Edge-based graph grammar: theory and support system

Xiaoqin Zeng[1] Yufeng Liu[1] Zhan Shi[1] Yingfeng Wang[2] Yang Zou[1] Jun Kong[3] Kang Zhang[4]

[1]Institute of Intelligence Science and Technology, Hohai University, Nanjing, Jiangsu, China

[2]School of Information Technology, Middle Georgia State University, Macon, GA 31206, USA

[3]Department of Computer Science, North Dakota State University, Fargo, ND 58102, USA

[4]Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75080, USA

***Abstract*—As a useful formal tool, graph grammar provides a rigorous but intuitive way for defining graphical languages and analyzing graphs. This paper presents a new context-sensitive graph grammar formalism called Edge-based Graph Grammar or EGG, in which a new methodology is proposed to tackle issues, such as the embedding problem, the membership problem and the parsing algorithm. It presents the formal definitions of EGG and its language with a proof of its decidability. Then, a new parsing algorithm with an analyses of its computational complexity is given for checking the structural correctness or validity of a given host graph. The paper finally describes the development of an EGG support system with friendly GUI.**

Keywords-component; graph grammar; graphical language; embedding problem; parsing; production rule

I. INTRODUCTION

With the development of human-computer interaction techniques, graphical languages have been applied to various application domains, such as modeling visual interaction processes [1, 2], designing graphical user interface in multimedia applications [3], visual queries to databases [4], and defining the layout of a GUI in multimedia applications [3]. Conceptually, objects described by graphical languages can be abstracted as graphs consisting of nodes and edges. For the specification and analysis of these types of graphs, graph grammars [5, 6] are an ideal formal and intuitive tool.

It is well-known that formal string grammar lays a solid theoretical foundation for the definition and parsing of programming languages. For the same reason, graphical languages also need the corresponding formal graph grammars. Compared with string grammar, graph grammars set a theoretical basis to visual languages [7]. However, the implementation of a graphical language is usually not as easy as implementing string languages [8]. This is mostly due to the fact that the extension from one-dimensional string grammars to two-dimensional graph grammars raises new issues [9] such as the embedding problem, the membership problem, high parsing complexity.

There have been a number of graph grammars and their applications in the literature [10-27]. According to the type of grammatical productions, graph grammars could be mainly divided into two categories: context-free and context-sensitive. The main differences between the two are the production formation and the expressive power. On the one hand, a context-free grammar requires that only a single non-terminal node be allowed on the left-hand side of a production [16]. In early years, many context-free grammars were proposed [17-21]. Since the productions of these graph grammars are quite simple, their expressive power is limited, which hinders the scope of their applications. On the other hand, in response to the increasing demands of intricate graph-oriented applications, researchers have developed several context-sensitive graph grammars, such as PLC (picture layout grammar) [21], CMG (constrain multiset grammar) [22], LGG (layered graph grammar) [23], RGG (reserved graph grammar) [8], SGG (spatial graph grammar) [24, 25]. These context-sensitive graph grammars allow the left-hand side of a production to be a graph rather than a node, so bring more expressive power. LGG and RGG are the most representatives of context-sensitive graph grammars.

Rekers and Schürr [23] proposed a context-sensitive graph grammar formalism called *Layered Graph Grammar* (LGG) for defining and parsing graphical visual languages. First, productions in LGG differ widely from others by introducing context nodes that are not replaced in a derivation or reduction operation. Second, to solve the embedding problem, LGG puts a restriction on the definition of a redex in a host graph by requiring its nodes that are isomorphic to non-context nodes in productions can only link to other nodes in the host graph that are isomorphic to the context nodes in the productions. This restriction ensures no creation of dangling edges when a redex in a host graph is replaced. Third, a very intricate layer decomposition constraint is introduced to solve the membership problem.

Based and improved on LGG, Zhang et al. [8] proposed another context-sensitive grammar called *Reserved Graph*

DOI Reference Number: 10.18293/VLSS-028

Grammar (RGG), which defines the structure of graphs by introducing a two-level structure for each node as a super-vertex containing sub-vertices connected with edges. In addition, RGG introduces a marking mechanism to tackle the embedding problem, in which a unique label is used to identify all context elements. Further, with the introduction of selection-free productions to graph grammars, a Selection-Free Parsing Algorithm (SFPA) is designed for a selection-free RGG, which only needs to consider one parsing path and thus can efficiently parse graphs with polynomial time complexity [8]. Later on, Kong et al. [24, 25] extended RGG by introducing spatial notations and mechanisms. The spatial specifications of the extended RGG, called *Spatial Graph Grammar* (SGG), can qualitatively express the spatial relationships among objects and reduce the parsing complexity using the spatial information.

Both LGG and RGG have been applied widely to the definition, analysis and transformation of visual languages [28-37], such as Visual XML Schemas [29, 30], Design Pattern Evolution and Verification [32, 33], Generic Visual Language Generation Environments [28]. However, they still have deficiencies. For example, the LGG's context nodes and layer decomposition constraint make productions difficult to design. RGG's two-level node structure and marking mechanism are not intuitive and make them difficult to apply to general graphs.

This paper presents our work on the improvements over the existing graph grammars with the following contributions.

- A new context-sensitive graph grammar formalism called EGG, which uses edges instead of nodes to concisely express the context in productions for simply and efficiently solving the embedding problem.
- A size-increasing constraint applied to the structure of productions for solving the membership problem, easing the design of productions.
- A new general parsing algorithm for checking the structural correctness and validity of given host graphs; and the implementation of an EGG graph grammar support system, which provides friendly GUI for end users to design and apply graph grammars.

The rest of the paper is organized as follows. Section 2 presents graphical and grammatical preliminaries, introducing new terms used in Section 3, which gives the formal definitions of EGG and its language with a proof of its decidability. Section 4 presents a parsing algorithm and its complexity analysis. Section 5 describes the developed EGG support system. Finally, Section 6 concludes the paper.

II. Graphical and Grammatical Preliminaries

In node-edge graphs, a node typically represents an abstract object and an edge represents some kind of relationship between two connected nodes. Each node n in a node set N can be connected with none or more edges, and each edge e in an edge set E is only connected with two nodes. An edge can be directed or undirected depending on whether it has a direction between the two connected nodes. Because an undirected edge can be treated as two directed edges with reverse directions, without loss of generality this paper only considers directed edges.

In string grammars, labels play an important role as identifiers, and so do labels in graph grammars. Let L be a finite set of labels. Depending on the usage of a label, L can further be divided into terminal label set L_T, nonterminal label set L_{NT}, and mark label set L_M, namely $L = L_T \cup L_{NT} \cup L_M$, $L_T \cap L_{NT} = \Phi$, and $L_M \cap (L_T \cup L_{NT}) = \Phi$.

