Automated Wiring Analysis of Integrated Circuit Geometric Data

by

Charles R. Lang

Submitted in Partial Fulfillment of the
Requirements for the Degree of Master of Science

CALIFORNIA INSTITUTE OF TECHNOLOGY
Computer Science Department

August 9, 1979

2891:TR:79

Abstract

Methods are presented by which the wiring data of an NMOS integrated circuit may be extracted from its mask information. The procedures involved utilize the capabilties of a general purpose polygon package. The polygon operations are defined to enhance their use in this application, however, the package is suitable for other uses, such as, design rule checking. The analysis is performed on hierarchial symbol definitions of mask geometry. The geometry is presumed to be described in CIF 2.0 (Caltech Intermediate Form). The analysis attempts to recognize three basic types of structures in the geometry: 1) transistor devices (and capacitors), 2) local interconnection structures and 3) global interconnection structures. Definitions are put forth for the distinction of global and local wires. The data extracted from the symbol geometry is the percent utilization of each symbol's area by each of the three types of structures. The purpose behind the extraction of this data is its use in the development and evaluation of wiring models for custom NMOS IC design. Two approaches are presented which extract such data. The first is heuristic and depends on built-in assumptions of how the NMOS process is generally used. This technique loses accuracy if a design style falls outside of these assumptions. The second technique is a method by which the topology of design may be extracted from the geometry. The geometric objects, from which devices and interconnections are made, are preserved, such that the wiring information can be obtained precisely. This method is complex and requires considerable computation, however,

the topology extracted may also be used to verify the geometric data against the original design topology.

Table of Contents

List of Figures

Chapter I

Introduction

1.1 Background

The design of complex integrated circuits requires consideration to be given to the resources utilized by the interconnection of devices and subcircuits. By most measures, the percentage of available area used by these interconnections is a large portion of the integrated circuit. The ability to estimate the size of a design and hence its manufacturing cost is very important to the eventual marketing or internal use of the part. Without appropriate models for the wiring of integrated circuits such estimates are usually poor. In addition, automatic design aids, such as, placement and routing programs are made more costly and less reliable if good interconnection models are not available.

In the past, considerable effort has been given to solutions of these problems for printed circuit cards. Some of these techniques have found applicability in the design of integrated circuits. However, two major differences exist between these two media. 1) The silicon chip is a homogeneous medium. Devices, wiring and subcircuits all share the same space. With printed circuit cards, devices of all kinds are packaged outside of the wiring space such that the device or the subcircuit makes

no demand for area on the interconnection medium. 2) The subcircuits and devices of a silicon chip have various input and output connections. The physical position of these "pins" need not be fixed but can be moved or swapped as required by the designer. The layout of a printed circuit board is restricted in that the devices and subcircuits (components) to be interconnected have a standard pinout that cannot be changed at the whim of the layout designer or layout program. This ability of a chip designer to optimize the pinout of various devices and cells is not always utilized in order that the cells be standardized in the same way as SSI/MSI parts have been standardized for printed circuit cards.

This paper is restricted to consideration of the design of large scale integrated circuits built using the NMOS process. Even within this area a variety of design styles exist. Gate array, standard cell and true custom methodologies have been used to design NMOS circuits. A model predicting the wirability of various sizes and types of gate arrays has been developed by Heller[1] and has been used successfully to predict the wiring space requirements and achievable gate density of several master slice designs. It is only by the collection of data from a number of high quality IC designs that such models can be developed or verified.

In addition to the design methods above, a regularized style which integrates the circuit topology and layout with the intended function of the design has been put forth by Mead and Conway [2]. This style is characterized by a hierarchy of circuit blocks which are defined by the designer in a top down manner but with close regard for the

physical and electrical constraints to be encountered at the low levels of the design. The architectures of machines are planned with data and control path topologies that will layout and fit together in a regular manner when implemented in mask geometries. What would otherwise be "random" logic is performed by PLA's and ROM's. Extensive use is made of "pass" transistors as steering logic and dynamic circuits are frequently utilized. The resulting designs are built using a small number of basic cell designs which are placed regularly about the IC with little or no "random" interconnections. In power consumption, area requirements, logic function and particularly in ease of design, these techniques offer advantages over traditional design styles.

1.2 Data Collection for Wiring Analysis

It is important that wiring models be extended (or new ones created) to include this type of design. Early investigations by Heller[3] indicate that the wiring models used for master slice designs do not predict the data collected from such regularized IC designs. The development of these models and those for future design techniques require massive amounts of data to be collected and evaluated using real IC designs. This thesis presents an automatic means of analyzing the mask geometry of integrated circuit designs to provide various measurements indicating the utilization of the chip by devices, interconnecting wires and subcircuits.

The data to be collected must preserve the hierarchy and partitioning specified by the designer. Otherwise, the

analysis will be limited to the lowest hierarchial level, that of transistors. More interesting though, is the wiring and space requirements of higher level functions, such as, registers, ALU's, PLA's, et cetera. If experience can be gained with a large number of designs, the data taken concerning these common functions can make possible good predictions of the areas required by them in future designs. This knowledge, if used in planning the chip, could result in fewer subsequent layout iterations and an early, accurate prediction of overall die size. To preserve the designer's structure in the analysis of the design, the geometry specification expressed in CIF (Caltech Intermediate Form) is used as the input to the analysis program. CIF is defined in Mead and Conway [2]. Central to the choice of CIF is it's ability to define geometry symbols in terms of basic shapes and subsymbols. Also, it is hoped that CIF will become a standard means of specifying IC geometry allowing a wider range of designs to be analyzed.

Before going further, a philosophical question must be answered. Why analyze the wiring of integrated circuit designs? Why not just build them and be done with it? Analyses of all kinds serve to aid in the development of predictive models in the hope that these can be subsequently used to reduce future costs and errors. The fact that project managers are not satisfied with the accuracy of size and cost estimates (designs <u>always</u> overrun initial estimates) and the fact that good placement and routing procedures do not exist for the truly custom layout of IC's, point out that the existing models for the wiring of custom IC's are not acceptable. However, the advent of the silicon compiler and other very fast automated design

generators could remove the need for size estimates and some of the traditional design tools. An IC design program know as Bristle Blocks has been developed by Johannsen[4] which can automatically assemble, layout and specify geometry for a large class of finite state machines in a matter of weeks. Such systems can remove the need for predictive tools if they are able to complete the real design in such a short time.

It is clear that for master slice designs wiring space prediction must be accurate since one master slice image must accomodate a large number of logic designs (part numbers). If the amount of space made available for wiring is inappropriate, either poor utilization of the devices on the die will result or the die will be larger than necessary. In both cases a higher cost will be paid to produce a given function. True custom designs have a different economy. Since each part need not be larger than necessary, the manufacturing cost is determined by the die size (and other factors not related to its wiring).

Why then, do we need these analysis tools and models in custom design? True custom IC design requires the mapping of a hierarchy of successively refined circuits onto the planar space of the chip. If this is done automatically, the mapping of each node of the tree requires that subcircuits and devices be placed and then interconnected. Since the subcircuits must be mapped (implemented in silicon) prior to their use in higher level circuits, this operation is recursive and continues to fall through the hierarchy until circuits built soley with primitive elements (transistors) are encountered. These circuits are placed and routed and the operation moves one level up the

tree, repeating the operation until the entire design has been implemented. At each node in the tree decisions must be made which place the devices and subcircuits in a relationship with each other which will result in a minimum amount of area used for wiring and an aspect ratio and pinout which are suitable for instances of the circuit in higher level functions. To make these decisions there must be some model that can be used to allocate a near optimal amount of space about the devices and subcircuits to be used for interconnections. The model would have available as input the size and number of devices, the size and pinout of each subcircuit. To build and evaluate such a model, the analysis proposed here must extract, for each node in the design hierarchy, the size and number of the devices, the size and number of pins (input/output connections) of each subcircuit and the area utilized in the design to interconnect the objects.

Another purpose behind the study of wirability is its use as a measurement tool. Companies and individuals frequently must make decisions which select one fabrication process out of several or one design aid over another. To make intelligent decisions some type of data is required from the candidates which embodies experiences with that process or design aid. Usually people rely on intuition or "gut reaction" for such data because good measures with which to compare the alternatives are not available. The extraction of simplistic measures such as the number of devices per unit area can be of importance since this is a good indication of how efficient the interconnection structures of the process are or how effective the placement and routing algorithms of a CAD package are. All area and distance measures should be in some type of

process independent unit to allow separation of wiring data from process and lithography, the lambda unit used by Mead[2] is suitable for this purpose. If, for example, it is necessary to decide between a master slice approach and a polycell style, such measures as the percentage of area used for wiring (outside the cells or book functions) are an accurate measure of how well each utilizes the chip. The data can be collected from several designs, even if they differ in design rules. The data is made comparable by expressing dimensions in lambda and/or percentages. Such data, if available to management, can make easier the long range planning of a company's technology path and make its success more probable.

1.3 Overview

It is the purpose of this thesis to show how wiring data may be extracted automatically from the CIF specification of an NMOS chip. The operations involved are based on a set of polygon manipulations and are applied hierarchially to the elements of the design. There are two approaches to the gathering of this information.

The first approach is heuristic and interprets the mask data correctly in the vast majority of cases. The exception cases will not affect the outcome and resulting statistics that are accumulated in any significant way. The heuristic method is straightforward and requires less processing than more exact methods. The data taken from IC designs is of little use in any absolute sense but is very useful when compared with data taken from other chips. Thus, it is more important that the data be extracted by a

standard method than that the method be absolutely precise in its calculations.

The second method is precise and much more extensive in complexity and processing time than the first. However, to justify itself it produces a reconstruction of the circuit topology as determined by the process masks. This information can be compared with the original specification of the chip topology made by the designer to verify the chip has be implemented faithfully. The actual comparison is beyond the scope of this paper. The extraction of the topology is an operation currently performed in design automation systems. However, if in this process, the program accounts for the silicon resources used in building the various circuit structures (transistors, interconnection wires, subcircuits), the data required by the wiring model can be accumulated.

Both methods are presented and explained, as well as, the polygon operations on which they are based. Some demonstration of their operation is also given using small NMOS circuits. The topology extraction requires more computation than the heuristic method. Insufficient test runs were made with both to precisely determine the performance difference between them. However, it can be safely said that the heuristic methods improve run times by at least a factor of three and probably more for large designs.

The programs were written entirely in SIMULA and the important sections of code are listed in the text and in the appendices. The SIMULA language provides a very good environment for writing such algorithms but incurs an

overhead that makes the execution of large polygon operations slower than it would otherwise be. In addition, the garbage collection system contains an error preventing large, long programs from running to completion. This problem has prevented the use of test cases of all but a small size. The problems on which the program has executed demonstrate that such data can be extracted from IC mask data automatically.

Chapter II

Definition of Wiring Statistics

2.1 Data Collection Requirements

Several types of data should be collected from IC designs. To be useful for the purposes described an analysis program must recognize various device and interconnection structures in the shapes appearing in the masks. This chapter selects what objects are to be recognized by the program and what computations are to be made using these objects.

The figures describing NMOS geometry in this paper utilize a standard set of colors to the different process masks involved. Blue represents aluminum, red represents polysilicon, green represents diffusion and black represents contact cuts. Black is also used here to indicate ion implantation.

To satisfy the requirements of wiring model verification or development, several features of the design must be extracted. As stated previously, the hierarchy of the design is to be preserved in the data extracted, such that those objects which belong to higher level structures are separated from those which are particular to the portion of the design at hand.

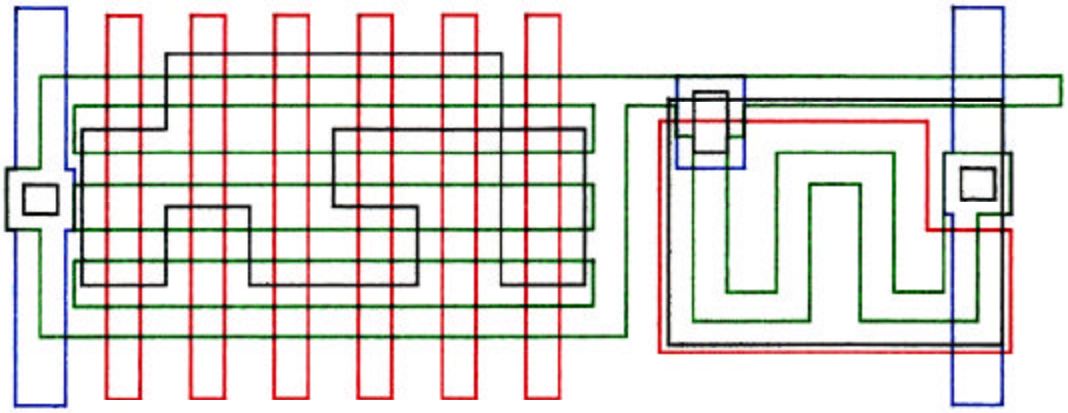


FIGURE 2.1
LAYOUT OF A COMPLEX GATE

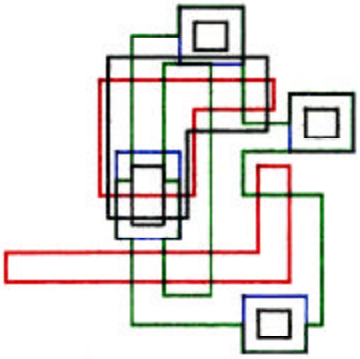


FIGURE 2.2
LAYOUT OF A SUPER BUFFER

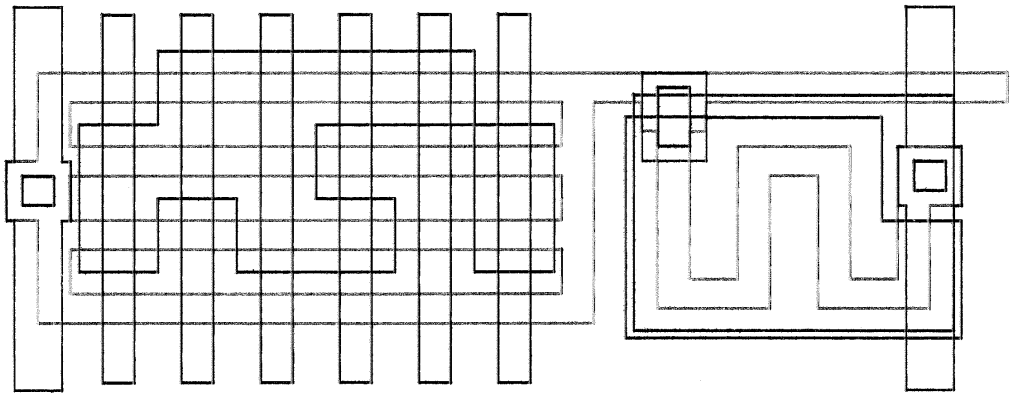


FIGURE 2.1
LAYOUT OF A COMPLEX GATE

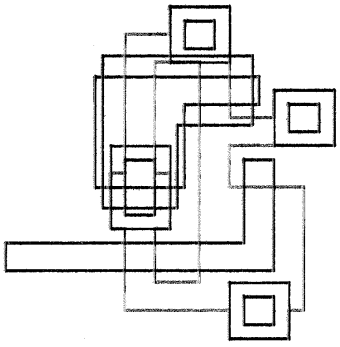


FIGURE 2.2
LAYOUT OF A SUPER BUFFER

The wiring data is to be extracted from mask shapes described by CIF 2.0. The design hierarchy is expressed in CIF as a tree of symbol definitions. Each symbol definition may consist of constructions of boxes, wires (tracks), simple polygons and instances of subsymbols. Recursion is not allowed, however, the depth of symbol definitions is not restricted. A complete definition of CIF 2.0 is found in Mead and Conway [2]. The syntax and semantics of CIF do not restrict the manner in which the designer specifies the IC. The wide range of variation possible within the bounds of CIF is a major cause of complexity in the extraction of wiring data. Presently, there does not seem to be a reasonable set of restrictions that would be acceptable to the users of CIF and it is desired that the program be operable on a wide range of designs. Thus, to be useful, the recognition of objects and the extraction data must be as general as possible and perform properly regardless of the designer's CIF style.

For each symbol in the hierarchy one fundamental distinction can be made. Devices can be distinguished from interconnection. This is not an easy or obvious classification of the geometrical shapes. Devices can include subsymbols and these are identified by the syntax of CIF. The remaining devices are transistors of various types. Of these, some may not have been required by the designer to perform the function of the chip. They may be present only as a less than optimum interconnection. Figure 2.1 illustrates a complex gate of a type which could find use where both the active high and active low senses of its input terms are available. In a case like this it can be seen that some of the transistors are not being used as such but as a connection from one side of a polysilicon

wire to another without requiring use of the metal layer. In addition, the use of NMOS transistors as capacitors for bootstrapping or charge storage purposes must not be confused with wiring structures.

Once device structures are separated from interconnection structures, further operations are necessary to winnow out wiring that belongs essentially to structures higher in the hierarchy than the one being studied. Such wires are those that make a demand on the area of the symbol but serve to interconnect objects not part of the current symbol. These portions of the wires in a symbol are termed global and those which exist to interconnect parts of the current symbol local. This is not a clear distinction since wires may pass through a symbol in order to connect external objects but may also connect to internal devices at one or more points. It seems reasonable to expect that some portion of such wires may be considered local and the remaining portion global.

Once these objects are separated from each other, it is a simple matter to compute their area. However, the total area of the symbol is not well defined especially if it does not fully utilize the three important interconnection layers (poly, diffusion and metal). To correctly account for unequal use of the layers, the area of the symbol must be represented as the sum of the area it requires on each of these three layers. The various objects can then be expressed as a percentage of the symbol's total area. Also, interconnection area should be expressed as wire length by dividing its total area by the center to center distance between wires (usually the minimum width plus the minimum spacing). The area computations provide the

measures of how efficiently the circuit was implemented and for high quality designs, represent a characteristic of the particular type of function that has been implemented.

It may also be necessary to determine the number of components making up the circuit (devices and subsymbols) and the average number of terminals (input/output connections) made to these components. This is the information that would normally be used in an a priori prediction of the necessary wiring space of the circuit. A clear example of its use would be in the estimation of the Rent exponent (Landman and Russo [5]) for LSI circuits. Most design systems make available the structural or topological data describing a design from which the parameters of the Rent equation can be extracted more easily than from the mask geometry. However, as part of the topological data extraction, the connections between components must be determined for verification with the original design.

The three basic elements that must be identified to provide the wirability data are: 1) the area occupied by devices (transistors); 2) the area occupied by local interconnection wires and 3) the area occupied by global interconnection wires that are present in the definition of a symbol. To give meaning to these numbers, the area of the entire symbol must be determined. An important measure can be derived by subtraction of the foregoing from the total area of the symbol giving the unused or wasted area of the design. Once the topology of the symbol is known, finer measures can be formulated, such as, the area occupied by power distribution structures. Most statistics can be expressed on a per layer basis giving some insight

into the manner in which different layers are utilized.

It should be noted that although these quantities are to be generated on a per symbol basis, there is the opportunity for any degree of instantiation of the symbol hierarchy in order to accumulate statistics for the entire design or any portion of it at the transistor level.

2.2 Definition of a Device

It is necessary that a standard be set that makes clear what is meant by a device. In some sense, all structures of an integrated circuit design are devices. To some extent, even what are meant to be wires act as series resistors and capacitances to the substrate. The primary distinction between such parasitics and a depletion load, for example, is the intent of the designer. That is, devices can be defined as those structures which act as active or passive circuit elements and were both placed and desired by the designer. Anything else must provide isolation between devices or connections between devices, the latter being wires in the ideal sense.

What is now needed is a more workable definition of a device which can approximate the intent of the designer. With few exceptions, NMOS devices must involve a polysilicon structure placed physically above a diffused area. Exceptions take the form of diffusion wires used as resistors, such as in the resistor ladder of an analog to digital converter. Every occurrence of polysilicon above diffusion is not a transistor. An example is a butting or buried contact, both have coincident poly and diffusion but

both are used as a portion of an interconnection. Other instances of poly and diffusion together may actually form a transistor but could be used as a wire (the implanted gate inputs of Figure 2.1) or could be used as a capacitor, which is another variation of a device. If the designer draws a device, there must be an intention to cause a significant interaction between the gate of the device and its drain and source. Once apparent devices which act as contacts are eliminated, this definition can serve to separate the remaining structures which are, in fact, MOS transistors but are not intended to be used as such by the designer. This does not eliminate capacitors, which though they are not used as transistors, there is nevertheless a significant interaction between their gate and drain/source connections making them devices.

This definition is difficult to apply in practice since it requires some determination of what is "significant" interaction between the gate and channel region. An example of the confusion possible is a comparison between the implanted input transistors of Figure 2.1 and a second stage pullup of a super buffer as shown in Figure 2.2. The first has been defined to be other than a device, the second is definitely a device yet both appear geometrically the same. Resolution of problems such as this requires some analysis of the topology of the circuit to second guess the intention of the designer. Since such analyses are generally not deterministic, it may be preferable to apply some simplification to solve the problem. It is expected that the contributions made in error by such methods will be insignificant.

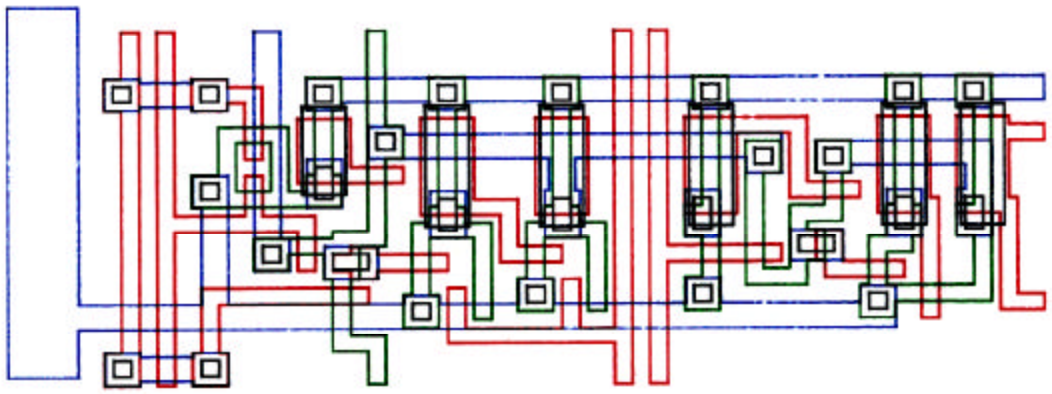


FIGURE 2.3

LAYOUT ILLUSTRATING GLOBAL and
LOCAL WIRE

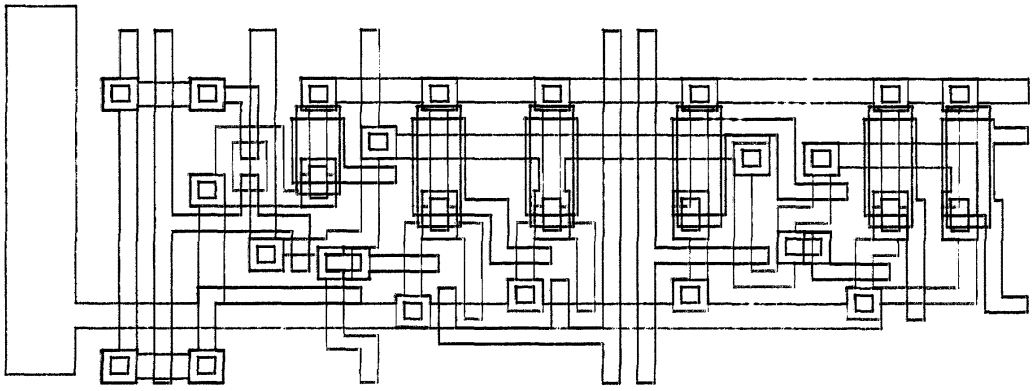


FIGURE 2.3

LAYOUT ILLUSTRATING GLOBAL and
LOCAL WIRE

2.3 Global versus Local Wire

One of the most difficult operations is that of separating local wiring structures from global ones. This results in part from the lack of a clear definition distinguishing the two. The attempt is made here to adequately define the differences between local wire and global wire.

To some extent, every connection by a symbol to objects external to itself involves the use of some global wire. It is possible that a particular wire in a symbol may only connect external objects. The other extreme is the wire which connects only internal objects and must surely be considered a local wire. Figure 2.3 shows a typical low level symbol which embodies wires some of which are clearly global or local and several which are not clearly either. The wires which run vertically from top to bottom clearly place a demand on the area of the cell in order to connect the next higher cell to the next lower one. This would indicate at least some portion of these wires must be considered global.

Any rule applied here to separate the global part from the local part will be arbitrary, depending on exactly what information one expects to derive from the data. It is important that some standard rule exist so that the information derived from different IC's will be comparable.

A definition has been chosen with some intuitive basis and partially due to its simplicity. Also, this definition has already been used by Heller[3] in the manual extraction of wiring data. It is desirable that the data extracted

automatically be computed on a similar basis with that previously collected by hand.

The definition defines global wire to be that portion which makes contact with an external object up to but not including any contact made to objects within the symbol. Thus wires which make no internal connection are determined to be entirely global and those which make no external connection whatsoever are defined as local wire. For simplicity, contacts with other mask layers are considered to be internal connections. This last qualification causes some confusion with wires that traverse several layers in making external connections. Parts of these, by this rule, are considered local wire.

As with the definition of devices, these rules do not always perform properly in practice, nor are they simple to implement. However, it is more important that a standard method be set for separating global and local wire than that the absolute numbers extracted be perfect. Therefore, simplifications will again be used where they are expedient to avoid expensive processing to handle exceptional circumstances.

2.4 Wiring Statistics

Once devices are recognized and remaining interconnection structures separated into global and local wire, several useful figures can be computed. For each CIF symbol, a wiring analysis program should compute the following for each layer and for the symbol as a whole:

1) area of the symbol (square lambda)
2) area and percent of (1) occupied by devices
3) area and percent of (1) occupied by subsymbols
4) area and percent of (1) occupied by connections
    a) area and percent area occupied by local wire
    b) area and percent area occupied by global wire
    c) (a) as a percentage of (4)
    d) (b) as a percentage of (4)
5) unused area

These quantities can be used to characterize part or all of an IC design and permit the development of wiring models for all types of NMOS LSI designs.

Chapter III

Description of Polygon Manipulations

3.1 Polygon Requirements

Discerning structures in the geometry of the masks of an integrated circuit requires the ability to manipulate two dimensional geometric shapes. In some cases, these shapes can be restricted to rectangles, or to orthogonal shapes, or to orthogonal shapes including 45 degree lines, etc. If restrictions can be made, simplifications usually result in the programming of operations on them. It may always be expected that at some point exceptions will have to be made and the restrictions dropped. The authors of automated design aids have never been long successful at limiting the use of a technology by legislating rules beyond those required by physical processes. It is for these reasons that the wiring analysis program must be capable of performing its function on arbitrary IC designs. To accomplish this it must be capable of performing operations on general two dimensional shapes.

The shapes commonly found in integrated circuit design are rectangles and wires (or tracks). Wires, as used here, indicate some path and an associated width. Polygons represented as a sequence of points enclosing an area may also be found. It is not unusual to encounter wires and polygons whose boundaries are non-orthogonal. At the

present time, the machines which produce integrated circuit masks do so using rectangular apertures. It is possible that circular apertures could be used in the same way as Gerber photoplotters are used to make printed circuit artwork, avoiding some of the processing problems associated with sharp corners. Complex shapes, such as the diffusion area of Figure 2.1, are specified using some set of the shapes above. In this way shapes with holes in themselves may be generated on the masks. In the manipulation of such shapes, it may be necessary to treat the entire enclosed area (with or without holes) as one entity. This leads to the concept of a general polygon as a collection of one or more sheets, where each sheet is a sequence of edges which enclose some area. Sheets must have a sense which indicates whether they are holes or substance. The edges of the sheets should be either straight lines or circular arcs to accomodate certain polygon manipulations which may produce nonlinear results.

The operations required upon polygons include the set of common logical operations but more significantly the ability to inflate or deflate shapes. Most of these same capabilities are central to design rule checking. Design verification requires many of these functions and may well be included as part of the same program.

The definition of polygons having multiple sheets and both curved and straight edges provides consistency among the operations. The various logical operations may result in single sheets being fragmented, however, the collection of fragments is also a single polygon. Inflation (bloating to some) and deflation, if thought of as the locus of points a particular distance within or beyond the original boundary,

will generate circular arcs at the vertices of the edges. Again, this definition of a polygon permits such nonlinear edges.

A description of the algorithms and techniques required to implement a polygon package of this type is found in Sutherland [6]. A package has been implemented in SIMULA by Sutherland and is also described in the referenced papers. The package goes as far as implementing all the attributes of edges, sheets and polygons including the self intersection of polygons, inflation (negative and positive), and the retracing of the sheets of a polygon. This permits the edges of a polygon to be intersected and the intersection points inserted into the sheets. The sheets of the polygon may then be retraced such that the edges of sheets do not cross edges of themselves or other sheets and are, at most, coincident with other edges. Edges are directed and thus have a right and left side from the point of view of someone traversing the edge in its defined direction. By convention, the substance of a sheet is always to the left of its edges, thus sheets which are outlined clockwise are holes, those outlined counterclockwise enclose substance. This polygon package works and has been used in the first implementation of a wiring analysis program.

