Pattern Recognition, Vol. 19, No. 4, pp. 279–288, 1986.
Printed in Great Britain.

0031-3203/86 $3.00+.00
Pergamon Journals Ltd.
Pattern Recognition Society

SHAPE GRAMMAR COMPILERS

THOMAS C. HENDERSON and ASHOK SAMAL
Department of Computer Science, The University of Utah, Salt Lake City, Utah, U.S.A.

(*Received* 28 *October* 1985)

Abstract—Compiler generation tools have been used quite successfully to produce parsers for certain classes of string grammars. Such techniques can also be applied to the development of syntactic shape parsers. We present a generalization of LR parsing to shape grammars based on the use of geometrical relations between the symbols. The components of this approach are:

(1) a grammar for defining classes of 2-D and 3-D shapes,

(2) a shape grammar compiler which produces a tabular representation of the explicit and implicit constraints between the parts of the shape,

(3) and a general parsing mechanism which uses these tables of constraints to perform the analysis of unknown shapes.

Syntactic methods Compiler generation Constraint techniques Shape analysis

1. INTRODUCTION

We believe that the syntactic method offers many advantages for shape analysis. The major advantage is the possibility of defining logical relations between anthropomorphically significant parts of a shape. Moreover, formal techniques allow both the automatic generation of constraint relations for grammatical descriptions of shape, and the application of these constraints during the analysis of shape. Shaw,[1] Fu,[2] Rosenfeld[3] and others[4–7] have proposed various approaches to syntactic or grammatical shape models, but in general, the parsing methods for these models are standard string parsers, e.g. Earley's algorithm. In order to obtain the most advantage from the grammatical approach, however, the relation between the shape grammar and the shape parsing method must be formally established. In this paper, we consider a bottom-up parsing mechanism and its relation to a particular class of shape grammars. Our goal is to outline a framework for a coherent approach to syntactic pattern recognition.

In choosing the parsing mechanism for a given shape grammar, the problem is much the same as that faced by a compiler writer trying to choose a recognizer for a string grammar. In the traditional bottom-up parsing approach,[8, 9] a recognizer is implemented in a general way using tables. These tables are derived from the given grammar and describe relations between the vocabulary symbols of the grammar. A constructor is designed which, given a grammar, checks it for suitability and builds the necessary tables for the recognizer (see Fig. 1). That is, to implement the recognizer for a given grammar, the constructor is run with the grammar as data, and the output is merged with the recognizer. We will show how this technique can be extended from string grammars to shape grammars.

Our general shape grammar scheme is to produce a shape parsing mechanism by means of a shape grammar compiler from a high-level shape grammar description. This is analogous to using an automatic parser generator to produce a string parser from a high-level programming language description. Note that this contrasts with most syntactic methods; they use a general context-free parser.

However, for an approach which is similar in spirit, see Bunke's attributed programmed graph grammar transformation system.[10] The shape parsing mechanism performs the actual analysis of unknown shapes and outputs an organization imposed on the shape primitives in terms of the underlying grammar.

Most proposed syntactic shape analysis methods have dealt with the shape grammar (or model) at length, while the corresponding parsing algorithm has been chosen *ad hoc* and from a string grammar perspective, e.g. You[11] uses Earley's algorithm. The shape parsing mechanism has usually been constructed manually. Finally, in most formalisms it is a tedious process to produce the shape grammar for any interesting class of shapes.

This provides an impetus for developing more

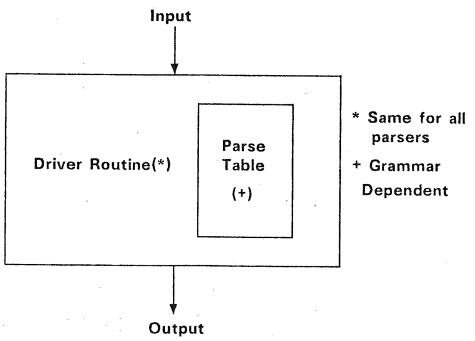

Fig. 1. Parts of an LR parser.

suitable user-oriented languages for describing a shape grammar. Thus, one problem faced is the design of suitable shape grammar description languages, and the subsequent problem is the construction of correct and efficient compilers for such languages.

As an example of one approach to syntactic shape analysis, we propose a generalization of LR parsing as the framework within which to define shape grammars and their parsers. For another example see Mohr and Masini.[12] The major motivation for this choice is that methods exist for automatically deriving the shape parsing mechanism. In particular, we have:

—a shape grammar formalism which accounts for most structural aspects of 2-D and 3-D shape,
—a table-driven parsing mechanism which uses constraints between pieces of the shape, and
—an automatic method to compute constraint relations between the vocabulary symbols of the grammar.