By combining the techniques of both graph theory and formal language, we introduce a series of new definitions and notations here.

Definition 2.1 $\mathrm{n} = (\mathrm{l})$ is a node with label l in a given finite label set L.

Definition 2.2 $\mathrm{e} = (\mathrm{n_s}, \mathrm{n_e})$ is a directed edge, where

- $\mathrm{n_s}$ is the start node of the directed edge;
- $\mathrm{n_e}$ is the end node of the directed edge.

Based on the above definitions of node and directed edge, we further introduce the following notations:

- $\mathrm{E_s}$ is a set of directed edges starting from a node;
- $\mathrm{E_e}$ is a set of directed edges ending to a node;
- $\mathrm{d(n)}$ is the degree indicating the number of directed edges connected to n, i.e. $\mathrm{d(n)} = |\mathrm{E_s} \cup \mathrm{E_e}|$;
- $\mathrm{d_s(n)}$ is the out-degree indicating the number of directed edges starting from n, i.e. $\mathrm{d_s(n)} = |\mathrm{E_s}|$;
- $\mathrm{d_e(n)}$ is the in-degree indicating the number of directed edges ending to n, i.e. $\mathrm{d_e(n)} = |\mathrm{E_e}|$.

Obviously, $d(n) = d_s(n) + d_e(n)$. For simplicity, notations like $n.l$ and $n.E_s$ express the corresponding components of node n, and are applicable to other definitions throughout this paper.

Unlike an undirected edge, a directed edge needs to distinguish start node and end node. Besides, an edge may also carry a label for clear identification.

Definition 2.3 $\mathrm{G} = (\mathrm{N}, \mathrm{E})$ is a graph on given label set L, where

- N is a node set that is associated with a two-way partition into $\mathrm{N_T}$ and $\mathrm{N_{NT}}$, the elements of $\mathrm{N_T}$ are called terminal nodes and the elements of $\mathrm{N_{NT}}$ are called non-terminal nodes;
- E is a directed edge set with $\mathrm{E} \subseteq \mathrm{N} \times \mathrm{N}$.

We then have the following mappings for mathematically expressing grammatical items.

- $\mathrm{f_{NL}}: \mathrm{N} \to \mathrm{L}$, a mapping from node n to label $\mathrm{l} \in \mathrm{L}$, i.e., $\mathrm{f_{NL}(n)} = \mathrm{n.l}$;
- $\mathrm{f_{EN_s}}: \mathrm{E} \to \mathrm{N}$, a mapping from directed edge e to its start node, i.e., $\mathrm{f_{EN_s}(e)} = \mathrm{e.n_s}$;
- $\mathrm{f_{EN_e}}: \mathrm{E} \to \mathrm{N}$, a mapping from directed edge e to its end node, i.e., $\mathrm{f_{EN_e}(e)} = \mathrm{e.n_e}$.

In EGG, dangling edge set $\dot{E}$ is introduced to represent contexts, in which each edge is connected with only one node being either a start or end node, namely $\dot{E} = \dot{E}_s \cup \dot{E}_e$ with $\dot{E}_s = \{\dot{e}_s | \dot{e}_s = (n_s, \emptyset)\}$, $\dot{E}_e = \{\dot{e}_e | \dot{e}_e = (\emptyset, n_e)\}$ and $\dot{E}_s \cap \dot{E}_e = \Phi$. In addition to dangling edges, a marking mechanism is also introduced to mark dangling edges. The concepts of dangling edge and marking mechanism solves the embedding problem in EGG. Fig. 1 illustrates a graph including dangling edges with

$\dot{E} = \{1,2,3\}$. The graph is called a *dangling edge graph* and can be defined as follows.

Definition 2.4 $\overline{\mathrm{G}} = (\mathrm{N}, \overline{\mathrm{E}}, \mathrm{M})$ is a *dangling edge graph* on given label set L, in which,

- N is a node set;
- $\overline{\mathrm{E}}$ is an edge set including dangling edges, which is associated with a two-way partition into E and $\dot{\mathrm{E}}$;
- $\mathrm{M} \subseteq \mathrm{L_M}$ is a mark set for marking dangling edges to distinguish different contexts.

Essentially, $\overline{\mathrm{G}}$ is an extension of G by introducing dangling edge and G can be regarded as a special case of $\overline{\mathrm{G}}$. Similarly, there is an extra mapping as follows.

- $\mathrm{f_{EM}}: \dot{\mathrm{E}} \to \mathrm{M}$, an injective mapping from dangling edge $\dot{\mathrm{e}}$ to its mark m, i.e., $\mathrm{f_{EM}}(\dot{\mathrm{e}}) = \mathrm{m}$.

Note that dangling edge set $\dot{\mathrm{E}}$ may be empty, which leads to the empty corresponding mark set *M* and mapping f_{EM}. Based on the above defined dangling edge graph, a grammatical production can be defined as follows.

Definition 2.5 A production p is the expression $\overline{\mathrm{G}}_\mathrm{L} ::= \overline{\mathrm{G}}_\mathrm{R}$, which consists of a left dangling edge graph $\overline{\mathrm{G}}_\mathrm{L}$ and right dangling edge graph $\overline{\mathrm{G}}_\mathrm{R}$ satisfying $\overline{\mathrm{G}}_\mathrm{L}.\mathrm{M} = \overline{\mathrm{G}}_\mathrm{R}.\mathrm{M}$.

In a production, dangling edges represent contexts and each pair of corresponding dangling edges between the left and right graphs are labeled by a unique mark to maintain their corresponding relationship. Using dangling edges and their corresponding marks, the replacement of a redex by either a left or right graph in a production can be done without ambiguity. In some special cases, a wildcard dangling edge is needed to represent an arbitrary number of edges, e.g., one entity may be connected with any number of attributes in an entity relationship diagram. For simplicity and without generality, the concept of wildcard edge is not discussed here. Fig. 2 is an example of a set of EGG productions specifying a process flow diagram with $\{\mathrm{begin}, \mathrm{assign}, \mathrm{fork}, \mathrm{join}, \mathrm{send}, \mathrm{receive}, \mathrm{if}, \mathrm{endif}\} \subseteq \mathrm{L_T}$ and $\{\mathrm{stat}\} \subseteq \mathrm{L_{NT}}$.

The function of a production is to transform a graph to another graph. However, the transformation needs to satisfy some conditions in which isomorphism is fundamental.

