A Context-sensitive Graph Grammar Formalism for the Specification of Visual Languages

DA-QIAN ZHANG[1], KANG ZHANG[2,4] AND JIANNONG CAO[3]

[1]*Coral Corporation, 1600 Carling Avenue, Ottawa, Canada*
[2]*Department of Computer Science, Box 830688, MS EC31, University of Texas at Dallas, Richardson, Texas 75083-0688, USA*
[3]*Department of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong*
Email: kzhang@utdallas.edu

Graph grammars may be used as natural and powerful syntax-definition formalisms for visual programming languages. Yet most graph-grammar parsing algorithms presented so far are either unable to recognize interesting visual languages or tend to be inefficient (with exponential time complexity) when applied to graphs with a large number of nodes and edges. This paper presents a context-sensitive graph grammar called reserved graph grammar, which can explicitly and completely describe the syntax of a wide range of diagrams using labeled graphs. The parsing algorithm of a reserved graph grammar uses a marking mechanism to avoid ambiguity in parsing and has polynomial time complexity in most cases. The paper defines a constraint condition under which a graph defined in a reserved graph grammar can be parsed in polynomial time. An algorithm for checking the condition is also provided.

Received 25 May 2000; revised 19 March 2001

1. INTRODUCTION

An important class of visual programming languages is the diagrammatic one, which is based on object-relationship abstractions (e.g. using nodes and edges). Frequently used diagrammatic visual languages include entity-relationship database design languages, data-flow programming languages (e.g. Petri nets), control flow programming languages, state transition specifications, and so on.

In the implementation of textual languages, formal grammars are commonly used to facilitate the language understanding and the parser creation. When implementing a diagrammatic visual programming language (in the rest of the paper, diagrammatic visual programming languages will simply be referred to as visual languages), this is not usually the case. Graph grammars with their well-established theoretical background may be used as natural and powerful syntax-definition formalisms [1] and the parsing algorithm based on a graph grammar may be used to check the syntactical correctness and to interpret the language semantics.

One obstacle for the application of graph grammars is that even for the most restricted classes of graph grammars the membership problem is NP-hard [2]. As a consequence, all the graph-grammar parsing algorithms proposed so far are either unable to recognize interesting languages of graphs, or tend to be inefficient when applied to graphs with a large number of nodes and edges.

Another problem is that nearly all known graph-grammar parsing algorithms [2, 3, 4, 5, 6, 7] deal only with context-free productions. A context-free grammar requires that only a single non-terminal is allowed on the left-hand side of a production [8]. A context-sensitive graph grammar, on the other hand, allows left-hand and right-hand graphs of a production to have an arbitrary number of nodes and edges. Most existing graph-grammar formalisms for visual languages are context-free. Yet not many visual languages can be specified by purely context-free productions. Additional features are required for context-free graph grammars to handle context-sensitivity; it is therefore difficult for context-free grammars to specify a large proportion of visual languages.

Rekers and Schürr [9] proposed *layered graph grammars* (LGGs) to specify visual languages. LGGs differ from most other grammars in two aspects: context sensitivity and graph formalism. Being context sensitive makes the graph grammars expressive. The graph formalism in LGGs is intuitive and thus easier to understand and to use than textual formalisms for specifying visual languages. However, although being expressive, the layered graph grammar is inefficient in its implementation. Its parsing algorithm is complicated and the parsing complexity generally reaches exponential time. It is reported that parsing grammars using the Rekers–Schürr algorithm is extremely hard [10].

[4]Author for correspondence.

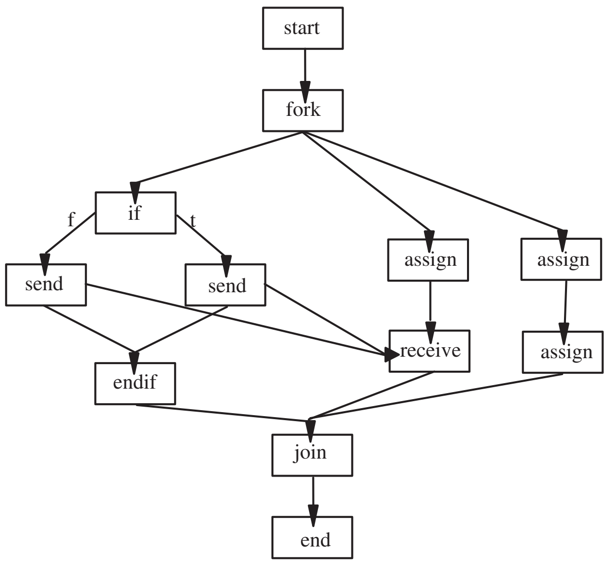


FIGURE 1. A process flow diagram.

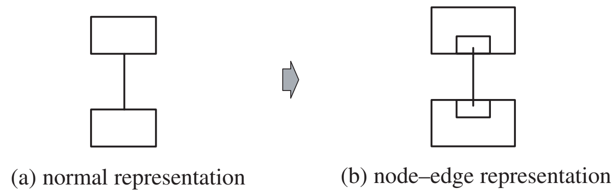


FIGURE 2. From a diagram to a node–edge representation.

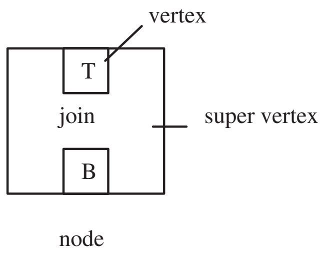


FIGURE 3. Node structure.

The LGG is therefore not suitable for a program with a large number of nodes and edges.

This paper presents a context-sensitive graph grammar called *reserved graph grammar* (RGG), which was motivated by the development of a general-purpose visual language generator. Because the targets of the generator are visual languages, their grammars are better specified using a graph formalism. As a part of the generator, a visual editor should be used to create visual programs based on the grammar specifications and parsing algorithms should be automatically created according to the grammar.

The RGG is developed based on the layered graph grammar, by using the layered formalism to allow the parsing algorithm to determine in finite steps whether a graph is valid. It uses labeled graphs to support the linking of newly created graphs into a parsed graph (traditionally called embedding). The node structure enhanced with additional visual notations in the RGG simplifies the transformation specification and also increases the expressiveness.

An RGG is complete and explicit in describing the syntax of a wide range of diagrams. Compared to the LGG where the context graph [9] must explicitly appear in the production, the embedding mechanism in the RGG allows the grammar representation to avoid most of the context specifications while being more expressive. This greatly reduces the expression complexity, and in turn increases the efficiency of the parsing algorithm.

A general RGG parsing algorithm, however, has exponential time complexity. This is solved by introducing a constraint into the RGG. It is not yet clear how this constraint limits the application scope, but we find that even the grammar of a complicated control flow diagram satisfies the constraint. With this constraint, a parsing algorithm of polynomial time complexity can be developed. An algorithm for checking whether an RGG satisfies the constraint is also developed.

The RGG formalism has been used in the implementation of a toolset called VisPro, which facilitates the generation of visual languages using the Lex/Yacc approach [11, 12].

The rest of the paper is organized as follows: Section 2 describes a case study that demonstrates the basic idea of the RGG. Section 3 provides a formal definition of the RGG formalism. Section 4 defines a selection-free condition which allows an RGG to be parsed in polynomial time. Section 5 describes the application of the RGG formalism in a visual-language generation system, followed by the comparison of related work in Section 6. Section 7 concludes the paper.

2. A CASE STUDY

2.1. Process flow diagrams

We use a process flow diagram (PFD) as an example to illustrate how an RGG works. A PFD has two types of constructs: structured and non-structured. For example, a fork–join construct provides a structure in a diagram, while the send–receive construct does not affect the structure of a diagram. Many diagrams used in computer science have such a mixture of constructs, which are difficult to specify using existing graph grammars except the layered graph grammar.

In the PFD shown in Figure 1, the *fork* statement splits one thread into multiple threads (three in the example). There are two *send* statements that send different messages to the same *receive* statement. Syntactically, a *receive* statement can receive information from any number of send statements, while a *send* statement can send to only one *receive*. A *fork* statement can split one thread into any number of threads.

