1997

Analysis of Digital Logic Schematics Using Image Recognition

James A. Giles
University of North Florida

ANALYSIS OF DIGITAL LOGIC SCHEMATICS USING IMAGE RECOGNITION

by

James A. Giles

A thesis submitted to the
Department of Computer and Information Sciences in partial
fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES

August, 1997

The thesis "Analysis Of Digital Logic Schematics Using Image Recognition" submitted by James A. Giles in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

Signature Deleted

7/10/97

Dr. Yap S. Chua
Thesis Adviser and Committee Chairperson

Signature Deleted

7/16/97

Dr. Ralph Butler

Signature Deleted

7/16/97

Dr. Robert F. Roggio

Accepted for the Department of Computer and Information Sciences:

Signature Deleted

7/16/97

Dr. Charles N. Winton
Chairperson of the Department

Accepted for the College of Computing Sciences and Engineering:

Signature Deleted

7/16/97

Dr. Charles N. Winton
Dean of the College

Accepted for the University:

Signature Deleted

7/23/97

Dr. William J. Wilson
Dean of Graduate Studies

ACKNOWLEDGMENT

There are several people without whose unwavering support and encouragement completion of a Master's degree would not have been a possibility for me. I would like to give a special "thank you" to my wife Linda for her patience and help in giving me the necessary free time for study and preparation. I would like to give heart-felt thanks to my mother and father, Jean and Ted Wittner, without whose help it would not have been possible for me to go back to college. I would like to thank Dr. Yap Chua for his encouragement throughout my time at UNF, for his excellent teaching, hard work, and concern for all of his students. Also, I want to thank Dr. Ralph Butler, Dr. Robert Roggio, Dr. Krissten Cooper, Dr. Charles Winton, Mr. Paul Higbee, Dr. Layne Wallace, and Dr. Susan Wallace, for their hard work in providing excellent classes at UNF.

CONTENTS

FIGURES

ABSTRACT

This thesis presents the results of research in the area of automated recognition of digital logic schematics. The adaptation of a number of existing image processing techniques for use with this kind of image is discussed, and the concept of using sets of tokens to represent the overall drawing is explained in detail. Methods are given for using tokens to describe schematic component shapes, to represent the connections between components, and to provide sufficient information to a parser so that an equation can be generated.

A Microsoft Windows-based test program which runs under Windows 95 or Windows NT has been written to implement the ideas presented. This program accepts either scanned images of digital schematics, or computer-generated images in Microsoft Windows bitmap format as input. It analyzes the input schematic image for content, and produces a corresponding logical equation as output. It also provides the functionality necessary to build and maintain an image token library.

Chapter 1

INTRODUCTION

The field of automated image recognition is an area of Computer Science which offers some unique challenges and difficulties. The goal of a typical recognition system is to detect shapes in a binary image, and recognize them as understandable objects of some type, usually by comparing them with a library of known shapes. Once lower level objects have been recognized, inferencing and/or domain specific knowledge is often applied to put the recognized shapes together into higher level objects. At the end of the process, the image is represented by a set of constructs which contain information about all of the recognized objects. The resulting representation is usually much more compact than the original binary form, and the recognized objects may be manipulated and changed as whole entities, rather than simply as collections of pixel values.

Two of the most difficult challenges for the researcher have to do with the representation of visual objects in a form useful for recognition, and with the construction of a library of known objects which can be accessed efficiently,

so that the recognition process does not take an inordinate amount of time.

The techniques presented in this paper, and implemented in the source code and executable program provided with the paper, address both of these issues, and have produced favorable results with the small set of test images used in the project. One of the fundamental concepts in this project is the use of tokens as shape descriptors. This works well in the context of schematic diagrams, and is a natural approach to the problem when the desired result is an equation which describes the image. This approach applies some of the ideas presented in the paper "Knowledge-Directed Interpretation Of Mechanical Engineering Drawings" [Joseph92]; most notably the use of a YACC-generated parser as a component of an image processing program.

In the remainder of this chapter we will provide a brief description of the image recognition process in general, and a summary of some of the processing techniques commonly used in the traditional approach to image processing. Then we will look at neural networks and how they can be applied, using methods which depart from the traditional approach.

1.1 The Image Recognition Process

Analysis and recognition of an image is usually a multi-step process, and when the traditional approach is used, each of these steps is fairly well defined. We can break down the process into four steps: Data Capture, Pre-Processing, Feature Extraction, and Recognition. In the following sections we will present a brief survey of the specific methods used in each of these four steps, and also how neural networks can be used in a non-traditional way to analyze images.

1.2 A Survey Of Recognition Techniques And Applications

1.2.1 Traditional Approaches

1.2.1.1 Data Capture

Image information may be captured by the computer in either an "off-line" or "on-line" mode. When paper documents are analyzed, they are typically read into the computer system using an image scanner. The scanner produces an electronic file which contains color and/or intensity information for

each pixel on the video display monitor or printer that will be used to display or re-print the image. This is "off-line" data capture, where all of the electronic information about the image is produced directly from the image itself, and is stored in a file that serves as the input for further image processing.

When "on-line" data capture is used, information about the image is captured directly as the image is being produced. A typical device which allows this kind of input is the digitizer pad and pen, where the user draws images (or hand-written text) on the electronic pad with the hand-held pen, and receives feedback information about the resulting image through a video display monitor and printer. When an image (or text) is captured in this manner, it is possible to provide more information to the computer system than can be provided using "off-line" capture. For example, pen speed, pressure, and the order and direction of individual strokes can be provided in addition to the actual resulting image data. This extra information can sometimes be very helpful in interpreting the resulting image.

The capture of visual information through direct input is an important technological advancement, which will have an increasing impact on daily life in the future. However, the

"off-line" capture of scanned paper images, medical CAT scans, radar images, and many other kinds of visual data presently has a much wider range of practical application. It is this kind of data capture which is the source of input for the project we are describing in this paper.

1.2.1.2 Pre-Processing

The next step in the traditional approach to image processing may be called "Pre-Processing", because operations are performed on the input data at this point which do not result directly in recognition, but which prepare the image for further analysis, and which help to simplify the computation necessary in later steps. We will look briefly at three important pre-processing functions: thresholding, noise reduction, and thinning. Although image pre-processing may include several other operations which are performed on the image, these three are typical functions used in many image applications, and they are often critical to successful recognition.

Thresholding is an operation which is performed on grayscale or color images in order to clearly separate the "background" of an image from the "foreground". The "foreground" consists of the objects of interest which are

depicted in the image. The "background" consists of the image area which surrounds the "foreground" objects. In order to have a clear distinction between "foreground" and "background", we would like to reduce the number of shades of gray (or color) so that there is one shade for the "foreground", and another distinct shade for the "background". In a typical grayscale image, there are 256 possible shades of color ranging from pure white, through many shades of gray, to pure black. A numeric value is assigned to each possible shade, with 0 being pure black, 255 being pure white, and all other values representing the various shades of gray. The goal (typically) of the thresholding function is to reduce the number of shades in the image to the values 0 and 255, where all "foreground" pixels have the value 0, and all "background" pixels have the value 255. This idea can of course be easily extended to color images, where typically there is a red, green, and blue component to every color in the image, and where the intensity of each component is represented by the same numeric value range: 0 - 255.

In order to accomplish proper thresholding, it is necessary to examine each pixel, and make an accurate decision as to whether it belongs to the image "foreground" or "background". In the case of images which have a fairly

uniform distinction between "background" and "foreground" colors across the entire image, it is possible to come up with a single numeric value, called a "global threshold", which can be used to make the "foreground"/"background" decision for each image pixel. Once the appropriate value has been found, it is compared with each pixel, and if the pixel color value falls on one side of the threshold, the pixel's color is changed to the pure "background" color; if it falls on the other side of the threshold value, the pixel's color is changed to the pure "foreground" color.

The process becomes more difficult when either the image contains a lot of "noise" (extraneous shades of color in many of the pixels which do not accurately represent the image, often occuring because of poor quality image scanning, faulty equipment, etc.), or the image contains many variations in shading or color at different locations. In this case, a "global threshold" will not provide the appropriate value for making the "foreground"/"background" decision in all parts of the image. In order to get around this problem, it becomes necessary to use "adaptive thresholding" techniques, which "adapt" to local shading values in different parts of the image. This is often accomplished by making the "foreground"/"background" decision based on the color values present in pixels which

surround the pixel of interest, or in other words, by looking at a small "window" of color data around the pixel, deriving a local threshold value based on that data, and changing the pixel of interest based on the local threshold.

Even adaptive thresholding as described above is not always sufficient to give accurate results when there are many variations in color throughout an image. In some cases it is necessary to make the window size (the amount of data analyzed around the pixel of interest) adjustable by the user, or even to make the window size dynamically adjustable by the program, and have it vary the size during analysis, based on domain-specific information about the image.

Regardless of whether a global or local threshold is used, it is necessary to have a method for determining the threshold value based on either the data in the entire image, or in a small window. Several different methods may be used, including (among others) manual adjustment, histogram-based selection, weighted-histogram selection, and statistical selection.

Manual adjustment is the simplest of the methods; the user selects a threshold value manually, and has the program apply it to the image. Adjustments to the value are then made manually, until the results are optimal.

Histogram-based selection is done by compiling a histogram of pixel color values throughout the image. The histogram can be thought of as a two-dimensional graph, with color intensity on the x-axis, and number of pixels on the y-axis. When there is a fairly clear distinction between "foreground" and "background" already present in the image, the graph will have two distinct "peaks", one for the primary "foreground" intensity, and one for the primary "background" intensity. The threshold can then be chosen in the "trough" between the two peaks. If there is not a fairly clear distinction between "foreground" and "background", then the "peaks" and "troughs" will not be distinct, and it becomes difficult to choose the proper threshold value.

Weighted-histogram selection is based on the same kind of histogram just described, except that the pixel count for a given intensity value is weighted, depending on the location of pixels of that intensity in the image. Pixels which are in the interior region of an object in the image

will be weighted more heavily than pixels near the edge of the object.

Statistical selection of the threshold value typically involves dividing the pixels in the image into foreground and background classes. Various threshold values are tried experimentally, and the resulting class membership is examined for each potential value. The threshold which maximizes the variance in intensity values between the two classes, and which minimizes the variance in intensities within each class is selected and applied to the image.

There are numerous other thresholding techniques which work well in different situations, such as the "YDH", "Nonlinear Adaptive", and "Integrated Function" techniques, as reported in [Kamel93]. For further general reading on this topic, see [O'Gorman95], chapter 2.

Noise reduction: "Noise", in the context of image processing, can refer to any kind of distortion of the information in the image. The distortion may result from the application of computerized algorithms to the image data, or it may simply be caused by errors introduced at the time the image was created. When we are talking about image pre-processing, we are primarily interested in the

removal of noise which was introduced during the creation of the image. More specifically, erroneous color values may be set in individual pixels in the image, usually because of scanning equipment problems, poor photographic reproduction, poor quality facsimile transmission, and the like. The noise often takes the form of isolated ON pixels within OFF regions, and isolated OFF pixels in ON regions. It can also appear as lightly shaded gray areas or white areas or black areas (in a grayscale image) in a region of the image which should be some other color, or as minor distortions in the shapes of objects within the image (or some combination of all of these).

Numerous methods have been used to correct image noise; we will briefly discuss two of them, both of which are based on the concept of analyzing a "window" of data around a pixel of interest, and setting that pixel's color value according to the results of the analysis. Usually, the "window" of data consists of a square matrix of pixels, with the pixel of interest at the center of the matrix. Each pixel in the image is examined (as the pixel of interest), and it's color value is set depending on the surrounding data.

The first method uses a pair of processes, called "erosion", and "dilation" in various combinations to smooth out the boundaries of shapes, remove extraneous white pixels from black areas and extraneous black pixels from white areas, join narrow gaps, fill in small "holes", and other similar processing. "Erosion" is the process of removing a thin layer of ON pixels from the boundaries of shapes in the image (usually with layers having one pixel of thickness) by turning them OFF. "Dilation" is just the opposite; a thin layer of OFF pixels surrounding the boundaries of the shapes in the image is turned ON (also typically with each layer having one pixel thickness). The "erosion" and "dilation" operations are applied to the image in groups, in order to perform "opening" or "closing" on the shapes in the image.

To perform "opening", where boundaries are smoothed out and small areas of noise are removed, one or more iterations of "erosion" are performed on the image, followed by the same number of iterations of "dilation". Although the layers of pixels removed by the "erosion" are replaced by the "dilation", the replacement is not exactly the same as the removal, and the result is the desired noise reduction.

The "Closing" operation smoothes out boundaries, joins narrow gaps, and fills small "holes" in the image caused by noise. It is accomplished by performing one or more iterations of "dilation", followed by the same number of iterations of "erosion". The extra layers of pixels added by the "dilation" are removed by the "erosion", but the removal is not exactly the same as the addition, and the result is the desired noise reduction.

The second noise reduction method (or filter) which we will look at is similar to "opening" and "closing", in that it modifies pixels in the image based on a "window" of surrounding pixel data, but it does not necessarily remove or add layers around the entire boundary of a shape. As in the previous example, the analysis "window" is passed over the whole image so that each pixel becomes the pixel of interest at some point, and that pixel is changed according to the rules of the particular algorithm being used. The filter may be applied one or more times to the image, until the desired results are achieved. One example of an algorithm for modifying the color of the pixel of interest can be found in [Andrews76]: the average color value of all of the pixels in the "window" is calculated, and is used as a local threshold. If the color value of the pixel of interest is less than (blacker than) the average value, it

is set to pure black. Otherwise, it is set to pure white. See [O'Gorman95], chapter 2 for more details about these and other noise reduction techniques.

Thinning: This is the third and final image pre-processing technique which we will cover in this introductory material. When it is applied to an image, all of the shapes are reduced to a set of thin lines (usually with one pixel thickness) which approximately traverse the center of each original shape, thus giving a "skeleton" outline of the image content. Thinning is particularly useful when we are dealing with elongated shapes such as lines, which we wish to describe as a set of vectors, or in some other way which is usable by a computer. It is much easier to produce vectors which accurately describe the original shape by following these "skeleton" lines than it is to attempt to follow thicker shape contours.

In order for the "skeleton" lines to be useful for image interpretation, it is necessary that the thinning algorithm adhere to several constraints, as given in [O'Gorman95], page 16:

1. Connected image regions must thin to connected line structures.
2. The thinned result should be minimally eight-

connected (explained below).

3. Approximate end line locations should be maintained.
4. The thinning results should approximate the medial lines.
5. Extraneous spurs (short branches) caused by thinning should be minimized.

Item 1 ensures that the representative "skeleton" lines will not loose any of the connections which were originally present among the shapes in the image. Item 2 stipulates that there should be only one pixel of thickness at connection points between shapes. The term "minimally eight-connected", as applied to any two connected pixels in an image, means that each pixel is one of the eight possible neighbors of the other, and there are no other immediately adjacent neighbors present. Item 3 ensures that "skeleton" lines are not shortened significantly (from the length of the original shape) by the thinning process. Item 4 specifies that the resulting "skeleton" lines should closely approximate the medial axis of each original shape. The medial axis consists of the set of points taken from the interior of the shape such that each point is equidistant from it's two closest neighbors on the shape's boundary. (In other words, it is approximately the center

line of each shape). Item 5 says that the thinning process should not leave behind short spur lines attached to the "skeleton" line data (because they can cause confusion in later processing). See [O'Gorman95], pages 14-18 for more general information about the thinning process.

As with all of the other techniques that we are considering, there are numerous ways to implement thinning. We will look in detail at one typical method, which is also used in the program developed for this project. For more information about the process we are about to describe, see [Lam92].

We need to define some notation and terminology (as specified in [Lam92])in order to describe this method, which was created by C. J. Hilditch in 1969. First, consider the neighborhood of pixels surrounding the pixel of interest (p), as in the diagram below.

Let N(p) represent this neighborhood of pixels around p. The pixels x_1, x_2, ..., x_8 are said to be the 8-neighbors of p, and are 8-adjacent to p. The pixels x_1, x_3, x_5, and x_7 are also said to be 4-neighbors of p, and they are 4-adjacent to p.

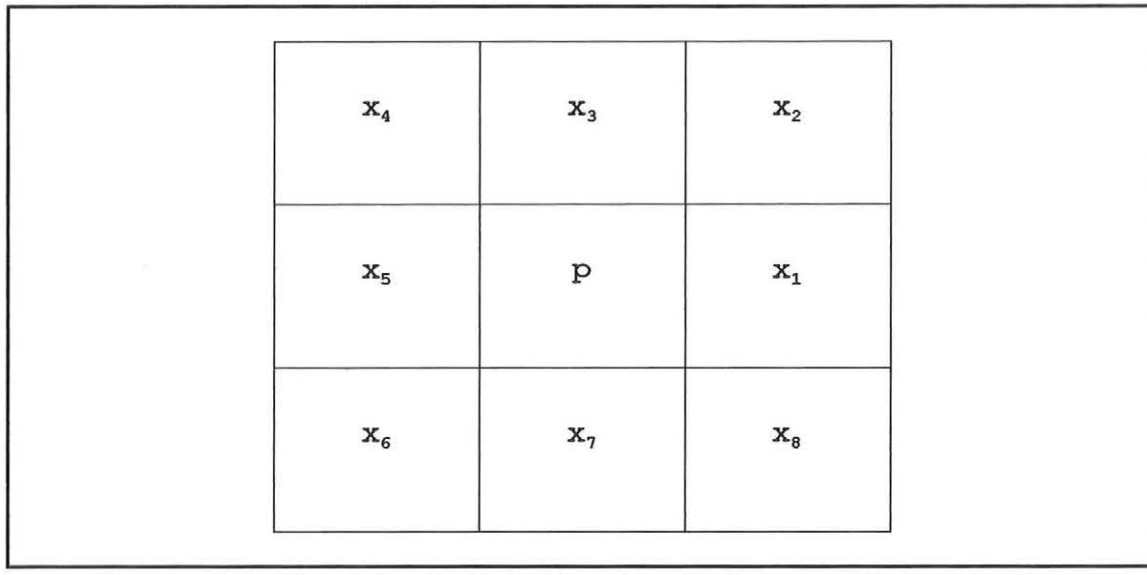


Figure 1: The Pixel Neighborhood Surrounding Pixel p

For each new location of the pixel p, we will need to calculate what Hilditch calls a "crossing number" and defines as "the number of times one crosses over from a white point to a black point when the points in N(p) are traversed in order, cutting the corner between 8-adjacent black four neighbors" [Lam92]. Algorithmically, we can calculate this number as follows, letting C be the "crossing" number:

1. C = 0.
2. If $x_1 = 0$ and $x_2 = 1$ and $x_3 = 1$, add 1 to C.
3. If $x_3 = 0$ and $x_4 = 1$ and $x_5 = 1$, add 1 to C.
4. If $x_5 = 0$ and $x_6 = 1$ and $x_7 = 1$, add 1 to C.
5. If $x_7 = 0$ and $x_8 = 1$ and $x_1 = 1$, add 1 to C.

With all of this in mind, we now want to pass the 3x3 "window" shown in figure 1 over the entire image, processing in raster scan order (left to right, and top to bottom), where the pixel of interest p is placed at each pixel location in the image. When p is located over a pixel on the edge of the image, the neighboring points about p which fall outside of the bounds of the image are considered to be 0 or OFF. At each new location of the "window", apply the rules listed below to determine if the point p should be set to 1 (ON) or 0 (OFF). Points which are to be turned off (or deleted) are not immediately turned off at the time the determination is made, because their present value is still needed to make a decision about neighboring points. They are flagged for deletion, and at the end of one complete raster scan, all points flagged for deletion are turned off. The raster scan is then repeated until no points have been flagged for deletion during the last pass.

Rules for deletion (setting the color value to 0, or OFF) of the point p:

1. p must be currently turned ON.
2. p must not be isolated, or an end point, which means that there must be at least two black (ON) neighbors adjacent to p.

3. p must be located on the contour of a shape, which means that p must have at least one white (OFF) 4-neighbor.

4. At least one black (ON) neighbor of p must not have already been flagged for deletion.

5. The "crossing number" must be 1 at the start of the current raster scan.

6. If x_3 has been flagged for deletion, then setting x_3 to 0 (OFF) must not change the crossing number which would be calculated at the current location of p.

7. If x_5 has been flagged for deletion, then setting x_5 to 0 (OFF) must not change the crossing number which would be calculated at the current location of p.

8. If rules 1-7 are met at the current location of p, then pixel p is flagged for deletion at the end of the current raster scan. Otherwise, p is not flagged for deletion.

This algorithm has been implemented in the program developed for this thesis, and in the cases tested by the author it successfully meets all of the criteria set forth by O'Gorman (above), except that some spur lines are created at locations where a lot of points are removed from the image by thinning, and the contours of the original shapes at those locations are not extremely smooth.

1.2.1.3 Feature Extraction

"Feature extraction" is a broad term which encompasses several different kinds of processes that are applied to the image after pre-processing has been completed. One of these processes is often called "segmentation", where the individual shapes that make up the content of the image are precisely located, distinguished from background information and from other shapes (even when the shapes partially overlap each other, in some cases), and are quantified into groups of numbers of some sort (often vector coordinates). At a smaller scale, "feature extraction" also means the identification of important features within individual shapes (the location and shape of the wings of an airplane, for example).

As we have already mentioned, there are many ways to go about each part of the image recognition process, and feature extraction is no exception. In order to remain within the scope of this project, we will briefly discuss a few of the more important techniques here, and leave further investigation to the reader. The references included at the end of the paper provide a good starting point for additional research.

One of our first concerns with regard to feature extraction has to do with the encoding of shapes into a form which can be manipulated by the computer. We will discuss two common methods which are used for this purpose: vectorization, and chain coding.

Vectorization involves the detection of lines and contours in the image, and the representation of those lines with a set of vectors. Coordinates for the vectors are typically stored in the computer system, and form the basis for further processing. In the author's opinion, vectorization is on the borderline between image pre-processing and feature extraction. When performed in isolation, it can be considered a low level technique, but it can also be combined with higher level processes such as curvature detection. As we will see shortly, the program developed for this project uses a hybrid technique which combines vectorization with both curvature measurement and polygonalization, all in one operation.

In [O'Gorman95], pages 22-23, there is a good introduction to vectorization, and methods are presented which can be used on a non-thinned image. In one method, horizontal straight lines in the image are found first, then adjacent horizontal lines on neighboring scan lines are grouped together and connected, and other remaining lines are then

found and appropriate connections to existing vectors are made. Another method is also discussed in which digitizing hardware has the capability of tracking along straight lines as they are scanned, and performing vectorization during creation of the image. In this paper, rather than going into detail on these methods, we will discuss the hybrid technique used in the program developed for this project.

This technique (better known as a "thick line" method) works on images which have been binarized and thinned, and it provides vectorized approximations of both straight lines and curves. The starting data point for each series (or chain) of vectors is found by performing a raster scan on the image. When a data point is found, the program switches immediately to a "line-following" mode of operation, where it follows connected data points in any direction. Figure 2 below illustrates the technique,using a typical OR gate as an example.

Starting at data point A, we wish to draw a series of vectors which describe the OR gate. We proceed along the connected ON pixels, and extend an imaginary line from A to the current end-point. We then find the perpendicular distance from that line to every ON pixel we have already

traversed. If any of the distance measurements exceed a user-specified maximum, then the approximation provided by the line is too far from the data, so we back up to the nearest ON pixel we have already traversed, and define a data point (or vector end-point) there. The process then starts over at the new data point. In the diagram, we extend a line to pixel C, and measure the distance from line segment AC to previous data point B. Given that this

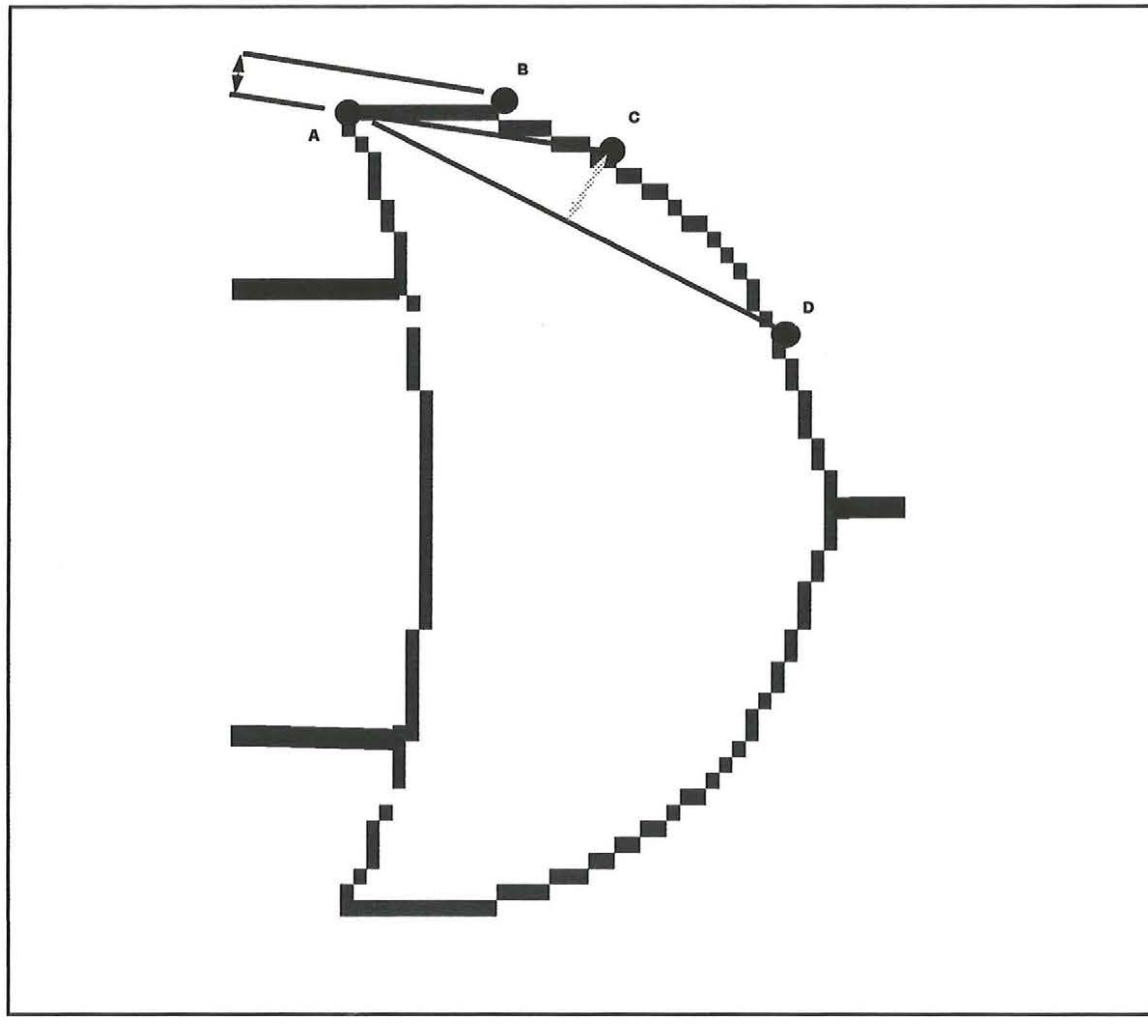


Figure 2: Thick Line Vectorization

distance is acceptable, we continue in like manner until we reach pixel D. Here, we find that the distance from the line to pixel C is too great. so we back up and establish a vector data point at the pixel just prior to D. Then we start over with this process from the new data point. (In the example we are showing pixels A, B, C, and D as isolated pixels some distance apart. In actual images there would be many connected pixels between A, B, C, and D.)

The vectorization technique also employs curvature measurement, and determination of right (clockwise) or left (counter-clockwise) curvature, which is calculated as illustrated in figure 3 below.

After the vectors AB and BC are generated, using the methods illustrated in figure 2, we want to measure the change in direction encountered as we traverse the shape from the first vector to the second. To do this, we extend vector AB out to point D such that BD has the same length as BC, and measure angle DBC, which we have labeled as ß in the drawing. Let r be the length of BD, and let q be the length of CD. Noting that triangle BCD is isosceles, we can extend a segment from B to the midpoint of CD, which we will call point Q. Then let h be the length of segment BQ.

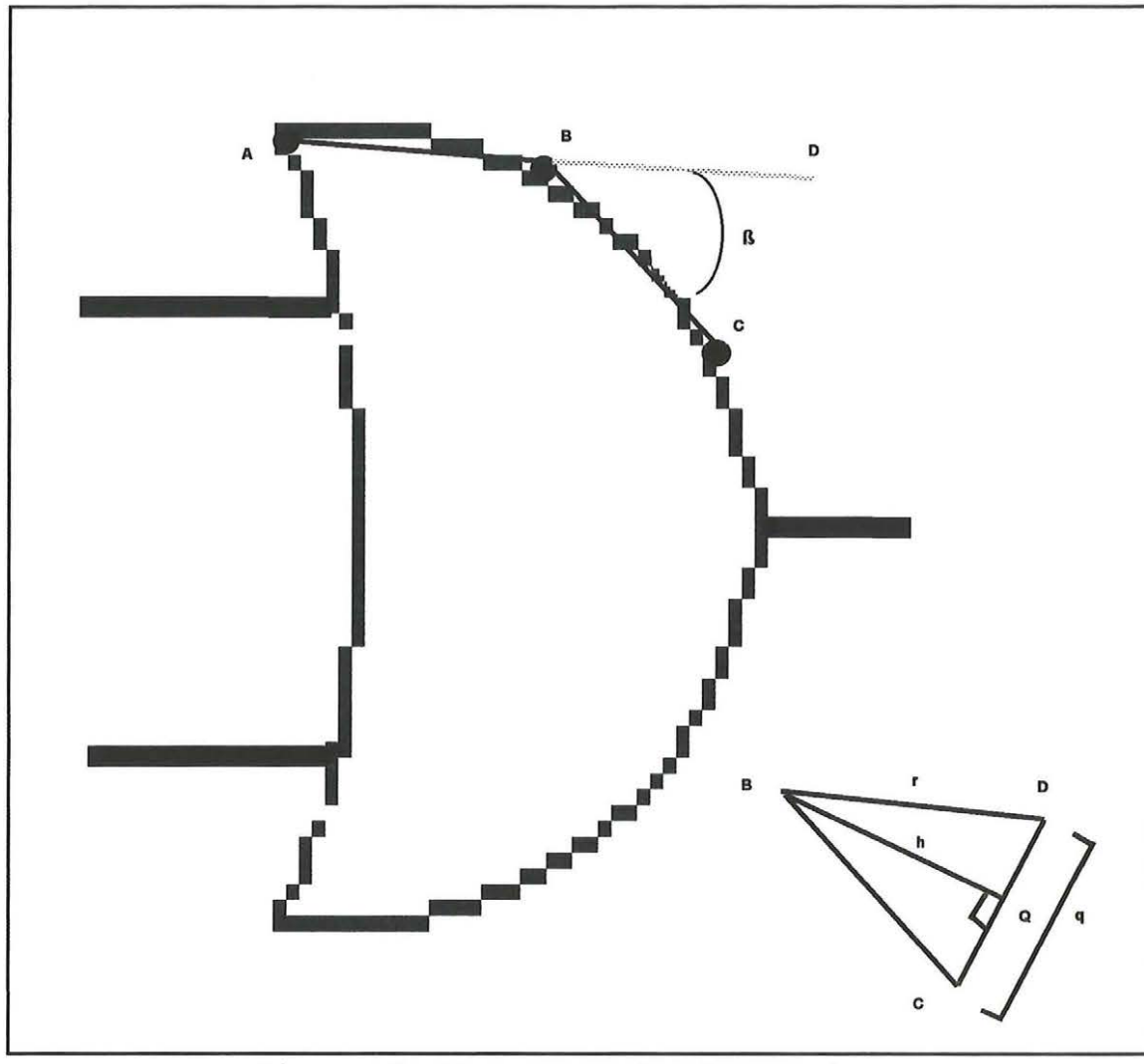


Figure 3: Measurement Of Curvature

Then, we have:

$$h = r \cos ß/2$$

$$\cos ß/2 = h/r$$

$$ß/2 = \arccos (h/r)$$

$$ß = 2 \arccos (h/r)$$

We can find r and q by using the distance formula, plugging in points B and D, and C and D respectively. We can also find h by using the Pythagorean theorem to write: $r^2 = h^2 +$

$(q/2)^2$, and solve for h. Then we can solve for ß given h and r.

To determine left or right curvature, imagine a three dimensional coordinate system with the xy plane on the surface of figure 3, and with the positive z axis extending straight up out of the paper toward the reader from the origin at point B. As we traverse the vectors from AB to BC, we have a clockwise change in direction of ß degrees, which we can also call "right curvature". We can apply the "right hand rule" for vector cross products, to see that BD X BC will result in a vector of non-zero length on the negative z axis for right (or clockwise) curvature, and it will result in a vector on the positive z axis for left (or counterclockwise) curvature. So to determine mathematically if we have right or left curvature in the example, let point D have the three dimensional coordinates (D1, D2, 0), and let point C have coordinates (C1, C2, 0). Then we have: (D1, D2, 0) X (C1, C2, 0) = (0, 0, Z), where Z is non-zero. If Z is negative, we have right curvature; if it is positive we have left curvature.

In the example of figure 3, we are traversing the OR gate shape in a clockwise direction (i.e. traversing from vector AB to vector BC), and we would determine that we have right

curvature in this traversal. If we started at point C and traversed the shape in the opposite direction, we would have found left curvature between these same two vectors. In the program developed for this project, when component shapes are detected, the program ensures that the traversal is always made in the counter-clockwise direction, so that the determination of left or right curves is always consistent. This method is adapted from a discussion on adaptive and multiscale methods for critical point detection in [O'Gorman95], which is nearly identical, except that in O'Gorman's method, vectors are extended on either side of a data point until a threshold distance from the actual curve is reached.

The program developed for this project also adds another extension to the "thick line" method: the user is able to specify both a maximum distance from the approximating vector to the actual data line, and also a maximum curvature. When the curvature between two vectors exceeds this maximum, a vector end point is forced at the location of high curvature.

Overall, this hybrid method combines vectorization, polygonalization (approximating curves with a series of straight lines), and critical point detection (finding

points of maximum curvature in shape contours) into one integrated algorithm.

When vectorization is performed on images which have already been thinned, or which have had other pre-processing techniques applied, the resulting vectors are subject to distortion error introduced by the earlier processes. Many variations of the basic vectorization method have been tried, with modifications to overcome distortion (and other) errors. One interesting example of this is a method described in [Hori92], in which document analysis was being performed on the buildings drawn in a city map. The lines which made up the building shapes were relatively thick, so that there was considerable distortion in the original vectorization (i.e. square buildings would be vectorized with rounded corners, etc.). The researcher overcame these problems by implementing a low level recognition process as a part of vectorization. Simple closed polygons with only a few sides and with an enclosed area which fell within a correct range (determined by statistical analysis) were determined to be buildings. When the vectors which described these polygons did not have parallel sides, their end points were adjusted to fit more closely with the expected shape of a building. This adjustment resulted in much greater overall accuracy.

For yet another method which performs vectorization similar to that described in figure 2, but with minimal calculations, see [Wall84].

Another method for encoding shapes which deserves mention is "chain coding". Rather than forming a reasonably close approximation of a shape, chain coding attempts to follow the data lines in an image exactly. This may or may not be desirable, depending on the application, because the approximations created by vectorization often help to smooth out local "noise", and make it easier to discover the true nature of the shape which is being encoded. The essence of chain coding is that each pixel in a line is encoded, and the direction from the current pixel to it's connected neighbor is stored, rather than the absolute coordinates of each pixel. A raster search is performed initially, and the absolute coordinates of the first pixel are saved, but from that point on, as connected pixels are found, the only information encoded is a single number which indicates direction. The number is usually in the range 0-7, and refers to one of the 8 possible 8-connected neighbors of the current pixel. See [O'Gorman95], pages 19-22 for more information.

The detection of "critical" or "dominant" points is another important part of feature extraction. The points of extreme curvature are vital in providing an accurate description of a shape. We have already discussed one method for measuring curvature, as shown in figure 3. If we look at all of the curvature measurements for all vectors, and apply a threshold value, such that all measured curvatures beyond the threshold indicate corners (or critical, or dominant points), we have a useful means of identifying the curvature extremes in the shape. There are many other ways of finding the critical points, but the example of figure 3 provides a good representative method, and is sufficient for the purposes of this project.

The overall goal of feature extraction and description is to provide an accurate representation of a shape in a form which can be successfully compared with known forms in order to arrive at an identification (in the context of model-based recognition, which is the focus of this paper). Polygonalization is one of several methods used to describe shapes in a manner useful for recognition. Other methods use known shapes which can be easily described mathematically, and fit them to the data points in the image.

One of the simplest of these methods fits a combination of straight lines and circular curves to the data. For data points which come close to forming a straight line, the equation of an exact straight line is found such that the error distance from this line to all of the data points it approximates is minimized. For data points which form curves, the equation of an exact circle is found which closely fits the data points. When straight lines merge into a curve, the transition points (from straight line to curve) are found, and then the location of the center of the circle (which approximates the curve data points) is found based on the transition points. Corners which have some rounding can be distinguished from curves by using a threshold value for the arc length of the curve. See [O'Gorman95], pages 102-103.

Other methods of shape description use higher order mathematical equations that produce smooth curves, and the curves are adjusted to fit the data points. One example of this is the B-spline, which is a curve generated by a polynomial (often third degree). Given a set of data points where there are two end points and a series of points in between which describe a curve, a B-spline can be generated which exactly fits the end points, and which forms a smooth curve that approximates the shape described by the

intermediate data points. See [O'Gorman95], pages 103-105 for more information, and [Medioni87] for a specific example of the use of B-splines.

The methods of higher level shape description used by the program developed for this project differ from all of the methods presented here, and will be discussed in detail in chapter 2.

1.2.1.4 Recognition

After the objects in an image have been encoded and higher level shape descriptions have been applied, the next major phase in a (model based) recognition system is to match the shape descriptions with a set of known shapes which have already been analyzed using the same descriptive technique, and identify the shapes in the image based on successful matching with the known shapes.

We will present here a summary of recognition methods used by several different researchers.

1. General recognition methodology [O'Gorman 95]. Lawrence O'Gorman presents four common methods for recognition:

Shape Metrics:

One of the easiest methods for distinguishing among shapes is to use simple measurements related to the size, such as the total area within the shape (which can be easily measured by counting the number of ON pixels or by using faster area calculations when shapes are regular), or the total perimeter around the boundary. If there is enough distinction between the metrics of each possible shape in a particular type of image, then this method may provide a simple solution to the problem.

Analysis Of Moments:

The "moments" of a shape can be used as a mathematical measure of the "nonroundness, eccentricity, or elongation" of a shape [O'Gorman95]. There are several different kinds of mathematical "moments", including: 1. The "first moment", which is the average (x, y) coordinate for all data points in the shape region, where all individual x coordinates are summed, and the result is divided by the number of points in the region, and the average y coordinate is calculated in a similar fashion. This average (x, y) coordinate pair gives the average location of the

shape. 2. The "second moment" is calculated like the "first moment", except that the squares of each x value (and y value) are summed, and the average is taken. 3. The "third moment" is calculated like the second, except that the cube of each coordinate value is summed. This continues in like manner for higher order moments. The first, second, third, (and so on) "central moments" are measurements which are normalized with respect to the first moment, so that they are independent of location in the image. The "second central moment" is calculated as follows:

$$m_x^{(2)} = (1/n)\Sigma(x_i - m_x^{(1)})^2$$

$$m_y^{(2)} = (1/n)\Sigma(y_i - m_y^{(1)})^2$$

where $m_x^{(2)}$ is the second central moment x value, $m_y^{(2)}$ is the second central moment y value, $m_x^{(1)}$ is the first moment x value, $m_y^{(1)}$ is the first moment y value, and x_i and y_i are the ith data coordinate pair.

If there is enough distinction between the moments of the expected shapes in a class of images, then this kind of analysis can be used for recognition.

Topological Features:

Another simple method of distinguishing shapes is to simply

note the number of holes and branches found in each shape. For some applications this is sufficient to distinguish among all of the possible shapes.

Fourier Descriptors:

The Fourier transform of the curvature around the boundary of a shape can be calculated, giving a description of the shape in terms of Fourier coefficients (or Fourier descriptors). In many cases, the set of Fourier descriptors may be unique for each possible shape, and they can thus be useful in distinguishing among shapes. It is possible, though, for different shapes to have the same frequency information, and thus the same set of Fourier descriptors, so this method is not always sufficient by itself.

2. Recognition of mechanical engineering drawings using shape blocks and mesh encoding vectorization [Vaxivière90], [Vaxivière92].

Pascal Vaxivière and Karl Tombre are two scientists and educators who have been working on the problem of recognition of mechanical engineering drawings for a number of years, and in particular, the conversion of paper drawings into a CAD format usable by the French CAD system known as CATIA, from Dassault Systemes. Some of their research has been focused on the development of a program

named CELESSTIN which implements their methodology. Many aspects of their work are interesting and unique, but two areas which stand out in particular are the method used to perform vectorization, and the way that recognition is performed on image blocks and entities, making use of artificial intelligence techniques to identify complex components.

Vectorization is performed by subdividing the input image with a carefully sized square mesh, and then looking at the intersections of lines in the image with the sides of each square in the mesh. The square size is chosen based on prior knowledge of mechanical drawing standards, which in the case of their research is the French standard NF E 04-103. This standard specifies the relative thickness of "thin" and "thick" lines ("thin" lines must have one fourth the thickness of "thick" lines), the minimum distance between any two thick lines (.8mm), and the allowable distances between any two hatching lines (1.5mm - 2mm). Based on this standard, the square size of the mesh is selected so that each square is larger than the maximum line thickness, and smaller than the minimum distance between any two lines. This ensures that there is at most one line in any particular direction which intersects a square, and that there are no lines thick enough so that

the cross section of the line would overlap more than two adjacent squares. Numerical codes are assigned to the different possible ways in which the sides of the squares can be intersected by lines in the image. Each square is examined, and the vectors which represent the drawing are encoded using a series of the mesh-square codes. Large regions of ON pixels in the image are initially encoded as mesh squares containing all ON pixels, and during a subsequent line-following process, they are handled separately from the lines.

It may not be possible to represent some complex areas in the image, such as compound line junctions, with any of the defined mesh-square codes. When this situation is detected by CELESSTIN, the region is further broken down into a set of overlapping squares, each of which contains a simpler portion of the complex area, which can be properly encoded.

The mesh squares used for the encoding process are not fixed in size. CELESSTIN dynamically adjusts the size and location of the squares so that junction points are centered within a square, and lines which overlap the border between squares are placed inside the adjusted squares.

After the mesh encoding has been completed, other processing is performed to classify lines in the image as "thick" or "thin", remove line spurs, perform other adjustments to improve accuracy, and identify dot-dashed lines which show the axis of symmetry for components.

From this mesh-square vectorization, higher level constructs are built which allow shape recognition. Minimal closed polygons (or blocks) bounded by thick lines, which may contain thin line "attributes" (such as hatching) are found first. Then, a set of simple rules derived from the standards for mechanical drawings are applied to the blocks to determine if they represent the solid matter of some component in the drawing, or empty space surrounding a component. (For example, "Any minimal closed polygon with thick lines sharing at least one edge with the object's external contour has to enclose matter. If the polygon is hatched, it lies in the section plane; if it is empty, it may represent a piece in front of or behind the section plane" [Vaxivière92]).

CELESSTIN begins the recognition process by looking for blocks of solid matter which are crossed by the dot-dashed lines that represent the axis of symmetry. When a block is found which clearly represents solid matter, additional

domain-specific rules are followed to determine neighboring blocks which must also represent solid matter. Recognition of this type proceeds through all neighboring blocks until the boundaries of larger components have been found. Groups of connected blocks which make up the larger components are linked together to form larger single entities. It is then possible to match the large scale entities with specific CAD components taken from a library, for which precise technical details are known. The recognized entities are then replaced in the drawing with the precise CAD library components.

Many important details have been left out of this brief description, such as the ways in which CELESSTIN checks on its own accuracy of recognition by attempting to disassemble components which have been recognized (because it must be possible to disassemble all mechanical components). For more information, see [Vaxivière90] and [Vaxivière92].

3. Recognition of mechanical engineering drawings using a YACC-generated parser to control segmentation and recognition [Joseph92]

In the paper "Knowledge-Directed Interpretation of Mechanical Engineering Drawings", a totally different approach to image recognition is presented, and is implemented in a research program named ANON. Instead of following the usual bottom-to-top process where low level information is analyzed and built into progressively higher level constructs which eventually result in recognition, ANON combines both a top-down and bottom-up approach to the problem. It works directly on grayscale images without benefit of pre-processing, and a knowledge-directed search is performed from starting "seed" points in the image, to locate the next part of particular drawing constructs, starting with the location determined to be the most likely candidate, based on prior results.

ANON's control structure consists of a set of "schemas" which contain descriptions of the drawing entities expected in input images, a YACC-generated parser which applies search rules to the data and the schemas, and a set of low level image analysis functions which operate directly on the image.

Analysis starts with the partitioning of an input image into 9 x 12 pixel sub-areas, each of which is analyzed for a grayscale threshold level, noise characteristics, and

concentrations of ON-pixels (large black areas). Then, the search for drawing components begins at the locations where large concentrations of ON pixels have been found. A drawing-level schema is made active initially, which starts the search (in dense ON-pixel areas). Once the search has been initiated, an iterative process is maintained, as follows: a. The current schema directs exploration of the image in locations most likely to produce results (initially near the seed points in the image), by calling the low level search routines. (The drawing-level schema is active at the start of the analysis, and many other schemas which control the analysis of hatched lines, shapes, dimension indications, words, letters, etc. may be made active at different times during the course of the analysis). b. The low level search routines return results to the calling schema, which modifies itself according to the feedback it receives. c. The current schema may remain active, or control may be shifted to another more appropriate schema, depending on the feedback from the low level searches. The management of schemas is performed at a higher level by the execution of rules in the parser. d. The set of schemas which eventually provide a description of the image, are continually modified throughout the analysis in a three part "perception cycle" where the schema directs exploration, which in turn samples

the data in the image, which in turn causes the content of the schema to be modified, which results in further exploration, and so on.

Based on the author's description of ANON, it is a research program intended for the exploration of knowledge-directed recognition techniques, rather than a production-ready system designed to produce complete CAD information, as we have seen in the CELESSTIN program. However, the research presented in the ANON paper is valuable and indicates that accurate recognition of complex drawings is possible with little human intervention. The only information provided by the user is a "position and radius of a cursor used to specify an initial circular search" [Joseph92], and this is necessary only when the program is run in interactive mode. It can also be run in a fully automatic mode, where all search locations are determined by the program. See [Joseph92] for more information, and for processing examples which demonstrate ANON's recognition ability.

4. Fingerprint recognition [Ratha96]

The example we present here is of a fingerprint matching and retrieval system, designed for use in the context of a large multimedia database. Although the emphasis is on matching and retrieval, we are still dealing with model-

based image recognition, where the database of fingerprints contains the models, the image of the fingerprints in question is the input, and a suspect's name is the label we want to apply to the input image as a result of successful recognition. [Ratha96] presents an unnamed research system which uses a four step approach to the problem of complex image matching against an extremely large library (up to many terabytes of information).

In this system, a fingerprint query consists of a hierarchy of sub-queries, some of which are text-based, and some of which deal specifically with the content of fingerprint images. The text-based queries are executed first, because they are much less computationally intensive, and because the database is structured so that the search space can be narrowed to as little as 25% of the original library size by text-based retrieval. The image content sub-queries are then applied to the remaining images.

To retrieve potential fingerprint matches from the database, the following steps are followed:

a. A text-based search is performed first, based on textual identifying information which may be available, such as last name, age range, and color of hair.

b. A text-based search is executed on the result set from step a to match on the class of the fingerprint, which may be one of five basic classes: arch, tented arch, left loop, right loop, and whorl.

Pre-filtering the database with these first two queries has been shown to reduce the size of the search space by as much as 75% in the tests run by the researchers.

c. An image content query is executed on the result set from step b to extract images which have a "ridge count" that falls within a specified range of the "ridge count" value calculated for the input image. Only images which are retrieved by this query are retained for further processing.

To calculate the "ridge count", "core" and "delta" points must first be found, and the "ridge count" is determined by counting the number of ridges which intersect with an imaginary line drawn from a "core" point to a "delta" point. The "core" point is the top-most point of the inner-most ridge in the fingerprint, and a "delta" point is a location in the print which has three ridges radiating from it.

Although this method is conceptually simple (just counting lines from a specified starting location to a specified ending location), the computation needed to first prepare the input image for analysis (such as applying line-thinning techniques, etc.) and then to successfully locate "core" and "delta" points and finally count ridge lines is massive.

d. A final image content query is executed on the result set of step c, to narrow down the result set to a size easily manageable by human analysis. This query performs the most detailed level of content matching by examining what are called "minutiae points". In order to determine which prints match the query, minutiae points are extracted from the input image, and are compared against pre-extracted points from the library images.

A minutiae point (as used in this context) is a location in the fingerprint where a ridge abruptly ends, or where it bifurcates into two ridges. A primary ridge direction is calculated and associated with each extracted minutiae point. The input image is then registered with a library image (in the result set from step three) and the two images are aligned as closely as possible (there often is no exact alignment, because two different prints from the

same finger may vary a lot, depending on how the print was made). The registration is done by finding transformation, rotation and scaling functions which bring the maximum number of minutiae points from both images into alignment with each other. The closeness of match between the input image and a library image is determined by the number of minutiae points which are closely aligned (within a specified tolerance).

From the description of the fourth query, it is easy to see that it is by far the most computationally intensive of all, and that it must be applied only against a relatively small number of library images. For this reason, the hierarchical structure of the query, and the text-based nature of the first two queries are both vital to the success of the system. The concept of using a hierarchy of progressively more complex queries in a high volume environment has wide application in image processing and other areas of computer science.

5. Recognition of objects using vectorized shape descriptors [Mehrotra95]

As in the case of the paper we have just discussed, the primary emphasis of [Mehrotra95] is on the efficient

retrieval of images from a library which successfully match with an input image, but a completely different matching mechanism is implemented.
Shapes are stored in a library as a set of high-dimensional vectors, and an index is built and used for fast retrieval based on similarity to an input shape. This method has been implemented in a research program named FIBSSR (Feature Index-Based Similar-Shape Retrieval), which was constructed only for research purposes; it is not used in any commercial application.

FIBSSR was designed to test a retrieval method that attempts to address three of the primary issues in image recognition which we have been discussing: how to properly represent shapes in a recognition model, how to measure the degree of similarity between the input image and the library image, and how to obtain and retrieve query results efficiently.