The abilities provided by the polygon package as it stands are insufficient for use by the wiring analysis. The package lacks the means to build polygons of the type described from CIF text. Also, the basic logical operations, such as intersection, union and difference must be constructed from the attributes of the package. A SIMULA skeleton of the desired operations and structure

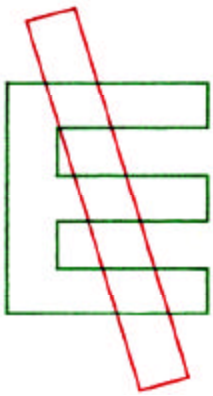
(from a user's point of view) of a polygon package suitable for wiring analysis, topology extraction and design rule checking is listed in Appendix I. The listing is commented to explain the function of the items shown. The polygon package of Sutherland [6] has been extended to provide these functions. Appendix V lists the Simula code required to make the extensions. In addition, procedures have been written to build a database file from the CIF specification and make the data available to the polygon package such that polygon objects can be built from the CIF through a database file.

The purpose of the database file is to permit the execution of the analysis on a substantial amount of data without the need for data to occupy more than a small amount of the addressing space of the computer (DEC PDP-20). The database manipulation programs were authored by Ullner [7] and provide relational abilities, as well as, random access to disk files. Appendix IV is a listing of a program which reads CIF files and builds a database file. The definition of the database relations and tuples is not general insofar as it is suited only to the storage of mask geometry. However, it satisfies the needs of representing CIF commands. The program (called CIFX) makes use of a standard CIF parser written by Tarolli and Rowson [8]. CIFX allows options to be set which control whether or not all geometric shapes are converted to simple polygons of the type understood by CIF. The analysis program requires that all CIF objects (boxes, wires and roundflashes) be translated to CIF polygons (represented by a sequence of points). CIFX is able to translate a CIF 2.0 file into a database at a rate of about 10 lines of CIF per CPU second on a DEC-20.

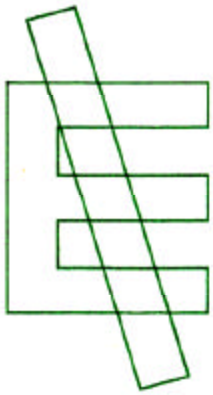
3.2 Logical Operations

Given the general type of polygons described above, a small set of operations on them is all that is required to fill the needs of the wiring analysis program. A complete list of these operations and descriptions of their function is in Appendix I. Beginning with the Sutherland [6] polygon package, some additional procedures are required in order to do logical operations on polygons.

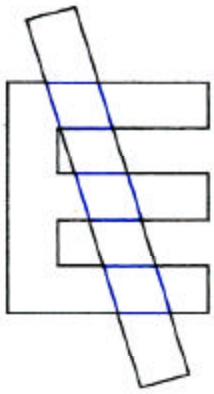
If two polygons (each with any number of sheets) are to be intersected or unioned, they may be combined into a single polygon containing all the sheets of each. The resulting polygon may then be selfintersected and its sheets may be retraced. The areas bounded by the sheets of the polygon may be said to be wrapped by the polygon. The wrap number of a point in the plane is the number of times a cursor would travel fully around the point in a counterclockwise direction as it traced out the edges of each sheet in their defined direction. The wrap number is more fully discussed in Sutherland [9]. The combination of two polygons which are originally well formed each having no selfintersections will combine to enclose their original areas with a wrap number of 1. In areas where sheets of one polygon overlay sheets of another, areas will be enclosed with a wrap number of 2. This is illustrated in Figure 3.1. A red box is combined with the green "E" to form a single multi-sheet polygon. The polygon is selfintersected and its sheets retraced. In the second part of Figure 3.1 the new set sheets is shown with black used to trace the sheets which separate an area of wrap number 1 from an area of 0. Blue



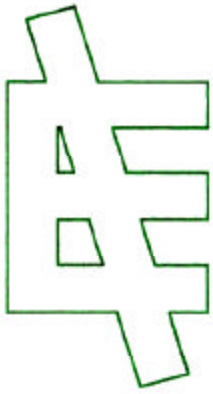
TWO POLYGONS



COMBINED



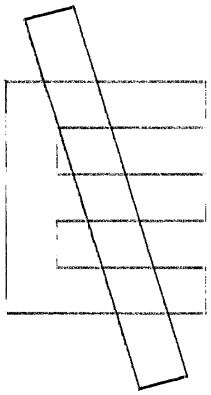
SELFINTERSECTED



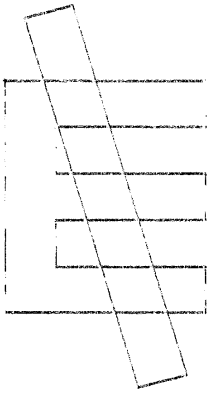
TRIMMED

FIGURE 3.1

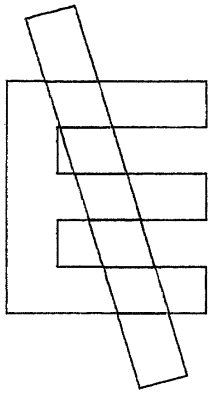
SELFINTERSECTION AND RETRACING OF POLYGONS



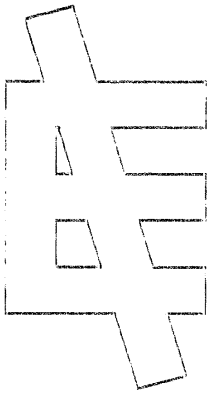
TWO POLYGONS



COMBINED



SELFINTERSECTED



TRIMMED

FIGURE 3.1

SELFINTERSECTION AND RETRACING OF POLYGONS

is used to trace those sheets which separate 2 from 1. This serves to illustrate both the retracing of sheets and the concept used in trimming sheets from a polygon in the generation of logical operations.

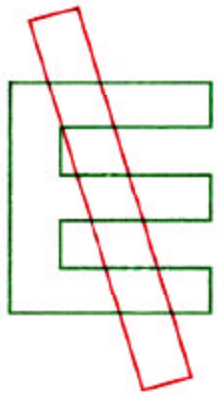
The trimming routine must compute the wrap number of points inside the sheet and the wrap number of those outside with respect to the entire polygon. Two points must be selected, one inside and one outside of each sheet. It is not possible, given any particular edge to compute the location of two points and be sure they will be in the desired relationship to the sheet. The operation proceeds by selecting an edge, computing the location of two points that will probably straddle the boundary of the sheet and computing their wrap numbers. If the points are on opposite sides of the sheet boundary then the wrap numbers will differ. If they do not differ, then both points are either inside or outside the sheet. If this case occurs, another edge is selected and a new pair of points is found. If the wrap numbers of the points differ by more than one, then the points are on opposite sides of the boundaries of more than one sheet. This is usually the result of multi-traced edges occurring where one sheet has coincident edges with another.

The trimming procedure takes a single integer argument which is used to determine which sheets are to be extracted from the polygon. Where the intersection of two polygons is desired, all those sheets which separate area of wrap number 2 from area of wrap number 1 are part of the intersection. Other sheets may be removed. Selection the sheets to be removed requires the trimming procedure to find two points for each sheet whose wrap numbers differ by

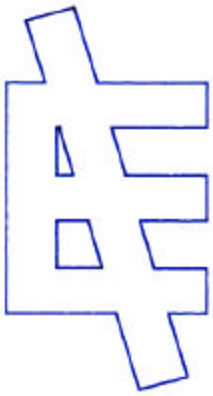
exactly one. Points whose wrap numbers differ by more than one give no indication of which of the two or more sheets involved should be removed. This algorithm breaks down when two sheets exactly overlay each other, having all edges coincident. However, it does permit the operations of intersection and union to be performed. For intersection, all those sheets which do not separate areas of wrap number 2 from area of wrap number 1 are removed. For union, all those not separating areas of wrap number 1 from wrap number 0 are removed. Operations with more than two operands, such as the intersection of three polygons, can be constructed by looking for sheets bounding areas of higher wrap number.

To make possible arbitrary logical operations on polygons, a negation operator must be available. This function must be able to reverse the sense of all sheets of a polygon such that all substance becomes holes and all holes become substance. Where sheets enclose other sheets, this description is not complete. However, the operation of negation is merely the reversing of the order of all the edges and vertices of each sheet and reversing the direction of each edge. Using negation the operations of "exclusive or" and subtraction may be constructed as well as any other logical operations.

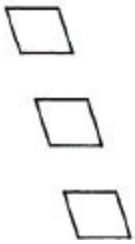
Figure 3.2 shows examples of common logical operations performed on the polygons of Figure 3.1. Intersection, union, exclusive or and subtraction are shown. Unfortunately the direction of the edges of the sheets is not visible in such drawings.



TWO POLYGONS

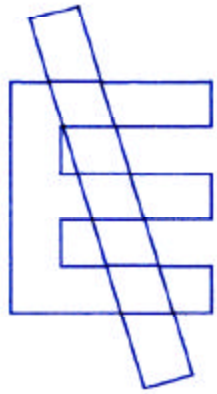


UNION



INTERSECTION

SUBTRACTION
(A-B)

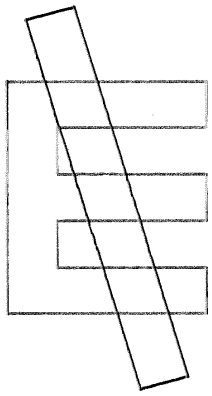


EXCLUSIVE OR
(PARALLEOGRAMS ARE HOLES)

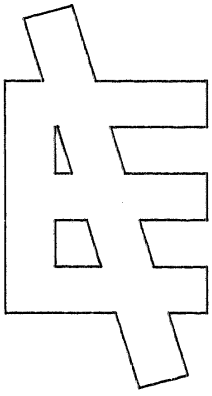
SUBTRACTION
(B-A)

FIGURE 3.2

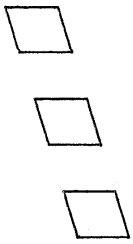
LOGICAL OPERATIONS USING POLYGONS



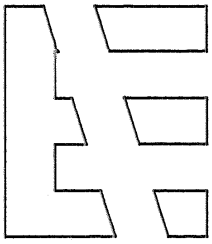
TWO POLYGONS



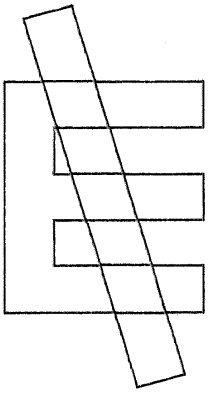
UNION



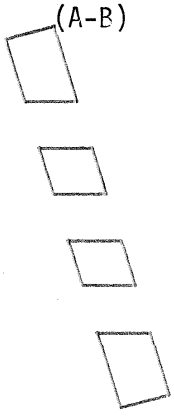
INTERSECTION



SUBTRACTION
(A-B)



EXCLUSIVE OR
(PARALLEOGRAMS ARE HOLES)



SUBTRACTION
(B-A)

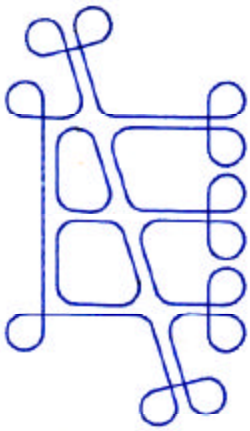
FIGURE 3.2

LOGICAL OPERATIONS USING POLYGONS

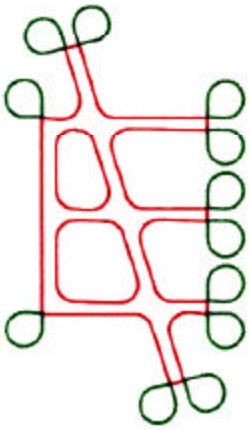
3.3 Inflation and Deflation

The recognition of devices and other applications such as design rule checking require that polygons be shrinkable and expandable. To be consistent, a polygon which is expanded by a particular size and subsequently shrunk by the same amount should reconstruct the original polygon. This and the desire to avoid the detection of false design rule errors brought about by corner to corner proximity of polygons lead one to the use of circular arcs as edges in general polygons. The algorithms and conventions used in the manipulation of circular arcs is given in Sutherland [10].

It is evident that the expansion of a sheet with convex corners will result in a new sheet having arcs about the corners of the original sheet. The arcs are the locus of points a fixed distance from the vertices of the sheet. Inside corners or concavities in a sheet cause "curlicues" to be generated upon expansion. These portions of the new sheet cause some areas of the sheet to become doubly wrapped. The shrinking of polygons has the opposite effect at sharp corners. Convex corners beget curlicues outside the substance of the sheet wrapping areas negatively, concave corners shrink to circular arcs (fillets). Figures 3.3 and 3.4 illustrate the result of inflation and deflation. Since the curlicues represent doubly wrapped areas or holes in free space and are redundant, they must be removed before logical operations can be performed on them. If left as is, the trimming procedure will not be able to distinguish intersecting areas from redundancies due to inflation or deflation.



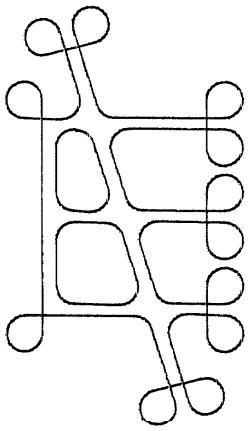
DEFLATE



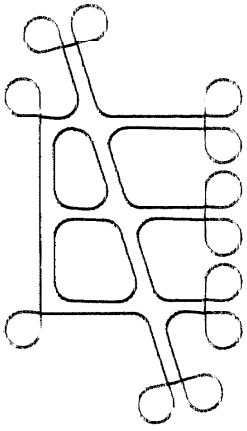
RETRACE

TRIM

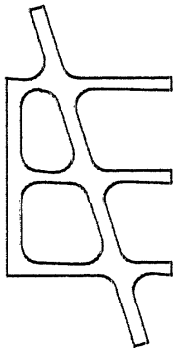
FIGURE 3.3
DEFLATION OF POLYGONS



DEFLATE



RETRACE



TRIM

FIGURE 3.3
DEFLATION OF POLYGONS

If the inflated sheets are selfintersected and retraced, the trimming procedure can be used to extract the redundant sheets before applying logical operations. The successive frames in Figures 3.3 and 3.4 illustrate this process. Positive sheets that are smaller than the amount of shrinkage applied will be shrunk into holes in free space and eliminated. Sheets representing holes will similarly disappear if expanded by more than their size.

The property of these polygons which causes them to reverse if shrunk sufficiently is very useful in the separation of different silicon structures from each other. It also provides a method by which the width and length of transistors may be determined, this is discussed fully in Chapter V.

3.4 Selection of Sheets

Two other functions, which do not fall in the category of logical operations, are necessary to topology extraction. These functions conditionally extract sheets from a polygon where a given sheet either encloses some particular point or where the sheet overlaps or encloses some part of another polygon. The trimming procedure described above falls into this category.

To enable a program to find those sheets of a given layer that make contact with other layers through cuts in the oxide, a procedure must be available which can test a sheet for overlap with another sheet (a contact cut) and conditionally remove the sheet intact from its polygon for

use in other operations. Such a procedure may be able to accomplish this function without the expense of fully intersecting one polygon with another. Processing can be reduced if only the fact of intersection need be known and not the precise geometry of it. This is obviously the case if sheets are to be selected on the basis of whether or not they enclose a particular point. The wrap number of the point with respect to the individual sheets must be computed but no more. Those sheets which yield a wrap number about the point which is non-zero are selected. The detection of overlap between two polygons for the extraction of certain sheets or for design rule checking can be simplified over general intersection. This problem is discussed by Sutherland and Sproull [10]. If one is interested in detecting the overlap of polygons which must always enclose another set of polygons, then the problem reduces to that of computing wrap numbers. This simplification can be applied to the selection of sheets which overlap a polygon of contact cuts provided that it is assured that design rules have been met.

3.5 Other Operations

Several other operations are necessary in the manipulation of polygons and the computation of wiring statistics.

The computation of the quantities outlined in Chapter II obviously require that the area of a polygon be known. The area of a polygon is a signed quantity where sheets representing holes contribute negatively to the total area of a polygon.

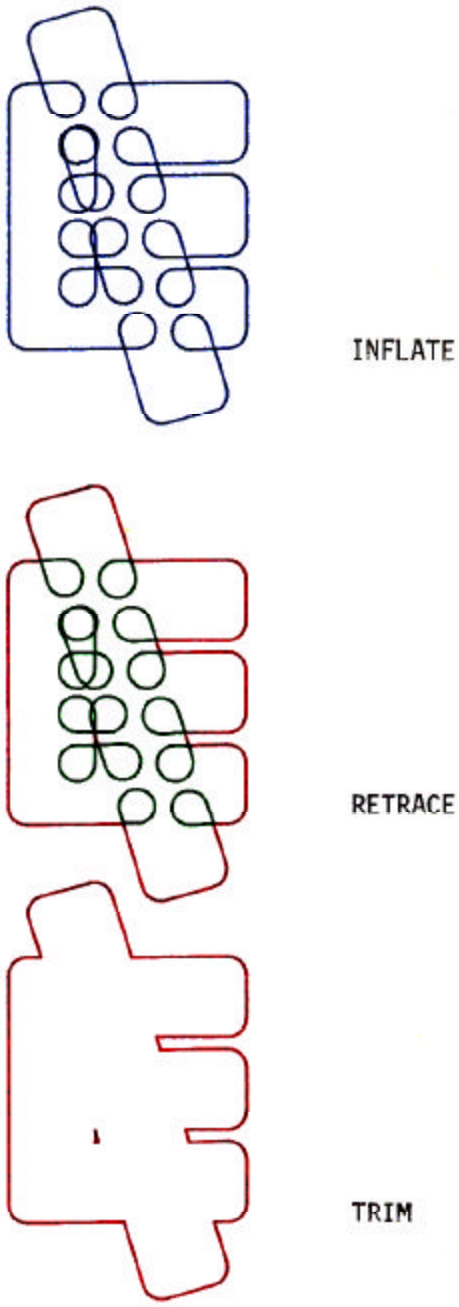


FIGURE 3.4

INFLATION OF POLYGONS

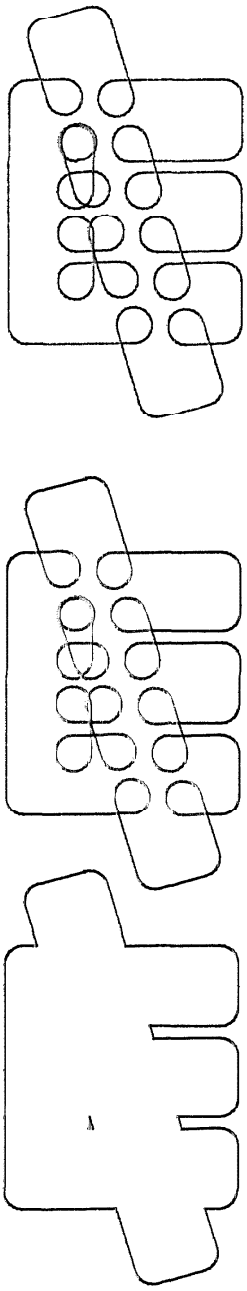


FIGURE 3.4

INFLATION OF POLYGONS

It will be found the computation of the width and length of transistors in Chapter V also requires a procedure for obtaining the perimeter of a polygon. For multi-sheet polygons, the perimeter is the sum of the length of all the edges in all the sheets. If this method is used, redundant sheets as described above will also contribute to the perimeter if present.

Two routines are necessary to improve the overall efficiency of the program. Successive operations upon polygons can cause unnecessary vertices and edges to be generated and made part of the sheets of the polygon. An example of this is two sequential collinear edges separated by a vertex. One edge would do. A procedure is needed to clean the sheets and eliminate such excesses. Also, the program will require less memory and computation time if the operations used on polygons are permitted to cannibalize the data structures of the operands in generating their result. If this is to be the case, a simple procedure is required which returns an independent copy of a polygon to be used to preserve the operands when necessary.

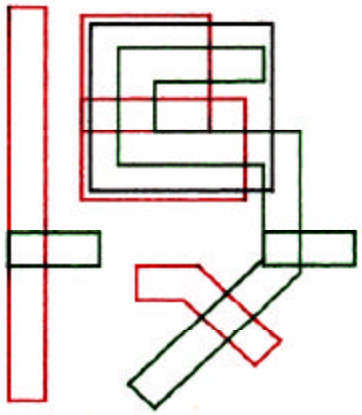
The symbol call of CIF allows the user to specify that the symbol being called be translated, rotated or mirrored when instantiating it in the symbol in which the call command appears. This operation is known as a transformation and can be thought of the multiplication of a 3 by 3 matrix with every point of the object to be transformed. A full discussion of these operations can be found in Newman and Sproull [12]. Any of the transformations possible in CIF (and a few more besides) can be represented by a 3 by 3 matrix where three of the elements are fixed values which

never change (requiring only six values to be stored). Transformations of transformations ad infinitum can be represented by one matrix which is the multiplication of all the individual matrices (in the proper order). When polygons, representing symbols, are to be instantiated in another symbol, a transforming procedure must be available to apply the transformation to the polygon.

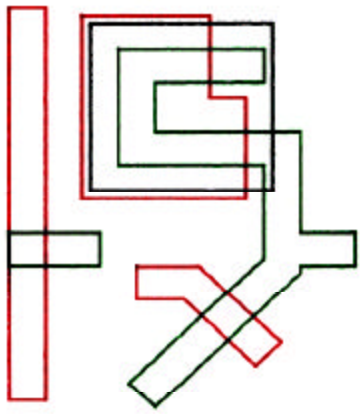
Even after CIF objects are converted to CIF polygons and the CIF polygons are used to produce the more general sheets of polygons described in this chapter, the resulting objects are not suitable for wiring analysis. The wiring analysis program treats each layer of each symbol as one entity, rather than a collection of simple shapes. This is necessary since if the abutting shapes making up a wire connection on the metal layer are individually shrunk, they will draw apart and cease to make a connection. The shapes of each layer must be coalesced into the complete shapes one would see on the finished masks before they can be manipulated. Figure 3.5 shows both the CIF specification of a symbol and the form it must take in the polygon package. Each layer must be freed of redundant overlaps and collapsed into the largest sheets possible before logical operations can be applied. This means only that all the shapes of each layer must be "ORed" with each other and the doubly wrapped areas trimmed off. In effect, each layer is then represented by one polygon containing the minimum set of sheets and edges required to outline the areas specified in the CIF file for that layer.

All the procedures and capabilities discussed in this chapter are listed in Appendix I and have been implemented in the original polygon package (Sutherland [61]) or in the

first version of the wiring analysis program. In addition, procedures for printing and plotting the polygons are available along with debugging aids. The attributes and capabilities of this polygon package, though general, execute at speeds and with memory requirements which make it unsuitable for large problems. It is hoped that the efficiency and algorithms involved can be improved to increase the useability of these functions.

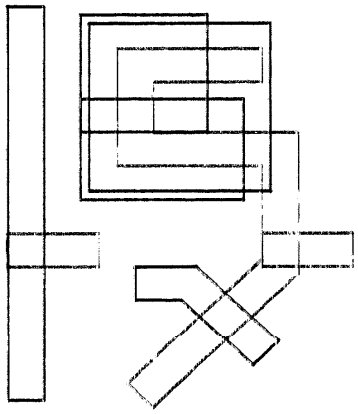


UNMERGED

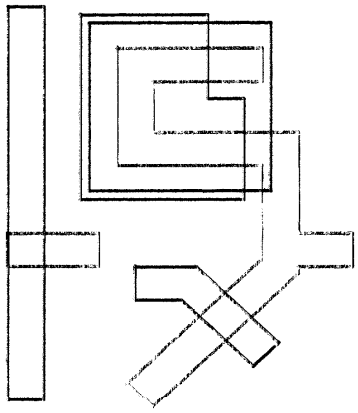


MERGED

FIGURE 3.5
MERGING OF SYMBOL WIRES and
BOXES INTO POLYGONS



UNMERGED



MERGED

FIGURE 3.5
MERGING OF SYMBOL WIRES and
BOXES INTO POLYGONS

Chapter IV

Heuristic Computation of Wiring Statistics

4.1 Recognition of Structures by Heuristic Methods

It is possible, though not precise, to extract wiring analysis data from geometry without the need to examine the topology of the circuit. The usages of NMOS for digital purposes have enough in common to permit a small number of rules to be used in the recognition of devices and wires. The application of the rules does not guarantee the results to be always correct. The obvious benefit of such a method, in spite of its inability to correctly interpret all situations, is its lower cost to use. The simplifications used must be accurate enough to prevent errors from causing significant deviations in the results.

The goal of this method is to compute the required wiring statistics using only polygon manipulations. Picture recognition of this type depends on experience with the structures to be detected in the generation of algorithms to do the job automatically. Unless care is taken, this can result in the misinterpretation of structures not normally encountered. It is important that the simplifications make as few assumptions about the design style as possible.

The succeeding sections of this chapter describe polygon operations for the extraction of devices from interconnection and global wire from local. The detection of devices is much less difficult than the distinction made between global and local wire. This is because devices must have some physical basis while the distinction between global and local wire is conceptual and has no necessary manifestation in the physical geometry of the chip. However, it does have meaning to the designer and the need for wiring data requires that methods be available to make this distinction. Both operations, as described here, mistake some types of structures for other in the analysis of the chip. The more frequent of these exceptional situations are explained in the last section.

4.2 Extraction of Devices

NMOS devices involve the interaction of polysilicon with diffusion. The operation of intersection used between these two layers results a polygon containing all the potential devices. To eliminate the degenerate devices, such as, butting and buried contacts, the polygon may be shrunk by one half the minimum size of a transistor (1 lambda) plus some tolerance. All degenerate transistors will then become sheets with negative areas and can be extracted from the polygon. The polygon can then be expanded by the amount it was shrunk. It is then necessary to remove from the poly and diffusion layers, those portions which make up the devices. To accomplish this the polygon representing the devices is expanded by one half the minimum spacing and subtracted from both the poly and diffusion layers. The sheets remaining in the poly,

diffusion and metal layers then represent interconnection structures only. The segment of Simula code below shows how this is done using the objects and attributes of Appendix I. The syntax and semantics of Simula are defined in Birtwistle, et al [13] and Birtwistle [14].

```
! Temporaries to hold copies of polygons;
REF(polygon) temppoly,tempdiff,tempdev;

! Make copies of poly and diffusion layers;
temppoly:-symbola.layer(poly).copy;
tempdiff:-symbola.layer(diffusion).copy;

! AND poly and diffusion layers,    ;
! put into devices  -- Then shrink them;
symbol.devices:-symbol.layer(poly)
    .intersect(symbol.layer(diffusion));
symbol.devices.shrink(1.001);

! Test each sheet of the devices for negative area, ;
! if negative extract it;
FOR s:-symbola.devices.nextsheet(s) WHILE s=/=NONE DO
BEGIN
    IF s.area<0 THEN symbola.devices.extractsheet(s);
END;

! Inflate the remaining devices and make a copy of them;
symbola.devices.inflate(1.001+2.0);
tempdev:-symbola.devices.copy;

! Subtract them from the poly layer, make a new copy;
symbola.layer(poly):-temppoly.subtract(tempdev);
tempdev:-symbola.devices.copy;
! Subtract devices from the diffusion layer;
symbola.layer(diffusion):-tempdiff.subtract(tempdev);
```

The inflation of the devices two lambda beyond their original size is done to include the polysilicon that must overlap the diffusion. This is a simplification, since in the direction of current flow, there is no overlapping

poly. The intent is to account for the area occupied by depletion loads and the overlapping of the pulldown by polysilicon as well as possible. In the direction of current flow, transistors may be packed very close, with only the two lambda polysilicon spacing required between them. In some complex gates this situation may occur frequently. This type of design is usually, though not always, limited to enhancement pulldowns. In the direction orthogonal to current flow, transistors may be packed with three lambda between them (minimum diffusion spacing). However, this implies that the gates of such transistors must all be in common resulting in a circuit that would have a very rare usage. In practice, depletion loads would almost never be constructed in this way because to do so shorts inverter outputs together. Enhancement pulldowns built in this manner could conceivably be used to control several inverters, but the diffusion from which the inverters would be built would not typically be next to each other.

