Attributed Programmed Graph Grammars and Their Application to Schematic Diagram Interpretation

HORST BUNKE

Abstract—Attributed programmed graph grammars are introduced in this paper and their application to the interpretation of schematic diagrams is proposed. In contrast with most of the approaches to syntactic pattern recognition, where the grammar controls a parser, the grammar in our system is used as a generative tool. Two classes of diagrams are studied, namely circuit diagrams and flowcharts. The task is in either case to extract a description from an input diagram.

Index Terms—Attributed grammars, circuit diagrams, error correction, flowcharts, graph grammars, line drawing interpretation, programmed grammars, syntactic pattern recognition.

I. Introduction

SYNTACTIC methods in pattern recognition and image processing have become well established during the last decade. The basic principle in syntactic pattern recognition is the representation of a class of patterns by a formal grammar which controls the classification and/or the extraction of a description from an input pattern. The subject is treated in several textbooks [1]-[3]. Applications of syntactic methods in pattern recognition are reported in [4]. Recent developments are summarized in [5].

Most of the approaches to syntactic pattern recognition are based on string grammars, i.e., a string grammar is used for describing the class of patterns under consideration and the input patterns are represented by strings after preprocessing and primitive component extraction. Due to the two-dimensional nature inherent to many interesting pattern classes, e.g., images, there has always been an interest in generalizing the concepts developed for one-dimensional strings to two-dimensional structures. The early efforts known from the literature are pseudo-two-dimensional in their nature, i.e., two-dimensional relations between various objects are represented by one-dimensional strings using special operators for describing two-dimensionality. Well-known examples are [6], [7].

Trees have been proposed in the next generation of approaches to two-dimensional syntactic pattern recognition [8]. It has turned out that trees are a powerful generalization of strings. What makes them very attractive for practical applications is the fact that the parsing can be efficiently performed.

Manuscript received July 22, 1981; revised March 8, 1982. This work was supported by a scholarship from the German Academic Exchange Service, NATO-Forschungsstipendium des DAAD, while the author was on leave at the School of Electrical Engineering, Purdue University, West Lafayette, IN.

The author is with the Lehrstuhl fuer Informatik 5, University of Erlangen, Erlangen, West Germany.

A straightforward generalization of trees is graphs. While the representation of knowledge by means of static graphs is fairly popular in pattern recognition, only a few applications of graph grammars are known from the literature [9]-[11]. This fact is possibly due to the unefficiency of parsing graph grammars [12]. Graph grammars have been originally introduced in [13]. Alternative approaches have been proposed in [14], [15]. In contrast with the pattern recognition domain, there are several areas of computer science where graph grammars have been proven to be an useful tool, e.g., databases or formal semantics of programming languages. A survey on graph grammars including applications is given in [16].

A twofold extension of conventional graph grammars is proposed in this paper. First, a facility for explicitly programming the application order of productions is provided and secondly, the underlying graphs are augmented by attributes. Programmed strings grammars have been introduced and interesting properties of the resulting classes of languages have been shown in [17]. Results on the generative power of programmed graph grammars are presented in [18]. The idea of augmenting a grammar by attributes is due to [19] where the semantics of string languages are studied. Stimulated by this work, attributed string grammars have been widely used in pattern recognition [20], [21].

II. Attributed Programmed Graph Grammars

Attributed programmed graph grammars are formally introduced in this chapter. The underlying graphs are finite with labeled nodes and edges. Let V and W be two alphabets for labeling the nodes and edges, respectively.

Definition 1: An *unattributed graph* (u-graph) is a 3-tuple $g = (N, E, \lambda)$ where

1) N is the finite set of nodes,

2) $E = (E_w)_{w \in W}$ is a tuple of relations $E_w \subseteq N \times N$ for each $w \in W$,

3) λ: $N \to V$ is the node labeling function.

A pair $(n, n') \in E_w$ is interpreted as a directed edge with label w from node n to node n'. In the sequel, we consider two sets of attributes, namely the set A of node attributes and the set B of edge attributes, where an attribute is a function associating attribute values with nodes or edges.

Definition 2: An *attributed graph* (a-graph) is a 5-tuple $g = (N, E, \lambda, \alpha, \beta)$, where

1) N, E, λ are the same as in Definition 1,

2) α: $N \to 2^A$ is a function which associates a set of node attributes with each node,

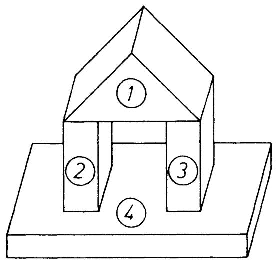


Fig. 1. A simple scene.

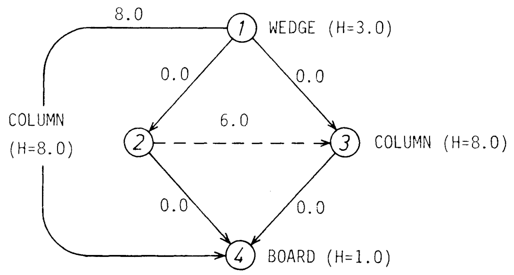


Fig. 2. A possible description of the scene in Fig. 1.

3) $\beta = (\beta_w)_{w \in W}$ is a tuple of functions β_w: $E_w \to 2^B$ associating a set of edge attributes with each w-edge, for each $w \in W$.

According to Definition 2, an attributed graph consists of a pure syntactic or structural part given by the entities N, E, λ and a semantic component which is defined by the attributes of each node and edge. The prefixes u- and a- are used when we have to make a clear distinction between unattributed and attributed graphs. The term "graph" refers to both unattributed and attributed graphs. Definitions in this chapter referring to a-graphs can also be applied to u-graphs by simply neglecting those conditions which affect the attributes. Let Γ denote the set of all attributed graphs.

Example 1: A possible description of the scene in Fig. 1 is given by the a-graph in Fig. 2. For this graph we have the following.

1) V = {WEDGE, COLUMN, BOARD}.

