To draw the right chart, the number of positive examples was fixed at 42, and the number of negative examples was varied from 1 to 200 in no particular order. In both cases, the input examples were drawn from the set of 42 positive and 200 negative examples that were supplied to the procedure of Fig. 7 when searching for a favorable input set for hydrocarbons. The parameters *beamWidth* and *maxVertexCount* were fixed at 10 and 5, respectively.

Table 1. The number of generated grammars and the total execution time with respect to the parameters *beamWidth* and *maxVertexCount*

	beamWidth				*maxVertexCount*			
	1	10	100	1000	3	5	7	9
Number of generated grammars	116	148	590	24 435	95	148	195	224
Execution time (in seconds)	6.8	7.2	12.2	370	3.4	7.2	12.4	17.5

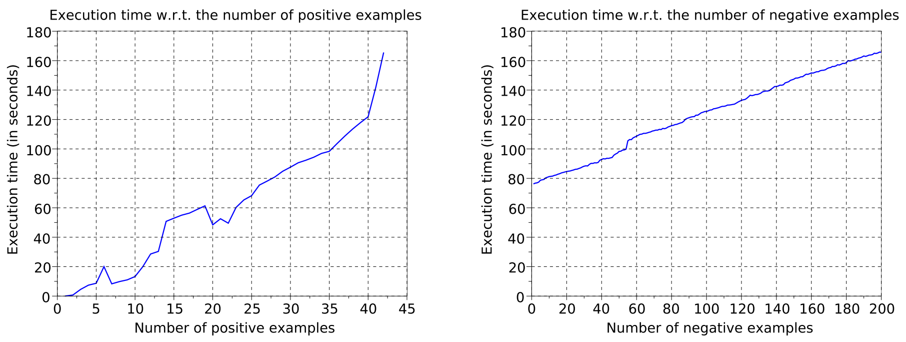

Fig. 10. Total execution time with respect to the number of input examples

The algorithm takes a little less than three minutes to finish if provided with the entire set of 42 positive and 200 negative examples. However, the user is not required to wait until the algorithm halts in order to obtain a meaningful grammar. Since the algorithm generates *only* grammars that are consistent with the input set, its result (the current minimum grammar) is valid at any point during its execution. The longer the algorithm runs, the smaller and the more general grammars it produces, but *all* induced grammars are valid with respect to the input set.

6 Conclusion

We have presented a novel graph grammar induction algorithm. Given a pair of disjoint graph sets, $\mathcal{G}^+$ and $\mathcal{G}^-$, the algorithm tries to find the smallest grammar that covers all graphs from $\mathcal{G}^+$ and none from $\mathcal{G}^-$. The induction process is realized as a parser-controlled specific-to-general search. We applied the proposed method to two meaningful and nontrivial graph languages. The algorithm exhibited a surprising inductive power when provided with favorable input.

In our 'chemical' example, a favorable input set was found by the algorithm in Fig. 7, which requires a pair of (large) initial input sets. To make the input selection process more 'user-friendly', we are working on a tool with the following interaction scenario: First, the user prepares a (small) set of positive input graphs. The tool induces a grammar from this set and generates a set of random graphs covered by the induced grammar. The user can then visually inspect the generated graphs and add to the negative input set all those that do not belong to the target language. If there are no such graphs, he or she may prepare some additional positive graphs and, by the help of the built-in parser, add to the positive input set all those graphs that are not covered by the induced grammar. After that, the tool induces a new grammar based on the updated input sets. The process repeats until the user is satisfied with the induced grammar.

At present, our research is focused on more general target grammar formalisms. In the formalism presented in this paper, a grammar for arbitrary hydrocarbons most probably does not exist. In the unrestricted LGG formalism, such a grammar comprises four simple productions (see the grammar GG_{HC} in Fig. 3 in [8]).

References

1. Aschenbrenner, N., Geiger, L.: Transforming scene graphs using triple graph grammars — a practice report. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 32–43. Springer, Heidelberg (2007)
2. Ates, K., Kukluk, J.P., Holder, L.B., Cook, D.J., Zhang, K.: Graph grammar induction on structural data for visual programming. In: 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2006). pp. 232–242. IEEE Computer Society, Washington, DC (2006)
3. Balogh, Z., Varró, D.: Model transformation by example using inductive logic programming. Software and Systems Modeling 8(3), 347–364 (2009)
4. Brijder, R., Blockeel, H.: On the inference of non-confluent NLC graph grammars. Journal of Logic and Computation (2012), to appear

5. Buchmann, T., Dotor, A., Uhrig, S., Westfechtel, B.: Model-driven software development with graph transformations: A comparative case study. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 345–360. Springer, Heidelberg (2007)
6. Cook, D.J., Holder, L.B.: Mining Graph Data. John Wiley & Sons, New Jersey (2006)
7. Dubey, A., Jalote, P., Aggarwal, S.K.: Learning context-free grammar rules from a set of programs. IET Software 2(3), 223–240 (2008)
8. Fürst, L., Mernik, M., Mahnič, V.: Improving the graph grammar parser of Rekers and Schürr. IET Software 5(2), 246–261 (2011)
9. Javed, F., Mernik, M., Gray, J., Bryant, B.R.: MARS: A metamodel recovery system using grammar inference. Information and Software Technology 50(9–10), 948–968 (2008)
10. Jeltsch, E., Kreowski, H.J.: Grammatical inference based on hyperedge replacement. In: Ehrig, H., Kreowski, H.J., Rozenberg, G. (eds.) 4th International Workshop on Graph-Grammars and Their Application to Computer Science. LNCS, vol. 532, pp. 461–474. Springer, London (1991)
11. Jonyer, I., Holder, L.B., Cook, D.J.: MDL-based context-free graph grammar induction and applications. International Journal of Artificial Intelligence Tools 13(1), 65–79 (2004)
12. Kong, J., Zhang, K., Zeng, X.: Spatial graph grammars for graphical user interfaces. ACM Transactions on Computer-Human Interaction 13(2), 268–307 (2006)
13. Kukluk, J.P., Holder, L.B., Cook, D.J.: Inferring graph grammars by detecting overlap in frequent subgraphs. Applied Mathematics and Computer Science 18(2), 241–250 (2008)
14. Kuramochi, M., Karypis, G.: Finding frequent patterns in a large sparse graph. Data Mining and Knowledge Discovery 11(3), 243–271 (2005)
15. Nakamura, K., Matsumoto, M.: Incremental learning of context free grammars based on bottom-up parsing and search. Pattern Recognition 38(9), 1384–1392 (2005)
16. Parekh, R., Honavar, V.: Grammar inference, automata induction, and language acquisition. In: Dale, R., Somers, H.L., Moisl, H. (eds.) Handbook of Natural Language Processing. pp. 727–764. Marcel Dekker, New York (2000)
17. Plasmeijer, M.J., van Eekelen, M.C.J.D.: Term graph rewriting and mobile expressions in functional languages. In: Nagl, M., Schürr, A., Münch, M. (eds.) AGTIVE 1999. LNCS, vol. 1779, pp. 1–13. Springer, Heidelberg (1999)
18. Rekers, J., Schürr, A.: Defining and parsing visual languages with Layered Graph Grammars. Journal of Visual Languages and Computing 8(1), 27–55 (1997)
19. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. World Scientific, River Edge, NJ, USA (1997)
20. Schürr, A.: Specification of graph translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) 20th International Workshop on Graph-Theoretic Concepts in Computer Science. LNCS, vol. 903, pp. 151–163. Springer, Berlin (1995)