Definition 2.6 Graphs G and Q are isomorphic, denoted as $\mathrm{G} \approx \mathrm{Q}$, $\mathrm{f_{NL}}$ and $\mathrm{f_{NL}}'$ are two mappings for G and Q respectively, if and only if there exist two bijective mappings $\mathrm{f_{NN}}: \mathrm{G.N} \leftrightarrow \mathrm{Q.N}$ and $\mathrm{f_{EE}}: \mathrm{G.E} \leftrightarrow \mathrm{Q.E}$, and the following are satisfied:

- $\forall \mathrm{n}(((\mathrm{n} \in \mathrm{G.N}) \vee (\mathrm{n} \in \mathrm{Q.N})) \to (\mathrm{f_{NL}}(\mathrm{n}) = \mathrm{f_{NL}}'(\mathrm{f_{NN}}(\mathrm{n}))))$;
- $\forall \mathrm{e}(((\mathrm{e} \in \mathrm{G.E}) \vee (\mathrm{e} \in \mathrm{Q.E})) \to (\mathrm{f_{NN}}(\mathrm{f_{EN_s}}(\mathrm{e})) = \mathrm{f_{EN_s}}(\mathrm{f_{EE}}(\mathrm{e}))) \wedge (\mathrm{f_{NN}}(\mathrm{f_{EN_e}}(\mathrm{e})) = \mathrm{f_{EN_e}}(\mathrm{f_{EE}}(\mathrm{e}))))$.

An isomorphism between two graphs means that their corresponding nodes have the same label, and the same out-degree and in-degree. In addition, the corresponding edges have the same start and end nodes.

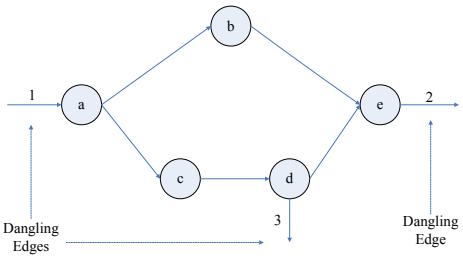


Figure 1. A dangling edge graph

Definition 2.7 Graph Q is the sub-graph of G, denoted as $\mathrm{Q} \in \mathrm{Sub}(\mathrm{G})$, if and only if the following are satisfied:

- $(\mathrm{Q.N} \subseteq \mathrm{G.N}) \wedge (\mathrm{Q.E} \subseteq \mathrm{G.E})$.

Graph Q is a sub-graph of G means that Q is part of G.

Definition 2.8 Graph Q is the core graph of $\overline{\mathrm{G}}$, denoted as $\mathrm{Q} = \mathrm{Cor}(\overline{\mathrm{G}})$, if and only if the following is satisfied:

- $(\mathrm{Q.N} = \overline{\mathrm{G}}.\mathrm{N}) \wedge (\mathrm{Q.E} = (\overline{\mathrm{G}}.\overline{\mathrm{E}} - \overline{\mathrm{G}}.\dot{\mathrm{E}})) \wedge (\mathrm{Q.L} = \overline{\mathrm{G}}.\mathrm{L})$.

Core graph Q is the sub-graph of graph $\overline{\mathrm{G}}$ obtained by removing all dangling edges from graph $\overline{\mathrm{G}}$ and keeping all the nodes and non-dangling edges of graph $\overline{\mathrm{G}}$. The graph in Fig. 3 is the core graph of that in Fig. 1.

Definition 2.9 If graph Q is a sub-graph of graph G and may include dangling edges, and $\overline{\mathrm{G}}_{\mathrm{L|R}}$ is a graph being left or right side of a production, Q is a redex of G with respect to $\overline{\mathrm{G}}_{\mathrm{L|R}}$, denoted as $\mathrm{Q} \in \mathrm{Redex}(\mathrm{G}, \overline{\mathrm{G}}_{\mathrm{L|R}})$, if and only if there exits bijective mappings $\mathrm{f_{NN}}: \mathrm{Q.N} \leftrightarrow \overline{\mathrm{G}}_{\mathrm{L|R}}.\mathrm{N}$ and $\mathrm{f_{EE}}: \mathrm{Q.E} \leftrightarrow \overline{\mathrm{G}}_{\mathrm{L|R}}.\mathrm{E}$, and the following are satisfied:

- $\mathrm{Cor}(\mathrm{Q}) \approx \mathrm{Cor}(\overline{\mathrm{G}}_{\mathrm{L|R}})$;
- $\forall \mathrm{n}((\mathrm{n} \in \mathrm{Q}) \to (\mathrm{d_s}(\mathrm{n}) = \mathrm{d_s}(\mathrm{f_{NN}}(\mathrm{n}))) \wedge (\mathrm{d_e}(\mathrm{n}) = \mathrm{d_e}(\mathrm{f_{NN}}(\mathrm{n}))))$.

To explain the above definition, we provide an example in the following three figures. Fig. 4 is graph $\overline{\mathrm{G}}_{\mathrm{L|R}}$, and Fig. 5 is a given host graph G. Obviously, graph Q in Fig. 6 is the sub-graph of G. According to *Definition 2.9*, Q is a redex of G with respect to $\overline{\mathrm{G}}_{\mathrm{L|R}}$.

In host graph G, if there is sub-graph Q being the redex of G with respect to $\overline{\mathrm{G}}_{\mathrm{L|R}}$ that is a left or right side graph of a production, then one could use the right or left side graph of the production to replace Q in G. This process is called *graph transformation* or *replacement*, as formally defined below.

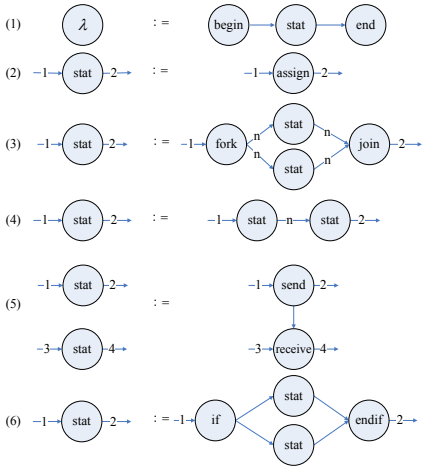


Figure 2. A set of EGG productions

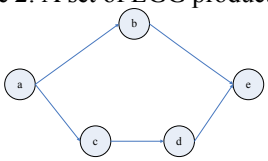


Figure 3. The core graph of the graph in Figure 1

Definition 2.10 An *L-application* to graph G is a transformation that generates graph G' using production p: $\overline{G}_L \coloneqq \overline{G}_R$, denoted as $G' = \mathrm{Tr}(G, Q, \overline{G}_L, \overline{G}_R)$, where $Q \in \mathrm{Redex}(G, \overline{G}_L)$, and $\mathrm{Cor}(\overline{G}_R)$ is used to replace Q in G. The L-application is also called *derivation* operation and denoted as $G \rightarrow^{p} G'$.