2) W = {RIGHT-OF, BELOW}. An edge labeled with "BELOW" is diagrammatically represented in Fig. 2 by a solid line while dashed lines represent RIGHT-OF-edges.

3) A = {HEIGHT}.

4) B = {DISTANCE}.

5) $\alpha(n)$ = {HEIGHT} for each node n. The notation H = 3.0 at node 1 means, for example, the height of the wedge having a value of 3.0.

6) $\beta(e)$ = {DISTANCE} for each edge e. The numerical values associated with the edges in Fig. 2 are the values of the edge-attribute DISTANCE.

All attribute values given in this example are hypothetical and should not be tried to be interpreted on any scale.

An a-graph g' is a subgraph of an a-graph g' (shorthand notation $g' \subseteq g$) if all nodes and all edges of g' also belong to g. Additionally, corresponding nodes and edges in g and g' must be identical with respect to their labels and attributes. Given two graphs g and g', with $g' \subseteq g$, let $g - g'$ denote the graph which remains after removing g' from g. The edges between the subgraph g' and the host graph $g - g'$ are called the embedding of g' in g (shorthand notation EMB (g', g)).

Definition 3: A *production* is a 5-tuple $p = (g_1, g_r, T, \pi, F)$ where

1) g_1 and g_r are u-graphs, the left-hand and right-hand side, respectively;

2) $T = \{L_w, R_w | w \in W\}$ is the embedding transformation with $L_w, R_w \subseteq N_1 \times N_r$;

3) π: $\Gamma \to$ {TRUE, FALSE} is the applicability predicate; and

4) F is a finite set of partial functions f_a: $N_r \to D_a$ and f_b: $E_r \cup \text{EMB}(g_r, g) \to D_b$ with $a \in A$, $b \in B$, and $g \in \Gamma$. The f_a are the node attribute and the f_b are the edge attribute transfer functions.

In this definition, N_1 and N_r denotes the nodes of g_1 and g_r, respectively, while E_r denotes the edges of g_r. The left-hand and right-hand side is a straightforward extension of the string grammar case. The embedding transformation T takes into account the peculiarities arising from subgraph replacement. Removing the left-hand side of a production from a graph, also the embedding of the left-hand side in the host graph vanishes. Consequently, the question arises how to embed the right-hand side in the (unaltered) host graph. The embedding transformation provides a solution of this problem. For each $w \in W$ we specify sets L_w and R_w consisting of pairs (n, n') of nodes with $n \in N_1$, and $n' \in N_r$. A pair (n, n') causes an edge between node n in the left-hand side and node n'' in the host graph to be transformed into an edge between node n' in the right-hand side and node n'', which has not been changed. L_w is for the transformation of edges terminating in the left-hand side while R_w controls the transformation of edges originating in the left-hand side. By means of the applicability predicate π, certain conditions can be formulated which the left-hand side of a production must fulfill in order to be replaced by the right-hand side. This enables us to express constraints which cannot be formulated in terms of nodes and edges only, e.g., constraints on the attributes. The functions f_a and f_b are a straightforward extension of the way the attributes are transferred in attributed string grammars. Particularly, $f_a(n)$ specifies the value of attribute a at node n and $f_b(e)$ defines the value of attribute b at edge e in the right-hand side.

Definition 4: The direct derivation of a graph g' from a graph g by means of a production p is defined by the following procedure (shorthand notation $g \overset{p}{\to} g'$).

1) Check whether the left-hand side of the production occurs as subgraph in g and check whether the applicability predicate is TRUE for this occurrence. If both conditions are fulfilled go to step 2).

2) Replace the left-hand by the right-hand side.

3) Transform the embedding of the left-hand side in g into that of the right-hand side in g'.

4) Attach attributes to the inserted right-hand side according to the functions f_a and f_b.

Example 2: The graph in Fig. 3 is a possible description of a scene identical to that in Fig. 1 up to the fact that the wedge on top of the columns is missing. The insertion of that wedge

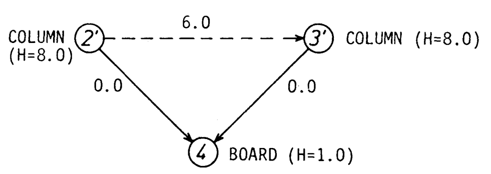


Fig. 3. A possible description of another scene.

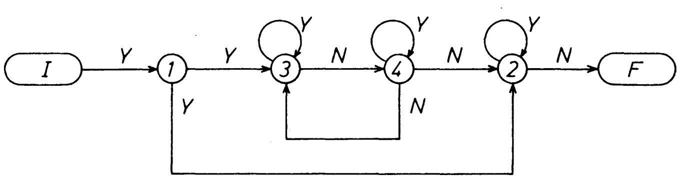


Fig. 5. Example of a control diagram.

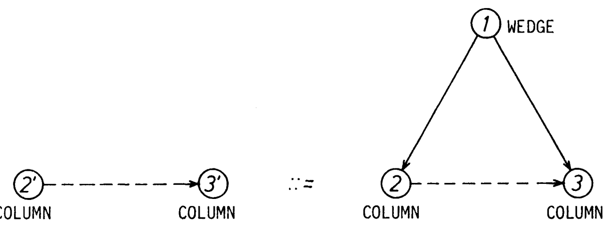


Fig. 4. Left-hand and right-hand side of a production for inserting a wedge.

can be described by a production, the left-hand and right-hand side of which is shown in Fig. 4. Let the other components of this production be defined in the following way.

1) Applicability predicate π: HEIGHT $(2') =$ HEIGHT $(3')$.

2) Embedding transformation: $T = \{R_{\text{BELOW}}\}$ with

$R_{\text{BELOW}} = \{(2', 2), (3', 3), (2', 1)\}$.