In this system, shape representation is accomplished by first performing polygonalization (in a manner similar to that which we have already discussed). The vertices of the resulting polygon are called "interest points". A "boundary feature" is defined to be a collection of a few "interest points". A feature can be represented by a set of two-

dimensional "interest point" coordinates (recognition is restricted to two-dimensional images in this project). A rigid shape is then defined as a collection of features. Articulated shapes (shapes with moving parts) are represented as a set of rigid components as described above, and a set of articulation points which are shared among the rigid components.

A feature encoding scheme is used which adjusts for scale, translation, and rotation, as follows: Each adjacent pair of calculated feature points are selected in turn to define a "basis vector" for the entire feature. The basis vector is normalized, and the lengths of all other vectors which are part of the feature (defined by pairs of feature points) are adjusted to match the basis vector. A coordinate system is then selected so that the origin is at one end point of the basis vector, and the point (1, 0) is at the other end point. All feature point coordinates are then adjusted to this coordinate system.

The resulting encoded feature is part of a set of features which describe the entire shape. If we let F_1, F_2, ..., F_n be the set of features which describe a particular shape in a library, then we can think of the individual values in F_1, F_2, ..., F_n as coordinates in a high-dimensional vector.

If we let I_1, I_2, ..., I_n be the set of features which describe an input image which is being used to query the library, we can use the Euclidean distance between the vector (F_1, F_2, ..., F_n) and (I_1, I_2, ..., I_n) as a measure of the similarity between the query shape and the library shape. If we build an index of feature vectors for each shape in the library, and use the extracted feature vector from the input shape as a key to access the index, we then have an image retrieval mechanism based on comparison of shapes. This is a rough description of what takes place in the FIBSSR system during the execution of a query.

A similarity tolerance amount is specified in the query, to determine how close to the input the retrieved shapes should be. When the result of a query contains a large number of similar shapes, the system attempts to take each retrieved shape, align it with the input image, and calculate the closeness of fit of the "interest points", in order to further narrow the result set.

6. Statistical recognition with shape models [Cootes95]

Up until this point, we have discussed recognition systems which are based on matching against a library of fixed models. [Cootes95] presents a different approach to the

structure of the image library, through the use of "deformable models". The basic idea is that a single model can be matched successfully against shapes which vary somewhat from the shape described by the model if there is a carefully constrained method for "deforming" the model to make it match the input shape. The deformation must be constrained so that the model is not changed to the point of no longer resembling its intended shape. An example of this, taken from [Cootes95] is the recognition of electrical resistors mounted on a printed circuit board. Although all of the resistors on a particular board may have the same basic shape, each individual resistor is likely to differ slightly from the others, because of the manufacturing method used. It is desirable in a computer vision system which recognizes resistors to store a single model which can be matched successfully, in spite of the small individual differences among actual resistors. The research in [Cootes95] indicates how to build a model of the typical shape of a resistor, and to add the permissible variability from that shape which may occur, as a part of the model. The method can be summarized as follows:

a.) Starting with a representative shape (the outline of a typical resistor, in the case of our example), identify points of interest (similar to the critical or dominant

points we have discussed) of the following three types: i) Points which indicate parts of the shape that are significant in terms of the shape's function (from our example, the wires coming out of the resistor are a functional part of the resistor's shape, so points which outline the shape of the wires would fall in this category). ii) Points which mark extremes in curvature, and distinguishing features, even when they do not have functional significance. iii) Additional points which can be interpolated from points of types i and ii, and which help fill in some of the details of the overall shape.

b.) Collect a training set of images of various resistor shapes, and identify the points of interest described in a) for each one. The manual labeling of points of interest occurs only during the training phase, while the model is being built.

c.) Align each member image of the training set with the representative shape described in a). This is done by performing scaling and rotation so that corresponding points of interest are as closely aligned as possible. The closest possible alignment can be calculated by minimizing a weighted sum of squares of distances between corresponding points. The details of this calculation are

in [Cootes95]. The weights for this calculation are chosen so as to give more importance to the points which are the most stable (the points which do not move significantly in their relative location from image to image). The more a particular point tends to vary between images, the lower the weight assigned to it.

d.) Determine the points of interest for a mean shape, where the points are derived from corresponding points on each shape in the training set. This is done by i) aligning each member of the training set with the representative shape determined in a), ii) calculating the mean shape from an average of the points of interest in the training set, iii) normalizing the orientation, scale, and origin of the mean calculated in ii) to a set of default values, such as the representative shape from a), iv) realigning each image in the training set with the normalized mean from iii), v) repeating steps ii - iv until the variance in each newly calculated mean falls below some threshold value (so that the calculated mean converges to a particular set of values).

e.) Use statistical methods (described in detail in the paper) to determine a region of allowable movement for each point of interest in the mean shape as found in d). The

statistical methods take into account the fact that the allowable movements for particular interest points are related to those of neighboring points.

f) Having built the shape model in steps a) - f), image matching can be performed by applying the model shape to an input image, and varying the model's points of interest in ways which conform to the regions of allowable movement determined in e). If the points of interest can be aligned with the edges of the input image using only allowable movements, then there is a successful match.

1.2.2 Neural Network Techniques

To complete our introduction to image processing techniques, we will look at another approach to the problem which involves the use of the artificial neural network (ANN). Distinguishing a shape from its background, or one kind of shape from another, or distinguishing multiple shapes which may partially block the view of each other in an image is quite often a non-trivial task; particularly when it is approached using the sequential algorithms necessary on a uni-processor. The task appears trivial at first glance because of the fact that the human mind and human vision system are so adept at performing very sophisticated image recognition without conscious effort.

The human visual neural network and brain are capable of working on many parts of the image processing problem simultaneously and performing "parallel processing" on an enormous scale, while the computer system, which may have one to a few thousand processors, is much more constrained to work on it's data in a sequential manner. One area of research in computer science and other disciplines for the past few years has been how to apply concepts discovered from analysis of brain and neural system function to improve our data processing methods. One of the results from this research has been the creation of artificial neural networks, which have been applied to a number of applications, including image processing. We will now summarize some of the more important aspects of this field of study by describing the makeup of the typical artificial neuron (which is based on some of the features of the human neuron), discussing some of the ways in which sets of artificial neurons are connected into networks, giving a brief description of training algorithms, and looking at a very simple example of how a neural network can be applied to an image recognition problem.

The typical makeup of the artificial neuron is shown in figure 4 below. The values x_{i1}, x_{i2}, ..., x_{in} represent input signals which may be binary or consist of a range of

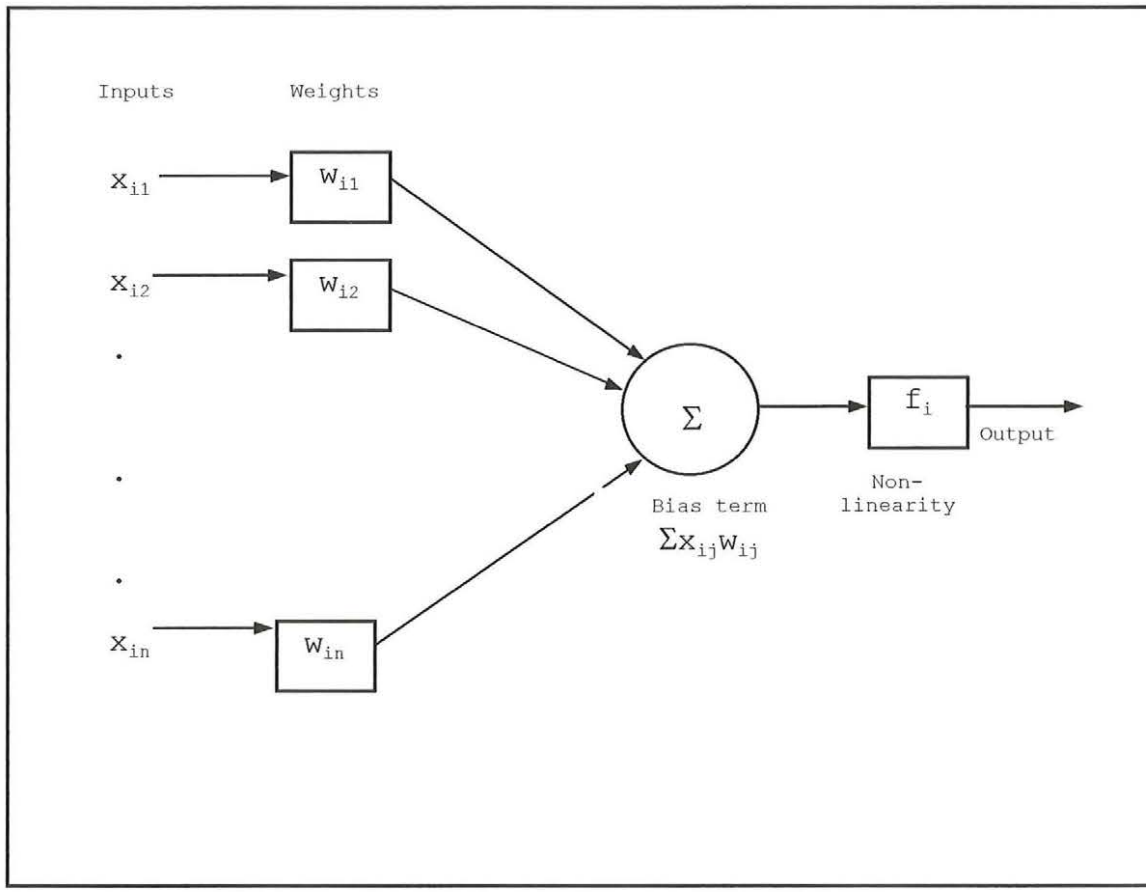


Figure 4: The Artificial Neuron [Kartalopoulos96]

several possible values. There are n inputs to the neuron, and the i subscript indicates that we are looking at the ith neuron out of a network of connected neurons. The boxes which contain the terms w_{i1}, w_{i2}, ..., w_{in} represent weighting factors which determine the strength of the signal which arrives at the neuron. Each input signal is multiplied by the corresponding weighting factor, and the result is sent to the neuron. The circle represents the neuron, where all of the weighted input signals are processed. The sum of all of the signals is determined, and

compared to a threshold value set for the individual neuron. If the result of the summation is greater than or equal to the threshold, the neuron "fires", or sends an output signal; otherwise, no signal is sent. The box labeled f_i represents a non-linearity function which limits the magnitude of possible output values to a specified range. There may be one or more output connections to other neurons or to an output circuit.

In order to process information, neurons are connected together in a number of different possible combinations, and the output of one neuron may become an input for one or more other neurons. We will limit ourselves here to the description of one type of network, as shown below in figure 5.

The multilayer feedforward network receives external inputs at the input layer. Each input arrow in the diagram may represent a single input signal, or a set of input signals. The neurons influence the processing of neighboring neurons to which their output is connected in a forward direction only (in the case of the feedforward network). The middle layer is called the hidden layer, because the outputs are not visible external to the network. The outputs from the entire network are handled exclusively by the output layer.

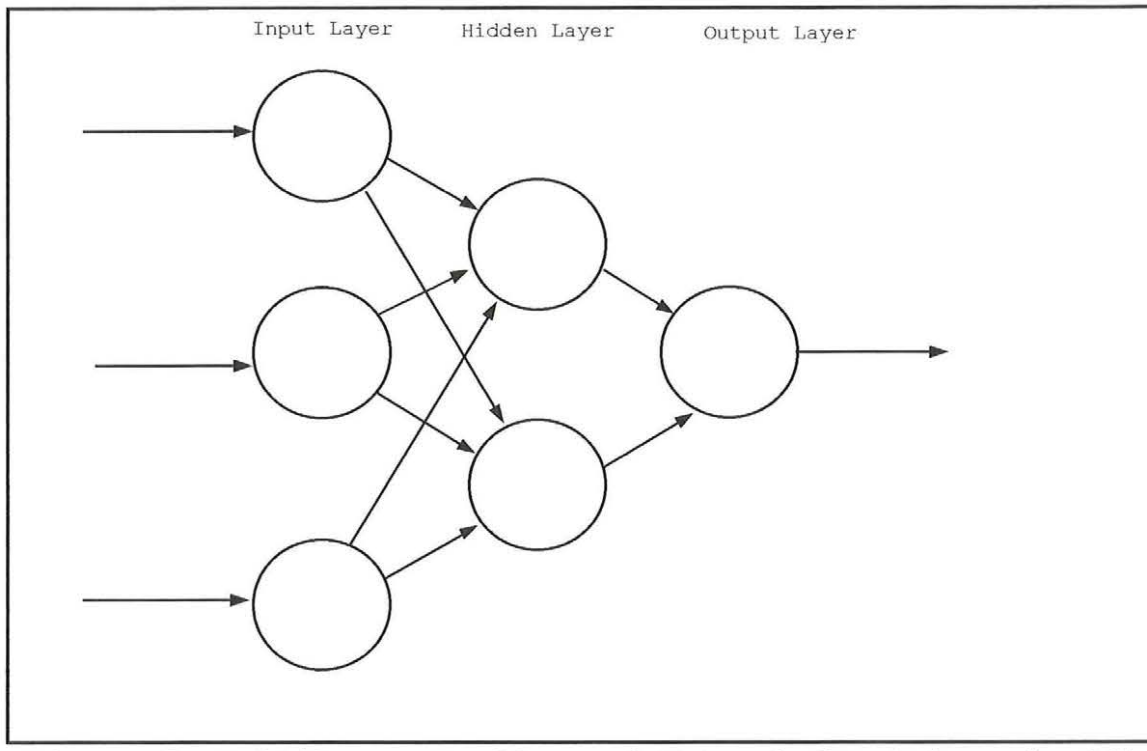


Figure 5: Multilayer Feedforward Network [Kartalopoulos96]

In other configurations, the output from an intermediate layer may be sent back as input to a prior layer, so that there is a feedback relationship.

One important part of the network which is not shown in figure 5 is a training circuit. The "program" which is executed by the network consists of the set of weights and thresholds assigned to each neural node. At the time the network is built, this program must be "written", by repetitively running sample sets of input data through the network, determining if the desired output has been achieved, and correcting the weights and thresholds so as

to obtain correct outputs. A typical way of doing this is to temporarily attach a training circuit to the network, which receives the output and compares it against expected output for a given data sample. When the output is not correct, an error signal is sent back to the neural nodes, which causes adjustments to be made to the input weighting factors and threshold values [Kartalopoulos96].

With this information in mind, we will now look at a very simple example of how a feedforward neural network can be applied to an image processing problem.

Suppose we have a conveyor belt in a factory which carries four different manufactured components, and that a robot arm is positioned over the belt, which selects two specific kinds of the components, picks them up, and drops them into shipping packages which are passing on another lower conveyor belt. The robot is equipped with a television camera mounted in the arm, and simple image recognition circuitry which can find the edges of the shapes, count the number of pixels enclosed by edges (in order to determine surface area), and determine that an object is either black or white, depending on the grayscale color intensity detected by the camera. Suppose also that the shape and coloring of the objects (as seen by the rather primitive

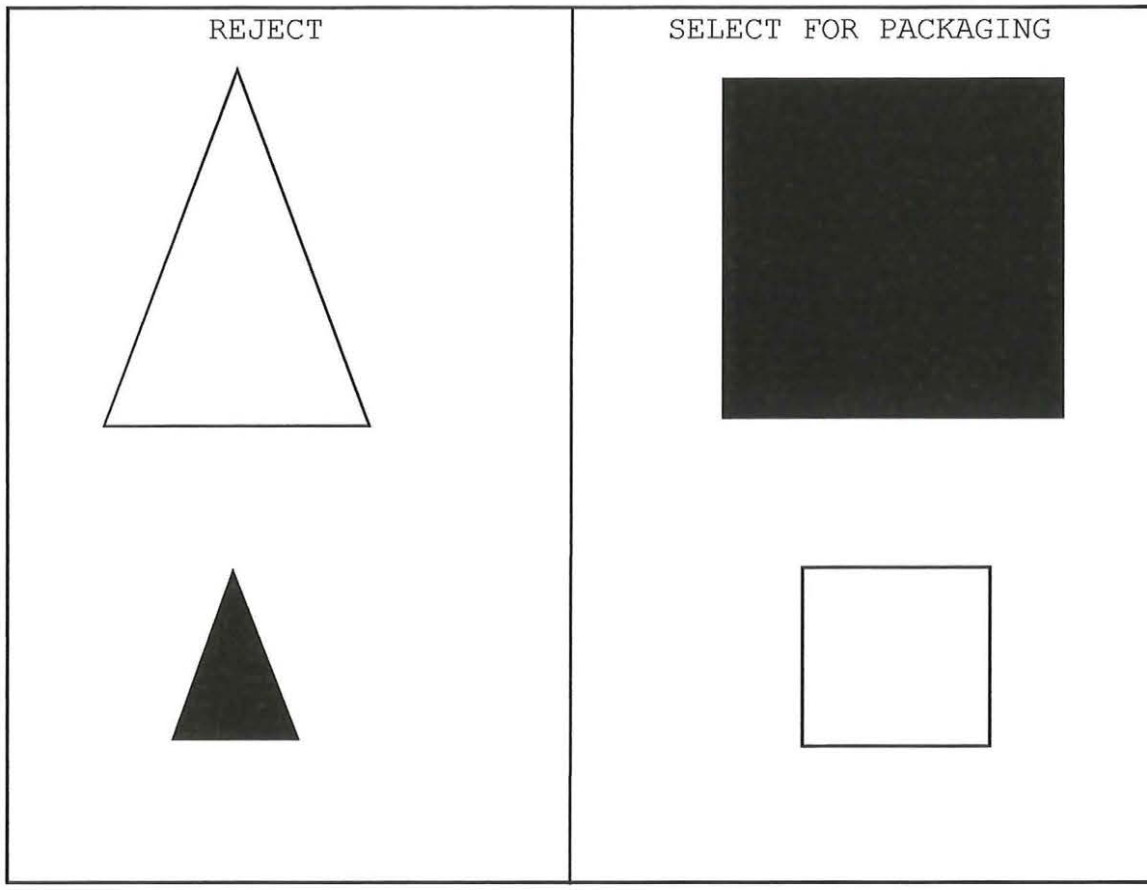


Figure 6: Component Shapes/Colors On A Conveyor Belt

camera) is as shown in figure 6 above. The recognition circuitry is able to classify the objects as either large or small (based on surface area), and black or white (based on grayscale intensity). We will see how a simple feedforward neural network can be given a set of weights and thresholds which will allow it to distinguish (for example) large black objects and small white objects successfully.

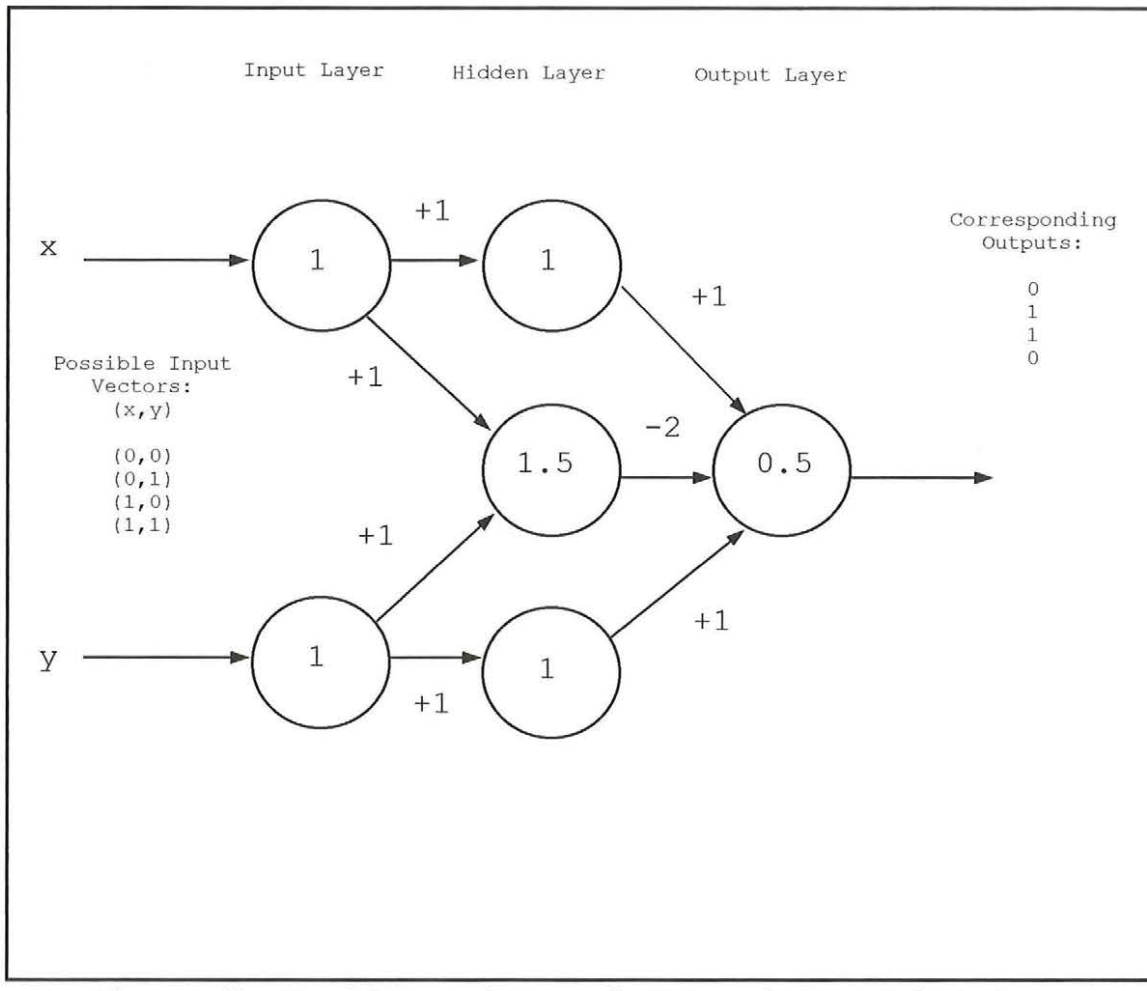


Figure 7: Feedforward Neural Network For Robot Arm, Adapted From [Kartalopoulos96]

Our first step will be to assign two-dimensional vectors to each shape seen through the camera, based on the output of the recognition circuitry, as follows:

x = 0 if the object is large, or 1 if the object is small.

y = 0 if the object is white, or 1 if the object is black.

These values will serve as input to our neural network shown in figure 7. The values shown inside the circles (neurons) are the threshold values; when the sum of the

inputs meets or exceeds the threshold, the neuron sends an output signal with magnitude shown by the numbers next to the arrows connecting the neurons. The output of the network will be a one if the robot arm should select the object, and a zero if it should reject an object.

One of the first things which comes to mind when looking at this example is that a neural network isn't needed to perform pattern matching for such a trivial application. It would be much easier to add a small additional circuit to the recognition hardware to make the same selection. While this is certainly true, the method shown here can easily be extended to much more complicated pattern matching and non-trivial networks. Suppose, for example, that instead of using a two dimensional "feature vector" containing size and color, we decide on a vector containing multi-dimensional coordinates for the points of interest (or critical points) that describe an object with a complex shape, such as those we discussed earlier in [Mehrotra95] (the FIBSSR program). Our method of matching might involve finding the Euclidean distance from an n-dimensional input vector to a set of allowable shape vectors. The dimension of the vectors involved could be as large as several hundred entries. In this kind of situation, the power of parallel processing and the neural network architecture may

offer a solution with much better performance than that which can be obtained by using a sequential algorithm.

[Gonzales92] offers another example which uses a non-trivial feedforward network to recognize the shapes of four different aircraft silhouettes in 48 different scales, including cases where the silhouettes are distorted by noise. The same basic principles from our trivial example also apply in this example. See [Gonzales92], pages 611-616 for more details.

1.2.3 Applications

Automated image recognition is used many different ways, some of which we have already seen. The rapidly growing demand for improved capabilities and new applications will provide challenges for the researcher well into the future.

In this final introductory section we will take a brief look at some current applications and challenges for future development.

Dr. Karl Tombre has described some of the more recently developed applications for document image analysis, along with suggestions for their improvement in a paper submitted

to the International Association For Pattern Recognition (IAPR). Some of the applications include programs to analyze mechanical engineering drawings and convert them from paper to computerized files in CAD format (as we have already seen in this paper), programs to analyze and understand architectural drawings, and programs to convert cartographic information from paper maps to computerized input for Geographical Information Systems (GIS) [Tombre95]. Many of the improvements needed in the existing programs have to do with the level of automated understanding of the document as a whole. An example of this with regard to the conversion of engineering drawings is the need for automated conversion from two-dimensional renderings on paper to computerized, three-dimensional projections. It must be possible to construct the projections from multiple two-dimensional views of the same object. Another example in the area of Geographic Information Systems, is the need to have automated matching of cartographic information from multiple sources into a single integrated and computerized source. For example, elevation information taken from paper maps should be matched and merged with aerial and/or satellite photographs of the same geographic area.

In [O'Gorman95], Lawrence O'Gorman makes some interesting predictions about future trends in image recognition: 1) As graphic libraries which contain symbols for specific applications such as electronic schematics, architectural structures, mechanical parts, etc. become more complete and comprehensive over many subject areas, recognition systems may begin to recognize drawings across multiple domains by matching on the kinds of individual symbols which are present in an image (similar to the way in which recognition of printed text in multiple fonts can be done now). 2) Improved noise reduction techniques, and faster matching and indexing methods will be developed, as well as improved machine/human interfaces, to make the recognition process easier and less time consuming for the user. 3) Construction of three dimensional projections from two-dimensional drawings will improve.

We will close this section with a final example application which supports content-based retrieval of still images and motion video clips from a multimedia database, which has been very recently developed by IBM: the QBIC (Query By Image Content) System [Flickner95].

In the QBIC system, queries can be based on similarity to example images, similarity to user-drawn sketches, selected

color and texture patterns, camera motion (such as a specific panning motion made with a camera while recording video images), object motion (the recorded movement of objects within video images), and other graphical information.

The QBIC system is divided into two main components: database population, and database query. Database population consists of loading an image into the database, and extracting features from the image which can be used for query comparisons. The features which are extracted include colors, textures, shapes, camera motion, and object motion. The queries which are executed against this extracted data are built in a graphical user environment.

The data model used by QBIC for query matching handles both still images, and video clips. Still images are analyzed to distinguish between objects in the image and the surrounding "scene". This makes it possible to query on individual objects within images. Video clips are broken into series of contiguous still image sequences called "shots". The boundaries of shot sequences are determined by abrupt changes in scene, changes in camera operation, changes in the objects which appear in the images (such as the appearance or disappearance of objects), and selection

of shot boundaries by the user. From these shots, representative image frames (or r-frames) are selected to represent the entire sequence of frames in the "shot". In some cases, a composite r-frame is built by creating a mosaic of all frames in the shot in such a way that image motion can be represented.

To construct the data model from the input images, QBIC combines automatic image recognition with a set of semi-automated tools which are used manually to help identify backgrounds and objects. A typical manual operation might be to click the mouse (once) when the cursor is over a background point, or when it is over an object point. The tool would then automatically select pixels in the image which are within a certain range of color and intensity around the selected pixel. In this way, entire objects or scenes can be defined with a minimal number of mouse operations.

The matching process, which compares query data with the data model, varies depending on the type of query being executed. Queries which are looking for certain percentages of a given color in an image may send a set of three-dimensional color coordinates as input, which are matched with a histogram of the color values found in a given image

in the library. Queries which are looking for similarity of objects in an image to a sketched object will reduce the input sketch to an edge map, and then compare it with extracted objects in the library using template-matching techniques. These are just two examples of a set of matching methods which are used by the system.
A demonstration version of the QBIC system is available on the Internet at http://wwwqbic.almaden.ibm.com/.

CHAPTER 2

THE DESIGN OF THE RECOGNITION PROGRAM

We will now turn our attention from a general survey of image recognition to the specific design of the program developed for this project, called TOKSCAN (TOken-based SChematic ANalysis).

TOKSCAN analyzes images of digital logic schematics, and generates logic equations which describe the circuits. It implements a simple method for representing the circuit components: a set of token lists which describe basic shapes, that are linked together through a relationship table to describe the more complex shapes of the circuit components.

Once the logic components in a schematic have been identified, they are isolated from their connecting signal lines, and connection analysis is then performed. During the analysis, electrical connections between the connector lines are located, and the connector lines are flagged at these locations so that all electrical paths are followed properly.

TOKSCAN is an experimental program which was written for research purposes to demonstrate the feasibility of token-based electrical circuit analysis in image recognition. It currently recognizes a limited set of schematic symbols: the AND, OR, NOT, NOR, NAND, and XOR gates, and circular "connectors" which represent the electrical connections between signal lines. Higher level components such as flip-flops, memory units, and other large scale circuits which are typically represented by rectangular blocks with input/output lines are not currently included in the project. Also, text recognition and automatic recognition of sequential circuits with feedback are not currently included in the project.

Although TOKSCAN works on a limited set of symbols and configurations, it strives for a complete understanding of the circuit being analyzed. The descriptive equation generated by the analysis is not many steps removed from the ability to label a circuit as "four-to-one multiplexer", or "full adder", for example. It also would not be difficult to add graphical tools which would allow the user to flag sub-circuits within more complex circuits for analysis.

The following sections in this chapter present the design details of TOKSCAN. Chapter three provides some additional notes on the implementation tools used, and how they helped to shorten the total development time. Chapter four provides samples of the analysis performed by TOKSCAN on several different schematics.

Figures 8 and 9 below provide a description of the processing flow in TOKSCAN. The program uses grayscale image scans of schematic circuits in Windows BMP file format as input, and it generates the resulting output equations at the bottom of the displayed bitmap. Other visual aids which track the analysis process are also generated, such as the highlighting and labeling of recognized components in the displayed schematic, the labeling of signal line connection points with large red circles, and the shading of signal lines in blue as they are followed during connection analysis. Although image pre-processing is not the focus of this project, a pre-processing step is included in TOKSCAN because binarization and line thinning are necessary for successful analysis. Immediately following figures 8 and 9 are comments on each processing step.

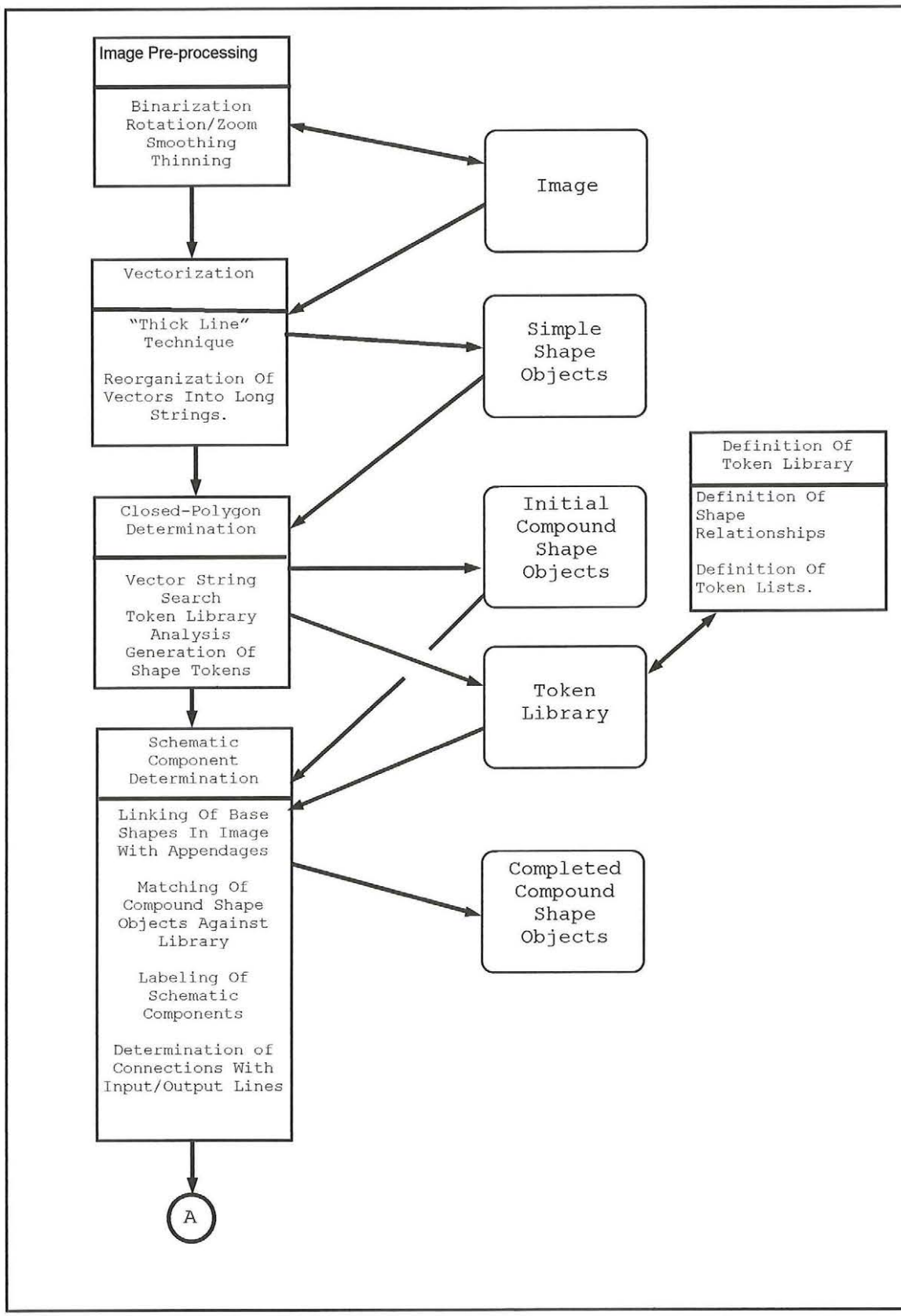


Figure 8: Image Analysis Processing Flow #1

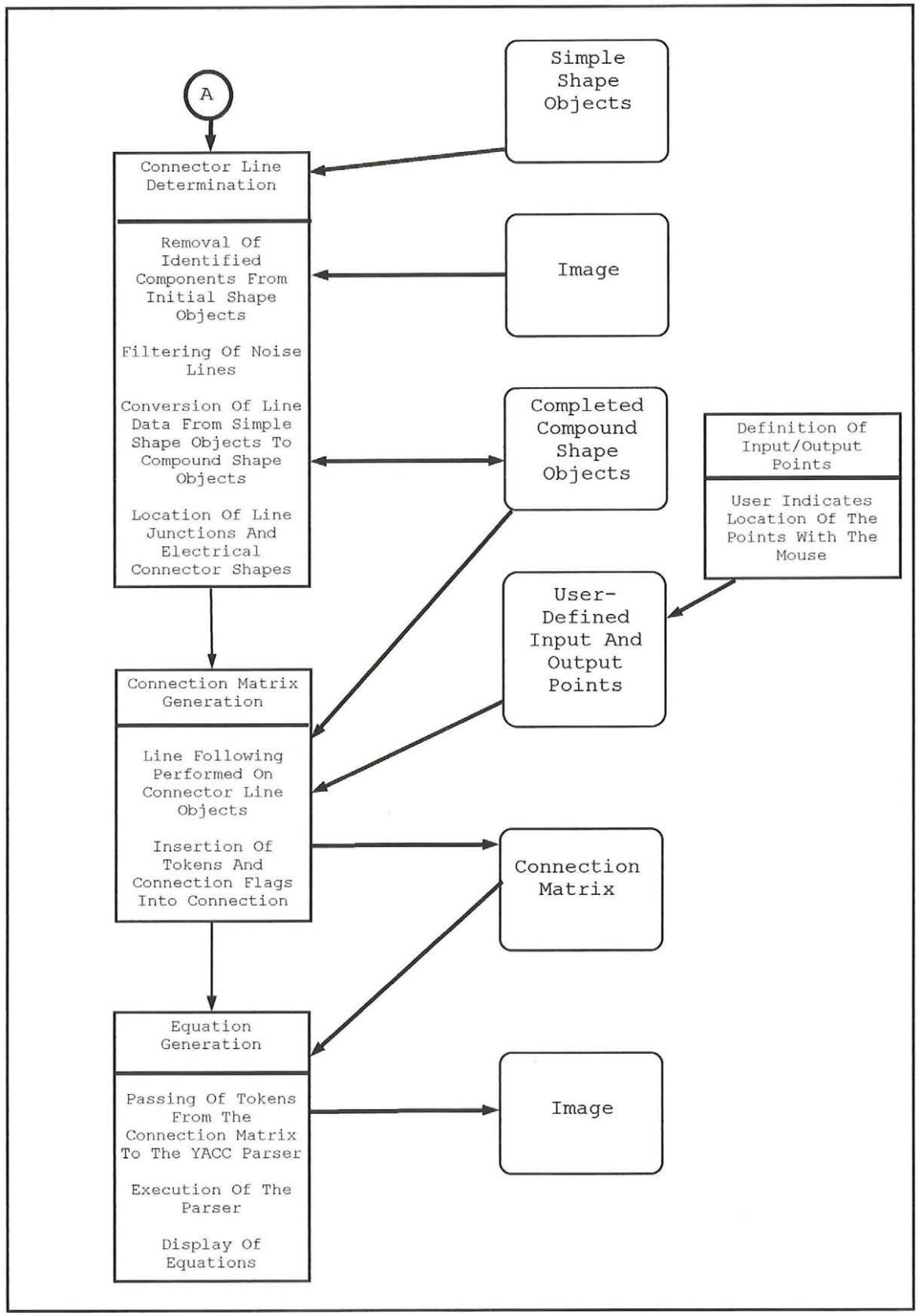


Figure 9: Image Analysis Processing Flow #2

Comments On The Processing Steps

1. Image Pre-Processing

Image pre-processing functions are provided and may be selected and applied at the option of the user. Thinning and binarization must be performed on raw grayscale images for proper vectorization and recognition. The binarization function works with a global grayscale threshold value (local or adaptive thresholding is not available currently in this program). Zoom and rotation functions are provided to assist the user with image alignment and scaling. A simple smoothing function has also been provided for optional use, which performs a raster scan of the input image, and uses a "window" of local surrounding data to determine how to set the color value of each "pixel of interest". The average grayscale color value of the entire "window" of data is compared with a user-set threshold value, and if the average value is greater than (whiter) the threshold, the "pixel of interest" is set to pure white (value 255). If the average value is less than (blacker) or equal to the threshold, the "pixel of interest is set to pure black (0).

2. Vectorization

TOKSCAN performs vectorization using the "thick line" technique which is described in the text accompanying figures 2 and 3 in the introduction. A reorganization process is also executed as a part of vectorization, which reorients sets of neighboring vector chains so that they all have a matching direction (as much as possible). Simple "shape objects" are created (which are actually program objects implemented with Visual C++) that contain a single vector chain in each object. The final output of this step is a collection of these "shape objects".

3. Closed Polygon Determination

In this step, the vector chains are followed by the program in order to detect closed, minimal polygons. These polygons form the basis for schematic component identification. A "skeleton" version of a "compound shape object" is created (another C++ object) for each detected minimal polygon, which contains (among other things) the vectors that represent the polygon. When the skeleton compound shape objects have been fully filled out with data from later analysis, there is sufficient information present to identify a schematic component, such as an AND gate or an

OR gate. Tokens which describe the shape formed by each detected polygon are generated, and embedded in the associated compound shape object for later analysis.

4. Schematic Component Determination

In this step, the polygons detected in step 3 which are close neighbors, and which have an acceptable shape description, are linked together to form compound shapes, with enough complexity to be identifiable as schematic components. Connecting input and output signal lines are also taken into account, and are used to aid in the identification process. The output of this step is a set of compound shape objects which have been labeled with schematic component names. A corresponding visual labeling takes place at the same time in the bitmap displayed for the user. An important part of this step is the comparison of linked compound shapes with a shape token library of pre-defined token lists, to aid in identification and labeling. Much more information on this step is provided in following sections.

5. Connector Line Determination

Two different kinds of objects were created in earlier steps: simple shape objects containing vector chains, and compound shape objects which represent minimal closed polygons and recognized schematic components. In this step, all minimal closed polygons which could not be recognized as a schematic component are removed from the collection of compound shape objects, and the vector chains which describe the recognized schematic components are removed from the collection of simple shape objects. After this removal has taken place, the remaining simple shape objects describe either signal lines which connect schematic components, or noise in the image. One of the functions performed in this step is the removal of very short noise vectors (with a length that falls below a user-specified maximum). After the filtering has been completed, the remaining simple shape objects are converted to compound shape objects with labels that indicate connecting signal lines. A final process which takes place in this step is the location of circular signal line connectors that indicate an electrical connection between signal lines. This is accomplished by examining a circular region around the end points of each of the connector lines. Each connector line at this point is represented by a properly

labeled compound shape object which contains vector coordinates for the line end points in the image. The color value for each pixel in the circular region (with a user-specified radius) is examined, and if all returned values are ON (or black), then the location is flagged as an electrical connection. The circular search is flexible enough to allow for distortion in the thinned image, because multiple circular searches are actually performed, each one with a center point that falls within a user-specified radius around each line end point. Another important point is that TOKSCAN has extracted data from the thinned image up until this point, but for identification of circular connectors, it is necessary to go back to the original bitmap, using the vector coordinates of line end points found in the thinned image (because the circular connectors have been removed by the thinning process).

The location of each circular connector is flagged in a vector coordinate record which is a member variable of a "LINE" type compound shape object. In the "Connection Matrix Generation" step (see below) we describe how this information is used in connection analysis for the circuit.

6. Connection Matrix Generation

Once the schematic components, signal lines, and circular connectors have been identified, the next step is to perform connection analysis on the components, by following the signal lines from their source through any intermediate components to their final output destination. In order to properly identify the starting and ending points for this analysis, TOKSCAN accepts mouse input from the user which identifies signal line end points where input signals are applied to the entire circuit, and where output signals are sent by the circuit. Only the initial inputs and final outputs are identified manually. Then, at user request, TOKSCAN starts at the identified starting points, and traces signal lines throughout the schematic, by following the vector chains contained in compound shape objects which are labeled as signal lines. At locations where circular connectors were found, recursive analysis is performed, to take into account every signal path emanating from an electrical junction. As the line-following process takes place, a "connection matrix" is built, which describes the entire circuit, complete with all connections, in a form which can be used to generate tokens that describe an equation for the circuit. The completed connection matrix is the final output from this step.

7. Equation Generation

After the connection matrix has been completed, the final step in the process is to send a series of tokens which describe the circuit to a YACC-generated parser, which translates the token string into a logic equation. The resulting equation is displayed on the circuit bitmap for the user. In order to generate the necessary token string, a process in this step starts at the output side of the circuit, and works backwards toward all inputs, retrieving token information from the connection matrix recursively. The tokens sent to the parser are character string labels which identify schematic components (AND GATE, OR GATE, etc.), and the order in which they are sent to the parser implicitly indicates the connections between them. Much more information on all parts of this step are found in the following sections.

We will now discuss each of these steps in more detail, from the point of view of program design and implementation.

2.1 "Thick Line" Vectorization And Corner Detection

In section 1.2.1.3 of this paper, we discussed the exact vectorization technique used by TOKSCAN in detail. See that section, and figures 2 and 3.

2.2 Detection Of Curvature

The vectorization technique discussed in section 1.2.1.3 of this paper includes measurement of curvature as an integral part of the technique. The method described there is precisely what is used in TOKSCAN. See that section, and figures 2 and 3.

2.3 Representation Of Digital Schematics As A Collection Of Related Shape Objects

From an implementation point of view, the goal of TOKSCAN is to represent every shape in a digital schematic image as a "compound shape object" which contains not only the vector chain that describes the shape, but enough additional information to positively identify each component, and relate it to the surrounding components. In TOKSCAN, there are two primary object types which are used for representation of the image. The first is called

CImageObject (see Appendix A, Code Fragment 1 for the C++ definition), which contains the shape vectors generated during the vectorization process. The second is called CCompoundImageObject (see Appendix A, Code Fragment 2 for the C++ definition), which contains sufficient information for recognition, and which is derived from CImageObject. The CCompoundImageObject class is used to represent each of the schematic components (AND, OR, NOT, XOR, NOR, and NAND gates), and the connecting signal lines. The various component types are distinguished by a member variable in the class which labels the component. The representation of the circular connector (which indicates an electrical connection between signal lines) is embedded in CCompoundImageObject as a flag in the vector coordinate member variable. It specifies the location of the circular connector in terms of a vector coordinate. The entire schematic image is represented by the CCInputBitmapDoc class (which is derived from the Microsoft Foundation Class "CDocument"). The signal input locations and signal output locations for the entire circuit which are specified by the user, are represented by an array of vector coordinates stored in a member variable of the CCInputBitmapDoc class.

Figure 10 below shows how the various components in a typical schematic image are represented by the three

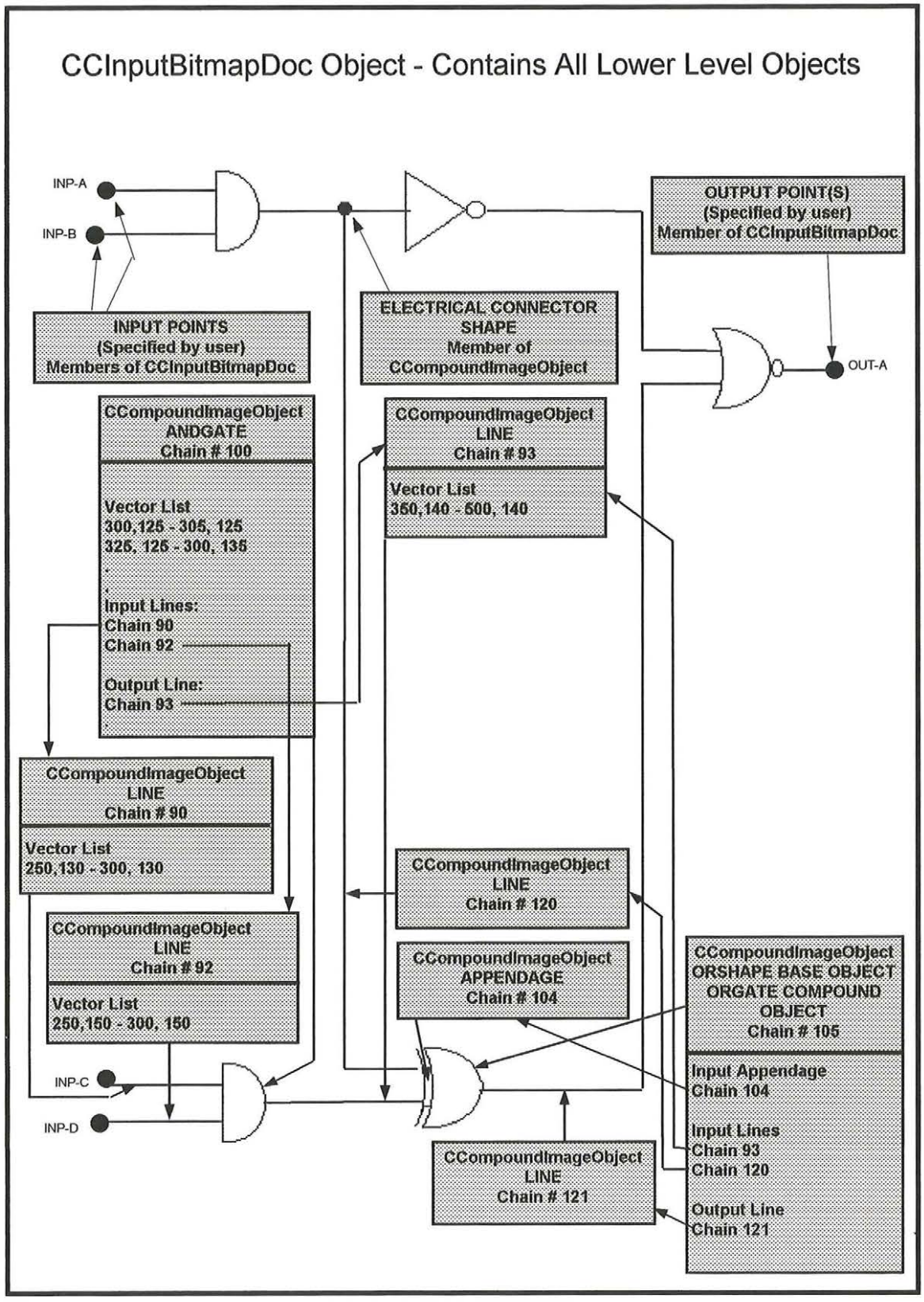


Figure 10: Representation Of The Entire Schematic By Shape Objects

objects we have just described. Each compound shape object is assigned a unique identifying number, called a chain number. Chain numbers of related objects are embedded in each object to link groups of objects together.

2.4 Representation Of Schematic Components As A Set Of Related Simple Shapes

Figure 11 below shows how the more complex shape which represents each schematic component is broken down into a set of simpler shapes, which can easily be described by a set of simple tokens. The dialog box which is part of Figure 11 shows the user's view of the relationship table in TOKSCAN which links descriptions of simple shapes together into a compound description of a schematic component.