If a sequence of L-applications for graph G is: $G \rightarrow^{p_1} {G_1}', {G_1}' \rightarrow^{p_2} {G_2}', \ldots, {G_{n-1}}' \rightarrow^{p_n} {G_n}'$, then $G \rightarrow^{*} {G_n}'$ can be used to concisely express this process.

Definition 2.11 An *R-application* to graph G is a transformation that generates graph G'' using production p: $\overline{G}_L \coloneqq \overline{G}_R$, denoted as $G'' = \mathrm{Tr}(G, Q, \overline{G}_R, \overline{G}_L)$, where $Q \in \mathrm{Redex}(G, \overline{G}_R)$, and $\mathrm{Cor}(\overline{G}_L)$ is used to replace Q in G. The R-application is also called *reduction* operation and denoted as $G \mapsto^{p} G''$.

Similar to L-applications, a sequence of R-applications, which is $G \mapsto^{p_1} {G_1}'', {G_1}'' \mapsto^{p_2} {G_2}'', \ldots, {G_{n-1}}'' \mapsto^{p_n} {G_n}''$, can be expressed as $G \mapsto^{*} {G_n}''$.

Fig. 7 shows a derivation process from an initial graph using the productions in Fig. 2.

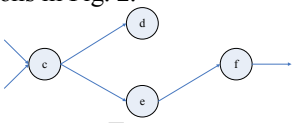


Figure 4. A graph $\overline{G}_{L|R}$ with dangling edges

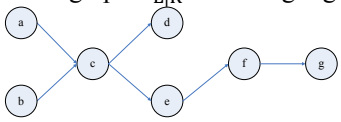


Figure 5. A host graph G

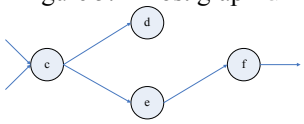


Figure 6. The sub-graph Q is a redex of G with respect to $\overline{G}_{L|R}$

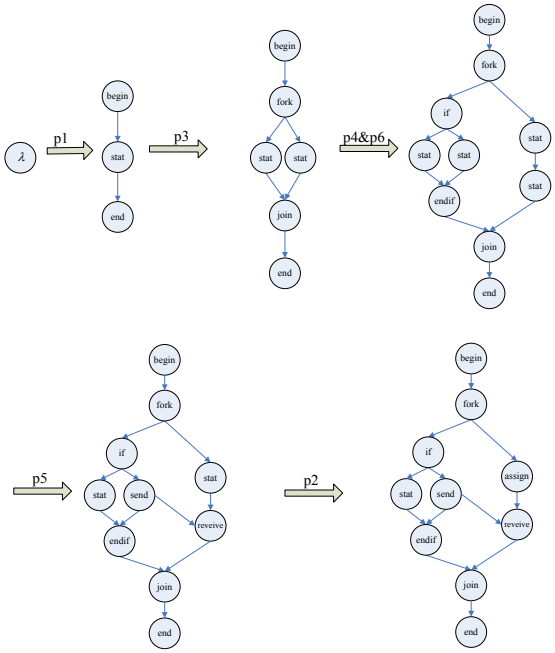


Figure 7. A graph L-application process using EGG productions

The formal definition of EGG and its language are discussed below.

III. An Edge-based Graph Grammar Formalism

To solve the embedding and membership problems, EGG employs edges rather than nodes in the two sides of a production to directly express contexts and introduces a size-increasing constraint to ensure the decidability of EGG.

3.1 Definition of EGG and its language

Based on the definitions in Section 2.1, an edge-based context-sensitive graph grammar formalism and its language can be defined as follows.

Definition 3.1 An EGG is a 3-tuple (λ, L, P), where:

- λ is an initial graph;
- L is a label set containing terminal and non-terminal labels, i.e., $L = L_T \cup L_{NT}$;
- P is a set of productions, and each production $p \in P$ in the form of $\overline{G}_L \coloneqq \overline{G}_R$ must satisfy the following constraints:
  - (1) λ must be a left graph of a production;
  - (2) $\overline{G}_R$ must be nonempty;
  - (3) The size of left graph must be no more than that of right graph, i.e., $|\overline{G}_L.N| \leq |\overline{G}_R.N|$. If they are equal, the number of terminal nodes in left graph must be less than that of right graph, i.e., $|\overline{G}_L.N_T| < |\overline{G}_R.N_T|$.

Similar to string grammars, graph grammars with arbitrary graphs on the left and right sides of productions may face the membership problem, that is, their languages are not decidable in general. EGG introduces a size-increasing constraint for each production to solve the membership problem. The constraint ensures that any given host graphs can be parsed with EGG productions within a finite number of R-

applications. Also, the constraint is weak with little impact on the flexibility of context-sensitive grammars and easier to implement than that of LGG and RGG for grammar designers.

Theoretically, a graph grammar is a formal tool for rigorously defining a graph language, which is a set of graphs that can be derived from the initial graph. Below is the formal definition of a graph language.

Definition 3.2 Let $\text{egg} = (\lambda, \text{L}, \text{P})$ be a grammar of EGG, its language $\Gamma(\text{egg})$ can be formally defined as $\Gamma(\text{egg}) = \{\text{G} | (\lambda \rightarrow^* \text{G}) \wedge (f_{\text{NL}}(\text{G}.\text{N}) \subseteq \text{L}_\text{T})\}$.

Practically, a graph grammar is a useful tool for automatically analyzing graphs' validity. If a given graph can be reduced to the initial graph with a finite series of R-applications of a graph grammar, this graph is regarded as belonging to the grammar's language. Otherwise, the graph does not belong to the graph language or the graph grammar is not decidable.

3.2 Decidability of EGG

When an EGG is given, its language is determined. It is decidable whether an arbitrarily given graph is in the language or not because of the support of the following theorem.

Theorem 1. For EGG $\text{egg} = (\lambda, \text{L}, \text{P})$ and arbitrary nonempty graph G, it is decidable whether or not G is in $\Gamma(\text{egg})$.

Proof: For arbitrarily given graph G with a finite number of terminal nodes, a sequence of graphs can be generated in an R-application process starting from G. Because of the size-increasing constraint and the number of nodes in the graph G being finite, the R-application process cannot execute circularly and must stop in finite steps, namely, $\text{G} \mapsto^* \text{G}_\text{n}$ and G_n being unable to reduce any more by R-application. Further, the number of such sequences without a loop is also finite. Thus it is feasible to enumerate all such sequences and check whether $\text{G} \mapsto^* \text{G}_\text{n}$ and $\text{G}_\text{n} = \lambda$ are held for at least one of the sequences. If there exists one, then $\text{G} \in \Gamma(\text{egg})$, otherwise $\text{G} \notin \Gamma(\text{egg})$.