These observations lead one to believe that MOS transistors may be placed two lambda apart if their source/drain connections are in series (a common configuration). However, the usual placement in the other direction requires six lambda spacing. This is brought about by the two lambda overlap required by each of the two transistors and the two lambda spacing required between their respective gates. To account precisely for this variable spacing would require that the channel direction of each transistor be known and the transistor "stretched" in one direction to account for the inequality. If the transistor is non-orthogonal, or of an unusual shape this can be very difficult. However, experience with NMOS technology shows

that these exception cases are rare and the vast majority of the transistors can be characterized with simple polygon manipulations. If the devices (the intersection of poly and diffusion minus the degenerate cases) are inflated two lambda beyond their normal size, depletion loads are reasonably accounted for. Enhancement devices may be placed closely in complex gate circuits but are small devices in comparison to the depletion loads and if they are over inflated a small amount of area may be accounted for twice by ajoining transistors. This can be removed by the following line of code.

```
symbols.devices.selfintersect.retrace.trim(1);
```

This will remove any doubly wrapped areas of the devices. There are always inaccuracies encountered when experience or frequency data is used to simplify processes. Here it is possible to regard more area as that of devices, due to the overinflation of enhancement devices, than may actually be the case. The experience on which the recognition is based would suggest that this will be a small amount.

If additional processing is available, much of this inaccuracy can be eliminated with additional operations. Since usage of the pullup loads and the pulldowns are different and to a large extent are visibly different types of devices, they can be recognized without regard for the topology and treated differently. Enhancement transistors are rarely used as pullups and depletion transistors are rarely used as pulldowns. A large amount of the inaccuracy make be removed if the implantation layer is used to distinguish depletion from enhancement devices. The above code for separating the devices from the interconnect would

be modified to become the following:

```
PROCEDURE Getdevices(symbola); REF(symbol)symbola;
BEGIN

    ! Temporaries to hold copies of polygons;
    REF(polygon) temppoly,tempdiff,tempddev,tempndev;

    ! Make copies of poly and diffusion layers;
    temppoly:-symbola.layer(poly).copy;
    tempdiff:-symbola.layer(diffusion).copy;

    ! AND poly and diffusion layers, ;
    ! put into devices  -- Then shrink them;
    symbol.ndevices:-symbol.layer(poly)
        .intersect(symbol.layer(diffusion));
    symbol.ndevices.shrink(1.001);

    ! Test each sheet of the devices for negative area;
    FOR s:-symbola.ndevices.nextsheet(s)
    WHILE s=/=NONE DO
    BEGIN
        IF s.area<0 THEN
            symbola.ndevices.extractsheet(s);
    END;

    ! Reinflate the remaining devices;
    symbola.ndevices.inflate(1.001);

    ! Separate the depletion transistors from ;
    ! enhancement;
    symbola.ddevices:-symbola.ndevices
        .extractmatching(symbola.layer(implant));

    ! Inflate each type of device appropriately ;
    ! and make copies;
    tempddev:-symbola.ddevices.inflate(3)
        .selfintersect.retrace.copy;
    tempndev:-symbola.ndevices.inflate(1)
        .selfintersect.retrace.copy;

    ! Subtract them from the poly layer, ;
    ! make a new copy;
    symbola.layer(poly):-temppoly.subtract(tempddev)
                         .subtract(tempndev);
    tempddev:-symbola.ddevices.copy;
    tempndev:-symbola.ndevices.copy;
```

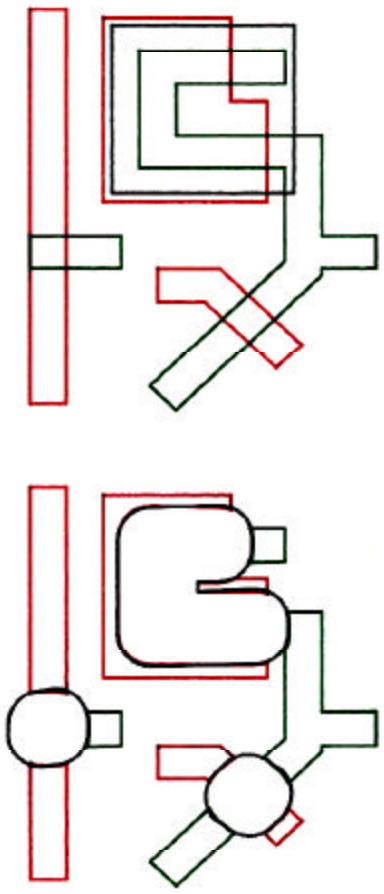
```
        ! Subtract devices from the diffusion layer;
        symbola.layer(diffusion):=tempdiff
                .subtract(tempddev).subtract(tempndev);

    END of Getdevices;
```

Figure 4.1 shows the operation of this algorithm on a small cell. The interconnection structures are left in their original colors and the devices are outlined in black. There are, of course, situations in which this algorithm misinterprets the geometry. It is believed that these would be too few to cause their combined effect to be seen the in the statistics accumulated for a design.

The algorithm also extracts capacitors properly and expands them by the correct three lambda. Capacitors do not share poly or diffusion with other devices unless they are part of the same net. This means that capacitors will usually have to be built with two lambda overlapping poly and one-half the minimum poly spacing built around them as are most depletion load devices.

Passive resistors built using long runs of poly or diffusion are not detected. However, this type of device is rare even in analog circuits and is not important to the wiring analysis of digital systems.

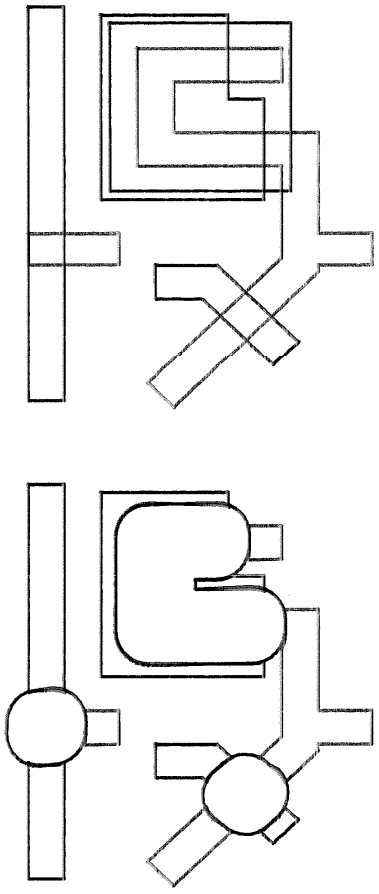
Other devices can be made if a second layer of polysilicon is available. Usually these are capacitors and can be detected by merely intersecting the two poly masks. The programs and algorithms of this thesis are currently limited to NMOS processes with single layer polysilicon and no buried contact mask. The extensions required to analyze designs that use these two features can be easily made but



DEVICE AREAS
IN BLACK

FIGURE 4.1

IDENTIFICATION OF DEVICES



DEVICE AREAS
IN BLACK

FIGURE 4.1

IDENTIFICATION OF DEVICES

require additional computations.

4.3 Estimation of Symbol Area

To make the statistics computed for a symbol meaningful, the area occupied by the entire symbol must be determined. As with many structures of a chip, it is difficult to define the boundaries of a symbol without knowledge of exactly what surrounds it. The heuristic method being discussed in this chapter attempts to estimate the area of the symbol using only the contents of the symbol as data. As with devices, there are situations in which such methods become unreliable. Experience with NMOS designs permits boundaries to be defined that are most probably correct.

The boundary of a symbol must extend at least one-half the minimum line spacing beyond the structures of the symbol. The structures of the symbol may occupy several layers, be disjoint and have holes or other areas that may be utilized by surrounding symbols. What is needed is a bounding hull that outlines the area of the symbol which external symbols must violate to affect the symbol. Those areas through which there is no room for a minimum size wire or which are completely enclosed must be inside the hull.

These boundaries must be computed separately for each interconnection layer (metal, poly and diffusion). They are constructed from the primitive structures of the symbol as well as the hulls of its subsymbols. The procedure to accomplish this is straightforward. For each layer, all the hulls of subsymbols (for that layer) and all the locally defined areas are OR'ed together. Then all the holes of the

resulting polygon are removed, since these represent enclosed areas. The remaining sheets are then inflated by one minimum spacing and one-half the minimum width for that layer, closing off any slots into which a minimum size wire cannot enter. The doubly wrapped areas are removed and the polygon is shrunk by one-half the minimum spacing and one-half the minimum width such that the remaining sheets are one-half a minimum spacing larger than they were originally but now have many of the slots in their "skyline" filled in. Redundancies are again removed and the resulting polygon is a reasonable outline of the symbol on that layer.

The bounding hulls of each layer of the symbol become attributes of the symbol and are used in further operations. The area of each bounding hull is also recorded and becomes an important statistic with which to compare the results of other computations. Some hulls may have zero area if a given symbol does not utilize a particular layer. It should be noted, that the units of area used here, are square lambda and are not process dependent. Thus they may be compared with the areas of designs employing older or newer design rules providing the basic underlying processes are not significantly different. Below is a Simula program which finds the bounding hull of each of the three layers as discussed above.

```
PROCEDURE Hulls(symbola); REF(symbol)symbola;
BEGIN

    REF(polygon)hull;
    INTEGER i,layer;

    ! Loop to get each layer of the symbol;
    FOR layer:=1 STEP 1 UNTIL 3 DO BEGIN
```

```
            !make a copy of the layer of the symbol;
            hull:-symbola.layer(layer).copy;

            ! Loop to get all the hulls of subsymbols ;
            ! for this layer;
            FOR i:=1 STEP 1 UNTIL symbola.subsymbols.length
            DO BEGIN
                REF(instance)inst;

                ! Get the hull of the subsymbol ;
                ! and combine a copy with hull;
                ! Call Hulls recursively ;
                ! if subhull is not present. ;
                inst:-symbola.subsymbols(i) QUA instance;
                IF inst.sym.hull(layer)==NONE THEN
                    Hulls(inst.sym);
                hull:-hull.combine(inst.sym.hull(layer).copy
                                   .Xform(inst.tr));
            END;

            ! Through out the doubly wrapped areas ;
            ! and all the holes;
            hull.selfintersect.retrace.trim(1);
            hull.extractholes;

            ! Inflate it, trim it, then deflate it ;
            ! and trim it;
            hull.inflate(spacing(layer)+width(layer)/2);
            hull.selfintersect.retrace.trim(1);
            hull.deflate((spacing(layer)+width(layer))/2);
            hull.selfintersect.retrace.trim(1);

            ! Put the hull into the symbol attributes;
            symbola.hull(layer):-hull;
        END;

    END of Hulls;
```

The execution of this procedure on a simple symbol is illustrated in Figure 4.2.

The sources of error in this algorithm are due to its ignorance of external connections of the symbol to other symbols. Where wires of one symbol abutt or overlap to

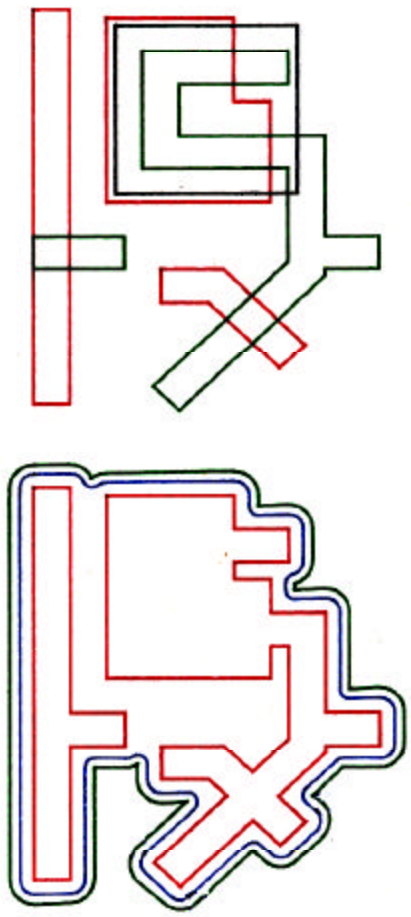


FIGURE 4.2

FINDING THE BOUNDARIES
OF A SYMBOL

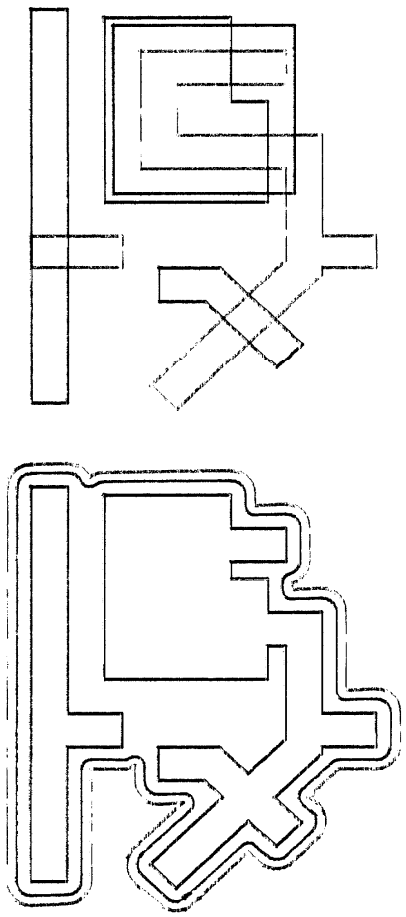


FIGURE 4.2

FINDING THE BOUNDARIES
OF A SYMBOL

make a connection, this algorithm will count the contact area to be in the hulls of both symbols making them appear slightly larger than they really are. In higher level symbols which combine two such symbols, this is corrected since it results in doubly wrapped area and is reduced to singly wrapped area when the hulls are ORed together. The error is then non-accumulating and merely results in some optimism in size of the symbol. The wasted area of a symbol, if it is not enclosed, will not be counted as part of the symbol area. This will overshadow any optimism due to external connections. Such wasted area will be accounted for in the next higher level of the symbol hierarchy.

4.4 Extraction of Global Wire from Local Wire

The extraction of global wire from local wire without the use of connection information is difficult not only because polygon operations are not easily able to recognize it but also because the definition of global wire is not a clear one. The definition requires some recognition of those structures that make external connections. This implies that internal connections be visible and distinguishable from external connections. Internal connections can be identified as places where wires intersect devices and subsymbols. External connections are difficult to detect and to be precise would require that the topology be known.

The detection of possible external connections requires some assumptions be made about the usual form in which they are made. Generally, a wire which is to connect to an external symbol will end at the edge of the symbol of which

it is a part. This need not always be true, since a wire from another symbol may intrude far into the symbol before making a connection. Also, a wire may be laid over a symbol from an external source and drop a contact to another layer through which a connection is made. These design techniques do not usually occur in regularized or structured design and hopefully do not represent a significant fraction of the external connections of symbols. If such techniques are important to the wiring analysis, then the topology extraction of Chapter V must be used to find all the external connections of a symbol.

One heuristic method of separating global wire from local wire using only polygon manipulations, uses the symbol boundary computed earlier. A new boundary is constructed using the union of the hulls of the three interconnection layers after they have each been shrunk by one-half their minimum spacing. The result is a polygon which encloses all of the symbol and is slightly larger than it needs to be to enclose the largest objects. From this a very thin annulus or donut is made. A copy of the polygon is made and shrunk by a small amount such that it is too small to enclose all the objects of the symbol. It is then reversed (made into a hole) and added as a sheet of the original polygon. The result is a thin ring about the symbol which the objects on the perimeter overlap. Figure 4.3 shows this annulus in red for the simple symbol. Below is a procedure which will generate the annulus of a symbol. The annulus is computed prior to the extraction of devices to be sure that the devices are included.

```
PROCEDURE Annulus(symbola);  REF(symbol)symbola;
BEGIN
```

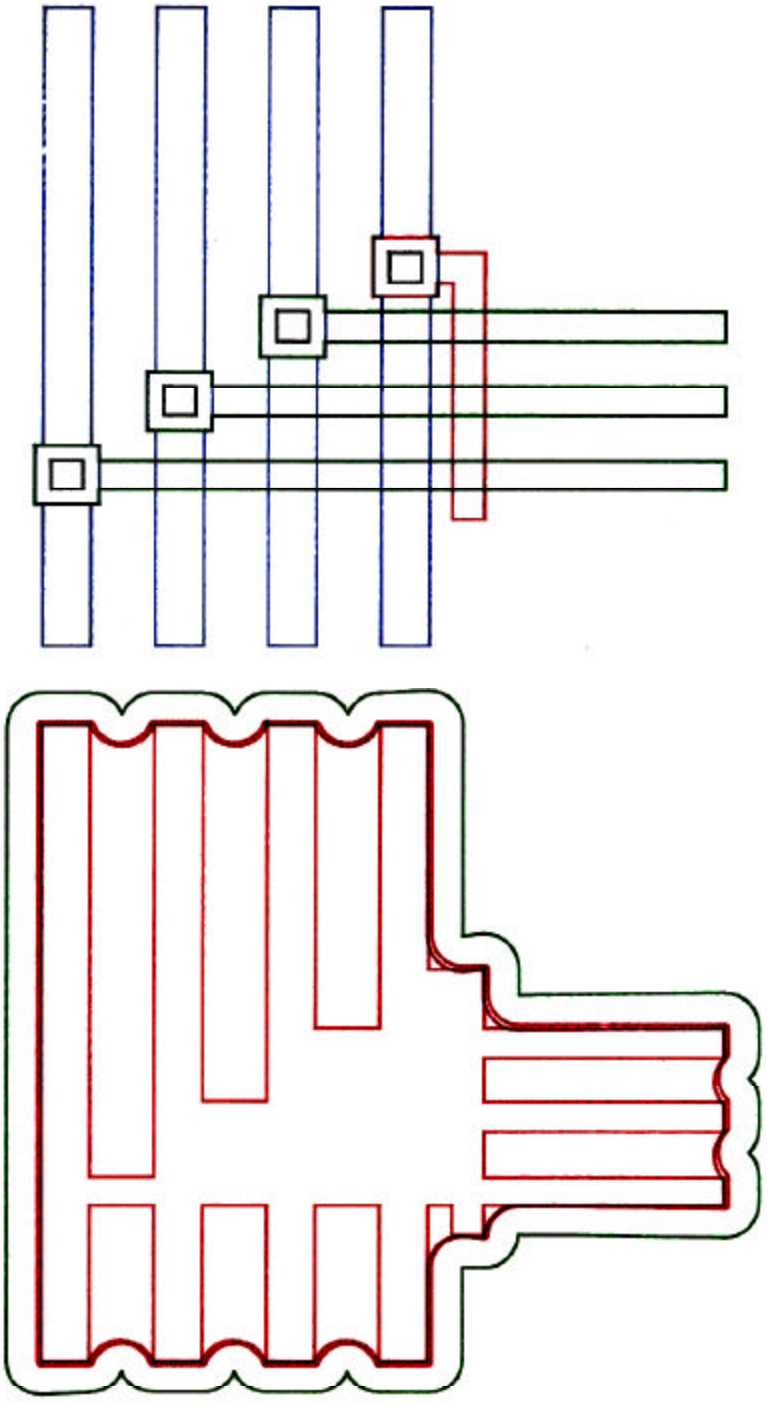


FIGURE 4.3

THE ANNULUS OF A SYMBOL

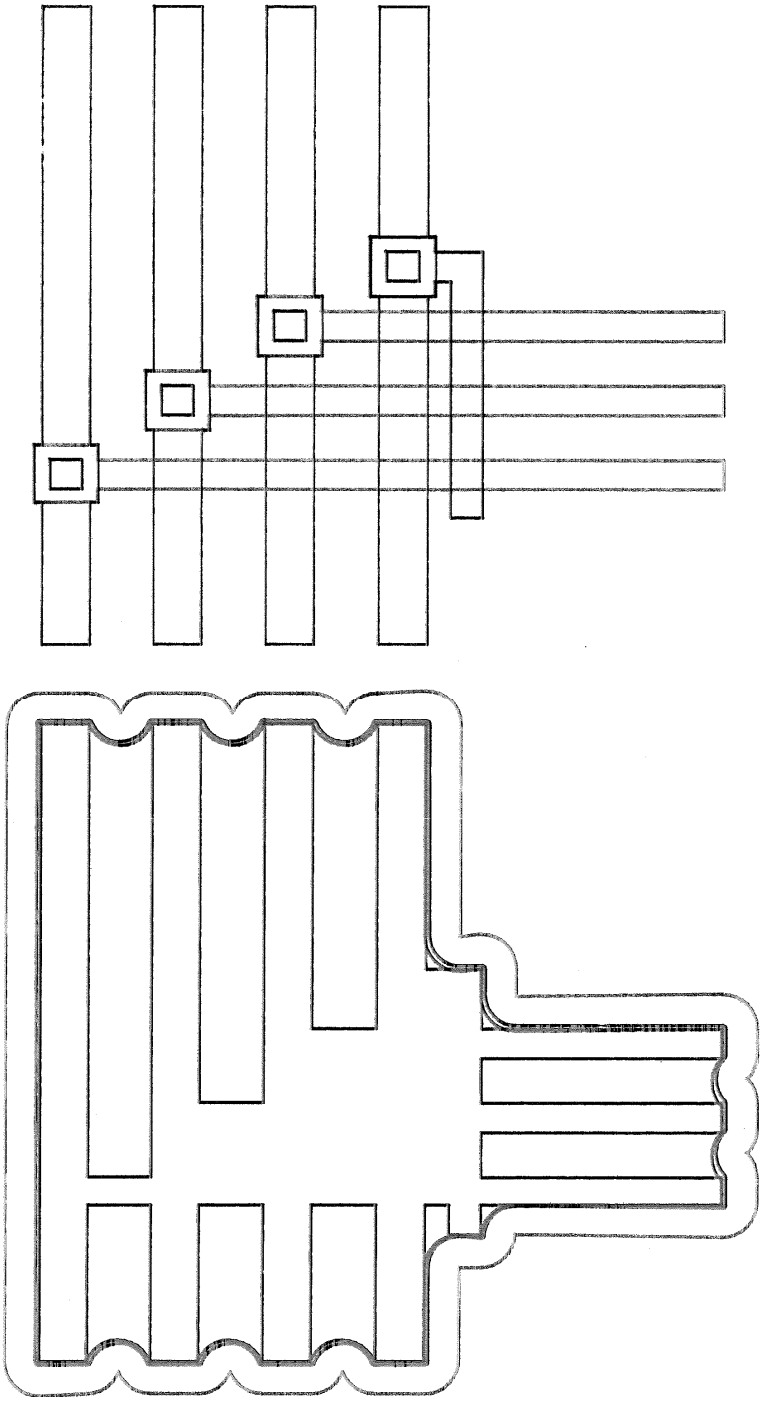


FIGURE 4.3

THE ANNULUS OF A SYMBOL

```
        REF(polygon)rim,hole;
        INTEGER i;

        ! Union all three layers after copying ;
        ! and deflating each;
        FOR i:=1 STEP 1 UNTIL 3 DO
            rim:-rim.combine(symbola.hull(i)
                .copy.deflate(spacing(i)-.001));
        rim.selfintersect.retrace.trim(1);

        ! Make a hole out of a deflated ;
        ! version of the rim;
        hole:-rim.copy.deflate(.001).reverse;

        ! Put the hole in the rim and assign it ;
        ! to the symbol;
        symbola.annulus:-rim.combine(hole);

    END of Annulus;
```

If this annulus is then intersected with the objects making up the symbol (excluding the subsymbols), possible external connection areas are found. As explained above, these may not be all the external connection areas but they should be most of them. Further, not all of the areas found by the intersection will be external connections, some may merely be the sides of structures adjoining the edge of the symbol.

Many of the intersections will "bend" around the corners of the ends of wires. To remove the portions of the intersections that correspond to the sides of the wire rather than the end, the thin intersections must be broken where they turn corners. To do this, a copy of the intersection may be inflated (by a size greater than the width of the intersection) generating "curlicues" at all inside or concave corners. The curlicues may be extracted and subsequently subtracted from the original intersection polygon, causing intersections with bends to be broken at

each turn. This leaves the polygon with a set of sheets that may or may not be external connection points.

To distinguish some of the external connections from most of the false intersections, the area of the intersection can be an indicator. Connections are usually made with minimum size wires, the exceptions are usually power distribution. If a maximum width is set for the width of an external connection, the various contacts with the annulus may be culled with respect to their area. The area of a sheet which is an external connection must be less than or equal to the chosen width multiplied by the width of the annulus (.002 as computed above). Thus those sheets of greater area may be removed and the remaining sheets have a high probability of actually being external connections. Also, it is known that the external connections must have a minimum area as well. This size can be used to cull out some of the small sheets that may have originated on the sides of wires and were broken from other parts of the intersection with the annulus.

Few external connections fall outside of this recognition procedure. Figure 4.4 indicates the external contacts seen by the described method. A Simula implementation is listed below. These contacts are found after the devices have been extracted such that devices are not mistaken for contacts (although some could have external contacts).

```
PROCEDURE Xcontacts(symbola); REF(symbol)symbola;
BEGIN
    INTEGER i;

    ! Loop to do each interconnection layer;
    FOR i:=1 STEP 1 UNTIL 3 DO BEGIN
```

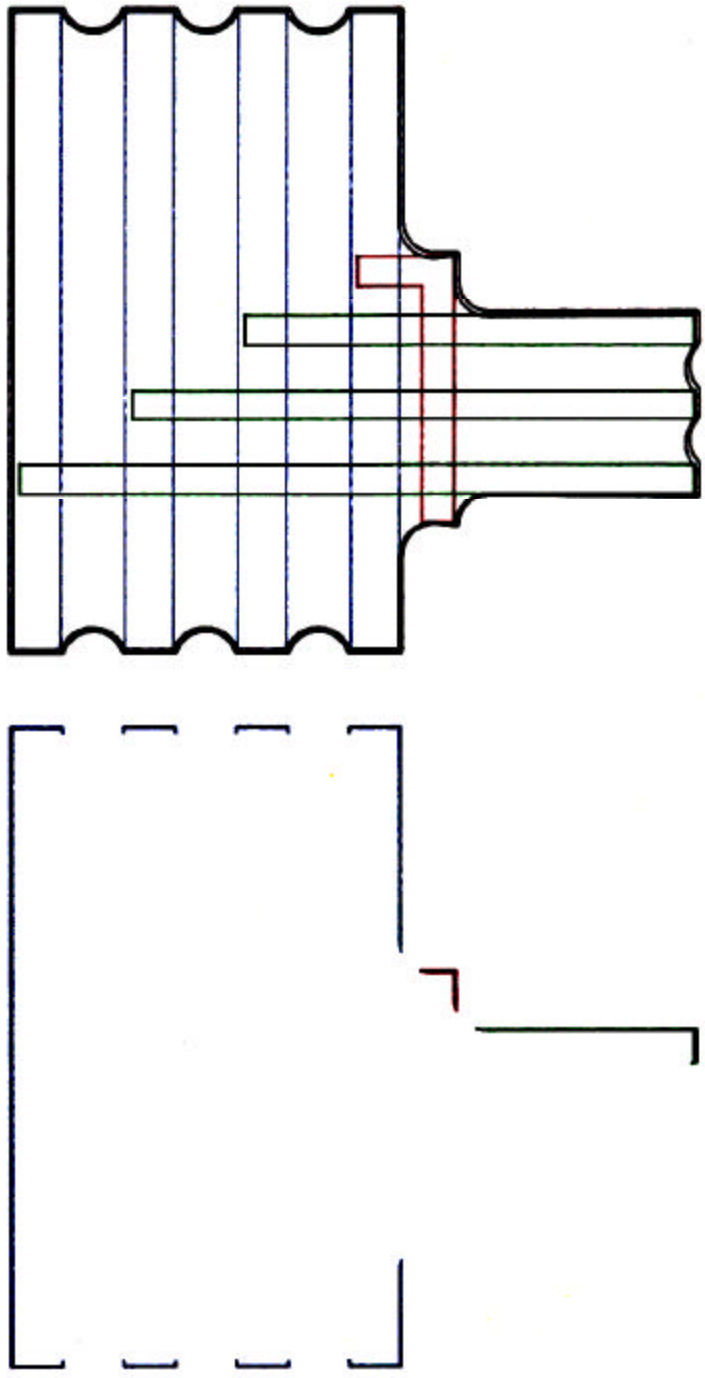


FIGURE 4.4

FINDING POSSIBLE EXTERNAL CONTACTS
OF A SYMBOL

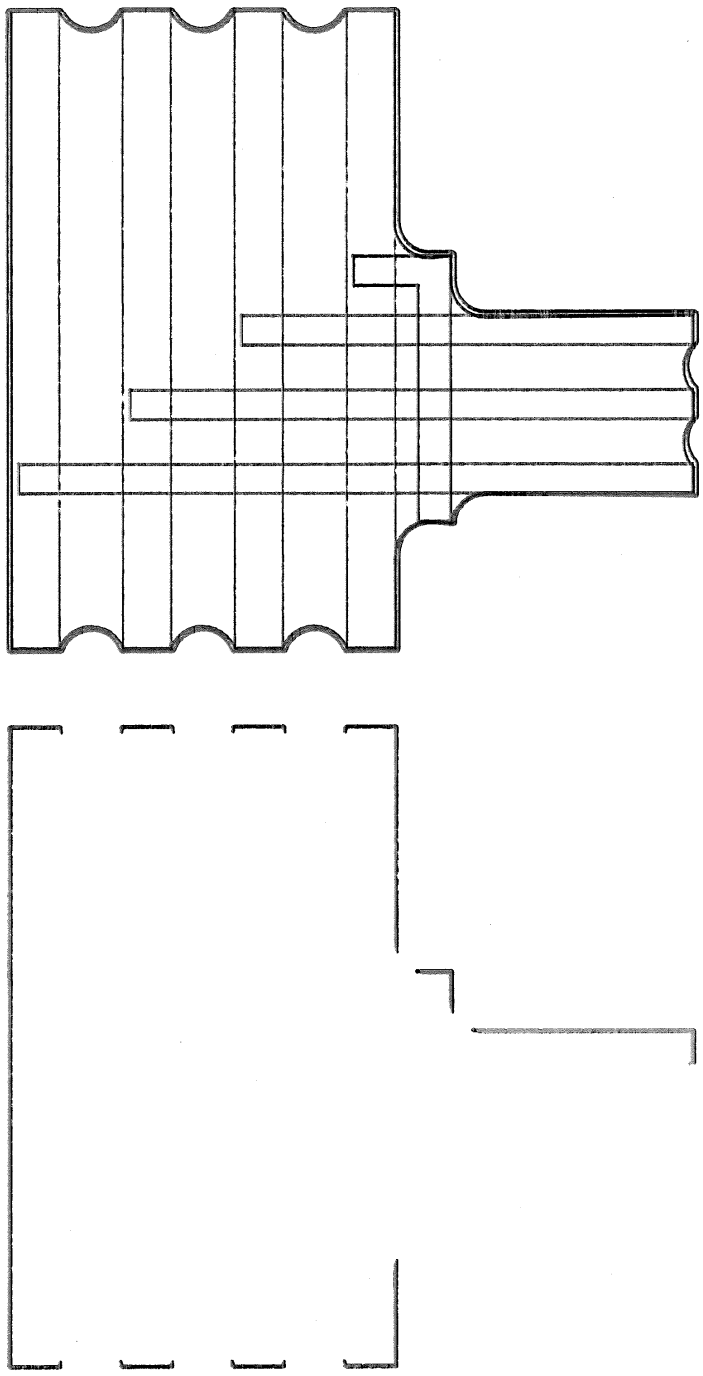


FIGURE 4.4

FINDING POSSIBLE EXTERNAL CONTACTS
OF A SYMBOL

```
            REF(sheet)s;

            ! Intersect a copy of the annulus with ;
            ! a copy of each layer;
            symbola.xcontact(i):-symbola.annulus.copy
                        .intersect(symbola.layer(i).copy);

            ! Inflate a copy of the intersection by ;
            ! more than the width of the intersections, ;
            ! then save the curlicues;
            curlicues:-symbola.xcontact(i).copy
                              .inflate(.003).trim(1);

            ! Subtract the curlicues from the original ;
            ! uninflated intersections, breaking open ;
            ! the intersections at bends;
            symbola.xcontact(i):-symbola.xcontact(i)
                              .subtract(curlicues);

            ! Compute the area of each sheet and remove if;
            ! it is outside the max and min then ;
            ! extract the sheet;
            FOR s:-symbola.xcontact(i).nextsheet(s)
            WHILE s=/=NONE DO BEGIN
                REAL a;
                a:=s.area;
                IF a)maxxcon OR a(minxcon THEN
                    symbola.xcontact(i).extractsheet(s);
            END of sheet loop;

        END of layer loop;

    END of Xcontacts;
```