Each schematic component (AND, OR, XOR, NOT, NOR, or NAND gate) can be decomposed into one or two input signal lines, an optional input appendage shape (used by the XOR gate only), one of three possible base shapes (NOT, AND, and OR), and optional output appendage shape (used by the NOT, NOR, and NAND gates only), and one output signal line. The dialog box in the figure shows how these items are linked together to describe a logic gate. For example, the ANDGATE compound shape consists of two input lines, no

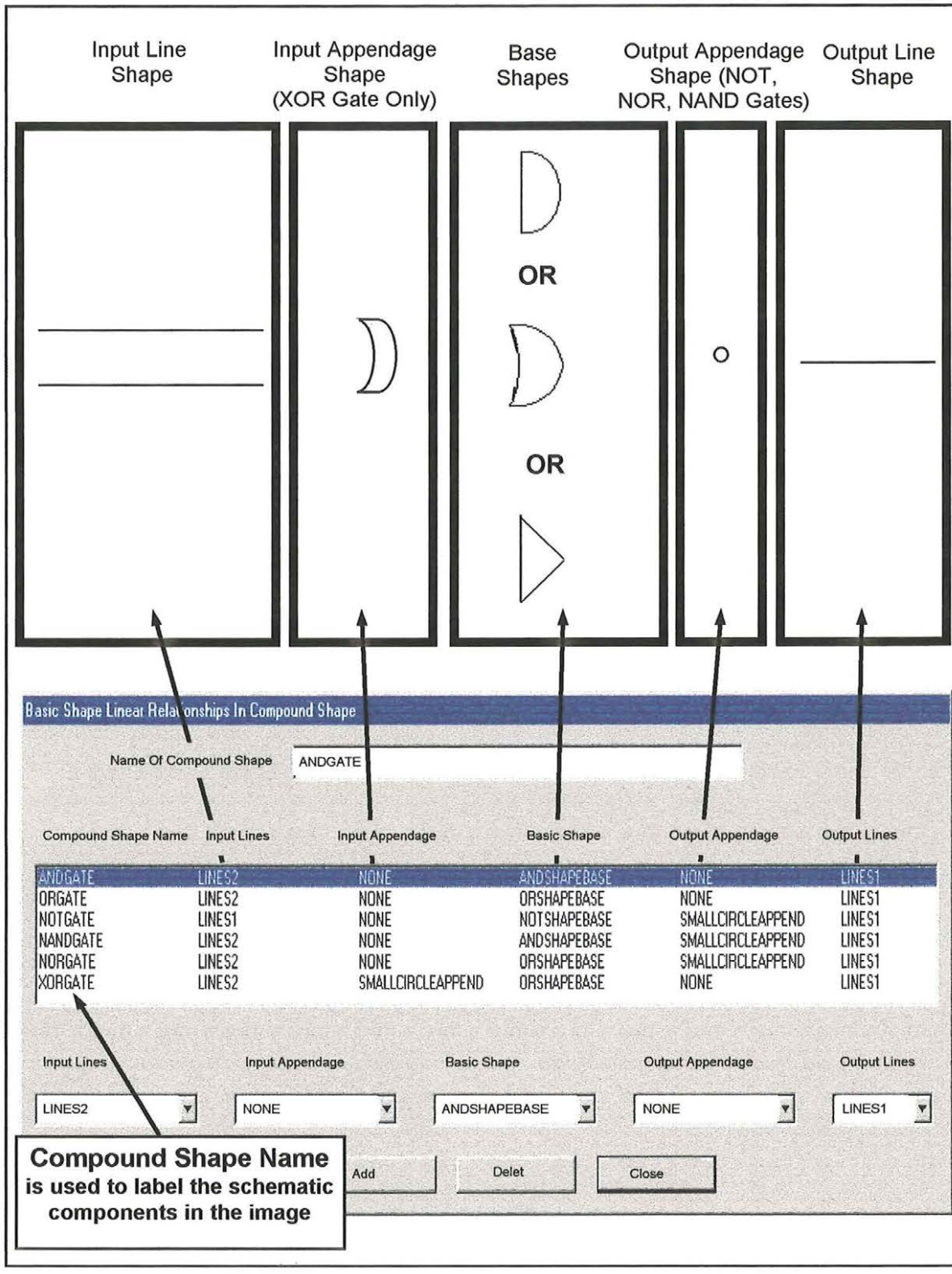


Figure 11: Breakdown Of Schematic Components Into Simple Shapes

input appendage, the AND base shape, no output appendage, and one output signal line.

2.5 Representation Of Simple Shapes As Token Lists

In figure 11 we saw how schematic components are decomposed into simple shapes, and how those shapes are assigned a relationship to each other. Figure 12 illustrates how a list of tokens with possible values "RIGHTCURVE", "LEFTCURVE", "STRAIGHTLINE", "SMALLPERIM_RIGHTCURVE", "SMALLPERIM_LEFTCURVE", "SMALLPERIM_STRAIGHTLINE", and "CORNER" can be assigned to a simple shape. The AND gate at the bottom of the figure is assigned the token list: "LEFTCURVE CORNER STRAIGHTLINE CORNER LEFTCURVE CORNER" as follows: 1) Choose an arbitrary starting point at a corner on the perimeter of the simple shape (In the example, the point where the output signal line intersects the base shape is chosen). 2) Proceeding in a counter-clockwise direction, traverse the arc between the starting corner and the next corner, and determine if it is a straight line or a left or right curve. (The text which explains figure 13 below gives precise details on how this determination is made). 3) Assign an appropriate token from the above list to describe the arc. 4) Insert the "CORNER" token to iindicate a corner.

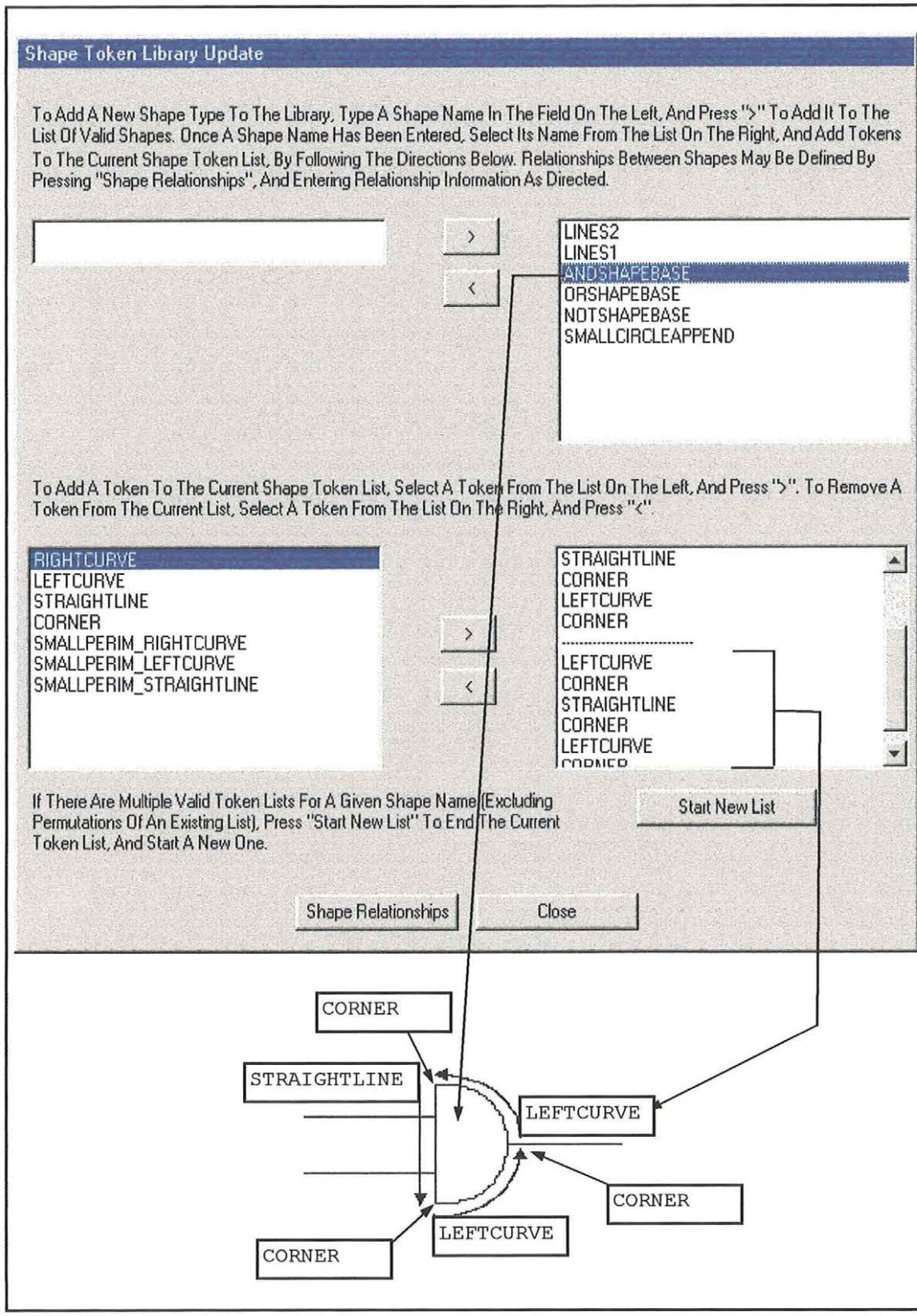


Figure 12: Relationship Between Token Lists And Schematic Components

5) Repeat steps two through four until we arrive back at the starting point.

An arbitrary starting corner can be used because TOKSCAN checks all permutations of each token list (which maintain the same relative ordering among entries) when matching against a library of known tokens. Token lists are always generated using a counter-clockwise traversal of the vector chain which describes the shape, in order to ensure consistency in the token lists.

In the AND gate of figure 12, a corner is shown at the point where the output signal line contacts the base shape. This is shown for illustrative purposes, because many AND gates will have a fairly sharp corner at the intersection point, caused by distortion introduced by the thinning process. In actuality, the particular AND gate in the example would be assigned the token list "LEFTCURVE CORNER STRAIGHTLINE CORNER", where the LEFTCURVE consists of the entire convex arc which contacts the straight line at its two end points. The screen print of the dialog box in figure 12 shows that TOKSCAN handles this situation by allowing multiple token lists to be assigned to the same base shape (Lists are separated by dashed lines in the

dialog box). Note that "STRAIGHTLINE CORNER LEFTCURVE CORNER" is another valid token list which describes the AND base shape. It would successfully match against the token list "LEFTCURVE CORNER STRAIGHTLINE CORNER", since that is a simple permutation of the list (which maintains relative order) in the dialog box.

Input and output appendage shapes are handled somewhat differently as follows: The user is given the capability of choosing a maximum shape perimeter length for appendage shapes which must be smaller than the minimum perimeter length for the smallest base shape. The appendage shapes are assigned tokens in the same fashion as other shapes, with the exception that a "SMALLPERIM_" prefix is added to each token when the total perimeter length of the shape is less than or equal to the user-defined threshold. When TOKSCAN performs recognition, it knows immediately that it is dealing with an appendage when it finds the "SMALLPERIM_" prefix, regardless of the other contents of the token list.

This simple token assignment scheme is quite sufficient for the limited set of shapes recognized by TOKSCAN. For more complex shapes, tokens would still be quite usable; one method of extending their use would be to add appropriate

prefixes and/or suffixes which could indicate position within the drawing, size, position relative to other shapes, etc. The use of tokens as descriptors throughout TOKSCAN was motivated by the desired final output; a descriptive equation. A YACC-generated parser provides an easy method for converting a token list into the desired results.

The determination that an arc is a "STRAIGHTLINE", "LEFTCURVE", or "RIGHTCURVE" is somewhat more complex, as

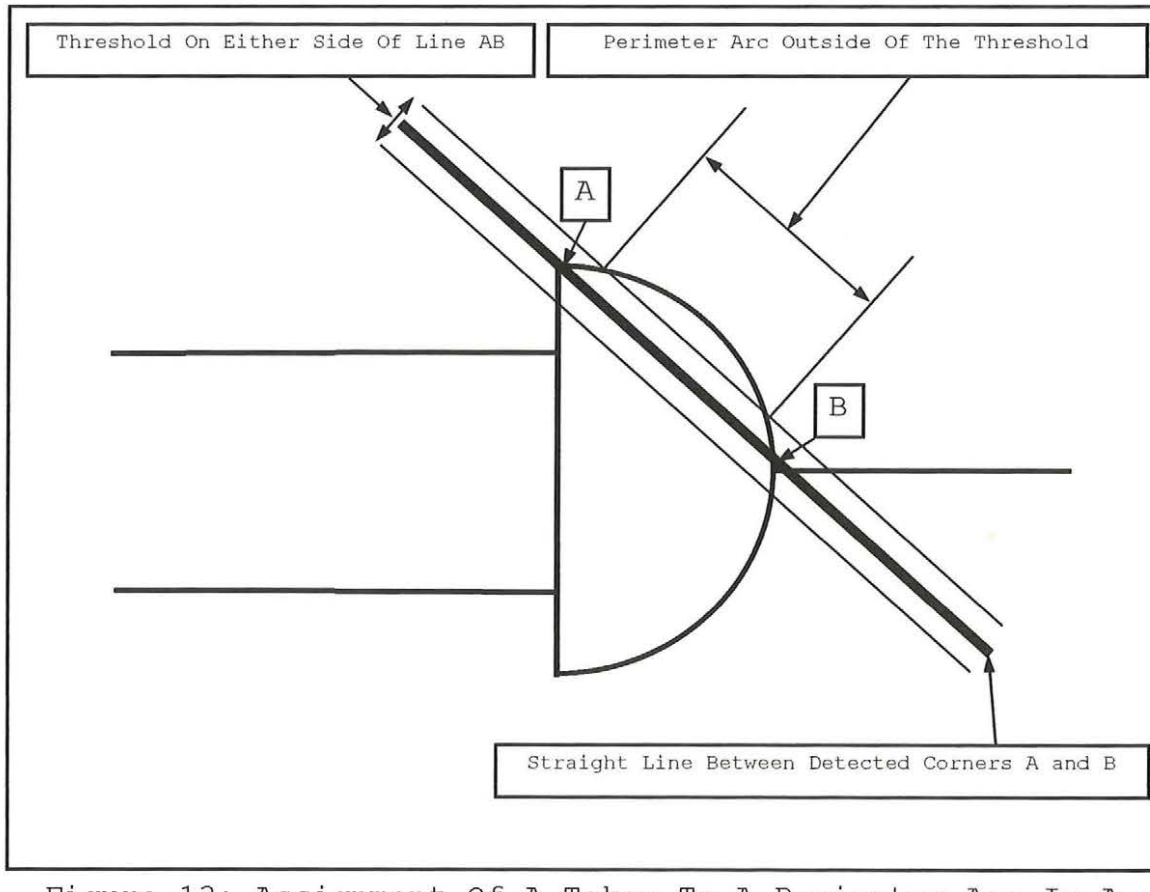


Figure 13: Assignment Of A Token To A Perimeter Arc In A Shape

shown in figure 13. The method chosen must work properly in the face of distortion caused by the thinning process. In particular, two or more vectors with sharp curvature are often introduced into the vectorization at the points where the two input signal lines intersect the base shape, with the result that corners may be detected improperly, and the originally straight line may have significant curvature. This is overcome by using a modified version of the "thick line" technique already discussed. To assign a token to arc BA in figure 13, we imagine a straight line drawn through corner points B and A. Then we "draw" imaginary threshold lines parallel to line BA and on either side of it. The distance from line BA to either threshold line is specified by the user. Next, we traverse arc BA (by following the vector chain), and determine the distance from each vector end point to line BA. We note the vector end points which are on one of the threshold lines, or beyond it, and measure the perimeter length which falls outside of the threshold. We then calculate the ratio of this measured perimeter length to the perimeter length of the entire shape. If the resulting ratio is greater than or equal to a user-specified maximum, then we have detected a curve. If we imagine that we are standing at corner B and looking at corner A, and if the detected curve is to our right, then we have an arc with counterclockwise (or LEFT) curvature,

and the proper token is "LEFTCURVE" (again, assuming a counter-clockwise traversal around the corners of the shape). Similarly, if the detected curve is to our left, then we have clockwise, or RIGHT curvature, and the proper token is "RIGHTCURVE". If all of the perimeter on arc BA falls within the two threshold lines, then we assume no curvature, and assign the "STRAIGHTLINE" token.

There are additional difficulties to overcome when assigning tokens that are also introduced by the thinning process. We mentioned above that points of high curvature may be introduced into the vectorization where the input signal lines intersect with the base shape. Often this curvature is high enough for TOKSCAN to improperly detect one or more corners at these locations. This is overcome by allowing successive vectors with opposite curvature to "cancel each other out" by looking at the net curvature for any two successive vectors. Another method used is to ensure that corners are only detected at a minimum of a user-specified distance apart. This, and a few other small details related to token assignment may be seen by looking at Code Fragment 3 in Appendix A.

2.6 Storage Of Shape Relationships And Token Lists In A Token Library

In figures 11 and 12, we have seen the dialog boxes which serve as the user interface to a library of token lists and relationships which is maintained by TOKSCAN. When the library is being built, TOKSCAN is run in a special "Library Analysis" mode, where it detects minimal closed polygons in input images and generates tokens that describe the polygons. The user is then able to select a detected polygon from the displayed image with the mouse (detected polygons are highlighted in purple in the visual display), retrieve the generated token list, assign a name to the list (such as "ANDSHAPEBASE"), and add it to the library. The user may also add token lists to the library manually by using the dialog box in figure 12.

After the library has been built, image matching and recognition are performed by extracting minimal closed polygons from a new input image, generating the corresponding token lists, analyzing simple shape relationships among the minimal polygons, and comparing the results with the token lists and relationships stored in the library.

2.7 Detection Of Closed Polygons - The Foundation For Schematic Analysis

Detection of the closed minimal polygons in an input image is the foundation for proper segmentation of the schematic into distinct components and connecting signal lines. It is the first major step performed by TOKSCAN after vectorization has been completed.

The output of vectorization is a set of simple shape objects, each of which contains a chain of connected vector coordinates. The polygon detection function follows each vector chain in each simple shape object, and checks the distance from successive vector end points to all prior end points which have already been traversed in the current chain. If the current end point either matches a prior point exactly, or if it is within a user-specified maximum distance from a prior point (as detected by performing a circular search around the current end point), and if certain other criteria (explained below) are met, then a minimal closed polygon has been detected. If the entire vector chain in a simple shape object is traversed without detecting a closed polygon, then a circular search is performed around the final endpoint in the chain to see if there are any neighboring chains which should be used to

continue the search. If neighboring chains are found, the vectors from these chains are also added to the list of traversed vectors, as they are being checked for a junction with a prior vector.

Other criteria which must be met for detection of a minimal closed polygon include the following: 1) a minimum perimeter length around the detected polygon (specified by the user, and intended to filter out extremely small polygons which are present because of noise in the image), and 2) the lack of any other closed polygons which are enclosed by the detected polygon.

This last criteria (no enclosed polygons) is met by making proper directional choices when there are two or more possible "next" vectors close to the current end point (i.e., we have followed a vector chain to a branch point with two or more possible paths). TOKSCAN checks all possible candidates for the "next" vector at any branch point, and uses a selection criteria which favors left curvature. In other words, at a branch point, TOKSCAN chooses the vector which has the greatest left (counter-clockwise) curvature from the direction of the preceding vector, or which has the least right curvature, if none of the possible paths have left curvature. In short, TOKSCAN

continually "turns left" when following vector chains, as much as possible.

When a minimal closed polygon has been detected, a "skeleton" compound shape object is created (as discussed earlier in this chapter) which contains all of the vectors in the "traversed vector" list. This compound shape object is then considered to be a POTENTIAL simple shape in the schematic. After matching has been performed against the token library, compound shape objects which did not match successfully are eliminated, under the assumption that they were formed in error from crossed signal lines or noise in the image.

2.8 Component Identification: The Detection Of Base Shapes, Appendage Shapes, And Input/Output Connections In Schematic Components

As we have said earlier, schematic component recognition is a two part process in which simple shapes are recognized first, and then the compound schematic components are identified from groups of simple shapes. The identification and labeling of simple shapes comes as a result of direct comparison of token lists which describe the shapes against the token library. Simple shapes which are close neighbors are linked together into POTENTIAL compound shapes. When

this linking takes place, information is extracted from the appendage shape objects and connecting input/output signal line objects which are neighbors with a simple base shape, and the information is placed in the base shape object. For example, a NOR gate has one output appendage, two input lines, one output line which is attached to the output appendage, and one OR base shape. There will be shape objects present which represent all of these shapes and which are all close neighbors. Information such as the "APPENDAGE" identifying label is taken from the appendage shape object (and other information from the signal line objects), and is inserted into special member variables in the base shape object. Then the base shape object is compared with the shape relationship table in the the token library, to see if the correct collection of parts has been identified in order to attach a schematic component label.

A successfully labeled schematic component is highlighted in red in the visual image, and a visual text label is also displayed inside the component on the bitmap.

2.9 Isolation Of Schematic Components From Connector Lines

At the beginning of this chapter, we discussed the process where vectors which represent schematic components are

placed into compound shape objects, and are removed from the initial simple shape objects. We mentioned that the remaining vectors in the simple shape objects describe the connecting signal lines, and are transferred to compound objects which are then identified with a LINE label.

Part of the problem of separating signal lines from components also has to do with distinguishing valid component shapes from invalid shapes detected in error because of multiple, crossed signal lines. When there are many signal lines in a schematic, there tends to be a number of rectangular shapes formed from crossed lines. The vectorization process, which tends to smooth out corners to some degree, can easily change a rectangular shape into a straight line attached to a smooth arc. When this happens, it is easy to mistake this closed polygon with the AND base shape (for example). Part of the separation process then, is to distinguish false base shapes from true ones. Three methods are used by TOKSCAN to make the proper distinctions. First, the user is able to specify the proportion of the perimeter of a shape which should be made up of LEFTCURVEs or RIGHTCURVEs. Secondly, it is also possible for the user to specify that multiple left or right curves should have approximately the same length. And thirdly, the user must specify a maximum base shape

perimeter. If a potential shape does not meet all of these specified criteria, it is removed from consideration as a schematic component.

2.10 Determination Of Electrical Connections Between Connector Lines Using Circular Connection Symbols

At the beginning of this chapter, in the description of the processing flow of TOKSCAN, we discussed in detail the process of locating circular line connectors, and flagging their location in the appropriate vector chains. See processing step 5, "Connector Line Determination", for a review of this information.

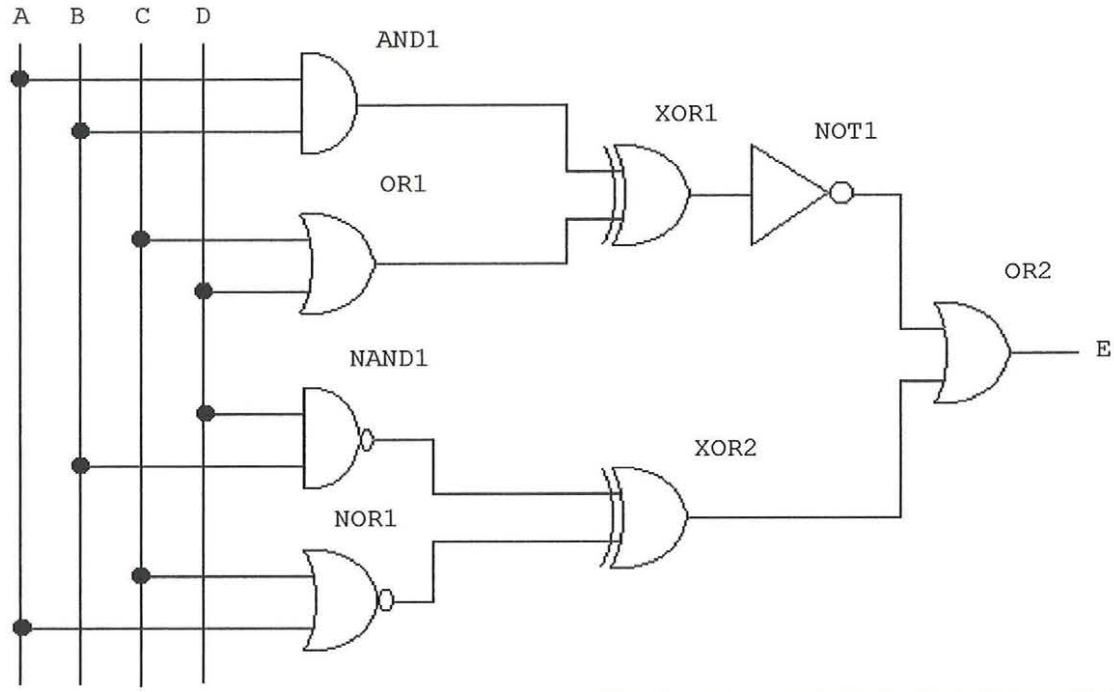
2.11 Schematic Component Connection Analysis

We provided a detailed overview of the connection analysis process at the beginning of this chapter. In this section we will describe the details of the connection matrix, and precisely how it is processed in order to formulate tokens for the equation which describes the circuit.
In figure 14 below, we show a typical circuit, and the accompanying connection matrix which would be generated by TOKSCAN during connection analysis. In this example, the

user selected circuit input points A, B, C, and D, and output point E, as labeled in the schematic.

After the component recognition processes have been run, the user requests for TOKSCAN to generate an equation. It begins (in the example) by starting at each of the user selected input points for the circuit (A, B, C, and D), and following all signal connection lines through all components until it reaches the indicated output point E.

If the schematic has multiple outputs, TOKSCAN creates an equation for each output. The connection matrix supports multiple outputs because it is dynamically allocated, and contains columns for each output indicated by the user. It is always a square matrix, and the columns are allocated as follows: 1) The first n columns are allocated for the user-defined input points to the circuit, where n inputs have been specified. 2) One additional column is allocated for each schematic component recognized in the image. 3) The last m columns are allocated for the user-defined output points from the circuit, where m outputs have been specified. There is a corresponding row in the matrix for each column, ordered as shown in figure 14. The data inserted in the matrix is always a 1 (connection) or 0 (no connection). Each column can be viewed as a list of all of



	INP A	INP B	INP C	INP D	AND 1	OR 1	XOR 1	NOT 1	NAN 1	NOR 1	XOR 2	OR 2	OUT E
INP A					1					1			
INP B					1				1				
INP C						1				1			
INP D						1			1				
AND 1							1						
OR 1							1						
XOR 1								1					
NOT 1												1	
NAN 1											1		
NOR 1											1		
XOR 2												1	
OR 2													1
OUT E													

Figure 14: Connection Matrix Contents

the components which are connected to the input side of the component which the column represents. For example, looking at the "AND 1" column, inputs A and B are connected to the input side of AND gate 1, as indicated by the ones in the first two rows of that column. Similarly, each row can be viewed as a list of all of the components which are connected to the output side of the component which the row represents. Looking at the "AND 1" row, for example, we see that the output of the AND 1 gate is connected only to the input side of the XOR 1 gate.

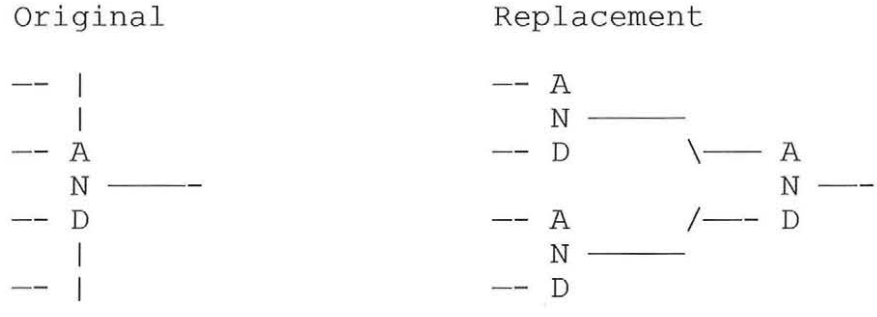
During the line-following process, when TOKSCAN determines that a signal line makes a connection with a component, it makes an entry in the connection matrix. At each point where a circular line connector has been identified, TOKSCAN performs recursive analysis, and follows each possible output from the electrical junction.

The final result of the line-following analysis for the example in figure 14 is as shown in the matrix. All entries which are blank contain zeros (which are omitted for clarity).

2.12 Construction Of A Token List For Equation Generation Based On The Connection Analysis

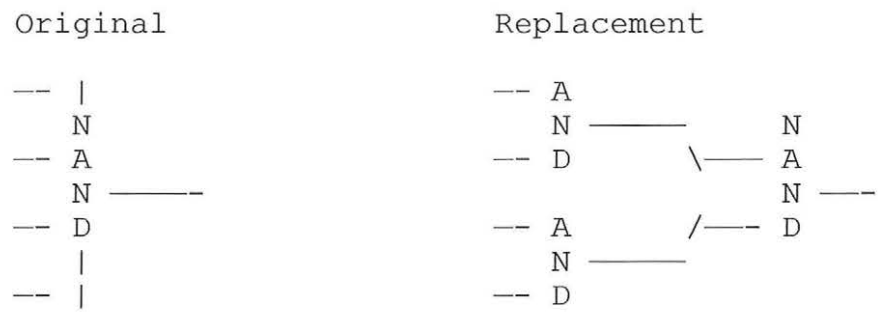
After the connection matrix has been completed, we have extracted sufficient information from the image to generate the desired equations. The next step performed by TOKSCAN is to start with the output columns in the matrix, and work backwards toward the user-indicated inputs to the whole circuit, generating a set of tokens (taken from the matrix row/column labels) which identify each schematic component. The generated tokens are ordered so as to indicate the connections between the components. In order to work backwards from output to input, TOKSCAN follows the connections indicated in the matrix. For example, looking back at figure 14, TOKSCAN starts with the OUT E output column and looks for entries in this column which indicate a connection. It finds one connection with the OR 2 component, and generates an OUT-E output token, followed by an OR-2 output token. Then it examines the OR 2 column, looking for any connections, which it finds with components NOT 1 and XOR 2. It generates NOT-1 and XOR-2 output tokens, and examines the NOT 1 and XOR 2 columns. This process continues in like manner until all connections have been processed, and all paths have been traced back to the user-indicated inputs. The output tokens are passed to a

YACC-generated parser which is described in the next section. TOKSCAN is capable of handling up to four input signal lines for each AND, OR, NAND, NOR, and XOR gate in a schematic. The parser is only capable of handling two-input components, so when a component is recognized in the image with more than two inputs, the single component is replaced (in the connection matrix only) with a logically equivalent set of two-input components. For example, a four input AND gate would be replaced in the matrix by the following set of gates:



The four-input OR and XOR gates have a similar replacement in the matrix.

A four-input NAND gate would be replaced by the following set of gates:



The replacement equivalent for the four-input NOR gate is like the NAND, substituting OR gates for AND gates, and a NOR gate for the NAND gate.

2.13 Construction Of A YACC-Generated Parser To Analyze Equation Token Lists

The final analysis step is to take the tokens generated from the connection matrix, and send them to the parser for conversion into a logic equation. We will look first at the input to the parser, and then at the rules used in the parser for conversion. The following is an example of the token input to the parser (which is the output of the connection matrix processing described above) for the example in figure 14. The sample tokens shown here are not in the exact format used by the program; they have been

altered to match the labeling in figure 14. The important points are that each token is a label which identifies a schematic component, and the tokens are ordered so that connections between them are preserved.

```
OUT.OUT-E
     OR_.OR2

OR_.OR2
     XOR.XOR2
     NOT.NOT1

XOR.XOR2
     NAN.NAND1
     NOR.NOR1

NAN.NAND1
     INP.INP-B
     INP.INP-D

NOR.NOR1
     INP.INP-A
     INP.INP-C

NOT.NOT1
     XOR.XOR1

XOR.XOR1
     AND.AND1
     OR_.OR1

AND.AND1
     INP.INP-A
     INP.INP-B

OR_.OR1
     INP.INP-C
     INP.INP-D
```

Blank lines and tabs have been added to the token list to make it easier to read and associate related components visually. Each group of tokens represents a specific

component in the schematic and the one or two associated components which are connected to it's input side. When the components which are connected to the input also have components connected to their input, additional groups of tokens are inserted for them, with the same meaning. The tokens are inserted in the list with the same output-to-input ordering which is used by the process that goes through the connection matrix and creates the tokens. Whenever that process encounters a component which has two input connections, it uses recursive analysis to create tokens for both input trees (the whole structure can be though of as a tree with the circuit output point as the root).

The first three characters of each token (INP, XOR, etc.) are passed to the YACC-generated parser. The parser replaces the tokens with appropriate Boolean operators and parentheses, and it extracts the input identification letters (A, B, C, etc.) and places them in the proper places among the operators and parentheses. The YACC rules used to generate the parser can be found in Appendix A, Code Fragment #4.

2.14 Connection Line-Following Techniques, Equation Determination, And Output

We have already discussed most of the important points related to the line-following function in TOKSCAN: 1) The line-following process is performed using the vector chains stored in compound shape objects labeled as LINEs. 2) Electrical connections between signal lines are detected by looking for circular line connectors, and at each detected junction, recursive analysis is used to follow all paths. 3) The user indicates input and output points for the circuit as a whole. 4) Using the vector chains, lines are followed from the circuit inputs through detected components, to circuit outputs, and a connection matrix is built at the same time. 5) Line following can be tracked visually by the user, because the signal lines are highlighted in blue during execution of the process.

One small additional point to be made here is that TOKSCAN checks all line end points for a junction with other line end points, and when found, forces every end point at the junction to have the same exact vector coordinates (prior to starting the line-following process).

After the line-following function and the parser have completed processing, the logic equation created by the parser is passed back to TOKSCAN, and is displayed visually on the image bitmap.

2.15 User-Adjustable Parameters For The Recognition Process

In our previous discussion, we have mentioned a number of variables in the recognition process which may be adjusted by the user in order to adapt TOKSCAN's processing to a particular image. In this section we will list those already mentioned, plus several more, and provide a brief explanation of how they are used to facilitate recognition. All user-adjustable parameters in TOKSCAN may be accessed from the USER PARMS menu. The parms are grouped together under the dialog box which displays them for the user.

Curvature And Junction Margin Of Error Dialog:

Curvature Range For Corner Detection: The user can set low and high threshold values for detection of a corner during the vectorization process. When the measured curvature between vectors falls within the threshold range, then

TOKSCAN identifies the location as a CORNER (when analyzing a schematic component).

Maximum Gap Between Base Shapes And Appendage Shapes Or Line End Points For A Junction To Exist: The user can specify how close neighboring objects must be to each other before they are linked together to form a compound object.

Curvature Detection - Threshold Distance: This is the threshold distance discussed in section 2.5, and illustrated in figure 13.

Curvature Detection - Percent Beyond Threshold: This is the percentage of the total perimeter of the shape which must be beyond the threshold value (above) in order to consider the arc between two corners as a left or right curve. See section 2.5 and figure 13.

Curvature Detection - Minimum Distance Between Shape Corners: This is the minimum distance between any two corners in a shape. See section 2.5 and figure 13.

Curvature Detection - Percent Of Perimeter For Maximum Curve Vector Lengths: If two successive vectors in a shape both have right or left curvature, and the first vector of

the pair is sufficiently small, the sum of their curvature will be calculated and used to determine the presence of a corner. Also, if two successive vectors in a shape have opposing curvature, and if the first vector is sufficiently small, then the net curvature of both vectors is calculated (so there is a "canceling out" effect) before determining if a corner exists at the location. This user-adjusted parameter specifies the maximum length of the first vector of the pair in both of these cases, expressed as a percentage of the total perimeter length of the shape.

Perimeter Values Dialog:

Base Shape Max Perimeter: The user can specify the maximum perimeter length for schematic component base shapes (AND, OR, and NOT shape). Shapes with a larger perimeter are ignored.

Appendage Max Perimeter: The user can specify the maximum perimeter of appendage shapes (used in the XOR, NOR, NAND, and NOT gates). Any detected closed minimal polygon with a perimeter length less than or equal to this threshold will be recognized as an appendage shape, provided that the perimeter is greater than the minimum perimeter for closed minimal polygons.

Line Smoothing Parameters Dialog:

Pixel Low Range - Pixel High Range: These two user-determined parameters specify the minimum and maximum length of vectors (in pixels) to which the associated parameters "Maximum Distance From Curve" and "Maximum Curvature For Any Two Vectors" should be applied, during the vectorization process.

Maximum Distance From Curve: The user can specify the maximum distance the approximating vectors can be from the actual data points in the image (in the vectorization process). See section 1.2.1.3 and figures 2 and 3.

Maximum Curvature For Any Two Vectors: The user can specify the maximum curvature between any two vectors for purposes of generating a vector end point. If, for example, a value of 45 degrees is specified, and a potential vector would have a curvature with the last vector of 45 degrees or more, then the vectorization algorithm backs up to a prior data point and forces a vector end point to be created. See section 1.2.1.3, and figures 2 and 3.

Close Polygon Search Parameters Dialog:

Maximum Number Of Chains: The search algorithm which detects closed minimal polygons searches through multiple simple shape objects in its attempt to find polygons. The user may specify the maximum number of neighboring chains which will be added to the "traversed list" before giving up on a particular search. This is to prevent excessive processing time when there are a large number of shape objects to search.

Maximum Distance In Pixels From End point Of Current Vector To A Data Point Within A Previously Processed Vector In Order To Detect A Closed Polygon: The user can specify the maximum gap between a vector end point and previously traversed vectors in order for a closed polygon to be detected.

Minimum Non-used Percentage: The user can specify the percentage of the perimeter length of a potential closed polygon which must not have already been used in other successfully detected closed polygons, in order for the potential polygon to be accepted. This is to prevent the same closed polygon from being detected multiple times in different searches.

Minimum Length Of Perimeter Of Detected Closed Polygon: The user can specify the minimum perimeter of a valid minimal closed polygon. Polygons with a smaller perimeter are ignored.

Maximum Gap In Pixels Between Vector End Points In Order To Join Two Shape Chains Into A Single New Shape Chain When Reorganizing Vectors: The vector chains contained in the simple shape objects are reorganized and combined into longer chains with a more uniform direction during vectorization. Neighboring vector chains may be joined together into a single chain if the gap between them is not greater than a user-specified maximum, which is this parameter.

Maximum Gap Between The End Of A Previous Vector And A Data Point On A New Shape Chain In Order To Determine A New Next Shape Chain To Add To The Current List When Determining Closed Polygons: As vector chains are merged into a list which eventually describes the perimeter of the closed polygon, each data point along the vector chain is examined to see if it is close enough to a data point in a neighboring vector chain so that the neighboring chain should be merged into the list. This parameter is the threshold (maximum) distance for that merge to take place.

Shape Object Parameters Dialog:

Maximum Length Of Vectors Which Are Automatically Removed From Compound Shape Objects Because Of Insignificant Length: After recognized schematic components are removed from the collection of simple shape objects, and before the remaining simple shape objects are converted into signal line objects, TOKSCAN looks for small vectors with a maximum length specified by this parameter, and eliminates them entirely from consideration. This is done to filter out extraneous "noise" vectors caused by distortion.

Minimum/Maximum Degrees Curvature For Detection Of Counterclockwise Traversal In A Closed Polygon: In order to ensure that the shape tokens which describe basic shapes are consistent, TOKSCAN must traverse all of the closed polygons in the same direction (clockwise or counter-clockwise) during vectorization and token generation, and the counter-clockwise direction has been arbitrarily chosen (either would be equally acceptable). This pair of user parameters must be set to a range of values which surround -360 degrees. The net curvature for the entire perimeter is summed during a traversal, with all right (clockwise) curvature being positive, and all left (counter-clockwise) curvature being negative. The result is compared with this

threshold range. If the result is within the range, then the vectors which describe the shape were generated with a counter-clockwise traversal; otherwise, the direction of all vectors in the chain is reversed (to make the traversal counter-clockwise). A threshold range is provided (rather than using -360 degrees precisely), because the measurement of curvature in a discreet x,y coordinate system can never be exact, and because of floating point rounding errors.

Center Point Search Radius: TOKSCAN performs a circular search around the end points of signal lines to see if circular line connectors are present. In order to provide a flexible search which can overcome distortion, multiple circular searches are actually performed, each with a center point that falls within a user-specified radius around the end point of a given line. This parameter is that user-specified radius.

Circular Line Connector Radius: When TOKSCAN searches for circular line connectors, it looks for a black circle in the input image with a minimum radius which is specified by the user. This parameter is that radius.

Maximum End Point Gap: The user can specify a maximum gap between two connector line objects in order for there to be

a junction between the lines. This parameter is that maximum gap.

Minimum Connector Line Length: The user can specify the minimum length of a connector line with this parameter. Shorter lines are removed from consideration during image recognition.

XOR Gate Spur Line Maximum Length: The XOR gate has two small "spur" lines which are attached to the input appendage as a normal part of the symbol. TOKSCAN removes these spur lines in order to avoid errors in the detection of input signal lines for XOR gates. The user can specify the maximum length of these spur lines with this parameter. Lines longer than the user-specified value are not removed, even when they are in the proper position on the input side of an XOR gate.

Token Edits Dialog:

Convex Perimeter Percent: This user-defined parameter is used to help distinguish valid component shapes from invalid closed polygons. The AND and OR shapes should have a certain percentage of their total perimeter which forms a convex arc. This parameter specifies that percentage.

Plus/Minus Percent: This parameter specifies a threshold range around the Convex Perimeter Percent value. If the length of the perimeter forming a convex arc is equal to the Convex Perimeter Percent plus or minus the value set in this parameter, then the shape has a correct proportion of convex arc.

CHAPTER 3

NOTES ON THE IMPLEMENTATION USING THE MICROSOFT FOUNDATION CLASSES AND A YACC-GENERATED PARSER

Microsoft's Visual C++, version 4.2 (Enterprise Edition) was used to implement TOKSCAN (it can also be compiled, linked, and tested successfully with version 4.0, without making any modifications). The development environment and the Microsoft Foundation Class library which are provided as a part of this software package both proved to be invaluable during project development. In this chapter, we will discuss a few of the most helpful features in the software, and we will also briefly discuss the use of the YACC (Yet Another Compiler Compiler) software package used to create the token parser.

The full source code, executable code, supporting data files and test image files of schematic circuits for this project are all available on a Zip Disk in the office of the College of Computing Science and Engineering at the University of North Florida. Small portions of the code were generated using the Microsoft utilities described in this chapter. A very few sections of code (primarily having to do with reading, writing, and displaying bitmap files)

were adapted to this project from sample source code provided by Microsoft Corporation.

3.1 The Development Environment

The "Microsoft Developer Studio" is the development environment which was used for this project. It consists of a set of integrated editors which can handle the plain text files used for C++ source code, and the various kinds of support file formats used in the Microsoft Windows environment, such as icon bitmaps, general purpose bitmaps, toolbar graphics files, dialog resources, and other various resource files.

3.1.1 Automatic Generation Of Skeleton Code And User Interface Resources

One of the features of the Developer Studio which worked well and saved a great deal of time was automatic skeleton code generation through the use of the "App Wizard" and the "Class Wizard" (Microsoft's terms for code generation functions which they provide). The "App Wizard" made it possible to generate a set of skeleton C++ source files and resource files at the time the project was created on the computer, which contained enough code to open a "standard"

Windows program with a multi-document interface - in a matter of a few minutes.

As the project was being developed, the "Class Wizard" made it possible to add new skeleton C++ files which supported various dialog windows (at a minimal level) with a few clicks of the mouse.

Dialog resource editors also made it possible to build new dialog windows quickly with the mouse, using a "drag and drop" graphical user interface.

Other tools made it possible to build graphical toolbars and mouse cursors, and to import standard class functionality into the program (such as a "progress" dialog box which gave a graphical indication of the amount of progress made in a long running task), all in a matter of a few minutes.

This does NOT imply that all of the development could be done with "point and click" methods. It took five months of intensive C++ coding, debugging, and refinement to fill in the "shell" programs that were created automatically. However, based on the author's past experience with Windows development without the benefit of the Developer Studio or

Foundation Classes, these tools saved a significant amount of time. Initially, they were slower and more tedious to use than other, more familiar development methods, but once mastered, the increase in productivity was rapid.

3.1.2 Visual Class Tree View With Fast Access To Member Functions And Class Definitions

A feature of Visual C++ version 4.0 and 4.2 which proved to be extremely helpful during development was the class "tree view" window in the Developer Studio. Whenever a new C++ class is defined and added to a project, a visual entry is made in this window. The user can expand the "tree" with mouse clicks, and all of the class member functions and member variables are displayed. If the user double-clicks on any of these displayed items, the Developer Studio automatically opens the source code at the place where the entity is defined. The tree view serves as a dynamic index to every source code entity in the project, with hypertext-like links to each item. During the development of the project, this index worked very well, and saved development time.

3.1.3 High Quality Debugger With Convenient Visual Features

A high quality program debugger was provided with Visual C++, which provided much better presentation of program

data than older versions of Microsoft debuggers. In addition to displaying variable states and data just by pointing the mouse cursor, it was possible to view the information contained within an entire object instance of a class with a few mouse clicks. Also, a "call stack" display was provided, which showed the various layers of function calls currently active at any point during program execution, including source code references for each call.

3.2 The Microsoft Foundation Classes

The Microsoft Foundation Class (MFC) library was the single most effective time saving tool provided with Visual C++. Complicated graphics manipulation, dialog box management, memory management, and many other processing functions could often be handled just by instantiating a particular MFC class, and making a few member function calls, often with only a very few lines of source code.

It must be emphasized that initially, it proved to be very slow, tedious work to use the MFC library, until familiarity was gained with the various classes, and how they work together. However, as in the case of the Developer Studio, once mastered, the gain in productivity was enormous, and well worth the initial effort.

3.2.1 Effective Encapsulation Of The Windows API

The Windows API (Application Programming Interface) consists of hundreds of complicated function calls, often with numerous, and sometimes obscure parameters being passed to the operating system. The purpose of many of the MFC classes is to encapsulate the API calls into a class interface which is more organized and easier to remember and use. In many cases during the development of this project, it proved possible to accomplish with two or three lines of source code using MFC what would have taken a page or more of source code using the Windows API directly.

3.2.2 Template Classes Which Support Dynamic Arrays, Lists, And Maps

The MFC template classes CArray and CMap, (and a few others) proved very useful in nearly every situation where dynamic allocation of an array or collection was needed. The CArray class can be defined to hold an array of object instances, or the primitive data types. The CMap class can be defined to both hold a collection of object instances, and provide keyed access to particular instances of the object, through the use of a hash table. The CMap class can handle very large numbers of instances of objects in an efficient manner. During testing, thousands of object

instances were created (at one time) and manipulated using CMap functionality without any problems related to the Microsoft implementation.

Performance was never a problem when using the template classes during the development of the project. Although there were many performance issues to resolve, none of them were related to the implementation of MFC.

3.2.3 Effective Memory Management And Support For Large Collections Of Objects

The nature of the TOKSCAN program made it necessary to manipulate hundreds or thousands of instances of objects in memory at one time. Dynamic memory allocation was a requirement throughout the program. When using MFC, the only source code which had to be written which was directly related to memory management had to do with freeing object instances which were created using the C++ "new" operator. This code consisted of issuing the "delete" command with a reference to a pointer to an object instance. (A few functions were written without using MFC functionality, such as those needed to read, write, and manipulate Windows BMP files. In this case, more coding was necessary to properly allocate and free memory).

3.2.4 Problems Encountered During Development

There was only one problem encountered during project development of any real significance which was directly related to the Microsoft MFC implementation. This had to do with using the DAO (Data Access Object) classes to access a Microsoft Access database file. MFC version 4 has an internal problem where it unloads a system DLL from memory during program shutdown before it has finished accessing the DLL. This showed up during the development of the project, and resulted in a program GPF (General Protection Fault) during shutdown. Microsoft provided a one line source change to work around the problem which was successful; to issue the command: "LoadLibrary("msjt3032.dll");" during program startup. (msjt3032.dll is a dynamic link library which supports access to Microsoft Access database files).

3.3 Development Of The Parser

In order to generate the parser for the TOKSCAN project, a port of the Bison YACC program which runs under Microsoft Windows 95 was selected, and downloaded from the Digital Equipment Corporation archive mirror site: http://ftp.digital.com. The full address to reach index

information about the program is:

http://ftp.digital.com/cgi-bin/grep-index?bison

The file downloaded for the project was:

/pub/micro/pc/winsite/win95/programr/flexbison.zip.

Information about the author was not easily available.

After installing this version of Bison on a PC, the source code containing the YACC rules was processed to create the necessary C source code (the full YACC source is shown in Appendix A, Code Fragment #4). The C code was compiled and linked into a static link library, which was then included into the TOKSCAN project.

The parser was developed with very few problems in a matter of a few days.

CHAPTER 4

EXPERIMENTAL RECOGNITION RESULTS

In the following pages are some figures which show output from TOKSCAN's recognition process.

Figure 15 is a test image which was created with the Microsoft PAINT utility program. It contains a sample of each type of logic gate that TOKSCAN can recognize. The logic equations for all of the images use the following symbols:

(+) = exclusive OR

* = AND

\+ = OR

' = NOT

Extensive use of parentheses is present to ensure non-ambiguity of results (at the expense of readability).

Figure 16 is a scanned test circuit. The complete image could not be displayed because of its size. The labels for

input and output were drawn in for publication, because the computer-generated labels were scrolled off the screen.

Figure 17 is scanned, and is an image of the first stage of a full adder circuit. A NOR gate at the top of the image (which provides circuit OUTPUT A) is not shown, because it is scrolled off the screen. Inputs A and B are signal lines for two binary values which are to be added together. Input C is a CARRY IN signal(which would be connected to the CARRY OUT from another adder circuit). Output A is the CARRY OUT signal from the adder, and output B is the result signal from the addition of the two input values.

Figure 18 is scanned, and is an image of a four-input multiplexer circuit. Input signal line A is not shown because it is scrolled off the screen. Inputs A, B, C, and D are the four primary inputs to the multiplexer. Inputs E and F (bottom of the screen) are the switching signals which trigger the selection from the four inputs. Output A is the output from the circuit.