We first translate the diagram in Figure 1 into a graphical form whose syntax is suitable for the RGG interpretation. We will call such a graphical form a *node–edge diagram*. The translation as shown in Figure 2 is very straightforward; the arrows may be ignored since the direction is unimportant

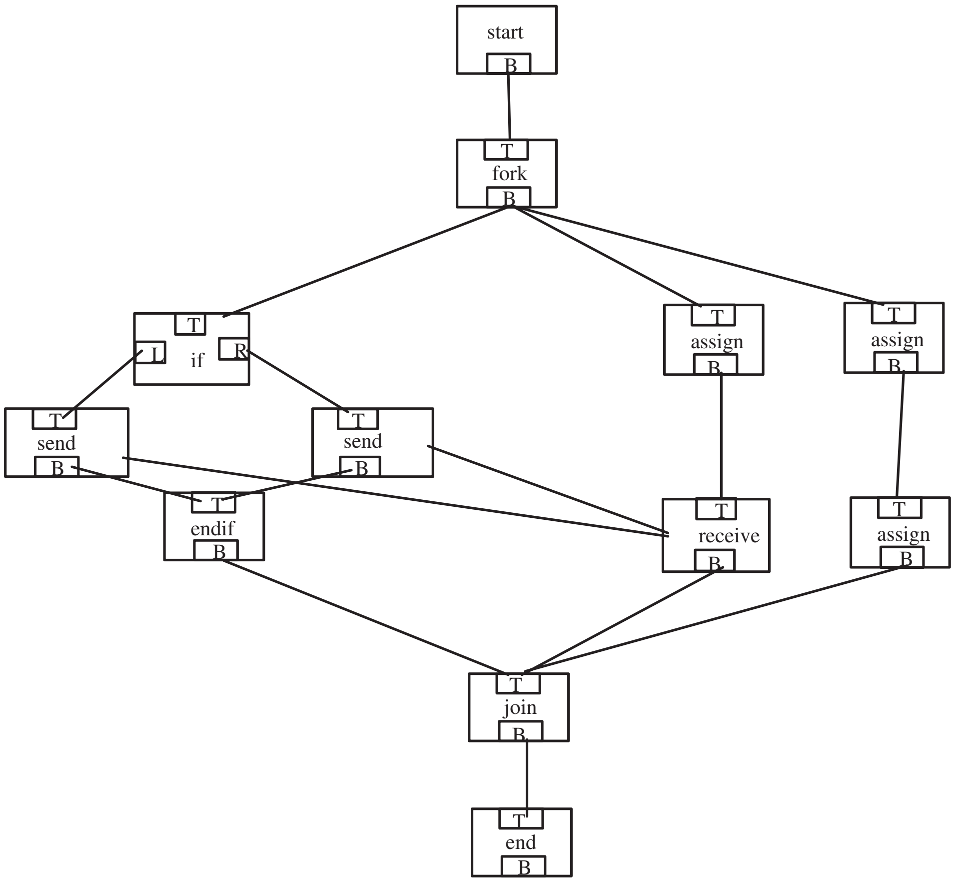


FIGURE 4. The node–edge form of the process flow diagram.

in our graph-grammar representation. A node in the node–edge representation is a two-level structure. Figure 3 depicts an example node called *join*. The first level is the large surrounding rectangle, which is called a *super vertex*. The small rectangles embedded in a super vertex are the second level, called *vertices*. A vertex can be connected to one or more edges; an edge is uniquely determined by two vertices in the involved nodes. A super vertex is also a vertex that can be connected to other vertices; RGG does not impose semantic difference between connecting to a vertex and connecting to a super vertex. The translated node–edge representation of the process flow diagram is shown in Figure 4.

In a node–edge diagram, all vertices should be labeled. For simplicity, we use T (top), B (bottom), L (left), R (right) to label the vertices according to their positions in a node.

2.2. Graph rewriting rules

A graph rewriting rule, also called a production, has two graphs which are called *left graph* and *right graph*. It can be applied to another graph (called *host graph*) in the form of an *L-application* or *R-application*. A production's L-application to a host graph is to find in the host graph a *redex* of the left graph of the production and replace the redex with the right graph of the production. An R-application is a reverse replacement (i.e. from the right graph to the left graph). A *redex* is a sub-graph in the host graph which is isomorphic to the right graph in an R-application or to the left graph in an L-application.

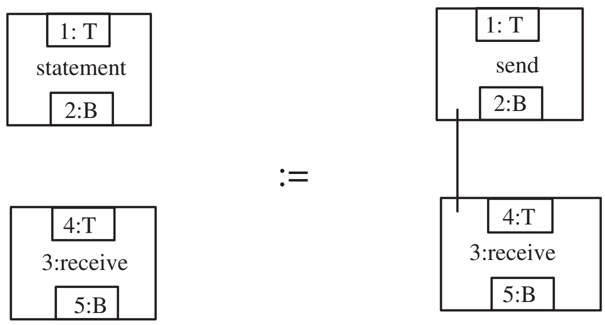


FIGURE 5. A graph rewriting rule.

In the case of linear textual languages, it is clear how to replace a non-terminal in a sentence by a corresponding sequence of (non-)terminals. However, with a visual language that has two-dimensional relationships among the language elements, a far more complicated mechanism is needed to establish relationships between the substitute of a redex and its adjacent elements.

There are three approaches to embedding a graph into a host graph [9].

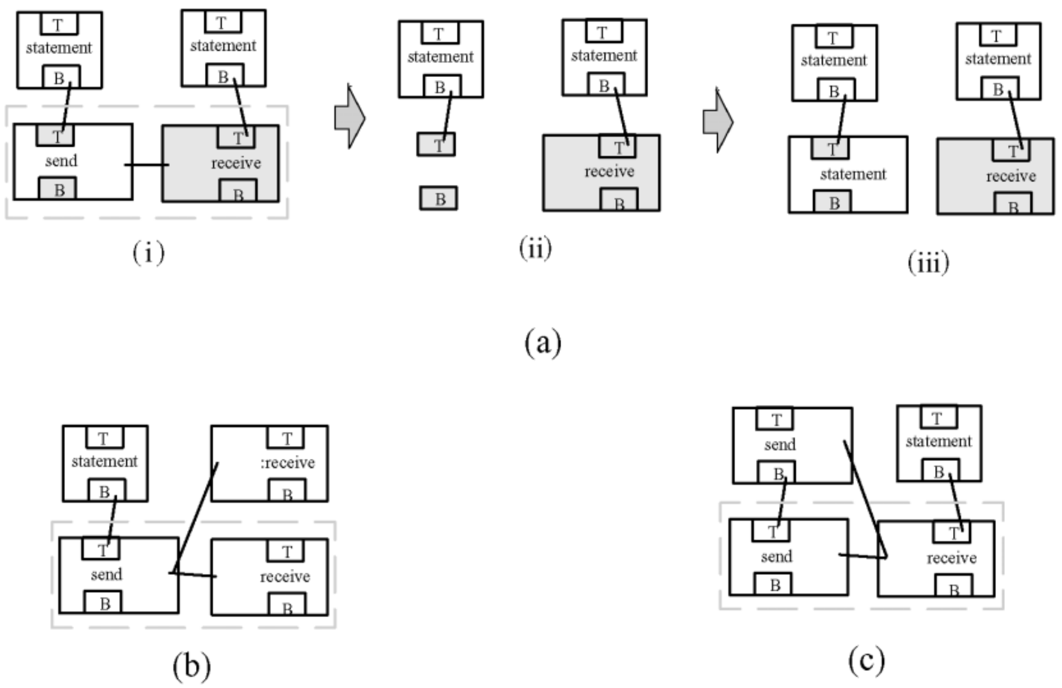


FIGURE 6. Examples of the R-application.

Implicit embedding. Formalisms such as picture layout grammars [4] and constraint multiset grammars [13] do not distinguish between vertices and edges. Relationships are implicitly defined as constraints over their attribute values. Attribute assignments within productions have the implicit side effect that creates new relationships to unknown context elements. Users are, therefore, not always aware of the consequences of attribute assignments, and parsers require considerable time to extract, from attributes and constraints, implicitly defined knowledge about the relationships.

Embedding rules. Some graph grammars such as the NLC graph grammar [2] and the DNECL graph grammar [14] have separate embedding rules which allow the redirection of arbitrary sets of relationships from a redex to its substitute. This approach is easy to implement. However, the embedding rules are often difficult to understand, and all known parsing algorithms for productions with embedding rules are either inefficient or impose very strict restrictions on the left- and right-hand sides of the productions. Furthermore, embedding rules are only able to redirect or re-label existing relationships. They cannot be used to define such productions as the one in Figure 5, which establishes new relations between previously unconnected vertices.

Context elements. Context elements can be used to establish the relationships between a newly created graph and the host graph. This approach is the easiest to understand, but an unrestricted use of context elements may complicate the graph rewriting rules. Furthermore, it is difficult to rewrite elements which may participate in a statically unknown number of relationships.

The reserved graph grammar combines the approaches of the embedding rule and the context elements to solve the embedding problem. By introducing context information, simple embedding rules can be sufficiently expressive to handle complicated programs. Moreover, the wildcards formalism used in the LGG is not needed in the RGG. The following paragraphs explain our new embedding approach by showing its application in the graph transformation process. In order to identify any graph elements which should be reserved during the transformation process, we mark each isomorphic vertex in a production graph by prefixing its label with a unique integer. The purpose of marking a vertex is to preserve the context.

We impose an embedding rule which states that if a vertex in the right graph of the production is unmarked and has an isomorphic vertex v in the redex of the host graph, then all edges connected to v should be completely inside the redex. With the above embedding rule which is usually called the *dangling condition* [1], each application of a production can ensure that a graph can be embedded in a host graph without creating dangling edges. The examples in Figure 6 illustrate the R-application process, where some host graphs have isomorphic graphs (enclosed in dashed boxes) of the right graph of the production in Figure 5. In Figure 6a(i), the isomorphic graph is a redex. The vertices corresponding to the isomorphic vertices marked in the right graph of the production are painted gray. The transformation deletes the redex while keeping the gray vertices, as shown in Figure 6a(ii). Then the left graph of the production is embedded into the host graph, as shown in Figure 6a(iii), while treating a marked vertex in the left graph the same as a gray vertex that has the same mark. We can see that the marking mechanism allows some edges of a vertex to be reserved after transformation. For example, in Figure 6a, two edges from B to T are reserved after transformation. Note that Figure 6a(ii) serves only as an illustration of 'reserving', and is not the result of a transformation.