3) Attribute transfer functions $F = \{f_H, f_{\text{DIS}}\}$ with $H =$ HEIGHT, DIS = DISTANCE:

$f_H(2) = H(2'); f_H(3) = H(3'); f_H(1) = 3.0.$

$f_{\text{DIS}}((2, 3)_{\text{RIGHT-OF}}) = \text{DIS}((2', 3')_{\text{RIGHT-OF}}),$

$f_{\text{DIS}}((1, 2)_{\text{BELOW}}) = f((1, 3)_{\text{BELOW}}) = 0.0,$

$f_{\text{DIS}}((2, -)_{\text{BELOW}}) = f_{\text{DIS}}((3, -)_{\text{BELOW}}) = 0.0,$

$f_{\text{DIS}}((1, -)_{\text{BELOW}}) = H(2).$

The application of this production to the graph in Fig. 3 yields the graph in Fig. 2. The applicability predicate expresses the physical constraint that both columns must have the same height when inserting the wedge.

In the attribute transfer functions, $(x, y)_w$ denotes a w-edge from node x to node y. Similarly, $(x, -)$ denotes a w-edges belonging to the embedding, i.e., leaving the right-hand side. For example, the edge attribute values 0.0 in Fig. 2 are generated through the functions $f_{\text{DIS}}((1, 2)_{\text{BELOW}})$, $f_{\text{DIS}}((1, 3)_{\text{BELOW}})$, $f_{\text{DIS}}((2, -)_{\text{BELOW}})$, and $f_{\text{DIS}}((3, -)_{\text{BELOW}})$. The function $f_{\text{DIS}}((1, -)_{\text{BELOW}})$ expresses the fact that the distance between the newly inserted wedge and the board is the same as the height of either column.

So far, we have not considered the programming of an attributed graph grammar. The approach proposed in the following was originally introduced in [18] and is equivalent to the method proposed in [17] with respect to its descriptive power.

Definition 5: Let P be a finite set of productions. *A control diagram* over P is an u-graph with the set $P \cup \{I, F\}$ as node labels and the set $\{Y, N\}$ as edge labels. Furthermore, the following conditions hold true.

1) There exists exactly one initial node n_I labeled with I.

2) There exists exactly one final node n_F labeled with F.

3) There exists no edge terminating in n_I.

4) There exists no edge originating in n_F.

An example of a control diagram is shown in Fig. 5. With exception of the initial and the final node, all nodes are labeled with productions. Applying productions according to the control diagram, we start with a production which is the label of a direct successor of the initial node and try its application. After successful application of a production, a Y-edges (*y*es) in the control diagram is tracked, while the tracking of a N-edge (*n*o) is caused by the failure of a production. A derivation sequence is stopped when the final node is reached.

Example 3: Since a control diagram can be used for string grammars as well as for graph grammars, we demonstrate the programming by means of the following productions:

1) $S \to A$,

2) $A \to a$,

3) $A \to BB$,

4) $B \to A$.

Starting with the symbol S and applying the productions according to the control diagram in Fig. 5, we obtain the set of all strings containing 2^n times the symbol a, $n \geqslant 0$. (Remark that this language is context sensitive.)

Definition 6: An attributed programmed graph grammar is a 7-tuple $G = (V, W, A, B, P, S, C)$ where

1) V and W are alphabets for labeling the nodes and edged, respectively;

2) A and B are finite sets of attributes for nodes and edges, respectively;

3) P is a finite set of productions;

4) S is a set of initial graphs;

5) C is a control diagram over P.

The above definition parallels the definitions of formal grammars found in the literature up to the point that there is no distinction between terminal and nonterminal labels for nodes and neither for edges. Such a distinction is useful for nonprogrammed grammars, primarily in order to implicitly control the stop of a derivation sequence. For programmed graph grammars, however, this distinction is no longer needed since we have the control diagram—an explicit tool for controlling the order of productions including the stop of a derivation sequence.

Definition 7: Let G be an attributed programmed graph grammar. The *language* of G consists of all a-graphs which can be derived in the following way.

1) Start with an initial graph.

2) Apply productions in an order defined by the control diagram.

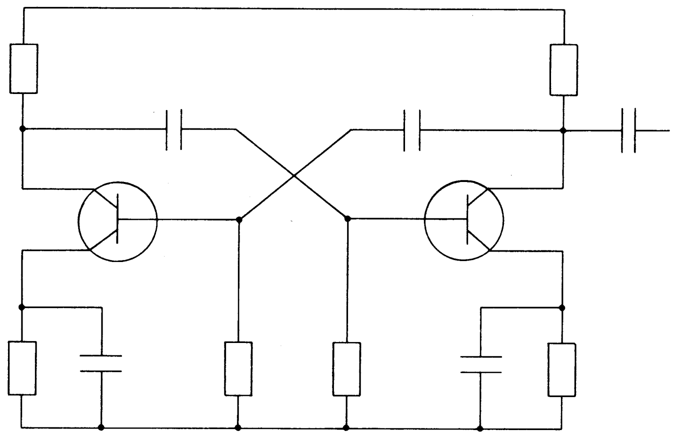


Fig. 6. Example of a circuit diagram.

3) Stop the derivation sequence when the final node in the control diagram has been reached.

Several particular situations, which may arise during a derivation sequence, are worth mentioning. First, there can be the lack of an outgoing Y-edge although the production belonging to the actual control diagram node has been successfully applied. In this case, the continuation of the current derivation sequence is not defined and no graph of $L(G)$ will be generated. The analogous situation arises when the considered production is not applicable and no outgoing N-edge exists. If there exists more than one Y-edge (N-edge) leaving the same control diagram node, any one may be chosen (cf. Example 3).

III. Application to the Interpretation of Line Drawings

Two classes of line drawings with similar characteristics are studied in this chapter, namely circuit diagrams and flowcharts. Sections III-A through III-D are concerned with circuit diagram interpretation while flowcharts are treated in Section III-E. Some experimental results are presented in Section III-F.

A. Basic Principles of Circuit Diagrams and Related Work

Circuit diagrams are line drawings which play an important role in electrical engineering. They are characterized by symbols which stand for single components of a circuit and by connections representing conductions. For each symbol there exist several terminals defining those points where symbols can be connected among each other.

