An Edge-based Graph Grammar Formalism and Its Support System

Xiaqin Zeng1, Yufeng Liu1, Zhan Shi1, Yingfeng Wang2, Yang Zou1, Jun Kong1, Kang Zhang4

1Institute of Intelligence Science and Technology, Hohai University, Nanjing, Jiangsu, China
2School of Information Technology, Middle Georgia State University, Macon, GA 31206, USA
3Department of Computer Science, North Dakota State University, Fargo, ND 58102, USA
4Department 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. Then, a new parsing algorithm 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. In view of the theoretical role played to string languages by string grammars, 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. 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.

Based and improved on LGG, Zhang et al. [8] proposed another context-sensitive grammar called Reserved Graph 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. With the introduction of selection-free productions to graph grammars, a
Selection-Free Parsing Algorithm (SFPAL) 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 and parsing algorithm complicated to implement with high time complexity. 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 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. Section 4 presents a parsing algorithm. 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 \( L = L_T \cup L_NT \cup L_M \). \( L_T \cap L_NT = \emptyset \), and \( L_M \cap (L_T \cup L_NT) = \emptyset \).

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

**Definition 2.1** \( n \) is a node with label \( l \) in a given finite label set \( L \).

**Definition 2.2** \( e = (n_s, n_e) \) is a directed edge, where

- \( n_s \) is the start node of the edge;
- \( n_e \) is the end node of the edge.

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

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

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** \( G = (N,E) \) is a graph on given label set \( L \), where

- \( N \) is a node set containing terminal and nonterminal nodes, i.e., \( N = N_T \cup N_NT \);
- \( E \) is an edge set with \( E \subseteq N \times N \).

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

- \( f_{NL} : N \rightarrow L \), a mapping from node \( n \) to label \( l \in L \), i.e., \( f_{NL}(n) = n.l \);
- \( f_{EN} : l \rightarrow N \), a mapping from edge \( e \) to its start node, i.e., \( f_{EN}(e) = e.n_s \);
- \( f_{EM} : e \rightarrow N \), a mapping from edge \( e \) to its end node, i.e., \( f_{EM}(e) = e.n_e \).

In EGG, dangling edge set \( \hat{E} \) is introduced to represent contexts, in which each edge is connected with only one node being either a start or end node, namely \( \hat{E} = \hat{E}_s \cup \hat{E}_e \) with \( \hat{E}_s = \{ e_s | e_s = (n_s, \emptyset) \} \) and \( \hat{E}_e = \{ e_e | e_e = (\emptyset, n_e) \} \) and \( \hat{E}_s \cap \hat{E}_e = \emptyset \). 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 \( \hat{E} = \{1,2,3\} \). The graph is called a dangling edge graph and can be defined as follows.

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

- \( N \) is a node set;
- \( E \) is an edge set including dangling edges, i.e., \( E = E \cup \hat{E} \);
- \( M \subseteq L_M \) is a mark set for marking dangling edges to distinguish different contexts.
Essentially, $G$ is an extension of $G$ by introducing dangling edges, and $G$ can be regarded as a special case of $G$. Similarly, there is an extra mapping as follows.

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

Note that dangling edge set $E$ 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 $G_L := G_R$, which consists of a left dangling edge graph $G_L$ and a right dangling edge graph $G_R$ satisfying $G_L,M = G_R,M$.

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. Fig. 2 is an example of a set of EGG productions specifying a process flow diagram with (begin, assign, fork, join, send, receive, if, endif) $\subseteq L_T$ and (stat) $\subseteq L_NT$.

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 $G \cong Q$, $f_{NL}$ and $f_{NL}'$ are two mappings for $G$ and $Q$ respectively, if and only if there exist two bijective mappings $f_{NN}:G,N \leftrightarrow Q,N$ and $f_{EE}:G,E \leftrightarrow Q,E$, and the following are satisfied:

- $\forall n((n \in G,N) \lor (n \in Q,N)) \rightarrow (f_{NL}(n) = f_{NL}'(f_{NN}(n)))$;
- $\forall e((e \in G,E) \lor (e \in Q,E)) \rightarrow (f_{EN}(f_{NL}(e)) = f_{EN}'(f_{NL}'(e)))$.

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 $Q \subseteq G$, if and only if the following are satisfied:

- $\forall (Q,N \subseteq G.N) \lor (Q,E \subseteq G.E)$.

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

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

- $\forall (Q,N \subseteq G.N) \lor (Q,E \subseteq G.E)$.

Core graph $Q$ is the sub-graph of graph $G$ obtained by removing all dangling edges from graph $G$ and keeping all the nodes and non-dangling edges of graph $G$. 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 $G_{LIR}$ is a graph being left or right side of a production, $Q$ is a redex of $G$ with respect to $G_{LIR}$, denoted as $Q \in \text{Redex}(G_{LIR})$, if and only if there exits bijective mappings $f_{NN}:G,N \leftrightarrow G_{LIR},N$ and $f_{EE}:G,E \leftrightarrow G_{LIR},E$, and the following are satisfied:

- $\forall (Q,N \subseteq G.N) \lor (Q,E \subseteq G.E)$.

To explain the above definition, we provide an example in the following three figures. Fig. 4 is graph $G_{LIR}$, 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 $G_{LIR}$.

In host graph $G$, if there is sub-graph $Q$ being the redex of $G$ with respect to $G_{LIR}$ 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.
Definition 2.10 An \( L \)-application to graph \( G \) is a transformation that generates graph \( G' \) using production \( p \): \( G \rightarrow G' \), denoted as \( G' = \text{Tr}(G, Q, \overline{G}_L, \overline{G}_R) \), where \( Q \in \text{Redex}(G, \overline{G}_L) \), and \( \text{Cor}(\overline{G}_R) \) is used to replace \( Q \) in \( G \). The \( L \)-application is also called derivation operation and denoted as \( G \rightarrow^* G' \).

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 \( p \): \( G \rightarrow G'' \), denoted as \( G'' = \text{Tr}(G, Q, \overline{G}_R, \overline{G}_L) \), where \( Q \in \text{Redex}(G, \overline{G}_R) \), and \( \text{Cor}(\overline{G}_L) \) is used to replace \( Q \) in \( G \). The \( R \)-application is also called reduction operation and denoted as \( G \rightarrow^0 G'' \).

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

Fig. 7 shows a derivation process from an initial graph using the productions in Fig. 2.
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 \((\lambda, L, P)\), where:

- \(\lambda\) 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 \(G_L \Rightarrow G_R\) must satisfy the following constraints:
  1. \(\lambda\) must be a left graph of a production;
  2. \(G_R\) must be nonempty;
  3. The size of left graph must be no more than that of right graph, i.e., \(|G_L. N| \leq |G_R. N|\). If they are equal, the number of terminal nodes in left graph must be less than that of right graph, i.e., \(|G_L. N_T| < |G_R. N_T|\).

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 \(\text{egg} = (\lambda, L, P)\) be a grammar of EGG, its language \(\Gamma(\text{egg})\) can be formally defined as \(\Gamma(\text{egg}) = \{G | (\lambda \rightarrow^* G) \land (f_M(G, N) \subseteq L_T)\}\).

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.

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. 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.

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

To trace all possible R-application paths starting from a given host graph, a mapping between a redex and its host graph is needed. 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.

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.

Fig. 9 illustrates the end user view of 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. 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. 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 9. End user view of EGGSS

Figure 10. A window of EGGSS’s user interface

Figure 11. The architecture of EGGSS
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 languages and graph transformations with productions as a solid foundation for namely EGG, which aims at making the design and human-computer interaction.

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