In our notion of process flow diagrams, a send node is allowed to connect to only one receive node. We show how such a restriction can be expressed and maintained in the RGG. The solution is simple: we leave the send node unmarked in the production. According to the embedding rule, the isomorphic graph in Figure 6b is not a redex because the super vertex in the send node has an edge that is not inside the isomorphic graph while its isomorphic super vertex in the right graph is unmarked. Therefore, the graph in Figure 6b is invalid. On the other hand, we allow a receive node to receive data from one or more send nodes. To support this, we mark the super vertex of the receive node in the production in Figure 5. The graph in Figure 6c is valid according to the embedding rule. There is a redex (in the dotted box) in the graph, because the super vertex of receive has its isomorphic vertex marked in the right graph of the production, even though it has an edge connected outside the isomorphic graph. Therefore, the marking mechanism helps not only in embedding a graph correctly, but also in simplifying the grammar definition.

2.3. A graph grammar for process flow diagrams

The graph grammar shown in Figure 7 explicitly and precisely depicts the syntax of the PFD language. It consists of a set of productions, where a box labeled $\langle i \rangle$ is Production i.

The L-application defines the language of a grammar. The language is defined by all possible graphs which have only terminal labels and can be derived using L-applications from an initial graph (i.e. λ). The R-application is used to parse a graph. If the graph is eventually transformed into an initial graph after a series of R-applications, the graph is proved to belong to the language. In the sequel, we prove that the R-application can precisely determine the language defined by the L-application for an RGG.

By applying the R-application of the RGG in Figure 7 repeatedly to a specific diagram, we can determine whether the diagram is a process flow diagram. The process of parsing the PFD drawn in Figure 1 is illustrated in Figure 8, where a label in an oval describes a possible R-application order (represented by a letter, e.g. c is after a) and the corresponding production (by a numeral). The notation d:2 means that the redex of Production 2 is applied after the R-applications a, b and c have been applied. The R-applications may be applied in different orders but will produce the same result.

In Figure 8a, the five sub-graphs in the dotted boxes are possible redexes, which can be applied with productions $\langle 6 \rangle$, $\langle 6 \rangle$, $\langle 2 \rangle$, $\langle 2 \rangle$ and $\langle 2 \rangle$ to produce the graph in Figure 8b. Similarly, the graph in Figure 8b can be transformed into the graph in Figure 8c, and so on. Finally, the graph is transformed into an initial graph. The original diagram is, therefore, a valid process flow diagram.

The following section presents a formal definition of the reserved graph grammar.

3. FORMAL DEFINITION

3.1. Preliminaries

In order to define the reserved graph grammar and its properties, we will first introduce some basic concepts, such as graph element, graph, and isomorphism. We then define the marking mechanism, which allows us to further define a redex and graph transformations including L- and R-applications.

DEFINITION 3.1. *$n := (s, V, l)$ is a node on a label set L, where*

- *V is a set of vertices,*
- *$s \in V$ is a super vertex, and*
- *$l : V \to L$ is an injective mapping from V to L.*

A super vertex contains a set of vertices, and is itself a vertex. A label serves as a type in an RGG. For simplicity, we will use the notations $n.V$ and $n.s$ to represent the corresponding parts of a node n; and this convention is applicable to other definitions.

DEFINITION 3.2. *Two nodes n_1 and n_2 are isomorphic, denoted as $n_1 \approx n_2$,* iff

- *they are defined over the same label set, and*
- *$\exists f ((f : n_1.V \to n_2.V$ is a bijective mapping$) \wedge \forall v \in n_1.V\ (n_1.l(v) = n_2.l(f(v))) \wedge n_2.s = f(n_1.s))$.*

The definition specifies that two nodes are isomorphic if they have the same types of vertices (including super vertices).

DEFINITION 3.3. *$G := (N, E)$ is a graph over a label set L, where*

- *N is a finite set of nodes over L,*
- *$E \subseteq N.V \times N.V$, where $N.V = \bigcup_{n \in N} n.V$ is a finite set of edges.*

Each edge connects from a vertex of a node to a vertex of another node and is defined by that pair of vertices.

Not all graphs are meaningful. Only certain types of graphs represent meaningful visual sentences. A graph grammar can be used to define those graphs that are valid visual sentences. To specify the graph grammar we need to define the following concepts.

DEFINITION 3.4. *A vertex v is said to be* marked*, denoted as* mark$(v) = m$*, if it is assigned an integer m called* mark.

DEFINITION 3.5. *$G := (N, E, M)$ is a marked graph over a label set L, where*

- *(N, E) is a graph over L, and*
- *$M : V \to I$ is a bijective mapping, where $V \subseteq N.V$, and I is a set of integers.*

A marked graph has unique integers in some of its vertices. Different vertices in a marked graph should have

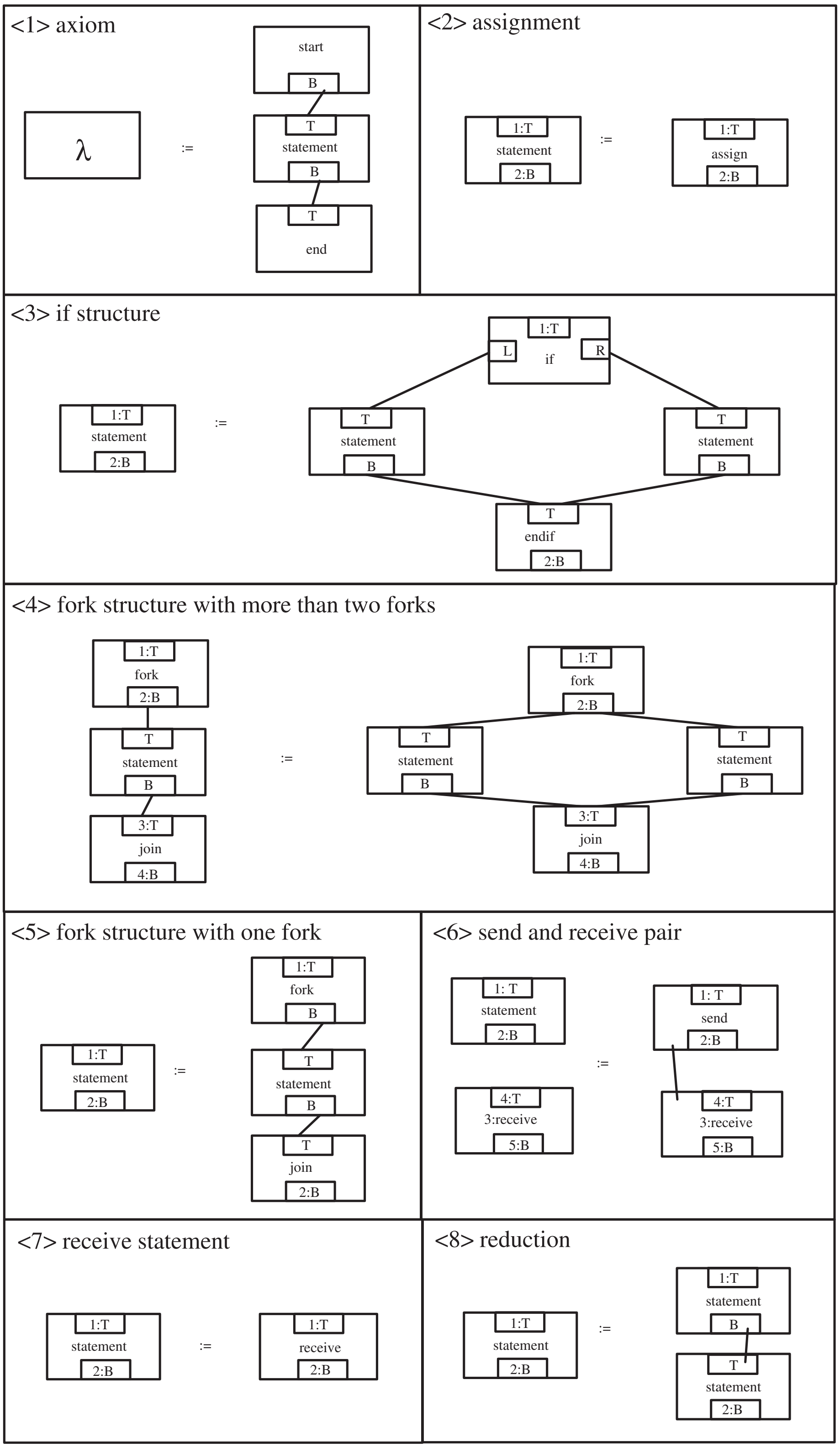


FIGURE 7. A reserved graph grammar specifying process flow diagrams.

different marks. We use mark(v) $=$ m to indicate that v is assigned an integer m, and mark(v) $=$ null to indicate that v is assigned nothing and is said to be unmarked.

DEFINITION 3.6. *Two vertices a and b in two different graphs are equivalent, denoted as $a \doteq b$,* iff mark(a) $=$ mark(b) *and* mark(a) $\neq$ null.