There is an interest in storing descriptions of circuit diagrams in a computer for several reasons, e.g., simulation or automatic layout generation. The task considered in this paper is the automatic interpretation of circuit diagrams, i.e., the automatic extraction of a description from an input diagram. In our approach, the description is given in terms of symbols and connections.

Several approaches to the interpretation of circuit diagrams have been proposed in the literature. They differ mainly in how the prior knowledge about symbols is represented. Procedural representation [24], [25], string grammars [26], decision trees [27], and relaxation [28] has been proposed.

This paper is not concerned with preprocessing and the extraction of lines from circuit diagrams. Therefore, it is assumed that a circuit diagram is given in terms of lines. A contribution to the problem of line extraction directly related with the interpretational approach proposed in this paper can be found in [29]. It is not assumed in the following that the line extraction procedure yields perfect result. Instead, several error types have been assumed to occur. They will be discussed in Section III-D.

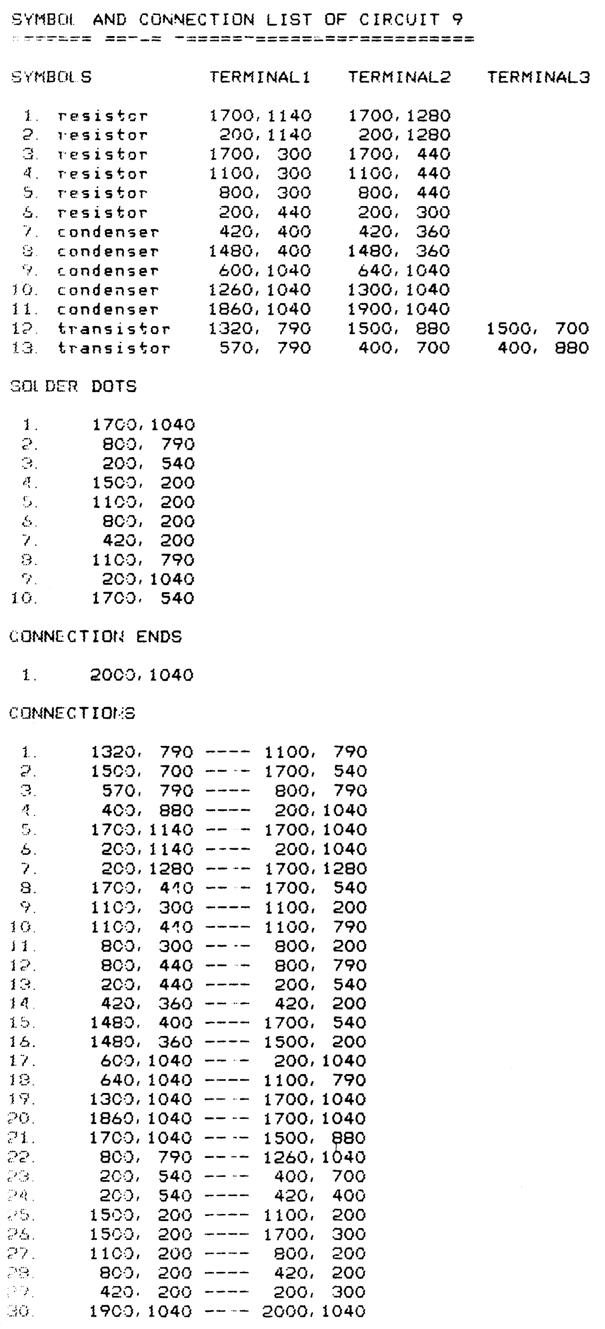
B. Representation of Circuit Diagrams

An example of a circuit diagram is shown in Fig. 6. The interpretation result of this diagram is given in Fig. 7. Symbols are represented by the coordinates of their terminals referring to the coordinate system used for the extraction of lines. Besides symbols and connections, also solder dots and connection ends are reported in Fig. 7.

Obviously, an input diagram like in Fig. 6 can be represented by a graph in several straightforward ways. A possibility for representing a resistor is shown in Fig. 8. The nodes in Fig. 8 correspond to vertices in Fig. 6. A vertex is a location where either several line segments meet each other or a line segment ends. A solder dot may be present at a vertex, too. The number of line segments coinciding in a vertex is used as node label. This means that a node label gives the degree of a node. The edges in Fig. 8 correspond to line segments in Fig. 6. No edge labels are used. We will refer to a graph like in Fig. 8 as input graph. It will be seen later that a representation like in Fig. 8 is insensitive to some types of errors which may occur in the line extraction stage.

Not only an input diagram, but also an interpretation result can be represented by a graph in a straightforward way by representing symbols by nodes and connections by edges. We will refer to a graph representing an interpretation result as output graph.

Both input and output graphs are augmented by attributes. For the case of an input graph, an attribute giving the location of the corresponding vertex is attached to each node. Similarly, attributes giving the location of terminals of symbols will be used in output graphs.



```
SYMBOL AND CONNECTION LIST OF CIRCUIT 9
====== === ==========================

SYMBOLS              TERMINAL1    TERMINAL2    TERMINAL3

 1. resistor         1700,1140    1700,1280
 2. resistor          200,1140     200,1280
 3. resistor         1700, 300    1700, 440
 4. resistor         1100, 300    1100, 440
 5. resistor          800, 300     800, 440
 6. resistor          200, 440     200, 300
 7. condenser         420, 400     420, 360
 8. condenser        1480, 400    1480, 360
 9. condenser         600,1040     640,1040
10. condenser        1260,1040    1300,1040
11. condenser        1860,1040    1900,1040
12. transistor       1320, 790    1500, 880    1500, 700
13. transistor        570, 790     400, 700     400, 880

SOLDER DOTS

 1.      1700,1040
 2.       800, 790
 3.       200, 540
 4.      1500, 200
 5.      1100, 200
 6.       800, 200
 7.       420, 200
 8.      1100, 790
 9.       200,1040
10.      1700, 540

CONNECTION ENDS

 1.      2000,1040

CONNECTIONS

 1.      1320, 790 ---- 1100, 790
 2.      1500, 700 ---- 1700, 540
 3.       570, 790 ----  800, 790
 4.       400, 880 ----  200,1040
 5.      1700,1140 ---- 1700,1040
 6.       200,1140 ----  200,1040
 7.       200,1280 ---- 1700,1280
 8.      1700, 440 ---- 1700, 540
 9.      1100, 300 ---- 1100, 200
10.      1100, 440 ---- 1100, 790
11.       800, 300 ----  800, 200
12.       800, 440 ----  800, 790
13.       200, 440 ----  200, 540
14.       420, 360 ----  420, 200
15.      1480, 400 ---- 1700, 540
16.      1480, 360 ---- 1500, 200
17.       600,1040 ----  200,1040
18.       640,1040 ---- 1100, 790
19.      1300,1040 ---- 1700,1040
20.      1860,1040 ---- 1700,1040
21.      1700,1040 ---- 1500, 880
22.       800, 790 ---- 1260,1040
23.       200, 540 ----  400, 700
24.       200, 540 ----  420, 400
25.      1500, 200 ---- 1100, 200
26.      1500, 200 ---- 1700, 300
27.      1100, 200 ----  800, 200
28.       800, 200 ----  420, 200
29.       420. 200 ----  200, 300
30.      1900,1040 ---- 2000,1040
```