Given the external contacts of the symbol, the global wire may be separated from the local wire. Devices have been previously removed from the wire layers thus breaking polysilicon wires which connect the gates of several transistors in sequence. The contact cuts of a symbol may usually be regarded as an internal connection. Infrequently, global wire may change layers using a contact cut. If the contact cuts are inflated to a size slightly larger than the minimum size wires, they may be subtracted from the interconnection cutting it up into segments. If a

segment contains an external contact, then it may be considered global, the rest are local wires. The parts subtracted added back to the local wire arbitrarily. The two types of wire, global and local may then be expanded by one-half their minimum spacing and their area computed for the statistics of the symbol. Figure 4.5 shows the wire separated from the symbol by these means. Below is the listing of a Simula procedure for implementing this separation.

```
PROCEDURE SeparateWires(symbola); REF(symbol)symbola;
BEGIN
    REF(polygon)bigcuts,tempw,tempc;

    ! Inflate the Contact cuts and put in a temporary;
    bigcuts:-symbola.layer(cuts).copy.inflate(1.001);

    ! For each layer, separate the global from local;
    FOR i:=1 STEP 1 UNTIL 3 DO BEGIN

        ! Subtract the Cuts from the wires;
        tempw:-symbola.layer(i).copy
                        .subtract(bigcuts.copy);
        ! Save the pieces removed from the wire;
        tempc:-symbola.layer(i).copy
                        .intersect(bigcuts.copy);
        ! Extract the wire that touches ;
        ! external contacts - this is global;
        symbola.globalwire(i):-tempw.extractmatching(
                                symbola.xcontacts(i));
        ! Put the subtracted pieces and ;
        ! whats left together - this is local;
        symbola.localwire(i):-tempw.union(tempc);

    END of layer loop;

END of SeparateWires;
```

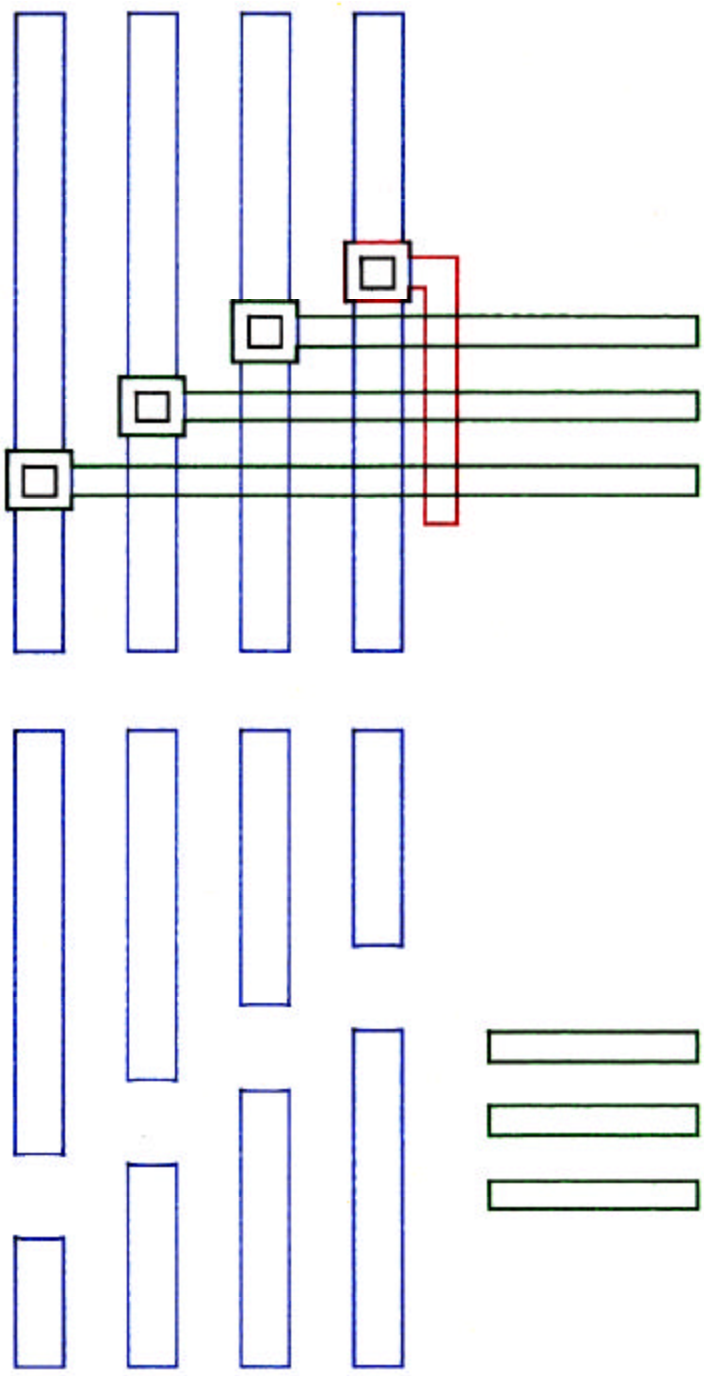


FIGURE 4.5

SEPARATION OF GLOBAL AND LOCAL WIRE

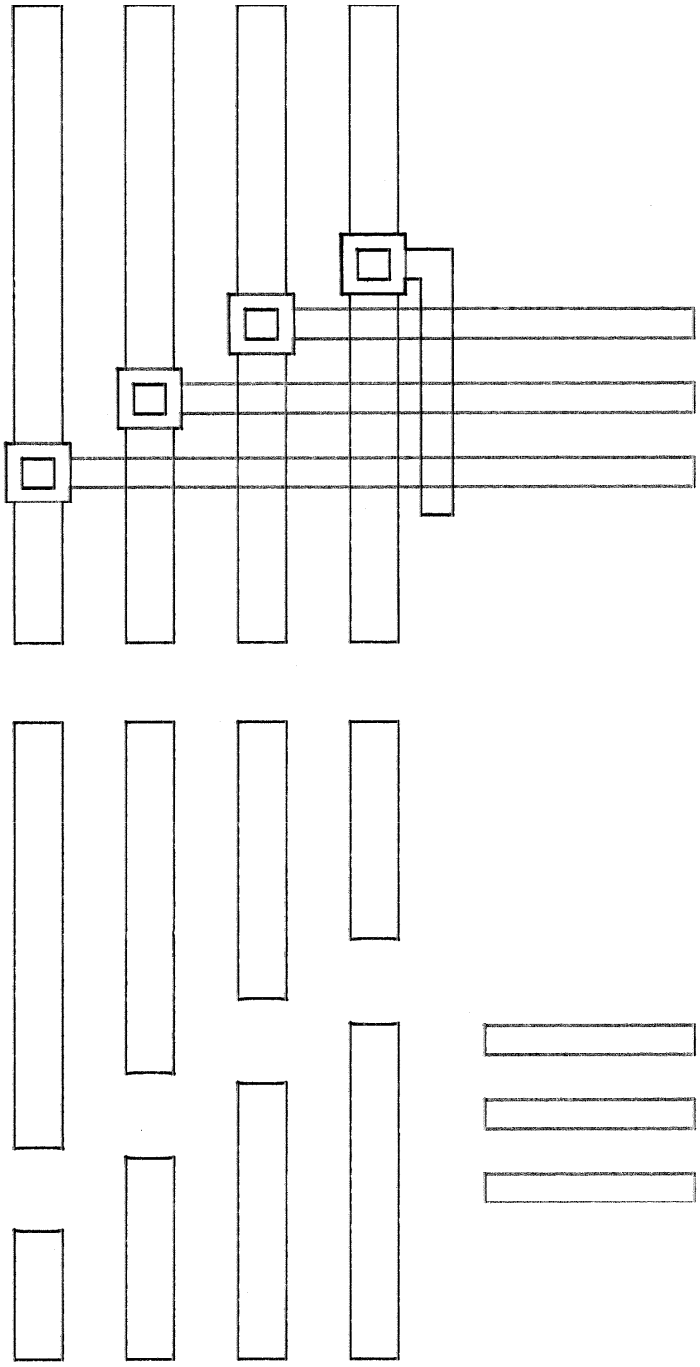


FIGURE 4.5

SEPARATION OF GLOBAL AND LOCAL WIRE

This heuristic method of separating global wire from local can misinterpret the geometry in many instances. Experience with the program and data taken from a number of designs will determine if these instances are rare enough to ignore. While the program has not run on a large body of designs, the algorithms presented have been applied by hand to a wide variety of symbols. The conclusion reached is that errors introduced into the wiring statistics by misinterpretation are less than 5 percent.

Chapter V

Extraction of Circuit Topology

5.1 Devices and Device Connections

As in the previous chapter, devices must be recognized and, in addition, their sizes and types must be determined. The source, drain and gate must be identified so that connections to them can be found. Of course, depletion mode transistors can be distinguished from enhancement mode devices by intersecting them with the ion implantation layer. However, capacitors must be recognized and differentiated from transistors. If the topology extracted from the mask data is to be used in the verification of the design, all the device and interconnection recognition must be precise. In practice, some assumptions have to be made to permit the programming of the problem with reasonable efficiency, however, exceptions to any rules or restrictions made by these procedures must be highly improbable.

The recognition of devices, the determination of their sizes and their connection points, can be carried out as part of one operation. First of all, as with the heuristic method, the polysilicon and diffusion layers are intersected giving sheets which may be transistors. Again, these are shrunk, culled by area and expanded back to original size to eliminate false devices caused by butting

contacts. The devices are then subtracted from the diffusion layer.

The devices are then expanded a small amount (.001 lambda). These sheets are individually intersected with first the poly layer and then the diffusion layer. The two thin sheets resulting from the intersection with the diffusion are the source and drain contacts. One can be arbitrarily assigned as the drain and the other the source. The distinction is not important since NMOS transistors are bilaterally symmetric electrically. The sheets resulting from intersection with the poly layer are both gate contacts.

Capacitors may be easily separated from transistors since the capacitors will have both their drain and source connected to the same circuit node. This usually means that there will be only one sheet in the intersection of the inflated device with the diffusion. It may also be that the connection of the drain to the source is made with structures that are not butting the transistor. If so, the capacitor will be recognized when the nets are collected.

The size of the transistor is expressed in two numbers, its width and length (sometimes called W/L or Z/L). The length of the device is the dimension (in lambda) between the drain and source, the direction of current flow. The width is the dimension orthogonal to the length. Transistors may be arranged arbitrarily in space and may be complex serpentine structures in which the direction of current flow changes with position in the device. The contacts that were found above, can be used to determine the width and length of a transistor. The length and width

computations are not required for capacitors, since their values are a function of their area and not their aspect ratio.

The area and width of the sheets representing the contacts are very small as intended in their generation. The two drain and source contacts outline the effective width of the device. The width is obtained from them by dividing their combined perimeters by 4. Similarly, the length of a transistor is one fourth the combined perimeter of the poly gate contacts. Figure 5.1 illustrates the contacts of several transistors and a capacitor. The contact sheets are so narrow that they appear as a single line but they are actually doubly traced by lines that are very close. Gate connection points are shown in red and the source/drain terminals are in green.

This technique works properly for the vast majority of devices designed using NMOS. There is one case in which it gives false sizes for a device. This occurs where devices are constructed using poly or diffusion structures whose width varies substantially in the channel region of the transistor. Such devices are virtually unknown and their effective width and length is not well defined except by detailed simulation.

The procedure below performs the operation described. Devices are located, separated from each other and their contacts and sizes are found.

```
PROCEDURE ProcessDevices(symbola); REF(symbol)symbola;
BEGIN
    REF(polygon)tempdev,temppoly;
```

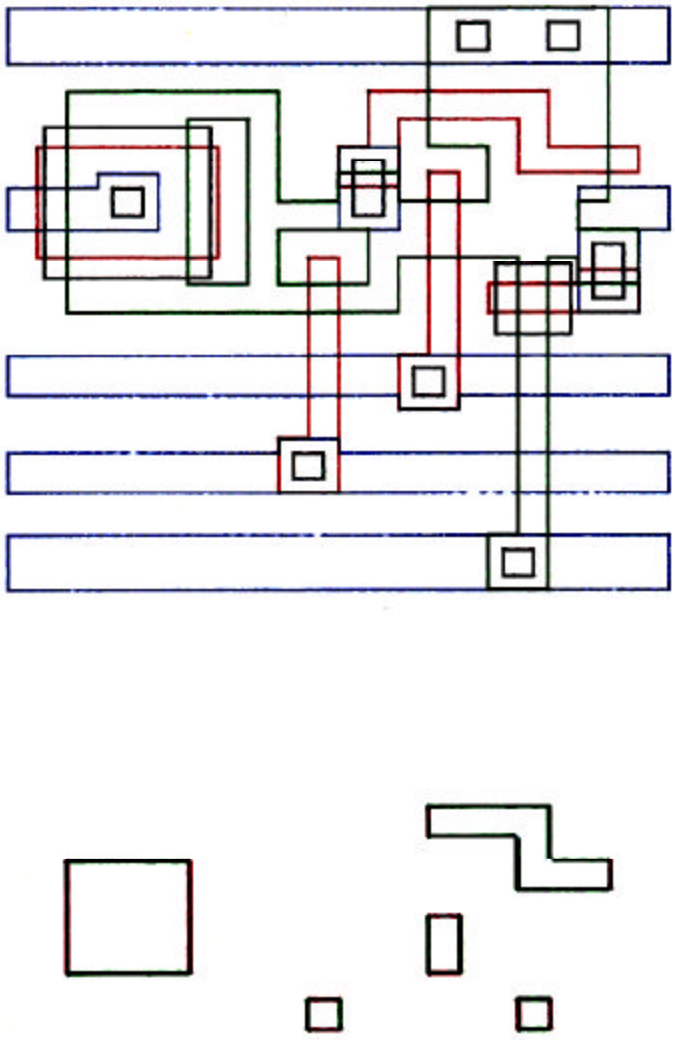


FIGURE 5.1

FINDING THE SIZES AND CONNECTIONS
TO DEVICES

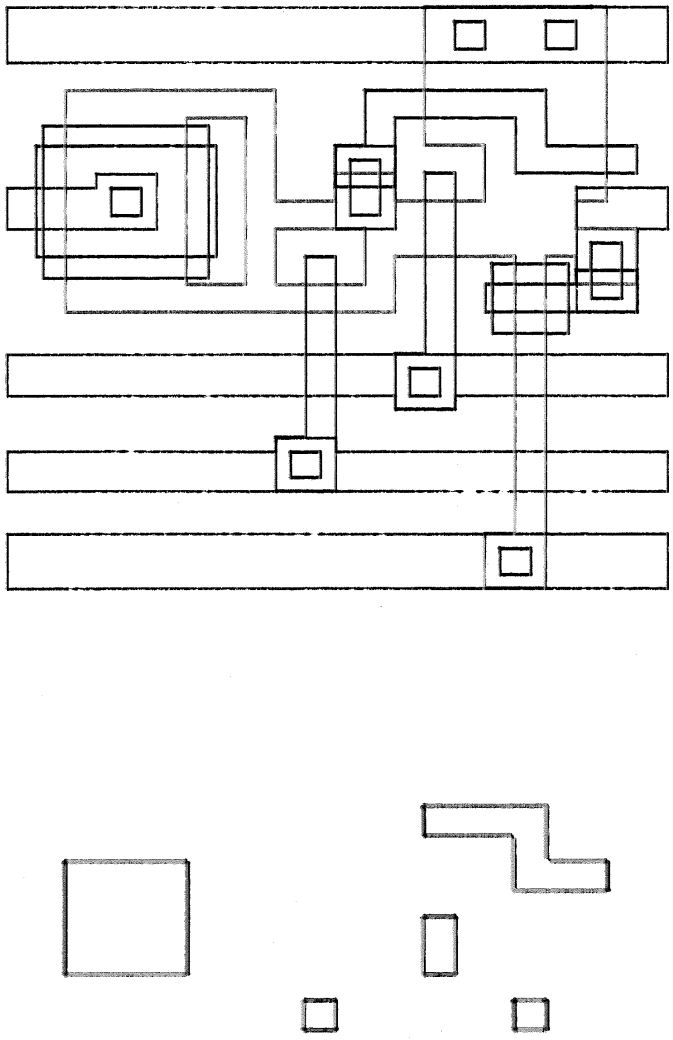


FIGURE 5.1

FINDING THE SIZES AND CONNECTIONS
TO DEVICES

```
REF(sheet)s;

! Intersect the poly and diffusion layers;
tempdev:-symbola.layer(poly).copy
        .intersect(symbola.layer(diffusion).copy);

! Remove the Butting contact overlaps;
tempdev.deflate(.501);
FOR s:-tempdev.nextsheet(s) WHILE s=/=NONE DO BEGIN
    IF s.area<0 THEN tempdev.extractsheet(s);
END;
tempdev.inflate(.501);

! Subtract the devices from the diffusion layers;
temppoly:-symbola.layer(poly).copy
                .subtract(tempdev.copy);
symbola.layer(diffusion):-symbol.layer(diffusion)
                .subtract(tempdev.copy);

! Process each individual sheet of the devices;
FOR s:-tempdev.nextsheet(s) WHILE s=/=NONE DO BEGIN
    REF(polygon)dev;

    ! Inflate the device slightly;
    dev:-s.copy.inflate(.001);

    ! Intersect it with poly and diffusion;
    gate:-temppoly.copy.intersect(dev.copy);
    drainsource:-symbola.layer(diffusion).copy
                .intersect(dev.copy);

    ! Test for a Capacitor ;
    IF drainsource.numsheets<2 THEN BEGIN
        REAL side;

        ! Store that capacitor in the symbol ;
        ! (stored as a transistor);
        side:=sqrt(dev.area);
        symbola.trans.append(
                NEW transistor(side,side,gate,
                        drainsource,drainsource));
    END
    ELSE BEGIN
        REAL width,length;
        REF(sheet)drain,source;

        ! Compute the width and length ;
        ! of the Transistor;
        width:=drainsource.perimeter/4;
        length:=gate.perimeter/4;
```

```
                ! Store the Transistor in the Symbol, ;
                ! separate drain and source;
                drain:-drainsource.nextsheet(NONE);
                source:-drainsource.nextsheet(drain);
                symbola.trans.append(NEW transistor(
                        width,length,gate,drain,source));
            END;
        END of Sheet loop;
    END of ProcessDevices;
```

5.2 Collecting Symbol Nets

The collection of symbol nets is the grouping of polygons which are electrically connected. The original entry of the mask data into the polygon package assures us that each sheet of a given layer represents either a complete connected area of the symbol or a hole in one. The contact cut layer and the gate contacts of transistors must be used to determine which sheets are connected to which other sheets.

For the purposes of this operation, it is necessary that polygons and sheets have a layer attribute. Nets may be composed of sheets from any of three layers and when grouped together by net, rather than layer, sheets must retain their layer information. Nets are vectors of sheets and can lengthen or shorten to accomodate a variable number of sheets. Nets also contain a vector of pointers to device contacts (and later, external contacts). Each symbol contains a vector of nets.

First, each sheet (and any holes in it) of the poly layer is initially defined to be a net. Then the gate, drain and source of each transistor is examined and a pointer to it is placed in the net to which it connects. The metal is then used to coalesce the poly and diffusion nets into

fewer larger nets. Metal areas which do not contain contact cuts become new nets. The code shown below, accomplishes this but has a computation time proportional to N squared where N is the number of devices and interconnection structures. Symbols do not generally include a large amount of locally defined devices and wires. This should prevent the computation of the topology from becoming excessive. If this occurs, more sophisticated techniques can be used to reduce the number of operations in the testing of the contacts and wires for overlap. The object of this chapter is to illustrate a structure recognition procedure for extracting topology, the implementation details have been simplified to emphasize the recognition aspects of the problem. The final result of the manipulations is the grouping of all the interconnection structures into the nets which they form and the listing of devices connections made by each net.

To finish the netlisting of the devices, each transistor drain and source (the gates have been previously dealt with) are tested against the diffusion sheets of each net. If there is a match, a pointer to that contact is made in the list of that net. A similar procedure is applied to the two contacts of capacitors.

To finish the compilation of the topology of the internal structures of a symbol, the netlists must include pointers to the external contacts of any subsymbols that might be present. The next section discusses how these contacts of a symbol are found. To include those of subsymbols, each such contact is tested against the appropriate sheets of each net in the same manner as the contacts of devices above. In this manner the topology of a symbol and all its