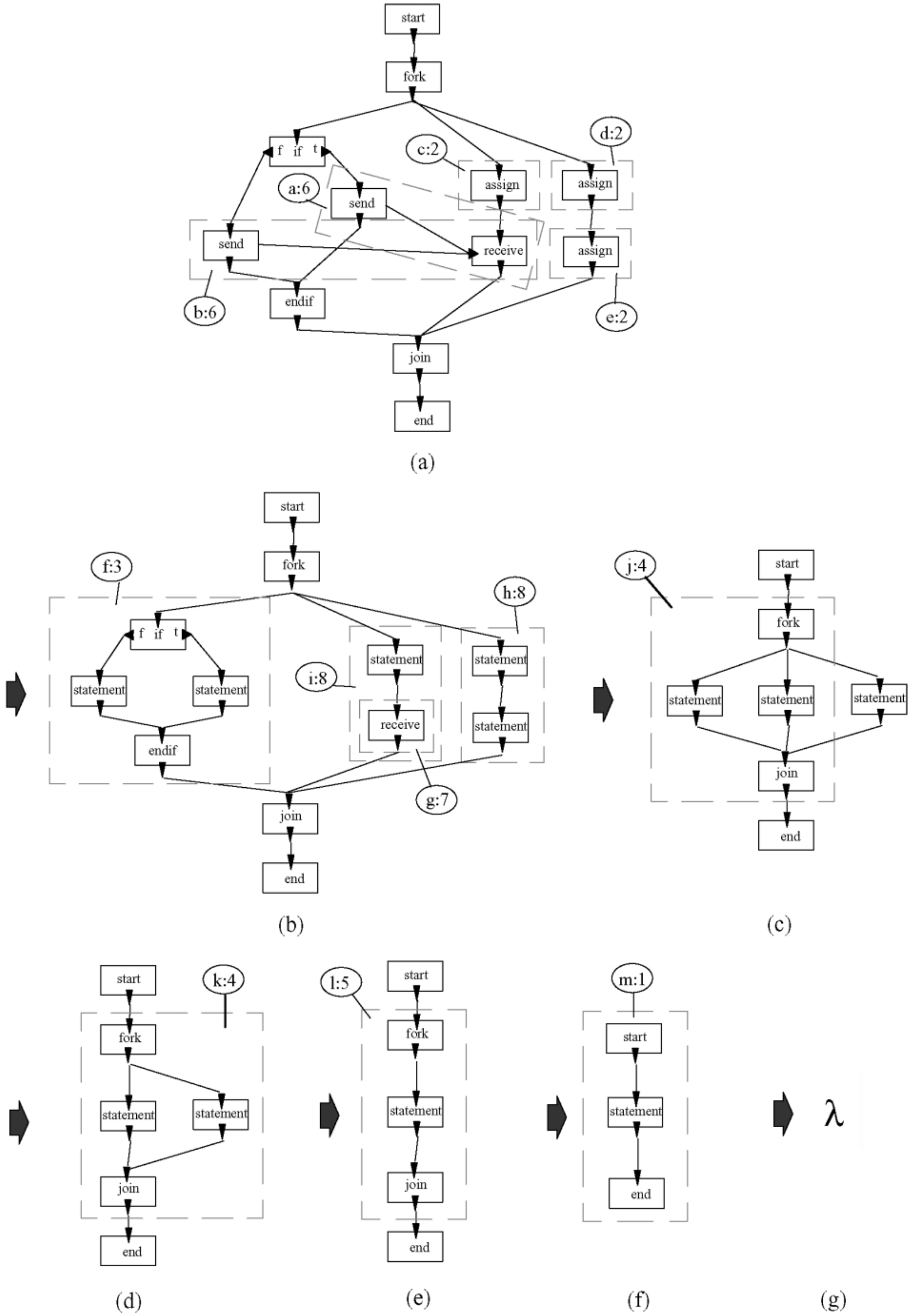


FIGURE 8. Graph transformations (parsing) when productions are applied.

DEFINITION 3.7. *Two graphs* G_1 *and* G_2 *are isomorphic, denoted as* $G_1 \approx G_2$, iff $\exists f : G_1 \to G_2$ *is a bijective mapping such that*

- $\forall n \in G_1.N : n \approx f(n)$*; and*
- $\forall e = (v_a, v_b) \in G_1.E : f(e) = (f(v_a), f(v_b)) \in G_2.E$.

To apply a production to a graph (called a *host graph*), we need to find a sub-graph in the host graph that matches the right graph (or left graph) of the production. Such a matching sub-graph in the host graph is called a *redex*.

DEFINITION 3.8. *A sub-graph* X *of a graph* H *is called a* redex *of a marked graph* G*, denoted as* $X \in \mathrm{Redex}(H, G)$*, iff* $\exists f : G \to X$ *is a bijective mapping and under the mapping*

- $X \approx G$*; and*
- $\forall v \in G.V((\mathrm{mark}(v) = \mathrm{null}) \wedge \forall v_1 \in H(e = (f(v), v_1) \in H \vee e = (v_1, f(v)) \in H) \to e \in X)$.

This definition specifies that a sub-graph X of a graph H can be a redex of a marked graph, G, if and only if X is isomorphic to G and every vertex in X that is isomorphic to an unmarked vertex in G should have edges completely inside X. The definition of a redex eliminates the possibility of any dangling edges resulted from a transformation.

A redex is always related to a mapping function and we will not specify the mapping function if this is clear in the context.

DEFINITION 3.9. *A production* $p := (L, R)$ *is a pair of marked graphs over the same label set, where* $L := (N_L, E_L, M)$ *and* $R := (N_R, E_R, M)$.

A pair of marked graphs in a production has the same mark set. They are called *left graph* and *right graph* respectively.

When a production is applied to a graph, the graph is said to be *transformed* by the application.

DEFINITION 3.10. *Let* X *be a redex of* G *in* H *determined by a bijective mapping* $f : G \to X$*. If* G *and* G' *are the left and right graphs in a production, then the* transformation *of* H *to* H' *after replacing* X *in* H *by* G' *is defined as follows.*

1. *Add* G' *to* H,
2. $\forall v' \in G'.V$ if $\exists v \in G.V$ *such that* $v \doteq v'$*, replace* v' *with* $f(v)$ *(called a* reserved node*), then delete* v'*, and*
3. *delete* X *from* H *except the reserved nodes.*

The result of H with the above operation is H', denoted as $H' = \mathrm{Tr}(H, G, G', X)$. Step 2 ensures that the edges connecting the vertices which are isomorphic to the marked vertices in G are reserved.

Based on the above definition of transformation, the L-application and R-application can be defined as follows.

DEFINITION 3.11. *An* L-application *of a production* $p := (L, R)$ *to a graph* H *is a transformation* $H' = \mathrm{Tr}(H, L, R, X)$*, where* $X \in \mathrm{Redex}(H, L)$*, denoted as* $H \mapsto^X H'$.

DEFINITION 3.12. *An* R-application *of a production* $p := (L, R)$ *to a host graph* H *is a transformation* $H' = \mathrm{Tr}(H, R, L, X)$*, where* $X \in (\mathrm{Redex}(H, R))$*, denoted as* $H \to^X H'$.

3.2. Reserved graph grammar and its properties

We now define the reserved graph grammar and some of its properties.

DEFINITION 3.13. *A* reserved graph grammar gg *is a tuple* (A, P, T, N)*, where* A *is an initial graph,* P *a set of graph-grammar productions,* T *a set of terminal labels with* $e_l \in T$ *(we define all edges to have the same label* e_l*), and* N *a set of non-terminal labels. For* $\forall p := (L, R) \in P$ *and* $\forall l \in T \cup N$:

1. *R is non-empty;*
2. *L and R are over the same label set* $T \cup N$*;*
3. $l \in L_i$ *where* $L_i \subset \{L_0, \ldots, L_n\}$ *is a global layer set and* $L_0 \cap \ldots \cap L_n = \varnothing$*; and*
4. $L < R$ *with respect to the following order of graphs:*

$$G < G' \Leftrightarrow \exists i : |G|_i < |G'|_i \wedge \forall j < i : |G|_j = |G'|_j$$
with $|G|_k$ *defined as* $|\{x \mid x \in G \wedge \mathrm{layer}(x) = k\}|$.

The last condition guarantees that a diagram can be parsed in finite steps with the grammar [9].

For simplicity, given an RGG $gg := (A, P, T, N)$, we use the notation $X \in \mathrm{Redex}(H)$ to denote $\exists p := (L, R) \in P \wedge \exists X : (X \in \mathrm{Redex}(H, R) \vee X \in \mathrm{Redex}(H, L))$, when this is clear in the context.

We denote the sequence of intermediate derivations $H \mapsto^{X1} H_1, H_1 \mapsto^{X2} H_2, \ldots, H_{n-1} \mapsto^{Xn} H_n$ as $H \mapsto^{X1} H_1 \mapsto^{X2} \ldots \mapsto^{Xn} H_n$; or simply $H \mapsto^{X1 \ldots Xn} H_n$. We use $H \mapsto^* H_n$ to denote $H \mapsto^{X1 \ldots Xn} H_n$, where n may be 0 in which case $H = H_n$ and $H \mapsto H$. This notation is also applicable to the R-application $\to$.