In the proof, the size-increasing constraint on the productions of EGG guarantees the decidability of EGG because the constraint requires that each R-application should at least either remove a node or change a terminal node to a non-terminal node in the reduced graph. Therefore, R-application can only be applied finite times to any host graph of a given size.

IV. PARSING ALGORITHM OF EGG

Generally, a graph grammar needs to be equipped with a parsing mechanism for automatically checking whether a given graph, called *host graph*, is structurally correct or valid with respect to the graph language defined by the grammar. Having proved that the membership problem is decidable for EGG in the previous section, this section presents a parsing algorithm, which checks if a host graph can be reduced to the initial graph by applying the EGG grammar's productions to perform a series of R-applications. A parsing algorithm usually needs to incorporate the following three interrelated actions:

- Search in the host graph for the redexes of a production's right graph;
- Perform an R-application with a found redex to generate a new host graph from the current host graph; and
- Trace all the R-application paths by applying in turn the above two actions until a path leading to the initial graph is found or all possible paths have been exhausted.

In the following, the above three actions are discussed in more detail. The first is the searching. The second is the R-application. Finally, the tracing combines the two to perform a parsing.

4.1 Search for redexes

A procedure for searching all redexes is given below, which takes host graph G and right graph $\overline{\text{G}}_\text{R}$ as input and returns a set of redexes found.

```
FindRedexForRight(Graph G, Graph G̅R)
{
  RedexSet = Φ;
  G-Nodes = OrderNodeSequence(G);
  G̅R-Nodes = OrderNodeSequence(G̅R);
  CandidateNodeSet = FindCandidateSet(G-Nodes, G̅R-Nodes);
  for each Candidate ∈ CandidateNodeSet
    RedexSet = RedexSet ∪ GenerateRedex(Candidate, G, G̅R);
  return(RedexSet);
}
```

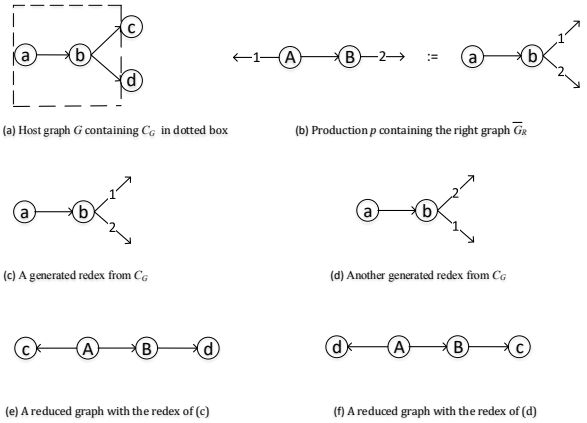
In the procedure, function OrderNodeSequence sequences the nodes in host graph G and right graph $\overline{\text{G}}_\text{R}$ separately according to their labels' alphabetical order. Function FindCandidateSet finds all possible node sequences from G-Nodes as candidates under the condition that all nodes in a candidate have the corresponding nodes in the $\overline{\text{G}}_\text{R}$-Nodes of the same degree. Function GenerateRedex generates all possible redexes derived from a candidate. Note that a candidate with all nodes plus their connected edges including dangling edges, notated as C_G, only has the same structure as $\overline{\text{G}}_\text{R}$, and may generate more than one redex. This is because a node in $\overline{\text{G}}_\text{R}$ may have more than one dangling edge in the same direction and different matches of the dangling edges between C_G and $\overline{\text{G}}_\text{R}$ may generate different redexes. Fig. 8 illustrates a case that uses the marking mechanism, where Fig. 8(a) is host graph G containing C_G in a dotted box, and Fig. 8(b) is a production containing the corresponding right graph $\overline{\text{G}}_\text{R}$. Since the node labelled 'b' in $\overline{\text{G}}_\text{R}$ is connected with two outgoing dangling edges, there are two ways of assigning the marks numbered '1' and '2' to $\text{C}_\text{G}.\dot{\text{E}}$, and thus two redexes are generated accordingly as illustrated in Fig. 8(c) and Fig. 8(d). Fig. 8(e) and Fig. 8(f) demonstrate that the two redexes are different and can reduce graph G to two different graphs.

4.2 R-application

A procedure for performing an R-application is given below, which takes host graph G, redex Q, and production p relevant to Q as inputs and generates a reduced graph.

```
RightApplication(Graph G, redex Q, Production p)
{
    AddMark(p.G̅R, G);
    InsertLeftGraph(p.G̅L, G);
```

```
    DeleteRedex(G, Q);
    G′=ClearMark(G);
    return(G′)
}
```



(a) Host graph G containing C_G in dotted box (b) Production p containing the right graph $\overline{G}_R$

(c) A generated redex from C_G (d) Another generated redex from C_G

(e) A reduced graph with the redex of (c) (f) A reduced graph with the redex of (d)

Figure 8. Reductions with two different redexes generated from a C_G

In R-application, function AddMark adds all the dangling edges' marks of $p.\overline{G}_R$ to G according to the edge mapping between them. Function InsertLeftGraph inserts $p.\overline{G}_L$ into G by connecting all dangling edges of $p.\overline{G}_L$ to the corresponding nodes in G according to the marks added previously. Function DeleteRedex deletes redex Q from G. Finally, function ClearMark clears all added marks in G to generate reduced graph G′.

4.3 Parsing

Based on the above discussions, it is now feasible to trace all possible R-application paths starting from a given host graph to check if there exists one path that leads to the initial graph. The tracing needs to maintain a mapping between a redex and its host graph for performing the corresponding R-application. As such a mapping is usually many to one, the tracing employs two stacks to separately store the redexes found and the intermediate host graph yielded, and employs a delimiter in the redex stack to delimit a group of redexes that correspond to the same host graph. The delimiter makes the correspondence manageable by synchronizing the contents in the two stacks. The function takes a graph and a set of productions as input and returns a definite answer indicating whether the graph is valid or not.

```
Parsing (Graph G, ProductionSet P)
{
loop-1:  while (G ≠ λ)
         {
           DELIMITER → RedexStack;
           // push
loop-2:    for all p ∈ P
           {
             RedexSet = FindRedexForRight(G, p.G̅R);
loop-3:          for all Redex ∈ RedexSet;
                   (Redex, p)→ RedexStack;
           // push
           }
           (Redex, p) ← RedexStack;
           // pop
loop-4:    while (Redex = DELIMITER)
           {
             If (HostStack != NULL ∧ RedexStack != NULL)
             G← HostStack;
             // pop
             (Redex, p) ← RedexStack;
             // pop
             else
             return("Invalid");
           }
           HostStack ←G ;
           // push
           G = RightApplication(G, Redex, p);
         }
         return("Valid");
}
```

4.3.1 Time complexity

This subsection examines the time and space complexities of the above parsing algorithm.