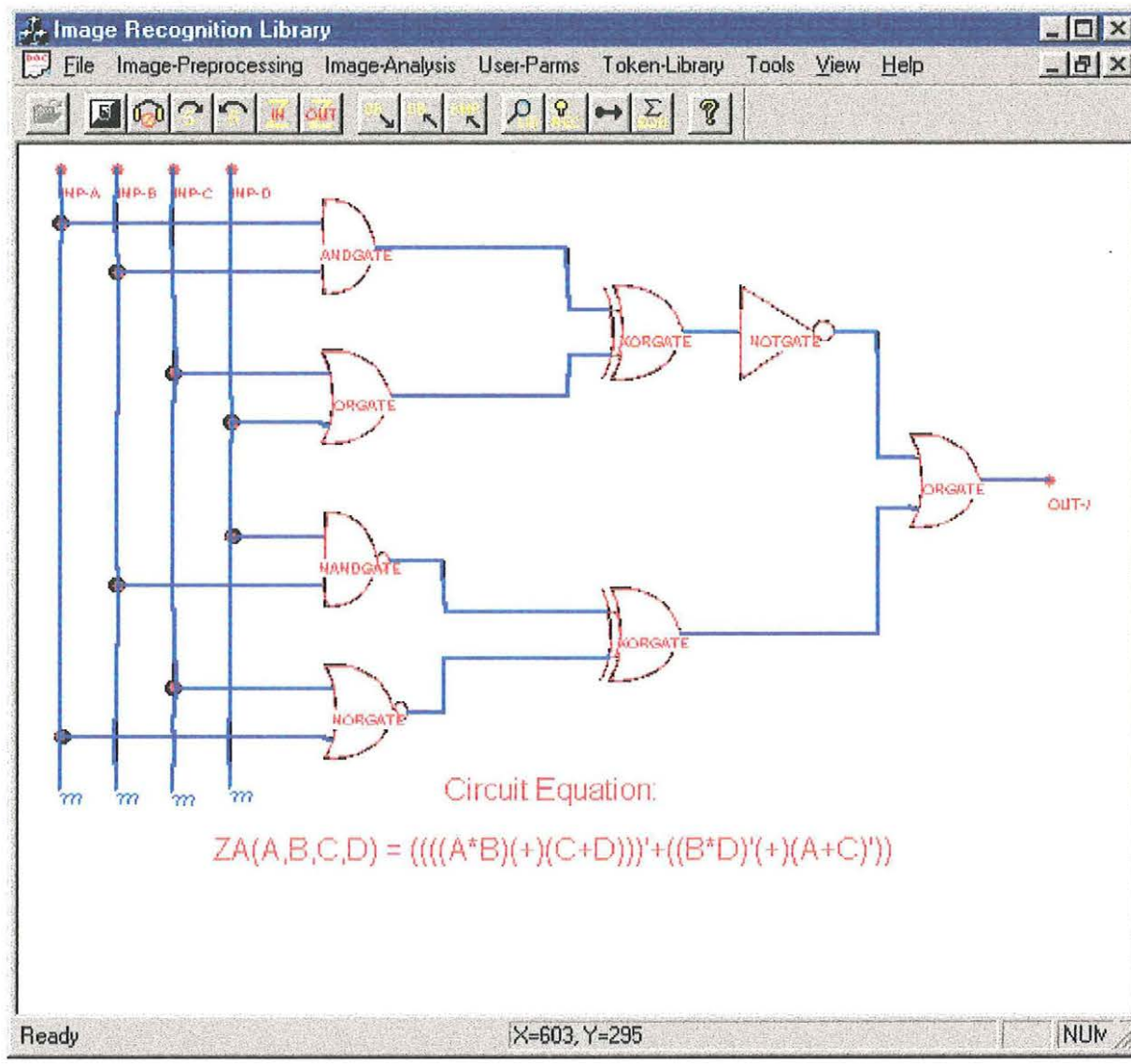


Figure 15: Computer-Drawn Test Image With Every Type Of Logic Gate Which Can Currently Be Recognized

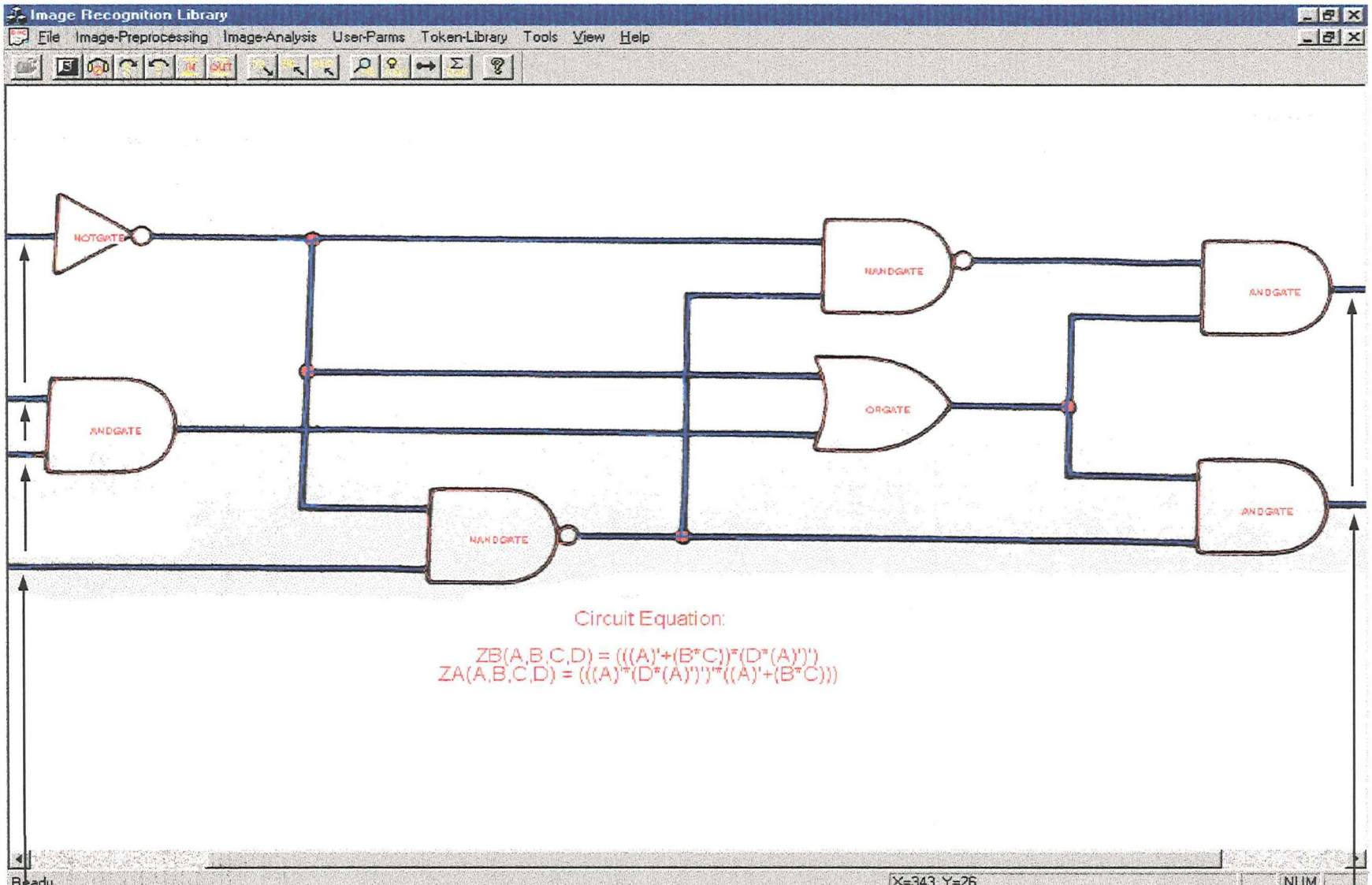


Figure 16: Scanned Image - Exercise Circuit
(Black arrows were drawn in later for labeling)

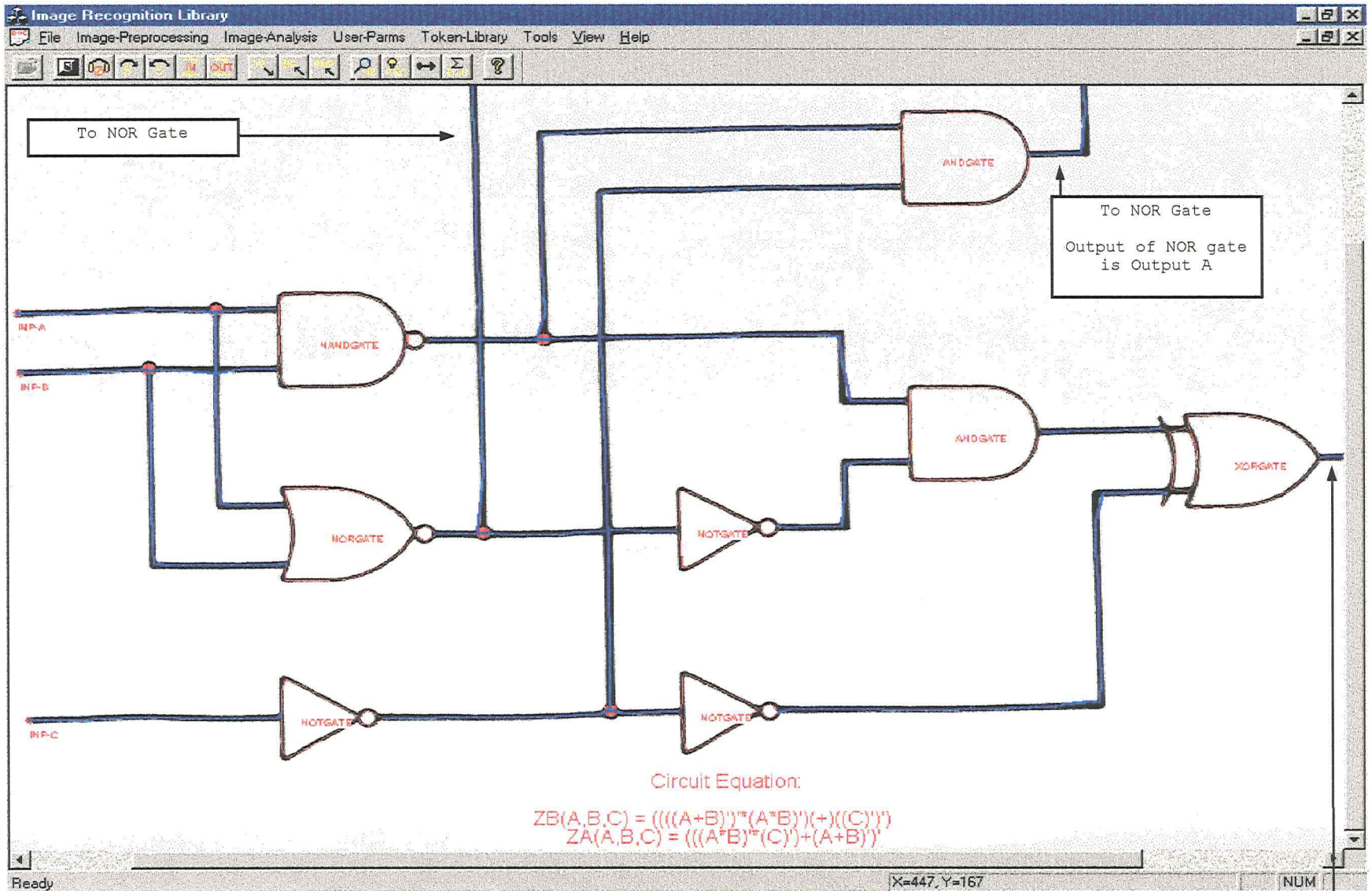


Figure 17: Scanned Image - First Stage Of Full Adder Circuit (NOR gate, upper right, not shown)
(Black arrows and boxes were drawn in later for labeling)

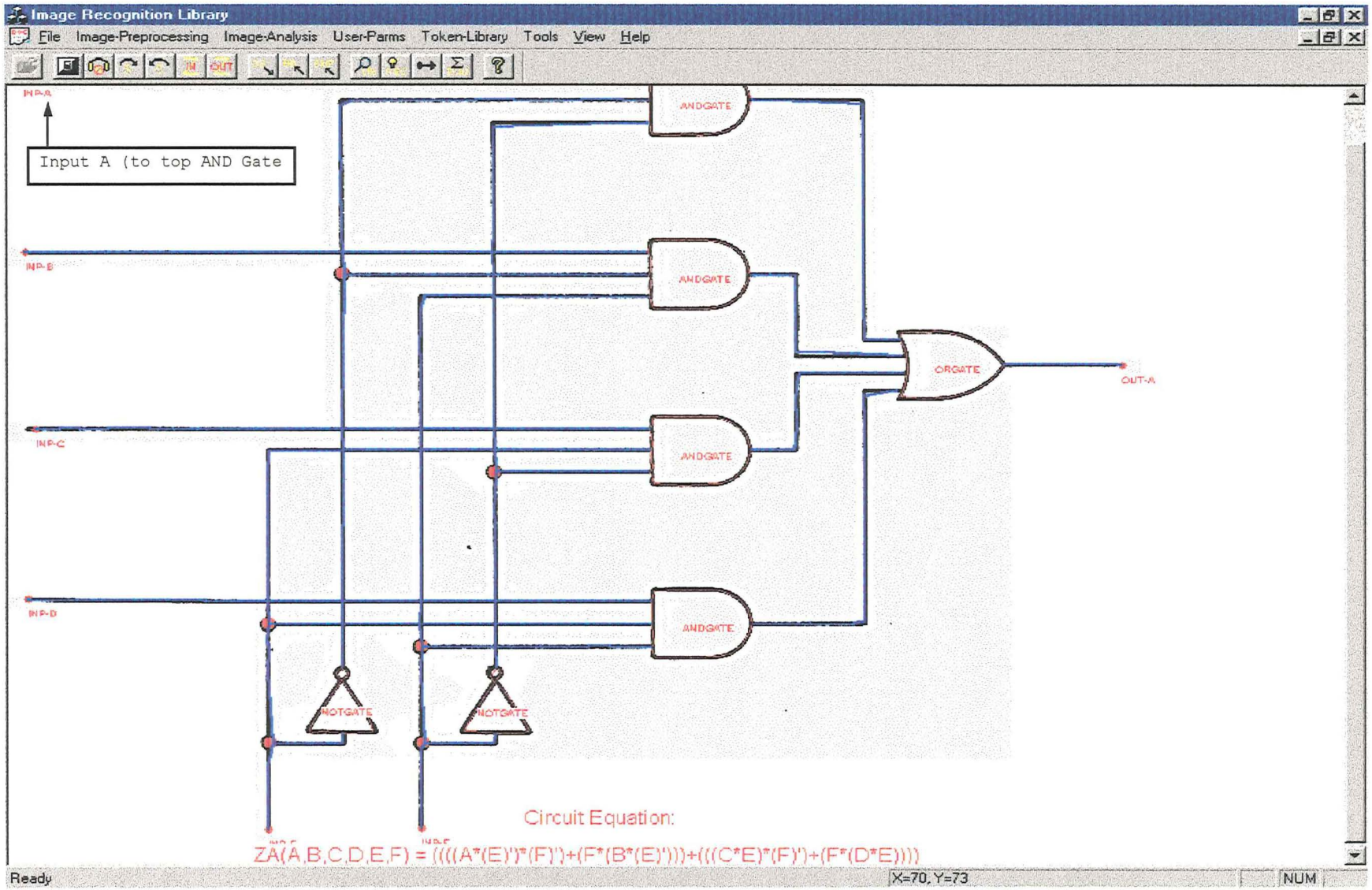


Figure 18: Scanned Image - Four Input Multiplexer
(Black arrow and box were drawn in later for labeling)

CHAPTER 5

CONCLUSION AND SUGGESTIONS FOR FURTHER DEVELOPMENT

In conclusion, we have shown that token-based analysis of schematic images is practical, and that it provides a simple method for representing logic components in a form which can be easily manipulated by the computer. TOKSCAN was developed as a prototype to show the usefulness of this approach. It is the hope of the author that others interested in image processing will work with it further, and extend its capabilities.

There are a number of improvements needed in TOKSCAN, that were beyond the scope and/or time constraints for this project. Some of these are as follows:

1. Improved pre-processing of images. The thinning function is adequate, but introduces significant distortion that impairs recognition. An improved algorithm should be implemented, preferably one which performs thinning with a single raster scan of the image. Additional processing to remove "spur" lines (which are artifacts left behind by the thinning process) would also be helpful. The binarization

function should use adaptive local thresholding rather than the simple global threshold which is currently implemented. The noise reduction function should be improved by using a different algorithm for smoothing. There was not sufficient time to experiment thoroughly with different methods for noise reduction.

2. The shape relationship information, shape description tokens, and equation generation tokens are implemented with some "hard coding". For example, to identify base shapes (as opposed to appendage or line shapes), the program looks at the name of each relationship in the relationship table, and searches for names ending with "BASE". It looks for names ending in "APPEND" to identify appendage shapes. It also identifies the kind of equation generation token to send to the parser by looking at the first three characters of the name in the relationship table, expecting to find "AND", "OR ", "NOT", "XOR", "NOR", "NAN", "INP", or "OUT". All code related to token usage should be reviewed, and should be modified to make the addition of new kinds of tokens possible without recompiling the program. A more extensive change would be to build a "rules" file which specifies the processing rules to be applied for a given token, thus generalizing the token functionality further.

3. The set of functions which identify closed minimal polygons should be made more robust. When new input images are analyzed, a few components are usually missed because of a failure to detect a closed minimal polygon. This problem can usually be corrected by either adjusting the user-specified parameters, or making a minor program change. When this problem occurs, it is usually related to noise in the image caused by distortion from the thinning process.

4. The set of user-specified parameters is large and complex. A rather extensive modification which would make TOKSCAN easier to use, would be to have the program manage more of the parameters, and perform automated adjustments to the initial parameter values, based on a comparison of processing results with a user-input set of expected results. For example, the user could provide a small amount of quick mouse input prior to image analysis, setting a boundary around each component in the image, so that TOKSCAN would know the number and approximate location of all components prior to analysis.

5. The set of recognizable components should be expanded to include the more complex symbols used in most logic schematics. Text segmentation and recognition would have to

be included as well, because in some cases text labels in the schematic help to identify the type of symbol.

6. The scale of recognition could easily be increased. It would be easily possible to build a library of equations, or equation prototypes, which could be linked with the names of more complex logic structures, such as memory units, J-K flip-flops, etc., and which could be associated with the current equation output. The result would be a high order labeling of more complex components.

7. Another useful extension of TOKSCAN would be to give it the ability to handle sequential circuits, and feedback. One approach which could be used, would be to give the user the ability to select combinational circuit components within sequential circuits (probably via visual selection with the mouse), and to analyze each combinational block separately. Feedback signals could be isolated in the diagram, and handled separately.

REFERENCES

[Andrews76]
Andrews, H., "Monochrome digital image enhancement", Applied Optics, 15, (February 1976), pp. 495-503. This article reprinted in Digital Image Processing and Analysis: Volume 1: Digital Image Processing, < Bill D. Carroll, Jack Cotton, Jerome R. Cox Jr, Ez Nahouraii, Chuan-lin Wu>, ed., IEEE Computer Society Press, Silver Spring, Maryland, 1985, pp. 431-439.

[Cootes95]
Cootes, T., et al, "Active Shape Models - Their Training and Application", Computer Vision And Image Understanding, 61, 1, (January 1995), pp. 38-59.

[Flickner95]
"Query by Image and Video Content: The QBIC System", Computer, 28, 9, (September 1995), pp. 23-31.

[Gonzales92]
Gonzales, R. and R. Woods, Digital Image Processing, Addison-Wesley Publishing Company, Reading, Massachusetts, 1992, pp. 81-156, 595-619.

[Hori92]
Hori, O. and A. Okazaki, "High Quality Vectorization Based on a Generic Object Model", Structured Document Image Analysis, Springer-Verlag, (1992), pp. 325 - 339. This article reprinted in Document Image Analysis, <Lawrence O'Gorman and Rangachar Kasturi>, ed., IEEE Computer Society Press, Los Alamitos, California, 1995, pp. 78-92.

[Joseph92]
Joseph, S. H., and T. P. Pridmore, "Knowledge-Directed Interpretation of Mechanical Engineering Drawings", IEEE Transactions on Pattern Analysis and Machine Intelligence, 14, 9, (September 1992), pp. 928 - 940. This article reprinted in Document Image Analysis, <Lawrence O'Gorman and Rangachar Kasturi>, ed., IEEE

Computer Society Press, Los Alamitos, California, 1995, pp. 453-465.

[Kamel93]
Kamel, M. and A. Zhao, "Extraction of Binary Character/Graphics Images from Grayscale Document Images", Computer Vision, Graphics, and Image Processing, 55, 3, (May 1993), pp. 203 - 217. This article reprinted in Document Image Analysis, <Lawrence O'Gorman and Rangachar Kasturi>, ed., IEEE Computer Society Press, Los Alamitos, California, 1995, pp. 29-43.

[Kartalopoulos96]
Kartalopoulos, S., Understanding Neural Networks And Fuzzy Logic - Basic Concepts And Applications, IEEE Press, Piscataway, New Jersey, 1996.

[Lam92]
Lam, L., et al, "Thinning Methodologies - A Comprehensive Survey", IEEE Transactions on Pattern Analysis and Machine Intelligence, 14, 9, (September 1992), pp. 869 - 885. This article reprinted in Document Image Analysis, Lawrence O'Gorman and Rangachar Kasturi>, ed., IEEE Computer Society Press, Los Alamitos, California, 1995, pp. 61-77.

[Medioni87]
Medioni, G. and Y. Yasumoto, "Corner Detection and Curve Representation Using Cubic B-Splines", Computer Vision, Graphics, and Image Processing, 39, (1987), pp. 267-278. This article reprinted in Document Image Analysis, <Lawrence O'Gorman and Rangachar Kasturi>,ed., IEEE Computer Society Press, Los Alamitos, California, 1995, pp. 133-144.

[Mehrotra95]
Mehrotra, R. and J. Gary, "Similar-Shape Retrieval In Shape Data Management", Computer, 28, 9, (September 1995), pp. 57-62.

[O'Gorman95]
O'Gorman, L. and R. Kasturi, Document Image Analysis, IEEE Computer Society Press, Los Alamitos, California, 1995.

[Ratha96]
Ratha, N., et al., "A Real Time Matching System For Large Fingerprint Databases", IEEE Transactions On

Pattern Analysis And Machine Intelligence, 18, 8, (August 1996), pp. 799-812.

[Tombre95]
Tombre, K., "Graphics recognition - general context and challenges", Pattern Recognition Letters, 16 (1995), pp. 883-891.

[Vaxivière90]
Vaxivière, P. and K. Tombre, "Interpretation of Mechanical Engineering Drawings for Paper-CAD Conversion", MVA'90 IAPR Workshop on Machine Vision Applications, (Nov. 28-30, 1990), pp. 203-206.

[Vaxivière92]
Vaxivière, P. and K. Tombre, "Celesstin: CAD Conversion of Mechanical Drawings", Computer, 25, 7, (July 1992), pp. 46-54. This article reprinted in Document Image Analysis, <Lawrence O'Gorman and Rangachar Kasturi>, ed., IEEE Computer Society Press, Los Alamitos, California, 1995, pp. 444-452.

[Wall84]
Wall, Karin, and P. Danielsson, "A Fast Sequential Method for Polygonal Approximation of Digitized Curves", Computer Vision, Graphics, and Image Processing, 28, (1984), pp. 220-227. This article reprinted in Document Image Analysis, <Lawrence O'Gorman and Rangachar Kasturi>, ed., IEEE Computer Society Press, Los Alamitos, California, 1995, pp. 111-118.

APPENDIX A

Source Code Listings

This appendix contains only selected sections of the source code which are referenced in the paper. The full source, along with the executable program and data files are provided on the accompanying Zip Disk which is available in the office of the College Of Information Science And Engineering.

Code Fragment 1: CImageObject (Simple Shape Object) This is the definition of the first, and simpler of the two primary objects which represent shapes in the schematic image. It contains vector chains which represent lines found initially in the schematic. The compound shape class in Code Fragment 2 below is derived from this class.

```
// CImageObject.h : header file
//

class CImageObject : public CDocument
{

friend class CCompoundImageObject;

protected:

// Attributes
public:
```

```
short   m_delflag;                  // 0=>not deleted from shape map, 1=>deleted
short   m_shapetype;                // shape type code
CPoint  m_upperleft;                // upper left corner coordinates of rectangle
                                    // which bounds the shape
CPoint  m_lowerright;               // lower right corner coordinates of rectangle
                                    // which bounds the shape
int             m_chainnum;         // chain number assigned by analysis routines
CString m_loopflag;                 // flag indicating if the chain forms a loop
CString m_nextvectortype;           // connect type of second vect in the shape chain
int             m_shapetoken;       // token id for this chain's shape
                                    // (leftcurve, rightcurve, etc)
short   m_linethickness;            // thickness of the curve in # pixels width
CArray<CurveInfo, CurveInfo> m_curves;              // array of curve information for
                                                    // each curvature change
CArray<ModVector, ModVector> m_curveschanged;       // array of old and new curvature
                                                    // info, where vector values
                                                    // were adjusted

ChainStartKey m_thisobjchain;       // CHAINSTART table key for the chain which
                                    // makes up this object
linesmoothparm sparms[maxlinesmoothingparms];       // line smoothing parms
short parmcount;                    // number of line smoothing parms
CPoint  m_endchain;                 // x,y coordinates of end of chain

CImageObject *m_startjunc;          // pointer to shape object which continues
                                    // on from this object's shape - start side
CPoint          m_startcoords;      // coordinates of junction point in other chain
CImageObject *m_endjunc;            // pointer to shape object which continues on
                                    // from this object's shape - end side
CPoint          m_endcoords;        // coordinates of junction point in other chain

private:

CCInputBitmapDoc *m_attacheddoc;    // The bitmap document to which this object
                                    // is attached
CPoint          start, prior, curr;

// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CImageObject)
        public:
        virtual void Serialize(CArchive& ar);   // overridden for document i/o
        protected:
        virtual BOOL OnNewDocument();
        //}}AFX_VIRTUAL

// Implementation
public:
                CImageObject();
                CImageObject(const CImageObject& imobjSrc);
virtual ~CImageObject();
const   CImageObject& operator=(const CImageObject& imobjSrc);
void    CopyObject(const CImageObject& imobjSrc);

void    StartNewChain(int xcoord,
                      int ycoord,
                      int chainnum,
                      int curvelow,
                      int curvehi,
                      int junctionerr);

void    InsertNextVector(CCInputBitmapDoc *pDoc,
                         int xcoord,
                         int ycoord,
                         int curvelow,
                         int curvehi,
                         int junctionerr);
```

```
int     MarkEndOfChain(CCInputBitmapDoc *pDoc,
                       int curvelow,
                       int curvehi,
                       int junctionerr);

void    SetSysParms(linesmoothparm *lsp, short parmct);

short   LoadVectorChains  (CTraverseVectorChain *tvect,
                           CDAOChainStartSet *pcset,
                           CDAOVectorChainSet *prset);

void    SetDocPointer(CCInputBitmapDoc *theDoc);
void    DetectCorners(int curvelow, int curvehi, int junctionerr);
void    DrawShapeObjectVectors(int type, CString label);
void    SaveShapeObjectVectors(CDAOChainStartSet *pcset, CDAOVectorChainSet *prset);
void    BuildCurveInfo(int ptcount, CArray<CPoint, CPoint>*ppoints);
void    ReverseShapeDirection(int updshapemapflag);

void    InsertShapeVectors(CImageObject *ImOb2,
                           int vectstart,
                           int vectdir,
                           int blendflag);

void    RecalcCurvature(void);

#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

        // Generated message map functions
protected:
        //{{AFX_MSG(CImageObject)
                // NOTE - the ClassWizard will add and remove member functions here.
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

Code Fragment 2: CCompoundImageObject (Compound Shape Object) This is the definition of the second, and more complex of the two primary objects which represent shapes in the schematic image. It contains all information necessary for recognition.

```
class CCompoundImageObject : public CImageObject
{
public:
        CCompoundImageObject();
        CCompoundImageObject(const CImageObject& imobjSrc);
        CCompoundImageObject(const CCompoundImageObject& imobjSrc);
const   CCompoundImageObject& operator=(const CImageObject& imobjSrc);
const   CCompoundImageObject& operator=(const CCompoundImageObject& imobjSrc);

int     FindContinuationShape(candidateshape *cnd,
                              int allowcircsearch,
                              int pastfirstvect);

void    CalcCurvature(CPoint ptbefore,
                      CPoint junction,
                      CPoint ptafter,
                      int *rtcurvcalc,
                      int *lfcurvcalc);

void    SetShapeDescriptorArray(int descriptor);
CString GetShapeDescriptorArray(int *idx);
int             AnalyzeShape(int startidx, int endidx, tokdata *tok);
void    InsertMergeKey(CString key);
int             MeasurePerimeter(void);
void    GenerateTokens(void);
int             HasCounterClockwiseTraversal();
void    ReverseTraversalDirection();
void    RemoveSmallVectors();
CString MatchBaseShapeTokens();
CString MatchAppendageTokens();
void    CopyOriginalVectors();
void    RemoveShapeFromVectorChains();
DistInfo CalcClosestVectorPoints(CCompoundImageObject *otherobj);
CPoint  HasRectOverlap(CCompoundImageObject *otherobj);
int             FindConnectedObjects(int objtype);
float   MeasurePerimeterBetweenConnections(CPoint con1, CPoint con2);
void    AnalyzeCompoundShape(void);
void    ForceConnectorLineAlignment(int largestlinesmootherrordist);
void    AdjustBoundingRect(CCompoundImageObject *CImOb);
void    CheckForCircularLineConnectors(int centerpointdiameter, int searchdiameter);
int     CheckForLineConnector(CPoint centerpoint, int searchdiameter);
void    DrawLineConnector(CPoint centerpoint, int diameter, int draw);
void    UpdateLineEndpointsWithConnector(CPoint location);
int     CheckVectorPairForCorner(int curridx);

int     IsXORSpurLine(CCompoundImageObject *ComponentOb,
                      CCompoundImageObject *LineOb,
                      CPoint connectlineend);

int     OtherEndptIsCloser  (CPoint thisendpt,
                             CPoint otherlinept,
                             CCompoundImageObject *CImOb,
```

```
                    CCompoundImageObject *OtherOB);

// Attributes
public:
CArray<int, int>      sdesc;          // Compound shape descriptor array
int                   sdesccount;     // Count of # elements in the array
int                   deleteflag;

CArray<CurveInfo, CurveInfo> m_curvesorig;          // array of original,
                                                    // non-adjusted curve information

int                   m_curvesorigcount;

int                   objecttype;                   // 0 => base shape, 1 => appendage,
                                                    // 2 => connecting line
CString               objectname;                   // Name of object from token library
CString               objectcompoundname;           // Name of compound object of
                                                    // which this object is a part

int                   linestartsatconnector;        // 0 => obj is not a line, or
                                                    // line does not start at
                                                    // a connector
                                                    // 1 => obj is a line, and line
                                                    // starts at a connector

int                   lineendsatconnector;          // 0 => obj is not a line, or
                                                    // line does not end at a connector
                                                    // 1 => obj is a line, and line
                                                    // ends at a connector

CArray<CString, CString>      inlines;              // Input Connecting Lines

CArray<int, int>              inlinechainnums;      // Chain Numbers of input
                                                    // connecting lines

CArray<CPoint, CPoint>        inlinecoords;         // Coordinates of junction

CString               inappendage;                  // Input Appendage Shape
int                   inappendchainnum;             // Chain Number of input appendage

CString               basename;                     // Name of base shape
int                   basechainnum;                 // Chain number of base shape

CString               outappendage;                 // Output Appendage Shape
int                   outappendchainnum;            // Chain number of output appendage

CString               outline;                      // Output connecting line
int                   outlinechainnum;              // Chain # of output connecting line
CPoint                outlinecoords;                // Coordinates of junction

CString               matrixkey;                    // Key for this object in the
                                                    // connection matrix, returned
                                                    // by function
                                                    // BuildMatrixKeyAndLabel()

CArray<CString, CString>      tokeninfo;            // Coordinates of corners in shape,
                                                    // and desc of associated line

int                   linefollowingnumlines;        // Number of input lines for
                                                    // this object as determined by the
                                                    // line-following function

int                   circularsearchparm;           // 0 => search from closest
                                                    // points to farthest points
                                                    // in circular search for
                                                    // a continuation vector chain
                                                    // (closed polygon search).
                                                    // 1 => search from farthest
                                                    // points to closest points

private:
CArray<CString, CString>      mergekeys;
```

```
int                     mergekeyscount;

// Operations

public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CCompoundImageObject)
        public:
        virtual void Serialize(CArchive& ar);   // overridden for document i/o
        protected:
        virtual BOOL OnNewDocument();
        //}}AFX_VIRTUAL

// Implementation
public:
        virtual ~CCompoundImageObject();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

        // Generated message map functions
protected:
        //{{AFX_MSG(CCompoundImageObject)
                // NOTE - the ClassWizard will add and remove member functions here.
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

Code Fragment 3: Member Functions Of CCompoundImageObject which assign tokens to basic shapes.

```
void CCompoundImageObject::GenerateTokens()
{

int                     size, idx, cornerscount;
int                     rightdeg, leftdeg;
float                   curvechange;
CString                 sbuf, sbuf2;
CurveInfo               cvinf, cvinf2, cvinf3, cvinf4, cvinf5;
CArray<int, int>        corners;
CArray<int, int>        leftlengths;
CArray<int, int>        rightlengths;
FloatCPoint             p1, p2, p3;
mathfunctions           mfunc;
tokdata                 tok;

        // First, remove any vectors with zero length, or very small length
        RemoveSmallVectors();

        size    = m_curves.GetSize();

        if (size < 2)
        {
                sbuf.Format("Error - Compound Shape # %d Has Less Than 2 Shape Vectors",
                                m_chainnum);
                AfxMessageBox(sbuf);
                return;
        }
```

```
// Clear the token array of any existing values
sdesc.RemoveAll();
sdesc.SetSize(1, 1);
sdesccount = 0;
tokeninfo.RemoveAll();
tokeninfo.SetSize(0, 1);

// Force the last vector endpoint to equal the first vector starting point, and
// recalculate length for the last vector
cvinf2 = m_curves.GetAt(size - 1);
cvinf3 = m_curves.GetAt(size - 2);
cvinf  = m_curves.GetAt(0);
cvinf2.m_endx = cvinf.m_startx;
cvinf2.m_endy = cvinf.m_starty;

cvinf2.m_length =
        (short) mfunc.RoundFloat(mfunc.distance((float) cvinf2.m_startx,
                                                (float) cvinf2.m_starty,
                                                (float) cvinf2.m_endx,
                                                (float) cvinf2.m_endy));

m_curves.SetAt(size - 1, cvinf2);

// Recalculate curvature for all shape vectors except the first
RecalcCurvature();

// Now, find the curvature between the last vector in the compound
// shape and the first,
// and save it to put in the first vector's curvature fields
p1.x = (float) cvinf2.m_startx;
p1.y = (float) cvinf2.m_starty;
p2.x = (float) cvinf2.m_endx;
p2.y = (float) cvinf2.m_endy;
p3.x = (float) cvinf.m_endx;
p3.y = (float) cvinf.m_endy;

rightdeg = 0;
leftdeg  = 0;
curvechange = mfunc.CalcCurvature(p1, p2, p3);
if (curvechange > 0)
        rightdeg = mfunc.ConvertToDegrees(curvechange);
else
        if (curvechange < 0)
                leftdeg = mfunc.ConvertToDegrees((float) -1 * curvechange);

cvinf.m_leftdegreechg  = leftdeg;
cvinf.m_rightdegreechg = rightdeg;
m_curves.SetAt(0, cvinf);

// Now, find the corners in the shape vector array
CImageLib* pApp = (CImageLib *) AfxGetApp();
corners.RemoveAll();
cornerscount = 0;
corners.SetSize(5, 1);

float curvedist, curvedist2;
FloatCPoint lastcurve, thiscurve, firstcurve;
lastcurve.x = (float) -1.0;
lastcurve.y = (float) -1.0;
firstcurve.x = (float) -1.0;
firstcurve.y = (float) -1.0;
curvedist2 = (float) 9999.0;
int perim = MeasurePerimeter();
int vectpaircheck;

// If the curvature between the last vector and the first is correct for a
// corner, then add a corner at the first vector
vectpaircheck = CheckVectorPairForCorner(0);

if ((rightdeg >= pApp->UserDat.curvelow && rightdeg <= pApp->UserDat.curvehi) ||
    (leftdeg  >= pApp->UserDat.curvelow && leftdeg  <= pApp->UserDat.curvehi) ||
     vectpaircheck == 1)
```

```
{
        if (vectpaircheck != 2)
        {
                corners.SetAtGrow(cornerscount++, 0);
                lastcurve.x = cvinf.m_startx;
                lastcurve.y = cvinf.m_starty;
                firstcurve = lastcurve;
        }
}

for (idx = 1; idx < size; idx++)
{
        cvinf = m_curves.GetAt(idx);
        vectpaircheck = CheckVectorPairForCorner(idx);
        if ((cvinf.m_rightdegreechg >= pApp->UserDat.curvelow &&
             cvinf.m_rightdegreechg <= pApp->UserDat.curvehi) ||
            (cvinf.m_leftdegreechg  >= pApp->UserDat.curvelow &&
             cvinf.m_leftdegreechg  <= pApp->UserDat.curvehi) ||
             vectpaircheck == 1)
        {
                thiscurve.x = cvinf.m_startx;
                thiscurve.y = cvinf.m_starty;
                if (lastcurve.x == (float) -1.0)
                {
                        // On the first curve, make sure the distance check
                        // will not prevent entry
                        lastcurve.x = thiscurve.x + 5000;
                        lastcurve.y = thiscurve.y;
                }
                curvedist = mfunc.distance(lastcurve.x, lastcurve.y,
                                           thiscurve.x, thiscurve.y);
                if (firstcurve.x != (float) -1.0)
                        curvedist2 = mfunc.distance(firstcurve.x, firstcurve.y,
                                                    thiscurve.x, thiscurve.y);
                // check curve dist against user-defined percentage of
                // length of shape perimeter
                if ((curvedist >
                     ((float) pApp->UserDat.mindistbetweenshapecorners
                    / (float) 100) * (float) perim) &&
                    (curvedist2 >
                     ((float) pApp->UserDat.mindistbetweenshapecorners /
                      (float) 100) * (float) perim))
                {
                        if (vectpaircheck != 2)
                        {
                                corners.SetAtGrow(cornerscount++, idx);
                                lastcurve = thiscurve;
                                if (firstcurve.x == (float) -1.0)
                                        firstcurve = lastcurve;
                        }
                }
        }
}

// Init the token generation data struct
tok.shapeperim = 0;
tok.smallperimflag = 0;
tok.leftcurvetotperim = 0;
tok.rightcurvetotperim = 0;
tok.straightlinetotperim = 0;
tok.thiscurveleftperim = 0;
tok.thiscurverightperim = 0;
tok.thiscurvestraightperim = 0;
tok.leftcurvecount = 0;
tok.rightcurvecount = 0;
tok.straightcount = 0;
leftlengths.RemoveAll();
leftlengths.SetSize(0, 1);
rightlengths.RemoveAll();
rightlengths.SetSize(0, 1);

// Now, analyze the shape vectors and corner index array,
```

```
// and place a set of shape tokens into the shape
// descriptor array which describe the shape

int wk4, wk5 = 0, maxlcurve = 0, maxrcurve = 0;
for (idx = 0; idx < cornerscount; idx++)
{
        wk4 = corners.GetAt(idx);
        if (idx < (cornerscount - 1))
                wk5 = (corners.GetAt(idx + 1) - 1);
        else
        {
                wk5 = (corners.GetAt(0) - 1);
                if (wk5 < 0)
                        wk5 = size - 1;
        }

        // scan between the m_curves entries referenced by
        // indices wk4 and wk5, and determine a shape descriptor
        int shapetok = AnalyzeShape(wk4, wk5, &tok);
        SetShapeDescriptorArray(shapetok);

        int tkidx;
        for (tkidx = 0; tkidx < numtokens; tkidx++)
        {
                if (shapetok == tokentab[tkidx].code) break;
        }
        if (tkidx < numtokens)
                sbuf = tokentab[tkidx].desc;
        else
                sbuf = "TOKENTABERROR";

        cvinf4 = m_curves.GetAt(wk4);
        cvinf5 = m_curves.GetAt(wk5);
        int rptperim;
        if (tok.thiscurveleftperim > 0)
                rptperim = tok.thiscurveleftperim;
        if (tok.thiscurverightperim > 0)
                rptperim = tok.thiscurverightperim;
        if (tok.thiscurvestraightperim > 0)
                rptperim = tok.thiscurvestraightperim;

        sbuf2.Format("%s: %d,%d - %d,%d  Length: %d",
                        sbuf, cvinf4.m_startx, cvinf4.m_starty,
                        cvinf5.m_endx,   cvinf5.m_endy,
                        rptperim);

        tokeninfo.SetAtGrow(tokeninfo.GetSize(), sbuf2);

        if (tok.thiscurveleftperim > 0)
        {
                int lsize = leftlengths.GetSize();
                leftlengths.SetAtGrow(lsize, tok.thiscurveleftperim);
                if (tok.thiscurveleftperim > maxlcurve)
                        maxlcurve = tok.thiscurveleftperim;
        }
        if (tok.thiscurverightperim > 0)
        {
                int rsize = rightlengths.GetSize();
                rightlengths.SetAtGrow(rsize, tok.thiscurverightperim);
                if (tok.thiscurverightperim > maxrcurve)
                        maxrcurve = tok.thiscurverightperim;
        }
        // add in a corner descriptor for each corner in the shape
        SetShapeDescriptorArray(CORNER);
}

// Scan the array of tokens, and eliminate duplicate entries (can happen at the
// start point of the shape vector array, if the start is not at a corner -
// could have STRAIGHTLINE, STRAIGHTLINE, ... for example
for (idx = 1; idx < sdesccount; )
{
        wk4 = sdesc.GetAt(idx);
```

```
        wk5 = sdesc.GetAt(idx - 1);
        if (wk4 == wk5)
        {
                sdesc.RemoveAt(wk4, 1);
                sdesccount--;
        }
        else
                idx++;
}
sdesc.SetSize(sdesccount, 1);

if (sdesccount > 1)
{
        wk4 = sdesc.GetAt(0);
        wk5 = sdesc.GetAt(sdesccount - 1);
        if (wk4 == wk5)
        {
                sdesc.RemoveAt(sdesccount - 1);
                sdesccount--;
                sdesc.SetSize(sdesccount, 1);
        }
}

// Determine if LEFTCURVE or RIGHTCURVE tokens describe the shape's convex curve

// it will be the token type which has the largest proportion of the total
// perimeter of the shape. If both types have equal perimeter, then we will
// arbitrarily choose LEFTCURVE as the convex curve descriptor
// (it doesn't matter, either could be choosen in this case). Then, see
// if the sum of the convex curve lengths is a correct percentage of the
// total perimeter length.

int convex, minuspct, pluspct;
if ((tok.leftcurvecount > 0 || tok.rightcurvecount > 0) && !tok.smallperimflag)
{
        if (tok.leftcurvetotperim > tok.rightcurvetotperim)
                convex = LEFTCURVE;
        else
                if (tok.rightcurvetotperim > tok.leftcurvetotperim)
                        convex = RIGHTCURVE;
                else
                        convex = LEFTCURVE;

        pluspct  = pApp->UserDat.convexperimpct + pApp->UserDat.plusminuspct;
        if (pluspct > 100) pluspct = 100;
        minuspct = pApp->UserDat.convexperimpct - pApp->UserDat.plusminuspct;
        if (minuspct < 0) minuspct = 0;
        int perimlow =
        (int) ((float) tok.shapeperim * (float) 0.01 * (float) minuspct);
        int perimhi  =
        (int) ((float) tok.shapeperim * (float) 0.01 * (float) pluspct);

        if (convex == LEFTCURVE)
        {
                if (tok.leftcurvetotperim < perimlow ||
                    tok.leftcurvetotperim > perimhi)
                        SetShapeDescriptorArray(REJECT);
        }
        else
        {
                if (tok.rightcurvetotperim < perimlow ||
                    tok.rightcurvetotperim > perimhi)
                        SetShapeDescriptorArray(REJECT);
        }
}

if (pApp->UserDat.multicurvesamelength && !tok.smallperimflag)
{
        if (convex == LEFTCURVE)
        {
                int lperimlow = (int) ((float) maxlcurve * (float) 0.01 *
                                (float) pApp->UserDat.plusminuspct);
```

```
                        int lperimhi = maxlcurve + lperimlow;
                        lperimlow = maxlcurve - lperimlow;
                        if (lperimlow < 0) lperimlow = 0;
                        int asize = leftlengths.GetSize();
                        int wklen;
                        for (idx = 0; idx < asize; idx++)
                        {
                                wklen = leftlengths.GetAt(idx);
                                if (wklen < lperimlow || wklen > lperimhi)
                                {
                                        SetShapeDescriptorArray(REJECT);
                                        break;
                                }
                        }
                }
                else
                {
                        int rperimlow = (int) ((float) maxrcurve * (float) 0.01 *
                                        (float) pApp->UserDat.plusminuspct);
                        int rperimhi = maxrcurve + rperimlow;
                        rperimlow = maxrcurve - rperimlow;
                        int asize = rightlengths.GetSize();
                        int wklen;
                        for (idx = 0; idx < asize; idx++)
                        {
                                wklen = rightlengths.GetAt(idx);
                                if (wklen < rperimlow || wklen > rperimhi)
                                {
                                        SetShapeDescriptorArray(REJECT);
                                        break;
                                }
                        }
                }
        }
}

int CCompoundImageObject::CheckVectorPairForCorner(int curridx)
{

CurveInfo       cvinf, cvinf2;
int                     size, totdeg = -1;
float           pctwk;

        CImageLib* pApp = (CImageLib *) AfxGetApp();
        pctwk = (float) pApp->UserDat.maxcornervectlenpctofperim;
        pctwk *= (float) 0.01; // Convert percent
        size    = m_curves.GetSize();
        cvinf = m_curves.GetAt(curridx);
        if (cvinf.m_rightdegreechg == 0 && cvinf.m_leftdegreechg == 0) return 0;
        int perim = MeasurePerimeter();

        if (curridx == 0)
        {
                cvinf2 = m_curves.GetAt(size - 1);
        }
        else
                cvinf2 = m_curves.GetAt(curridx - 1);

        if (cvinf2.m_length > (int) ((float) perim * pctwk)) return 0;

        if ((cvinf.m_rightdegreechg >= pApp->UserDat.curvelow &&
             cvinf.m_rightdegreechg <= pApp->UserDat.curvehi) ||
             (cvinf.m_leftdegreechg  >= pApp->UserDat.curvelow &&
              cvinf.m_leftdegreechg  <= pApp->UserDat.curvehi))
        {
                if (cvinf.m_leftdegreechg  > 0 && cvinf2.m_rightdegreechg  > 0)
                {
                        totdeg = cvinf.m_leftdegreechg - cvinf2.m_rightdegreechg;
                        if (totdeg < 0) totdeg *= (-1);
                }
                if (cvinf.m_rightdegreechg > 0 && cvinf2.m_leftdegreechg > 0)
                {
```

```
                totdeg = cvinf.m_rightdegreechg - cvinf2.m_leftdegreechg;
                if (totdeg < 0) totdeg *= (-1);
            }
            if (totdeg != -1)
            {
                if (totdeg >= pApp->UserDat.curvelow &&
                    totdeg <= pApp->UserDat.curvehi)
                    return 0;
                else
                    return 2;
            }
        }
        else
        {
            if (cvinf.m_leftdegreechg  > 0 && cvinf2.m_leftdegreechg  > 0)
            {
                totdeg = cvinf.m_leftdegreechg + cvinf2.m_leftdegreechg;
            }
            if (cvinf.m_rightdegreechg > 0 && cvinf2.m_rightdegreechg > 0)
            {
                totdeg = cvinf.m_rightdegreechg + cvinf2.m_rightdegreechg;
            }
            if (totdeg != -1)
            {
                if (totdeg >= pApp->UserDat.curvelow &&
                    totdeg <= pApp->UserDat.curvehi)
                    return 1;
            }
        }
        return 0;
}
```

Code Fragment 4: YACC Parser Generation Rules

```
%{
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define YYSTYPE char *        /* Input values are char strings */
#define YYERROR_VERBOSE
#define YYDEBUG 1

void            SetGroupSymbols(char *operatorfld, int newgrp);
void            EditTokensINP(int numtokens);
extern void     YaccOutput(char *data);
void            InsertInputs(char *input);

char            operators[500] = "", token1[40], token2[40], outdata[540], inputs[500] =
"";
char            outputname[40];
int                 insertflag = 0, opidx, lenoperators;
%}

%token      OUTTOK OR XOR NAN INP NOR NOT AND
%%

statement:      outexpr toklist3 expression
                {
                    int idx;
                    strcpy(outdata, "Z");
                    strcat(outdata, outputname + 4);
                    strcat(outdata, "(");
                    for (idx = 0; idx < (int)     strlen(inputs); idx++)
                    {
                        strncat(outdata, &inputs[idx], 1);
                        if (idx < (int) strlen(inputs) - 1)
```

```
                        strcat(outdata, ",");
                }
                strcat(outdata, ") = ");
                strcat(outdata, operators);
                YaccOutput(outdata);
                operators[0] = '\0';
                inputs[0]    = '\0';
                insertflag   = 0;
        }                                       |

        outexpr toklist3
        {
                int idx;
                strcpy(outdata, "Z");
                strcat(outdata, outputname + 4);
                strcat(outdata, "(");
                for (idx = 0; idx < (int) strlen(inputs); idx++)
                {
                        strncat(outdata, &inputs[idx], 1);
                        if (idx < (int) strlen(inputs) - 1)
                                strcat(outdata, ",");
                }
                strcat(outdata, ") = ");
                EditTokensINP(1);
                strcat(outdata, token1);
                YaccOutput(outdata);
                operators[0] = '\0';
                inputs[0]    = '\0';
                insertflag   = 0;
        };

expression:     expression andgroup
                {
/*                      EditTokensINP(2);       */
                }                               |
                expression orgroup
                {
/*                      EditTokensINP(2); */
                }                               |
                expression xorgroup
                {
/*                      EditTokensINP(2); */
                }                               |
                expression norgroup
                {
/*                      EditTokensINP(2); */
                }                               |
                expression nangroup
                {
/*                      EditTokensINP(2); */
                }                               |
                expression notgroup
                {
/*                      EditTokensINP(1); */
                }                               |
                andgroup
                {
/*                      EditTokensINP(2); */
                }                               |
                orgroup
                {
/*                      EditTokensINP(2); */
                }                               |
                xorgroup
                {
/*                      EditTokensINP(2); */
                }                               |
                norgroup
                {
/*                      EditTokensINP(2); */
                }                               |
                nangroup
```

```
                {
/*                      EditTokensINP(2); */
                }                                       |
                notgroup
                {
/*                      EditTokensINP(1); */
                }                                       |
                INP
                {
                        InsertInputs($1 + 4);
                };

andgroup:       AND toklist1 toklist2
                {
                        if (!insertflag)
                                SetGroupSymbols("AND", 0);
                        SetGroupSymbols(token1, 0);
                        SetGroupSymbols(token2, 1);
                };

orgroup:        OR      toklist1 toklist2
                {
                        if (!insertflag)
                                SetGroupSymbols("OR ", 0);
                        SetGroupSymbols(token1, 0);
                        SetGroupSymbols(token2, 1);
                };

xorgroup:       XOR toklist1 toklist2
                {
                        if (!insertflag)
                                SetGroupSymbols("XOR", 0);
                        SetGroupSymbols(token1, 0);
                        SetGroupSymbols(token2, 1);
                };

norgroup:       NOR toklist1 toklist2
                {
                        if (!insertflag)
                                SetGroupSymbols("NOR", 0);
                        SetGroupSymbols(token1, 0);
                        SetGroupSymbols(token2, 1);
                };

nangroup:       NAN toklist1 toklist2
                {
                        if (!insertflag)
                                SetGroupSymbols("NAN", 0);
                        SetGroupSymbols(token1, 0);
                        SetGroupSymbols(token2, 1);
                };

notgroup:       NOT toklist1
                {
                        if (!insertflag)
                                SetGroupSymbols("NOT", 0);
                        SetGroupSymbols(token1, 1);
                };

toklist1:       AND
                {
                        strcpy(token1, $1);
                }                                       |
                OR
                {
                        strcpy(token1, $1);
                }                                       |
                NOT
                {
                        strcpy(token1, $1);
                }                                       |
                XOR
```

```
                {
                        strcpy(token1, $1);
                }                                               |
                NOR
                {
                        strcpy(token1, $1);
                }                                               |
                NAN
                {
                        strcpy(token1, $1);
                }                                               |
                INP
                {
                        strcpy(token1, $1);
                        InsertInputs($1 + 4);
                };
toklist2:       AND
                {
                        strcpy(token2, $1);
                }                                               |
                OR
                {
                        strcpy(token2, $1);
                }                                               |
                NOT
                {
                        strcpy(token2, $1);
                }                                               |
                XOR
                {
                        strcpy(token2, $1);
                }                                               |
                NOR
                {
                        strcpy(token2, $1);
                }                                               |
                NAN
                {
                        strcpy(token2, $1);
                }                                               |
                INP
                {
                        strcpy(token2, $1);
                        InsertInputs($1 + 4);
                };

toklist3:       AND
                {
                        strcpy(token1, $1);
                }                                               |
                OR
                {
                        strcpy(token1, $1);
                }                                               |
                NOT
                {
                        strcpy(token1, $1);
                }                                               |
                XOR
                {
                        strcpy(token1, $1);
                }                                               |
                NOR
                {
                        strcpy(token1, $1);
                }                                               |
                NAN
                {
                        strcpy(token1, $1);
                }                                               |
                INP
                {
```

```
                        strcpy(token1, $1);
                        InsertInputs($1 + 4);
                };

outexpr:        OUTTOK
                {
                        strcpy(outputname, $1);
                };
%%

void SetGroupSymbols(char *operatorfld, int newgrp)
{

int             idx;
char    work[500], data[40];

        if (!strncmp(operatorfld, "INP", 3))
                strcpy(data, operatorfld + 4);
        if (!strcmp(operatorfld, "AND"))
                strcpy(data, "(&*&)");
        if (!strcmp(operatorfld, "OR "))
                strcpy(data, "(&+&)");
        if (!strcmp(operatorfld, "XOR"))
                strcpy(data, "(&(+)&)");
        if (!strcmp(operatorfld, "NOR"))
                strcpy(data, "(&+&)'");
        if (!strcmp(operatorfld, "NAN"))
                strcpy(data, "(&*&)'");
        if (!strcmp(operatorfld, "NOT"))
                strcpy(data, "(&)'");

        if (!insertflag)
        {
                insertflag = 1;
                strcpy(operators, data);
                lenoperators = strlen(operators);
                opidx = 0;
        }
        else
        {
                for (idx = opidx; idx < lenoperators; idx++)
                {
                        if (operators[idx] == '&')
                        {
                                break;
                        }
                }
                if (idx < lenoperators)
                {
                        strcpy(work, &operators[idx + 1]);
                        strcpy(&operators[idx], data);
                        strcpy(&operators[idx + strlen(data)], work);
                        opidx = idx + strlen(data);
                        lenoperators = strlen(operators);

                        for (idx = opidx; idx < lenoperators; idx++)
                        {
                                if (operators[idx] == '&') break;
                        }
                        if (idx == lenoperators)
                        {
                                opidx = 0;
                        }
                }
                else
                        strcpy(operators, "GENERATION ERROR");
        }

        if (newgrp)
                opidx = 0;
}
```

```
void EditTokensINP(int numtokens)
{
        if (strncmp(token1, "INP", 3))
                YaccOutput("ERROR - Could not find path from output back to input");
        if (numtokens == 2)
        {
                if (strncmp(token2, "INP", 3))
                YaccOutput("ERROR - Could not find path from output back to input");
        }
}

void InsertInputs(char *input)
{

char    work[500];
int             idx, len;

        len = strlen(inputs);
        for (idx = 0; idx < len; idx++)
        {
                if (!strncmp(input, &inputs[idx], 1)) break;
        }
        if (idx == len)
        {
                for (idx = 0; idx < len; idx++)
                {
                        if (inputs[idx] > input[0]) break;
                }
                if (idx < len)
                {
                        strcpy(work, &inputs[idx]);
                        strncpy(&inputs[idx], input, 1);
                        strcpy(&inputs[idx + 1], work);
                }
                else
                        strncat(inputs, input, 1);
        }
}
```

APPENDIX B

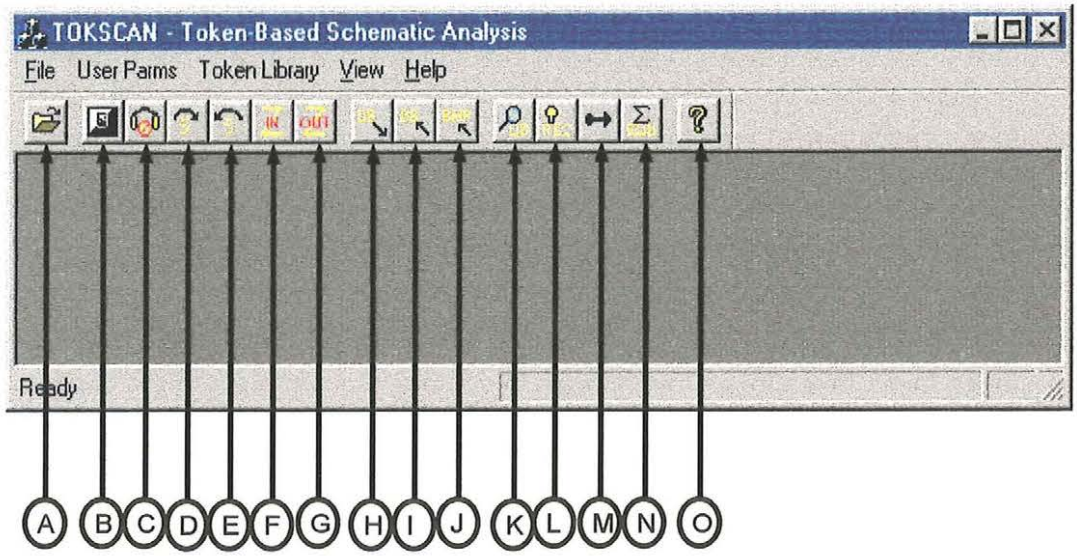
User Manual For The TOKSCAN Program

This user manual has two parts: 1) A description of the graphical user interface, which explains the operation of the various visual controls, and 2) Operational instructions for processing the test images provided with the program. The Zip Disk which comes with this paper contains the set of test image files as well as the program executables and supporting data files.