This process can be viewed as a generalization of traditional table-driven grammar techniques in that the grammars involve constraints between string grammar symbols. With string grammars, bottom-up parsing involves scanning from left to right until the tail of the handle is found, then scanning right to left from the tail until the head of the handle is found. This works well enough for string grammars, but shape grammars pose the problem of complicated relations between the symbols, and these relations must be accounted for and taken advantage of by the shape parsing mechanism.

2. DETAILS OF THE APPROACH

Most syntactic approaches which consider the parsing problem at all typically use parsing algorithms which are applicable to the entire class of context-free grammars, e.g. the Cocke–Younger–Kasami algorithm or Earley's algorithm. Although these are general parsing methods, they are computationally expensive even for string grammars, and they require on the order of n^3 time and order n^2 space. On the other hand, it is possible that for specific grammars, these requirements can be reduced. Other methods, such as LR(k) parsing, are available, though, which do not suffer from these deficiencies, and are the recommended technique for implementing parsers for programming languages. It is in this vein that we explore the extension of the LR(k) approach to shape grammar parsing and the automatic generation of parsers from shape grammar descriptions.

2.1. *LR parsing*

The LR(k) grammars are a class of context-free grammars which can be parsed deterministically.[8] The parse uses a left-to-right scan (the L in LR) of the input to produce (the reverse of) a right sentential parse (the R in LR) and can scan k input symbols ahead. Suppose that is a rightmost derivation of w, where rm means rightmost and p_i means production i is applied. Then an LR(k) parser produces $p_m, p_{m-1}, \ldots, p_1$. Basically, the parser shifts input symbols onto a pushdown list. When a handle appears

$$S = \alpha_0 \underset{\text{rm}}{\overset{p_1}{\Rightarrow}} \alpha_1 \underset{\text{rm}}{\overset{p_2}{\Rightarrow}} \ldots \underset{\text{rm}}{\overset{p_m}{\Rightarrow}} \alpha_m = w$$

at the top of the pushdown list, the handle is reduced. This process continues until the start symbol is produced or an error is detected. For LR(k) grammars, it is possible to determine which nonterminal should replace the handle by scanning at most k symbols to the right of the handle.

Thus, an LR(k) parser must decide whether to shift a new input symbol onto the pushdown list or whether a handle is present already. Once it is determined that a handle is present, it is necessary to find the left end of the handle. Finally, the appropriate nonterminal to replace the handle must be chosen (this involves finding the right production to apply).

An LR(k) parser is a table which encodes the current state of the parse in terms of (1) a *parsing action function* and (2) a *goto function.* A parse proceeds by applying the parsing action to the head of the pushdown list. The four possible actions are: **shift, reduce, error** or **accept.** If **shift,** then the next input symbol is shifted onto the pushdown list. If **reduce,** then the indicated production is used to remove the symbols on the top of the pushdown list which correspond to the right hand side of the production. The left hand side of the production is placed on the pushdown list. If **error,** the parse is halted with an error message, or some kind of recovery may be attempted. If **accept,** the parse is halted and the parse is known.

One advantage of LR(k) parsing is that it is possible to optimize LR(k) parsers. LR(k) grammars represent one of the largest classes (of unambiguous grammars)

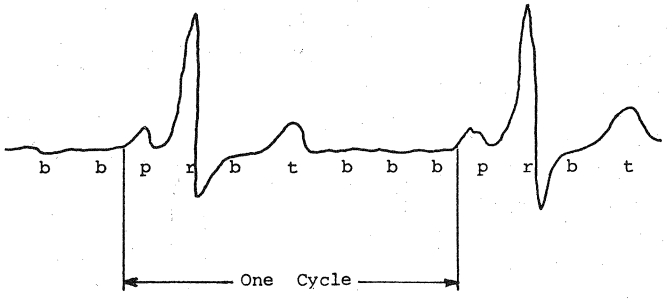

Fig. 2. The regular pattern appearing in a normal ECG.

for which deterministic parsers can be constructed, and LR parsers can compete quite well with other kinds of parsers. The basic problem for producing an LR(k) parser is to generate the LR(k) table which controls the parsing. Although it is possible to automatically produce these tables, they are often too large to be practicable. However, since LR(k) parsers have the desirable properties of being fast and capable of early error-detection, the return is high if an optimizing step is taken.

2.2. *Extending LR techniques for shape grammars*

Now let's consider how this approach applies to shape grammars. First of all, it is possible to encode a class of shapes directly as an LR(k) grammar using terminal symbols which correspond to some shape primitives. Such parsers can be produced using standard parser generators such as *lex* (lexical analyzer generator) and *yacc* (yet another compiler compiler). For the examples given below, we have used these tools running under Berkeley Unix 4.2. Thus, for any class of shapes which can be described by LR(0) grammars, it is quite straightforward to generate the corresponding recognizers using such parser generator tools.