Theorem 2. The time complexity of the parsing algorithm is $O\left(\left(\frac{n}{r!}\right)^h (hh!)^r (d!\,h)^{rh})\right)$, where h is the number of nodes in the host graph to be parsed, r is the maximal number of nodes in the right graphs of all productions with d being the maximal number of dangling edges for each node, and n is the number of productions in the given EGG grammar.

Proof: According to the structure of the parsing algorithm, its maximal time complexity can be expressed as:

$$t = O(l_1(l_2(t_1 + l_3) + l_4 + t_2)),$$

where l_1 is the maximal number of iterations in the outermost loop-1, l_2 is the number of iterations in the first inner loop-2, l_3 is the number of iterations in the innermost loop-3, l_4 is the number of iterations in the second inner loop-4, and t_1 and t_2 are the worst time complexities of functions FindRedexForRight and RightApplication respectively.

In function FindRedexForRight, since the maximal possible number of selecting r nodes from h nodes is $A_h^r = h(h-1)\ldots(h-r+1)$, the worst complexity of searching for all candidates of a right graph in a given host graph is $O(h^r)$. Further, since a candidate may generate more than one redex due to different assignments of its dangling edges, the maximal possible number of actions to generate redexes from one candidate is $rA_d^d = rd!$. In fact, r and d can be considered constants when an EGG grammar is given. Thus, we have t_1 in $O(rd!\,h^r) = O(h^r)$.

In function RightApplication, since each of its sub-functions needs at most traversing the host graph once, thus it has $t_2 = O(h)$.

As to the four iterations, l_2 needs first to be considered, which is in fact the number of productions, i.e., $l_2 = n$. l_3 is the number of redexes found in the host graph with respect to the right graph of a given production. Since the maximal number of candidates with respect to the right graph of a given

production is $C_h^r = A_h^r/r!$ (by reasonably assuming $r \ll h$ for making sure that C_h^r is maximal) and each candidate can generate at most $(d!)^r$ redexes by taking dangling edges into consideration, it has $l_3 \le (d!)^r C_h^r = O(h^r)$. Since $l_2 l_3$ is the total number of actions on pushing redexes into the redex stack and l_4 is the partial number of actions on popping redexes from the redex stack, l_4 should be no more than $l_2 l_3$, and thus can be ignored. The left is iteration number l_1, the worst case is when the algorithm's result is 'invalid', and all redexes found during parsing enter the stack. Each of the redexes, when popped out of the stack, leads to an iteration of the outmost loop. Therefore, l_1 is equal to the number of all redexes found.

An iteration of the outmost loop generates no more than $n(d!)^r C_h^r$ redexes for n productions and performs one R-application. According to the size-increasing condition, each R-application would reduce the size of the derived host graph. Since there are at most *h* R-applications that may not reduce the host graph size and an R-application would reduce the host graph size by at least 1, the following holds for l_1:

$$\begin{aligned}
L_1 &\le (n(d!)^r C_h^r)^{h+1} n(d!)^r C_{h-1}^r n(d!)^r C_{h-2}^r \dots n(d!)^r \\
&\quad C_{h-(h-r-1)}^r n(d!)^r C_{h-(h-r)}^r \\
&= (n(d!)^r)^{2h-r+1} (C_h^r)^h C_{h-1}^r C_{h-2}^r \dots C_{r+1}^r C_r^r \\
&= (n(d!)^r)^{2h-r+1} \left(\frac{h!}{(h-r)!r!}\right)^h \frac{(h-1)!}{(h-1-r)!r!} \dots \frac{(r+1)!}{1!r!} \frac{r!}{0!r!} \\
&= (n(d!)^r)^{2h-r+1} \left(\frac{h!}{(h-r)!r!}\right)^h \prod_{u=1}^{h-r-1} \frac{(u+r)!}{r!u!} \\
&= \frac{(n(d!)^r)^{2h-r+1}}{(r!)^{2h-r-1}} \big(h(h-1)\dots(h-r+1)\big)^h \prod_{u=1}^{h-r-1} (u+ \\
&\quad r)(u+r-1)\dots(u+2)(u+1) \\
&= O\left(\left(n\frac{(d!)^r}{r!}\right)^{2h} h^{rh} \prod_{u=1}^{h-r-1} u^r\right) \\
&= O\left(\left(n\frac{(d!)^r}{r!}\right)^{2h} h^{rh} ((h-r-1)!)^r\right) \\
&= O\left(\left(n\frac{(d!)^r}{r!}\right)^{h} \big(h^h h!\big)^r\right) \\
&= O\left(\left(\frac{n}{r!}\right)^h (h!)^r (d!\,h)^{rh}\right) \qquad (1)
\end{aligned}$$

Combining all the above discussions, one can finally obtain:

$$t = O\left(\left(\frac{n}{r!}\right)^h (hh!)^r (d!\,h)^{rh}\right).$$

4.3.2 Space complexity

Theorem 3 The space complexity of the parsing algorithm is $O(h^{r+1})$, where *h* is the number of nodes in the host graph to be parsed, *r* is the maximal number of nodes in all the right graphs of productions.

Proof: The main space-consuming components are the redex stack and the host graph stack used in the parsing algorithm. We can, therefore, express the maximal space complexity as:

$$s = s_1 + s_2,$$

where s_1 is the space used by the redex stack and s_2 is the one by host graph stack. Without loss of generality, we can assume that the space taken by a redex is r and that by a host graph is h. Different from time complexity, the use of the stack space is not always increasing because pop operations would release space for reuse. Hence, the worst case is the maximal occupied space along with the longest R-application path, and the following holds for the redex stack and the host graph stack respectively.

$$\begin{aligned}
s_1 &\le (rhnC_h^r(d!)^r + rnC_h^r(d!)^r + rnC_{h-1}^r(d!)^r + \cdots \\
&\quad + rnC_{h-(h-r)}^r(d!)^r) \\
&= rn(d!)^r(hC_h^r + C_h^r + C_{h-1}^r + \cdots + C_r^r) \\
&= rn(d!)^r\left(\frac{(h+1)!}{(h-r)!r!} + \frac{(h-1)!}{(h-1-r)!r!} + \cdots + \frac{r!}{0!r!}\right) \\
&= \frac{n}{(r-1)!}(d!)^r\left(\frac{(h+1)!}{(h-r)!} + \sum_{u=0}^{h-r-1} \frac{(u+r)!}{u!}\right) \\
&= \frac{n}{(r-1)!}(d!)^r((h+1)h(h-1)\dots(h-r+1) + \\
&\quad \sum_{u=0}^{h-r-1}(u+r)(u+r-1)\dots(u+1)) \\
&= O(h^{r+1} + \textstyle\sum_{u=0}^{h-r-1} u^r) \\
&= O(h^{r+1}); \\
s_2 &= hh + (h-1) + \cdots + r \\
&= O(h^2). \qquad (2)
\end{aligned}$$