DEFINITION 3.14. *Let* $gg := (A, P, T, N)$ *be an RGG, its language* L *is defined by* $L(gg) = \{G \mid A \mapsto^* G$*, where* G *contains only elements with terminal labels*$\}$.

We now prove that the R-application can determine whether a diagram is a language defined by a reserved graph grammar.

LEMMA 3.1. *Let* $gg := (A, P, T, N)$ *be an RGG.* $\exists X_1 : H \mapsto^{X1} H_1 \Rightarrow \exists X_2 : H_1 \to^{X2} H$.

Proof. Let X_1 be a redex determined by a production $p := (L, R)$. According to the definitions of the RGG and the transformation process, if $\exists X_1 : H \mapsto^{X1} H_1$, then H_1 has a redex X_2, which is transformed from X_1 and is determined by R. Hence we have $\exists X_2 : H_1 \to^{X2} H'$. But according to the transformation process, we have $H' \approx H$. So $\exists X_2 : H_1 \to^{X2} H$. □

LEMMA 3.2. *Let* $gg := (A, P, T, N)$ *be an RGG.* $\exists X : H \to^X H_1 \Rightarrow \exists X' : H_1 \mapsto^{X'} H$.

Proof. Similar to Lemma 3.1. □

LEMMA 3.3. *Let* $gg := (A, P, T, N)$ *be a graph grammar, if* $A \mapsto^* G$ *then* $G \rightarrow^* A$.

Proof.

$$
\begin{aligned}
A \mapsto^* G &\Rightarrow A \mapsto^{X1} G_1 \mapsto^{X2} G_2 \mapsto \ldots \mapsto^{Xn} G \\
&\Rightarrow G \rightarrow^{Xn'} G_{n-1}, \ldots, \rightarrow^{X1'} A \quad \text{(Lemma 3.1)} \\
&\Rightarrow \exists G \rightarrow^* A. \qquad \square
\end{aligned}
$$

Similarly we have:

LEMMA 3.4. *Let* $gg := (A, P, T, N)$ *be a graph grammar, if* $G \rightarrow^* A$ *then* $A \mapsto^* G$.

THEOREM 3.1. $G \in L(gg)$ iff $\exists \mathfrak{R} : G \rightarrow^{\mathfrak{R}} A$, *where* $\mathfrak{R}$ *is a list of redexes.*

Proof. It is straightforward from Lemma 3.3 and Lemma 3.4. □

Theorem 3.1 states that R-applications determine exactly the language defined by L-applications. This theorem indicates that if one can find a parsing path (i.e. $\mathfrak{R}$) which transforms a graph to the initial graph, the graph is valid. A recursive algorithm is needed for parsing, which is rather inefficient for parsing a large graph.

4. GRAPH PARSING

Parsing is a process that attempts to reduce a sentence according to a grammar. A reduction (R-application) is performed when a production is applied. Parsing a graph may be more complicated than parsing a piece of text.

4.1. A parsing algorithm

The process of parsing a graph with a grammar consists of selecting a production from the grammar and applying an R-application of the production to the graph; this process continues until no productions can be applied (called a *single parsing path*). If the graph has been transformed into the initial graph after R-applications, the graph is valid (i.e. the parsing succeeds); otherwise, the above process is repeated with different selections (i.e. different parsing paths). If all the possibilities have been tried without success, the graph is invalid.

The first stage of any graph parsing algorithm consists of searching in a graph to find a redex of any production. When such a redex is found, the question arises whether the production should be applied or not. The application of one production may inhibit the application of another production and it subsequently causes the entire parsing process to fail. Therefore, every production instance represents a choice point in the algorithm.

Carrying out the above parsing process is time-consuming as it needs to attempt the R-applications for all productions. We have developed a simple parsing algorithm, called *selection-free parsing algorithm (SFPA)*, which only tries one parsing path, as shown in Figure 9. SFPA is effective for an RGG only in the case that, when parsing any graph with SFPA, if one parsing path fails, any other parsing paths will also fail.

```
Parsing(Graph host){
   while(host!=null){
         matched=false;
         for all p∈P
         {
             redex=FindRedexForR(host, p);
             if(redex!=null){
                R-application(host, p, redex);
                matched=true;
             }
         }
         if(matched==false){
            print("invalid");
            exit(0);
         }
   }
}
```

FIGURE 9. The selection-free parsing algorithm.

More formally, only those RGGs with selection-free productions can use SFPA, where the selection-free property for a production set is defined as follows.

DEFINITION 4.1. *Graph* G *is a merger of graph* G_1 *and graph* G_2, *if*

- G_1 *and* G_2 *are sub-graphs of* G,
- $\forall v \in G.V : v \in G_1.V \vee v \in G_2.V$, *and*
- $\forall e \in G.E : e \in G_1.E \vee e \in G_2.E$.

DEFINITION 4.2. *Let* G_1 *and* G_2 *be graphs,* merge(G_1, G_2) *is a set of mergers of* G_1 *and* G_2.

In the following definition, we will use $p.R$ and $p.L$ to represent the right graph and the left graph of the production p respectively.

DEFINITION 4.3. *Let* P *be a set of productions.* P *is* selection-free*, if, for any* $p_1 \in P$, $p_2 \in P$, R_1, R_2, L_1 *and* L_2 *are graphs isomorphic to* $p_1.R$, $p_2.R$, $p_1.L$ *and* $p_2.L$ *respectively, and*

$$
\begin{aligned}
\forall G \in \text{merge}(R_1, R_2) \wedge R_1 \in \text{Redex}(G, p_1.R) \\
\wedge R_2 \in \text{Redex}(G, p_2.R),
\end{aligned}
$$

we have

$$
\begin{aligned}
\exists G_a, G_{ab}, G_b, G_{ba} :\ & G_a = \text{Tr}(G, p_1.R, p_1.L, R_1) \wedge \\
& G_{ab} = \text{Tr}(G_a, p_2.R, p_2.L, R_2) \wedge \\
& G_b = \text{Tr}(G, p_2.R, p_2.L, R_2) \wedge \\
& G_{ba} = \text{Tr}(G_b, p_1.R, p_1.L, R_1) \wedge \\
& G_{ab} \approx G_{ba}.
\end{aligned}
$$

The definition specifies that a production set is selection-free if a graph with two redexes corresponding to two productions' right graphs is applied by the two productions in different orders, the resulting graphs are the same. According to this definition, an algorithm for checking

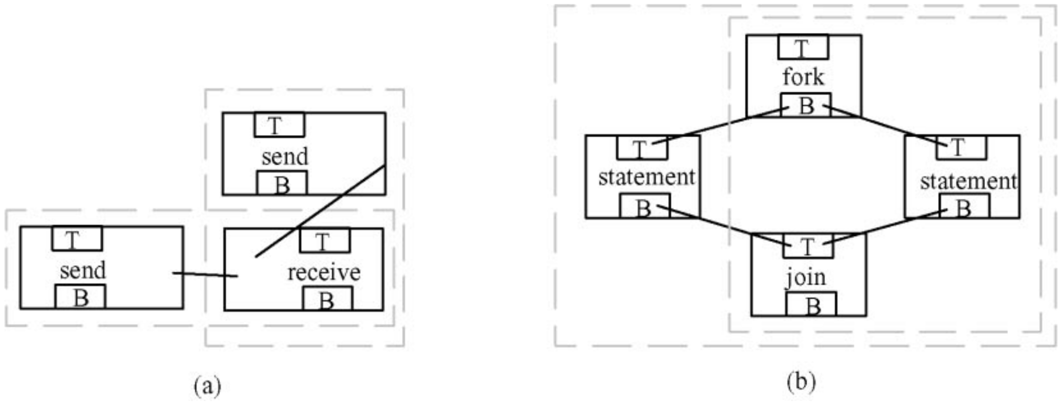


FIGURE 10. Examples of checking the selection-free condition.

whether a reserved graph grammar has a select-free production set can be developed.

To check whether a production set is selection-free, we need to check all the possible combinations of any two productions' right graphs. If one combination does not satisfy the definition, the production set is not selection-free. Figure 10 shows examples of the checking process. In Figure 10a, two copies (enclosed in dashed boxes) of the right graph of Production 6 are merged. According to the embedding rule, different orders of the R-applications to the redexes (i.e. the copies) result in the same graph. Figure 10b appears to be merged by the right graphs of Productions 4 and 5, but the embedding rule determines that no redex of Production 5 exists. So the productions satisfy the selection-free condition.

The production set of the reserved graph grammar illustrated in Figure 7 is selection-free under the definition, so we can use SFPA to parse any diagrams to check if they are valid process flow diagrams. In the following subsection, we will prove that a reserved graph grammar with a selection-free production set can use SFPA to parse diagrams correctly.

4.2. Selection-free grammars

The selection-free property of an RGG means that for a valid graph, any selection of an R-application to the graph can lead to a successful parsing. Obviously, a selection-free RGG can use the selection-free parsing algorithm to parse its languages. The selection-free property of a grammar can be formally defined as:

DEFINITION 4.4. *Let* $gg := (A, P, T, N)$ *be an RGG. If* $\forall(G \to^* G_i \to^* A)$, *for any* $X \in \text{Redex}(G_i)$, $\exists G \to^* G_i \to^X G_{i+1} \to^* A)$, *then gg is said to be* selection-free.