First, parsers can be produced rather easily for any class describable by regular expressions. For example, normal ECGs (see Fig. 2) can be characterized by the following regular expression (from Gonzalez[5]):

$$[prbt\,(b|bb|bbb)]+.$$

The input to *lex* is given in Appendix A. This is all that is required to produce the finite state recognizer.

Context-free grammars require the use of *yacc*. Consider the following two examples from Fu,[4] one for median chromosomes and the other for acrocentric chromosomes. These shapes are shown in Figs 3 and 4.

Grammar for median chromosome

$$G_1 = (T, N, P, S)$$

$$T = \{a, b, c, d\}$$

$$N = \{S, A, B, D, E, F, H, J\}$$

where a, b, c and d correspond to

$\frown$, | , $\backsim$, (, respectively.

Productions (P):

$$S \to A\ A$$

$$A \to c\ B$$

$$B \to F\ B\ E|H\ D\ J$$

$$D \to F\ D\ E|d$$

$$E \to b$$

$$F \to b$$

$$H \to a$$

$$J \to a.$$

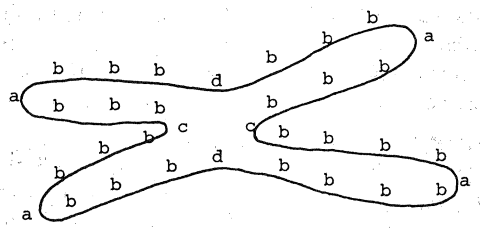

Fig. 3. Median chromosome shape.

The LR parsing table produced for this grammar is given in Appendix B.

Grammar for acrocentric chromosome

$$G_2 = (T, N, P, S)$$

$$T = \{a, b, c, d\}$$

$$N = \{S, A, B, D, E, F, G, H, J, L, R, W\}.$$

Productions (P):

$$S \to A\ A$$

$$A \to c\ B$$

$$B \to F\ L|R\ E$$

$$D \to F\ G|W\ E$$

$$E \to b$$

$$F \to b$$

$$G \to F\ G|d$$

$$H \to a$$

$$J \to a$$

$$L \to H\ D\ J|F\ L$$

$$R \to H\ D\ J|R\ E$$

$$W \to W\ E|d.$$

The LR parsing table for this grammar is given in Appendix C.

However, for more complex geometrical shape grammars, the extension to LR(k) parsing must be more general. It is important to note that LR(k) tables form the basis for LR(k) parsing. Moreover, LR(k) tables encode the structure of the strings of a language in terms of their neighborhood relations; e.g. PRECEDES and FOLLOWS. That is, the LR(k) tables represent in a finite and deterministic way the combinatorics of these relations. This suggests that one generalization of LR grammars to shape gram-

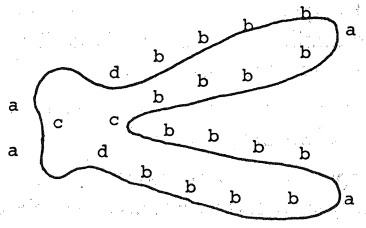

Fig. 4. Acrocentric chromosome shape.

mars can be based on the use of relations between symbols, where these relations include geometrical relations between the shape primitives. Several issues must be dealt with:

(1) the definition and use of the parsing tables,
(2) the generalization of "left-to-right" scanning,
(3) the nature of the parse which is produced, i.e. the control of the parse, and
(4) the noise and ambiguity in the data and shape primitives.

The goal of this paper is to define the general advantages and disadvantages of this approach and to indicate how some of the previous work in syntactic pattern recognition addresses these issues. It is clear that many possibilities exist for tackling these problems.

The automatic generation of parsing tables is at the heart of the approach. It is possible to define table-driven shape parsing methods, where the tables encode various geometric relations of interest between symbols in a convenient way. Even though a deterministic parsing method does not necessarily result, it is possible to apply intrinsic shape constraints during a parse. This approach can exploit the mechanism of table-driven methods in two fundamental ways.

(1) First, it is possible to use the tables directly defined in a similar way as string grammar tables. E.g. we have already seen that shapes can be defined as strings. Typically this approach is possible if a complete, finite, unambiguous set of shape primitives is available. Moreover, simple extensions to more general shapes are possible (and the left-to-right scan can be appropriately modified). In general, though, this direct approach runs into too many problems with missing or ambiguous shape primitives, noise, etc.

(2) Alternatively, the tables (which define the shape relations) can be used to drive a parsing process which is based on using the error information in the tables. In fact, the hierarchical constraint process (HCP) implements just this notion. The process works as follows:

—shape primitives are assigned a set of possible terminal symbol interpretations,

—these multiple labelings give rise to a host of possible shapes depending on which combination of labels is chosen,

—the failure of a constraint on a given vocabulary symbol hypothesis for a particular shape primitive causes that hypothesis to be removed.