Fig. 7. Result of the interpretation of Fig. 6.

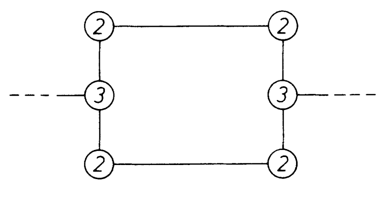


Fig. 8. Graph representation of a resistor.

C. *Transformation of Input Graphs into Output Graphs*

It follows immediately from the last section that the interpretation task, i.e., the extraction of a description from a circuit diagram, can be considered as being equivalent to the task of transforming an input graph, which represents a circuit diagram as shown in Fig. 8, into an output graph, which is a description of the input diagram. An attributed programmed graph grammar according to the definitions in Chapter 2 is used for this transformation.

Since there are some differences between the method proposed here and most of the approaches to syntactic pattern recognition known from the literature, we focus our attention on these differences first before we consider the grammar in more detail. A block diagram of a classical syntactic pattern recognition system is shown in Fig. 9, where it does not matter whether the grammar operates on strings, trees, or graphs. An overview of our system is given in Fig. 10. The drawing to be interpreted is represented by an input graph after a number of preprocessing and primitive extraction steps. Until this time, the method is identical to that in Fig. 9. Instead of parsing, however, the input graph is now transformed into its

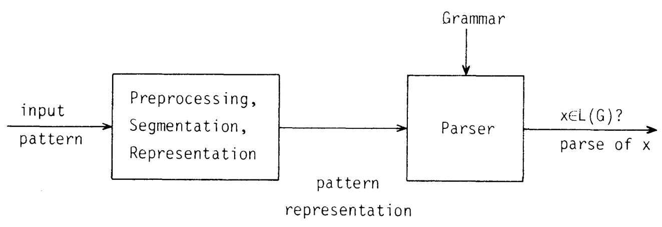


Fig. 9. Block diagram of a conventional syntactic pattern recognition system.

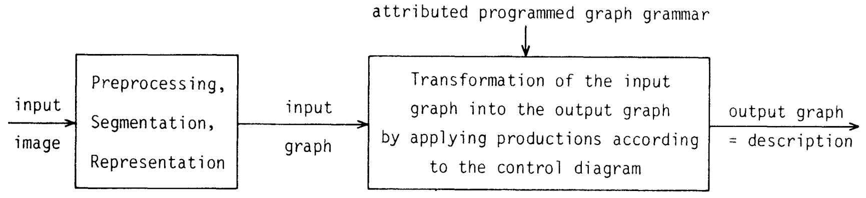

attributed programmed graph grammar
input image
Preprocessing, Segmentation, Representation
input graph
Transformation of the input graph into the output graph by applying productions according to the control diagram
output graph = description

Fig. 10. Block diagram of the proposed system.

corresponding output graph by applying a programmed sequence of productions. This means that we use an attributed programmed graph grammar as a generative tool for directly transforming an input graph into the intended output description. Thus the parsing of graph grammars, which is known to be computationally costly, can be avoided in favor of the generative mechanism of a programmed graph grammar, which can be rather efficiently implemented. In contrast with Fig. 9 where the parse plays the role of a hierarchical description of the input pattern, the output graph in Fig. 10 is a one-level description. Notice that the case $x \notin L(G)$ in Fig. 9 has a counterpart in Fig. 10. It is the situation where there exists no production sequence which leads, starting with an input graph g, to a final node in the control diagram. In this case, g has no corresponding output graph. In other words, the input drawing is rejected.

In order to illustrate the grammar, we consider an example of a production which transforms the input representation of a resistor into the corresponding output representation. The left-hand and right-hand side of this production is shown in Fig. 11. We write node labels inside a node, while denotations for nodes are written outside a node. The left-hand side shows the input graph of a resistor with nodes 3 and 4 corresponding to the terminals. The right-hand side shows the output representation. Node 8 is for representing the symbol type (R = resistor) while nodes 7 and 9 stand for the terminals. The embedding transformation of this production is to be specified as $L = R = \{(3, 7), (4, 9)\}$ which means that the terminals in the output graph are to be connected with the same nodes as the terminals in the input graph. The applicability predicate is to be defined in such a way that parts of an input graph which are structural identical with the left-hand side of our production but which do not correspond to a resistor because of a different geometrical shape will be not interpreted as resistors. A possible applicability predicate for our example is π = parallel ((1, 2); (5, 6)) which states the two long lines in the resistor to be parallel up to a certain threshold. Using this predicate, a symbol like that in Fig. 12, which has the same input graph as a resistor with respect to the syntactic component, will be excluded from being transformed by

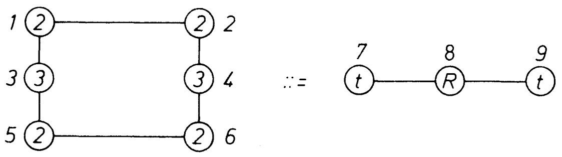


Fig. 11. Left-hand and right-hand side for transforming the input graph of a resistor into the corresponding output graph.