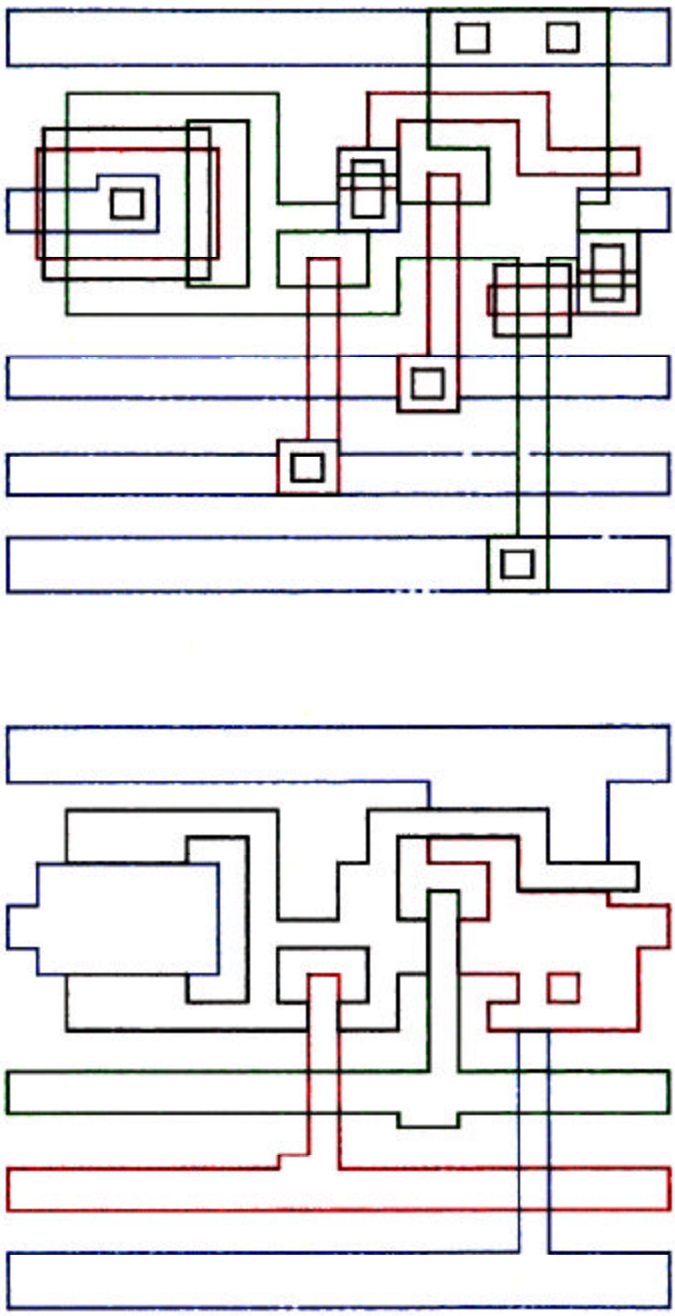


FIGURE 5.2

GROUPING INTERCONNECTION STRUCTURES
INTO NETS

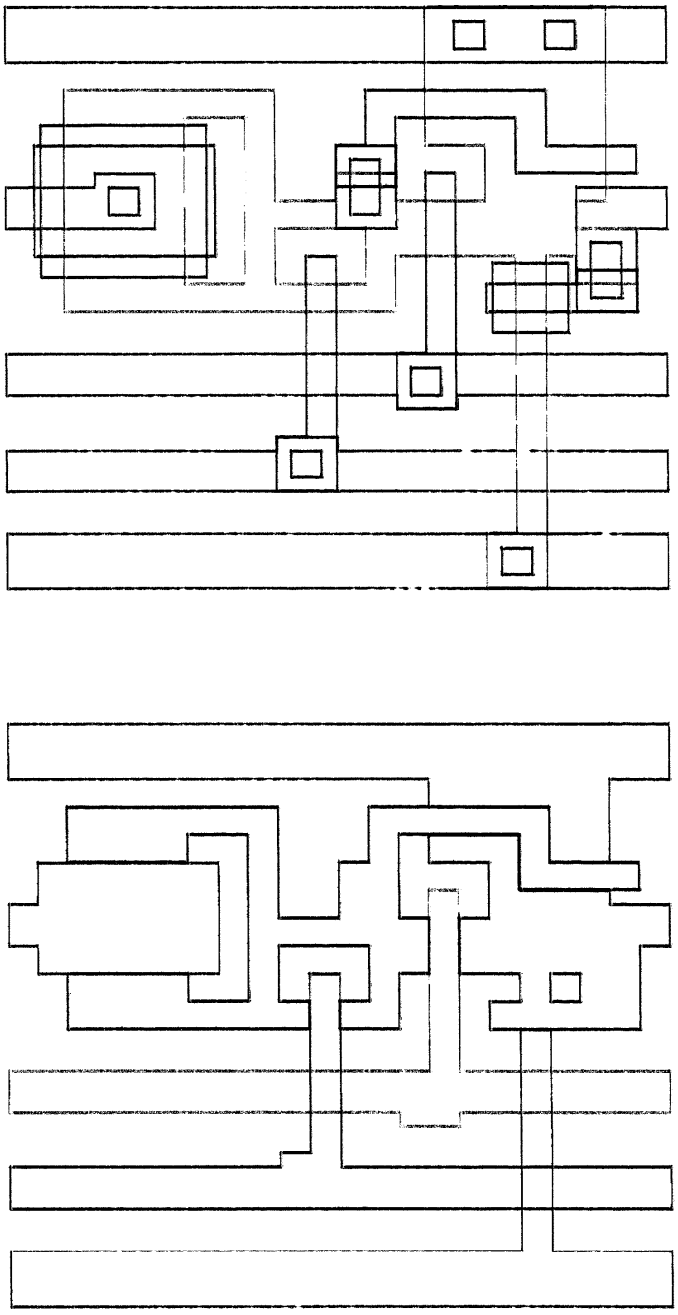


FIGURE 5.2

GROUPING INTERCONNECTION STRUCTURES
INTO NETS

subsymbols may be determined. The external contacts of the symbol at hand with nets of symbols further up in the hierarchy have yet to be determined. Figure 5.2 illustrates the grouping of the interconnection structures of a simple symbol. Each set of interconnection structures is shown coalesced in its own color. Actually the various pieces of the net retain their individual identity within the program. The procedure below, performs the grouping discussed above.

```
PROCEDURE Nets(symbola); REF(symbol)symbola;
BEGIN
    REF(vector)netvector;
    REF(sheet)s;
    INTEGER i;
    REF(polygon)poly,diff,metal,cuts,polyholes,
                diffholes,metalholes;

    ! Define objects to be used in the net vectors;
    thing CLASS Net;
    BEGIN
        REF(polygon)poly,diff,metal;
        REF(vector)netsheets,neticons,netxcons;
        BOOLEAN empty;

        ! This procedure combines two nets into one;
        PROCEDURE Coalesce(netb); REF(net)netb;
        BEGIN
            ! simple vector copy procedure;
            REF(vector) PROCEDURE vcopy(a,b);
            REF(vector)a,b;
            BEGIN
                INTEGER i;
                IF b.length\=0 THEN BEGIN
                    FOR i:=1 STEP 1 UNTIL b.length DO
                        a.append(b.val(i));
                END;
                vcopy:-a;
            END of vcopy;

            ! Copy in stuff from other net ;
            ! and mark it empty;
```

```
            neticons:-vcopy(neticons,netb.neticons);
            netxcons:-vcopy(netxcons,netb.netxcons);
            poly.combine(netb.poly);
            diff.combine(netb.diff);
            metal.combine(netb.metal);
            netb.neticons:-NONE;
            netb.netxcons:-NONE;
            netb.poly:-NONE;
            netb.diff:-NONE;
            netb.metal:-NONE;
            netb.empty:=TRUE
        END;

        ! Initialization code;
        neticons:-NEW vector;
        netxcons:-NEW vector;
        poly:-NEW polygon;
        diff:-NEW polygon;
        metal:-NEW polygon;
    END of Class Net;

    ! Define an object to represent contact areas;
    thing CLASS NetContact(p,obj);
    REF(polygon)p; REF(thing)obj;
    BEGIN INTEGER layer; END;

    ! initialize the net vector and make ;
    ! copies of the poly and diff layers;
    netvector:-NEW vector;
    diff:-symbola.layer(diffusion).copy;
    poly:-symbola.layer(poly).copy;
    polyholes:-poly.extractholes;
    diffholes:-diff.extractholes;

    ! Make nets out of each sheet of the poly ;
    ! and diffusion layers. Holes in sheets are ;
    ! associated with the sheet they are a hole in;
    FOR s:-poly.nextsheet(s) WHILE s=/=NONE DO BEGIN
        REF(net)neta;
        net:-NEW net;
        neta.poly.combine(s)
            .combine(polyholes.extractmatching(s));
        netvector.append(neta QUA thing);
    END;
    FOR s:-diff.nextsheet(s) WHILE s=/=NONE DO BEGIN
        REF(net)neta;
        net:-NEW net;
        neta.diff.combine(s)
            .combine(diffholes.extractmatching(s));
        netvector.append(neta QUA thing);
    END;
```

```
! Loop on all the transistors (and capacitors);
! of the symbol.  Put their contacts into the ;
! proper nets, combine nets when necessary;
FOR i:=1 STEP 1 UNTIL symbola.trans.length
WHILE symbola.trans.length\=0 DO BEGIN
    REF(transistor)t;
    REF(net)netmatch;
    INTEGER j;
    t:-symbola.trans.val(i) QUA transistor;

    ! Loop through every net ;
    ! and test for connections;
    ! Do not scan further than the ;
    ! first net that matches;
    FOR j:=1 STEP 1 UNTIL netvector.length
    WHILE NOT match DO BEGIN
        REF(net)neta;
        neta:-netvector.val(i) QUA net;
        IF NOT neta.empty THEN BEGIN

            ! TEST the gate contact ;
            ! against the poly part;
            IF t.gate.overlaps(neta.poly) THEN BEGIN
                neta.neticons.append(
                    NEW netcontact(t.gate,t)
                    QUA thing);
            match:=true;
            END;

            ! Test the source contact ;
            ! against the diffusion;
            IF t.source.overlaps(neta.diff) THEN
            BEGIN
                neta.neticons.append(
                    NEW netcontact(t.source,t)
                    QUA thing);
            match:=true;
            END;

            ! If the transistor is a capacitor, ;
            ! the drain and source will point ;
            ! at the same sheet.  If they do not, ;
            ! test the drain against the diffusion;
            IF t.source=/=t.drain THEN BEGIN
                IF t.drain.overlaps(neta.diffusion)
                THEN BEGIN
                    neta.neticons.append(
                        NEW netcontact(t.drain,t)
                        QUA thing);
                match:=true;
                END;
        END;
```

```
    END of net loop;
END of transistor loop;


! Examine each sheet of the metal and give ;
! it to the net whose poly or diffusion it ;
! touches.  Coalesce the nets needed.  Metal ;
! with no contacts form new nets;
metal:-symbols.layer(metal).copy;
cuts:-symbols.layer(cuts).copy;
metalholes:-metal.extractholes;
FOR s:-metal.nextsheet(s) WHILE s=/=NONE DO BEGIN
    INTEGER i;
    REF(polygon)mycuts,p;
    REF(net)netmatch;
    p:-s.combine(metalholes.extractmatching(s));

    ! Get the contact cuts of this metal sheet and;
    ! loop through all the nets to find ones that ;
    ! match these cuts;
    mycuts:-cuts.extractmatching(p);
    IF netvector.length\=0 THEN BEGIN
        FOR i:=1 STEP 1 UNTIL netvector.length DO
        BEGIN
            REF(net)neta;
            neta:-netvector.val(i) QUA net;
            IF NOT neta.empty THEN BEGIN

                ! Test for poly or diffusion ;
                ! of each net which touches the ;
                ! cuts, if found, record the net ;
                ! (or coalesce it with another;
                IF neta.diff.overlaps(mycuts) OR
                neta.poly.overlaps(mycuts) THEN
                BEGIN
                    IF netmatch=NONE THEN
                        netmatch:-neta
                    ELSE netmatch.coalesce(neta);
                END;
            END;
        END of net loop;
    END;

    ! If No match is found with the metal sheet, ;
    ! make it a new net, otherwise, combine it ;
    ! with the net(s) that matched;
    IF netmatch=/=NONE THEN netmatch.metal
                                    .combine(p)
    ELSE BEGIN
        REF(net)neta;
        neta:-NEW net;
        neta.metal.combine(p);
```

```
            netvector.append(neta QUA thing);
        END;
    END of metal sheet loop;

    ! Now loop on subsymbols and their ;
    ! external contacts to see which nets ;
    ! of this symbol they connect to;
    FOR i:=1 STEP 1 UNTIL symbola.subsymbols.length
    WHILE symbola.subsymbols.length\=0 DO BEGIN
        REF(symbol)subsym;
        INTEGER j;
        subsym:-symbola.subsymbols.val(i) QUA symbol;
        FOR j:=1 STEP 1 UNTIL subsym.xcons.length
        WHILE subsym.xcons.length\=0 DO BEGIN
            REF(polygon)con;
            INTEGER k;
            BOOLEAN match;
            con:-subsym.xcons.val(j) QUA netcontact.p;

            ! Loop through all the nets until a match ;
            ! is found.  When found insert the contact ;
            ! into the netlist;
            FOR k:=1 STEP 1 UNTIL netvector.length
            WHILE NOT match DO BEGIN
                REF(net)neta;
                neta:-netvector.val(k) QUA net;
                IF NOT neta.empty THEN BEGIN

                    ! Test the contact against ;
                    ! the proper layer of the net ;
                    ! and record any match;
                    IF con.layer=poly THEN
                        match:=neta.poly
                                .overlaps(con)
                    ELSE IF con.layer=diffusion THEN
                        match:=neta.diffusion
                                .overlaps(con)
                    ELSE IF con.layer=metal THEN
                        match:=neta.metal
                                .overlaps(con);
                    IF match THEN neta.neticons
                                .append(con);

                END of IF NOT empty;
            END of net loop;
        END of subsymbol external contact loop;
    END of subsymbol loop;

    ! Transfer all non-empty nets into the ;
    ! symbol net vector;
    FOR i:=1 STEP 1 UNTIL netvector.length DO BEGIN
        REF(net)neta;
```

```
            neta:-netvector.val(i) QUA net;
            IF NOT neta.empty THEN
                symbola.nets.append(neta QUA thing);
        END of net loop;

    END of Nets;
```

5.3 External Connections

What remains is the determination of those points of a symbol which are used as connections with symbols other than its subsymbols. Up to this point, all the operations applied to symbols could be performed on the definitions of symbols from the bottom of the hierarchy to the top. The various ways that external connections may be made between symbols prevent the same technique from being used to find the external connections.

First of all, different instances of the same symbol may connect differently. Connections may be made in one instance that are not made in another. This immediately requires the checking of individual instances of the symbol.

Second, the external connections of subsymbols can be made by symbols at any place in the hierarchy, effectively shorting levels and branches in the design hierarchy. From the point of view of a single symbol, the external connections cannot be known merely by using the information supplied by the subsymbols and the symbol to whom this symbol is a child. It may be possible to impose restrictions that require the designer to interconnect symbols strictly within the hierarchy. Thus, a wire is required in a symbol definition wherever an external connection to a subsymbol is to be made available to a

super symbol. This would require a much larger amount of data than is necessary to describe the symbol. It would be more convenient for the designer explicitly identify all the points of a symbol to which external connection can be made. CIF contains no constructions intended for this purpose although additional layers could be defined upon which wires and boxes could identify external connection points.

The foregoing procedures capture all the internal topology of a symbol with the exception of subsymbol to subsymbol connections that do not involve structures local to the symbol. Once external connection points are known, these can be compared between subsymbols and such connections found. The following procedure supplements the "Nets" procedure with this capability. This procedure suffers from N squared complexity.

```
PROCEDURE MoreNets(symbola); REF(symbol)symbola;
BEGIN
    IF symbola.subsymbols.length\=0 THEN BEGIN
        INTEGER length,i,j;
        length:=symbola.subsymbols.length;

        FOR i:=1 STEP 1 UNTIL length DO BEGIN
            REF(vector)symx;
            symx:-symbola.subsymbols.val(i)
                    QUA symbol.xcons;

            FOR j:=1 STEP 1 UNTIL length DO BEGIN
                REF(vector)symy;
                IF i\=j THEN BEGIN
                    INTEGER k,l;
                    REF(net)neta;
                    REF(netcontact)c1,c2;
                    symy:-symbola.subsymbols.val(i)
                        QUA symbol.xcons;
                    IF symx.length>0 AND symy>0 THEN
```

```
                    BEGIN

                        FOR k:=1 STEP 1
                        UNTIL symx.length DO BEGIN
                            c1:-symx.val(k)
                                QUA netcontact;

                            FOR l:=1 STEP 1
                            UNTIL symy.length DO BEGIN
                                c2:-symy.val(l)
                                    QUA netcontact;

                                IF c1.layer=c2.layer
                                THEN BEGIN
                                    IF c1.p
                                    .overlaps(c2.p)
                                    THEN BEGIN
                                        neta:-NEW net;
                                        neta.neticons
                                            .append(c1
                                            QUA thing)
                                            .append(c2
                                            QUA thing);
                                        symbola.nets
                                            .append(neta
                                            QUA thing);
                                    END of overlap;
                                END of layer check;
                            END of 2nd contact loop;
                        END of 1st contact loop;
                    END of IF;
                END of IF;
            END of 2nd symbol loop;
        END of 1st symbol loop;
    END of IF;
END of MoreNets;
```

This procedure is obviously expensive if the total number of contacts of a symbol's subsymbols exceeds a small number. Since the external contacts of the subsymbols are the intersections of the subsymbols with other symbols, it appears that such a procedure is repeating the process of intersection that was original performed to find the external contacts. It would be preferable to extract the interconnectivity of symbols as part of the same operation

which detects the external contacts. The above procedure would then not be required.

The detection of external contacts requires that all the instances of symbol definitions be regarded as individual entities. Each layer of each symbol would have to be intersected with the respective layers of all its neighbors. Instances of subsymbols of a particular symbol would not be included in this operation on the symbol, but each would be compared in a separate operation of its own. To make this comparison in less than N squared time, the entire design may be subdivided horizontally and/or vertically using a Warnock algorithm [15] to partition the symbols with regard to their minimum bounding boxes. This permits those symbols which are close to the particular symbol to be quickly determined. The bounding boxes of these may be used to further cull the list of candidates for intersection. And finally, those left are intersected with the symbol instance to detect where it contacts other symbols. The connections made may be recorded in pinlists as this is done. This global pinlist may be made hierarchial as the internal topology of the individual symbol definitions is computed.

5.4 New Devices Created by Symbol Interactions

The operations presented thus far have assumed that the interactions of symbols did not result in the creation of new devices. Detection of this condition again requires all instances of symbol definitions to be treated separately, since the internal topology of a symbol will be changed by this action making it, in effect, a new and

different symbol.

Accommodating designs which use such techniques requires that both the poly and diffusion layers of all symbols be intersected with the diffusion and poly layers of all other symbols rather than just the poly to poly and diffusion to diffusion intersection required above. This is much more expensive to compute and requires further consideration in the storage requirements of the program. Not only must the global intersymbol pinlist be made, but separate copies of the symbol geometry for each instance of the symbol must be made. The reason for this is the need to provide for internal symbol geometry that may change due to a new device in one instance and may not change or change differently in another.

The conclusion this leads to is the need for a design restriction. If the topology is to be extracted automatically is is incumbent that the designer generate devices using only CIF wires, boxes and polygons in a given symbol definition. Interactions among subsymbols can not be permitted to form new devices. This restriction greatly reduces the complexity and cost of automated verification of this type without placing an excessive burden on the designer. It can also simplify some of the difficulties encountered in design rules checking.

Chapter VI

Results of Investigation

6.1 Program Performance

This first attempt to perform wiring analysis automatically requires the use of a polygon package and computer system which are unable to reliably operate on a problem of significant size. Some steps can be taken to improve their performance but these are not immediately forthcoming. The Simula run-time package which performs memory management and garbage collection, contains a hidden error which prevents large, long running jobs, such as wiring analysis, from running to completion. In spite of these difficulties, some statements can be made regarding experience with the program.

The size and long runtime of the program are themselves a result. Polygon manipulation is an important aspect of automated IC design aides, particularly as used in design rule checking. Given the existing polygon package and the machine on which it was used (DEC-20), the performance of the heuristic wiring analysis program was poor, preventing its use on all but small problems. The run time required to analyze a single symbol having no subsymbols with approximately 300 edges was in excess of one hour of CPU time. In reality, a successful run of this length was unusual in light of the Simula garbage collection errors.

Topology extraction could be performed on only the smallest symbols. Where it executed at all, it required approximately three times the computation time of the heuristic method. This can be considered a minimum ratio of performance between the two methods since topology extraction is of higher computational complexity and its run time will grow with the size of the design at a faster rate than that of the heuristic method.

The most expensive operation, by far, in the polygon package is the selfintersection procedure. In this procedure, all the edges which touch another edge must be found and new vertices and edges entered into the sheets. The usual methods of making comparisons between edges are plagued by execution times proportional to the number of edges squared. It is necessary that algorithms be selected for the purpose of reducing the relationship to N log N.

A second time consuming task is that of temporarily storing intermediate results in secondary storage to free areas of the program address space. The writing and subsequent reading of polygons from the data base file adds considerable overhead to the processing of the symbols. The overhead is estimated to be between 25 and 30% based on the limited experience with program on small problems. It must be expected that large amounts of data will be generated by polygon manipulations and that it will not all fit into the real memory of the machine at once. Virtual memory architectures provide for the efficient handling of this problem where there is sufficient virtual address space to accomodate all the data. The eighteen bit address size of the DEC-20 limits its virtual address space to a

size much too small for this purpose. This makes necessary the use of the database for temporary storage as described. The delay time in accessing secondary storage cannot be avoided since it is required by the virtual memory faults, as well as by database operations. The overhead of converting the polygons to a form suitable for storage in the database and then their reconstruction upon their retrieval is not required if the virtual memory address space is sufficiently large. It is estimated that the space required is in excess of 16 million bytes to handle chip designs of significant size. This indicates that a machine having a 32 bit virtual address would be desirable for doing polygon manipulations of this magnitude. The estimated performance improvement realized by such a machine is approximately a factor of two. The more important benefit of such a large virtual address space is in the reduction in complexity of the program and in reduced program development time. In general, the burden of managing secondary storage introduces excessive delays and errors upon the programmer. This responsibility should fall to the operating system and the machine hardware and be made transparent to the application programmer.

6.2 Analysis Data

The performance difficulties associated with the program prevent it from being used to gather data from a sizable design. It was hoped that the data would be useful in characterizing the regularized design style. It appears that these designs achieve shorter average connection length and a higher function density than are normally possible with conventional techniques. A cursory look a

such designs reveals that there are few areas occupied by wires alone, there are usually devices and logic functions implemented under them, using the wires in some manner. Also, groups of wires rarely make expensive right angle turns as is common in the "random" designs. These factors should lead to statistics which show the regularized design to have a higher percentage of its area occupied by devices (and/or subsymbols), and a lesser amount used for wiring than conventional chips. In addition, since the bus wires of regular designs are carefully planned to flow through the logic functions in an efficient manner, it should be expected that the symbols of the regular design will have a higher percentage of their wires be global rather than local.

All of these observations should be verifiable through the use of this wiring analysis program. The statistics of one or more regularized designs could be computed and compared with those of typical custom IC designs. The program is capable of providing this data, however, without improvement in the efficiency of the algorithms and the speed and storage capabilities of the host machine, the extraction of the data is much too expensive.

A key aspect of the wiring analysis is its distinction between global and local wiring structures. It is necessary that a number of designs be analyzed to check that the heuristics developed for the program make acceptable decisions between these two types of wire. Several definitions other than the one used here for global wire could be used. One other interesting definition would permit wires to be both global and local. This allows the "double duty" of some wiring structures to be measured

where a wire satisfies a local connection requirement as well as global wiring demand. Analysis of regularized designs should show a larger percentage of wire acting as both global and local than should conventional designs. Such a study must await solutions to the performance difficulties listed above.

Chapter VII

Conclusions

7.1 Factors Influencing Polygon Manipulations

One of the causes for the excessive execution time of the wiring analysis program is the generality of the polygon operations. The extension of polygons to include circular arcs, non-orthogonal edges and holes all increase the cost of polygon manipulations. Existing IC designs show quite definitely that non-orthogonal edges must be available. It is likewise clear that enclosed areas of certain masks must include holes, this is most easily seen in the diffusion layer. However, it is not clear that circular arcs are necessary for the purposes of this program nor necessary for design rule checking.

At the present time IC masks are usually produced by the exposure of rectangular areas of a photographic material. Thus, the designer's specification of the artwork cannot contain shapes which are not decomposable into rectangles. Even straight line polygons having concave vertices or acute angles can be difficult to decompose to rectangles. The only consideration given to circular arcs comes with inflation and deflation of the shapes. The desirable property that circular arcs provide is a more accurate treatment of corner to corner distances in design rule checking. Another property that circular arcs provide not

usually present in straight line polygon packages is symmetry between the inflation and deflation operations. The ability of a package to inflate a polygon by some amount and subsequently deflate it by the same amount resulting in the original polygon can be provided in straight line packages. It does not appear that this ability is necessary since in design rule checking and in this wiring analysis program, polygons are inflated or deflated, used in a logical operation and then thrown away. It is rare that the ability to expand or shrink a polygon back to its original shape is needed.

Wiring analysis, unlike design rule checking, is not concerned with the precise measurement of corner to corner distances. Furthermore, the generation of circular arcs at the vertices of a sheet being inflated doubles the number of edges and vertices of the sheet. It is this effect that incurs most of the cost of maintaining this generality. This increase in the number of edges and vertices results in much longer run times in the selfintersection of polygons using algorithms that are, at best, N log N. Since the properties of circular arcs are not required and more than double the execution time of the program, they should be removed from the polygon package used in the wiring analysis program.

7.2 Desirable Restrictions in the Specification of Artwork

Much of the activity of the wiring analysis program is involved in the recognition of structural elements of the circuit which have a one to one correspondence with physical elements. The transistors and layer to layer

contacts are among these. The required use of special CIF constructs or particular symbols for specifying transistors and contacts would save much effort in the wiring analysis.

If transistors were specified explicitly in the CIF, no recognition procedure would have to be performed. The devices and wires would be separated from the start. There is then no confusion possible in extracting one from the other or in determining the connections to the devices. The designer would then be required to identify all the transistors rather that merely lay polysilicon over diffusion. No doubt design rule checking would be much simplified also.

The exclusive use of symbols to specify transistors should be extended to prohibit the generation of devices by the overlapping of one symbol with another. In the current usage of CIF, designers are permitted to form transistors by allowing the polysilicon of one symbol to overlap the diffusion of another symbol. The automatic detection and checking of such spurious devices is a burden to design aids. The requirement that all transistors be instances of specific symbols and that poly and diffusion do not otherwise cross, will help minimize the complexity of wiring analysis and design rule checking programs.

Similar benefits are to be had if the designer identifies those points on the interconnection layers where external connections are to be made. This information would remove much of the ambiguity inherent in the heuristic methods and would greatly reduce the time required to extract the circuit topology. In addition, it would aid design rule checking programs by defining all legal places for

interaction between symbols.

The current usage of CIF at Caltech includes the practice of using predefined symbols to place contacts between layers. If this practice were required, the wiring analysis program could take advantage of it in two ways. First, if these connections are represented as a point rather than the usual rectangles, the test for possible connection between wires on two different layers is reduced to determining if the contact point is enclosed by both wires. Second, if the rectangular shapes of metal, poly and diffusion that make up proper contacts are not merged into the other wires and shapes of the symbol, a large reduction can be made in the number of edges in the symbol. This would improve the run time of the program dramatically.

The restrictions suggested here in the specification of artwork go beyond the intent of CIF. CIF has been designed to be independent of the process or technology for which the artwork is being specified. The construction of transistors and contacts cannot be standardized between the various MOS processes or with bipolar technologies. It is desirable that CIF remain independent and not become anything but artwork specification. The use of agreed upon symbols, particular to a given process, for transistors and for contacts would provide the needed structural information without violating the intent of CIF. A set of such symbols for each process would also permit the designer to move a design between similar processes by using the proper set of standard symbol definitions.

7.3 The Requirement for Wiring Analysis

The motivation behind analyzing the wiring of IC's changes with the design style and the degree of automation used in the design process. In the case of master slice designs, as with printed circuit cards, a particular function is made to fit a fixed size physical space. The selection of the space (board size, number of layers, master slice dimensions and image, etc.) is done to optimize the number of chips (or boards) and their cost to implement some range of functions. IC designs with a higher degree of customization are not restricted to a particular size die or a fixed set of pin connections or logic elements. The two factors (of interest here) that measure the design of a custom chip are the final dimensions of the die and the time required to design and debug it. Thus the wiring analysis of custom IC's must provide a tool that will serve to reduce the overall size of the die.

Two types of custom IC design are contrasted here. The conventional method involves the design and placement of numerous cells which are then interconnected, usually automatically, to form a higher level function. The polycell approach does exactly this but with only one level of subcells. The general custom chip may have several hierarchial levels but may be placed and routed by hand because few tools exist for automatically assembling arbitrary IC designs. It is easy to imagine hierarchial box packing, placement and routing for designs of this nature. To insure that designs are assembled and interconnected using near minimum area, cost functions and wiring models are required which allocate chip resources

during chip assembly in a way that minimizes the overall die size. The wiring analysis program can provide data on which to base assembly and placement algorithms and a means by which they can be evaluated.

The other type of custom design is the regularized style of Mead and Conway[2]. The Bristle Blocks program [4] is the epitome of the type of design performed automatically for a class of sequential machine architectures. This prototype silicon compiler is capable of generating mask geometry from a single sheet description of the desired machine and a library of procedurized cells. As a general rule the execution of the Bristle Blocks program to completely design a chip is less than the time required to perform the wiring analysis upon it (using similar machines and programming languages). No placement or routing is performed by Bristle Blocks in the conventional way. The overall interconnection strategy is dictated by the bus architecture and the user specification. It is apparent that if such design systems do not require wiring models to insure good placement and routing algorithms and if the time required to completely design the chip is less than a month, the need for some of the area estimates and cost functions provided by the data extracted by a wiring analysis program may disappear. However, if the cost of obtaining the wiring data can be reduced to a point where it remains less that then cost of the design, then the extraction of wiring data will remain attractive.

One method of reducing these costs would be to integrate the computation of wiring statistics into the silicon compiler. As it executes, generating the geometry of the design, it could also compute global and local wire lengths

and all the necessary and interesting quantities involved. If these computations are built in, they become very much cheaper to provide. This is due primarily to the existence of topological information within the silicon compiler. Once the geometry is produced much of the topology of the design is lost or hidden and a large amount of computation is required to extract it.

Both types of custom design can be expected to be used in the future. A silicon compiler such as Bristle Blocks may provide an easy means of automatically designing a wide variety of IC's but will nevertheless be unsatisfactory for some purposes. In these cases designs can be expected to utilize conventional design automation tools. This second type requires the feedback of wiring data to develop and tune its algorithms. Both types of design can be evaluated by the extraction of wiring data.

As with design rule checking, the extraction of wiring data is a type of reverse engineering applied to the geometry. It endeavors to reconstruct the topology of the design and to some extent recreate the designer's thought processes. The generation of this type of data is many times ambiguous and subject to many exceptions. There appears to be a fundamental difficulty associated with the automatic generation of data across any two of the three description domains: physical, structural and behavioral. The usual purpose behind reverse engineering designs is to provide verification of correctness. Such programs are usually very expensive to use and the author's program is no exception. One additional conclusion is suggested by the economics of CAD systems. That is, rather than attempt to exhaustively check and analyze designs, perhaps more

emphasis should be placed on design tools that generate correct designs by construction. This is a desirable goal but it is also one that is very difficult to attain. Large programs of any type are notoriously bug ridden, particularly in their introductory periods. Many are never made error free. This propensity for programming errors is a serious matter itself and it introduces yet another source of errors, that is, manual intervention.

Whether or not design tools have generated errors, it is difficult to prevent, or to prevent the occasional need for, manual modification of the geometry of a design. If this is ever done to a design, any prior assurances of correctness are worthless and the new geometry must be checked and analyzed. Even minor, local changes to the geometry can cause major pertubations. These are analogous to the binary patches made to operating systems bypassing all of the checking performed by compilers (or to a lesser extent by assemblers). It can be considered axiomatic that such low level modifications of a complex system are never successful over the life of the system.

A method which provides checking and analysis but avoids reverse engineering of designs is to incorporate incremental checks and statistics computations within the design tools. Rather than run checking and analysis programs on the output of design systems the checking and analysis should be integrated into the programs which generate the geometry. There are two reasons why this reduces the cost of checking and analyzing geometry. First, the checking and analysis functions are made hierarchial and incremental merely because they are executed in conjunction with design tools which are

inherently hierarchial and incremental. Second, vastly more information is available to checking and analysis functions from inside these design tools than is found in the geometry. Topology, behavior and to some extent the intent of the designer may be accessible to the checking and analysis procedures. This eliminates the need for the greatest part of the computations associated with checking and analysis programs. In addition, the accuracy of these programs is improved since the errors, exceptions and assumptions associated with the extraction of the topology or designer intent are also eliminated. With lower costs and improved accuracy, such geometric analysis programs become very attractive tools.

References

[1] W. R. Heller, W. F. Mikhail and W. E Donath, "Prediction of Wiring Space Requirements for LSI", Journal of Design Automation and Fault Tolerant Computing, June 1978, pp. 117-144.

[2] C. A. Mead and L. A. Conway, Introduction to VLSI Systems, Addison-Wesley, 1980.

[3] W. R. Heller, "Wirability Study for Custom Chips", Silicon Structures Project File 1452, California Institute of Technology, May 6, 1978.

[4] D. Johannsen, "Bristle Blocks: A Silicon Compiler", 16th Design Automation Conference Proceedings, June 1979, pp. 310-313.

[5] B. S. Landman and R. L. Russo, "On a Pin Versus Block Relationship for Partitions of Logic Graphs", IEEE Transactions on Computers C-20 (1971) pp. 1469-1479.