Since $r \ge 1$, the following can be obtained:

$$s = O(h^{r+1}).$$

From the above analysis, we notice that the time complexity is extremely high while the space complexity is bounded by a polynomial factor. We also note that the structure of productions plays an important role in determining the algorithm's complexity. For example, if a stronger constraint such as $|p.\overline{G}_L.N| < |p.\overline{G}_R.N|$ is enforced on productions, then the first *h* R-applications that do not reduce the host graph size can be removed from (1). In addition, we find that the algorithm itself may be further improved to increase its efficiency, especially its average time cost. Moreover, like RGG, if the condition of Selection-Free [28] is satisfied, the Selection-Free Parsing Algorithm with polynomial time complexity can be used for EGG.

V. Implementation of an EGG support system

A graph grammar support system is a software platform that can be helpful for end users to easily use graph grammars. This section briefly describes the architecture and functions of an EGG support system, abbreviated as EGGSS.

From a user point of view, EGGSS supplies, besides normal GUI of Windows, extra graphical and grammatical tools to assist the user to draw graphs, design graph productions, define graph languages, perform graph transformations and parse graphs. They are visualized in a friendly fashion explained below.

- Graph Editor: performs all kinds of graph related operations, such as graph drawing, saving, deleting.
- Production Designer: for designing productions based on the Graph Editor, such as production generation and modification.
- Transformer: automatically performs L-application for transforming graph from one to another based on a given production.
- Language Definer: specifies labels, marks, etc. for defining the graph language via Productions and L-application.
- Parser: automatically performs a series of R-applications for checking the validity of a given graph.

Fig. 9 illustrates the end user view of EGGSS. Fig. 10 is an example window of EGGSS's user interface, where the upper row is the main menu with all operational items including not only graphical and grammatical operations but also other

Window GUI operations. On the left, a tree view allows users to manage XML files with saving, accessing and deleting operations. They can read graphs in XML format from the memory and save graph data to an XML file. On the right, the upper part shows an edited host graph and the lower part shows a designed production.

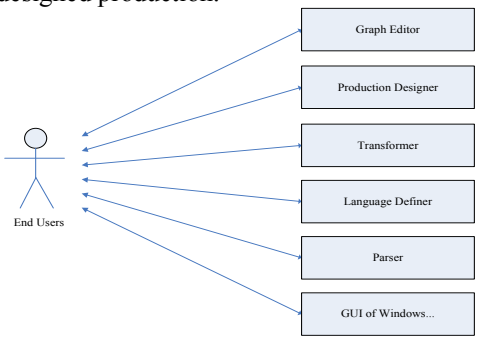


Figure 9. End user view of EGGSS

From a system point of view, EGGSS consists of basic modules, organized logically in layers to realize the system's grammatical functions explained below.

- Graph Transformation: automatically completes the transformation from one graph to another using productions. This module is essentially an L-application.
- Graph Parsing: performs grammatical analysis for a given host graph by automatically searching for all possible graph reduction operations to finally reduce to the initial symbol, namely the host graph is grammatically valid if and only if it could be reduced to the initial symbol. This module is essentially a series of R-applications.
- Graph Matching: finds redexes in a graph according to a given production.
- Graph Substitution: replaces a sub-graph in a given graph using the left or right graph of a production.
- XML Description: transfers graph expressions to XML descriptions and vice versa.

Fig. 11 shows the system architecture with relevant modules. In the architecture, three upper layers are implemented using C++ in the environment of Visual Studio 2005, while two lower layers are implemented using the existing XML open sources and software tools.

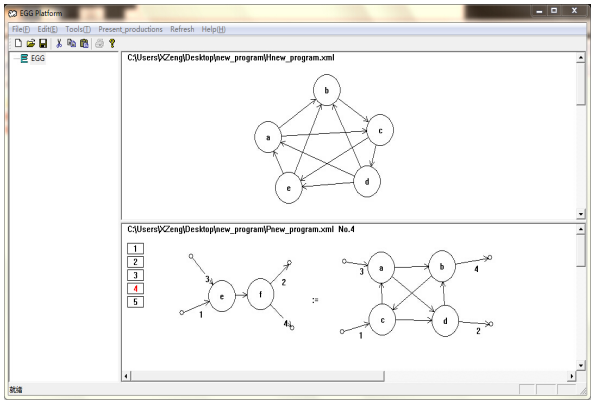


Figure 10. A window of EGGSS's user interface

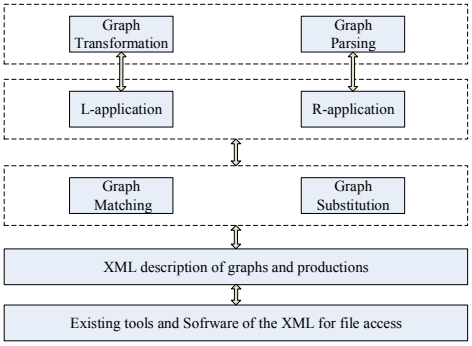


Figure 11. The architecture of EGGSS

I. CONCLUSIONS

This paper has proposed a new graph grammar formalism, namely EGG, which aims at making the design and implementation of a graph grammar simple without weakening the expressive power of the grammar. The proposed EGG lays a solid foundation for a wide range of applications using graph grammars. Specifically, EGG focuses on tackling general graph languages and graph transformations with productions as simple as possible. First, EGG simplifies the expression of productions, in which the context nodes are eliminated and only edges linked to context nodes are kept. In this way, the structural information of graphs is still kept. Second, using dangling edges and their corresponding marks, the replacement of a redex by either a left or right graph in a production can be easily done without ambiguity. Third, the introduction of size-increase constraint to productions solves the membership problem, making EGG parsing algorithm terminable.

As a future research, we will attempt to find the way to reduce the parsing complexity, to further improve EGGSS to be friendlier for end users.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China under grant 61170089.

REFERENCES

[1] P. Bottoni, S. K. Chang, M. F. Costabile, S. Levialdi, P. Mussio. On the specification of dynamic visual languages, Proc. IEEE Symposium on Visual Languages, 14-21, 1998.

[2] P. Bottoni, S. K. Chang, M. F. Costabile, S. Levialdi, P. Mussio. Modeling visual interactive systems through dynamic visual languages, IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, 32(6): 654-669, 2002.