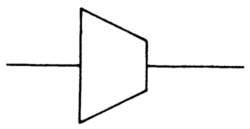
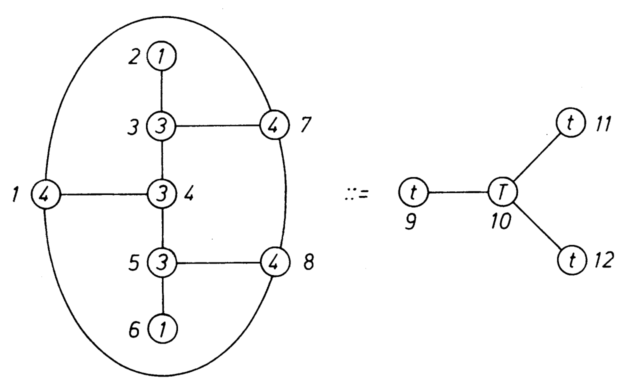


Fig. 12. A symbol the input graph of which has the same syntactic component as a resistor.

means of the production in Fig. 11. The set F of attribute transfer functions is for the definition of the predicates in the output graph. Since we are interested in the location of terminals of a symbol, we state $f_x(7) = x(3)$, $f_y(7) = y(3)$, $f_x(9) = x(4)$, and $f_y(9) = y(4)$. This means that the location of the terminals in the input graph is inherited to those nodes which represent the terminals in the output graph.

Productions for transforming the input representations of other symbols and connections into corresponding output representations can be defined in the same way as discussed above. One more example for transforming a transistor is shown in Fig. 13. The control diagram structure for both Figs. 11 and 13 is given in Fig. 14.

The programming of productions is a very convenient tool. If some structure in an input graph is syntactically or semantically ambiguous in a local context, programming can be used to resolve the ambiguity. For example, a rectangular corner can be a part of a resistor as well as a part of a connection. Having applied a production p like that in Fig. 11 according to a control diagram order shown in Fig. 14 as many times as possible, it is clear that the remaining corners cannot belong to a resistor since all resistors have been replaced. Thus, local ambiguities can be resolved through programming. In our system, the productions and the control diagram has been



```
Π : TRUE
T : L = R = {(1,9), (7,11), (8,12)}
F : f_x(9) = x(1); f_y(9) = y(1);
    f_x(11)= x(7); f_y(11)= y(7);
    f_x(12)= x(8); f_y(12)= y(8).
```

Fig. 13. A production for transforming a transistor.

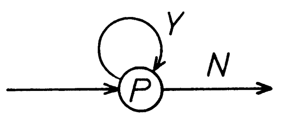


Fig. 14. A control diagram substructure enforcing production p to be applied as many times as possible.

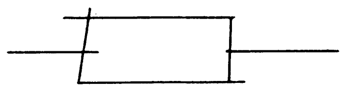
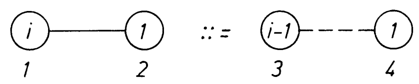
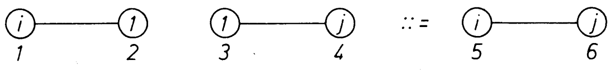


Fig. 15. A resistor distorted by substitutions and insertions.



```
Π : TRUE
T : L=R= {(1,3)}
F : f_x(3)=x(1); f_y(3)=y(1);
    f_x(4)=x(2); f_y(4)=y(2).
```

Fig. 16. A production for cleaning an input graph from insertions.



```
Π : distance (2,3) ≤ θ ∧ collin (1,2,3,4)
T : L=R= {1,5), (4,6)}
F : f_x(5)=x(1); f_y(5)=y(1);
    f_x(6)=x(4); f_y(6)=y(4).
```

Fig. 17. A production for closing a gap within a line.

inferred heuristically based on the shape of symbols occurring in the diagrams. Since the overall algorithm for diagram interpretation is expressed through the productions and the control diagram, the inference of the control diagram is dependent on the productions and vice versa.

D. Error Correction

There are three prominent error types in structural and syntactic pattern recognition, namely substitution (misclassification), deletion, and insertion of primitive pattern components [5]. These errors have been assumed to occur in the following instances in our system: 1) misclassification of line segments, 2) insertion of line segments, 3) deletion of line segments, 4) gaps within line segments. Following are correction means for these errors.

1) *Misclassification of Line Segments:* Misclassifying a straight horizontal line segment as a straight sloping line segment, for example, does not cause any confusion in our system since the type of a line segment is not used anywhere, neither in productions nor in an input or output graph. Therefore, no further correction means for errors of this type are needed.

2) *Insertion of Line Segments:* This type of error has been restricted to insertions where one end of the inserted line segment touches an already existing vertex while the other end is free. In this way, the input graph is only locally corrupted. For an example, see Fig. 15.

We can correct this type of error by a kind of preprocessing relabeling all free ending line segments by means of a production as shown in Fig. 16. Using the control diagram structure shown in Fig. 14, one can enforce the application of the production in Fig. 16 as many times as possible. When this production is no longer applicable, it is possible to apply the productions constructed for undistorted symbols and connections regardless of the number of insertions. However, applying productions like that in Fig. 16 to an input graph can result in relabeling certain line segments which are "legal" constituents rather than being due to insertions. Therefore, one has to take into account the effect of productions like that in Fig. 16 when the productions for the undistorted case are constructed.

3) *Deletion of Line Segments:* Similar to insertions, also deletions have been restricted to those cases where one end of the inserted line segment ends free. As a consequence, such an error has the same effect on an input graph as the preprocessing production shown in Fig. 16. Therefore, no further correction means are needed.