[6] I. E. Sutherland, "The Polygon Package", Silicon Structures Project File 1438, California Institute of Technology, February 15, 1978.

[7] M. Ullner, "Relational Database Program (RDP) User's Guide", Silicon Structures Project File 2803, California Institute of Technology, May 1, 1979.

[8] G. Tarolli and J. Rowson,"CIF20 Instruction Manual", Silicon Structures Project File 2777, California Institute of Technology, April 25, 1979.

[9] I. E. Sutherland, "Computing the Wrap Number", Display File 1146, California Institute of Technology, November 9, 1977.

[10] I. E Sutherland, "Broadening Polygons: Straight and Circular Edge Conventions", Display File 1129, California Institute of Technology, November 9, 1977.

[11] I. E Sutherland and R. Sproull, "An Overlap Detector", Silicon Structures Project File 1322, California Institute of Technology, January 19, 1978.

[12] W. Newman and R. Sproull, Principles of Interactive Computer Graphics, McGraw-Hill, New York, 1973.

[13] G. Birtwistle, L. Enderin, M. Ohlin and J. Palme, DECSYSTEM-10 SIMULA Language Handbook, Swedish National Defense Research Institute and the Norwegian Computing Center (reports C8398, C8399 and C10045).

[14] G. Birtwistle, O. Dahl, B. Myhrhaug and K. Nygaard, Simula Begin, Petrocelli/Charter, New York, 1975.

[15] J. E. Warnock, "A Hidden-Surface Algorithm for Computer Generated Half-tone Pictures", TR 4-15, Computer

Science Department, University of Utah, 1969.

[16] E. Barton and I. Buchanan, "The Polygon Package", Department of Computer Science, University of Edinburgh, 1979.

[17] J. Wipfli, "A Simula Graphics Package", Silicon Structures Project File 1929, California Institute of Technology, 1978.

[18] B. Locanthi and J. Rowson, "Things": Simula Program, Computer Science Department, California Institute of Technology, 1978.

Appendix I

A Skeletal Specification of a Simula Polygon Package

```
!                                                                    }
!    THIS IS A SKELETAL SPECIFICATION FOR A POLYGON PACKAGE          }
!                                                                    }
!                                          Dick Lang 6/79            }
OPTIONS(/e);

!*****************************************************************;
!  This is CLASS Plygns.                                         *;
!  Externals things, displa and views are necessary along        *;
!  with their own particular externals (listed below).           *;
!*****************************************************************;
EXTERNAL CLASS Things,Displa,Views;
EXTERNAL INTEGER PROCEDURE hash,imax,imin,xwd,lnot,lor,land;
EXTERNAL REAL PROCEDURE rmin,rmax;
EXTERNAL TEXT PROCEDURE upcase,getitem,frontstrip,conc,initem;
EXTERNAL BOOLEAN PROCEDURE jsys;
EXTERNAL CHARACTER PROCEDURE getch;
EXTERNAL PROCEDURE enterdebug,sleep,qwkout;


Views CLASS Plygns;
BEGIN


    Thing CLASS Polygon;
    !A polygon is a set of zero or more sheets.                      ;
    !Sheets are ordered sets of directed edges which enclose some    ;
    !two-dimensional area. The substance of a sheet is assumed to    ;
    !be to the left of the edge (when facing in its direction)       ;
    VIRTUAL:
    PROCEDURE Print,Plot;
    INTEGER PROCEDURE Wrap;
    REAL PROCEDURE Area,Perimeter;
    REF(polygon) PROCEDURE Clean,Xform,Copy,Inflate,Inverted,ClippedBy,
                           Deflate,Combine;
    REF(point) PROCEDURE Centroid;
    BOOLEAN PROCEDURE Overlaps;

    BEGIN

        REF(polygon) PROCEDURE TakeVectorAsSheet(v); REF(vector)v;
        !This procedure takes a vector (see things.sim for definition)       ;
        !of points (also in things.sim) uses them in sequence to             ;
        !generate a new sheet and put it in the set of sheets of this polygon.;
```

```
!This polygon is returned and v is not modified.  Sheets              ;
!entered in this manner are required to be well-formed
!(non-selfinterseting).                                               ;
!There is an assumed edge between the last vertex of the vector       ;
!and the first vertex.  All edges implied by the vector of points     ;
!are assumed to be straight lines.                                    ;
BEGIN
END of TakeVectorAsSheet;

REF(polygon) PROCEDURE Combine(p); NAME p; REF(polygon)p;
!This procedure takes all the sheets of p and assigns them to         ;
!this polygon and returns this polygon.  P gets NONE.                 ;
BEGIN
END of Combine;

REF(rectangle) PROCEDURE Mbb;
!This procedure returns the minimum bounding box of this polygon.     ;
BEGIN
END of Mbb;

PROCEDURE Plot(device); REF(plotter)device;
!This procedure plots this polygon on the given plotter.              ;
!Other parameters may be desirable to provide for various             ;
!types of plots and options. Views.sim and Displa.sim                 ;
!contain procedures for manipulating various devices.                 ;
BEGIN
END of Plot;

PROCEDURE Print(outf); REF(outfile)outf;
!This procedure prints all the sheets of this polygon in              ;
!any suitable form.                                                   ;
BEGIN
END of Print;

REF(polygon) PROCEDURE Copy;
!This procedure generates and returns a new copy of this              ;
!polygon.                                                             ;
BEGIN
    REF(sheet)s;
    REF(polygon)p;
    p:-NEW polygon;
    FOR s:-nextsheet(s) WHILE s=/=NONE DO p.TakePolygon(s.Copy);
    Copy:-p;
END of Copy;

REF(polygon) PROCEDURE ClippedBy(box); REF(rectangle)box;
!This procedure modifies this polygon and returns it after            ;
!intersecting it with the box parameter.                              ;
BEGIN
    REF(sheet)s;
    FOR s:-nextsheet(s) WHILE s=/=NONE DO s.ClippedBy(box);
    ClippedBy:-THIS polygon;
```

```
END of ClippedBy;

INTEGER PROCEDURE NumSheets;
!This procedure returns the integer number of sheets belonging      ;
!to this polygon.                                                   ;
BEGIN
END of NumSheets;

REF(sheet) PROCEDURE NextSheet(s); NAME s; REF(sheet)s;
!This procedure provides access to the sheets of a polygon.         ;
!If S==NONE then this procedure returns the first sheet             ;
!of this polygon, Otherwise it returns the sheet following          ;
! S. If no more sheets remain, NONE is returned. No ordering       ;
!of sheets is implied.                                              ;
BEGIN
END of NextSheet;

INTEGER PROCEDURE Wrap(pt); REF(point)pt;
!This procedure returns an integer which is the absolute            ;
!wrap number of all the sheets of this polygon about the            ;
!point pt.  If pt is on an edge of a sheet the result is undefined. ;
BEGIN
    REF(sheet)s;
    INTEGER w;
    FOR s:-nextsheet(s) WHILE s=/=NONE DO w:=w+s.wrap;
    Wrap:=w;
END of Wrap;

REAL PROCEDURE Area;
!This procedure returns the net area of this polygon.               ;
BEGIN
    REAL a;
    REF(sheet)s;
    FOR s:-nextsheet(s) WHILE s=/=NONE DO a:=a+s.Area;
    Area:=a;
END of Area;

REAL PROCEDURE Perimeter;
!This procedure returns the total perimeter of this polygon         ;
!(the sum of the perimeters of all the sheets).                     ;
BEGIN
    REAL p;
    REF(sheet)s;
    FOR s:-nextsheet(s) WHILE s=/=NONE DO p:=p+s.perimeter;
    Perimeter:=p;
END of Perimeter;

REF(point) PROCEDURE Centroid;
!This procedure returns a point which is the centroid of            ;
!this polygon.                                                      ;
BEGIN
END of Centroid;
```

```
REF(polygon) PROCEDURE Xform(tr); REF(transform)tr;
!Given a transform (defined in things.sim) this procedure       ;
!applies the transformation to this polygon, modifying it and   ;
!returning it.                                                  ;
BEGIN
    REF(sheet)s;
    FOR s:-nextsheet(s) WHILE s=/=NONE DO s.Xform(tr);
    Xform:-THIS polygon;
END of Xform;

REF(sheet) PROCEDURE ExtractSheet(s); REF(sheet)s;
!This procedure removes sheet s from this polygon and returns  ;
!it.  An error results if s does not belong to this polygon.   ;
BEGIN
END of ExtractSheet;

REF(polygon) PROCEDURE SelfIntersect;
!This procedure examines this polygon and modifies it as follows:  ;
!New vertices are added to sheets wherever an edge of one sheet    ;
!intersects an edge of itself or another sheet of this polygon.    ;
!The new vertices are redundant and do not affect the shape or     ;
!outline of the polygon.  This polygon is returned.                ;
BEGIN
END of SelfIntersect;

REF(polygon) PROCEDURE Retrace;
!This procedure modifies this polygon by reassigning the edges of  ;
!the polygon between its sheets.  The resulting set of sheets      ;
!are non-overlapping but may be coincident at a point or edge.     ;
!That is, edges of one sheet may not cross other edges, either     ;
!its own or those of other sheets. Note that again neither the     ;
!shape or outline of the polygon is affected by this operation     ;
!Also, the direction associated with edges is not affected         ;
!preserving the same wrap number of all points with respect to     ;
!this polygon.                                                     ;
BEGIN
END of Retrace;

REF(polygon) PROCEDURE Clean;
!This procedure cleans each sheet of this polygon in turn.         ;
!It removes all redundant vertices and edges and effectively       ;
!undoes what SelfIntersect does.                                   ;
BEGIN
    REF(sheet)s;
    FOR s:-nextsheet(s) WHILE s=/=NONE DO s.Clean;
    Clean:-THIS polygon;
END of Clean;

REF(polygon) PROCEDURE Inverted;
!This procedure reverses the direction of all edges of all sheets  ;
!in this polygon (thus modifying it).  This effectively turns      ;
```

```
!holes into substance and substance into holes.                    ;
BEGIN
   REF(sheet)s;
   FOR s:-nextsheet(s) WHILE s=/=NONE DO s.Inverted;
   Inverted:-THIS polygon;
END of Inverted;

REF(polygon) PROCEDURE Union(p); NAME p; REF(polygon)p;
!This procedure finds the logical union of this polygon and P.     ;
!P is destroyed and this polygon becomes the union of the two.     ;
!This polygon is returned and P gets NONE.                         ;
BEGIN
END of Union;

REF(polygon) PROCEDURE Intersect(p); NAME p; REF(polygon)p;
!This procedure finds the intersection of this polygon and         ;
!P.  P is destroyed and P gets NONE. This polygon becomes the      ;
!intersection of the two and is returned.                          ;
BEGIN
END of Intersect;

REF(polygon) PROCEDURE ExclusiveOr(p); NAME p; REF(polygon)p;
!This procedure finds the exclusive or of this polygon and p       ;
!this is equivalent to:                                            ;
!                                                                  ;
!  p1.copy.intersect(p2.copy.inverted).union(p2.intersect(p1))     ;
!                                                                  ;
!Again, p is destroyed and gets NONE, and this polygon is          ;
!modified to contain the result that is returned.                  ;
BEGIN
END of ExclusiveOr;

REF(polygon) PROCEDURE Subtract(p); NAME p; REF(polygon)p;
!this procedure finds the difference of this polygon and p         ;
! it is equivalent to:                                             ;
!                                                                  ;
!                      p1.intersect(p2.inverted)                   ;
!                                                                  ;
!Again, p is destroyed and gets NONE, and this polygon is          ;
!modified to contain the result that is returned.                  ;
BEGIN
END of Subtract;

REF(polygon) PROCEDURE ExtractMatching(p); REF(polygon)p;
!This procedure finds those sheets of this polygon which           ;
!overlap those of p.  These sheets are extracted from this         ;
!polygon and a new polygon is created from them and returned.      ;
!P is not modified by this procedure and this polygon is changed   ;
!only to the extent of the extraction of the matching sheets.      ;
BEGIN
END of ExtractMatching;
```

```
BOOLEAN PROCEDURE Overlaps(p); REF(polygon)p;
! This procedure returns true if this polygon touches, or        ;
! overlaps, or encloses, or is enclosed by p.  Neither           ;
! this polygon or p is modified by this procedure.               ;
! Enclosed means in a counterclockwise manner here.              ;
BEGIN
END;

REF(polygon) PROCEDURE ExtractWrapping(pt); REF(point)pt;
!This procedure removes all sheets of this polygon which        ;
!wrap the point pt with a non-zero wrap number.  The sheets      ;
!selected are put into a new polygon and it is this polygon that ;
!returned.                                                      ;
BEGIN
    REF(sheet)s;
    REF(polygon)p;
    p:-NEW polygon;
    FOR s:-nextsheet(s) WHILE s=/=NONE DO
    IF s.wrap(pt)\=0 THEN p.TakePolygon(ExtractSheet(s));
    ExtractWrapping:-p;
END of ExtractWrapping;

REF(polygon) PROCEDURE ExtractHoles;
!This procedure removes from this polygon and returns as a new  ;
!polygon, all sheets of this polygon which enclose an area in a ;
!clockwise direction.  This is characterized by the wrap number ;
!of points within the sheet being lower than that of points     ;
!outside the sheet with respect to all the other sheets of this ;
!polygon.  This operation only has meaning if the sheets are all ;
!well formed (selfintersection and retracing has be done).      ;
BEGIN
END of ExtractHoles;

REF(polygon) PROCEDURE Trim(w); INTEGER w;
!This procedure finds those sheets which enclose points         ;
!having a wrap number greater than or equal to w.               ;
!These sheets are returned as a new polygon.                    ;
!This polygon is destroyed and replaced by the new one.         ;
!Sheets which enclose points with a wrap number greater         ;
!than w which themselves are enclosed by sheets wrapping w      ;
!are excluded.                                                  ;
BEGIN
END of Trim;

REF(polygon) PROCEDURE Inflate(size); REAL size;
!This procedure modifies and returns this polygon after         ;
!inflating each of its sheets by "size" (or deflating if size is ;
!negative). This may or may not generate circular edges         ;
!depending on the option flags.                                 ;
BEGIN
    REF(sheet)s;
    FOR s:-nextsheet(s) WHILE s=/=NONE DO s.Inflate(size);
```

```
    END of Inflate;

    REF(polygon) PROCEDURE Deflate(size); REAL size;
    !This procedure only provides selfdocumentation by calling        }
    !Inflate with a negative parameter.                               }
    BEGIN
        REF(sheet)s;
        FOR s:-nextsheet(s) WHILE s=/=NONE DO s.Inflate(-size);
    END of Deflate;

END of CLASS polygon;


polygon CLASS sheet;
!Sheets are ordered sets of edges (and/or vertices).                  }
!The set must enclose some area or areas one or more                  }
!times with a single continuous sequence of edges.                    }
!The "meat" of the sheet is assumed to be to the left of              }
!the edges when facing in the direction of the edge.                  }
BEGIN

    REF(rectangle) PROCEDURE Mbb;
    !This procedure returns the minimum bounding box of               }
    !this sheet.                                                      }
    BEGIN
    END of Mbb;

    PROCEDURE Print(outf); REF(outfile)outf;
    !This procedure prints the contents of this sheet is some         }
    !suitable form.                                                   }
    BEGIN
    END of Print;

    PROCEDURE Plot(device); REF(plotter)device;
    !This procedure plots this sheet on the selected device           }
    !parameters to this procedure may change to accomodate            }
    !plotting options.                                                }
    BEGIN
    END of Plot;

    REF(polygon) PROCEDURE Inflate(size); REAL size;
    !This procedure moves every edge of this sheet to the right       }
    !by amount "size". If size is negative, the edges are             }
    !moved to the left. The directions are with reference to          }
    !the direction of the edge. This sheet is modified and returned   }
    !(as a polygon).                                                  }
    BEGIN
        Inflate:-THIS sheet;
    END of Inflate;

    REF(polygon) PROCEDURE Deflate(size); REAL size;
    !This procedure merely calls Inflate with a negative parameter.   }
```

```
Deflate:-Inflate(-size);

REF(polygon) PROCEDURE Copy;
!This procedure generates a new sheet which is a copy of this   ;
!sheet and returns it as a polygon.                             ;
BEGIN
    REF(sheet)s;
    Copy:-s;
END of Copy;

INTEGER PROCEDURE Wrap(pt); REF(point)pt;
!This procedure returns the wrap number of this sheet about pt.  ;
BEGIN
END of Wrap;

REAL PROCEDURE Area;
!This procedure computes the signed area enclosed by this sheet.   ;
!Holes are defined to have negative area.                          ;
BEGIN
END of Area;

REAL PROCEDURE Perimeter;
!This procedure returns the sum of the length of all the        ;
!edges of this sheet.                                           ;
BEGIN
END of Perimeter;

REF(point) PROCEDURE Centroid;
!This procedure returns a point which is the centroid of this sheet.  ;
BEGIN
END of Centroid;

REF(polygon) PROCEDURE Xform(tr); REF(transform)tr;
!This procedure modifies and returns this sheet (as a polygon)     ;
!after applying to it transformation matrix tr.                    ;
BEGIN
    Xform:-THIS sheet;
END of Xform;

REF(polygon) PROCEDURE Clean;
!This procedure removes all redundant vertices and edges        ;
!from this sheet and returns it (as a polygon).                 ;
BEGIN
    Clean:-THIS sheet;
END of Clean;

REF(polygon) PROCEDURE Inverted;
!This procedure reverses the order and direction of all         ;
!edges and vertices of the sheet.  This sheet is modified       ;
!and returned.                                                  ;
BEGIN
    Inverted:-THIS sheet;
```

```
    END of Inverted;

    BOOLEAN PROCEDURE Overlaps(p); REF(polygon)p;
    ! This procedure returns true if this polygon touches, or        ;
    ! overlaps, or encloses, or is enclosed by p.  Neither           ;
    ! this polygon or p is modified by this procedure.               ;
    ! Enclosed means in a counterclockwise manner here.              ;
    BEGIN
    END;

    REF(polygon) PROCEDURE ClippedBy(box); REF(rectangle)box        ;
    !This procedure replaces this sheet with the intersection       ;
    !of this sheet and box and returns itself.                      ;
    BEGIN
    END of ClippedBy;

    REF(polygon) PROCEDURE Combine(p); REF((polygon)p;
    ! This procedure takes all the sheets of p and attaches them    ;
    ! to a new polygon made using this one sheet.  This polygon     ;
    ! is returned and p becomes an empty polygon.                   ;
    BEGIN
    END of Combine;


END of CLASS sheets;


!Option and Trace functions and procedures;

REAL tol;                   !tolerance for geometric operations;
BOOLEAN circles;            !boolean to indicate if circular arcs;
                                !are to be generated;
BOOLEAN ARRAY debug[1:9];   !other flags for trace purposes (TBD);

REF(pflags)PolygonFlags;    !object to manipulate flags;

thing CLASS pflags;
!A simple-minded object for the setting and clearing of polygon ;
!package option and trace flags.                                 ;
BEGIN

    PROCEDURE TurnOnCircles;
    circles:=TRUE;

    PROCEDURE TurnOffCircles;
    circles:=FALSE;

    PROCEDURE Tolerance(size); REAL size;
    tol:=size;

    PROCEDURE DebugOn(n); INTEGER n;
    debug[n]:=TRUE;
```

```
        PROCEDURE DebugOff(n); INTEGER n;
        debug[n]:=FALSE;

    END of pflags;


    !Init Code for the Polygons Block;
    PolygonFlags:-NEW pflags;

END of Plygns Block;
```