The Graphical User Interface

The user controls in each window and dialog box are shown in the next few pages, with an explanation of the function provided.

1. Main Application Window - Toolbar



A. Open new file.

A file-open dialog box is displayed, where the user can select an image file to be opened and displayed. The file must be in Windows BMP format.

B. Perform Binarization

A dialog box is displayed which allows the user to select a binarization global threshold value in the range 0 - 255, where 0 is pure black, 255 is pure white, and every other value is a shade of gray in between. From the dialog box, the user can apply binarization to the image using the selected grayscale value.

C. Perform Noise Reduction

A dialog box is displayed which allows the user to select a threshold value for the noise reduction operation, and to apply noise reduction to the image. A "window size" can also be selected, which specifies the number of pixels which surround a pixel of interest that are used to calculate an average grayscale value for the pixel of interest. The average value is compared with the selected

threshold value, and the color of the pixel of interest is set to either pure white or pure black accordingly.

D. Rotate Image 5 Degrees To Right

The displayed image is rotated clockwise by 5 degrees. Menu options are available which allow the user to rotate the image by incrementally larger amounts.

E. Rotate Image 5 Degrees To Left

The displayed image is rotated counterclockwise by 5 degrees. Menu options are available which allow the user to rotate the image by incrementally larger amounts.

F. Zoom Image In By 5 Percent

The displayed image is magnified by 5 percent, and the total size of the bitmap is increased by 5 percent. Menu options are available which allow the user zoom in by incrementally larger amounts.

G. Zoom Image Out By 5 Percent

The displayed image is compressed in size by 5 percent. Menu options are available which allow the user zoom out by incrementally larger amounts.

H. Load Image Vectors From Database Into Memory

Vectors which describe the currently displayed image, which were previously created by TOKSCAN and saved in the database, are loaded into memory for processing. This function is provided in order to avoid having to perform vectorization every time an image is loaded for recognition. A file-open dialog box is displayed, so the user can select a "pre-thinned" binary image for processing. The original image must have already been thinned, and the result saved in the file which is opened at this point.

I. Save Image Vectors From Memory Into Database

Vectors which describe the currently displayed image, which have just been created by the vectorization process, and which currently reside in memory, are saved in the database. They may be reloaded into memory, as described in H.

J. Save Image File To Disk In Windows BMP File Format

A file-save dialog box is displayed, which allows the user to save the displayed image in Windows BMP file format on a hard disk or diskette.

K. Perform Token Library Analysis

Pressing this button causes TOKSCAN to perform image recognition on the displayed image up to the point of detecting closed minimal polygons, and generating the shape tokens which describe the polygons. If the image has already been thinned, and if vectorization has already been performed, the program will go immediately into polygon detection; otherwise it will perform thinning and vectorization as needed. It is possible to "pre-thin" and "pre-vectorize" an image, and save the results on disk. Then, the thinned image and the accompanying set of vectors can be reloaded at a later time, and image recognition can be performed without having to go through thinning and vectorization again. In the second section of this manual, instructions are given for processing the images provided with the software. Specific directions are provided for "pre-thinning" and "pre-vectorization".

L. Perform Full Image Recognition

Pressing this button causes TOKSCAN to perform full image recognition on the displayed image. Components are located and labeled, and connecting signal lines and circular line connectors are located. After this function completes execution, the image is ready for the user to specify circuit input and output points, and to request generation of the equation.

M. Toggle Mouse Cursor Between Standard Cursor And Circuit Input/Output Indicator

The user specifies the location of all input signals to the circuit, and all output signals from the circuit, by selecting line end points with the mouse. To do this, the user must change the mouse cursor from its standard form to an input/output indicator form. This toolbar button toggles the mouse cursor between standard form and input/output indicator form.

N. Generate Equation For Recognized Schematic Image

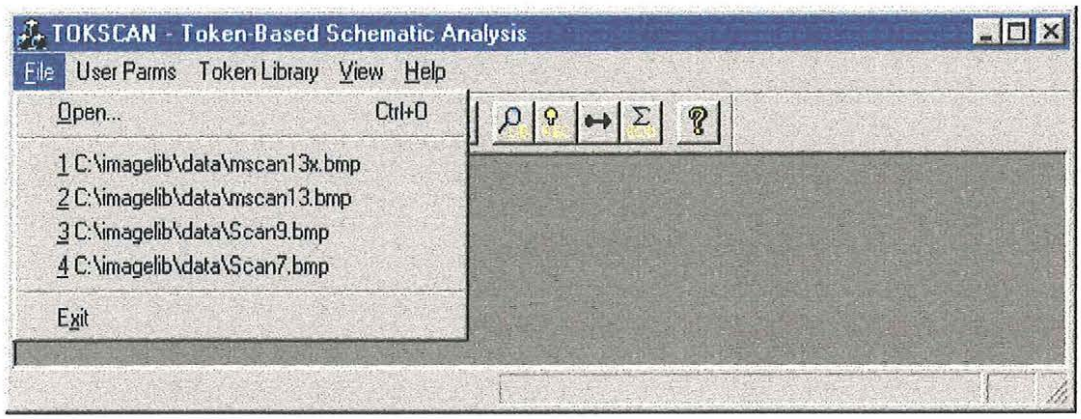
The user presses this button after full image recognition has been completed, and after all circuit input and output points have been identified (manually).

TOKSCAN follows all connecting signal lines from input to final output, and generates one or more logic equations which describe the recognized circuit.

O. Display The "ABOUT" Dialog Box

The user presses this button to display a dialog box with information about TOKSCAN.

1. Main Application Window - Menu Selections



File Menu

A. Open

A file-open dialog box is displayed, where the user can select an image file to be opened and displayed. The file must be in Windows BMP format.

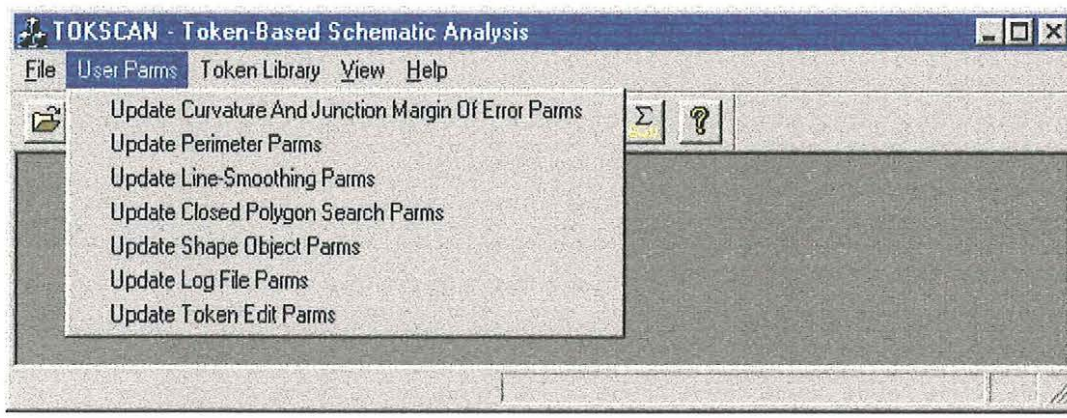
B. Recent File List

If a file from this list is selected (a recently opened image file), it will be opened without displaying the file-open dialog box.

C. Exit

The application closes.

1. Main Application Window - Menu Selections Continued

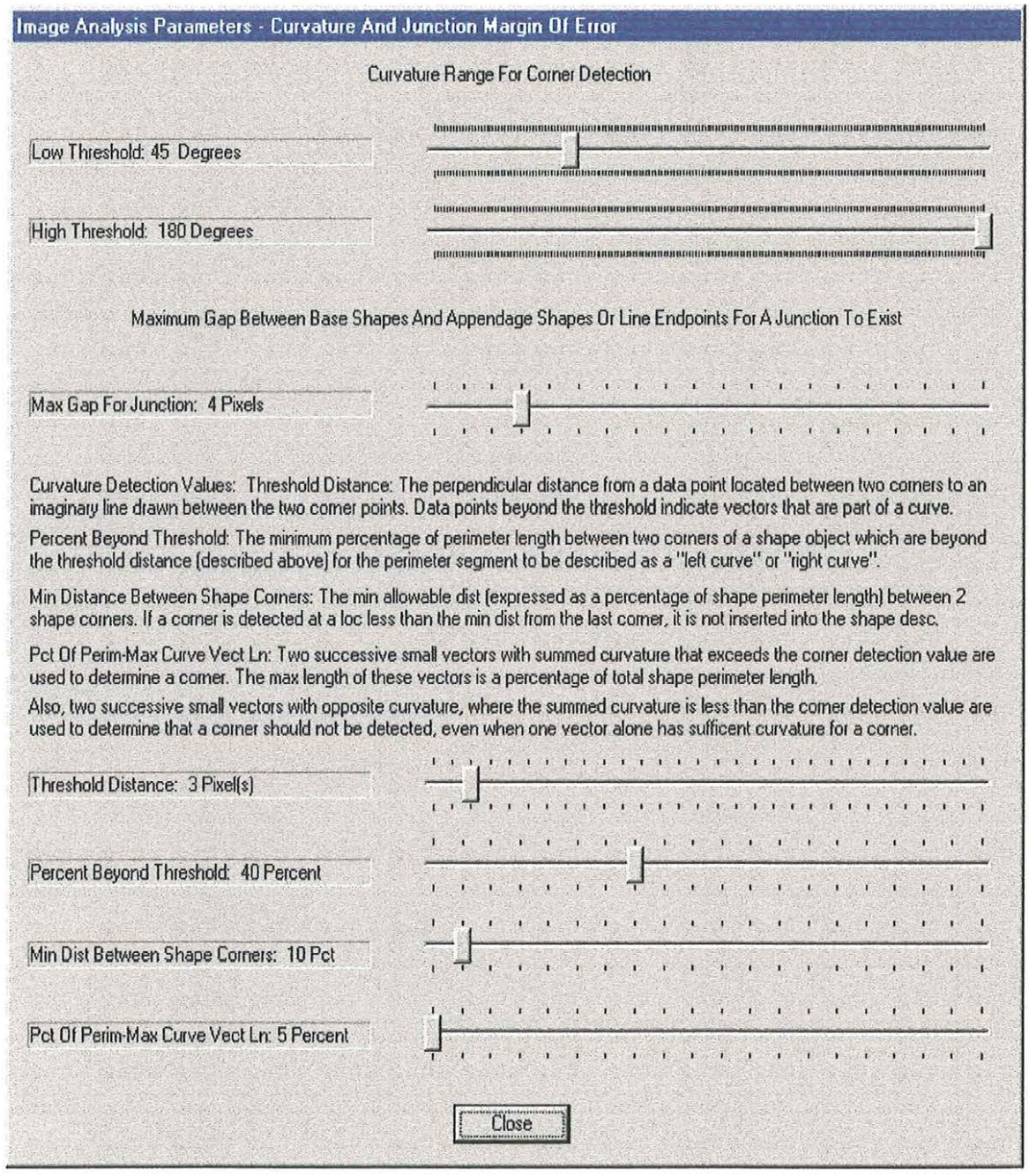


User Parms Menu

Each item in this menu opens a dialog box which provides access to a set of user-adjustable parameters which influence or monitor the image recognition process. In all cases except for the line-smoothing parameters and log file parameters, slider controls are used to allow the user to pick a valid numeric value from a given range. In the line-smoothing and log file dialog boxes, the user types data into a set of edit controls. Each parameter from all of these dialog boxes (except for the log file parameters) is explained in detail in section 2.15 - "User-Adjustable Parameters For The Recognition Process", and the explanations are grouped in order by dialog box.

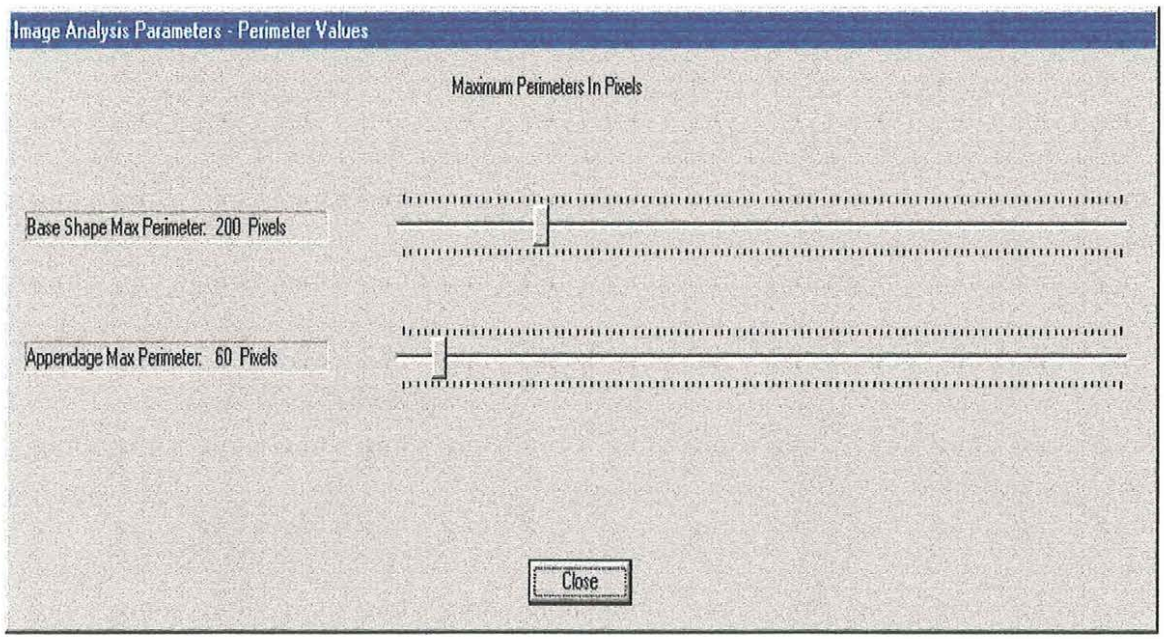
An example of each dialog box is shown in the following pages.

Curvature And Junction Margin Of Error Dialog



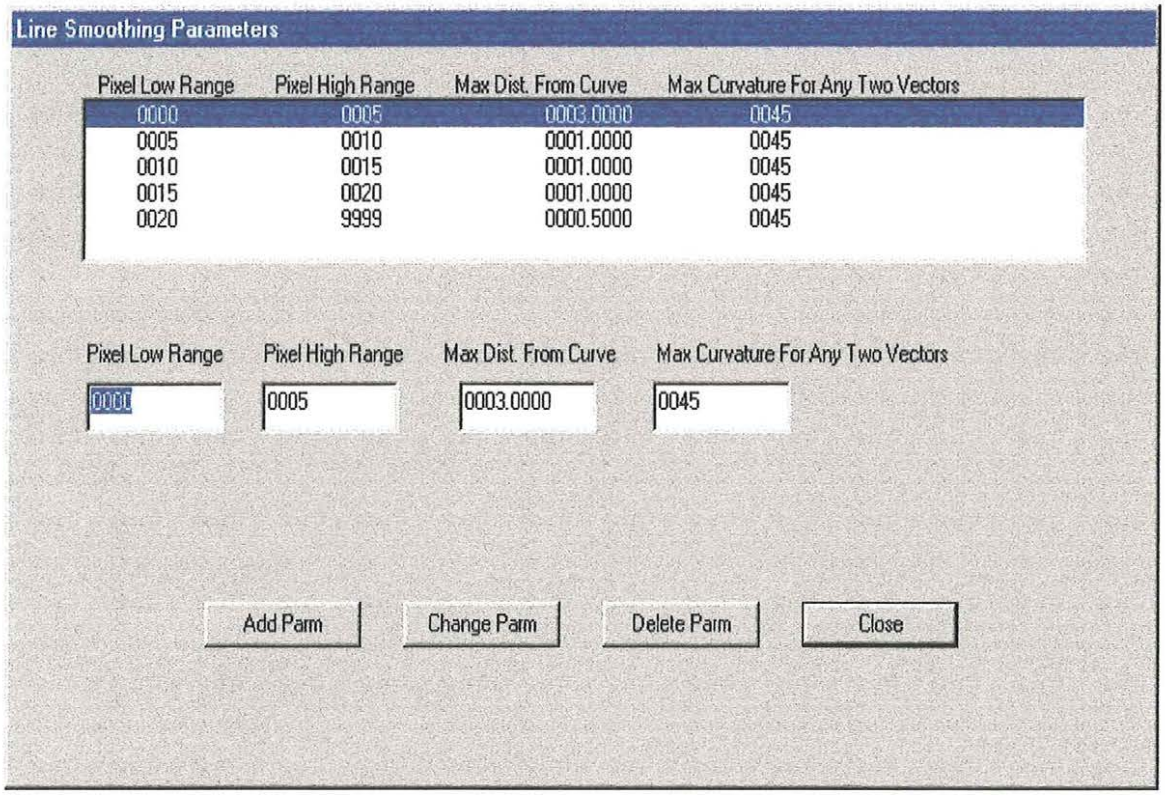
The parameters in this dialog box have to do with the threshold curvature for corner detection, the maximum distance between shapes in order for them to be linked together into a compound shape, and the threshold values for shape token generation. See section 2.15 for a detailed explanation of each parameter.

Perimeter Values Dialog



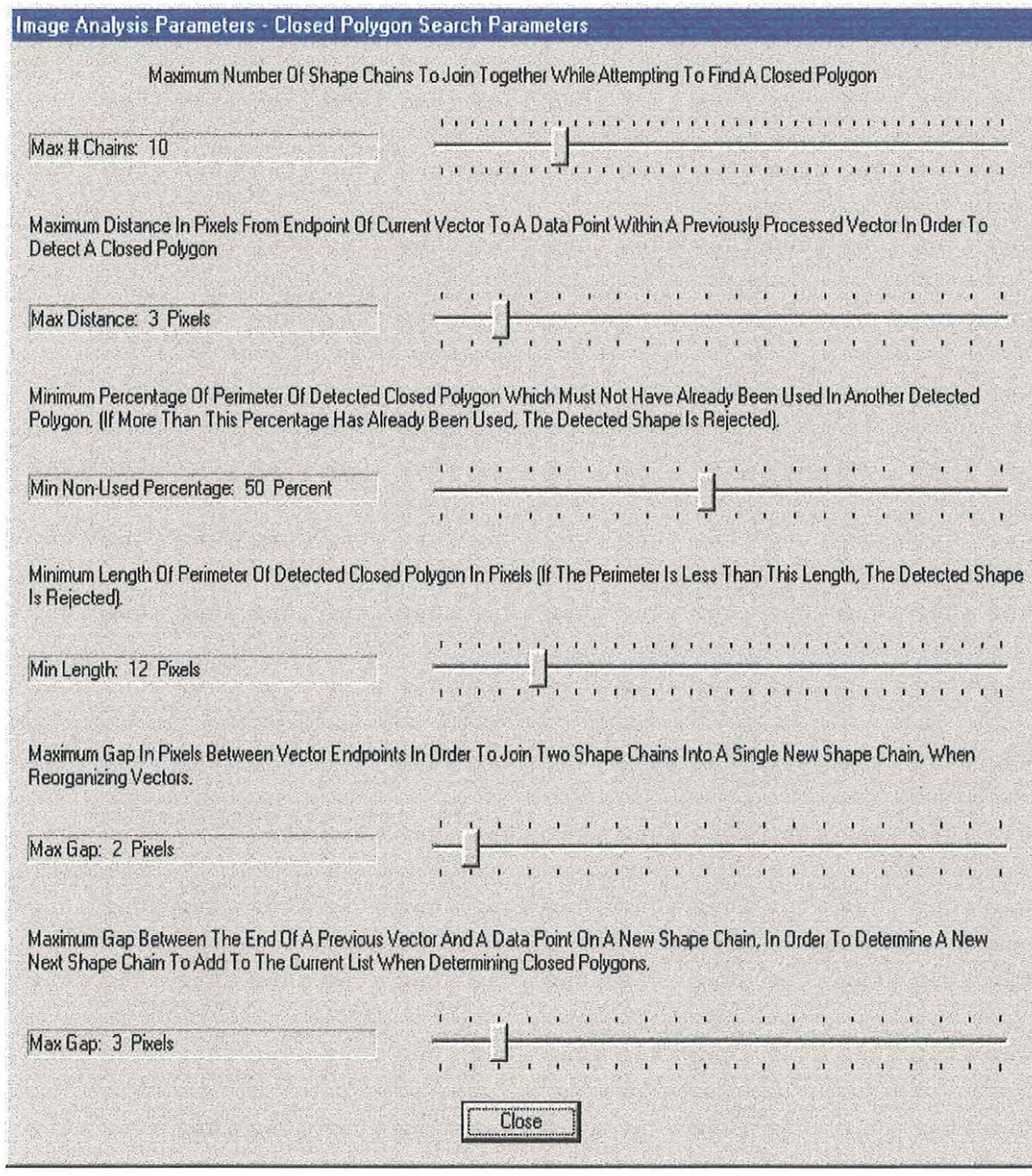
The parameters in this dialog box have to do with the maximum perimeter lengths for base shapes and appendage shapes. See section 2.15 for a detailed explanation of each parameter.

Line Smoothing Parameters Dialog



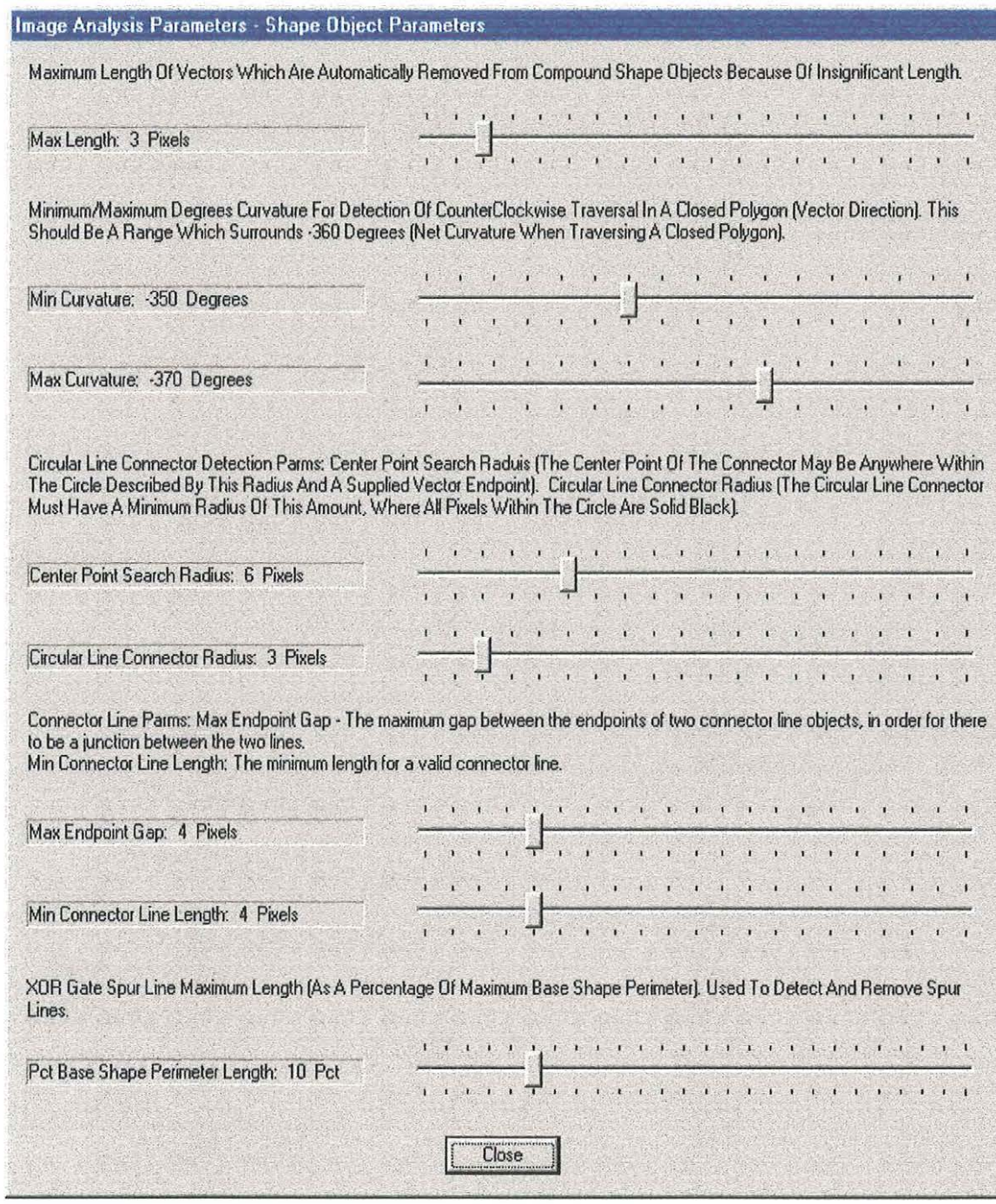
The parameters in this dialog box have to do with line smoothing during the vectorization process. See section 2.15 for a detailed explanation of each parameter.

Closed Polygon Search Parameters Dialog



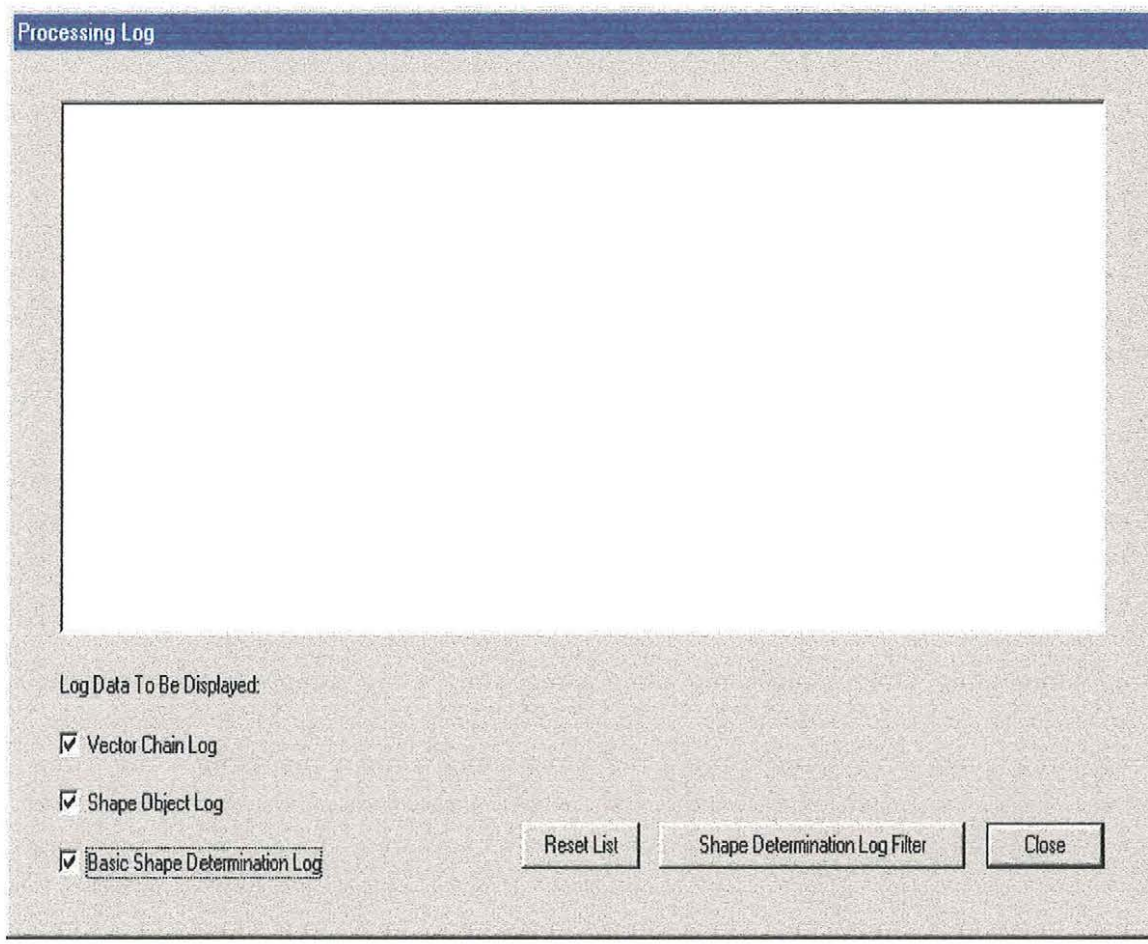
The parameters in this dialog box have to do with the detection of closed minimal polygons. See section 2.15 for a detailed explanation of each parameter.

Shape Object Parameters Dialog



The parameters in this dialog box have to do with the processing of the various shape objects which are created during the recognition process. See section 2.15 for a detailed explanation of each parameter.

Processing Log Dialog



The Processing Log dialog differs from the other user parameter dialogs, in that the user sets values which control the display of a processing log which monitors the image recognition process. In the lower left corner, the user can select the kinds of messages which should be included in the log from any of three categories:

Vector Chain Log - lists the vectors which describe the simple shape objects selected by the shape determination log filter.

Shape Object Log - lists information about compound shape objects selected by the shape determination log filter.

Basic Shape Determination Log - lists processing information during the detection of closed minimal polygons.

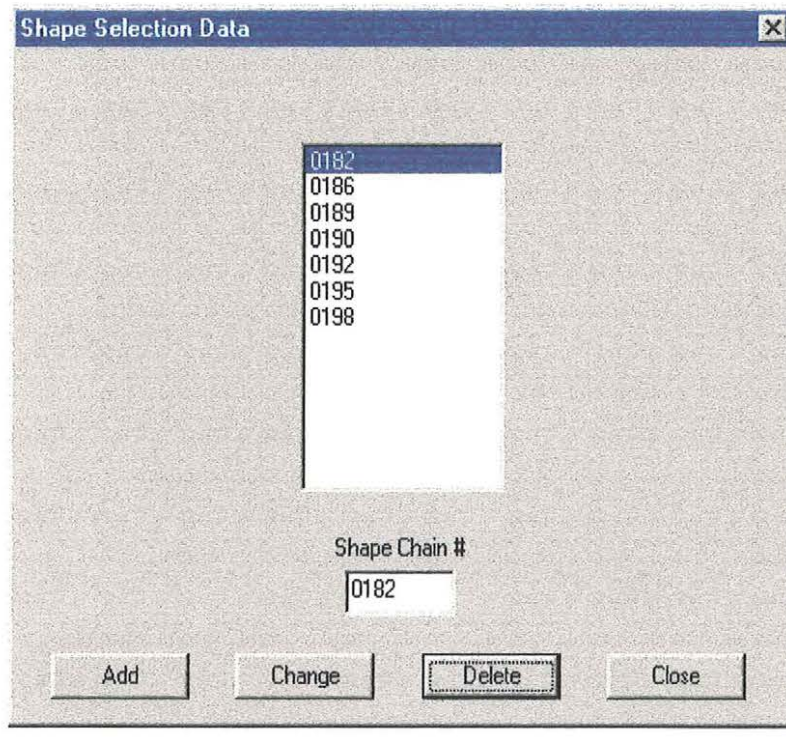
The processing log is used for debugging purposes to gather detailed information about the program execution when an image is not recognized properly.

Simple and compound shape objects can be selected for log information by entering their identifying chain number in one of two ways: 1) They may be entered manually by pressing the "Shape Determination Log Filter" button, and typing in the proper chain numbers. 2) Objects in the image may be selected directly from the image by using the mouse. When an object is selected, a dialog box opens which provides recognition information about the object, including its identifying chain number. The user can either type in the displayed chain number (#1 above), or double click on the chain number displayed in the object information dialog to have it automatically inserted into the list of object numbers for which log information is being collected. To select an object with the mouse, the mouse cursor should be placed near the desired object, and the left mouse button should be held down, while moving the mouse toward the object. A selection rectangle will be drawn in the window, and any objects which fall within that rectangle will be selected and displayed in the dialog box.

There are two different dialog boxes which may be displayed when an object is selected, one for simple shape objects, and the other for compound shape objects. The simple shape object dialog is displayed when an object is selected before Library Analysis or Image Recognition is run. The compound shape object dialog is displayed when an object is selected after Library Analysis or Image Recognition is run.

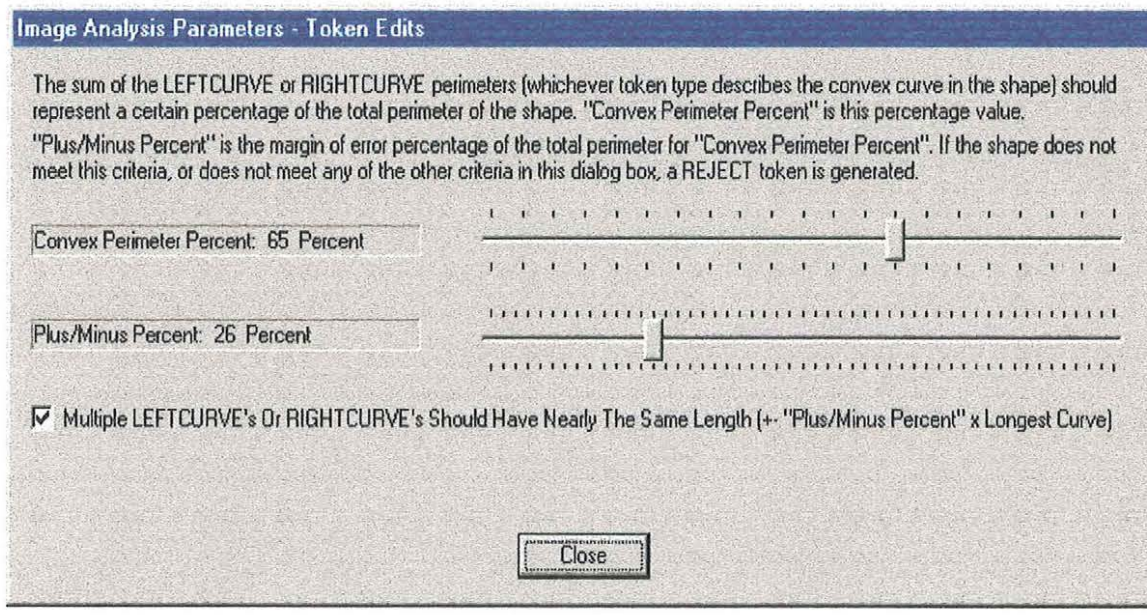
These two dialog boxes are shown and described later in this manual.

Processing Log Dialog

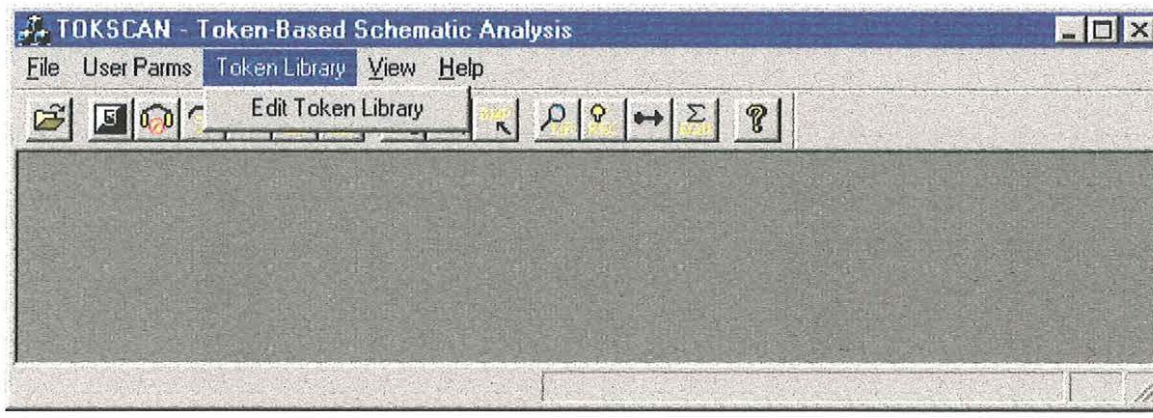


This dialog box is displayed when the "Shape Determination Log Filter" button is pressed in the Process Log dialog box. It allows the entry of identifying chain numbers for shape objects in order to display program execution information about the objects in the process log. The list may be updated manually using this dialog, or entries may be added by direct selection from the image, as described above in the "Process Log Dialog" information.

Token Edits Dialog



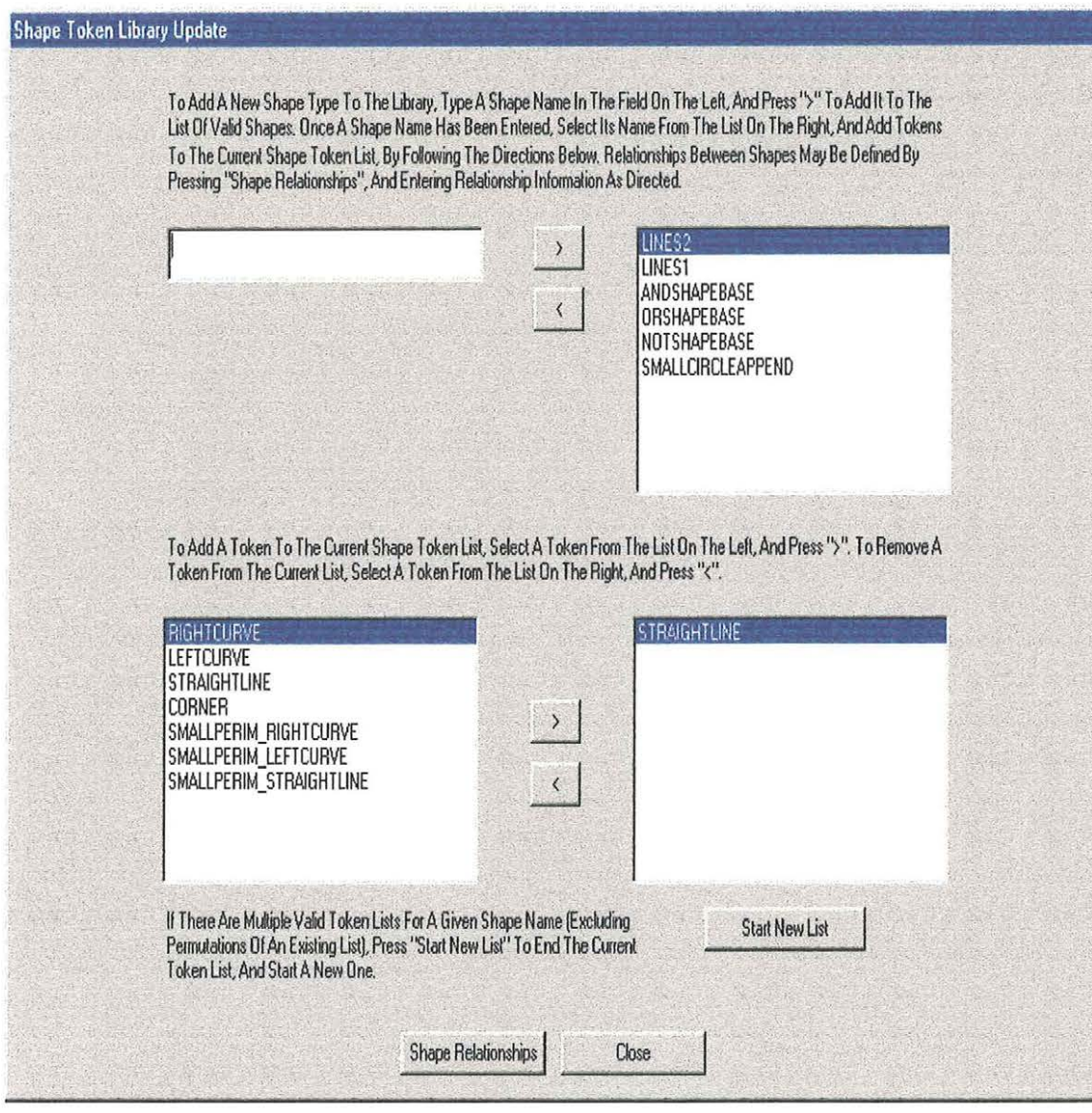
The parameters in this dialog box have to do with distinguishing between valid component shapes and invalid shapes formed from crossed signal connector lines. See section 2.15 for a detailed explanation of each parameter.



Token Library Menu

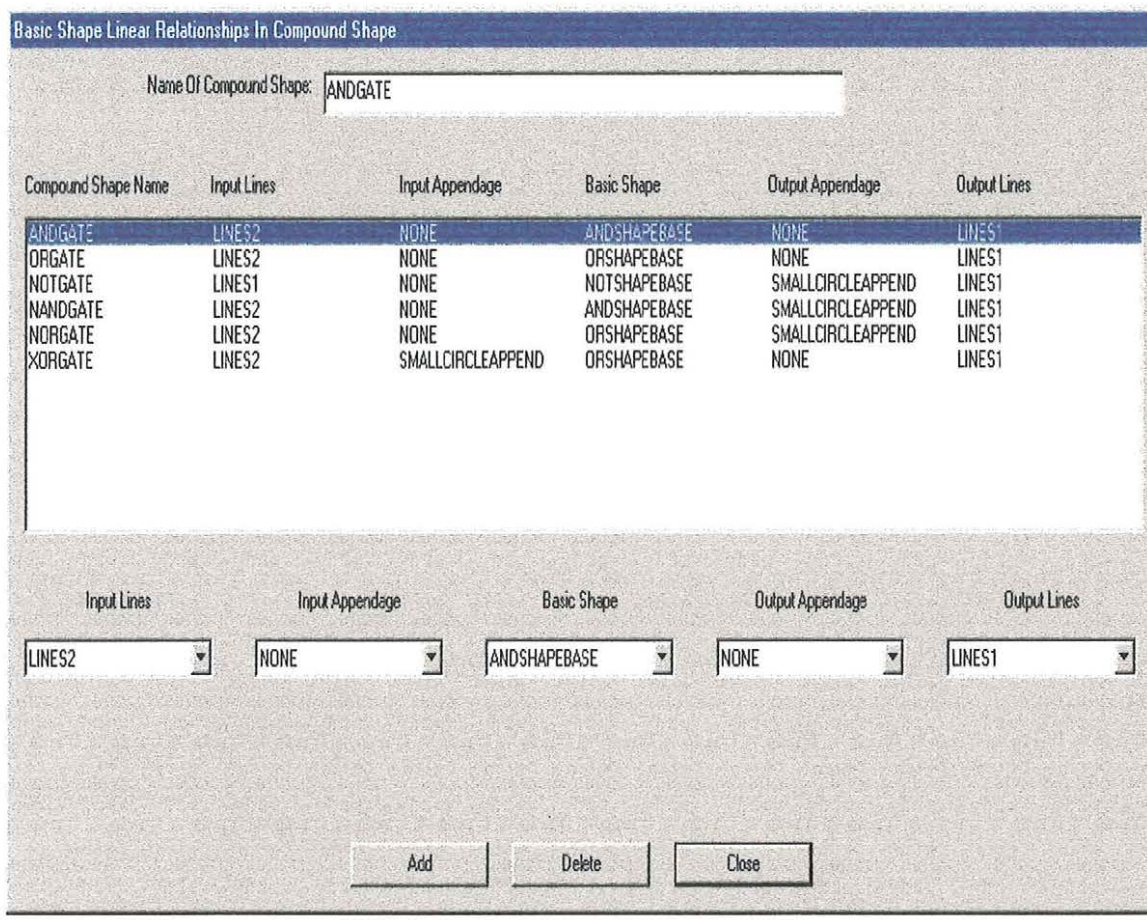
The EDIT TOKEN LIBRARY selection displays a dialog box which allows descriptive token lists to be added or updated in the token library. The dialog box is documented on the following page.

Shape Token Library Update Dialog



This dialog box is used to update the shape token library with new token lists. Lists may be added or removed, by following the directions shown in the dialog box. Token lists may also be added to the library by using the "Compound Shape Object Information" Dialog, which is described later in this manual.