DEFINITION 4.5. *Let* $gg := (A, P, T, N)$ *be an RGG. If for any* $G \to^* A$, $X_a \in \text{Redex}(G) \wedge X_b \in (\text{Redex}(G) \wedge \neg(X_a = X_b))$ *such that*

$$\exists(G \to^{Xa} G_a \to^{Xb} G_{ab}) \wedge \exists(G \to^{Xb} G_b \to^{Xa} G_{ba})$$
$$\Rightarrow G_{ab} \approx G_{ba},$$

then gg is said to be order-free.

The order-free property is similar to the finite Church Rosser property [14] but applicable to context-sensitive graph grammars in that productions are applied to subgraphs rather than to single nodes. For simplicity, if $G \approx G'$, we will use G instead of G' in the sequel. We now show that if the production set of an RGG is selection-free, the RGG is selection-free.

The following lemma implies that a redex of a graph defined in an order-free graph grammar can be applied with an R-application and the graph can be reduced to the initial graph.

LEMMA 4.1. *Let a graph grammar* $gg := (A, P, T, N)$ *be order-free, if* $G \to^* A \wedge \exists X \in \text{Redex}(G)$ *then* $\exists i : G \to^* G_i \to^X G_{i+1} \to^* A$.

Proof.

- $G \to^* A \wedge \exists X \in \text{Redex}(G) \Rightarrow \exists G \to^{X0} G_1 \ldots \to^{Xn} A$, where $n > 0$. We have two cases:
- Case 1: $X_0 = X \Rightarrow \exists G \to^X G_1 \to^* A$.
- Case 2: $X_0 \neq X \Rightarrow \exists G \to^{X0} G_1 \to^* A \wedge \exists X \in \text{Redex}(G_1)$ (Definition 4.5).
- This process can continue:

$$\exists G \to^{X0} G_1 \to^{X1,\ldots,Xm} G_m \to^* A$$
$$\wedge \exists X \in \text{Redex}(G_m)$$

where $m \leq n$.
- As n is finite (the property of the layered definition), we have

$$\exists i \leq n : G \to^* G_i \to^X G_{i+1} \to^* A. \qquad \square$$

Lemma 4.2 presented below implies that a redex can be applied anywhere in the R-application process.

LEMMA 4.2. *Let a graph grammar* $gg := (A, P, T, N)$ *be order-free and* $\forall G_0 \to^* A$. *If* $\exists X \in \text{Redex}(G_0) \wedge \exists G_0 \to^* G_n \to^X G_{n+1}$ *then* $\exists G_0 \to^X G'_1 \to^* G_{n+1}$.

```
Redex FindRedexForR(host, p)
{
       nodeSequence=findNodeSequence(p.R);
       allCandidates=findAllNodeSequences(host, nodeSequence);
       for all candidate∈allCandidates
       {
                 redex=match(candidate, host, p);
                 if(redex!=null)
                          return redex;
       }
       return null;
}
```

FIGURE 11. The algorithm `FindRedexForR`.

Proof.

$$
\begin{aligned}
& G0 \to^* G_n \to^X G_{n+1}\\
& \Rightarrow \exists G_0 \to^{X0} G_1 \to^{X1} G_2 \to \ldots \to^{Xn-1} G_n \to^X G_{n+1}\\
& \Rightarrow \exists G_{n-1} \to^{Xn-1} G_n \to^X G_{n+1}\\
& \Rightarrow \exists G_{n-1} \to^X G'_n \to^{Xn-1} G_{n+1} \quad \text{(Definition 4.5)}\\
& \Rightarrow \exists G_{n-2} \to^X G'_{n-1} \to^{Xn-2} G'_0 \to^{Xn-1} G_{n+1}\\
& \Rightarrow \ldots\\
& \Rightarrow \exists G_0 \to^X G'_1 \to^{X0} G'_2 \to \ldots\\
& \quad \ldots \to^{Xn-2} G'_n \to^{Xn-1} G_{n+1}\\
& \Rightarrow \exists G_0 \to^X G'_1 \to^* G_{n+1}. \qquad \square
\end{aligned}
$$

THEOREM 4.1. *If $gg := (A, P, T, N)$ is order-free, then gg is selection-free.*

Proof.

$$
\begin{aligned}
& G \to^* G_i \to^* A \wedge X \in \text{Redex}(G_i)\\
& \quad \Rightarrow \exists G \to^* G_i \to^* G_j \to^X G_{j+1} \to^* A \quad \text{(Lemma 4.1)}\\
& \quad \Rightarrow \exists G \to^* G_i \to^X G_{i+1} \to^* G_{j+1} \to^* A \text{ (Lemma 4.2)}\\
& \quad \Rightarrow \exists G \to^* G_i \to^X G_{i+1} \to^* A. \qquad \square
\end{aligned}
$$

THEOREM 4.2. *Let $gg := (A, P, T, N)$, if P is selection-free, then gg is order-free and thus selection-free.*

Proof.

- Suppose $G \to^* A \wedge X_1 \in \text{Redex}(G) \wedge X_2 \in \text{Redex}(G)$, we have $\exists p_1 \in P \wedge \exists p_2 \in P$ so that $X_1 \approx p_1.R$ and $X_1 \approx p_1.R$.
- Since P is selection-free and $(X_1 \cup X_2) \subseteq G$, G can be transformed by applying X_1 and X_2 in any order and the resulting graphs are the same.
- The transformation process derives that $\forall G \to^* A$ if $X_1 \in \text{Redex}(G) \wedge X_2 \in \text{Redex}(G) \wedge \neg(X_1 = X_2)$ then $\exists G \to^{X2} G_1 \to^{X1} G_2 \wedge \exists G \to^{X1} G'_1 \to^{X2} G_2$.
- Hence gg is order-free.
- According to Theorem 4.1, gg is selection-free. □

Theorem 4.2 says that if the production set of an RGG is selection-free, the RGG is selection-free and can use SFPA to parse its languages.

4.3. Parsing complexity

To study the time complexity of SFPA, we construct an algorithm `FindRedexForR(G, p)` shown in Figure 11, which is the main part of the SFPA. To explain the algorithm, we first give some definitions.

DEFINITION 4.6. *A node sequence of a graph G is an ordered list of all the nodes in G.*

DEFINITION 4.7. *Let $L_1 = [n_{11}, n_{12}, \ldots, n_{1k}]$ and $L_2 = [n_{21}, \ldots, n_{2m}]$ be ordered node lists. L_1 is isomorphic to L_2 if $m = k \wedge n_{1i} \approx n_{2i}$ where $i \in \{1, \ldots, m\}$.*

THEOREM 4.3. *The algorithm* `FindRedexForR(G, p)` *has $O(|G|^m)$ time complexity, where m is the maximum number of nodes in any right graph of a set of productions.*

Proof. The function `findNodeSequence(p.R)` finds a node sequence of the right graph of a production p. It lists all the nodes of $p.R$ in a certain order. For a graph grammar, the number of nodes in the right graph of a production is given, so the function takes $O(1)$.

The function `findAllNodeSequences(host, nodeSequence)` collects all the possible node sequences from the host, each of which is isomorphic to `nodeSequence`. For a graph G, the number of all possible node sequences, each having m nodes, is k^m, where k is the number of nodes in G. So the time complexity for the function `findAllNodeSequences` is $O(|G|^m)$.

The function `match` checks whether a candidate in the host is a redex of the production p, if so, the candidate is returned as a redex, otherwise, a null is returned. The time complexity for the function `match(candidate, host, p)` is $O(m)$.

As the number of `allCandidates` is no more than $|G|^m$, the maximum time taken is $O(|G|^m)$. □

THEOREM 4.4. *The time complexity of SFPA is $O(|G|^{m+1})$, where G is a graph to be parsed by SFPA and m is the maximum number of nodes of all the right graphs of productions.*

Proof. Suppose that $T(k) = (2C)^k A_0 + (2C)^{k-1} A_1 + \ldots + (2C)A_{k-1} + A_k$ is a function and next() is an operation applicable to $T(k)$, where A_i, C and k are integers, and

$C > 0, k \geq 0, A_i \geq 0, 0 \leq i \leq k$.

Let

$$\begin{aligned} T(k).\text{next}(i) &= (2C)^k(A_0) + (2C)^{k-1}(A_1) + \ldots \\ &\quad + (2C)^{k-i+1}(A_{i-1}) + (2C)^{k-i}(A_{i-1}) \\ &\quad + (2C)^{k-i-1}(A_{i+1} + C) + \ldots \\ &\quad + (2C)(A_{k-1} + C) + (A_k + C) \end{aligned}$$

be $T(k)$ after i executions of next() operation, where $A_i > 0$, $C > 0$, $k \geq 0$, we have

$$\begin{aligned} T(k).\text{next}(i) &= (2C)^k A_0 + (2C)^{k-1} A_1 + \ldots \\ &\quad + (2C)^{k-i+1} A_{i-1} + (2C)^{k-i} A_i \\ &\quad + (2C)^{k-i-1} A_{i+1} + \ldots \\ &\quad + A_k - ((2C)^{k-i} - (2C)^{k-i-1} C - \ldots - (2C)C - C) \\ &= T(k) - (2^{k-i} C^{k-i} - 2^{k-i-1} C^{k-i} - \ldots - 2C^2 - C). \end{aligned}$$

As

$$\begin{aligned} &(2^{k-i} C^{k-i} - 2^{k-i-1} C^{k-i} - \ldots - 2C^2 - C) \\ &\quad \geq (2^{k-i} C^{k-i} - 2^{k-i-1} C^{k-i} - \ldots - 2C^{k-i} - C^{k-i}) \\ &\quad = C^{k-i}(2^{k-i} - 2^{k-i-1} - \ldots - 2 - 1) \\ &\quad = C^{k-i} > 0, \end{aligned}$$

we have $T(k) > T(k).\text{next}(i)$. (1)

This means $\exists n : n \leq T(k)$ such that $T(k)$ will be zero after n executions of its 'next' operation.