Appendix II

Simula Definition of CLASS Symbol

```
OPTIONS(/e);
!*********************************************************;
!                                                         ;
!     This is class SYM which uses:                       ;
!          things,displa,views,plygns,plyrdp.             ;
!                                                         ;
!     This implements a class object which has            ;
!     the geometric and structural attributes of          ;
!     of layout symbol.  SYMBOLs get their data           ;
!     from a relational database file that can be         ;
!     made using CIFX.  Also, the polygonal data          ;
!     and other stuff inside the symbol may be            ;
!     stored on disk in temporary relational database     ;
!     files and later retrieved permitting better         ;
!     memory utilization.                                 ;
!                                                         ;
!                                Dick Lang 6/79           ;
!*********************************************************;

EXTERNAL CLASS Things,Displa,Views,Plygns,Plyrdp;

!Things, Displa and Views externals (those not used by RDP);
EXTERNAL INTEGER PROCEDURE hash,imax,imin;
EXTERNAL REAL PROCEDURE rmin,rmax;
EXTERNAL TEXT PROCEDURE upcase,getitem,frontstrip,conc,initem;
EXTERNAL CHARACTER PROCEDURE getch;
EXTERNAL PROCEDURE enterdebug,sleep,qwkout;

!RDP externals;
EXTERNAL         PROCEDURE tinp, tinsertsort, tmove, tpmove, tout,
                           txtinp, txtout;
EXTERNAL BOOLEAN PROCEDURE jsys, teq, tlt, tpeq;
EXTERNAL INTEGER PROCEDURE aaddr, getf, ioaddress, land, left, lnot, lor,
                           lshift, lxor, realasbits, right, setf, taddr,
                           tpartition, tpxor, tsize, xwd;
EXTERNAL  REAL   PROCEDURE bitsasreal;

! New externals;
REAL PROCEDURE random;

!*********************************************************************;
```

```
Plyrdp CLASS sym;

BEGIN
    INTEGER poly,diffusion,metal,cuts,implant,butcon,glass;
    REF(relation)calls,polys,boxes,wires,bboxes;
    REF(relation)savepolys;
    REF(dictionary)allsymbols;
    REAL ARRAY spacing(1:7), width(1:7);



    thing CLASS instance(sym,tr); REF(symbol)sym; REF(transform)tr;
    ! This object serves to allow subsymbols to be put in vectors  ;
    ! with a transformation.                                        ;
    BEGIN END;



    thing CLASS Symbol(symbolnumber); INTEGER symbolnumber;
    BEGIN
        REF(polygon) ARRAY layer(1:7), hull(1:3);, xcontacts(1:3),
                           localwire(1:3), globalwire(1:3);
        REF(polygon) annulus, ndevices, pdevices;
        REF(vector) subsymbols;
        BOOLEAN empty,inuse,analyzed;
        REF(rectangle)bb;


        REF(symbol) PROCEDURE Store;
        ! This procedure puts all of the data attributes of this  ;
        ! symbol into the temporary database file.                 ;
        ! This symbol is returned, but is then empty.              ;
        BEGIN
            IF NOT empty THEN BEGIN
                INTEGER pnumber,i;

                PROCEDURE putinrdb(p); REF(polygon)p;
                ! This procedure puts a single polygon into       ;
                ! the database.                                    ;
                BEGIN
                    IF p=/=NONE THEN BEGIN
                        REF(sheet)s;
                        REF(rlntuple)t;
                        INTEGER sheetnumber,itemnumber;
                        t:-savepolys.prototuple;
                        sheetnumber:=0;
                        t.setint(1,symbolnumber).setint(2,pnumber);
                        ! Get each sheet of the polygon;
                        FOR s:-p.nextsheet(s) WHILE s=/=NONE DO BEGIN
                            REF(ringer)item;
                            itemnumber:=0;
                            sheetnumber:=sheetnumber+1;
```

```
                    t.setint(3,sheetnumber);
                    ! Get each item of each sheet and put into the rdb;
                    FOR item:-s.next(item) WHILE item=/=NONE DO BEGIN
                        itemnumber:=itemnumber+1;
                        t.setint(4,itemnumber);
                        INSPECT item
                        WHEN vertex DO
                            t.setint(5,1).setflt(6,x).setflt(7,y)
                                                .setflt(8,0.0)
                        WHEN line DO
                            t.setint(5,2).setflt(6,a).setflt(7,b)
                                                .setflt(8,c)
                        WHEN circle DO
                            t.setint(5,3).setflt(6,x).setflt(7,y)
                                                .setflt(8,r);
                        savepolys.taketuple(t);
                    END;
                END;
            END;
        END of putinrdb;

        ! Take each polygon belonging to this symbol and  ;
        ! put it inot the database and set it to none to  ;
        ! permit the memory space to be freed.            ;
        FOR i:=1 STEP 1 UNTIL 7 DO BEGIN
            putinrdb(layer(i));     layer(i):-NONE;
        END;
        FOR i:=1 STEP 1 UNTIL 3 DO BEGIN
            putinrdb(hull(i));      hull(i):-NONE;
            putinrdb(xcontacts(i)); xcontacts(i):-NONE;
            putinrdb(globalwire(i)); globalwire(i):-NONE
            putinrdb(localwire(i)); localwire(i):-NONE;
        END;
        putinrdb(annulus);   annulus:-NONE;
        putinrdb(ndevices);  ndevices:-NONE;
        putinrdb(ddevices);  ddevices:-NONE;

        savepolys.reset;
        numberstored:=numberstored+1;
    END;

    Store:-THIS symbol;
    empty:-TRUE;
END of Store;


REF(symbol) PROCEDURE Retrieve;
! This procedure restores the data to a symbol if it is  ;
! empty.  If it is not empty, it does nothing.           ;
BEGIN
    IF empty THEN BEGIN
        REF(relation)mypolys;
```

```
INTEGER i,n;

REF(polygon) PROCEDURE getnextpoly;
! This procedure gets this symbol's next polygon  ;
! and returns it as its value. The polygon number ;
! is returned in n.                               ;
BEGIN
    REF(polygon)p;
    REF(sheet)s;
    REF(rlntuple)t;
    BOOLEAN moreforp;
    INTEGER sheetnumber,lastsheet,itemtype;
    t:-mypolys.prototuple;
    moreforp:=TRUE;
    lastsheet:=0;
    p:-NEW polygon;
    ! Loop through the tuples of this symbol;
    WHILE mypolys.more AND moreforp DO BEGIN
        mypolys.givetuple(t);
        ! Test to see if polygon n is next;
        IF t.getint(2)=n THEN BEGIN
            ! If it is, then get a new sheet number;
            sheetnumber:=t.getint(3);
            IF lastsheet\=sheetnumber THEN BEGIN
                ! If it a new sheet, put the last one into;
                ! the polygon and start a new one.        ;
                IF lastsheet\=0 THEN p.combine(s);
                s:-NEW sheet;
                lastsheet:=sheetnumber;
            END;
            ! Get the vertex,line or circle out of the tuple ;
            ! and put in in the current sheet of the polygon.;
            itemtype:=t.getint(5);
            IF itemtype=1 THEN
                s.take(NEW vertex(t.getflt(6),t.getflt(7))
            ELSE IF itemtype=2 THEN
                s.take(NEW line(t.getflt(6),t.getflt(7),
                                            t.getflt(8))
            ELSE IF itemtype=3 THEN
                s.take(NEW circle(t.getflt(6),t.getflt(7),
                                            t.getflt(8));
        END ELSE BEGIN
            ! If this tuple is not one of n's then increment ;
            ! n, fall out of loop and back up to previous tuple;
            n:=n+1;
            moreforp:=FALSE;
            mypolys.backspace;
        END;
    END;
    getnextpoly:-p;
END of getnextpoly;
```

```
        ! Get all the tuples saved for this symbol. ;
        mypolys:savepolys.-protorelation;
        template:-mypolys.prototuple.setint(1,symbolnumber);
        mypolys:-savepolys.extracttuple(template,1);
        n:=1;

        ! Get the polygons for each symbol attribute;
        FOR i:=1 STEP 1 UNTIL 7 DO layer(i):-getnextpoly;
        FOR i:=1 STEP 1 UNTIL 3 DO BEGIN
            hull(i):-getnextpoly;
            xcontacts(i):-getnextpoly;
            globalwire(i):-getnextpoly;
            localwire(i):-getnextpoly;
        END;
        annulus:-getnextpoly;
        ndevices:-getnextpoly;
        ddevices:-getnextpoly;

        ! Release the temporary relation and check to see if;
        ! some other symbol should be purged.              ;
        mypolys.shatter;
        numberstored:=numberstored-1;
        inuse:=TRUE;
        possiblypurge;
        inuse:=FALSE;
    END;

    Retrieve:-THIS symbol;
    empty:-FALSE;
END of Retrieve;


REF(rectangle) PROCEDURE Mbb;
! This procedure returns the Minimum bounding box of this symbol.;
BEGIN
    Mbb:-bb;
END of Mbb;


REF(symbol) PROCEDURE instantiate(s); REF(instance)s;
! This procedure instantiates all geometry and subsymbols ;
! of symbol s into this symbol.  S must be a pointer      ;
! to a symbol matching a pointer to a symbol in the       ;
! subsymbols vector of this symbol.  If s==NONE then      ;
! all the symbols in  the subsymbols vector are           ;
! instantiated.  Only copies of the data to be            ;
! instantiated are used to avoid affecting anything but   ;
! this symbol.  This symbol is returned.  This is a one   ;
! level instantiation, that is, no subsymbols of          ;
! subsymbols are instantiated.                            ;
BEGIN
    INTEGER length,i;
```

```
BOOLEAN ARRAY addition(1:7);

PROCEDURE bringin(s); REF(symbol)s;
! Bring in all the stuff of one subsymbol  ;
! but don't use selfintersect or retrace.  ;
BEGIN
    INTEGER length,i;

    ! Get all the subsymbols of subsymbol s ;
    length:=s.subsymbols.length;
    IF length\=0 THEN FOR i:=1 STEP 1 UNTIL length DO BEGIN
        REF(instance)inst;
        inst:-s.subsymbols.val(i) QUA instance;
        newinst:-NEW instance;
        newinst.sym:-inst.sym;
        newinst.tr:-s.tr.concatenate(inst.tr);
        subsymbols.append(newinst QUA thing);
    END;

    ! Get the shapes in each layer of subsymbol s and      ;
    ! transform a copy of them to be put into this symbol. ;
    IF s.sym.empty THEN s.sym.retrieve;
    FOR i:=1 STEP 1 UNTIL 7 DO
    IF s.sym.layer(i)==NONE THEN BEGIN
        layer(i).combine(s.sym.layer(i).copy.xform(s.tr));
        addition(i):=TRUE;
    END;
END of bringin;

length:=subsymbols.length;
oldsubs:-subsymbols;
subsymbols:-NEW vector;
inuse:=TRUE;

IF s==NONE THEN
    ! If s is NONE then instantiate all the subsymbols ;
    FOR i:=1 STEP 1 UNTIL length DO
    bringin(oldsubs.val(i) QUA instance)
ELSE BEGIN
    FOR i:=1 STEP 1 UNTIL length DO
        IF oldsubs.val(i)=/=s THEN subsymbol.append(oldsubs.val(i));
    bringin(s);
END;

FOR i:=1 STEP 1 UNTIL 7 DO
    IF addition(i) THEN layer(i).selfintersect.retrace.trim(i);

inuse:=FALSE;
Instantiate:-THIS symbol;
```

```
END of Instantiate;


REF(symbol) PROCEDURE Collapsed;
! This procedure instantiates all levels until this symbol  ;
! is devoid of subsymbols and is strictly geometry.          ;
BEGIN
    WHILE subsymbols.length\=0 DO instantiate(NONE);
    Collapsed:-THIS symbol;
END of Collapsed;


PROCEDURE Plot;
! This procedure plots all the geometry of this symbol     ;
! and asks all its subsymbols to do the same.              ;
BEGIN
    INTEGER length,i;

    IF empty THEN retrieve;

    FOR i:=1 STEP 1 UNTIL 7 DO BEGIN
        selectcolor(i);
        layer(i).plot;
    END;

    length:=subsymbols.length;
    FOR i:=1 STEP 1 UNTIL length DO
        subsymbols.val(i) QUA symbol.plot;

END of Plot;


REF(symbol) PROCEDURE Xform(tr); REF(transform)tr;
! This procedure applies transformation tr to all the      ;
! polygons and instances of this symbol and returns        ;
! this symbol.                                             ;
BEGIN
    INTEGER ilength,;

    IF empty THEN retrieve;

    ! Apply the transform to all the polygons of the symbol;
    FOR i:=1 STEP 1 UNTIL 7 DO layer(i).Xform(tr);
    FOR i:=1 STEP 1 UNTIL 3 DO hull(i).Xform(tr);
    FOR i:=1 STEP 1 UNTIL 3 DO xcontacts(i).Xform(tr);
    annulus.Xform(tr);
    ndevices.Xform(tr);
    ddevices.Xform(tr);

    ! Apply tr to each instance of a subsymbol;
    length:=subsymbols.length;
    FOR i:=1 STEP 1 UNTIL length DO BEGIN
```

```
        REF(instance)inst;
        inst:-subsymbols.val(i) QUA instance;
        inst.tr:-inst.tr.concatentate(tr);
    END;

    Xform:-THIS symbol;
END of Xform;


! Init code for symbols. Get the stuff of the symbol from  ;
! the relations of the the CIF database.                    ;
BEGIN
    REF(relation)mycalls,mypolys,myboxes,mywires,mybbox;
    REF(rlntuple)call,poly,box,wire,template,bbox;
    REF(vector)v;
    INTEGER objectnumber,currentlayer,currentwidth,i;
    empty:=TRUE;

    ! First get the subsymbols.                                ;
    template:-calls.prototuple.setint(1,symbolnumber);
    mycalls:-calls.copyouttuple(template,1);
    mycalls.reset;
    call:-mycalls.prototuple;
    subsymbols:-NEW vector;
    WHILE mycalls.more DO BEGIN
        REF(transform)tr;
        INTEGER snum;
        REF(symbol)sym;
        mycalls.givetuple(call);
        tr:-NEW transform(NONE);
        tr.ti1:=call.getflt(3);
        tr.ti2:=call.getflt(4);
        tr.t2i:=call.getflt(5);
        tr.t22:=call.getflt(6);
        tr.tx:=call.getflt(7);
        tr.ty:=call.getflt(8);
        ! Get the subsymbol number and try to find it in the dictionary;
        snum:=call.getint(2);
        sym:-findindict(snum);
        ! If it is not found get the symbol and put in in;
        IF sym==NONE THEN BEGIN sym:-NEW symbol(snum);
        inst:-NEW instance(sym,tr);
        subsymbols.append(inst QUA thing);
    END;
    mycalls.shatter;

    ! Put default polygons in all the layers;
    FOR i:=1 STEP 1 UNTIL 7 DO layer(i):-NEW polygon;

    ! Get the polygons.    ;
    template:-polys.prototuple.setint(1,symbolnumber);
    mypolys:-polys.copyouttuple(template,1);
```

```
    mypolys.reset;
    poly:-mypolys.prototuple;
    objectnumber:=0;
    WHILE mypolys.more DO BEGIN
        mypolys.givetuple(poly);
        IF objectnumber\=poly.getint(3) THEN BEGIN
            IF objectnumber\=0 THEN
                layer(currentlayer).takevectorassheet(v);
            objectnumber:=poly.getint(3);
            currentlayer:=poly.getint(2);
            v:-NEW vector;
        END;
        v.append(NEW point(poly.getflt(5),poly.getlft(6));
    END;
    IF objectnumber\=0 THEN
        layer(currentlayer).takevectorassheet(v);
    mypolys.shatter;

    ! Get all the wires;
    template:-wires.prototuple.setint(1,symbolnumber);
    mywires:-wires.copyouttuple(template,1);
    mywires.reset;
    wire:-mywires.prototuple;
    objectnumber:=0;
    WHILE mywires.more DO BEGIN
        mywires.givetuple(wire);
        IF objectnumber\=wire.getint(4) THEN BEGIN
            IF objectnumber\=0 THEN
                layer(currentlayer).takevectorassheet
                                    (wireaspoly(v,currentwidth));
            objectnumber:=wire.getint(4);
            currentlayer:=wire.getint(2);
            currentwidth:=wire.getflt(3);
            v:-NEW vector;
        END;
        v.append(NEW point(wire.getflt(6),wire.getlft(7));
    END;
    IF objectnumber\=0 THEN
                layer(currentlayer).takevectorassheet
                                    (wireaspoly(v,currentwidth));
    mywires.shatter;

    ! Get the boxes;
    template:-boxes.prototuple.setint(1,symbolnumber);
    myboxes:-boxes.copyouttuple(template,1);
    myboxes.reset;
    box:-myboxes.prototuple;
    WHILE mywires.more DO BEGIN
        REAL dirx,diry;
        REF(rectangle)rect;
        myboxes.givetuple(box);
        dirx:=box.getflt(4);
```

```
            diry:=box.getflt(5);
            rect:-NEW rectangle(NEW point(box.getflt(7),box.getflt(9)),
                        NEW point(box.getflt(6),box.getflt(8)));
            IF diry\=0 AND dirx>0 THEN
                layer(box.getint(2)).takevectorassheet(rect.asvec)
            ELSE BEGIN
                REF(polygon)temppoly;
                REF(transform)tr;
                temppoly:-NEW polygon;
                temppoly.takevectorassheet(rect.asvec);
                tr:-NEW transform(NONE);
                tr.rotatedby(NEW point(dirx,diry));
                temppoly.xform(tr);
                layer(box.getint(2)).combine(temppoly);
            END;
        END;
        myboxes.shatter;

        ! Get this symbol's bounding box.    ;
        template:-bboxes.prototuple.setint(1,symbolnumber);
        mybboxes:-bboxes.copyouttuple(template,1);
        mybboxes.reset;
        bbox:-mybboxes.prototuple;
        mybboxes.givetuple(bbox);
        bb:-NEW rectangle(NEW point(bbox.getflt(3),bbox.getflt(5)),
                          NEW point(bbox.getflt(2),bbox.getflt(4)));

        ! Check loading of memory to possibly store symbol       ;
        ! if necessary.  Put this symbol int the master list.    ;
        putindict(symbolnumber,THIS symbol);
        possiblepurge;
        empty:=FALSE;
    END of Init Code;

END of CLASS symbol;



REF(symbol) PROCEDURE FindInDict(symbolnumber); INTEGER symbolnumber;
! This precedure looks up the symbol number in the master ;
! dictionary.  If found it returns a pointer to the symbol.;
! If not, it returns NONE.                                 ;
BEGIN
    REF(thing)tempvalue;
    TEXT symtxt;
    REF(string)symstr;

    ! Generate a string from the symbol number and look it up;
    ! in the dictionary.;
    symtxt:-blanks(11);
    symtxt.putint(symbolnumber);
    symstr:-NEW string(symtxt);
```

```
    tempvalue:-allsymbols.lookup(symstr);
    findindict:- IF tempval==NONE THEN NONE ELSE tempvalue QUA symbol;
END of FindInDict;


PROCEDURE PutInDict(symbolnumber,sym); INTEGER symbolnumber;
REF(symbol)sym;
! This procedure enters a symbol number and its symbol into the list.;
BEGIN
    TEXT symtxt;
    REF(string)symstr;

    symtxt:-blanks(11);
    symtxt.putint(symbolnumber);
    symstr:-NEW string(symtxt);
    allsymbols.insert(symstr,sym QUA thing);

END of PutInDict;



BOOLEAN PROCEDURE PossiblyPurge;
! This procedure examines the number of symbols in memory  ;
! and in use.  It may purge symbols if necessary.          ;
! If purged then TRUE will be returned otherwise FALSE.    ;
BEGIN
    INTEGER length,possibles,i;
    REF(thing)sym;

    length:=allsymbols.length;
    FOR i:=1 STEP 1 UNTIL length DO BEGIN
        sym:-allsymbols.val(i);
        IF sym=/=NONE THEN BEGIN
            IF NOT sym QUA symbol.empty THEN possibles:=possibles+1;
        END;
    END;
    IF possibles>25 THEN BEGIN
        WHILE NOT purged DO BEGIN
            sym:-allsymbols.val(random(.51,length+.49));
            IF sym=/=NONE THEN BEGIN
                IF NOT sym QUA symbol.empty AND
                    NOT sym QUA symbol.inuse THEN BEGIN
                    sym QUA symbol.store;
                    purged:=TRUE;
                END;
            END;
        END;
        possiblypurge:=purged;
    END;
END of PossiblyPurge;
```

```
!Init code for SYM;
! set array index constants;
diffusion:=1;   spacing(diffusion):=3;  width(diffusion):=2;
poly:=2;        spacing(poly):=2;       width(poly):=2;
metal:=3;       spacing(metal):=3;      width(metal):=3;
cuts:=4;                                width(cuts):=2;
implant =5;     spacing(implant):=1.5;
burcon:=6;
glass:=7;
allsymbols:-NEW dictionary;

!get and open the database file;
outtext("Enter Database File Name: "); breakoutimage;
inimage;
outtext(IF databasefile(sysin.image.strip) THEN "[ New" ELSE "[ Existing");
outtext(" Database File ]"); outimage;

! Get all the CIF relations;
calls:-rln("CIFsymbolcalls");
polys:-rln("CIFpolygons");
boxes:-rln("CIFboxes");
wires:-rln("CIFwires");
bboxes:-rln("CIFsymbolboundingboxes");

! Test for all relations needed present;
IF calls==NONE OR polys==NONE OR boxes==NONE OR
    wires==NONE OR bboxes==NONE THEN BEGIN
    outimage;
    outtext(" ERROR: Cannot find CIF relations in database file.");
    outimage;
END ELSE

    ! Make a temporary relation in which to store whole symbols.;
    savepolys:-field("symbolnumber").field("polygonnumber")
           .field("sheetnumber").field("itemnumber").field("itemtype")
           .field("x-or-a").field("y-or-b").field("c-or-r");

END of CLASS sym;
```

Appendix III

Simula Program for Computing Wiring Statistics

```
BEGIN
!*********************************************************;
!                                                         ;
!   This is WIRES, a program to extract wiring data       ;
!      from mask geometry heuristicly.  Wires uses        ;
!      Things,Views,Displa,Plygns,Plyrdp, and Sym.        ;
!                                                         ;
!                                 Dick Lang 6/79          ;
!*********************************************************;

EXTERNAL CLASS Things,Displa,Views,Plygns,Plyrdp,Sym;

!Things, Displa and Views externals (these not used by RDP);
EXTERNAL INTEGER PROCEDURE hash,imax,imin;
EXTERNAL REAL PROCEDURE rmin,rmax;
EXTERNAL TEXT PROCEDURE upcase,getitem,frontstrip,conc,initem;
EXTERNAL CHARACTER PROCEDURE getch;
EXTERNAL PROCEDURE enterdebug,sleep,qwkout;

!RDP externals;
EXTERNAL         PROCEDURE tinp, tinsertsort, tmove, tpmove, tout,
                           txtinp, txtout;
EXTERNAL BOOLEAN PROCEDURE jsys, teq, tlt, tpeq;
EXTERNAL INTEGER PROCEDURE aaddr, getf, ioaddress, land, left, lnot, lor,
                           lshift, lxor, realasbits, right, setf, taddr,
                           tpartition, tpxor, tsize, xwd;
EXTERNAL  REAL   PROCEDURE bitsasreal;

! Sym externals;
REAL PROCEDURE random;

!*****************************************************************;

   Sym BEGIN


      !*************************************************;
      !                                                 ;
      !    The procedures GetDevices, Hulls, Annulus,   ;
      !      and SeparateWires (in the text of the      ;
      !      the thesis) would be placed here;          ;
      !                                                 ;
      !*************************************************;
```

```
PROCEDURE AnalyzeWires(symbolnumber);
! This is a recursive procedure which will traverse the;
! hierarchy of symbols and perform the heuristic       ;
! analysis on each of them.          ;
BEGIN
    REF(symbol)mysymbol;
    INTEGER length,i,j;

    mysymbol:-findindict(symbolnumber);
    IF mysymbol==NONE THEN BEGIN
        outtext(" ERROR: Cannot find symbol number ");
        outint(symbolnumber,11);
        outimage;
    END
    ELSE BEGIN
        ! Make sure the symbol is not ejected from memory;
        ! while it is being worked on.                    ;
        IF mysymbol.empty THEN mysymbol.retrieve;
        mysymbol.inuse:=TRUE;

        ! This is where all the work is done;
        getdevices(mysymbol);
        hulls(mysymbol);           ! hulls will recursively find ;
        annulus(mysymbol);         ! all the subhulls.           ;
        xcontacts(mysymbol);
        separatewires(mysymbol);

        ! Now compute all the necessary numbers;
        INSPECT mysymbol BEGIN
            REAL totalarea,devarea,ndevarea,ddevarea,unusedarea,
                 subsymarea,interarea,localarea,globalarea;
            INTEGER npins,ndev,nndev,nddev;

            PROCEDURE outperarea(r,t); VALUE t; TEXT t; REAL r;
            ! Print out r and r as a percentage of total area;
            BEGIN
                outtext(blanks(10));
                outtext(t);
                outfix(r,4,10);
                outtext(blanks(10));
                outfix(r/totalarea,4,10);
                outimage;
            END of outperarea;

            ! Get device areas and devices counts;
            ddevarea:=ddevices.area;
            ndevarea:=ndevices.area;
            devarea:=ddevarea+ndevarea;
            nddev:=ddevices.numsheets;
            nndev:=ndevices.numsheets;
            ndev:=nddev+nndev;
```

```
! Get the subsymbol area;
length:=susbsymbols.length;
FOR i:=1 STEP 1 UNTIL length DO BEGIN
    REF(symbol)subsym;
    subsym:-subsymbols.val(i) QUA instance.sym;
    IF subsym.empty THEN subsym.retrieve;
    FOR j:=1 STEP 1 UNTIL 3 DO
        subsymarea:=subsymarea+subsym.hull(j).area;
END;

! Get the other stuff;
FOR i:=1 STEP 1 UNTIL 3 DO BEGIN
    interarea:=interarea+layer(i)
               .copy.inflate(spacing(i)/2);
    localarea:=localarea+localwire(i)
              .copy.inflate(spacing(i)/2);
    globalarea:=globalarea+globalwire(i)
               .copy.inflate(spacing(i)/2);
    npins:=npins+xcontacts.numsheets(i);
    totalarea:=totalarea+hull(i).copy
               .inflate(spacing(i)/2).area;
END;

unusedarea:=totalarea-devarea-interarea-subsymarea;
mysymbol.inuse:=FALSE;
possiblypurge;

! Print out all the numbers;
outimage; outimage;
outtext(" Symbol Number "); outint(symbolnumber,11);
outimage;
outtext(" Total area: "); outfix(totalarea,2,10);
outimage;
outperarea(devarea," Total Device Area ");
outperarea(ddevarea," Depletion Device Area ");
outperarea(ndevarea," Enhancement Device Area ");
outperarea(subsymarea," Subsymbol Area ");
outperarea(interarea," Total Interconnection Area");
outperarea(localarea," Local Wire Area ");
outperarea(globalarea," Global Wire Area ");
outperarea(unusedarea," Unused Area ");
outtext(" Total Number of Devices: ");
outint(ndev,10);
outimage;
outtext(" Depletion Devices: ");
outint(nddev,10);
outimage;
outtext(" Enhancement Devices: ");
outint(nndev,10);
outimage;
```

```
            END;


            ! Now recurse and analyze all the subsymbols;
            length:=mysymbol.subsymbols.length;
            FOR i:=1 STEP 1 UNTIL length DO BEGIN
                REF(symbol)subsym;
                subsym:-mysymbol.subsymbols.val(i) QUA instance.sym;
                analyzewires(subsym.symbolnumber);
            END;
        END;
    END of AnalyzeWires;


    ! A short main program -- ;
    NEW symbol(0);
    analyzewires(0);      ! Note that the root symbol is always;
                          ! numbered zero.                     ;

END of Sym Block;

END of PROGRAM Wires;
```