Basic Shape Linear Relationships In Compound Shape Dialog



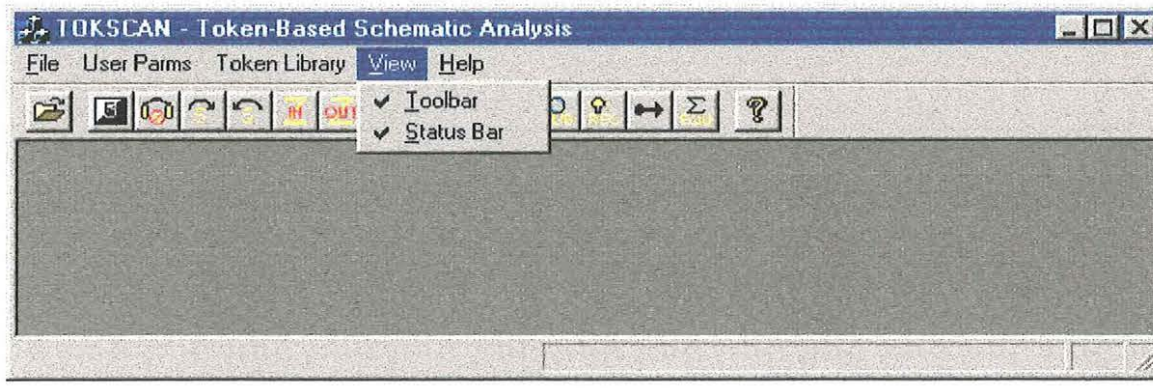
This dialog is displayed by pressing the "Shape Relationships" button in the "Shape Token Library Update" dialog box. It allows the user to specify the relationship between the simple shapes which make up the more complex schematic component. As an example of how this is used, look at the first line of data shown above in the dialog box.

The compound shape, named "ANDGATE" is a schematic component, and when a successful match is made with an AND logic gate in a schematic image, the component will be labeled as an "ANDGATE". Reading from left to right, the data line which describes the ANDGATE may be interpreted as follows: the ANDGATE has 2 input lines, no input appendage, it uses the AND base shape, it has no output appendage, and it has one output line. All of these simple shapes are arranged in the linear order given by reading left to right.

A new compound shape can be added by entering a name in the "Name Of Compound Shape" edit control, and then selecting appropriate entries from each of the "drop-down combo-boxes" below it, and pressing the ADD button. Deletes

may be done by selecting an existing entry, and pressing the DELETE button. Note that if a new shape is added, corresponding "hard coding" will have to be added to the source code in the current version of TOKSCAN to generate new token types for the parser, so that the new shape is included in the resulting equation. See Appendix B for more information about hard coding.

1. Main Application Window - Menu Selections Continued



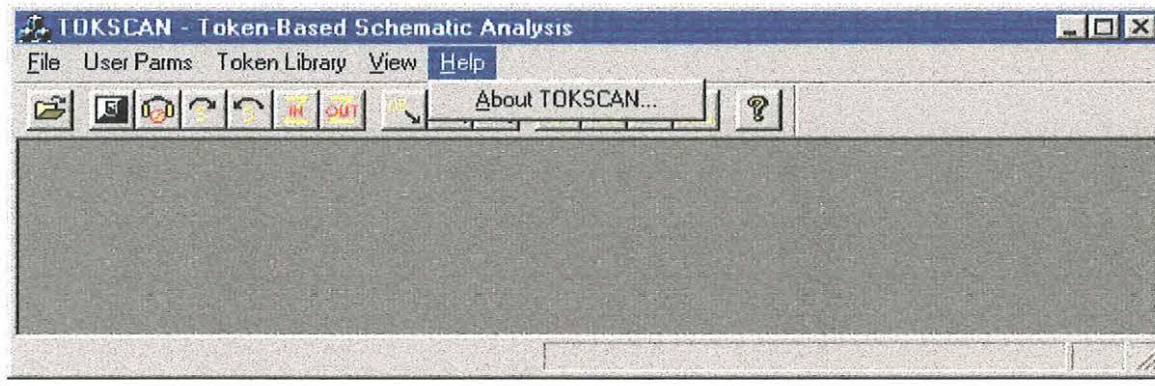
View Menu

A. Toolbar

Selecting this item toggles the appearance of the toolbar, making it appear or disappear from the window.

B. Status Bar

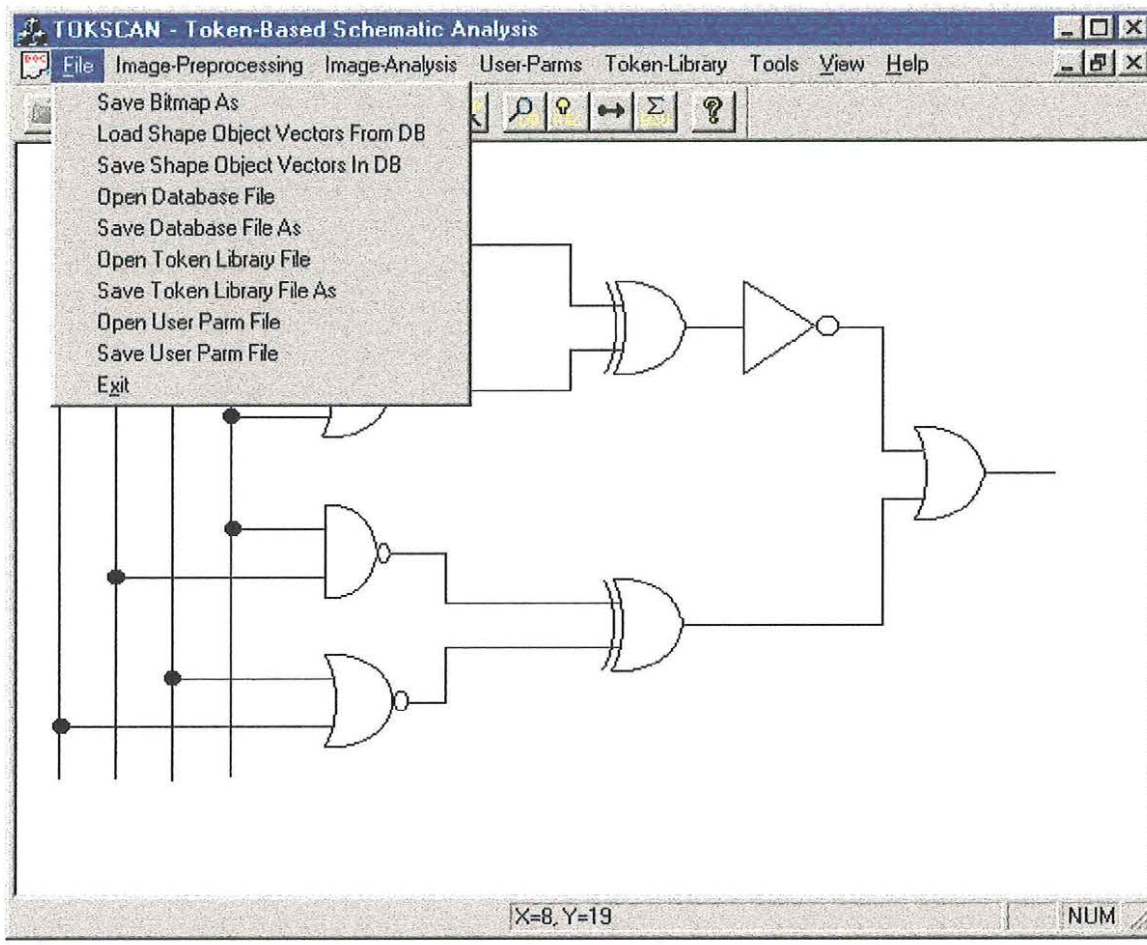
Selecting this item toggles the appearance of the status bar at the bottom of the window, making it appear or disappear.



Help Menu

Selecting About TOKSCAN displays a dialog box with information about the program.

2. Document Window - Menu Selections



A. Save Bitmap As

A file-save dialog box is displayed which allows the displayed bitmap to be saved in Windows BMP file format.

B. Load Shape Object Vectors From DB

Vectors which describe the currently displayed image, which were previously created by TOKSCAN and saved in the database, are loaded into memory for processing. This function is provided in order to avoid having to perform vectorization every time an image is loaded for recognition. A file-open dialog box is displayed, so the user can select a "pre-thinned" binary image for processing. The original image must have already been thinned, and the result saved in the file which is opened at this point.

C. Save Shape Object Vectors In Db

Vectors which describe the currently displayed image, which have just been created by the vectorization process, and which currently reside in memory, are saved in the database. They may be reloaded into memory, as described in B.

D. Open Database File

TOKSCAN is capable of maintaining a library of database files, each of which contains the vectorization output for one image which has been processed by the program. When images are reprocessed, the appropriate database of vectors may be opened and transferred into memory, and recognition may be performed without having to perform vectorization again. A file-open dialog is displayed, which allows the user to select a .dbs file from the library.

E. Save Database File As

After vectorization has been completed on an image, the user can save the resulting database file into the library of vectorization files (described in D). A file-save dialog is displayed, which allows the user to save the database in the library with a .dbs file type.

F. Open Token Library File

TOKSCAN is capable of maintaining a library of token files. The token library for a specific image or group of images is saved in a single file, but TOKSCAN can handle multiple distinct token libraries in separate files. A file-open dialog is displayed which allows the user to open a token library file of type .tok.

G. Save Token Library File As

A token library may be saved in a single file, which may be added to a library of token files, each of which is a separate token library. The user may choose an appropriate token library for use with a specific image or group of images, as described in F. A file-save dialog is displayed which allows the user to save the token file in the library with a .tok file type.

H. Open User Parm File

TOKSCAN is capable of maintaining a library of user parameter files. Each parameter file contains one complete set of user parameters, which control the recognition process. Using this feature, the user can automatically set all of the user-adjustable parameters at one time to values which are appropriate for a

particular image. A file-open dialog is displayed, which allows the user to open a user parameter file of type .upr.

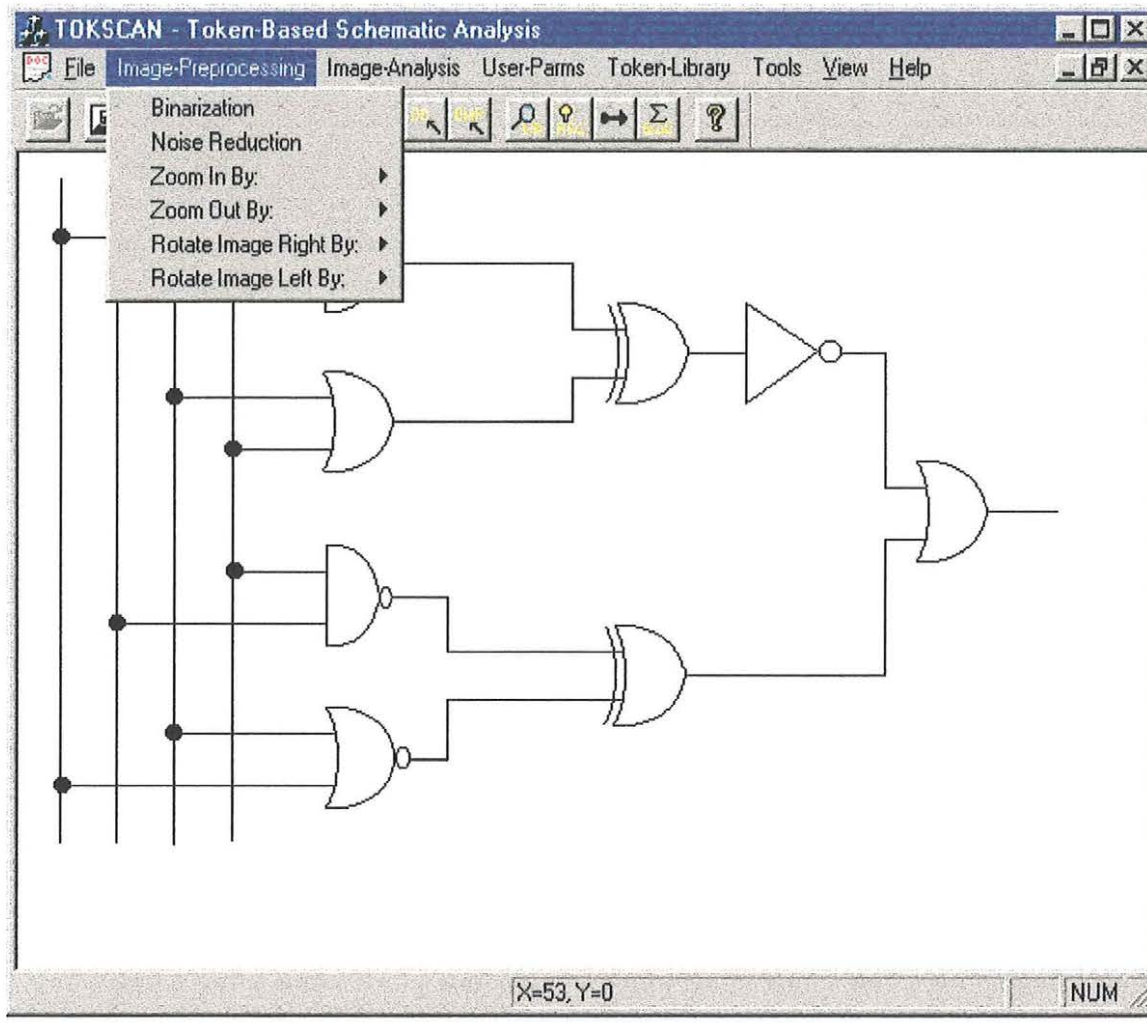
I. Save User Parm File

After the user sets the processing parameters for a particular image, they may be saved permanently in a user parameter file. This file can be opened later for use with any desired image. When it is opened, all of the adjustable parameters are set according to the values saved in it. TOKSCAN can save and open multiple user parm files from a library, so that a distinct parm file can be maintained for each image. A file-save dialog is displayed, which allows the user to save the parm file with file type .upr.

J. Exit

The application is closed.

2. Document Window - Menu Selections Continued



A. Binarization

A dialog box is displayed which allows the user to select a binarization global threshold value in the range 0 - 255, where 0 is pure black, 255 is pure white, and every other value is a shade of gray in between. From the dialog box, the user can apply binarization to the image using the selected grayscale value.

B. Noise Reduction

A dialog box is displayed which allows the user to select a threshold value for the noise reduction operation, and to apply noise reduction to the image. A "window size" can also be selected, which specifies the number of pixels which surround a pixel of interest that are used to calculate an average grayscale value for the pixel of interest. The average value is compared with the selected

threshold value, and the color of the pixel of interest is set to either pure white or pure black accordingly.

C. Zoom In By

When this menu item is selected, another menu is displayed which gives the user the option of several different "zoom percentage values". After a percentage value is selected, the displayed image is magnified by the selected percentage, and the total size of the bitmap is increased by that percentage.

D. Zoom Out By

When this menu item is selected, another menu is displayed which gives the user the option of several different "zoom percentage values". After a percentage value is selected, the displayed image is compressed by the selected percentage, and the total size of the bitmap is decreased by that percentage.

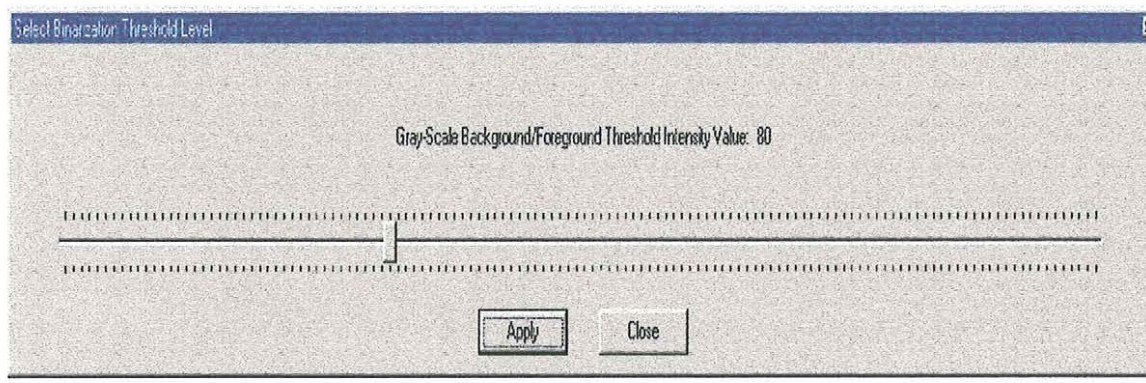
E. Rotate Image Right By

When this menu item is selected, another menu is displayed which gives the user the option of several different "rotation degree values". After a degree value is selected, the displayed image is rotated clockwise by the selected number of degrees.

F. Rotate Image Left By

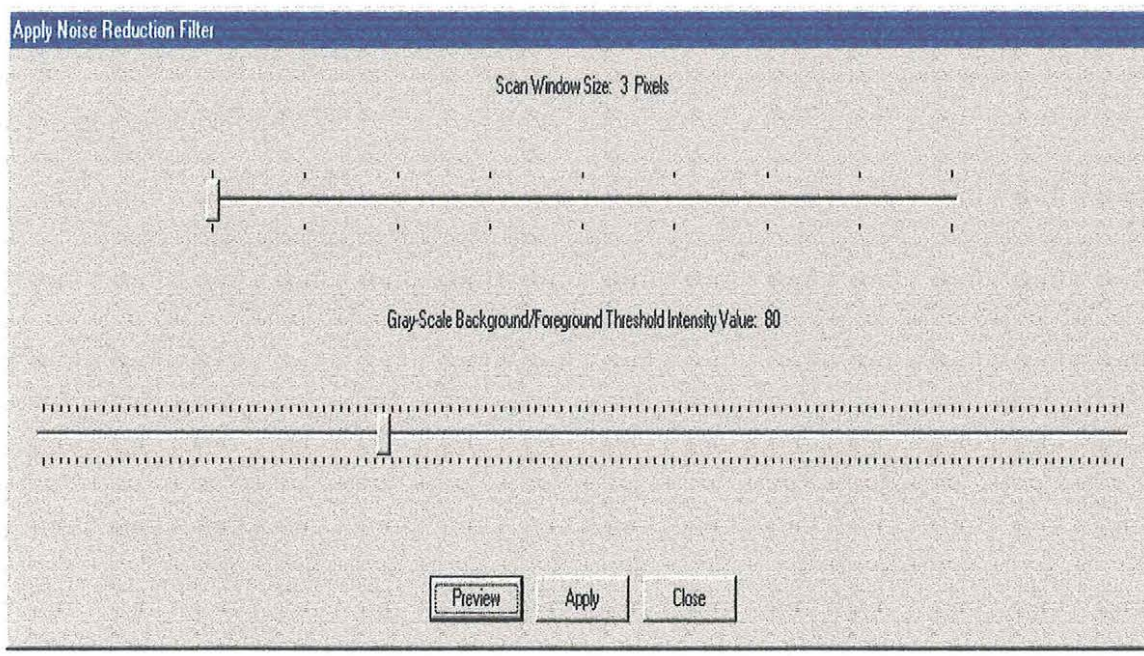
When this menu item is selected, another menu is displayed which gives the user the option of several different "rotation degree values". After a degree value is selected, the displayed image is rotated counterclockwise by the selected number of degrees.

Select Binarization Threshold Level Dialog



This dialog box is used to control the binarization process. The slider is used to set the global threshold to a value in the range 0 - 255, and it is applied to the image by pressing the APPLY button.

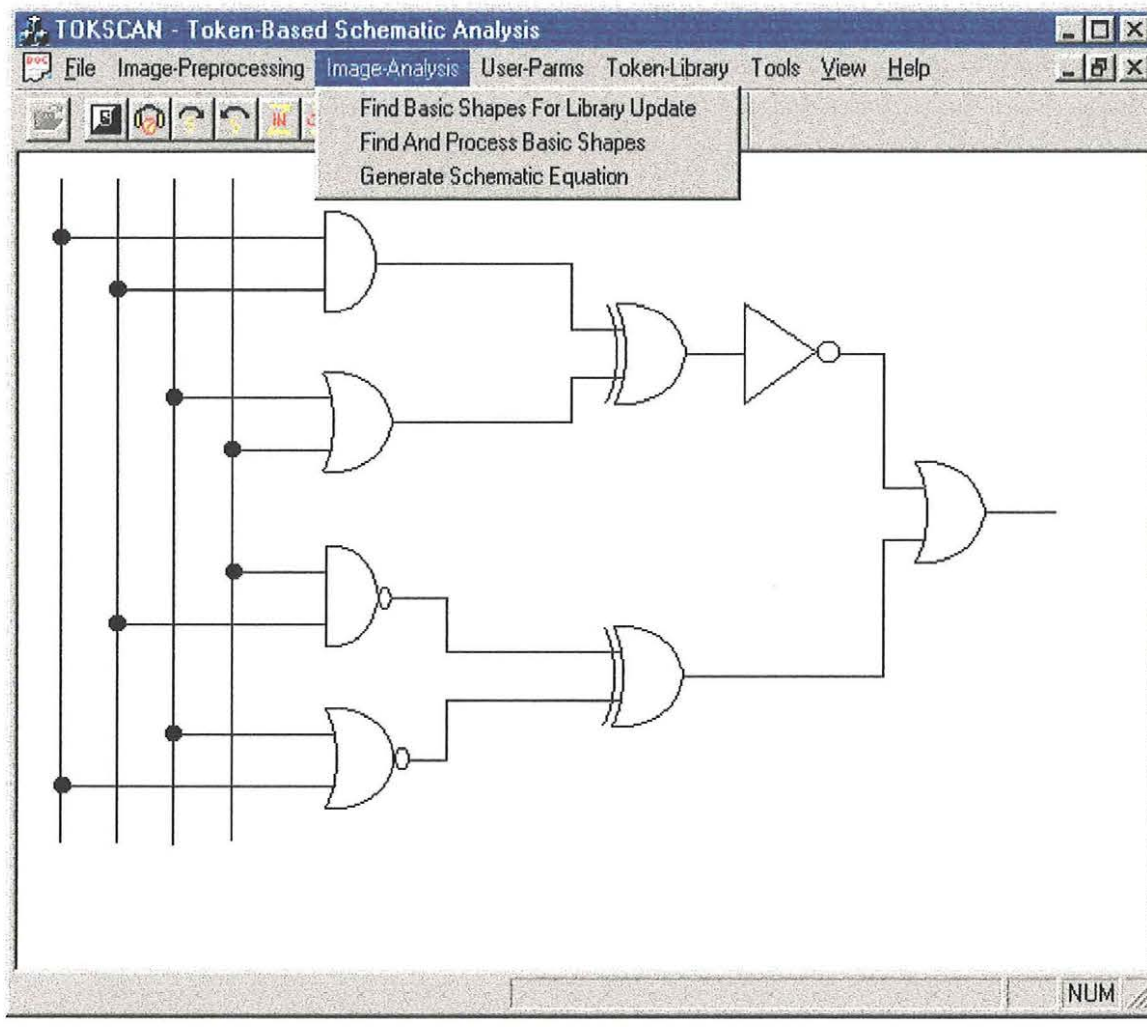
Apply Noise Reduction Filter Dialog



This dialog box is used to control the noise reduction process. The sliders are used to set a "window" size (of data around a pixel of interest), and a global threshold value that is compared with the average grayscale intensity in the window. The effects of the settings may be observed by pressing the PREVIEW button. Each time PREVIEW is pressed, the image reverts to its original state

before noise reduction is applied again. When the APPLY button is pressed, the last noise reduction performed is applied to the image for the duration of processing.

2. Document Window - Menu Selections Continued



NOTE: The USER-PARMS and TOKEN-LIBRARY menu items have exactly the same functionality as the corresponding menu items in the main application window, which have already been discussed. They are not presented again in this section.

A. Find Basic Shapes For Library Update

Making this selection causes TOKSCAN to perform image recognition on the displayed image up to the point of detecting closed minimal polygons, and generating the shape tokens which describe the polygons. If the image has already been thinned, and if vectorization has already been performed, the program will go immediately into polygon detection; otherwise it will perform thinning and vectorization as needed. It is possible to "pre-thin" and "pre-

vectorize" an image, and save the results on disk. Then, the thinned image and the accompanying set of vectors can be reloaded at a later time, and image recognition can be performed without having to go through thinning and vectorization again. In the second section of this manual, instructions are given for processing the images provided with the software. Specific directions are provided for "pre-thinning" and "pre-vectorization".

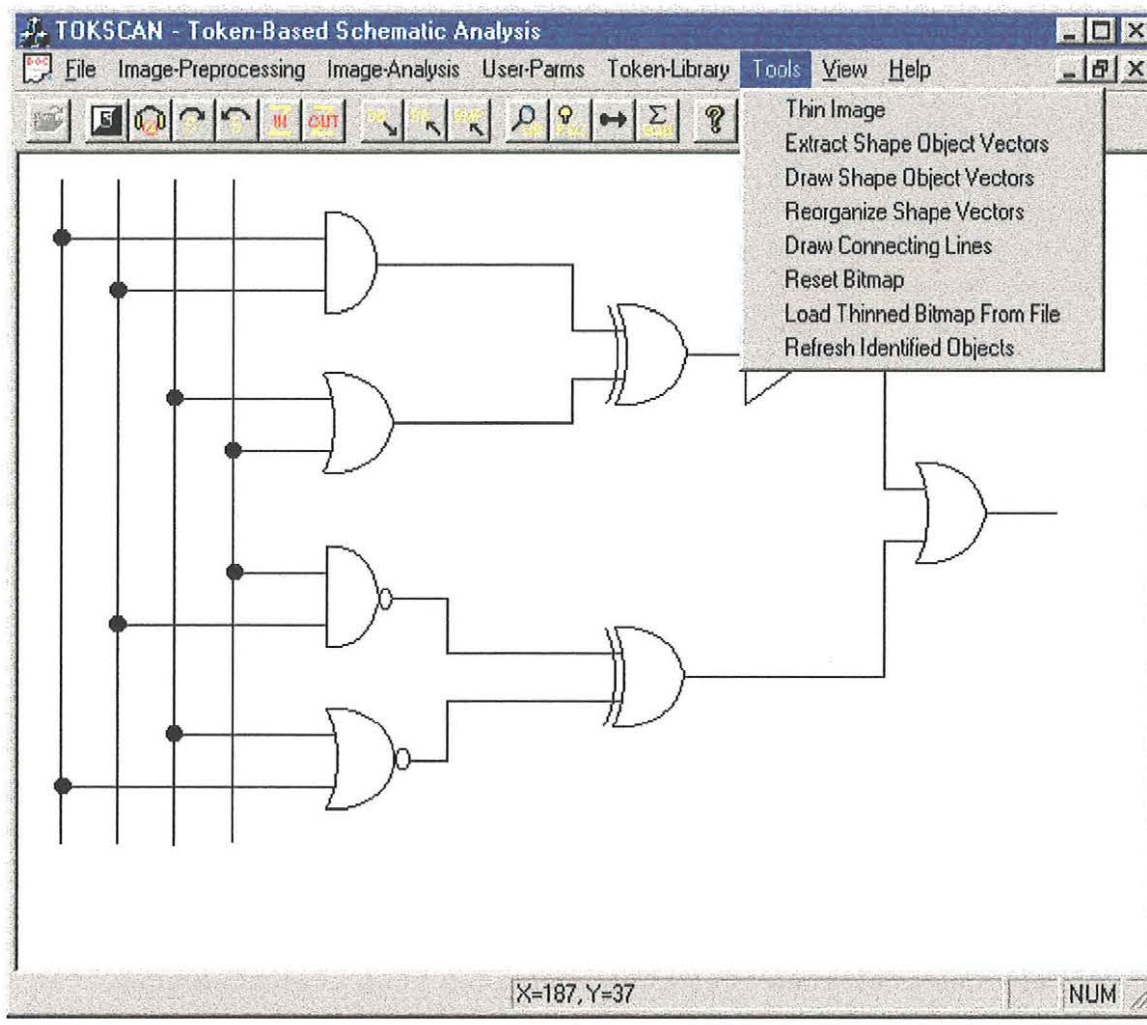
B. Find And Process Basic Shapes

Making this selection causes TOKSCAN to perform full image recognition on the displayed image. Components are located and labeled, and connecting signal lines and circular line connectors are located. After this function completes execution, the image is ready for the user to specify circuit input and output points, and to request generation of the equation.

C. Generate Schematic Equation

The user makes this selection after full image recognition has been completed, and after all circuit input and output points have been identified (manually). TOKSCAN follows all connecting signal lines from input to final output, and generates one or more logic equations which describe the recognized circuit.

2. Document Window - Menu Selections Continued



NOTE: The VIEW and HELP menu items have exactly the same functionality as the corresponding menu items in the main application window, which have already been discussed. They are not presented again in this section.

A. Thin Image

Making this selection causes TOKSCAN to perform thinning on the displayed image.

B. Extract Shape Object Vectors

Making this selection causes TOKSCAN to perform vectorization on the displayed image, and save the resulting vectors in the database.

C. Draw Shape Object Vectors

Making this selection causes TOKSCAN to draw all of the shape object vectors which were placed in simple shape objects as a result of the vectorization process.

D. Reorganize Shape Vectors

Making this selection causes TOKSCAN to reorganize the vector chains created during vectorization into a more usable form where gaps between end points are eliminated, and where the vectors in neighboring chains are given the same direction.

E. Draw Connecting Lines

Making this selection causes TOKSCAN to draw the recognized signal connector lines.

F. Reset Bitmap

Making this selection causes TOKSCAN to refresh the currently displayed bitmap image from the file on the hard disk.

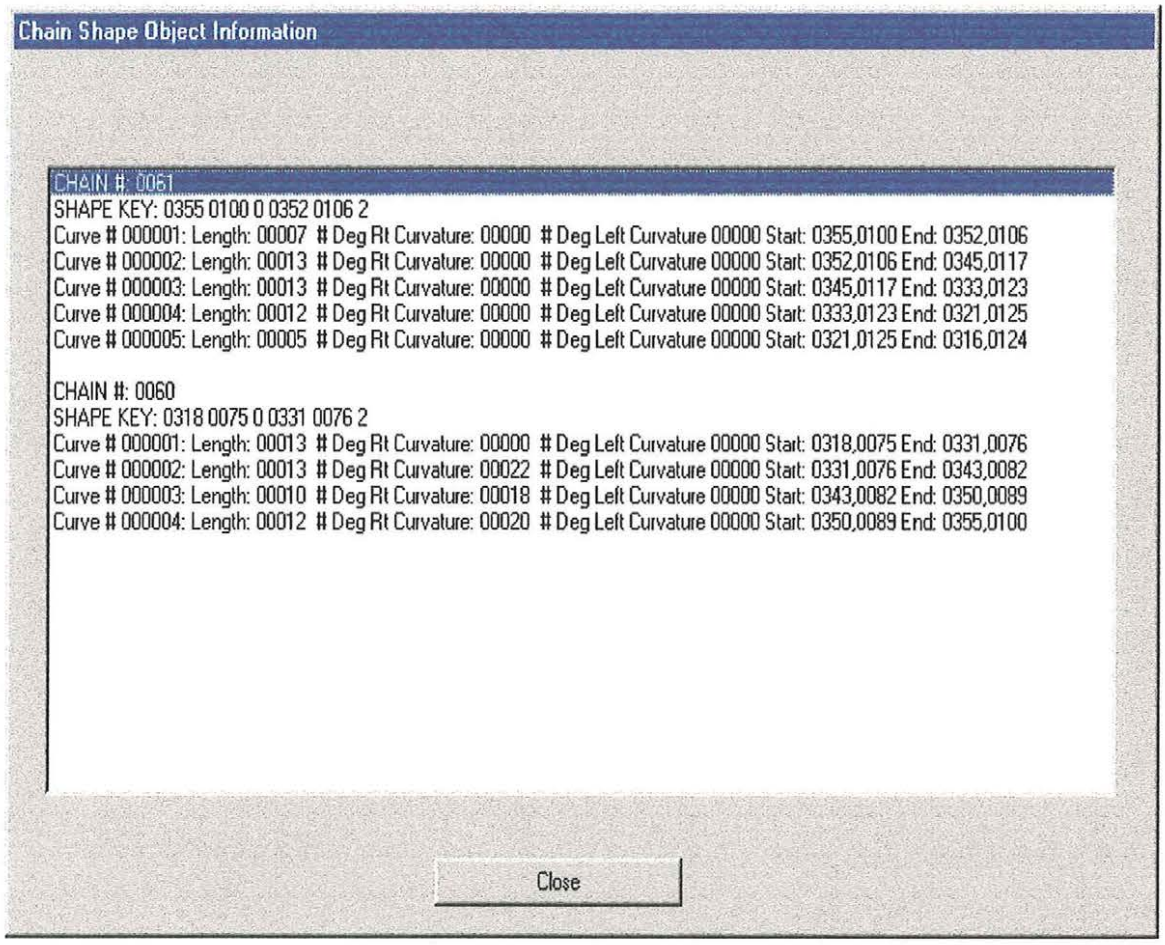
G. Load Thinned Bitmap From File

Images may be "pre-thinned" and "pre-vectorized", in order to save processing time when they are called up for testing multiple times. In order to accomplish this, it is necessary to perform thinning on the original image, and then to save the thinned image as a separate file. When this selection is made, a file-open dialog is displayed which allows the user to select a "pre-thinned" image to load for image recognition. The next section provides instructions for working with "pre-thinned" and "pre-vectorized" images.

H. Refresh Identified Objects

Making this selection causes TOKSCAN to perform image recognition on the

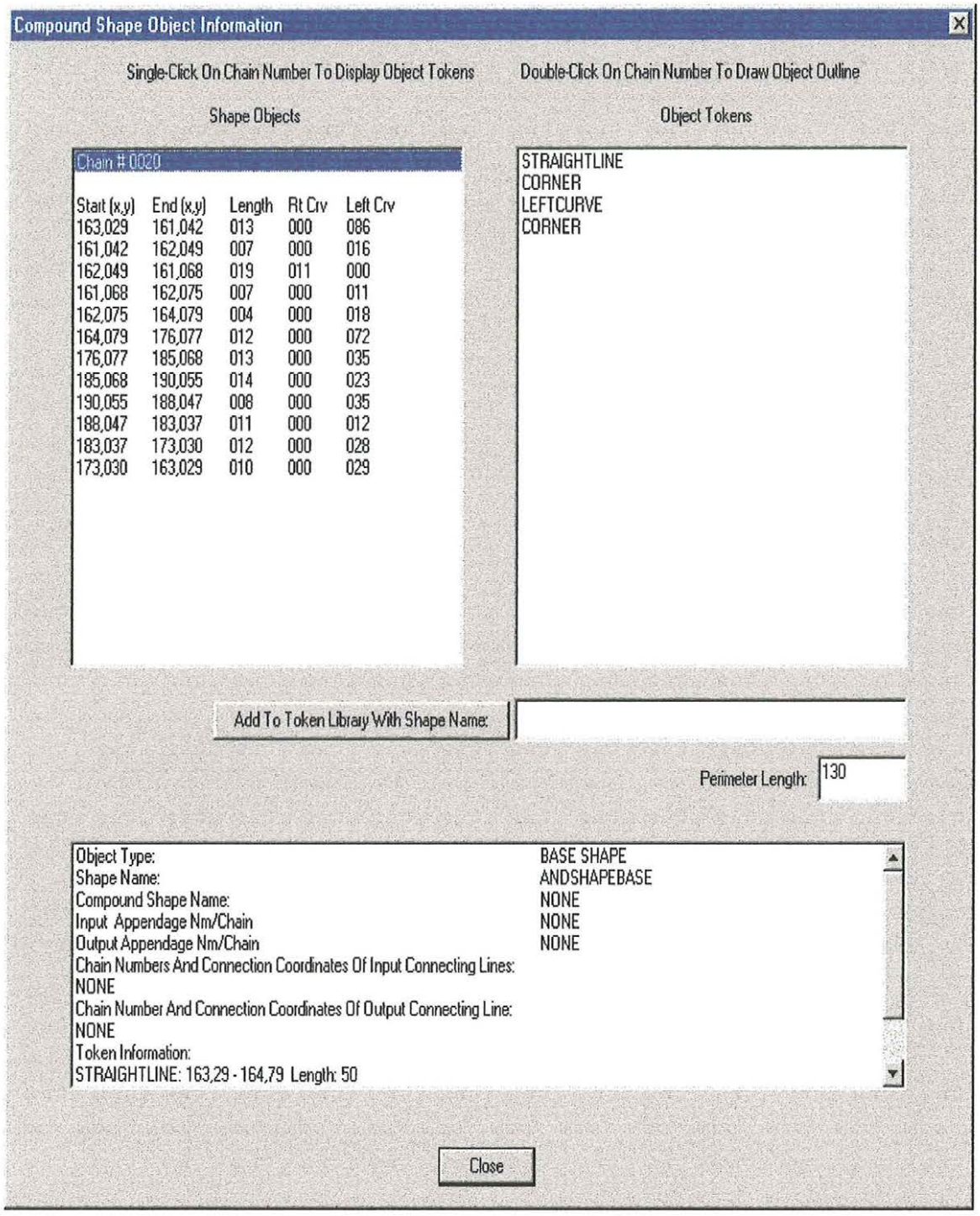
Chain Shape Object Information Dialog



This dialog is displayed when an image has been opened, and either vectorization has been performed, or the database of vectors from a previous vectorization has been loaded into memory. The user has used the mouse to select a shape object by placing the mouse cursor near the desired object(s), holding down the left button, and moving the mouse over the object(s) so that the visual selection rectangle overlaps the object(s). The user has then released the left mouse button, and the above dialog is displayed. The information displayed is as follows: 1) Chain number: the internal identification number used for all simple and compound shape objects. 2) Shape key: another internal key used to access both kinds of shape objects. 3) The vector chain coordinates which are contained in the selected shape.

If the user double-clicks the mouse on the "Chain #" line in the listbox, two things happen: 1) The shape which is described by the displayed vectors is highlighted in red in the displayed bitmap. 2) The chain number is added to the list of process log selection entries, so that debugging information can be displayed for the object in the process log.

Compound Shape Object Information Dialog



This dialog is displayed when an image has been opened, and either Library Analysis or Image Recognition has been performed. The user has used the

mouse to select a shape object by placing the mouse cursor near the desired object(s), holding down the left button, and moving the mouse over the object(s) so that the visual selection rectangle overlaps the object(s). The user has then released the left mouse button, and the above dialog is displayed. The information displayed is as follows: Top-left listbox: 1) Chain number: the internal identification number used for all simple and compound shape objects. 2) The vector chain coordinates which are contained in the selected shape. Top-right listbox: The shape tokens which were generated to describe the object. Bottom listbox: Information stored in the compound shape object.

NOTE: when the listbox is first opened, only the information in the top left listbox is shown. If the user selects the chain number of the desired object with the mouse, the other listboxes are filled in with information about the selected object. If the user selected more than one object from the image, then the top-left listbox will contain information about each selected object.

If the user double-clicks the mouse on the "Chain #" line in the listbox, two things happen: 1) The shape which is described by the displayed vectors is highlighted in green in the displayed bitmap. 2) The chain number is added to the list of process log selection entries, so that debugging information can be displayed for the object in the process log.

Zip Disk Installation Instructions

The Zip disk which is available as a part of this project contains all source files, the executable file, all necessary supporting data files, and a set of test image files which can be successfully processed by TOKSCAN.

To install the program, it is only necessary to copy the entire directory structure (TOKSCAN and all sub-directories) to the hard disk, and optionally, to create a Windows 95 or Windows NT Shortcut icon which points to the executable file. The debug version of the executable has path name TOKSCAN\DEBUG\tokscan.exe, and the release version has path name TOKSCAN\RELEASE\tokscan.exe. If the shortcut is not created, start TOKSCAN by opening the Windows Explorer, going to directory TOKSCAN\RELEASE or TOKSCAN\DEBUG, and double-clicking the mouse on file tokscan.exe. When copying files and directories, use the Windows Explorer CUT/PASTE operations, so as to preserve long file names.

To create a Windows 95 shortcut, specify that the program should start in the TOKSCAN\Release (release version) or TOKSCAN\Debug (debug version) directory. The target should be TOKSCAN\RELEASE\TOKSCAN.EXE (release version) or TOKSCAN\DEBUG\TOKSCAN.EXE (debug version).

When setting up Visual C++ (version 4.0 or 4.2 - Enterprise Edition) to compile and test this program, make sure that the database installation option is selected (to install database components). Also, install the DAO (Data Access Object) Software Development Kit (SDK). The DAO SDK should be installed even if Visual C++ is not installed. The project file which should be opened from within Visual C++ is TOKSCAN\imagelib.mdp.

If there are any problems executing the program, it may be necessary to install the DAO SDK redistribution package from Microsoft (included in the Zip disk, in directory TOKSCAN\EXTRA\DAOUPGRADE), and/or Microsoft Access. The database used by TOKSCAN is a Microsoft Access file.

If the directories are copied to a hard drive other than C:, two changes will be necessary in the project make file. Start Visual C++, and open the project workspace - TOKSCAN\imagelib.mdp. Select menu BUILD, and item SETTINGS. When the project settings dialog box opens, select imagelib - WIn32 Debug from the list on the left side of the dialog box. Select the LINK folder on the right side of the dialog box. Change the drive specification in the "Object/Library modules" edit control to the correct drive. Next, select imagelib - Win32 Release from the list on the left side of the dialog box. Select the LINK folder on the right side of the dialog box. Change the drive specification here to match what was done for the debug link. Press OK, and close the project

workspace in order to write the changes to the project workspace file.

In order to run debug sessions from Microsoft Developer Studio, select the DEBUG folder from the project settings dialog box, and make sure that the "Working Directory" is set to the TOKSCAN directory.

The DEBUG version of the program (built by setting the current configuration to DEBUG in the Microsoft Developer Studio) must be run under Windows 95, because it uses a profile file (imagelib.ini) which works properly only under Windows 95. The RELEASE version of the program (built by setting the current configuration to RELEASE) will run properly either under Windows NT or Windows 95.

Directory For Zip Disk Files

The following is a list of the directories and some of the files on the Zip disk, with explanations. (Hard disk C: is assumed here).

Directories:

C:\tokscan	Contains all source files and subdirectories
C:\tokscan\data	Contains test image files and supporting files
C:\tokscan\Debug	Build directory with obj files (debug)
C:\tokscan\Extra	Extra untested image files, and DAO Install
C:\tokscan\Extra\DAOUpgrade	DAO Redistribution Files
C:\tokscan\Extra\ExtraImages	Extra untested image files
C:\tokscan\FlexBison	FLEX and BISON port to Windows 95
C:\tokscan\FlexBison\BISON124	BISON port to Windows 95
C:\tokscan\FlexBison\FLEX247	FLEX port to Windows 95
C:\tokscan\FlexBison\FlexBisonTest	FLEX and BISON test files
C:\tokscan\Release	Build directory with obj files (release)
C:\tokscan\res	Windows resource files for project
C:\tokscan\YACC	YACC parser test files for project
C:\tokscan\YACC\Debug	Build directory for YACC parser test
C:\tokscan\yacclib	YACC parser source and .lib files
C:\tokscan\yacclib\Debug	Build directory for YACC .lib (debug)
C:\tokscan\yacclib\Release	Build directory for YACC .lib (release)

Selected Files

C:\tokscan\imagelib.mak	Project make file
C:\tokscan\imagelib.mdp	Project workspace file
C:\tokscan\imaglib.dat	Run file which points to directory with data

C:\tokscan\data\adder.bmp	Scanned image file for full adder schematic
C:\tokscan\data\adder.dbs	Vector database for full adder schematic
C:\tokscan\data\adder.upr	User parm file for full adder schematic
C:\tokscan\data\adderthin.bmp	Thinned image file for full adder schematic
C:\tokscan\data\exercisescan.bmp	Scanned image file for exercise schematic
C:\tokscan\data\exercisescan.dbs	Vector database for exercise schematic
C:\tokscan\data\exercisescan.upr	User parm file for exercise schematic
C:\tokscan\data\exercisescanthin.bmp	Thinned image file for exercise schematic
C:\tokscan\data\imagdata.mdb	Database work file used by TOKSCAN
C:\tokscan\data\imagetok.dat	Token library file
C:\tokscan\data\multiplexer.bmp	Scanned image file for multiplexer schematic
C:\tokscan\data\multiplexer.dbs	Vector database for multiplexer schematic
C:\tokscan\data\multiplexer.upr	User parm file for multiplexer schematic
C:\tokscan\data\multiplexerthin.bmp	Thinned image file for multiplexer schematic
C:\tokscan\data\schematic1.bmp	Drawn image file for test schematic
C:\tokscan\data\schematic1.dbs	Vector database for test schematic
C:\tokscan\data\schematic1.upr	User parm file for test schematic
C:\tokscan\data\schematic1thin.bmp	Thinned image file for test schematic
C:\tokscan\Debug\tokscan.exe	Debug executable
C:\tokscan\Debug\imaglib.dat	Run file which points to data directory
C:\tokscan\Release\tokscan.exe	Release executable
C:\tokscan\Release\imaglib.dat	Run file which points to data directory
C:\tokscan\YACC\driver.c	Source for YACC parser test driver
C:\tokscan\YACC\Test.y	YACC input source (rules)
C:\tokscan\YACC\testinp.dat	YACC parser test input file
C:\tokscan\YACC\test_tab.c	Test output C source from YACC
C:\tokscan\YACC\test_tab.h	Test output C header source from YACC
C:\tokscan\YACC\workpj.mak	YACC parser test make file
C:\tokscan\YACC\workpj.mdp	YACC parser test project file
C:\tokscan\yacclib\Alloca.c	Memory allocation source (from FLEXBISON)

C:\tokscan\yacclib\yaccparser.c	YACC parser source (created from test_tab.c)
C:\tokscan\yacclib\Debug\yacclib.lib	YACC parser link library (debug)
C:\tokscan\yacclib\Release\yacclib.lib	YACC parser link library (release)

Performing Full Image Processing On An Image

TOKSCAN can perform image recognition in two different modes. In the fully automatic mode, an image file is opened, binarization is done, and then TOKSCAN is requested to perform recognition. It automatically goes through the complete process of thinning, vectorization, determination of minimal closed polygons, determination of schematic components, determination of connecting signal lines, and location of circular signal line connectors. The user then identifies circuit inputs and outputs, and requests equation generation.

In the manual mode, the user can "pre-vectorize" and "pre-thin" an image, and save the

results in files which can later be opened along with the image, in order to avoid performing vectorization and thinning again. This is useful when testing is being performed on an image, and it is necessary to perform library analysis or recognition multiple times on the same image.

In this section, we will illustrate the fully automatic mode by providing specific instructions for the test input image "schematic1.bmp". The same instructions apply for all other test images provided with the program.

Instructions for fully automatic recognition on "schematic1.bmp"

1. Start the TOKSCAN program.

2. Verify that the token library file has been located and opened by doing the following:

    a. Select menu item "Token-Library".
    b. Select "Edit Token Library" from the drop-down menu.
    c. Press the "Shape Relationships" button in the "Shape Token Library Update" dialog box which is opened after step b is completed.
    d. Verify that there are six entries in the listbox displayed in the "Basic Shape Linear Relationships In Compound Shape" dialog box which is opened after step c is completed. If there is no data in the listbox, then check that the starting directory in the shortcut is C:\TOKSCAN\RELEASE (assuming drive C:), or that

the program was started using Windows Explorer, as described above.

3. Select the FILE menu item, and then the OPEN menu item (from the drop-down menu).

4. From the file-open dialog box, move to the TOKSCAN\DATA directory, and open the file "schematic1.bmp". If a message box is displayed indicating that there is no user profile file for the selected image, then the starting directory for TOKSCAN is not set correctly. (It should be C:\TOKSCAN\RELEASE, assuming that the C: drive is used).

5. Select the menu item "Image-Preprocessing", and the menu item "Binarization" from the drop-down menu.

6. In the "Select Binarization Threshold Level" dialog, select a global threshold value of 80, and press the "Apply" button. After the image has been binarized, press the "Close" button.

7. Select the menu item "Image-Analysis".

8. Select the menu item "Find And Process Basic Shapes" from the drop-down menu displayed after completing step 5.

9. After step 6 is complete, indicate the circuit input and output points for TOKSCAN, by doing the following:

    a. Press the toolbar button which changes the mouse cursor to the "In/Out Points" mode (third button from the right). The mouse cursor should change, and display the words "In/Out Points".

    b. For each of the four circuit input points (at the top left side of the image), press and hold the left mouse button to display a selection rectangle, and move the rectangle so that it is over the end point of one of the input signal lines. Then, release the button. TOKSCAN should flag the location with a red dot, and with the words "INP-A", "INP-B", "INP-C", and "INP-D".

    c. For the circuit output point (at the right side of the drawing, about half way down), press and hold the right mouse button to display a selection rectangle, and move the rectangle so that it is over the end point of the output signal line. Then, release the button. TOKSCAN should flag the location with a red dot, and with the words "OUT-A".

d. Select the menu item "Image-Analysis".

e. Select the menu item "Generate Schematic Equation" from the drop-down menu displayed when step d is completed.

After completing steps 1 - 9, TOKSCAN should follow all of the connector lines from the indicated inputs to the indicated output, highlighting the lines in blue as it executes, and then it should display an equation at the bottom of the bitmap for the analyzed circuit.

Performing "Pre-vectorization" and "Pre-thinning"

Step 8 above usually takes a lot of processing time, and if an image must be processed repeatedly, it saves time to perform "pre-thinning" and "pre-vectorization", and then to load the results into memory along with the image when it is processed the next time.

To perform "pre-thinning" on the schematic image "schematic1.bmp, for example, do the following:

1. Select the FILE menu item, and then the OPEN menu item (from the drop-down menu).

2. From the file-open dialog box, move to the TOKSCAN\DATA directory, and open the file "schematic1.bmp".

3. Select the menu item "Image-Preprocessing", and the menu item "Binarization" from the drop-down menu.

4. In the "Select Binarization Threshold Level" dialog, select a global threshold value of 80, and press the "Apply" button. After the image has been binarized, press the "Close" button.

5. Select the menu item "Tools", and then "Thin Image" (from the drop-down menu).

6. After the image has been thinned (when step 3 completes), select menu item "FILE", and then "Save Bitmap As" (from the drop-down menu). In the file-save dialog box which appears, enter a file name for the thinned image, and press OK to save it to the hard disk. Make note of the file name used.

To perform "pre-vectorization" on this schematic image after completing step 6 above, do the following:

1. Select the menu item "Tools", and the item "Extract Shape Object Vectors" (from the drop-down menu). This step extracts the vectors and places them in memory.

2. Select the menu item "Tools", and the item "Reorganize Shape Vectors" (from the drop-down menu). This step reorganizes the vectors and saves them in the working database used by TOKSCAN.

3. Select the menu item "FILE", and the item "Save Database File As" (from the drop-down menu). A file-save dialog box will appear, which allows you to assign a permanent file name for the vector database in a library of database files. The file is assigned a file type (or DOS extension) of .dbs. Enter the desired file name, and press okay to save the database. Make note of the file name.