4) *Gaps Within Line Segments:* Productions as shown in Fig. 17 can be used for closing gaps. The parameter θ in the applicability predicate controls the maximum width of a gap while the condition collin (1, 2, 3, 4) ensures the two line segments to be connected lying approximately on the same line.

By means of the control diagram, the application order of the productions can be programmed such that first all gaps within lines are closed (cf. Fig. 17). Next, productions as shown in Fig. 16 are applied as many times as possible. Afterwards, the symbols and connections can be transformed by means of productions like in Fig. 11.

In contrast with error correcting parsing [5], not all possible errors can be corrected by the correction means provided above. However, the errors covered seem to be a good compromise from a practical point of view. If one has to take into account more errors, the grammar is to be augmented by additional productions.

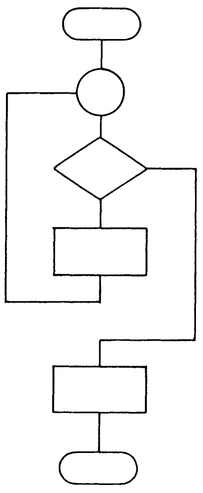


Fig. 18. An example of a flowchart.

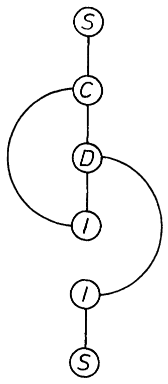


Fig. 19. The output graph corresponding to Fig. 18.

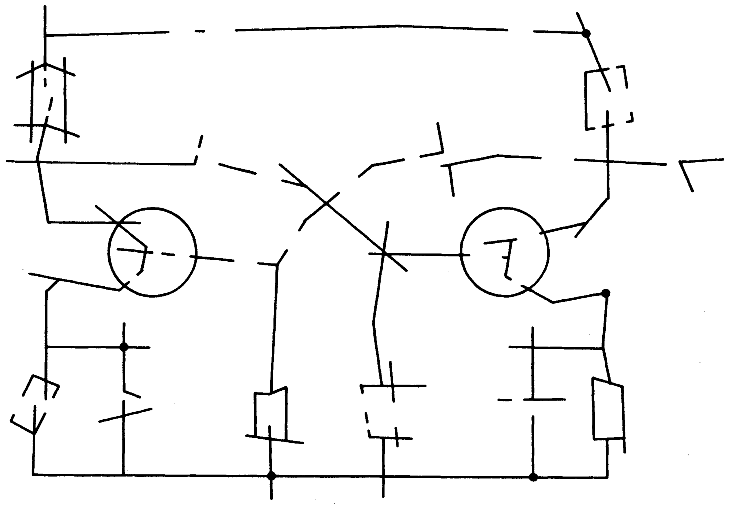


Fig. 20. A distorted version of the circuit diagram in Fig. 6.

E. Interpretation of Flowcharts

Flowcharts have many characteristics in common with circuit diagrams. The principal constituents of flowcharts are symbols and connections. Symbols represent instructions, decisions, etc., while connections indicate the flow of control in a program. There are many interesting applications for automatic flowchart interpretation. For instance, combining line interpretation with text recognition can result in a system which automatically generates a program from a given input flowchart. Furthermore, a description obtained by the system described in this paper can be used as input to another system which checks the well-structuredness of the program. Similarly, such a description can be used in the context of program verification. An approach to the automatic interpretation of flowcharts has been proposed in [30].

The representation of a flowchart by an input graph is done in the same way as for circuit diagrams. The same types of errors as in circuit diagrams have been assumed to occur. Consequently, the same means for error correction as described in Section III-D have been provided.

An example of a flowchart is shown in Fig. 18. Its interpretation result, the output graph, is given in Fig. 19 where the node labels have the following meaning: S = start/stop, I = instruction, D = decision, C = connector.

F. Experimental Results

The proposed method has been implemented in Fortran on a PDP 11/45. Two attributed programmed graph grammars, one for circuit diagrams and one for flowcharts, have been manually constructed. Both grammars comprise error correcting capabilities as discussed in Section III-D. The grammar for circuit diagrams contains about 60 productions, while about 30 productions are sufficient for flowchart interpretation. The input graphs have been manually extracted from a series of schematics. Various distortions have been simulated in order to gain experience with the error correction capabilities. So far, the system has shown satisfactory performance. Having once become familiar with attributed programmed graph grammars, it has turned out that this method allows the representation of both syntactic and semantic *a priori* knowledge about line drawings on a high and user oriented level. According to the experience gained so far, also the debugging of a grammar can be performed in a fairly simple way.

About 30 diagrams have been used for testing the proposed method. One example of an undistorted circuit diagram is shown in Fig. 6. A distorted version of the same diagram is shown in Fig. 20. The interpretation result of Fig. 6 is given in Fig. 7. All errors in Fig. 20 can be properly corrected and the interpretation result of Fig. 20 is equivalent to that in Fig. 7. The programs run on 28K and computing time for interpreting a diagram like that in Fig. 6 is typically less than 1 s depending on the size of a particular diagram and the degree of distortion.

IV. Concluding Remarks

Attributed programmed graph grammars have been introduced as a tool for expressing both structural and semantic *a priori* knowledge. A high and user oriented level of knowledge representation is achieved by using graphs rather than one-dimensional strings as basic structures. Augmenting graphs by attributes results in a flexible model where both concepts based on structural principles and concepts based on numerical principles can be integrated. The programming of a grammar supports modularity of knowledge representation. Each rule can be expressed through a production while the overall control algorithm for applying these rules is formulated independently. In this way, the perspicuity of the grammar can be improved. Furthermore, the programming of the grammar supports the use of productions for correcting structural errors in an input pattern. Semantic variations and distortions can be handled by means of appropriate applicability predicates in productions. Using a graph grammar as generative tool, as proposed in this paper, rather than controlling a parser through a grammar, can result in an efficient implementation as the experiments have shown.

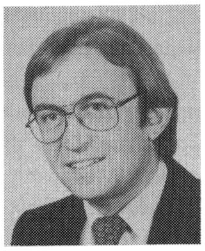
The applications in this paper have been limited to two simple classes of line drawings. However, it is obvious that attributed programmed graph grammar can be applied to other problems, too. Providing suitable preprocessing methods for extracting input graphs, the proposed grammar model can also be used for the interpretation of gray level images.