The major idea is that the tables encode general 2-D and 3-D shape relations and can be used by a shape parser to eliminate invalid hypotheses. Thus, the parse proceeds in two modes:

—a "disprover" mode in which inconsistent hypotheses are discarded, and

—an "apply productions" mode in which higher-level vocabulary symbols are produced from the current set of symbols.

The table techniques can be used in both of these modes. The tables used in this process can be automatically produced from the shape grammar which defines the class of shapes to be recognized. (Methods for producing tables of the second mode can be found in Aho and Ullman,[8] and table generation methods for the "disprover" mode can be found in Henderson.[13])

In general, it is not possible to talk about or define a "left-to-right-scan" for a set of shape primitives, unless the shapes are described as silhouettes. Even in this case, it is not easy to define a unique starting primitive. In our previous work on HCP, we have actually defined and used grammars for silhouettes. However, in order to take noise and ambiguity into account (also see below), no left-to-right scan was defined or used. Thus, in the most general case, the vocabulary symbols are considered in parallel. (Note that, for a shape grammar, a vocabulary symbol corresponds to a part of the shape, where terminal symbols define the lowest level granularity of shape description, and the start symbol represents a complete shape.) Also, no uniquely appropriate sentential parse has been defined.

Finally, the problem of noise and ambiguity must be addressed in applying syntactic techniques to shape analysis, especially if real world images are to be analyzed. Several solutions have been proposed for this problem, including:

—relax the conditions for extracting the shape primitives,

—introduce probabilities on the productions and strings which describe the shapes, and

—allow multiple hypotheses for the shape primitives and delete inconsistent or unsupported hypotheses.

All of these approaches can be used within the shape grammar compiler paradigm proposed here. Our work with HCP has concentrated on the last of these approaches. The choice of one of these methods should be made on the basis of the kind of data that will be available, the class of shapes to be recognized, and the desired robustness.

Other important issues, such as the control of the parse, the choice of constraints, etc. can be discussed once a particular method is chosen. For example, if the shapes can be modeled by an LR string grammar, then the parse is a deterministic LR parse, and the constraints are determined by the handles of the language. On the other hand, HCP can use a wide range of constraints and controls the parse through hypothesis elimination.

3. A 3-D SHAPE GRAMMAR EXAMPLE

To illustrate these ideas in the context of 3-D shape modeling, we first review the hierarchical constraint process and then give an example. We present a grammar for the "cup" shape [compare, for example, the grammar in Lin[14]], where the shape primitives are certain generalized cylinders.

Stratified shape grammars have been described elsewhere in detail,[13] and we give only a brief summary here. A stratified context-free grammar, **G**, is quadruple (**T, N, P, S**), where **T** is the set of terminal symbols, **N** is the set of non-terminal symbols, **P** is the set of productions, and **S** is the start symbol. Let $\mathbf{V} = (\mathbf{N} \cup \mathbf{T})$ be the set of all vocabulary symbols. Associated with every symbol $v \in \mathbf{V}$ is a level number, $ln(v)$: $\mathbf{V} \rightarrow \{0, 1, \ldots, n\}$, where $ln(\mathbf{S}) = n$, and $\forall\, v \in \mathbf{T}$, $ln(v) = 0$.

T consists of symbols which correspond to relatively large pieces of the shapes modeled by the grammar. In particular, each terminal symbol corresponds to one of the following: a circle, a cylinder, or a curve segment.

N consists of symbols each of which has a level number from 1 to **n** associated with it. In any rule $v := \alpha$ (the rewrite part of a production), if $ln(v) = k$, then every symbol in the string a is at level $k - 1$. Furthermore, $\forall\, v \in \mathbf{V}$

v = ⟨name part⟩ {attachment part} [semantic part], where

⟨name part⟩ is a unique name by which the symbol v is known,

{attachment part} is a set of attachment points of the symbol,

[semantic part] is a set of predicates which describes certain **P** consists of productions of the form ($v := \alpha$, A, C, G, G), and $v := \alpha$

that indicates the replacement of the symbol v by the group of symbols α, where $v \in \mathbf{N}$ and $\alpha = v_1 v_2 \ldots v_k$ such that $v_i \in \mathbf{V}$ and $ln(v_i) = (ln(v) - 1)$ for $i = 1, k$: A is a set of applicability conditions on the syntactic arrangement of the v_i; C is set of semantic consistence conditions on the v_i and consists of various predicates describing geometric and other properties of the v_i; G_a is a set of rules for generating the attachment part for v, the new symbol; G_s is a set of rules for generating the semantic part of v, the new symbol.