Loading And Performing Recognition On A "Pre-vectorized" And "Pre-thinned" Image

After performing "pre-thinning" and "pre-vectorization", schematic1.bmp can now be processed more quickly by calling up the thinned image file and the database of vectors which were saved. Starting from the point where TOKSCAN is running, and no image is loaded, the following steps should be followed to perform recognition on schematic1.bmp, taking advantage of "pre-thinning" and "pre-vectorization".

1. Select the FILE menu item, and then the OPEN menu item (from the drop-down menu).

2. From the file-open dialog box, move to the TOKSCAN\DATA directory, and open the file "schematic1.bmp".

3. Select the menu item "FILE", and item "Open User Parm File" (from the drop-down menu).

4. From the file-open dialog which is displayed, move to the TOKSCAN\DATA directory, select the file "schematic1.upr", and press OK. This will open the pre-defined user parameter file built for this image. (Note: this step is actually performed automatically whenever an image is opened, and there is a file with the .upr extension and a matching first node in the same directory as the image file). If the user parms are changed and it is necessary to save the changes back into the schematic1.upr file, select the menu item "FILE" and item "Save User Parm File" (from the drop-down menu), enter schematic1.upr as the file name, and press OK.

5. Select the menu item "FILE", and item "Open Database File" (from the drop- down menu).

6. From the file-open dialog, move to the TOKSCAN\DATA directory, select the file "schematic1.dbs", and press OK. This selects the database vector file from the library, and copies it to TOKSCAN's working database.

7. Complete steps 7 - 9 from "Instructions for fully automatic recognition on schematic1.bmp" above.

Steps 1 - 6 can be done very quickly, compared to the time required for the typical thinning and vectorization processes.

Performing Library Analysis On schematic1.bmp

Library Analysis consists of detecting the closed minimal polygons in an image, generating the tokens which describe the shape of the polygons, and saving the tokens in the token library for future recognition. To do this for the schematic1.bmp image, do the following:

1. Complete steps 3 - 7 from "Instructions for fully automatic recognition on schematic1.bmp" above.

2. Select "Find Basic Shapes For Library Update" from the drop-down menu.

3. After step 2 has completed, all detected closed minimal polygons are highlighted in the image in purple.

4. Select a polygon for which the tokens should be saved in the token library. To do this, place the mouse cursor near the polygon, press and hold the left mouse button, move the mouse to create a selection rectangle, place the selection rectangle over the polygon, and release the left mouse button. This will cause the "Compound Shape Object Information" dialog box to open .

5. Use the mouse to select the "Chain #" line of the desired polygon in the upper left listbox (multiple polygons will be selected if the selection rectangle from step 4 overlaps more than one polygon).

6. In the "Add To Token Library With Shape Name" edit box, type the name which you would like to assign to the token list when it is saved in the token library.

7. Press the "Add To Token Library With Shape Name" button. This saves the token list in the library. It can then be used in shape relationships to define a compound shape.

Matched Sets Of Image Files And Supporting Files

There are four test images provided in the TOKSCAN\DATA directory which have been thoroughly tested for proper recognition with TOKSCAN. Each image file is accompanied by a vector database file, a user parameter file, and a thinned image file, as shown in the following list.

C:\tokscan\data\adder.bmp	Scanned image file for full adder schematic
C:\tokscan\data\adder.dbs	Vector database for full adder schematic
C:\tokscan\data\adder.upr	User parm file for full adder schematic
C:\tokscan\data\adderthin.bmp	Thinned image file for full adder schematic
C:\tokscan\data\exercisescan.bmp	Scanned image file for exercise schematic
C:\tokscan\data\exercisescan.dbs	Vector database for exercise schematic
C:\tokscan\data\exercisescan.upr	User parm file for exercise schematic
C:\tokscan\data\exercisescanthin.bmp	Thinned image file for exercise schematic
C:\tokscan\data\multiplexer.bmp	Scanned image file for multiplexer schematic
C:\tokscan\data\multiplexer.dbs	Vector database for multiplexer schematic
C:\tokscan\data\multiplexer.upr	User parm file for multiplexer schematic
C:\tokscan\data\multiplexerthin.bmp	Thinned image file for multiplexer schematic
C:\tokscan\data\schematic1.bmp	Drawn image file for test schematic
C:\tokscan\data\schematic1.dbs	Vector database for test schematic
C:\tokscan\data\schematic1.upr	User parm file for test schematic
C:\tokscan\data\schematic1thin.bmp	Thinned image file for test schematic

There are additional image files which have not been tested with TOKSCAN that are included in directory TOKSCAN\EXTRA\EXTRAIMAGES. Successful recognition with these images will require testing and careful adjustment of the user parameter file. They may also require some modification to TOKSCAN: debugging has been carried out completely for the four sample images, but time constraints for the project did not allow complete debugging for all of these additional images. With some additional debugging, TOKSCAN should

successfully recognize most of the included extra images. REMEMBER: this is a prototype program which demonstrates that the techniques implemented will work: it has not yet been tested to the point of being ready for use against any desired schematic image.

Modifying And Testing The YACC Parser

The directory TOKSCAN\YACC contains the source code necessary to test the YACC parser separately from TOKSCAN, using a small driver program named driver.c. A Visual C++ project has been set up in this directory to build the test parser with the test driver. The project workspace file is named "workpj.mdp".

To modify the YACC parser rule set, and recreate the test parser, do the following:

1. Open file TOKSCAN\YACC\test.y, and modify the YACC rules as needed. Then save the changed file.

2. Install FLEX/BISON, using the directions provided with the download, and run the test.y file through it to produce C source code for the parser (called test_tab.c). Update the existing test_tab.c provided in the TOKSCAN\YACC directory with the new version, and build the workpj project from within Visual C++.

3. A test input file with tokens that have the same format used by TOKSCAN is provided in the TOKSCAN\YACC directory, named "testinp.dat". To execute the test parser built in step 2, open a DOS window, and enter the command:

   workpj < testinp.dat > testout.txt

   This will execute the parser using the test input file, and will produce a text output file with the results of the parse called "testout.txt". The proper results for the current version of the parser are provided in the existing file TOKSCAN\YACC\testout.txt.

4. After the test version of the parser has been tested, it must be added to the link library which is included in the TOKSCAN project. To accomplish this, do the following:

   a. Rename the parser c source code file from test_tab.c to yaccparser.c, and copy it to the TOKSCAN\YACCLIB directory, overlaying the existing yaccparser.c file.

b. Rebuild the TOKSCAN project. The yacclib.lib library will automatically be rebuilt as a part of the overall project build, and the parser will be link edited into TOKSCAN.

Modifying And Testing The TOKSCAN Project

TOKSCAN can be modified using Visual C++, version 4.0 through 4.2 (Enterprise

Edition). It was developed using version 4.2. To load the project, open the workspace file TOKSCAN\imagelib.mdp. See the comments at the beginning of part 2 of this manual for more information.

Introduction To The Class Structure In TOKSCAN

After the project workspace file (TOKSCAN\imagelib.mdp) has been opened in Visual C++, all of the user-defined C++ classes can be seen in the class "tree view" window. Visual C++ provides fast access to the source definitions of each class, and of each member variable and member function through the use of this tree view. The user can expand the view by double-clicking the mouse on an entry. Double-clicking the mouse on an entity inside of a class causes the source code to be opened at the location where the entity is defined.

The TOKSCAN project was originally generated using Microsoft's "App Wizard", and the original set of classes generated by that tool were retained in the final structure. Numerous other classes have been added, with many interrelationships between the classes.

Visual C++ has a class browser utility which helps the user understand the structure of the program, and find where classes are referenced. It is a good idea for a new user to review the browser file provided in the Zip Disk (TOKSCAN\DEBUG\imagelib.bsc). The following page contains a list of the user-defined classes in the project, with a brief explanation of the function of each.

Classes Generated By The Microsoft "App Wizard" (Heavily Modified)

CImageLib	Main application class

CMainFrame	Main frame window class (of MDI interface); user defined program initialization is done here
CCInputBitmapDoc	Document class which holds all image data
CCInputBitmapView	View class which controls display of Doc class data
CChildFrame	Child window class for MDI interface
CAboutDlg	"About" dialog box class

User-Defined Classes

CBinarize	Dialog box class which controls image binarization
CChainStartCount	DAO (Microsoft Data Access Object) class which gets a count of the number of vector chains saved in the working database
CCompoundImageCoordIdx	Manages a hash table of keys which index a set of CCompoundImageObject objects
CCompoundImageObject	Compound Shape Object implementation
CCompoundShapeDialog	Dialog box class which drives the "Compound Shape Object Information" dialog box
CConnectionMatrix	Implements the connection matrix used to analyze the connections between schematic components
CDAOChainStartSet	DAO class which retrieves vector information from the database
CDAOVectorChainSet	DAO class which retrieves vector information from the database
CImageObject	Simple shape object implementation
circularregion	Supports "circular searches" in the image bitmap around a specific set of coordinates
CkFillFilter	Dialog box class which controls noise reduction

CLineSmoothParms	Dialog box class which supports the line smoothing parameters which are set by the user
CProcessLog	Dialog box class which supports the process log, which is used for debugging
CProfileDB	DAO class which supports the retrieval of user parameter information from the working database
CProgressDlg	Dialog box class which supports the progress bars that indicate time remaining on long running tasks
CShapeLinearRelDialog	Dialog box class which supports the "Basic Shape Linear Relationships In Compound Shape" dialog
CShapeMap	Supports a hash table which references a collection of CCompoundImageObject instances
CShapeNumEntryDialog	Dialog box class which supports the "Shape Selection Data" dialog
CShapeObjectDialog	Dialog box class which supports the "Chain Shape Object" dialog
CShapeTokenLibDialog	Dialog box class which supports the "Shape Token Library Update" dialog
CTokenLibData	Supports the retrieval and update of token library information in memory, and file i/o
CTraverseVectorChain	Supports traversals through a set of vectors contained in a shape object
CUserParm4	Dialog box class which supports the "Shape Object Parameters" dialog
CUserParms	Dialog box class which supports the "Curvature And Junction Margin Of Error" dialog

CUserParms2	Dialog box class which supports the "Perimeter Values" dialog
CUserParms3	Dialog box class which supports the "Closed Polygon Search Parameters" dialog
CUserParms5	Dialog box class which supports the "Token Edits" dialog
ImageVectors	Supports the "thick line" vectorization process
mathfunctions	Provides necessary mathematical functions such as distance measurement, arccosines, etc.

NOTE ON RUNNING THE TEST IMAGES ON DIFFERENT PC'S

In some cases, the test images will not process correctly when this program is transferred to another PC system. The problem has to do with the Window's color palette, which varies from system to system, and which causes the test bitmaps to display with slightly different intensities in some cases. If this problem occurs in a new installation, it may help to load the test bitmap, then load an image color file which has been provided in the ZIP disk in the DATA directory. This will reset the grayscale intensity values to those originally used to test the image.

There are three image color files provided, one for each of the three scanned test images (schematic1.bmp should not have this problem), with names adder.cif, exercisescan.cif, and multiplexer.cif. To apply the color information file to adder.bmp, for example, open adder.bmp, then from the FILE menu, select OPEN IMAGE COLOR FILE. When the file dialog box opens, select adder.cif as the file to load. This will reset the bitmap display, and image recognition can then be performed. The changed image can also be saved as a new bitmap.

NOTE ABOUT DATA FILE PATHS

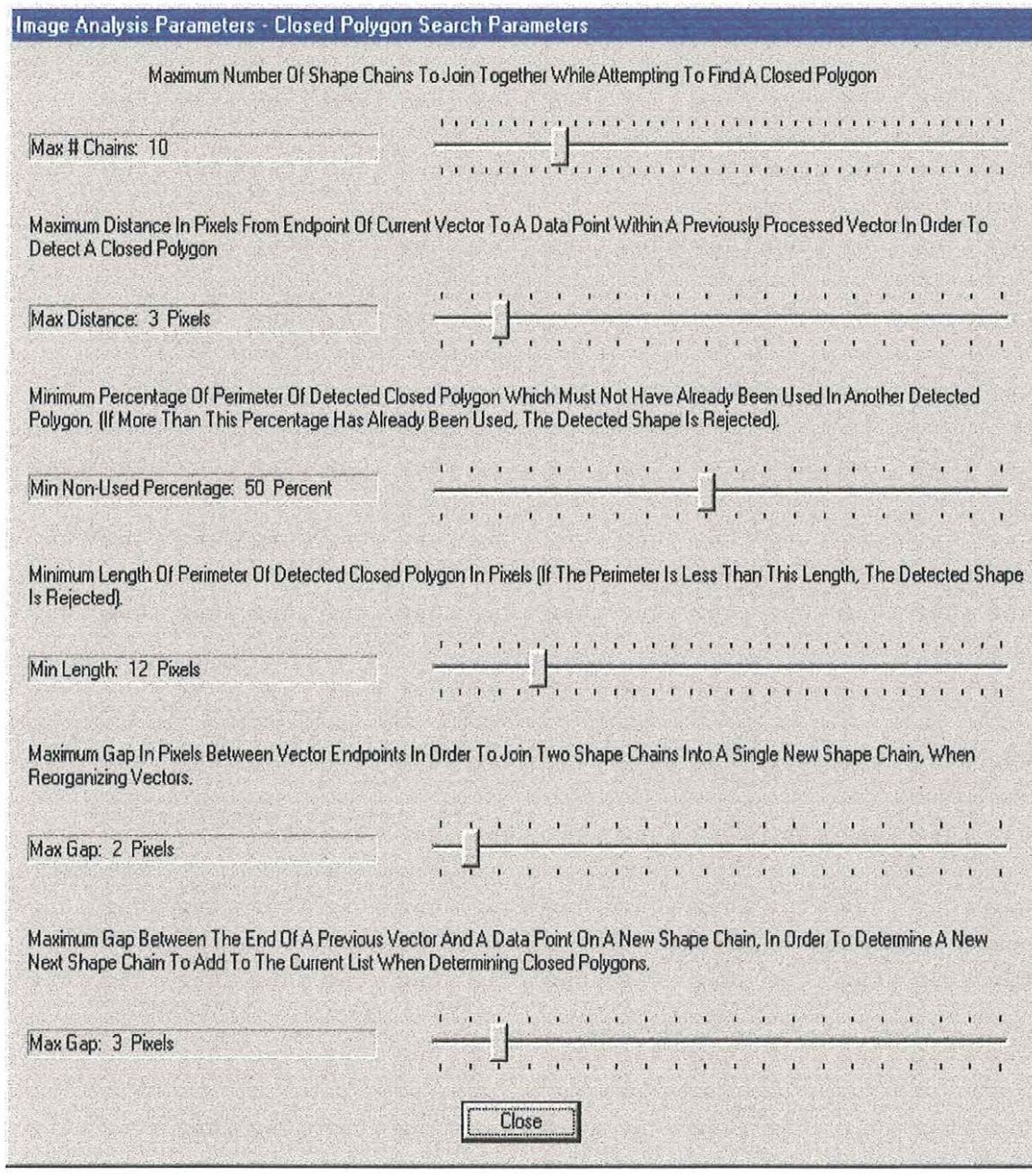
If you use a drive/directory other than c:\tokscan for the installation, make sure that you change the path name in file imaglib.dat to match the directory used. This file is located in the RELEASE, DEBUG, and TOKSCAN directories on the Zip disk.

VITA

James A. (Jim) Giles has a Bachelor of Arts degree from the University of South Florida in Mathematics, and expects to receive a Master of Science degree in Computer and Information Sciences from the University of North Florida, August, 1997. Dr. Yap S. Chua of the University of North Florida is serving as Jim's thesis advisor.

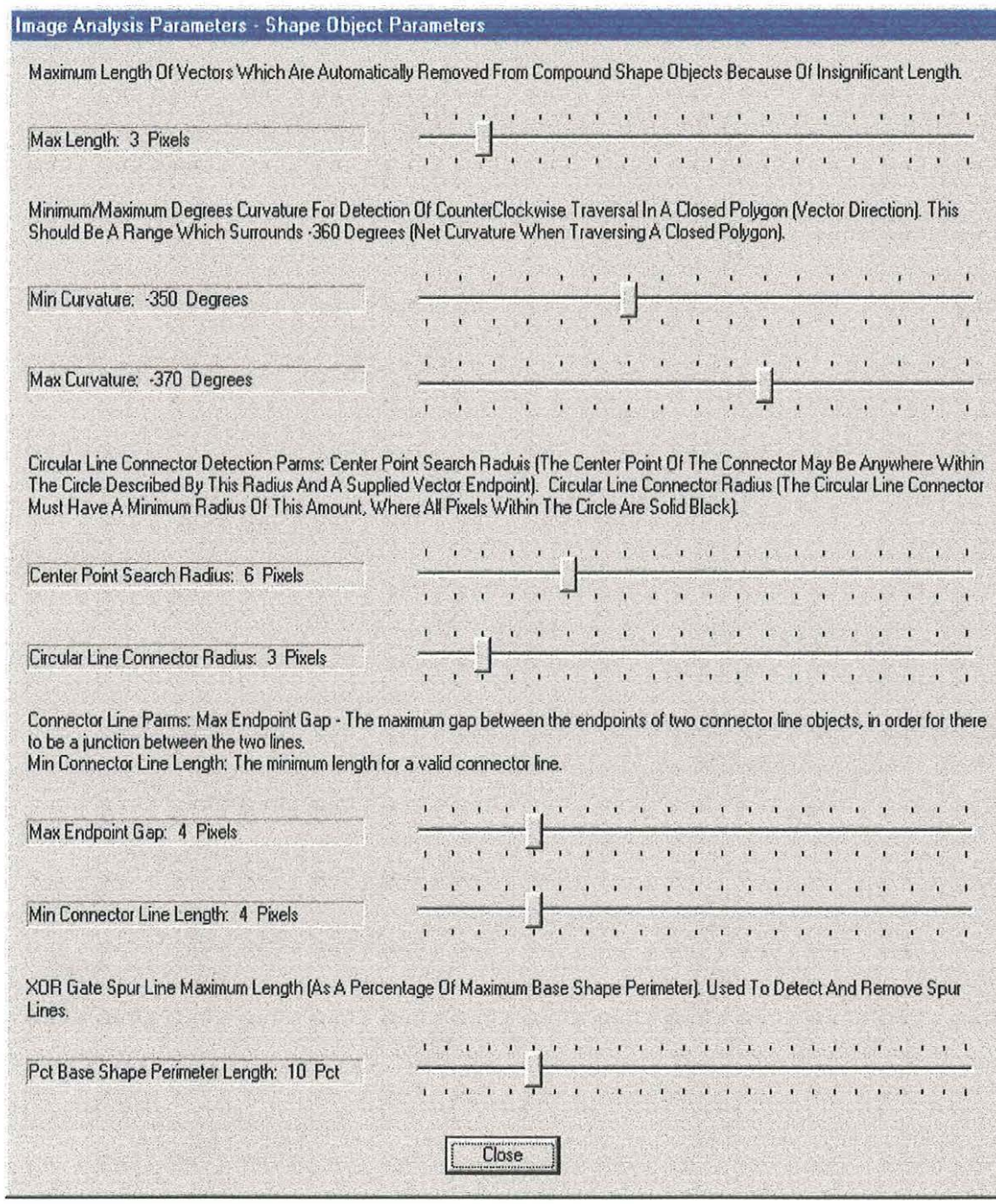
Jim has a background in business data processing in the life insurance and transportation industries, and on-going interests in mathematics, engineering, image processing, and computer networks.

Closed Polygon Search Parameters Dialog



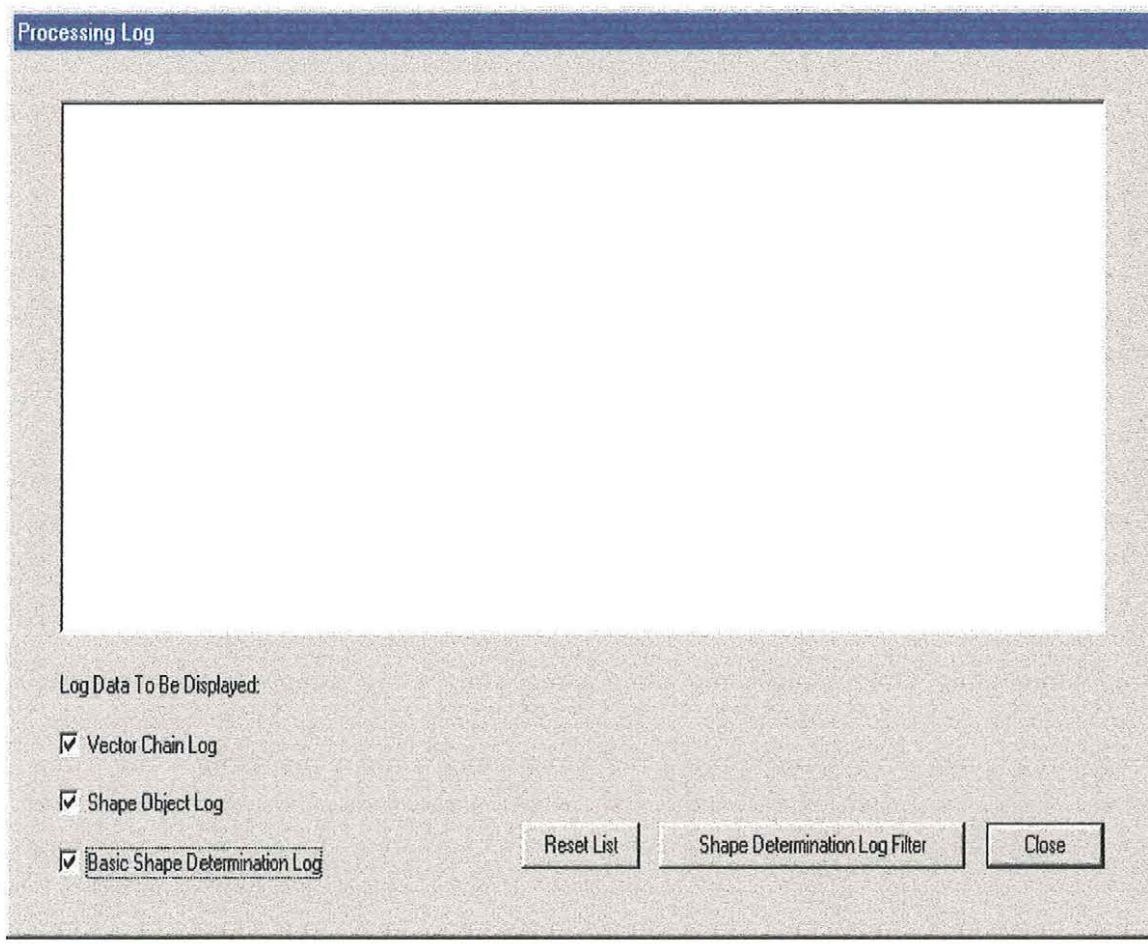
The parameters in this dialog box have to do with the detection of closed minimal polygons. See section 2.15 for a detailed explanation of each parameter.

Shape Object Parameters Dialog



The parameters in this dialog box have to do with the processing of the various shape objects which are created during the recognition process. See section 2.15 for a detailed explanation of each parameter.

Processing Log Dialog



The Processing Log dialog differs from the other user parameter dialogs, in that the user sets values which control the display of a processing log which monitors the image recognition process. In the lower left corner, the user can select the kinds of messages which should be included in the log from any of three categories:

Vector Chain Log - lists the vectors which describe the simple shape objects selected by the shape determination log filter.

Shape Object Log - lists information about compound shape objects selected by the shape determination log filter.

Basic Shape Determination Log - lists processing information during the detection of closed minimal polygons.

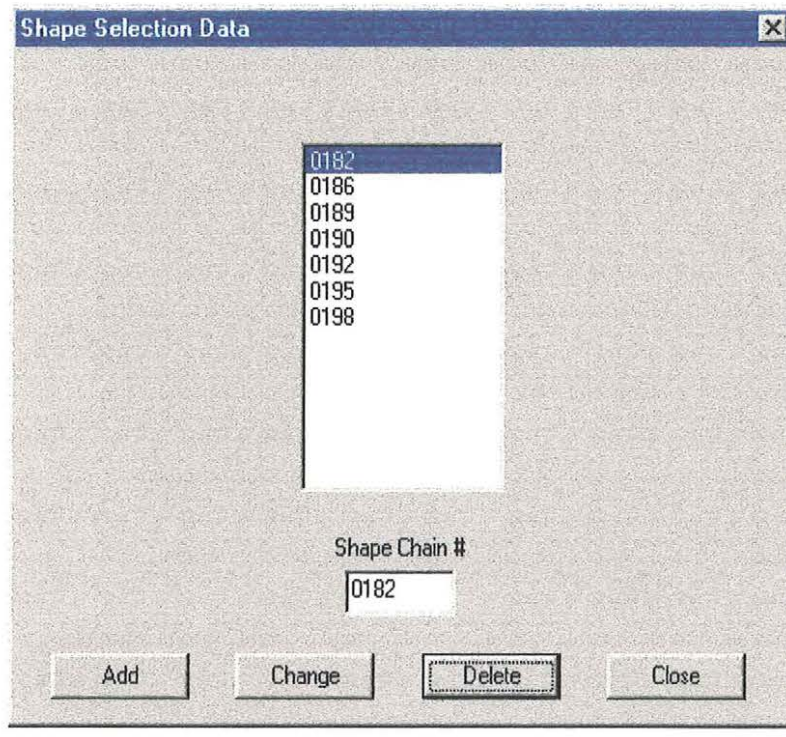
The processing log is used for debugging purposes to gather detailed information about the program execution when an image is not recognized properly.

Simple and compound shape objects can be selected for log information by entering their identifying chain number in one of two ways: 1) They may be entered manually by pressing the "Shape Determination Log Filter" button, and typing in the proper chain numbers. 2) Objects in the image may be selected directly from the image by using the mouse. When an object is selected, a dialog box opens which provides recognition information about the object, including its identifying chain number. The user can either type in the displayed chain number (#1 above), or double click on the chain number displayed in the object information dialog to have it automatically inserted into the list of object numbers for which log information is being collected. To select an object with the mouse, the mouse cursor should be placed near the desired object, and the left mouse button should be held down, while moving the mouse toward the object. A selection rectangle will be drawn in the window, and any objects which fall within that rectangle will be selected and displayed in the dialog box.

There are two different dialog boxes which may be displayed when an object is selected, one for simple shape objects, and the other for compound shape objects. The simple shape object dialog is displayed when an object is selected before Library Analysis or Image Recognition is run. The compound shape object dialog is displayed when an object is selected after Library Analysis or Image Recognition is run.

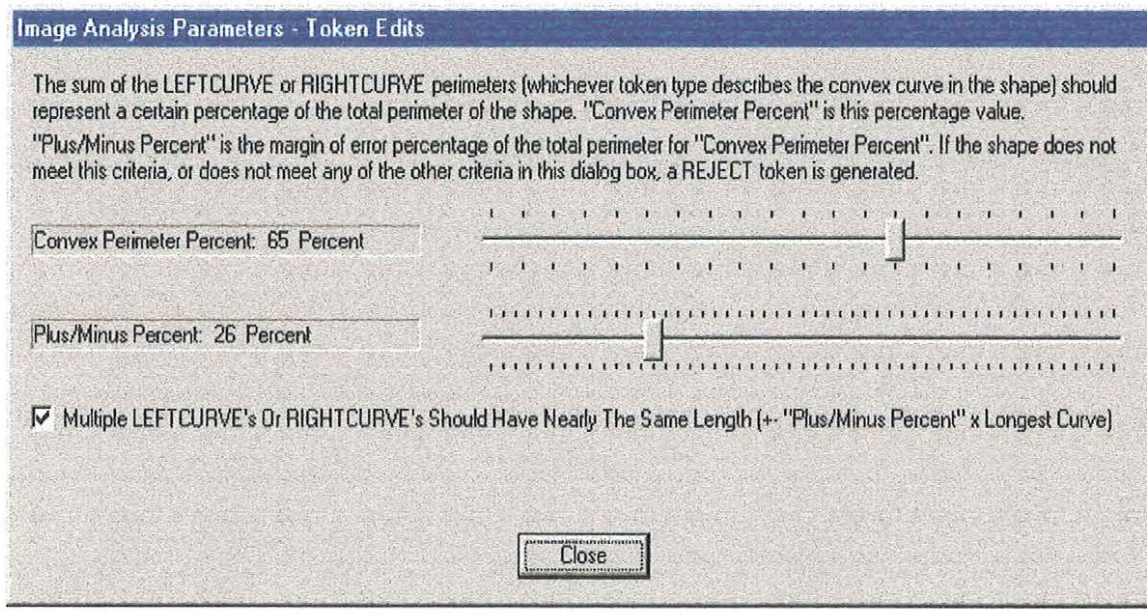
These two dialog boxes are shown and described later in this manual.

Processing Log Dialog

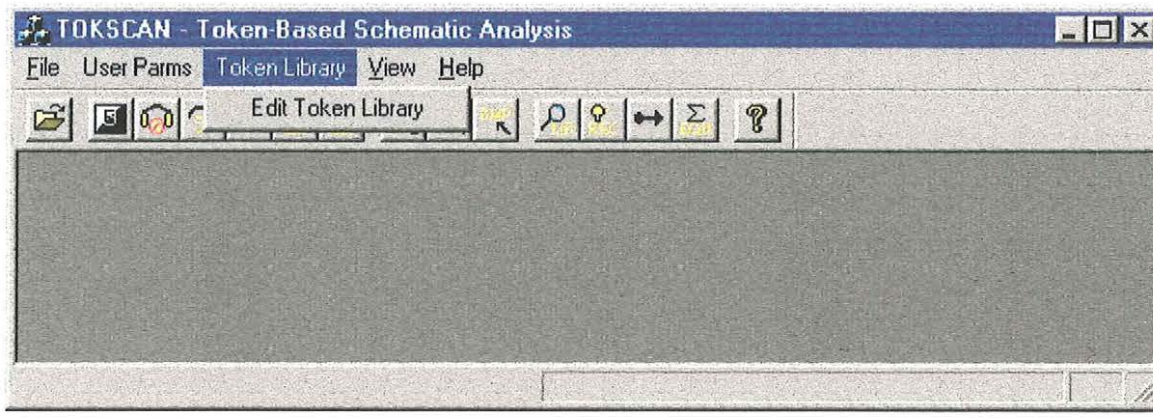


This dialog box is displayed when the "Shape Determination Log Filter" button is pressed in the Process Log dialog box. It allows the entry of identifying chain numbers for shape objects in order to display program execution information about the objects in the process log. The list may be updated manually using this dialog, or entries may be added by direct selection from the image, as described above in the "Process Log Dialog" information.

Token Edits Dialog



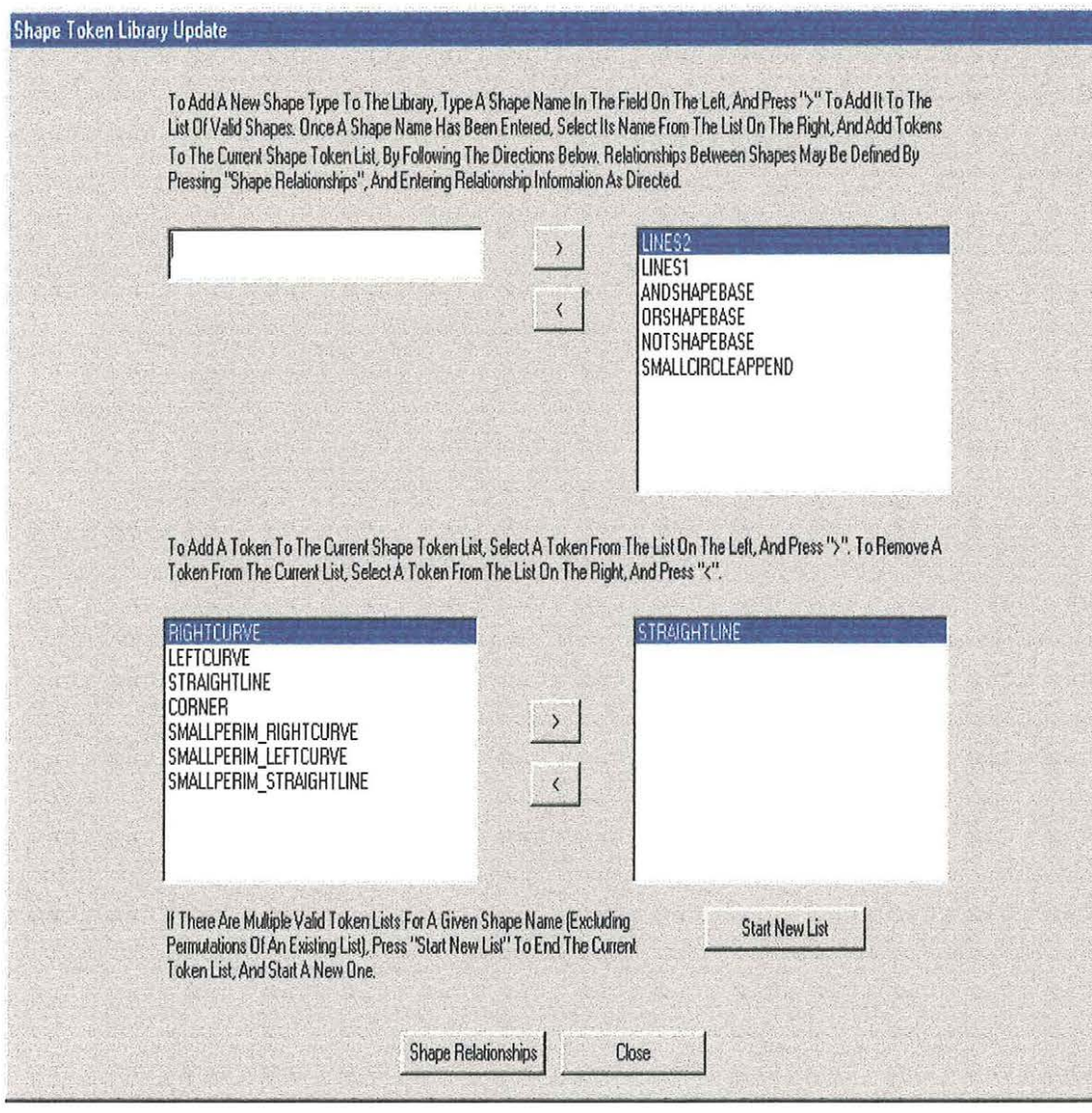
The parameters in this dialog box have to do with distinguishing between valid component shapes and invalid shapes formed from crossed signal connector lines. See section 2.15 for a detailed explanation of each parameter.



Token Library Menu

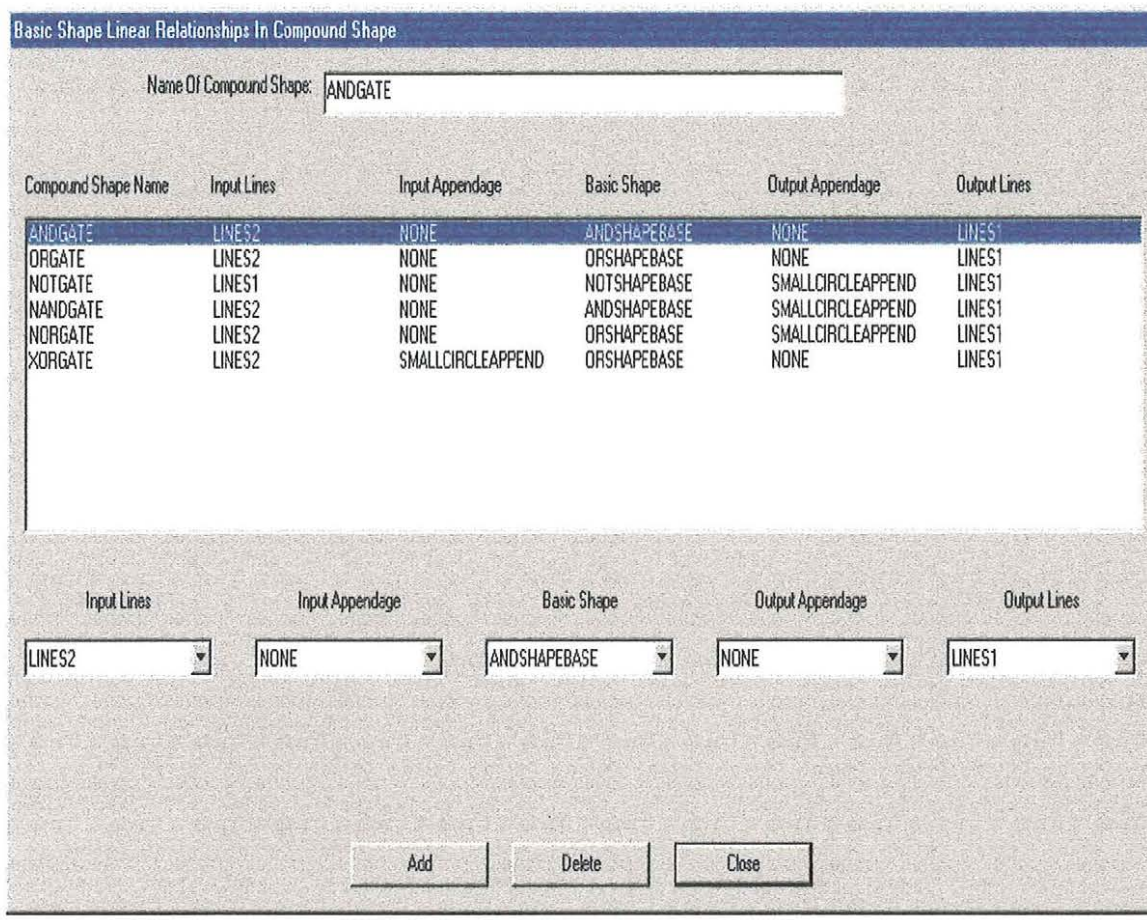
The EDIT TOKEN LIBRARY selection displays a dialog box which allows descriptive token lists to be added or updated in the token library. The dialog box is documented on the following page.

Shape Token Library Update Dialog



This dialog box is used to update the shape token library with new token lists. Lists may be added or removed, by following the directions shown in the dialog box. Token lists may also be added to the library by using the "Compound Shape Object Information" Dialog, which is described later in this manual.

Basic Shape Linear Relationships In Compound Shape Dialog



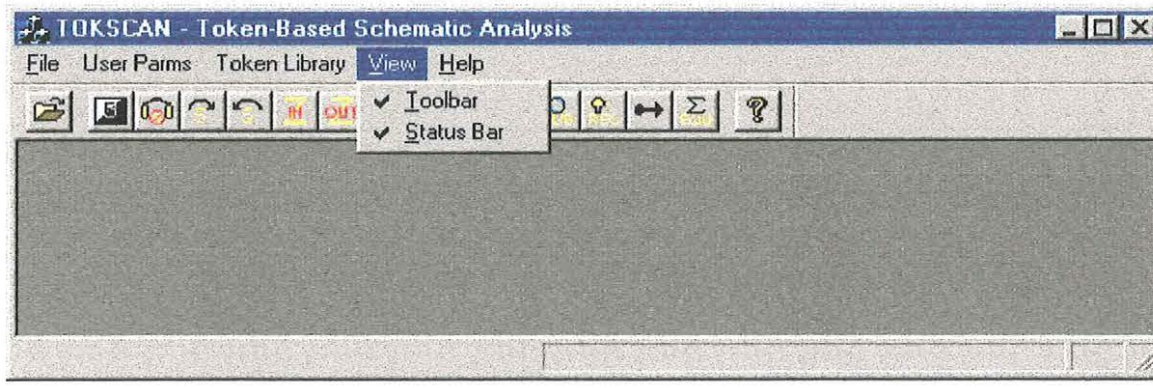
This dialog is displayed by pressing the "Shape Relationships" button in the "Shape Token Library Update" dialog box. It allows the user to specify the relationship between the simple shapes which make up the more complex schematic component. As an example of how this is used, look at the first line of data shown above in the dialog box.

The compound shape, named "ANDGATE" is a schematic component, and when a successful match is made with an AND logic gate in a schematic image, the component will be labeled as an "ANDGATE". Reading from left to right, the data line which describes the ANDGATE may be interpreted as follows: the ANDGATE has 2 input lines, no input appendage, it uses the AND base shape, it has no output appendage, and it has one output line. All of these simple shapes are arranged in the linear order given by reading left to right.

A new compound shape can be added by entering a name in the "Name Of Compound Shape" edit control, and then selecting appropriate entries from each of the "drop-down combo-boxes" below it, and pressing the ADD button. Deletes

may be done by selecting an existing entry, and pressing the DELETE button. Note that if a new shape is added, corresponding "hard coding" will have to be added to the source code in the current version of TOKSCAN to generate new token types for the parser, so that the new shape is included in the resulting equation. See Appendix B for more information about hard coding.

1. Main Application Window - Menu Selections Continued



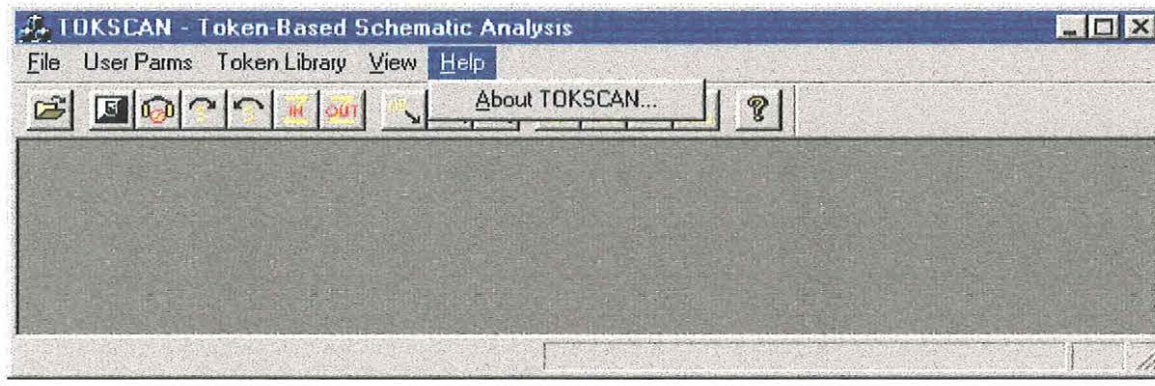
View Menu

A. Toolbar

Selecting this item toggles the appearance of the toolbar, making it appear or disappear from the window.

B. Status Bar

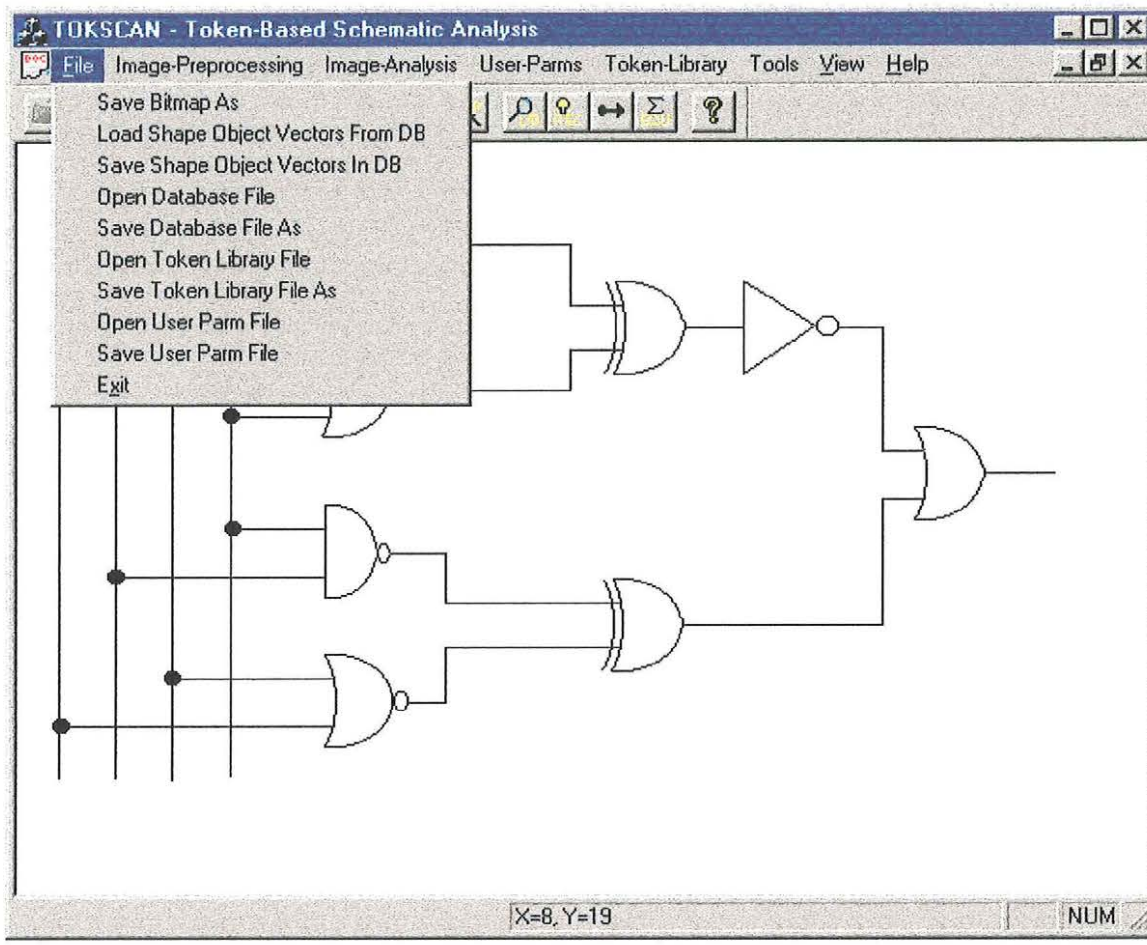
Selecting this item toggles the appearance of the status bar at the bottom of the window, making it appear or disappear.



Help Menu

Selecting About TOKSCAN displays a dialog box with information about the program.

2. Document Window - Menu Selections



A. Save Bitmap As

A file-save dialog box is displayed which allows the displayed bitmap to be saved in Windows BMP file format.

B. Load Shape Object Vectors From DB

Vectors which describe the currently displayed image, which were previously created by TOKSCAN and saved in the database, are loaded into memory for processing. This function is provided in order to avoid having to perform vectorization every time an image is loaded for recognition. A file-open dialog box is displayed, so the user can select a "pre-thinned" binary image for processing. The original image must have already been thinned, and the result saved in the file which is opened at this point.

C. Save Shape Object Vectors In Db

Vectors which describe the currently displayed image, which have just been created by the vectorization process, and which currently reside in memory, are saved in the database. They may be reloaded into memory, as described in B.

D. Open Database File

TOKSCAN is capable of maintaining a library of database files, each of which contains the vectorization output for one image which has been processed by the program. When images are reprocessed, the appropriate database of vectors may be opened and transferred into memory, and recognition may be performed without having to perform vectorization again. A file-open dialog is displayed, which allows the user to select a .dbs file from the library.

E. Save Database File As

After vectorization has been completed on an image, the user can save the resulting database file into the library of vectorization files (described in D). A file-save dialog is displayed, which allows the user to save the database in the library with a .dbs file type.

F. Open Token Library File

TOKSCAN is capable of maintaining a library of token files. The token library for a specific image or group of images is saved in a single file, but TOKSCAN can handle multiple distinct token libraries in separate files. A file-open dialog is displayed which allows the user to open a token library file of type .tok.

G. Save Token Library File As

A token library may be saved in a single file, which may be added to a library of token files, each of which is a separate token library. The user may choose an appropriate token library for use with a specific image or group of images, as described in F. A file-save dialog is displayed which allows the user to save the token file in the library with a .tok file type.

H. Open User Parm File

TOKSCAN is capable of maintaining a library of user parameter files. Each parameter file contains one complete set of user parameters, which control the recognition process. Using this feature, the user can automatically set all of the user-adjustable parameters at one time to values which are appropriate for a

particular image. A file-open dialog is displayed, which allows the user to open a user parameter file of type .upr.

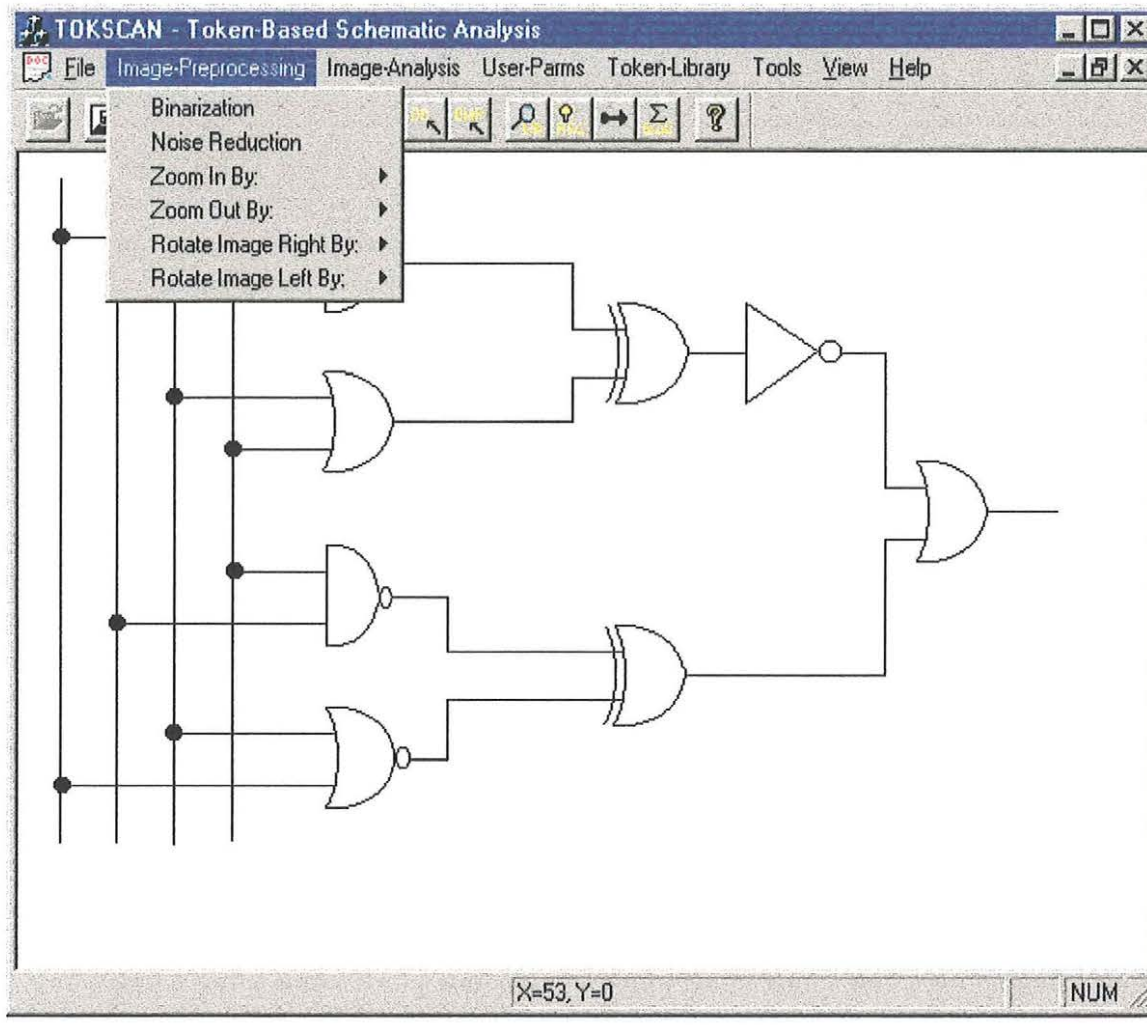
I. Save User Parm File

After the user sets the processing parameters for a particular image, they may be saved permanently in a user parameter file. This file can be opened later for use with any desired image. When it is opened, all of the adjustable parameters are set according to the values saved in it. TOKSCAN can save and open multiple user parm files from a library, so that a distinct parm file can be maintained for each image. A file-save dialog is displayed, which allows the user to save the parm file with file type .upr.