Acknowledgment

The author would like to thank Prof. K. S. Fu of Purdue University for many stimulating ideas and helpful suggestions.

References

[1] K. S. Fu, *Syntactic Methods in Pattern Recognition.* New York: Academic, 1974.
[2] R. C. Gonzales and M. G. Thomason, *Syntactic Pattern Recognition.* Reading, MA: Addison-Wesley, 1978.
[3] T. Pavlidis, *Structural Pattern Recognition.* Berlin: Springer-Verlag, 1977.
[4] K. S. Fu, Ed., *Syntactic Pattern Recognition: Applications.* Berlin: Springer-Verlag, 1977.
[5] K. S. Fu, *Syntactic Pattern Recognition and Applications.* Englewood, Cliffs, NJ: Prentice-Hall, 1981.
[6] A. C. Shaw, "Parsing of graph-representable pictures," *J. Ass. Comput. Mach.*, vol. 17, no. 3, pp. 453–581, 1970.
[7] T. Feder, "Plex languages," *Inform. Sci.*, vol. 3, pp. 225–241, 1971.
[8] K. S. Fu and B. K. Bhargava, "Tree systems for syntactic pattern recognition," *IEEE Trans. Comput.*, vol. C-22, pp. 1087–1099, 1973.
[9] J. M. Brayer, P. H. Swain, and K. S. Fu, "Modeling of earth resources satellite data," in *Syntactic Pattern Recognition: Applications*, K. S. Fu, Ed. Berlin: Springer-Verlag, 1977, pp. 215–242.
[10] J. M. Brayer and K. S. Fu, "Web grammars and their application to pattern recognition," Purdue Univ., Rep. TR-EE 75-1, Dec. 1975.
[11] J. L. Pfaltz, "Web grammars and picture description," *Comput. Graphics Image Processing*, vol. 1, pp. 193–210, 1972.
[12] J. M. Brayer, "Parsing of web grammars," in *Proc. IEEE Workshop Data Description and Management*, Long Beach, CA, 1977.
[13] J. L. Pfaltz and A. Rosenfeld, "Web grammars," in *Proc. 1st Int. Joint Conf. Artificial Intell.*, Washington, 1969, pp. 609–619.
[14] U. G. Montanari, "Separable graphs, planar graphs and web grammars," *Inform. Contr.*, vol. 16, pp. 243–267, 1970.
[15] T. Pavlidis, "Linear and context-free graph-grammars," *J. Ass. Comput. Mach.*, vol. 19, pp. 11–23, 1972.
[16] M. Nagl, "A tutorial and bibliographical survey on graph grammars," in *Lecture Notes in Computer Science*, vol. 73, 1979, pp. 70–126.
[17] D. J. Rosenkrantz, "Programmed grammars and classes of formal languages," *J. Ass. Comput. Mach.*, vol. 16, pp. 107–131, 1969.
[18] H. Bunke, "Programmed graph grammars," in *Lecture Notes in Computer Science*, vol. 56, 1977, pp. 155–166.
[19] D. E. Knuth, "Semantics of context-free languages," *Math. Syst. Theory*, vol. 2, pp. 127–146, 1968.
[20] G. V. Tang and T. S. Huang, "A syntactic-semantic approach to image understanding and creation," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-1, pp. 135–144, Apr. 1979.
[21] W. H. Tsai and K. S. Fu, "Attributed grammar–A tool for combining syntactic and statistical approaches to pattern recognition," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-10, pp. 873–885, Dec. 1980.
[22] J. K. Cheng and T. S. Huang, "Algorithms for matching relational structures and their applications to image processing, Purdue Univ., Rep. TR-EE 80-53, 1980.
[23] W. H. Tsai and K. S. Fu, "Error-correcting isomorphisms of attributed relational graphs for pattern analysis," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-9, pp. 757–768, Dec. 1979.
[24] R. L. T. Cederberg, "Automatic interpretation of hand-drawn electrical schematics," presented at the Int. Conf. CAD and Manufacture of Electron. Components, Circuits, and Syst., Univ. Sussex, 1979.
[25] S. Kakumoto, Y. Fujimoto, and J. Kawasaki, "Logic diagram recognition by divide and synthesize method," in *Artificial Intelligence in PR and CAD*, J. Latombe, Ed. North-Holland, 1978, pp. 457–477.
[26] B. Zavidovique and G. Stammon, "An automated process for electronics scheme analysis," in *Proc. 5th ICRP*, Miami, FL, 1980, pp. 248–250.
[27] H. Bunke, "Automatic interpretation of lines and text in circuit diagrams," in *Proc. NATO Adv. Study Inst. Pattern Recognition, Theory and Applications*, Oxford, England, Mar./Apr. 1981, pp. 297–310.
[28] H. Bunke and G. Allermann, "Probabilistic relaxation for the interpretation of electrical schematics," *Proc. IEEE Conf. Pattern Recognition and Image Processing*, Dallas, TX, Aug. 1981, pp. 438–440.
[29] H. Bley, "Digitization and segmentation of circuit diagrams," in *Proc. 1st Scandinavian Conf. Image Anal.*, Linköping, Sweden, 1980, pp. 264–269.
[30] C. Y. Suen, "Recognition of hand-drawn flowcharts," in *Proc. 3rd Int. Joint Conf. PR*, Coronado, 1976, pp. 424–428.



Horst Bunke was born in Langenzenn, West Germany, on July 30, 1949. He received the M.S. and Ph.D. degrees in computer science from the University of Erlangen, Erlangen, West Germany, in 1974 and 1979, respectively.

Since 1976 he has been a member of the scientific staff at the University of Erlangen. From August 1980 until July 1981 he was visiting the School of Electrical Engineering, Purdue University, West Lafayette, IN. His major interests are in structural and syntactic pattern recognition with a particular emphasis on applications.