Let $gg := (A, P, T, N)$ be a reserved graph grammar and $G \rightarrow^* A$.

A graph G can be mapped to

$$T(k) = (2C)^k A_0 + (2C)^{k-1} A_1 + \ldots + (2C) A_{k-1} + A_k,$$

where $A_i = |G|_i$, k equals the maximum number of layers, and C is the maximum number of nodes of all the right graphs of productions. We denote $G.T(k)$ as the $T(k)$ that is mapped from G.

Suppose $G \rightarrow^X G'$. According to the definition of the grammar layer and the transformation rules, we have $G < G'$, where $\exists i : |G|_i < |G'|_i \wedge \forall j < i : |G|_j = |G'|_j$ with $|G|_k$ defined as $|\{x | x \in G \wedge \text{layer}(x) = k\}|$.

This means that in the layer i, the element number of G' is less than the element number of G by $|G|_i - |G'|_i$ elements. In a layer larger than i, the number of additional elements are no more than C. So after the transformation,

$$\begin{aligned} G'.T(k) &\leq (2C)^k(A_0) + (2C)^{k-1}(A_1) + \ldots \\ &\quad + (2C)^{k-i+1}(A_{i-1}) + (2C)^{k-i}(A_i - (|G|_i - |G'|_i)) \\ &\quad + (2C)^{k-i-1}(A_{i+1} + C) + \ldots + (2C)(A_{k-1} + C) \\ &\quad + (A_k + C) \leq G.T(k).\text{next}(i). \end{aligned}$$

So we have $\exists i : G'.T(k) \leq G.T(k).\text{next}(i)$.

For any graph G, $G.T(k) \geq 0$, so according to (1), $G \rightarrow^* A$ must finish within $G.T(k)$ steps. As

$$\begin{aligned} G.T(k) &= (2C)^k A_0 + (2C)^{k-1} A_1 + \ldots + (2C) A_{k-1} \\ &\quad + A_k \leq (2C)^k (A_0 + A_1 + \ldots + A_k) = (2C)^k |G|, \end{aligned}$$

according to Theorem 4.3, the time complexity of the algorithm SFPA is $(2C)^k|G|^*\mathrm{O}(|G|^m) = \mathrm{O}(|G|^{m+1})$. □

We now discuss the space complexity of SFPA. We implement an index for each element of a graph. The indices are organized as follows: they are listed in the same array if they refer to the graph elements that have the same label. Thus, a graph is a set of arrays, each of which is a list of elements with the same label. A `nodeSequence` (in Figure 11) can be implemented as a set of pointers, each pointing to an element of an array. The next node sequence can be found by moving pointers in a proper way, and a candidate of a redex is the pointer set. In this case, the extra space is unnecessary except for the pointers. Thus, SFPA has a linear space complexity.

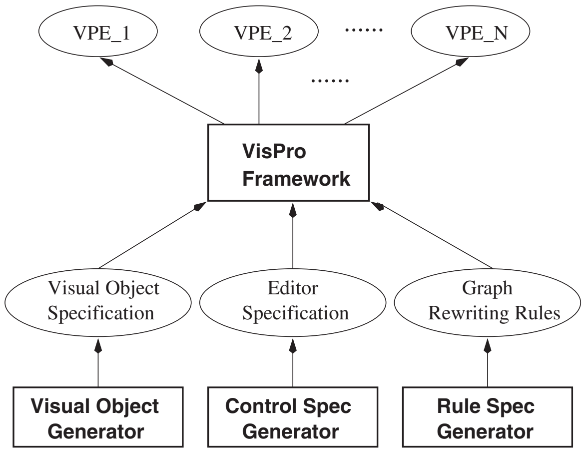


FIGURE 12. Constructing VPEs with VisPro.

5. APPLICATION IN A VPL GENERATION SYSTEM

Reserved graph grammars have been used in a visual language generation system called VisPro [12]. VisPro provides a generic VPE and a set of visual programming tools for constructing domain-oriented VPEs. The construction process is similar to the textual language construction process using Lex/Yacc. The process can be described as customization. The generic VPE can be customized to any domain VPEs once the domain specifications are provided through these tools. Figure 12 shows the generation process, which is supported by the following three tools:

- visual object generator—that is used to specify visual objects with desired appearances to be used in the target visual language;
- rule specification generator—that is used to provide parsing specifications for the target visual language in the form of graph rewriting rules based on the reserved graph grammar; and
- control specification generator—that is used to specify the control commands for each generated visual

object manipulated in a visual editor, which is to be automatically generated.

In VisPro, the object-oriented language Java serves as a lower level specification tool for details that may not be effectively or accurately specified in these visual specification tools. This arrangement allows us to precisely construct effective visual programming environments.

The tools are metavisual programming languages that are used to specify domain VPEs through direct manipulation. First, the visual object generator is used to construct visual objects—it creates the appearance of each visual object, and attaches to the visual object a specification of its behavior produced by the control specification generator (see below), or another visual program as its logical function. The user then uses the control specification generator to specify the behaviors of constructed visual objects. The specifications will define and automatically generate a visual editor for the target visual language. Finally, with the rule specification generator, the user can describe the grammar of the visual language according to the reserved graph grammar. The rules can be specified as either graphical productions or textual ones written in Java. Having obtained all the required specifications, the generic VPE becomes customized to the desired domain VPE that integrates the target visual language editor and compiler.

With VisPro, a complete VPL can be specified by a lexicon definition and a grammar specification. A lexicon definition describes the VPL's visual objects and the editor with which the visual objects can be used to create a program. A grammar specification (syntax and semantics) defines whether the program is valid and what it means. A visual programming environment is created automatically based on the definition and the specification.

6. RELATED WORK

Growing interest in visual languages has motivated research in the specification and parsing of multi-dimensional structures. Several specification methods have been proposed and proven to be useful in practical applications. Examples include Web and array grammars [15], positional grammars [16], relational grammars [7, 17], unification grammars [18], attributed multiset grammars [4], constraint multiset grammars [19], and layered graph grammars [9]. In this section, we discuss some of the related grammars and compare them with reserved graph grammars.

The relational grammars of Wittenburg [7] are restricted to relational structures, where relationships of the same type define partial orders. Ferrucci *et al.* [17] proposed 1NS-RG grammars, that are adapted from the Boundary NLC graph grammars of Rozenberg and Welzl [2]. The right-hand sides of productions in a 1NS-RG grammar may not contain non-terminals as neighbors in order to guarantee local confluence. Parsing can be done in polynomial time if the generated graphs are all connected and the maximum number of edges at any single vertex is known in advance. This latter restriction also applies to Brandenburg's DNELC graph grammar [14].

Marriott's constraint multiset grammars [19] provide context elements. Introducing 'not exits' constraints prevents any possible overlap between the right-hand sides of productions, but also makes syntax specifications deterministic. Golin's picture layout grammars [4] allow productions with one non-terminal on the left-hand side and at most two terminals or non-terminals on the right-hand side.

Rekers and Schürr have shown that it is difficult for the aforementioned grammars to generate abstract syntax graphs for connected ER diagrams [9]. They proposed a context-sensitive grammar formalism, known as LGGs [9], which can specify a wide range of visual languages. The graphical specifications of LGGs are more intuitive and easier to understand than textual grammars.

6.1. Improvements on the LGG

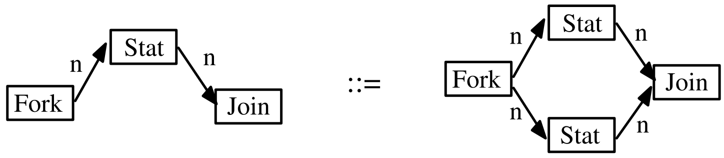
Figures 13a, b show two productions of the layered graph grammar for parsing the `fork` statement, where the elements `B?` and `T?` (wildcards) are used as the context elements. For instance, `B?` means `begin`, `fork`, or `if`, as shown in Figure 13c. After a transformation, say R-application, the relationships between the new node `Stat` and the host graph are determined by the `B?` and `T?`, which are part of the host graph. New nodes can be embedded into the host graph when they are linked with the matching nodes labeled with `B?` and `T?`. Without the wildcards, the number of productions required will be multiplied [9].

The productions in Figures 13a, b lead to ambiguity. For example, if a graph has a redex of the right graph in the production in Figure 13a, it also has a redex of the right graph in Figure 13b because the right graph in Figure 13a is a part of the right graph in Figure 13b. Applications of the productions in LGGs with different redexes may produce different results. A complex algorithm is then needed to ensure that all possible applications of productions are attempted.