What is important to note about such grammars is that it is possible to derive very useful geometric constraints from them, and that such constraints can be used in a table-driven way. In fact, the constraints are nothing but relations implemented as tables.

We will not go through the details of the constraint compilation techniques. From the shape grammar, these techniques produce an enlarged set of relations which are only partially explicit in the grammar. For example, it may be possible to discover that two pieces of a shape (more technically, their syntactic symbols) are indeed parallel, even though no explicit statement of that is given in the productions.

Such relations can be used to significantly reduce the amount of work necessary to determine if an unknown shape is in the class defined by the shape grammar. The mechanism which performs this function is the Hierarchical Constraint Process. HCP uses the hypothesis elimination mode to account for ambiguity and noise in the data. In fact, the ambiguity of the underlying data is a major problem faced by any syntactic shape analysis method, but not by string parsers.

Usually no clear-cut decision can be made in associating the terminal symbols of a grammar with the shape primitives. Thus, the parsing mechanism must not only overcome the problem of parsing a complicated arrangement of symbols (i.e. concatenation is no longer the only relation between symbols), but must also disambiguate the interpretations of a given shape primitive. HCP solves this in the following way. Given a set of shape primitives (i.e. circles, cylinders, or curve segments detected in a scene), and a stratified shape grammar for the class of shapes, HCP performs the following actions:

—associate with each shape primitive a set of possible interpretations, i.e. terminal symbols,

—determine the initial network of hypotheses, that is, for each possible interpretations of each shape primitive, insert a node in the network; two nodes of the network are connected if their underlying shape primitives are physically adjacent,

—apply procedures to the network until the network is empty or the start symbol is produced.

The association of terminal symbols with shape primitives will (in the limit) be to hypothesize every terminal symbol for each primitive. However, methods for reducing the number of hypotheses include using a more global analysis to derive indications of appropriate scale, orientation, etc. from simple global properties; e.g. one can histogram selected features of the primitives themselves to infer properties of particular vocabulary symbols.

The network of hypotheses represents all possible sentential forms for the given shape primitives. Every path in the network represents a distinct set of interpretations of the primitives and must be parsed. However, this is usually much too large a set to be parsed one after the other. The hierarchical constraint process computes a bottom-up parse of all the paths in parallel. This is done by applying the constraints to the network and can be described by specifying three simple procedures and two sets which these procedures manipulate.

BUILD—Given level k of the network, BUILD uses the productions of the grammar to construct level $k + 1$ nodes. Physically adjacent hypotheses are linked, and a record is kept of which nodes are used to construct each level $k + 1$ node. All level $k + 1$ nodes are put into the CONSTRAIN-SET, and all level k nodes are put in the COMPACT-SET (both of these sets are initially empty).

CONSTRAIN—While the CONSTRAIN-SET is not empty, CONSTRAIN examines each member of that set; if a node fails to satisfy the constraints, then its neighbors are put into the CONSTRAIN-SET, any nodes it helped produce and the nodes used to produce it are put into the COMPACT-SET, and it is deleted from the network.

Neighbors:

	bot	cyl	bod	han	cup
bot		1			
cyl	1				
bod				1	
han			1		
cup					

Parallel:

	bot	cyl	bod	han	cup
bot		1	1	1	1
cyl	1		1	1	1
bod	1	1		1	1
han	1	1	1		1
cup	1	1	1	1	

Fig. 5.

COMPACT—While the COMPACT-SET is not empty, COMPACT examines each member of that set; given a node n from the COMPACT-SET, if one of the lower level nodes used to produce n has been deleted, or if n has not helped produce a node at the level above it, (and that level has been built), then n's neighbors are put into the CONSTRAIN-SET, any nodes it helped produce and the nodes used to produce n are put into the COMPACT-SET, and n is deleted from the network.

This then is the shape parsing mechanism. The constraint propagation is based on the discrete relaxation techniques developed by Waltz[15] and Rosenfeld.[16] As an example of this approach, consider the following 3-D shape grammar for the class "cup".

T = {bottom, cylinder, handle}

N = {cup, body}

S = {cup}, and

p = {

Production 1:

⟨cup⟩ { } [vertical_axis diameter height]: =
⟨body⟩ {contact_1 contact_2} [vertical_axis' diameter' height']
+ ⟨handle⟩ {end_1' end_2'} [height" vertical_axis"]

A: [($\text{contact}_1 = \text{end}_1$) and ($\text{contact}_2 = \text{end}_2$)
or ($\text{contact}_1 = \text{end}_2$) and ($\text{contact}_2 = \text{end}_1$)]

S: [(vertical_axis' = vertical_axis") and (height' = height")]

G_a: [ ]

G_s: [(vertical_axis: = vertical_axis') (diameter: = diameter')
(height: = height')]

Production 2:

⟨body⟩ {contact_1 contact_2} [vertical_axis diameter height]: =
⟨bottom⟩ {end'} [normal' diameter']
+ ⟨cylinder⟩ {end'} [axis" diameter" height"]

A: [end' = end"]

S: [(normal' = axis") and (diameter": = diameter')]

G_a: [(contact_1: = "free") (contact_2: = "free")]

G_s: [(vertical_axis: = axis') (diameter: = diameter')
(height: = height')]

where "free" is on the cylinder and matches anything.