J. Exit

The application is closed.

2. Document Window - Menu Selections Continued



A. Binarization

A dialog box is displayed which allows the user to select a binarization global threshold value in the range 0 - 255, where 0 is pure black, 255 is pure white, and every other value is a shade of gray in between. From the dialog box, the user can apply binarization to the image using the selected grayscale value.

B. Noise Reduction

A dialog box is displayed which allows the user to select a threshold value for the noise reduction operation, and to apply noise reduction to the image. A "window size" can also be selected, which specifies the number of pixels which surround a pixel of interest that are used to calculate an average grayscale value for the pixel of interest. The average value is compared with the selected

threshold value, and the color of the pixel of interest is set to either pure white or pure black accordingly.

C. Zoom In By

When this menu item is selected, another menu is displayed which gives the user the option of several different "zoom percentage values". After a percentage value is selected, the displayed image is magnified by the selected percentage, and the total size of the bitmap is increased by that percentage.

D. Zoom Out By

When this menu item is selected, another menu is displayed which gives the user the option of several different "zoom percentage values". After a percentage value is selected, the displayed image is compressed by the selected percentage, and the total size of the bitmap is decreased by that percentage.

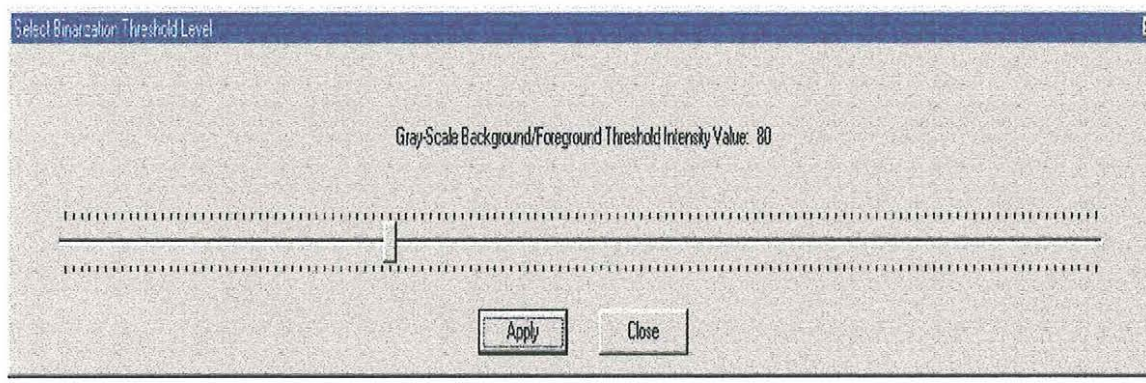
E. Rotate Image Right By

When this menu item is selected, another menu is displayed which gives the user the option of several different "rotation degree values". After a degree value is selected, the displayed image is rotated clockwise by the selected number of degrees.

F. Rotate Image Left By

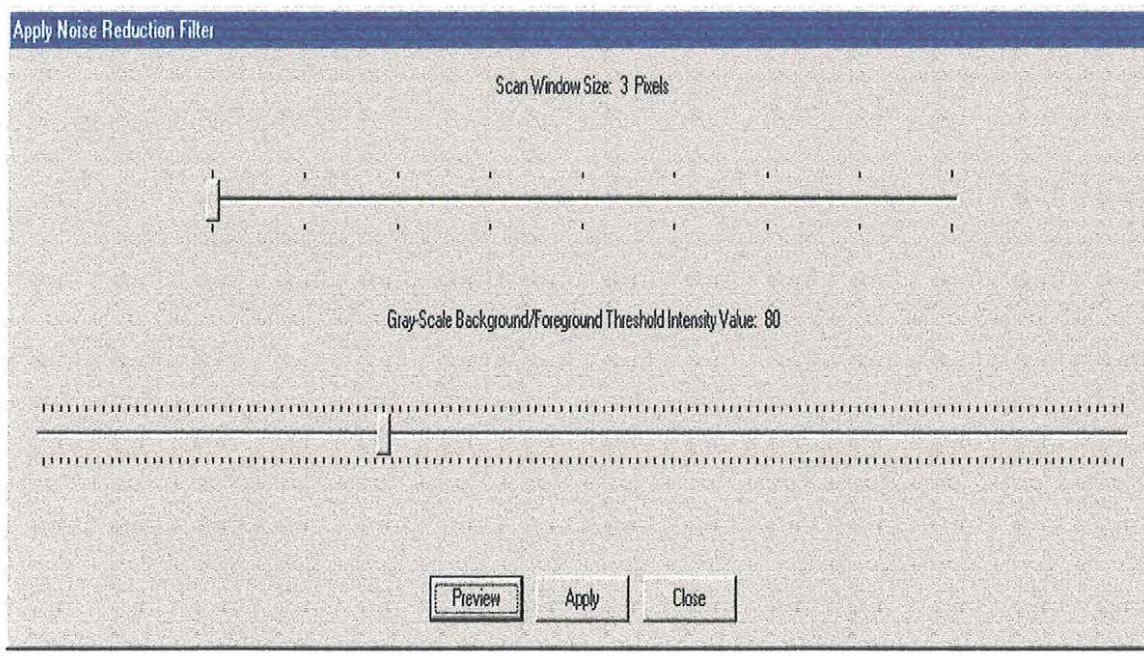
When this menu item is selected, another menu is displayed which gives the user the option of several different "rotation degree values". After a degree value is selected, the displayed image is rotated counterclockwise by the selected number of degrees.

Select Binarization Threshold Level Dialog



This dialog box is used to control the binarization process. The slider is used to set the global threshold to a value in the range 0 - 255, and it is applied to the image by pressing the APPLY button.

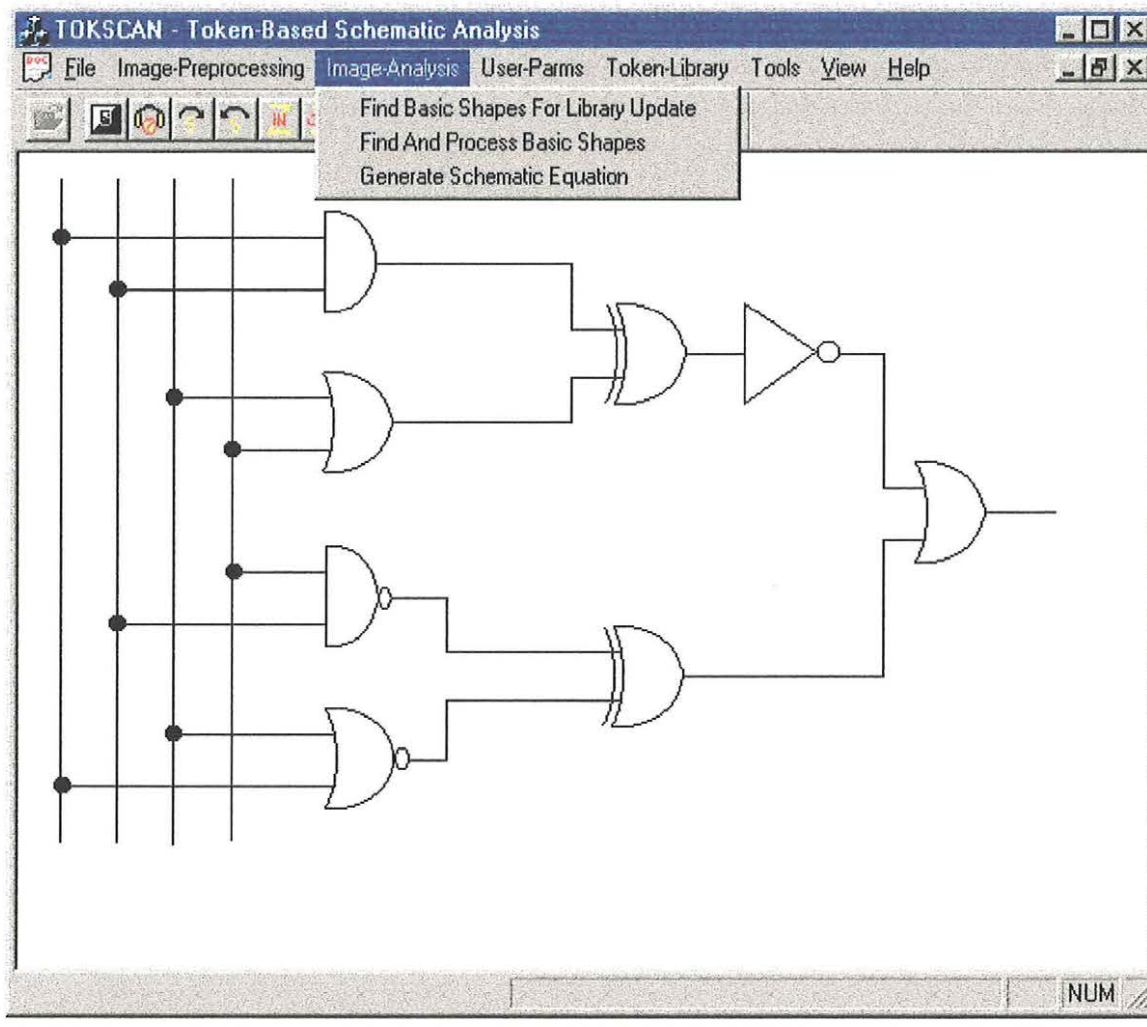
Apply Noise Reduction Filter Dialog



This dialog box is used to control the noise reduction process. The sliders are used to set a "window" size (of data around a pixel of interest), and a global threshold value that is compared with the average grayscale intensity in the window. The effects of the settings may be observed by pressing the PREVIEW button. Each time PREVIEW is pressed, the image reverts to its original state

before noise reduction is applied again. When the APPLY button is pressed, the last noise reduction performed is applied to the image for the duration of processing.

2. Document Window - Menu Selections Continued



NOTE: The USER-PARMS and TOKEN-LIBRARY menu items have exactly the same functionality as the corresponding menu items in the main application window, which have already been discussed. They are not presented again in this section.

A. Find Basic Shapes For Library Update

Making this selection causes TOKSCAN to perform image recognition on the displayed image up to the point of detecting closed minimal polygons, and generating the shape tokens which describe the polygons. If the image has already been thinned, and if vectorization has already been performed, the program will go immediately into polygon detection; otherwise it will perform thinning and vectorization as needed. It is possible to "pre-thin" and "pre-

vectorize" an image, and save the results on disk. Then, the thinned image and the accompanying set of vectors can be reloaded at a later time, and image recognition can be performed without having to go through thinning and vectorization again. In the second section of this manual, instructions are given for processing the images provided with the software. Specific directions are provided for "pre-thinning" and "pre-vectorization".

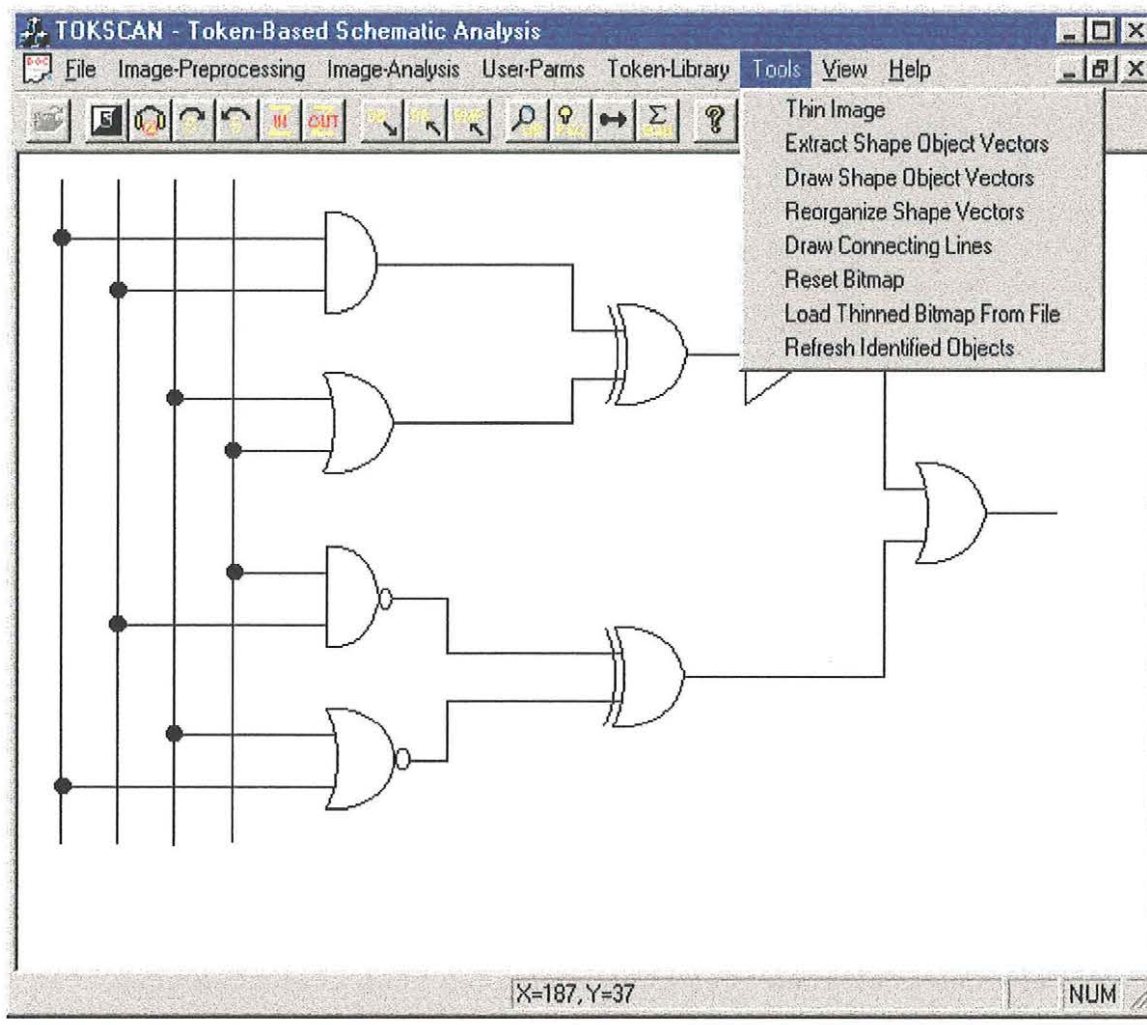
B. Find And Process Basic Shapes

Making this selection causes TOKSCAN to perform full image recognition on the displayed image. Components are located and labeled, and connecting signal lines and circular line connectors are located. After this function completes execution, the image is ready for the user to specify circuit input and output points, and to request generation of the equation.

C. Generate Schematic Equation

The user makes this selection after full image recognition has been completed, and after all circuit input and output points have been identified (manually). TOKSCAN follows all connecting signal lines from input to final output, and generates one or more logic equations which describe the recognized circuit.

2. Document Window - Menu Selections Continued



NOTE: The VIEW and HELP menu items have exactly the same functionality as the corresponding menu items in the main application window, which have already been discussed. They are not presented again in this section.

A. Thin Image

Making this selection causes TOKSCAN to perform thinning on the displayed image.

B. Extract Shape Object Vectors

Making this selection causes TOKSCAN to perform vectorization on the displayed image, and save the resulting vectors in the database.

C. Draw Shape Object Vectors

Making this selection causes TOKSCAN to draw all of the shape object vectors which were placed in simple shape objects as a result of the vectorization process.

D. Reorganize Shape Vectors

Making this selection causes TOKSCAN to reorganize the vector chains created during vectorization into a more usable form where gaps between end points are eliminated, and where the vectors in neighboring chains are given the same direction.

E. Draw Connecting Lines

Making this selection causes TOKSCAN to draw the recognized signal connector lines.

F. Reset Bitmap

Making this selection causes TOKSCAN to refresh the currently displayed bitmap image from the file on the hard disk.

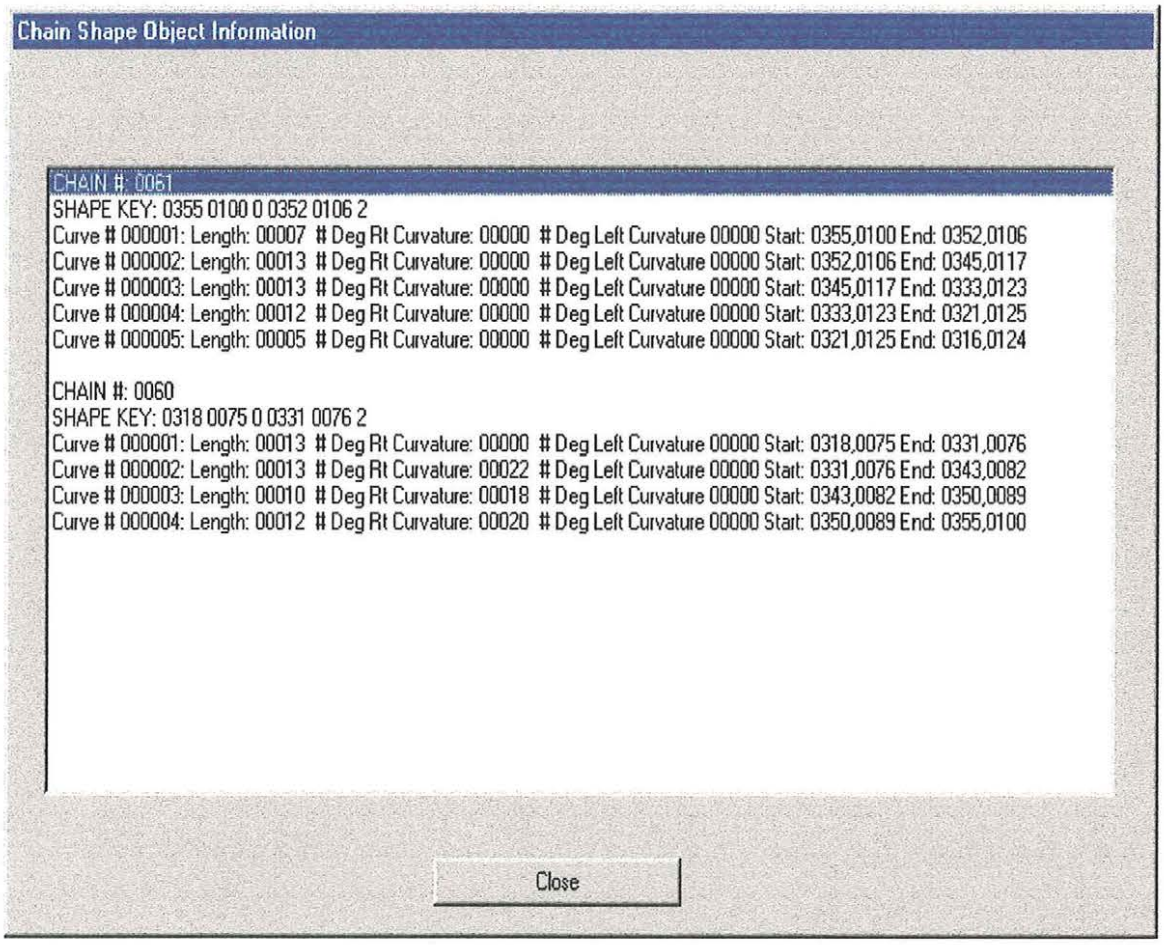
G. Load Thinned Bitmap From File

Images may be "pre-thinned" and "pre-vectorized", in order to save processing time when they are called up for testing multiple times. In order to accomplish this, it is necessary to perform thinning on the original image, and then to save the thinned image as a separate file. When this selection is made, a file-open dialog is displayed which allows the user to select a "pre-thinned" image to load for image recognition. The next section provides instructions for working with "pre-thinned" and "pre-vectorized" images.

H. Refresh Identified Objects

Making this selection causes TOKSCAN to perform image recognition on the

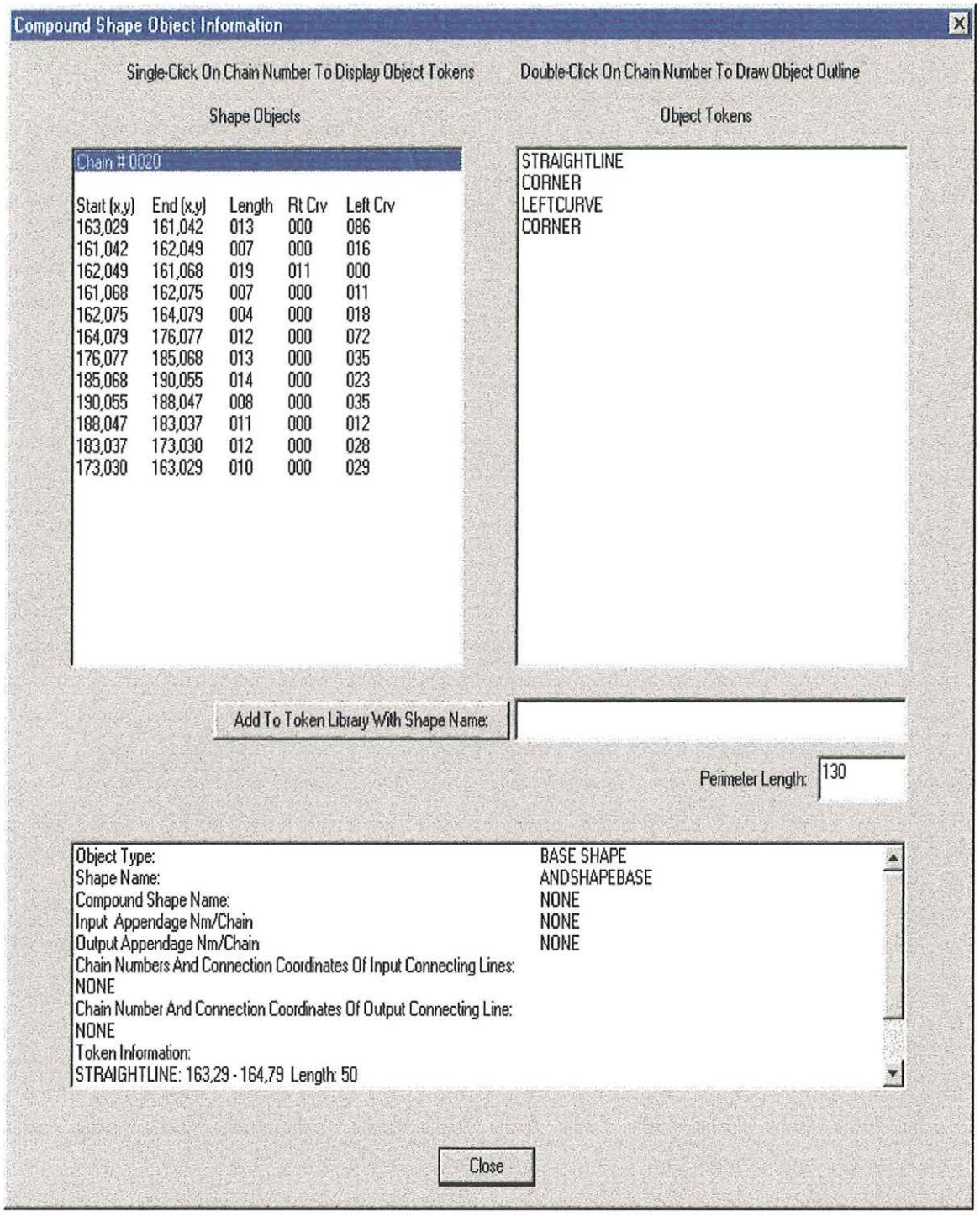
Chain Shape Object Information Dialog



This dialog is displayed when an image has been opened, and either vectorization has been performed, or the database of vectors from a previous vectorization has been loaded into memory. The user has used the mouse to select a shape object by placing the mouse cursor near the desired object(s), holding down the left button, and moving the mouse over the object(s) so that the visual selection rectangle overlaps the object(s). The user has then released the left mouse button, and the above dialog is displayed. The information displayed is as follows: 1) Chain number: the internal identification number used for all simple and compound shape objects. 2) Shape key: another internal key used to access both kinds of shape objects. 3) The vector chain coordinates which are contained in the selected shape.

If the user double-clicks the mouse on the "Chain #" line in the listbox, two things happen: 1) The shape which is described by the displayed vectors is highlighted in red in the displayed bitmap. 2) The chain number is added to the list of process log selection entries, so that debugging information can be displayed for the object in the process log.

Compound Shape Object Information Dialog



This dialog is displayed when an image has been opened, and either Library Analysis or Image Recognition has been performed. The user has used the

mouse to select a shape object by placing the mouse cursor near the desired object(s), holding down the left button, and moving the mouse over the object(s) so that the visual selection rectangle overlaps the object(s). The user has then released the left mouse button, and the above dialog is displayed. The information displayed is as follows: Top-left listbox: 1) Chain number: the internal identification number used for all simple and compound shape objects. 2) The vector chain coordinates which are contained in the selected shape. Top-right listbox: The shape tokens which were generated to describe the object. Bottom listbox: Information stored in the compound shape object.

NOTE: when the listbox is first opened, only the information in the top left listbox is shown. If the user selects the chain number of the desired object with the mouse, the other listboxes are filled in with information about the selected object. If the user selected more than one object from the image, then the top-left listbox will contain information about each selected object.

If the user double-clicks the mouse on the "Chain #" line in the listbox, two things happen: 1) The shape which is described by the displayed vectors is highlighted in green in the displayed bitmap. 2) The chain number is added to the list of process log selection entries, so that debugging information can be displayed for the object in the process log.

Zip Disk Installation Instructions

The Zip disk which is available as a part of this project contains all source files, the executable file, all necessary supporting data files, and a set of test image files which can be successfully processed by TOKSCAN.

To install the program, it is only necessary to copy the entire directory structure (TOKSCAN and all sub-directories) to the hard disk, and optionally, to create a Windows 95 or Windows NT Shortcut icon which points to the executable file. The debug version of the executable has path name TOKSCAN\DEBUG\tokscan.exe, and the release version has path name TOKSCAN\RELEASE\tokscan.exe. If the shortcut is not created, start TOKSCAN by opening the Windows Explorer, going to directory TOKSCAN\RELEASE or TOKSCAN\DEBUG, and double-clicking the mouse on file tokscan.exe. When copying files and directories, use the Windows Explorer CUT/PASTE operations, so as to preserve long file names.

To create a Windows 95 shortcut, specify that the program should start in the TOKSCAN\Release (release version) or TOKSCAN\Debug (debug version) directory. The target should be TOKSCAN\RELEASE\TOKSCAN.EXE (release version) or TOKSCAN\DEBUG\TOKSCAN.EXE (debug version).

When setting up Visual C++ (version 4.0 or 4.2 - Enterprise Edition) to compile and test this program, make sure that the database installation option is selected (to install database components). Also, install the DAO (Data Access Object) Software Development Kit (SDK). The DAO SDK should be installed even if Visual C++ is not installed. The project file which should be opened from within Visual C++ is TOKSCAN\imagelib.mdp.

If there are any problems executing the program, it may be necessary to install the DAO SDK redistribution package from Microsoft (included in the Zip disk, in directory TOKSCAN\EXTRA\DAOUPGRADE), and/or Microsoft Access. The database used by TOKSCAN is a Microsoft Access file.

If the directories are copied to a hard drive other than C:, two changes will be necessary in the project make file. Start Visual C++, and open the project workspace - TOKSCAN\imagelib.mdp. Select menu BUILD, and item SETTINGS. When the project settings dialog box opens, select imagelib - WIn32 Debug from the list on the left side of the dialog box. Select the LINK folder on the right side of the dialog box. Change the drive specification in the "Object/Library modules" edit control to the correct drive. Next, select imagelib - Win32 Release from the list on the left side of the dialog box. Select the LINK folder on the right side of the dialog box. Change the drive specification here to match what was done for the debug link. Press OK, and close the project

workspace in order to write the changes to the project workspace file.

In order to run debug sessions from Microsoft Developer Studio, select the DEBUG folder from the project settings dialog box, and make sure that the "Working Directory" is set to the TOKSCAN directory.

The DEBUG version of the program (built by setting the current configuration to DEBUG in the Microsoft Developer Studio) must be run under Windows 95, because it uses a profile file (imagelib.ini) which works properly only under Windows 95. The RELEASE version of the program (built by setting the current configuration to RELEASE) will run properly either under Windows NT or Windows 95.

Directory For Zip Disk Files

The following is a list of the directories and some of the files on the Zip disk, with explanations. (Hard disk C: is assumed here).

Directories:

C:\tokscan	Contains all source files and subdirectories
C:\tokscan\data	Contains test image files and supporting files
C:\tokscan\Debug	Build directory with obj files (debug)
C:\tokscan\Extra	Extra untested image files, and DAO Install
C:\tokscan\Extra\DAOUpgrade	DAO Redistribution Files
C:\tokscan\Extra\ExtraImages	Extra untested image files
C:\tokscan\FlexBison	FLEX and BISON port to Windows 95
C:\tokscan\FlexBison\BISON124	BISON port to Windows 95
C:\tokscan\FlexBison\FLEX247	FLEX port to Windows 95
C:\tokscan\FlexBison\FlexBisonTest	FLEX and BISON test files
C:\tokscan\Release	Build directory with obj files (release)
C:\tokscan\res	Windows resource files for project
C:\tokscan\YACC	YACC parser test files for project
C:\tokscan\YACC\Debug	Build directory for YACC parser test
C:\tokscan\yacclib	YACC parser source and .lib files
C:\tokscan\yacclib\Debug	Build directory for YACC .lib (debug)
C:\tokscan\yacclib\Release	Build directory for YACC .lib (release)

Selected Files

C:\tokscan\imagelib.mak	Project make file
C:\tokscan\imagelib.mdp	Project workspace file
C:\tokscan\imaglib.dat	Run file which points to directory with data

C:\tokscan\data\adder.bmp	Scanned image file for full adder schematic
C:\tokscan\data\adder.dbs	Vector database for full adder schematic
C:\tokscan\data\adder.upr	User parm file for full adder schematic
C:\tokscan\data\adderthin.bmp	Thinned image file for full adder schematic
C:\tokscan\data\exercisescan.bmp	Scanned image file for exercise schematic
C:\tokscan\data\exercisescan.dbs	Vector database for exercise schematic
C:\tokscan\data\exercisescan.upr	User parm file for exercise schematic
C:\tokscan\data\exercisescanthin.bmp	Thinned image file for exercise schematic
C:\tokscan\data\imagdata.mdb	Database work file used by TOKSCAN
C:\tokscan\data\imagetok.dat	Token library file
C:\tokscan\data\multiplexer.bmp	Scanned image file for multiplexer schematic
C:\tokscan\data\multiplexer.dbs	Vector database for multiplexer schematic
C:\tokscan\data\multiplexer.upr	User parm file for multiplexer schematic
C:\tokscan\data\multiplexerthin.bmp	Thinned image file for multiplexer schematic
C:\tokscan\data\schematic1.bmp	Drawn image file for test schematic
C:\tokscan\data\schematic1.dbs	Vector database for test schematic
C:\tokscan\data\schematic1.upr	User parm file for test schematic
C:\tokscan\data\schematic1thin.bmp	Thinned image file for test schematic
C:\tokscan\Debug\tokscan.exe	Debug executable
C:\tokscan\Debug\imaglib.dat	Run file which points to data directory
C:\tokscan\Release\tokscan.exe	Release executable
C:\tokscan\Release\imaglib.dat	Run file which points to data directory
C:\tokscan\YACC\driver.c	Source for YACC parser test driver
C:\tokscan\YACC\Test.y	YACC input source (rules)
C:\tokscan\YACC\testinp.dat	YACC parser test input file
C:\tokscan\YACC\test_tab.c	Test output C source from YACC
C:\tokscan\YACC\test_tab.h	Test output C header source from YACC
C:\tokscan\YACC\workpj.mak	YACC parser test make file
C:\tokscan\YACC\workpj.mdp	YACC parser test project file
C:\tokscan\yacclib\Alloca.c	Memory allocation source (from FLEXBISON)

C:\tokscan\yacclib\yaccparser.c	YACC parser source (created from test_tab.c)
C:\tokscan\yacclib\Debug\yacclib.lib	YACC parser link library (debug)
C:\tokscan\yacclib\Release\yacclib.lib	YACC parser link library (release)

Performing Full Image Processing On An Image

TOKSCAN can perform image recognition in two different modes. In the fully automatic mode, an image file is opened, binarization is done, and then TOKSCAN is requested to perform recognition. It automatically goes through the complete process of thinning, vectorization, determination of minimal closed polygons, determination of schematic components, determination of connecting signal lines, and location of circular signal line connectors. The user then identifies circuit inputs and outputs, and requests equation generation.

In the manual mode, the user can "pre-vectorize" and "pre-thin" an image, and save the

results in files which can later be opened along with the image, in order to avoid performing vectorization and thinning again. This is useful when testing is being performed on an image, and it is necessary to perform library analysis or recognition multiple times on the same image.

In this section, we will illustrate the fully automatic mode by providing specific instructions for the test input image "schematic1.bmp". The same instructions apply for all other test images provided with the program.

Instructions for fully automatic recognition on "schematic1.bmp"

1. Start the TOKSCAN program.

2. Verify that the token library file has been located and opened by doing the following:

    a. Select menu item "Token-Library".
    b. Select "Edit Token Library" from the drop-down menu.
    c. Press the "Shape Relationships" button in the "Shape Token Library Update" dialog box which is opened after step b is completed.
    d. Verify that there are six entries in the listbox displayed in the "Basic Shape Linear Relationships In Compound Shape" dialog box which is opened after step c is completed. If there is no data in the listbox, then check that the starting directory in the shortcut is C:\TOKSCAN\RELEASE (assuming drive C:), or that

the program was started using Windows Explorer, as described above.

3. Select the FILE menu item, and then the OPEN menu item (from the drop-down menu).

4. From the file-open dialog box, move to the TOKSCAN\DATA directory, and open the file "schematic1.bmp". If a message box is displayed indicating that there is no user profile file for the selected image, then the starting directory for TOKSCAN is not set correctly. (It should be C:\TOKSCAN\RELEASE, assuming that the C: drive is used).

5. Select the menu item "Image-Preprocessing", and the menu item "Binarization" from the drop-down menu.

6. In the "Select Binarization Threshold Level" dialog, select a global threshold value of 80, and press the "Apply" button. After the image has been binarized, press the "Close" button.

7. Select the menu item "Image-Analysis".

8. Select the menu item "Find And Process Basic Shapes" from the drop-down menu displayed after completing step 5.

9. After step 6 is complete, indicate the circuit input and output points for TOKSCAN, by doing the following:

    a. Press the toolbar button which changes the mouse cursor to the "In/Out Points" mode (third button from the right). The mouse cursor should change, and display the words "In/Out Points".

    b. For each of the four circuit input points (at the top left side of the image), press and hold the left mouse button to display a selection rectangle, and move the rectangle so that it is over the end point of one of the input signal lines. Then, release the button. TOKSCAN should flag the location with a red dot, and with the words "INP-A", "INP-B", "INP-C", and "INP-D".

    c. For the circuit output point (at the right side of the drawing, about half way down), press and hold the right mouse button to display a selection rectangle, and move the rectangle so that it is over the end point of the output signal line. Then, release the button. TOKSCAN should flag the location with a red dot, and with the words "OUT-A".

d. Select the menu item "Image-Analysis".

e. Select the menu item "Generate Schematic Equation" from the drop-down menu displayed when step d is completed.

After completing steps 1 - 9, TOKSCAN should follow all of the connector lines from the indicated inputs to the indicated output, highlighting the lines in blue as it executes, and then it should display an equation at the bottom of the bitmap for the analyzed circuit.

Performing "Pre-vectorization" and "Pre-thinning"

Step 8 above usually takes a lot of processing time, and if an image must be processed repeatedly, it saves time to perform "pre-thinning" and "pre-vectorization", and then to load the results into memory along with the image when it is processed the next time.

To perform "pre-thinning" on the schematic image "schematic1.bmp, for example, do the following:

1. Select the FILE menu item, and then the OPEN menu item (from the drop-down menu).

2. From the file-open dialog box, move to the TOKSCAN\DATA directory, and open the file "schematic1.bmp".

3. Select the menu item "Image-Preprocessing", and the menu item "Binarization" from the drop-down menu.

4. In the "Select Binarization Threshold Level" dialog, select a global threshold value of 80, and press the "Apply" button. After the image has been binarized, press the "Close" button.

5. Select the menu item "Tools", and then "Thin Image" (from the drop-down menu).

6. After the image has been thinned (when step 3 completes), select menu item "FILE", and then "Save Bitmap As" (from the drop-down menu). In the file-save dialog box which appears, enter a file name for the thinned image, and press OK to save it to the hard disk. Make note of the file name used.

To perform "pre-vectorization" on this schematic image after completing step 6 above, do the following:

1. Select the menu item "Tools", and the item "Extract Shape Object Vectors" (from the drop-down menu). This step extracts the vectors and places them in memory.

2. Select the menu item "Tools", and the item "Reorganize Shape Vectors" (from the drop-down menu). This step reorganizes the vectors and saves them in the working database used by TOKSCAN.

3. Select the menu item "FILE", and the item "Save Database File As" (from the drop-down menu). A file-save dialog box will appear, which allows you to assign a permanent file name for the vector database in a library of database files. The file is assigned a file type (or DOS extension) of .dbs. Enter the desired file name, and press okay to save the database. Make note of the file name.

Loading And Performing Recognition On A "Pre-vectorized" And "Pre-thinned" Image

After performing "pre-thinning" and "pre-vectorization", schematic1.bmp can now be processed more quickly by calling up the thinned image file and the database of vectors which were saved. Starting from the point where TOKSCAN is running, and no image is loaded, the following steps should be followed to perform recognition on schematic1.bmp, taking advantage of "pre-thinning" and "pre-vectorization".

1. Select the FILE menu item, and then the OPEN menu item (from the drop-down menu).

2. From the file-open dialog box, move to the TOKSCAN\DATA directory, and open the file "schematic1.bmp".

3. Select the menu item "FILE", and item "Open User Parm File" (from the drop-down menu).

4. From the file-open dialog which is displayed, move to the TOKSCAN\DATA directory, select the file "schematic1.upr", and press OK. This will open the pre-defined user parameter file built for this image. (Note: this step is actually performed automatically whenever an image is opened, and there is a file with the .upr extension and a matching first node in the same directory as the image file). If the user parms are changed and it is necessary to save the changes back into the schematic1.upr file, select the menu item "FILE" and item "Save User Parm File" (from the drop-down menu), enter schematic1.upr as the file name, and press OK.

5. Select the menu item "FILE", and item "Open Database File" (from the drop- down menu).

6. From the file-open dialog, move to the TOKSCAN\DATA directory, select the file "schematic1.dbs", and press OK. This selects the database vector file from the library, and copies it to TOKSCAN's working database.

7. Complete steps 7 - 9 from "Instructions for fully automatic recognition on schematic1.bmp" above.

Steps 1 - 6 can be done very quickly, compared to the time required for the typical thinning and vectorization processes.

Performing Library Analysis On schematic1.bmp

Library Analysis consists of detecting the closed minimal polygons in an image, generating the tokens which describe the shape of the polygons, and saving the tokens in the token library for future recognition. To do this for the schematic1.bmp image, do the following:

1. Complete steps 3 - 7 from "Instructions for fully automatic recognition on schematic1.bmp" above.

2. Select "Find Basic Shapes For Library Update" from the drop-down menu.

3. After step 2 has completed, all detected closed minimal polygons are highlighted in the image in purple.

4. Select a polygon for which the tokens should be saved in the token library. To do this, place the mouse cursor near the polygon, press and hold the left mouse button, move the mouse to create a selection rectangle, place the selection rectangle over the polygon, and release the left mouse button. This will cause the "Compound Shape Object Information" dialog box to open .

5. Use the mouse to select the "Chain #" line of the desired polygon in the upper left listbox (multiple polygons will be selected if the selection rectangle from step 4 overlaps more than one polygon).

6. In the "Add To Token Library With Shape Name" edit box, type the name which you would like to assign to the token list when it is saved in the token library.

7. Press the "Add To Token Library With Shape Name" button. This saves the token list in the library. It can then be used in shape relationships to define a compound shape.

Matched Sets Of Image Files And Supporting Files

There are four test images provided in the TOKSCAN\DATA directory which have been thoroughly tested for proper recognition with TOKSCAN. Each image file is accompanied by a vector database file, a user parameter file, and a thinned image file, as shown in the following list.

C:\tokscan\data\adder.bmp	Scanned image file for full adder schematic
C:\tokscan\data\adder.dbs	Vector database for full adder schematic
C:\tokscan\data\adder.upr	User parm file for full adder schematic
C:\tokscan\data\adderthin.bmp	Thinned image file for full adder schematic
C:\tokscan\data\exercisescan.bmp	Scanned image file for exercise schematic
C:\tokscan\data\exercisescan.dbs	Vector database for exercise schematic
C:\tokscan\data\exercisescan.upr	User parm file for exercise schematic
C:\tokscan\data\exercisescanthin.bmp	Thinned image file for exercise schematic
C:\tokscan\data\multiplexer.bmp	Scanned image file for multiplexer schematic
C:\tokscan\data\multiplexer.dbs	Vector database for multiplexer schematic
C:\tokscan\data\multiplexer.upr	User parm file for multiplexer schematic
C:\tokscan\data\multiplexerthin.bmp	Thinned image file for multiplexer schematic
C:\tokscan\data\schematic1.bmp	Drawn image file for test schematic
C:\tokscan\data\schematic1.dbs	Vector database for test schematic
C:\tokscan\data\schematic1.upr	User parm file for test schematic
C:\tokscan\data\schematic1thin.bmp	Thinned image file for test schematic

There are additional image files which have not been tested with TOKSCAN that are included in directory TOKSCAN\EXTRA\EXTRAIMAGES. Successful recognition with these images will require testing and careful adjustment of the user parameter file. They may also require some modification to TOKSCAN: debugging has been carried out completely for the four sample images, but time constraints for the project did not allow complete debugging for all of these additional images. With some additional debugging, TOKSCAN should

successfully recognize most of the included extra images. REMEMBER: this is a prototype program which demonstrates that the techniques implemented will work: it has not yet been tested to the point of being ready for use against any desired schematic image.

Modifying And Testing The YACC Parser

The directory TOKSCAN\YACC contains the source code necessary to test the YACC parser separately from TOKSCAN, using a small driver program named driver.c. A Visual C++ project has been set up in this directory to build the test parser with the test driver. The project workspace file is named "workpj.mdp".

To modify the YACC parser rule set, and recreate the test parser, do the following:

1. Open file TOKSCAN\YACC\test.y, and modify the YACC rules as needed. Then save the changed file.

2. Install FLEX/BISON, using the directions provided with the download, and run the test.y file through it to produce C source code for the parser (called test_tab.c). Update the existing test_tab.c provided in the TOKSCAN\YACC directory with the new version, and build the workpj project from within Visual C++.

3. A test input file with tokens that have the same format used by TOKSCAN is provided in the TOKSCAN\YACC directory, named "testinp.dat". To execute the test parser built in step 2, open a DOS window, and enter the command:

   workpj < testinp.dat > testout.txt

   This will execute the parser using the test input file, and will produce a text output file with the results of the parse called "testout.txt". The proper results for the current version of the parser are provided in the existing file TOKSCAN\YACC\testout.txt.

4. After the test version of the parser has been tested, it must be added to the link library which is included in the TOKSCAN project. To accomplish this, do the following:

   a. Rename the parser c source code file from test_tab.c to yaccparser.c, and copy it to the TOKSCAN\YACCLIB directory, overlaying the existing yaccparser.c file.

b. Rebuild the TOKSCAN project. The yacclib.lib library will automatically be rebuilt as a part of the overall project build, and the parser will be link edited into TOKSCAN.

Modifying And Testing The TOKSCAN Project

TOKSCAN can be modified using Visual C++, version 4.0 through 4.2 (Enterprise

Edition). It was developed using version 4.2. To load the project, open the workspace file TOKSCAN\imagelib.mdp. See the comments at the beginning of part 2 of this manual for more information.

Introduction To The Class Structure In TOKSCAN

After the project workspace file (TOKSCAN\imagelib.mdp) has been opened in Visual C++, all of the user-defined C++ classes can be seen in the class "tree view" window. Visual C++ provides fast access to the source definitions of each class, and of each member variable and member function through the use of this tree view. The user can expand the view by double-clicking the mouse on an entry. Double-clicking the mouse on an entity inside of a class causes the source code to be opened at the location where the entity is defined.

The TOKSCAN project was originally generated using Microsoft's "App Wizard", and the original set of classes generated by that tool were retained in the final structure. Numerous other classes have been added, with many interrelationships between the classes.

Visual C++ has a class browser utility which helps the user understand the structure of the program, and find where classes are referenced. It is a good idea for a new user to review the browser file provided in the Zip Disk (TOKSCAN\DEBUG\imagelib.bsc). The following page contains a list of the user-defined classes in the project, with a brief explanation of the function of each.

Classes Generated By The Microsoft "App Wizard" (Heavily Modified)

CImageLib	Main application class

CMainFrame	Main frame window class (of MDI interface); user defined program initialization is done here
CCInputBitmapDoc	Document class which holds all image data
CCInputBitmapView	View class which controls display of Doc class data
CChildFrame	Child window class for MDI interface
CAboutDlg	"About" dialog box class

User-Defined Classes

CBinarize	Dialog box class which controls image binarization
CChainStartCount	DAO (Microsoft Data Access Object) class which gets a count of the number of vector chains saved in the working database
CCompoundImageCoordIdx	Manages a hash table of keys which index a set of CCompoundImageObject objects
CCompoundImageObject	Compound Shape Object implementation
CCompoundShapeDialog	Dialog box class which drives the "Compound Shape Object Information" dialog box
CConnectionMatrix	Implements the connection matrix used to analyze the connections between schematic components
CDAOChainStartSet	DAO class which retrieves vector information from the database
CDAOVectorChainSet	DAO class which retrieves vector information from the database
CImageObject	Simple shape object implementation
circularregion	Supports "circular searches" in the image bitmap around a specific set of coordinates
CkFillFilter	Dialog box class which controls noise reduction

CLineSmoothParms	Dialog box class which supports the line smoothing parameters which are set by the user
CProcessLog	Dialog box class which supports the process log, which is used for debugging
CProfileDB	DAO class which supports the retrieval of user parameter information from the working database
CProgressDlg	Dialog box class which supports the progress bars that indicate time remaining on long running tasks
CShapeLinearRelDialog	Dialog box class which supports the "Basic Shape Linear Relationships In Compound Shape" dialog
CShapeMap	Supports a hash table which references a collection of CCompoundImageObject instances
CShapeNumEntryDialog	Dialog box class which supports the "Shape Selection Data" dialog
CShapeObjectDialog	Dialog box class which supports the "Chain Shape Object" dialog
CShapeTokenLibDialog	Dialog box class which supports the "Shape Token Library Update" dialog
CTokenLibData	Supports the retrieval and update of token library information in memory, and file i/o
CTraverseVectorChain	Supports traversals through a set of vectors contained in a shape object
CUserParm4	Dialog box class which supports the "Shape Object Parameters" dialog
CUserParms	Dialog box class which supports the "Curvature And Junction Margin Of Error" dialog

CUserParms2	Dialog box class which supports the "Perimeter Values" dialog
CUserParms3	Dialog box class which supports the "Closed Polygon Search Parameters" dialog
CUserParms5	Dialog box class which supports the "Token Edits" dialog
ImageVectors	Supports the "thick line" vectorization process
mathfunctions	Provides necessary mathematical functions such as distance measurement, arccosines, etc.

NOTE ON RUNNING THE TEST IMAGES ON DIFFERENT PC'S

In some cases, the test images will not process correctly when this program is transferred to another PC system. The problem has to do with the Window's color palette, which varies from system to system, and which causes the test bitmaps to display with slightly different intensities in some cases. If this problem occurs in a new installation, it may help to load the test bitmap, then load an image color file which has been provided in the ZIP disk in the DATA directory. This will reset the grayscale intensity values to those originally used to test the image.

There are three image color files provided, one for each of the three scanned test images (schematic1.bmp should not have this problem), with names adder.cif, exercisescan.cif, and multiplexer.cif. To apply the color information file to adder.bmp, for example, open adder.bmp, then from the FILE menu, select OPEN IMAGE COLOR FILE. When the file dialog box opens, select adder.cif as the file to load. This will reset the bitmap display, and image recognition can then be performed. The changed image can also be saved as a new bitmap.

NOTE ABOUT DATA FILE PATHS

If you use a drive/directory other than c:\tokscan for the installation, make sure that you change the path name in file imaglib.dat to match the directory used. This file is located in the RELEASE, DEBUG, and TOKSCAN directories on the Zip disk.

VITA

James A. (Jim) Giles has a Bachelor of Arts degree from the University of South Florida in Mathematics, and expects to receive a Master of Science degree in Computer and Information Sciences from the University of North Florida, August, 1997. Dr. Yap S. Chua of the University of North Florida is serving as Jim's thesis advisor.

Jim has a background in business data processing in the life insurance and transportation industries, and on-going interests in mathematics, engineering, image processing, and computer networks.