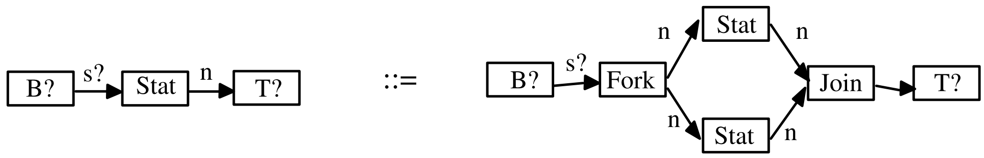
A reserved graph grammar can avoid the ambiguity. As a result, its parsing algorithm can be simple and efficient. Therefore, compared with the layered graph grammar [9], the reserved graph grammar has the following three major improvements:

- it avoids the use of wildcards;
- it simplifies the specification through an embedding rule; and
- parsing an unambiguous reserved graph grammar can be done in polynomial time.

As discussed earlier, our RGGs are based on LGGs, and improve over LGGs. Apart from the improvements discussed above, the major differences between the RGG formalism and the LGG formalism are that the former can be implemented more efficiently using the presented parsing algorithm; and that it uses simple embedding rules rather than context elements (as used in the latter) so that grammar specifications are simplified. Table 1 compares the discussed grammars with RGGs.



(a)



(b)

B?∈{begin, fork, if}

T?∈{end, assign, fork, join, send, receive, if}

s?∈{n, f, t}

(c)

FIGURE 13. Productions in a LGG with different embedding mechanisms.

TABLE 1. A comparison of graph grammars discussed.

Grammar	Left-hand side	Right-hand side	Context	Embedding rules	Additional restrictions	Complexity
Relational G	Non-terminal	Relational structure	No	Yes	Explicit vertex order	Exponential
1NS-RG	Non-terminal	Relational structure	No	Yes	Bounded degree; no non-terminal neighbors	Exponential
Boundary NLC GG	Non-terminal	Graph	No	Yes	Bounded degree; no non-terminal neighbors	Exponential
Constraint multiset G	Non-terminal	Multiset	Yes	Implicit	Deterministic	Polynomial
Picture layout G	Non-terminal	Max two (non-) terminals	1 terminal	Implicit	Finite set of attribute values	Polynomial
Layered GG	Graph	Graph	Graph	No	Layering	Exponential
Reserved GG	Graph	Graph	Graph	Yes	Selection-free	Polynomial

The six attributes used to distinguish various grammars in the table are proposed by Rekers and Schürr [9]. They serve our purpose well in comparing these grammars. Minas [20] has recently adapted RGGs to the DiaGen hypergraph environment [21]. The selection-free constraint imposed in RGGs is relaxed to allow more types of hypergraphs to be specified. However, additional information has to be provided in the form of negative application conditions (NACs). A production with a matching left-hand side is not applicable if one of its NACs is satisfied. The addition of NACs modifies the original grammar and it is unclear how additional complexity is introduced into the parsing process.

7. CONCLUSION

This paper has presented the reserved graph grammar (RGG), which can be used to specify grammars of diagrammatic visual languages. An RGG is a collection of graph rewriting rules represented labeled graphs. It is context-sensitive and its right and left graphs can have an arbitrary number of nodes and edges. The grammar uses an enhanced node structure with a marking mechanism in its graph representation. It is this structure that makes an RGG effective in specifying a wide range of visual languages and efficient in parsing a certain class of visual languages. Although the time complexity of the parsing algorithm for a general RGG is exponential, parsing a selection-free reserved graph grammar can be done in polynomial time. The paper has presented such a polynomial time parsing algorithm and proved its time and space complexities. To ensure that a reserved graph grammar is unambiguous, we also proposed a checking criterion and proved its correctness.

There have been some applications of RGGs; for example, for generating a visual language for modeling distributed systems [22]. A wide range of applications, such as interpreting hand-written mathematical notations [23], has been reported for using layered graph grammars [24], upon which RGGs improve. We are currently investigating the application of RGGs to multimedia authoring and Web site design and maintenance. Another future direction is to develop a diagram layout mechanism based on geometrical graph rewriting. In such a rewriting rule (production), the left graph represents a desired layout while the right graph represents a logical connection.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insightful comments and corrections that helped us to improve the final presentation.

REFERENCES

[1] Rozenberg, G. (ed.) (1997) *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*, Vol. 1. World Scientific, Singapore.

[2] Rozenberg, G. and Welzl, E. (1986) Boundary NLC graph grammars—basic definitions, normal forms, and complexity. *Inform. Control*, **69**, 136–167.

[3] Bunke, H. and Haller, B. (1989) A parser for context free plex grammars. In *Proc. 15th Int. Workshop on Graph-Theoretic Concepts in Computer Science. Lecture Notes in Computer Science*, **411**, 136–150. Springer, Berlin.

[4] Golin, E. J. (1991) *A Method for the Specification and Parsing of Visual Languages*. PhD Thesis, Brown University.

[5] Kaul, M. (1982) Parsing of graphs in linear time. In *Proc. 2nd Int. Workshop on Graph Grammars and Their Application to Computer Science. Lecture Notes in Computer Science*, **153**, 206–218. Springer, Berlin.

[6] Wills, L. M. (1992) *Automated Program Recognition by Graph Parsing*. PhD Thesis, MIT Artificial Intelligence Laboratory, Cambridge, MA, Technical Report 1358.

[7] Wittenburg, K. (1992) Earley-style parsing for relational grammars. In *Proc. 8th IEEE Workshop on Visual Languages*, September 15–18, Seattle, WA, pp. 192–199. IEEE Computer Society Press, Los Alamitos, CA.

[8] Wittenburg, K. and Weitzman, L. (1998) Relational grammars: theory and practice in a visual language interface for process modeling. In Marriott, K. and Meyer, B. (eds), *Visual Language Theory*, pp. 193–217. Springer, Berlin.

[9] Rekers, J. and Schürr, A. (1997) Defining and parsing visual languages with layered graph grammars. *J. Visual Languages Comput.*, **8**, 27–55.

[10] Vermeulen, J. T. (1996) *Viability of a Parsing Algorithm for Context-sensitive Graph Grammars*. Technical Report, Leiden University.

[11] Zhang, D.-Q. and Zhang, K. (1998) VisPro: a visual language generation toolset. In *Proc. 14th IEEE Symp. on Visual Languages*, September 1–4, Halifax, NS, Canada, pp. 195–202. IEEE Computer Society Press, Los Alamitos, CA.

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

[13] Chok, S. and Marriott, K. (1995) Parsing visual languages. In *Proc. 18th Australasian Computer Science Conf.*, Glenolg, South Australia, pp. 90–98. CSA Press, Sydney.

[14] Brandenburg, F. J. (1988). On polynomial time graph grammars. In *Proc. 5th Conf. on Theoretical Aspects of Computer Science. Lecture Notes in Computer Science*, **294**, 227–236. Springer-Verlag, Berlin.

[15] Rosenfeld, G. (1976) Array and Web grammars: an overview. In Lindenmayer, A. and Rozenberg, G. (eds), *Automata, Languages, and Development*. North-Holland, Amsterdam.

[16] Costaglioga, G., De Lucia, A., Orefice, S. and Tortora, G. (1995) Automatic generation of visual programming environments. *IEEE Computer*, **28**, 56–66.

[17] Ferrucci, F., Tortora, G., Tucci, M. and Vitellio, G. (1994) A predictive parser for visual languages specified by relational grammars. In *Proc. 10th IEEE Symp. on Visual Languages*, October 4–7, St. Louis, MI, pp. 245–252. IEEE Computer Society Press, Los Alamitos, CA.

[18] Wittenburg, K., Weitzman, L. and Talley, J. (1991) Unification-based grammars and tabular parsing for graphical languages. *J. Visual Languages Computing*, **2**, 347–370.

[19] Marriott, K. (1994) Constraint multiset grammars. In *Proc. 10th IEEE Symp. on Visual Languages*, St. Louis, MI, October 4–7, pp. 118–125. IEEE Computer Society Press, Los Alamitos, CA.

[20] Minas, M. (2000) Hypergraphs as a uniform diagram representation model. *Lecture Notes in Computer Science*, **1764**, 281–295.

[21] Minas, M. and Viehstaedt, G. (1995) DiaGen: a generator for diagram editors providing direct manipulation and execution of diagrams. In *Proc. 11th IEEE Symp. on Visual Languages*, September 5–9, Darmstadt, Germany, pp. 203–210. IEEE Computer Society Press, Los Alamitos, CA.

[22] Zhang, D.-Q. and Zhang, K. (1998) On a visual distributed programming environment and its construction by a meta toolset. In *Proc. 10th Int. Conf. on Software Engineering and Knowledge Engineering*, June 18–20, San Francisco, CA, pp. 347–350. Knowledge Systems Institute, Skokie IL.

[23] Blostein, D. and Grbavec, A. (1997) Recognition of mathematical notation. In Bunke, H. and Wang, P. (eds), *Handbook of Character Recognition and Document Image Analysis*, pp. 557–582. World Scientific, Singapore.

[24] Blostein, D. and Schürr, A. (1998) Visual modeling and programming with graph transformations. In *Tutorial at 14th IEEE Symp. on Visual Languages*, September 1–4, Halifax, NS, Canada.