Identities:

⟨bottom⟩ {end} [n d] = ⟨circle⟩ {end} [n d]

⟨handle⟩ {e_1 e_2} [h a] = ⟨curve segment⟩ {e_1 e_2} [h a].

The compiled constraints "Neighbors" and "Parallel" are shown in Fig. 5 where a blank in the table means zero. Also, 0 means the relation does not hold, while a 1 means that it does. These tables can then be used in the normal manner by HCP as described in Henderson.[13] Note that the parallel constraint is quite powerful since every symbol is parallel to every other. This makes it possible to detect errors very quickly.

4. DISCUSSION

We have shown one possible approach to constructing a framework for shape parsing, and have shown how to implement a bottom-up, constraint-driven parsing mechanism. In addition, we have explained how it relates to traditional string parser theory. For the syntactic approach to prove useful, it is necessary that a clear conceptual relation should exist between the grammar and the parser. Moreover, parser generators must be made available to make the development of shape grammars and their parsers feasible and effective.

In analyzing a class of shapes, we proceed as follows:

—define a shape grammar for the class of shapes,

—derive the syntactic and semantic constraints between the vocabulary symbols of the grammar, and

—apply the parsing procedure to a set of shape

primitives using the constraints to produce a parse (perhaps by eliminating incorrect hypotheses).

Successful experiments have been run for detecting various kinds of shapes. However, several problems have been encountered. Shape grammars can have many productions, and a convenient means for defining a grammar has yet to be developed. Thus, at present, shape grammars are a major source of error and usually require much debugging. One solution to this is an interactive, graphical shape grammar specification system; another attractive approach is to generate such grammars directly from a CAD design.

There are two other critical issues which we are now studying. First, we would like to be able to provide deterministic parse tables for shape grammars. This could tremendously speed up the parsing process. The second issue is the choice of constraints (perhaps even the combination of several constraints). Currently, this is guided directly by the kinds of relations specified in the grammar (e.g. parallel, relative length, etc.). However, a deeper analysis might radically reduce the size and number of tables used in the parser.

SUMMARY

We believe that the syntactic method offers many advantages for shape analysis. The major advantage is the possibility of defining logical relations between anthropomorphically significant parts of a shape. Moreover, formal techniques allow both the automatic generation of constraint relations for grammatical descriptions of shape, and the application of these constraints during the analysis of shape. Various approaches have been proposed for syntactic or grammatical shape models, but in general, the parsing methods for these models are standard string parsers, e.g. Earley's algorithm. In order to obtain the most advantage from the grammatical approach, however, the relation between the shape grammar and the shape parsing method must be formally established. In this paper, we consider a bottom-up parsing mechanism and its relation to a particular class of shape grammars. Our goal is to outline a framework for a coherent approach to syntactic pattern recognition.

Our general shape grammar scheme is to produce a shape parsing mechanism by means of a shape grammar compiler from a high-level shape grammar description. This is analogous to using an automatic parser generator to produce a string parser from a high-level programming language description. Note that this contrasts with most syntactic methods; they use a general context-free parser. (However, for an approach which is similar in spirit, see Bunke's attributed programmed graph grammar transformation system.) The shape parsing mechanism performs the actual analysis of unknown shapes and outputs an organization imposed on the shape primitives in terms of the underlying grammar.

As an example of one approach to syntactic shape analysis, we propose a generalization of LR parsing as the framework within which to define shape grammars and their parsers. The major motivation for this choice is that methods exist for automatically deriving the shape parsing mechanism. In particular, we have:

—a shape grammar formalism which accounts for most structural aspects of 2-D and 3-D shape,

—a table-driven parsing mechanism which uses constraints between pieces of the shape, and

—an automatic method to compute constraint relations between the vocabulary symbols of the grammar.

This process can be viewed as a generalization of traditional table-driven grammar techniques in that the grammars involve constraints between string grammar symbols. With string grammars, bottom-up parsing involves scanning from left to right until the tail of the handle is found, then scanning right to left from the tail until the head of the handle is found. This works well enough for string grammars, but shape grammars pose the problem of complicated relations between the symbols, and these relations must be accounted for and taken advantage of by the shape parsing mechanism.