[3] S. K. Chang. Extending visual languages for multimedia, IEEE Multimedia, 3(3): 18-26, 1996.

[4] S. K. Chang. A visual language compiler for information retrieval by visual reasoning, IEEE Transactions on Software Engineering, 16(10): 1136-1149, 1990.

[5] G. Rozenberg, H. Ehrig, Handbook of graph grammars and computing by graph transformation, Handb. Graph Grammars. 1 (1997) 1–8.

[6] H. Fahmy, D. Blostein, A Survey of Graph Grammars: Theory and Applications, in: IAPR Int. Conf. Pattern Recognit., 1992: pp. 294–298.

[7] C. Ermel, M. Rudolf, G. Taentzer, The AGG Approach: Language and Environment, in: Handb. Graph Grammars 2, 1999: pp. 551-603.

[8] D.-Q. Zhang, K. Zhang, J. Cao, A context-sensitive graph grammar formalism for the specification of visual languages, Comput. J. 44 (2001)

[9] X.-Q Zeng, K. Zhang, J. Kong, G.-L. Song, RGG+: An enhancement to the reserved graph grammar formalism, in: Proc. 2005 IEEE Symp. Vis. Lang. Human-Centric Comput., 2005: pp.272–274.

[10] D. Goik, K. Jopek, M. Paszyński, A. Lenharth, D. Nguyen, K. Pingali, Graph grammar based multi-thread multi-frontal direct solver with Galois scheduler, in: Procedia Comput. Sci., 2014: pp.960–969.

[11] L. Fürst, M. Mernik, V. Mahnič, Converting metamodels to graph grammars: doing without advanced graph grammar features, Softw. Syst. Model 14 (2013) 1297–1317.

[12] J. Heinen, C. Jansen, J.-P. Katoen, T. Noll, Verifying pointer programs using graph grammars, Sci. Comput. Program. 1 (2013) 7–12.

[13] Z. Shi, X.-Q. Zeng, S. Huang, H. Li, Transformation between BPMN and BPEL based on graph grammar, in: Proc. 5th Int. Conf. Comput. Commun. Netw. Technol., 2014: pp.1-6.

[14] F. Hermann, S. Gottmann, N. Nachtigall, H. Ehrig, B. Braatz, G. Morelli, A. Pierre, T. Engel, C. Ermel, Triple Graph Grammars in the Large for Translating Satellite Procedures, Theor. Prac. Model Transforms. 8568 (2014) 122-307.

[15] Y. Ong, K. Streit, M. Henke, W. Kurth, An approach to multiscale modelling with graph grammars, Ann. Bot. 114 (2014) 813–827.

[16] K. Wittenburg, L. Weitzman, Relational grammars: theory and practice in a visual language interface for process modeling. Vis. Lang. Theor. (1998) 193-217.

[17] G. Rozenberg, E. Welzl, Boundary NLC graph grammars-Basic definitions, normal forms, and complexity, Inf. Control. 69 (1986) 136–167.

[18] D. Janssens, G. Rozenberg, Graph grammars with neighbourhood-controlled embedding, Theor. Comput. Sci. 21 (1982) 55–74.

[19] F. Drewes, H.-J. Kreowski, A. Habel, Hyperedge Replacement Graph Grammars, in: Handb. Graph Grammars 1, 1997: pp.95–162.

[20] K. Wittenburg, Earley-style parsing for relational grammars, Proc. 8th IEEE Workshop. Vis. Lang., 1992: pp. 192-199.

[21] E. Golin, A Method for the specification and parsing of visual languages, PhD Thesis, 1991, Department of Computer Science, Brown University.

[22] K. Marriott, Constraint Multiset Grammars, Proc. IEEE Symp. Vis. Lang., 1994: pp. 118–125.

[23] J. Rekers, a Schürr, Defining and parsing visual languages with layered graph grammars, J. Vis. Lang. Comput. 8 (1997) 27–55.

[24] J. Kong, K. Zhang, X.-Q. Zeng, Spatial graph grammars for graphical user interfaces, ACM Trans. Comput. Interact. 13 (2006) 268–307.

[25] J. Kong, K. Zhang, Parsing Spatial Graph Grammars, Proc. 2004 IEEE Symp. Vis. Lang. Hum. Centric Comput. (2004) 99–101.

[26] M. Decker, H. Che, A. Oberweis, P. Stürzel, M. Vogel, Modeling mobile workflows with BPMN, in: ICMB GMR 2010 - 2010 9th Int. Conf. Mob. Business/2010 9th Glob. Mobil. Roundtable, 2010: pp.272–279.

[27] C. Kim, M. Ando, Node replacement graph grammars with dynamic node relabeling, Theor. Comput. Sci. 583 (2015) 40–50.

[28] K. Zhang, D.-Q. Zhang, J. Cao, Design, construction, and application of a generic visual language generation environment, IEEE Trans. Softw. Eng. 27 (2001) 289–307.

[29] G. Song, K. Zhang, Visual XML schemas based on reserved graph grammars, in: Proc. Int. Conf. Inf. Technol. Coding and Computing, 2004: pp. 687-691.

[30] K. Zhang, D.-Q. Zhang, Y. Deng, A Visual Approach to XML Document Design and Transformation, Proc. IEEE Symp. Human-Centric Comput. Lang. Environ., 2001: pp.312–319.

[31] K.-B. Zhang, M.A. Orgun, K. Zhang, A prediction-based visual approach for cluster exploration and cluster validation by HOV3. Lec. Notes Comput. Sci. 4702 (2007) 336-349.

[32] C. Zhao, J. Kong, K. Zhang, Design pattern evolution and verification using graph transformation, Proc. 40th Annual Hawaii International Conference on System Sciences (HICSS'2007), 2007: pp.290a-290a.

[33] C. Zhao, J. Kong, J. Dong, K. Zhang, Pattern-based Design Evolution Using Graph Transformation, J. Vis. Lang. Comput. 18 (2007) 378–398.

[34] C. Zhao, J. Kong, K. Zhang, Program behavior discovery and verification: A graph grammar approach, IEEE Trans. Softw. Eng. 36 (2010) 431–448.

[35] K. Zhang, J. Kong, Exploring semantic roles of Web interface components, Proc. Int. Conf. Mach. Web Intell., 2010: pp.8-14.

[36] K. Zhang, J. Kong, M. Qiu, G. Song, Multimedia layout adaptation through grammatical specifications, Multimedia Syst. 10 (2005) 245-260.

[37] J. Kong, K.-L. Ates, K. Zhang, Y. Gu, Adaptive mobile interfaces through grammar induction, Proc. Int. Conf. Tools with Artif. Intell. ICTAI, 2008: pp.133–140.