REFERENCES

1. A. C. Shaw, A formal picture description scheme as a basis for picture processing systems, *Inf. Control* **14,** 9–52 (1969).
2. K. S. Fu and B. K. Bhargava, Tree systems for syntactic pattern recognition, *IEEE Trans. Comput.* **C-22,** 1087–1099 (1973).
3. A. Rosenfeld and D. Milgram, Web automata and web grammars, *Machine Intell.*, pp. 3307–3324. Edinburgh University Press, Edinburgh (1972).
4. K. S. Fu, *Syntactic Methods in Pattern Recognition, Mathematics in Science and Engineering,* Vol. 112. Academic Press, New York (1974).
5. C. R. Gonzalez and M. G. Thomason, *Syntactic Pattern Recognition, Applied Mathematics and Computation,* Vol. 14. Addison-Wesley, Reading, MA (1978).
6. T. Pavilidis, *Structural Pattern Recognition.* Springer, Berlin (1977).
7. Q. Y. Shi and K. S. Fu, Parsing and translation of (attributed) expansive graph languages for scene analysis, *IEEE Trans. Pattern Anal. Mach. Intell.* **PAMI-5,** 472–484 (1983).
8. S. Aho and J. D. Ullman, *The Theory of Parsing, Translation and Compiling,* Vol. 2. Prentice Hall, Englewood Cliffs, NJ (1973).
9. D. Gries, *Compiler Construction for Digital Computers.* John Wiley, New York (1969).
10. H. Bunke, Attributed programmed graph grammars and their application to schematic diagram interpretation, *IEEE Trans. Pattern Anal. Mach. Intell.* **PAMI-4,** 574–582 (1982).
11. K. You and K. S. Fu, Syntactic shape recognition, Tech. report in Image Understanding and Information Extraction, Summary Report, Purdue University, March (1977).
12. G. Masini and R. Mohr, Mirabelle, a system for structural analysis of drawings, *Pattern Recognition* **16,** 363–372 (1983).
13. T. Henderson and L. Davis, Hierarchical models and analysis of shape, *Pattern Recognition* **14,** 197–206 (1981).

14. W. C. Lin and K. S. Fu, A syntactic approach to 3-D object representation, *IEEE Trans. Pattern Anal. Mach. Intell.* **PAMI-6**, 351–364 (1984).
15. D. Waltz, Understanding line drawings of scenes with shadows, *The Psychology of Computer Vision,* P. H. Winston, Ed. McGraw-Hill, New York, pp. 19–91 (1975).
16. A. Rosenfeld, R. Hummel and S. Zucker, Scene labeling by relaxation operations, *IEEE Trans. Syst. Man Cybernet.* **Vol. SMC-6,** 420–433 (1976).

About the Author—THOMAS C. HENDERSON received the Ph.D. in Computer Science from the University of Texas in 1979. He then spent one year at Deutsche Vorschungs- und Versuchsanstalt fuer Luft- und Raumfahrt near Munich, West Germany as a Research Associate in the Image Analysis Group. The next year was spent as a visiting professor at the Institute National de la Recherche en Informatique et en Automatique near Roquencourt, France. In 1982, he took a regular faculty position at the University of Utah, where he is presently an Associate Professor.

Professor Henderson's research interests include artificial intelligence, computer vision and robotics, and he has published numerous papers in these areas. He is co-author of *Relaxation Techniques in Computer Vision* to be published by Oxford University Press.

About the Author—ASHOK SAMAL received his B.Tech. degree from the Indian Institute of Technology Kanpur, India in 1983. Since then he has worked as a Teaching Assistant and Research Assistant at the Department of Computer Science, University of Utah, Salt Lake City, USA, where he is working for his Ph.D.

APPENDIX A

The input to lex for the ECG waveform grammar is:

```
%%
[(prbt) (b|bb|bbb)]+   {printf("Normal ECG Pattern/n"); };
[a–z]+                 {printf("Abnormal ECG Pattern/n"); };
%%
```

6. APPENDIX B

The states generated by *yacc* for the median chromosome grammar are:

```
state 0
        $accept: _S $end

        c    shift 3
        .    error

        S    goto 1
        A    goto 2

state 1
        $accept: S_$end

        $end accept
        .    error

state 2
        S:   A_A

        c    shift 3
        .    error
        A    goto 4

state 3
        A:   c_B

        a    shift 9
        b    shift 8
        .    error

        B    goto 5
        F    goto 6
        H    goto 7

state 4
        S:   A A_    (1)

        .    reduce 1

state 5
        A:   c B_    (2)
        .    reduce 2

state 6
        B:   F_B E

        a    shift 9
        b    shift 8
        .    error

        B    goto 10
        F    goto 6
        H    goto 7

state 7
        B:   H_D J

        b    shift 8
        d    shift 13
        .    error

        F    goto 12
        D    goto 11

state 8
        F:   b_      (7)

        .    reduce 7

state 9
        H:   a_      (9)

        .    reduce 9

state 10
        B:   F B_E

        b    shift 15
        .    error

        E    goto 14

state 11
        B:   H D_J

        a    shift 17
        .    error

        J    goto 16

state 12
        D:   F_D E

        b    shift 8
        d    shift 13
        .    error

        F    goto 12
        D    goto 18

state 13
        D:   d_      (6)

        .    reduce 6
```

```
state 14
        B:  F B E_      (3)
        .   reduce 3
state 15
        E:  b_      (8)
        .   reduce 8
state 16
        B:  H D J_      (4)
        .   reduce 4
state 17
        J:  a_      (10)
        .   reduce 10
state 18
        D:  F D_E
        b   shift 15
        .   error
        E   goto 19
state 19
        D:  F D E_      (5)
        .   reduce 5

7/127 terminals, 8/300 nonterminals
11/600 grammar rules, 20/750 states
0 shift/reduce, 0 reduce/reduce conflicts reported
8/350 working sets used
memory: states, etc. 84/12000, parser 16/12000
11/600 distinct lookahead sets
3 extra closures
13 shift entries, 1 exceptions
14 goto entries
2 entries saved by goto default
Optimizer space used: input 48/12000, output 27/12000
27 table entries, 6 zero
maximum spread: 260, maximum offset: 259
```

APPENDIX C

The states generated by *yacc* for the acrocentric chromosome grammar are:

```
state 0
        $accept: _S $end
        c   shift 3
        .   error
        S   goto 1
        A   goto 2
state 1
        $accept: S_$end
        $end accept
        .   error
state 2
        S:  A_A
        c   shift 3
        .   error
        A   goto 4
state 3
        A:  c_B
        a   shift 10
        b   shift 8
        .   error
        B   goto 5
        F   goto 6
        R   goto 7
        H   goto 9
state 4
        S:  A A_      (1)
        .   reduce 1
state 5
        A:  c B_      (2)
        .   reduce 2
state 6
        B:  F_L
        a   shift 10
        b   shift 8
        .   error
        F   goto 13
        L   goto 11
        H   goto 12
state 7
        B:  R_E
        R:  R_E
        b   shift 15
        .   error
        E   goto 14
state 8
        F:  b_      (12)
        .   reduce 12
state 9
        R:  H_D J
        b   shift 8
        d   shift 19
        .   error
        F   goto 17
        D   goto 16
        W   goto 18
state 10
        H:  a_      (17)
        .   reduce 17
state 11
        B:  F L_      (3)
        .   reduce 3
state 12
        L:  H D J
        b   shift 8
        d   shift 19
        .   error
        F   goto 17
        D   goto 20
        W   goto 18
state 13
        L:  F_L
        a   shift 10
        b   shift 8
        .   error
        F   goto 13
        L   goto 21
        H   goto 12
state 14
        B:  R E_      (4)
        R:  R E_      (14)
        b   reduce 14
        .   reduce 4
```

```
state 15
        E:  b_      (7)
        .   reduce 7
state 16
        R:  H D_J
        a   shift 23
        .   error
        J   goto 22
state 17
        D:  F_G
        b   shift 8
        d   shift 26
        .   error
        F   goto 25
        G   goto 24
state 18
        D:  W_E
        W:  W_E
        b   shift 15
        .   error
        E   goto 27
state 19
        W:  d_      (16)
        .   reduce 16
state 20
        L:  H D_J
        a   shift 23
        .   error
        J   goto 28
state 21
        L:  F L_    (11)
        .   reduce 11
state 22
        R:  H D J_  (13)
        .   reduce 13
state 23
        J:  a_      (18)
        .   reduce 18
state 24
        D:  F G_    (5)
        .   reduce 5
state 25
        G:  F_G
        b   shift 8
        d   shift 26
        .   error
        F   goto 25
        G   goto 29
state 26
        G:  d_      (9)
        .   reduce 9
state 27
        D:  W E_    (6)
        W:  W E_    (15)
        b   reduce 15
        .   reduce 6
state 28
        L:  H D J_  (10)
        .   reduce 10
state 29
        G:  F G_    (8)
        .   reduce 8

7/127 terminals, 12/300 nonterminals
19/600 grammar rules, 30/750 states
0 shift/reduce, 0 reduce/reduce conflicts reported
12/350 working sets used
memory: states, etc. 142/12000, parser 29/12000
12/600 distinct lookahead sets
2 extra closures
20 shift entries, 3 exceptions
24 goto entries
3 entries saved by goto default
Optimizer space used: input 78/12000, output 37/12000
37 table entries, 4 zero
maximum spread: 260, maximum offset: 259.
```