Appendix IV

CIFX, A Simula Program for Building Database Files From CIF

```
BEGIN
!**************************************************************************
!
!
!    CIFX     - reads and parses a CIF 2.0 file and
!               generates five database relations.
!
!
!       The database relations are as follows:
!
!             "CIFsymbolcalls" - symbol,subsymbol,t11,t12,t21,t22,tx,ty
!                                (integer,integer,real,real,real,real,real,real)
!
!             "CIFwires" - symbol,layernumber,width,objectnumber,
!                                pointnumber,point.x,point.y  (integer,integer,
!                                real,integer,integer,real,real)
!
!             "CIFboxes" - symbol,layernumber,objectnumber,direction.x,
!                                direction.y,ur.x,ll.x,ur.y,ll.y
!                                (integer,integer,real,real,real,real,real,real)
!
!             "CIFpolygons" - symbol,layernumber,objectnumber,pointnumber,
!                                point.x,point.y
!                                (integer,integer,integer,integer,real,real)
!
!             "CIFsymbolboundingboxes" - symbol,ur.x,ll.x,ur.y,ll.y
!                                (integer,real,real,real,real)
!
!             "CIFgeometricboundingboxes" - symbol,objectnumber,ur.x,
!                                ll.x,ur.y,ll.y  (integer,integer,real,real,real,
!                                real)
!
!               All roundflashes are converted to octagons (with
!               the average of the inside and outside diameters set
!               to equal the given diameter value).  Optionally, the
!               program will convert all wires and boxes to polygons,
!               in which case no tuples will appear in "CIFwires" or
!               "CIFboxes".  Optionally, the program will produce
!               the minimum bounding boxes of all geometric items
!               (polygons,boxes,wires,roundflashes) and puts these in
!               relation "CIFgeometricboundingboxes". The program puts the
!               minimum bounding boxes of all symbols in
!               "CIFsymbolboundingboxes".
!
```

```
!             symbol numbers (and subsymbol numbers)
!             --------------
!             are generated using a deletion count to provide
!             for unique symbol identifiers and deletion commands.
!             The symbol numbers stored in the database file are
!             (defined symbol number)X1000+(number of previous
!             deletions).
!             The global symbol of the CIF file is assigned zero
!             as a symbol number, this encloses any global calls.
!             This insures that it will always be the first item
!             of the relations.
!
!             layernumbers
!             ------------
!             used in "CIFpolygons", "CIFboxes" and "CIFwires"
!             are determined as follows:
!
!                    ND   1         NI   5
!                    NP   2         NB   6
!                    NC   3         NG   7
!                    NM   4
!
!
!             objectnumbers
!             -------------
!             are integers assigned to geometric items (polygons,
!             wires,boxes,roundflashes) sequentially as they are
!             encountered in the CIF file. The sequences begin with
!             one and are restarted for each symbol definition.
!
!             pointnumbers
!             ------------
!             are used in the "CIFpolygons" and "CIFwires" relations.
!             Beginning with 1, each point in the path of a polygon or
!             wire is sequentially assigned a point number to maintain
!             the ordering of points in the relation. The first point
!             of every polygon or wire is assigned a 1.
!
!             Hopefully, the other field names of the relations are
!             self evident.
!
!             Dick Lang
!             Caltech          Spring 1979
!
!*********************************************************************;


EXTERNAL CLASS things,displa,views,cif20,cifrdp;

EXTERNAL PROCEDURE tinp,tmove,tout,txtinp,txtout,enterdebug,sleep;
EXTERNAL PROCEDURE tinsertsort,tpmove,qwkout;
```

```
EXTERNAL BOOLEAN PROCEDURE jsys,teq,tlt,skipin,tpeq;
EXTERNAL INTEGER PROCEDURE aaddr,land,left,lnot,lor,lshift;
EXTERNAL INTEGER PROCEDURE realasbits,right,taddr,tsize,xwd,hash,imin;
EXTERNAL INTEGER PROCEDURE imax,lxor,save,getf,setf;
EXTERNAL INTEGER PROCEDURE ioaddress,tpartition,tpxor;
EXTERNAL REAL PROCEDURE bitsasreal,rmin,rmax;
EXTERNAL TEXT PROCEDURE upcase,frontstrip,conc,gatitem,initem,today;
EXTERNAL TEXT PROCEDURE daytime,scanto,rest;
EXTERNAL CHARACTER PROCEDURE getch;


cifrdp BEGIN

    !*****************   global stuff  ***********************************;

    REF(cif)cif2;
    REF(relation)usergeom,usersubc,boxes,wires,ubxes,sboxes,fboxes;
    REF(dictionary)symtab;
    INTEGER cursymnum,symult,errors,olderrs;
    REF(symval)currentval,globalval;
    TEXT cfile,efile,dfile;
    BOOLEAN ok,super,convertpoly,fwddef,fwdrefok,geombb,globalitems;
    REF(rlntuple)subc,apoly,awire,abox;
    REF(vector)symbolsinuse;



    thing CLASS symval(symnum); INTEGER symnum;

    !this object is used as dictionary entries for each symbol;
    !its attributes contain items relating to the deletions;
    !and definitions of the symbol, the bounding box of the symbol;
    !the layer the symbol is using, and the number of geometric;
    !objects;
    BEGIN
        INTEGER def,del,sym,pgnum,layer;
        REF(rectangle)bb;
        REF(point)p1,p2;

        BOOLEAN PROCEDURE defined;        !to test for the symbol;
        defined:=def=del;                 !defined, del must equal def;

        PROCEDURE delete;
        BEGIN
            !to delete a symbol number, increment del (del will;
            !no longer equal def) and test for too large;
            del:=del+1;
            IF del>=symult THEN
            cifwarn("Deletion count overflow.");
        END;
```

```
    PROCEDURE define;
    BEGIN
        !to define a symbol number, set def equal to del and;
        !reset the layer, object count and bounding box;
        def:=del;
        pgnum:=1;
        layer:=0;
        p1:-NEW point(1,1);
        p2:-NEW point(0,0);
        bb:-NEW rectangle(p1,p2);
        p1.x:=-10.0**20;
        p1.y:=-10.0**20;
        p2.x:=10.0**20;
        p2.y:=10.0**20;
    END;

    !when creating a dictionary entry clear all attributes;
    !and set the symbol number;
    def:=del:=layer:=0;
    pgnum:=1;
    sym:=symnum;
    p1:-NEW point(1,1);
    p2:-NEW point(0,0);
    bb:-NEW rectangle(p1,p2);
    p1.x:=-10.0**20;
    p1.y:=-10.0**20;
    p2.x:=10.0**20;
    p2.y:=10.0**20;
END of symval;



PROCEDURE includeinmbb(path); REF(vector)path;
BEGIN
    INTEGER l,i;  REF(rectangle)bb;
    REF(rlntuple)boxtuple;

    !this procedure takes a path (polygon) and;
    !expands the minimum bounding box of the current;
    !symbol (if necessary) to include all the points of the path;
    l:=path.length;
    bb:-NEW rectangle(path.val(1) QUA point,path.val(2) QUA point);
    IF l>2 THEN FOR i:=3 STEP 1 UNTIL l DO
    bb:-bb.including(path.val(i) QUA point);

    currentval.bb:-currentval.bb.including(bb.ur).including(bb.ll);

    IF geombb THEN BEGIN
        boxtuple:-boxes.prototuple;
        boxtuple.setint(1,cursymnum).setint(2,currentval.pgnum)
        .setflt(3,bb.ur.x).setflt(4,bb.ll.x).setflt(5,bb.ur.y)
```

```
        .setflt(6,bb.ll.y);
        boxes.taketuple(boxtuple);
    END;

END of includeinmbb;



thing CLASS inthing(int); INTEGER int; BEGIN END;
!make an integer thing so integers can be put in vectors;



REF(rectangle) PROCEDURE verifyandfixmbb(symbolnum);
INTEGER symbolnum;
!recursive routine to follow symbol calls and at the;
!same time update symbol bounding boxes to their true size;
BEGIN
    REF(relation)calls,box,sbox2;
    INTEGER i,l;



    PROCEDURE printchain;
    BEGIN
        INTEGER i;

        IF l>0 THEN BEGIN
            INSPECT cif2.cfile DO BEGIN

                !print out the chain of symbol calls preceding this one;
                FOR i:=i STEP 1 UNTIL l DO BEGIN
                    outtext(blanks(20));
                    outtext("Symbol ");
                    outint(symbolsinuse.val(i) QUA inthing.int,11);
                    outtext(" calls ..."); outimage;
                END;
            END;
        END;
    END;



    !set box to zero;
    verifyandfixmbb:-NEW rectangle(NEW point(0,0),NEW point(0,0));

    !check to see if this symbol is currently in use;
    ok:=true;
    l:=symbolsinuse.length;
    IF length=0 THEN ok:=true
```

```
    ELSE
    FOR i:=1 STEP 1 UNTIL 1 DO
    ok:=ok AND (symbolnum\=symbolsinuse.val(i) QUA inthing.int);

    !if symbol is currently in use, issue error and quit;
    IF NOT ok THEN BEGIN
        INSPECT cif2.efile DO BEGIN
            outimage;
            outtext(" Recursive symbol call ... "); outimage;
            printchain;
            outtext(blanks(20)); outtext("Symbol ");
            outint(symbolnum,11); outtext(" (Duplicate call above.)");
            outimage;
            errors:=errors+1;
        END;
    END

    ELSE BEGIN
        REF(rlntuple)calltuple,boxtuple,cboxtuple;
        REF(rectangle)bb;
        BOOLEAN visited,defined;

        !if no recursion, get the relations associated with;
        !symbol;
        cboxtuple:-sboxes.prototuple;
        cboxtuple.setint(1,symbolnum);
        sbox2:-sboxes.copyouttuple(cboxtuple,1);
        sbox2.reset;

        IF sbox2.more THEN BEGIN
            visited:=defined:=true;
            sbox2.givetuple(cboxtuple);
            bb:-NEW rectangle(NEW point(cboxtuple.getflt(3),
                                        cboxtuple.getflt(5)),
                              NEW point(cboxtuple.getflt(2),
                                        cboxtuple.getflt(4)));
        END
        ELSE BEGIN
            boxtuple:-fboxes.prototuple;
            boxtuple.setint(1,symbolnum);
            box:-fboxes.extracttuple(boxtuple,1);
            visited:=false;
            box.reset;
            IF box.more THEN BEGIN
                defined:=true;

                !put the old box in as the new box;
                box.givetuple(boxtuple);
                bb:-NEW rectangle(NEW point(boxtuple.getflt(3),
                                            boxtuple.getflt(5)),
                                  NEW point(boxtuple.getflt(2),
                                            boxtuple.getflt(4)));
```

```
    END
    ELSE defined:=false;
    box.shatter;
END;
sbox2.shatter;

IF defined AND NOT visited THEN BEGIN

    !put this symbol into the used list;
    symbolsinuse.append(NEW inthing(symbolnum));

    calltuple:-usersubc.prototuple;
    calltuple.setint(1,symbolnum);
    calls:-usersubc.copyouttuple(calltuple,1);
    calls.reset;

    !determine recursively, the true size of the box;
    !and whether or not any lower recursion exists;
    WHILE calls.more DO BEGIN
        REF(vector)boxaspoly;

        !get one of the calls and its true bounding box;
        calls.givetuple(calltuple);
        boxaspoly:-verifyandfixbox(calltuple.getint(2)).asvec;

        !step through each point of the box and be sure;
        !its included in the symbols bounding box;
        FOR i:=1 STEP 1 UNTIL 4 DO BEGIN
            REAL newx,newy,x,y;

            !transform one point of the called symbols box;
            newx:=boxaspoly.val(i) QUA point.x;
            newy:=boxaspoly.val(i) QUA point.y;
            x:=calltuple.getflt(3)*newx+calltuple.getflt(5)
                  *newy+calltuple.getflt(7);
            y:=calltuple.getflt(4)*newx+calltuple.getflt(6)
                  *newy+calltuple.getflt(8);

            !compare new points with old box, update box;
            !if points are outside;
            bb:-bb.including(NEW point(x,y));
        END;

    END;
    calls.shatter;

    !take this symbol out of the in use list;
    symbolsinuse.length:=symbolsinuse.length-1;

    !update sboxes relation to use new bounding box;
    cboxtuple.setint(1,symbolnum).setflt(2,bb.ur.x)
```

```
            .setflt(3,bb.ll.x).setflt(4,bb.ur.y)
            .setflt(5,bb.ll.y);
            sboxes.taketuple(cboxtuple);
            sboxes.reset;

            verifyandfixmbb:-bb;
        END;

        IF NOT defined THEN BEGIN
            INSPECT cif2.efile DO BEGIN
                !if there is no box for the symbol, then it;
                !must be undefined -- issue an error message;
                outimage; outtext(" Undefined symbol call ... ");
                outimage;
                printchain;
                outtext(blanks(20)); outtext("Symbol ");
                outint(symbolnum,ii); outtext(" (Undefined)");
                outimage;
                errors:=errors+1;
            END;
        END;
    END;
END of verifyandfixmbb;



REF(symval) PROCEDURE findindict(n); INTEGER n;
!this procedure looks up in in the symbol number;
!dictionary and returns the associated symval (or none);
!if symbol number is not found;
BEGIN
    REF(thing)tempval;
    TEXT sym;
    REF(string)symst;

    !generate a string from the symbol number and look;
    !it up in the dictionary;
    sym:-blanks(ii);
    sym.putint(n);
    symst:-NEW string(sym);
    tempval:-symtab.lookup(symst);
    findindict:- IF tempval==NONE THEN NONE ELSE tempval QUA symval;

END of findindict;



PROCEDURE putindict(n,val); INTEGER n;  REF(symval)val;
 !this procedure makes a new dictionary entry;
 BEGIN
     TEXT tzero;
```

```
     REF(string)tstring;

    tzero:-blanks(ii);
    tzero.putint(n);
    tstring:-NEW string(tzero);
    symtab.insert(tstring,val);

END of putindict;




!*****************  procedures given to the CIF parser  ***************;



BOOLEAN PROCEDURE defsym(symnum,a,b); INTEGER symnum,a,b;
!process DS commands;
BEGIN
    REF(symval)val;

    !test for an illegal symbol number and take appropriate action;
    IF symnum<=0 THEN BEGIN
        IF symnum=0 THEN ciferror("Zero symbol number not permitted.")
            ELSE BEGIN
            cifwarn(
        "Negative symbol number not permitted, absolute value used.");
            symnum:=-symnum;
            END;
    END;

    !test to be sure the symbol definition is not within another;
    IF cursymnum=0 OR fwddef THEN BEGIN

        val:-findindict(symnum);

        !if there is no such symbol number in the dictionary;
        !then make a new entry containing it;
        IF val==NONE THEN BEGIN
            val:-NEW symval(symnum);
            putindict(symnum,val);
        END

        ELSE BEGIN

            !if there is an entry found, test to see if the;
            !the number is currently defined, if so and forward ;
            !references are not allowed, then issue a warning ;
            !and delete it;
            IF val.defined AND NOT fwdrefok THEN BEGIN
```

```
            cifwarn("Previous symbol definition overwritten.");
            val.delete;
        END;

        !then define the symbol number;
        val.define;
    END;

    IF NOT fwddef THEN BEGIN

        !set current dictionary pointer to the entry and;
        !set the current symbol number to the defined symbol;
        !and update the last definiton number to the new number;
        currentval:-val;
        cursymnum:=val.sym$symultval.del;
    END;
END
ELSE
!if a definition is currently in progress issue error ;
ciferror("Nested symbol definition -- Command ignored.");
defsym:=FALSE;
END of defsym;



BOOLEAN PROCEDURE deffin;
!process DF command;
BEGIN
    REF(rlntuple)nbb;

    !test for symbol definition in progress or supervisory call;
    IF cursymnum=0 AND NOT super THEN

    !if no definition in progress issue warning;
    cifwarn("Excess DF command -- Command ignored.")

    ELSE BEGIN
        !if there is, put the current symbol bounding box;
        !in the boxes relation.  at this point the bounding;
        !box takes into account only the polygons (if any);
        !defined in the symbol and not the subsymbols.;
        nbb:-fboxes.prototuple;
        nbb.setint(1,cursymnum).setflt(2,currentval.bb.ur.x)
        .setflt(3,currentval.bb.ll.x).setflt(4,currentval.bb.ur.y)
        .setflt(5,currentval.bb.ll.y);
        fboxes.taketuple(nbb);                !put box into relation;

        !reset dictionary and current symbol to global;
        !environment;
        cursymnum:=0;
        currentval:-globalval;
```

```
        deffin:=FALSE;
    END;
END of deffin;



BOOLEAN PROCEDURE poly(path); REF(vector)path;
!process P command;
BEGIN
    INTEGER i;
    REAL xmax,xmin,ymax,ymin,x,y;
    REF(rintuple)geom;
    geom:-usergeom.prototuple;
    xmax:=xmin:=ymax:=ymin:=0;

    !test for degeneracy or layer undefined;
    IF path.length (= 2 THEN
    cifwarn("Degenerate polygon -- Command ignored.")
    ELSE
    IF currentval.layer=0 THEN
    ciferror("Layer not defined -- Command ignored.")
    ELSE BEGIN

        !if no problem, put polygon points into geometry relation;
        INTEGER i,length;
        length:=path.length;
        geom.setint(1,cursymnum).setint(2,currentval.layer)
        .setint(3,currentval.pgnum);
        FOR i:=1 STEP 1 UNTIL length DO BEGIN

            !loop to process each point in the path;
            x:=path.val(i) QUA point.x;
            y:=path.val(i) QUA point.y;
            geom.setint(4,i).setflt(5,x).setflt(6,y);
            usergeom.taketuple(geom);          !put it in the relation;

        END;

        includeinmbb(path);
        currentval.pgnum:=currentval.pgnum+1;
        IF cursymnum=0 THEN globalitems:=true;
    END;

    poly:=FALSE;
END of poly;



BOOLEAN PROCEDURE box(rect,dir); REF(rectangle)rect; REF(point)dir;
!process B command;
```

```
BEGIN
    REF(vector)path;
    REF(point)pt1,pt2,pt3,pt4;
    REAL cos,sin,hyp,xc,yc,x,y;

    !the box is to be converted to a polygon;
    !first generate a vector of four points;
    path:-NEW vector;
    pt1:-NEW point(0,0);
    pt2:-NEW point(0,0);
    pt3:-NEW point(0,0);
    pt4:-NEW point(0,0);

    !compute length of direction vector;
    !test for zero length;
    hyp:=sqrt(dir.x*dir.x+dir.y*dir.y);
    IF hyp=0 THEN BEGIN

        !if zero, issue warning and assume x=1, y=0;
        cifwarn(
        "Box direction vector has zero length -- (1,0) assumed.");
        cos:=1;  sin:=0;
    END
    ELSE BEGIN

        !otherwise compute the sin and cos of the direction angle;
        cos:=dir.x/hyp;
        sin:=dir.y/hyp;
    END;

    !compute the center of the box and de-translate it;
    ! x and y become the coordinates of the upper right;
    !as if the center of the box were at 0,0;
    xc:=(rect.ll.x+rect.ur.x)/2;
    yc:=(rect.ll.y+rect.ur.y)/2;
    x:=rect.ur.x-xc;
    y:=rect.ur.y-yc;

    !compute the coordinates of the four corners of the;
    !rotated box and translate them to the old box center;
    pt1.x:=x*cos-y*sin+xc;
    pt2.x:=x*cos+y*sin+xc;
    pt3.x:=-x*cos+y*sin+xc;
    pt4.x:=-x*cos-y*sin+xc;
    pt1.y:=x*sin+y*cos+yc;
    pt2.y:=x*sin-y*cos+yc;
    pt3.y:=-x*sin-y*cos+yc;
    pt4.y:=-x*sin+y*cos+yc;

    !stick the points into the path vector (clockwise);
    path.append(pt1);
    path.append(pt2);
```

```
    path.append(pt3);
    path.append(pt4);

    !test to see if the box should be stored as a polygon;
    IF converttopoly THEN poly(path)
    ELSE BEGIN
        REF(rlntuple)ubox;

        !make a box tuple and put it in the userbox relation;
        ubox:-uboxes.prototuple;
        ubox.setint(1,cursymnum).setint(2,currentval.layer)
        .setint(3,currentval.pgnum).setflt(4,dir.x).setflt(5,dir.y);
        ubox.setflt(6,rect.ur.x).setflt(7,rect.ll.x)
        .setflt(8,rect.ur.y).setflt(9,rect.ll.y);
        includeinmbb(path);
        currentval.pgnum:=currentval.pgnum+1;
        uboxes.taketuple(ubox);
        IF cursymnum=0 THEN globalitems:=true;
    END;
    box:=FALSE;
END of box;



BOOLEAN PROCEDURE rflash(dia,center); INTEGER dia; REF(point)center;
!process the R command;
BEGIN
    REF(vector)path;
    REAL v1,v2;


    PROCEDURE nextpt(x,y);  REAL x,y;
    BEGIN
        REF(point)pt;

        !a useful procedure which makes a new point,translates;
        !it and sticks it into a path vector;
        pt:-NEW point(0,0);
        pt.x:=x+center.x;
        pt.y:=y+center.y;
        path.append(pt);
    END of nextpt;



    !we are going to make an octagon with average diameter dia;
    !first check for a legal diameter;
    IF dia>0 THEN BEGIN

        !make a new path vector and determine the two possible;
        !coordinates of the octagon;
```

```
        path:-NEW vector;
        v1:=dia*0.480216935;
        v2:=dia*0.198912367;

        !set the points of the octagon inot the path in a;
        !clockwise manner and put the polygon into the relations;
        nextpt(-v2,v1);
        nextpt(-v1,v2);
        nextpt(v1,v2);
        nextpt(v2,v1);
        nextpt(v2,-v1);
        nextpt(v1,-v2);
        nextpt(-v1,-v2);
        nextpt(-v2,-v1);
        poly(path);
    END
    ELSE
    !if illegal diameter issue warning and go on;
    cifwarn("Illegal diameter -- Command ignored.");

    rflash:=FALSE;
END of rflash;



BOOLEAN PROCEDURE wire(path,width); REF(vector)path; INTEGER width;
!process W command;
BEGIN

    !if width is legal, enter wire into relations as a;
    !polygon using the "wireaspoly" procedure of VIEWS;
    IF width>0 THEN BEGIN
        IF converttopoly THEN poly(wireaspoly(path,width))
        ELSE BEGIN
            REF(rlntuple)wiretuple;
            INTEGER i,l;

            !make wire entries in the wire relation;
            wiretuple:-wires.prototuple;
            wiretuple.setint(1,cursymnum).setint(2,currentval.layer)
            .setflt(3,width).setint(4,currentval.pgnum);
            includeinmbb(wireaspoly(path,width));
            currentval.pgnum:=currentval.pgnum+1;
            l:=path.length;
            FOR i:=1 STEP 1 UNTIL l DO BEGIN
                wiretuple.setint(5,i);
                wiretuple.setflt(6,path.val(i) QUA point.x);
                wiretuple.setflt(7,path.val(i) QUA point.y);
                wires.taketuple(wiretuple);
            END;
            IF cursymnum=0 THEN globalitems:=true;
```

```
        END;
    END
    ELSE

    cifwarn("Illegal wire width -- Command ignored.");
    wire:=FALSE;
END of wire;

BOOLEAN PROCEDURE layer(t); TEXT t;
!process L command;
BEGIN
    INTEGER lll;
    lll:=0;

    !test for known layer;
    IF t="ND  " THEN lll:=1 ELSE
    IF t="NP  " THEN lll:=2 ELSE
    IF t="NC  " THEN lll:=3 ELSE
    IF t="NM  " THEN lll:=4 ELSE
    IF t="NI  " THEN lll:=5 ELSE
    IF t="NB  " THEN lll:=6 ELSE
    IF t="NG  " THEN lll:=7 ELSE
    ciferror("Unknown layer -- Command ignored.");

    !if layer known change to new layer;
    IF lll\=0 THEN currentval.layer:=lll;
    layer:=FALSE;
END of layer;



BOOLEAN PROCEDURE delete(symnum); INTEGER symnum;
!process DD command;
BEGIN


    PROCEDURE delsym(val); REF(thing)val;
    !this routine tests for a given symbol dictionary entry;
    !greater than an integer and , if true, deletes it;
    IF val QUA symval.sym )= symnum
    THEN val QUA symval.delete;


    !test for legal DD operand;
    IF symnum)0 THEN BEGIN

        !test for DD within a symbol definition;
        IF cursymnum\=0 THEN

        !if true, issue error;
        ciferror
```

```
        ("DD command not permitted within DS... DF -- Command ignored.")

        ELSE BEGIN  !if not, apply delsym to all dictionary entries;
        syntab.apply(delsym);
        END;
    END
    ELSE
    !if illegal issue error and go on;
    ciferror("Illegal number in DD command -- Command ignored.");
    delete:=FALSE;
END of delete;



BOOLEAN PROCEDURE symcall(symnum,tr); INTEGER symnum; REF(transform)tr;
!process C command;
BEGIN
    REF(symval)val;
    BOOLEAN undef;

    val:-findindict(symnum);

    !test for symbol defined in the dictionary;
    !this test requires all symbols to be defined textually;
    !before they are called;
    IF val==NONE THEN undef:=TRUE ELSE
    undef:=NOT val.defined;
    IF (undef AND fwdrefok) OR NOT undef THEN BEGIN
         REF(rlntuple)subc;

        !if symbol is not defined, define it. if it is never;
        !defined, then the error is caught in verifyandfixmbb;
        IF undef THEN BEGIN
            fwddef:=true;
            defsym(symnum,0,0);
            val:-findindict(symnum);
            fwddef:=false;
        END;

        !if defined, test for symbol directly calling itself;
        IF symnum\=(cursymnum//symult) THEN BEGIN

            !if it is not, generate a tuple for the call;
            !and enter it in the usersubsymbols relation;
            subc:-usersubc.prototuple;
            subc.setint(1,cursymnum).setint(2,val.sym%symult+val.del)
            .setflt(3,tr.ti1).setflt(4,tr.ti2).setflt(5,tr.t2i)
            .setflt(6,tr.t22).setflt(7,tr.tx).setflt(8,tr.ty);
            usersubc.taketuple(subc);                    !enter tuple;
        END
        ELSE
```

```
            !if self-call issue error and go on;
            ciferror("Recursive symbol call -- Command ignored.");
    END
    ELSE ciferror("Call to undefined symbol -- Command ignored.");

    symcall:=FALSE;
END of symcall;



BOOLEAN PROCEDURE endcom;
!process E command;
BEGIN

    !test for E command within symbol definition;
    IF cursymnum\=0 THEN BEGIN

        !if it is issue a warning and do a DF anyway;
        cifwarn(
        "End command encountered, DF expected -- DF supplied.");
        deffin;
    END;

    !set procedure value to true to stop parser;
    !note that anything following the E command will;
    !not be read at all;
    endcom:=TRUE;
END of endcom;



BOOLEAN PROCEDURE userext(num,txt); INTEGER num; TEXT txt;
!process userextensions;
BEGIN

    !issue warning and do nothing;
    cifwarn("User Extensions not supported -- Ignored.");
END of userext;



BOOLEAN PROCEDURE ciferror(msg); VALUE msg; TEXT msg;
!process CIF errors;
BEGIN

    !call routine supplied by CIF parser;
    cif2.defaultciferror(msg);
END of ciferror;
```

```
BOOLEAN PROCEDURE cifwarn(msg); VALUE msg; TEXT msg;
!process CIF warnings;
BEGIN

    !call routine suplied by CIF parser;
    cif2.defaultcifwarning(msg);
END of cifwarn;



BOOLEAN PROCEDURE cifdone(lines,warns,errs); INTEGER lines,errs,warns;
!end of parser routine;
BEGIN

!print out the number of warning, error messages and lines read;
outint(warns,10);  outtext(" Warning messages issued."); outimage;
outint(errs,10);   outtext(" Error messages issued.");   outimage;
outint(lines,10);  outtext(" Lines of CIF 2.0 read.");   outimage;
errors:=errs;    olderrs:=errs;
END of cifdone;



!************************  main program  ************************;



!initialize, set current symbol number to the global symbol 0;
!make a cif2 (to access the parser) and set the symbol multiplier;
cursymnum:=0;
cif2:-NEW cif;
symult:=1000;

!setup semantic option flags;
fwdrefok:=true;            !allows forward symbol references;

!get all the necessary file names;
ok:=false;
cfile:-blanks(sysin.image.length);
efile:-blanks(sysin.image.length);
dfile:-blanks(sysin.image.length);
WHILE NOT ok DO BEGIN
    outimage; outimage;
    outtext("CIFX - CIF 2.0 to Database Translation (DL 3/24/79)");
    outimage; outimage;
    outtext("CIF 2.0 Filename? "); breakoutimage; inimage;
    cfile:=sysin.image;
    outtext("CIF Error Filename? "); breakoutimage; inimage;
```

```
    efile:=sysin.image;
    outtext("Database Filename? "); breakoutimage; inimage;
    dfile:=sysin.image;
    outimage; outimage;
    outtext(conc("CIF 2.0 - ",cfile.strip));      outimage;
    outtext(conc("CIF Errors - ",efile.strip));   outimage;
    outtext(conc("Database - ",dfile.strip));     outimage;

    outimage;
    outtext("Filenames OK?  (Confirm with y(cr)): "); breakoutimage;
    inimage;
    IF sysin.image.more THEN BEGIN
        IF sysin.image.getchar='y' THEN ok:=true;
    END;
END;
outimage;

!ask the user if he wants wires and boxes converted to polygons;
outtext("Convert everything to polygons? "); breakoutimage;
inimage;  converttopoly:=false;
IF sysin.image.more THEN BEGIN
    IF sysin.image.getchar='y' THEN converttopoly:=true;
END;
outimage;

outtext("Generate bounding boxes of geometric items? ");
breakoutimage; inimage; geombb:=false;
IF sysin.image.more THEN BEGIN
    IF sysin.image.getchar='y' THEN geombb:=true;
END;


!open database file and test;
outimage; outtext(blanks(10));
outtext(IF databasefile(dfile) THEN "I New" ELSE "I Existing");
outtext(" Database file. I"); outimage; outimage;

!define database relations and their field names;
usergeom:-field("symbol").field("layernumber").field("objectnumber")
.field("pointnumber").field("point.x").field("point.y");
usersubc:-field("symbol").field("subsymbol").field("t11")
.field("t12").field("t21").field("t22").field("tx").field("ty");
boxes:-field("symbol").field("objectnumber").field("ur.x")
.field("ll.x").field("ur.y").field("ll.y");
wires:-field("symbol").field("layernumber").field("width")
.field("objectnumber") .field("pointnumber").field("point.x")
.field("point.y");
uboxes:-field("symbol").field("layernumber").field("objectnumber")
.field("dir.x").field("dir.y").field("ur.x").field("ll.x")
.field("ur.y").field("ll.y");
sboxes:-field("symbol").field("ur.x").field("ll.x").field("ur.y")
.field("ll.y");
```

```
fboxes:-sboxes.protorelation;

!start up the dictionary, make an initial entry for the;
!global symbol 0, set the current dictionary point to this entry;
syntab:-NEW dictionary;
globalval:-currentval:-NEW symval(0);
putindict(0,globalval);
super:=fwddef:=globalitems:=false;

!open the CIF file (and error file), parse the CIF;
cif2.open(cfile,efile);
cif2.parseCIF(defsym,deffin,poly,box,rflash,wire,layer,delete,
symcall,nullcomment,userext,endcom,ciferror,cifwarn,cifdone);

!sort and reset the relations;
usergeom.reset; usersubc.reset;
uboxes.reset;   wires.reset;


!test to see if any tuples were entered into any relation;
!with a symbol number of zero;
IF globalitems THEN BEGIN

    INSPECT cif2.efile DO BEGIN
       !if there were no global calls or geometry made, then;
       !this is a degenerate CIF file, issue a message;
       outimage;
       outtext(" No symbol calls or geometry in the global symbol.");
       outimage;
       outtext(" CIF file calls for no instaniation");
       outtext(" -- CIF file in error.");
       outimage;
       errors:=errorsti;
    END;

END
ELSE BEGIN

    !if there are references to symbol 0, then "finish";
    !the definition of symbol 0;
    super:=true;
    deffin;

    boxes.reset; fboxes.reset;


    !at this point all the information is in the relations, however;
    !the symbol calls have not been checked for recursion and;
    !the bounding boxes associated with symbols do not take into
    !account the bounding boxes of their subsymbols;
    !call recursive routine to follow symbol calls and check for;
    !recursive calls and at the same time update the symbol;
```

```
        !bounding boxes to their true sizes;
        outimage; outtext("         Verifying symbol calls.");
        outimage;
        symbolsinuse:-NEW vector;
        verifyandfixmbb(0);

        !if no errors,name the relations and make them permanent;
        IF errors=0 THEN BEGIN
            usergeom.nameis("CIFpolygons");
            usersubc.nameis("CIFsymbolcalls");
            boxes.nameis("CIFgeometricboundingboxes");
            wires.nameis("CIFwires");
            uboxes.nameis("CIFboxes");
            sboxes.nameis("CIFsymbolboundingboxes");
        END;
    END;

    !check for new errors issued since end of CIF file;
    IF errors-olderrs>0 THEN BEGIN
        outimage; outint(errors-olderrs,11);
        outtext(" Additional errors encountered.");
        outimage; outimage;
    END;

    cif2.close;

END;

END of CIFX;
```

Appendix V

Implementation of Additional Polygon Operations

```
!*****************************************************;
!                                                     ;
!    These procedures may be added to CLASS polygons  ;
!    in the polygon package (I. E. Sutherland, 1978)  ;
!    to provide most of the functions specified in    ;
!    in Appendix I.  These procedures use the         ;
!    the procedures already defined in the package    ;
!    to allow the logical operations required.        ;
!                                                     ;
!                                  Dick Lang 6/79     ;
!*****************************************************;



REF(polygon)PROCEDURE Inverted;
! This procedure reverses the sense of edges - gives a new polygon;
BEGIN
    REF(sheet)s;
    REF(polygon)g;
    g:-NEW polygon;
    FOR s:-nextsheet(s) WHILE s=/=NONE DO
    s.inverted.putintobox(g);
    Inverted:-g;
END of Inverted;


PROCEDURE Print;
! This procedure prints out all the stuff of a polygon;
BEGIN
    INTEGER ns,nr;
    REF(ringer)r;
    REF(sheet)s;

    TEXT PROCEDURE cvsi(i); INTEGER i;
    BEGIN
        TEXT t;
        t:-blanks(12);
        t.putint(i);
        cvsi:-frontstrip(t);
    END of cvsi;

    TEXT PROCEDURE cvs2(r); REAL r;
    BEGIN
```

```
        TEXT t;
        t:-blanks(12);
        t.putfix(r,3);
        cvs2:-frontstrip(t);
    END of cvs2;

    outimage;
    FOR s:-nextsheet(s) WHILE s=/=NONE DO BEGIN
        ns:=ns+1;
        outtext("SHEET "); outtext(cvsi(ns)); outimage;
        FOR r:-s.next(r) WHILE r=/=NONE DO BEGIN
            nr:=nr+1;
            outtext("       item "); outtext(cvsi(nr));
            outtext("  -) ");
            INSPECT r
            WHEN vertex DO BEGIN
                outtext("vertex  x=");
                outtext(cvs2(pt.x/300));
                outtext("  y=");
                outtext(cvs2(pt.y/300));
            END
            WHEN edge DO BEGIN
                INSPECT equ
                WHEN line DO BEGIN
                    outtext("line  a=");
                    outtext(cvs2(a/300));
                    outtext("  b=");
                    outtext(cvs2(b/300));
                    outtext("  c=");
                    outtext(cvs2(c/300));
                END
                WHEN circle DO BEGIN
                    outtext("circle  x=");
                    outtext(cvs2(x/300));
                    outtext("  y=");
                    outtext(cvs2(y/300));
                    outtext("  r=");
                    outtext(cvs2(r/300));
                END;
            END;
            outimage;
        END;
    END;
END of print;


REF(polygon) PROCEDURE Xform(tr); REF(transform)tr;
! This procedure makes a new copy of this polygon ;
! and applied transformation tr to it;
BEGIN
    REF(sheet)s;
    REF(polygon)g;
```

```
    g:-NEW polygon;
    FOR s:-nextsheet(s) WHILE s=/=NONE DO BEGIN
        REF(sheet)ss;
        REAL a1,a2;
        a1:=s.area;
        ss:-ss.xform(tr);
        a2:=ss.area;
        IF a1*a2<0 THEN ss:-ss.inverted;
        ss.putintobox(g);
    END;
    Xform:-g;
END of Xform;


PROCEDURE Straddlepoints(e,ptl,ptr);
NAME ptr,ptl; REF(point)ptl,ptr; REF(edge)e;
! Given an edge e, this procedure attempts to compute;
! a point on each side of it;
BEGIN
    REAL dx,dy,x,y,h,inc,deltax,deltay;
    REF(point)p1,p2;
    inc:=.01;
    IF e=/=NONE THEN BEGIN
        p1:-e.bp;
        p2:-e.tp;
        IF e.equ IS line THEN BEGIN
            REF(line)ln;
            ln:-e.equ QUA line;
            dx:=ln.a;
            dy:=ln.b;
            x:=(p1.x+p2.x)/2;
            y:=(p1.y+p2.y)/2;
            h:=sqrt(dx*dx+dy*dy);
            deltax:=inc*dx/h;
            deltay:=inc*dy/h;
        END
        ELSE IF e.equ IS circle THEN BEGIN
            REAL dx1,dx2,dy1,dy2;
            REF(circle)c;
            c:-e.equ QUA circle;
            IF c.r\=0 THEN BEGIN
                dx1:=(p1.x-c.x)/c.r;
                dx2:=(p2.x-c.x)/c.r;
                dy1:=(p1.y-c.y)/c.r;
                dy2:=(p2.y-c.y)/c.r;
            END
            ELSE BEGIN
                outtext("warning-zero radius edge"); outimage;
            END;
            dx:=dx1+dx2;
            dy:=dy1+dy2;
            h:=sqrt(dx*dx+dy*dy);
```

```
            dx:=dx/h;
            dy:=dy/h;
            x:=c.x+c.r*dx;
            y:=c.y+c.r*dy;
            deltax:=inc*dx;
            deltay:=inc*dy;
        END;
        ptl:-NEW point(x-deltax,y-deltay);
        ptr:-NEW point(x+deltax,y+deltay);
    END;
END of Straddlepoints;


REF(polygon) PROCEDURE ExtractMatching(pi); REF(polygon)pi;
! Remove those sheet of this polygon that enclose sheets of pi;
! and return them ;
BEGIN
    IF pi=/=NONE THEN BEGIN
        REF(polygon)pout;
        REF(sheet)si;
        FOR si:-pi.nextsheet(si) WHILE si=/=NONE DO BEGIN
            REF(edge)e;
            REF(sheet)s2;
            REF(point)pti;
            FOR e:-si.nextedge WHILE si=/=NONE AND pti==NONE DO BEGIN
                REF(point)ptl,ptr;
                straddlepoints(e,ptl,tpr);
                IF si.wrap(ptl)\=0 THEN pti:-ptl
                ELSE IF s2.wrap(ptr)\=0 THEN pti:-ptr;
            END;
            s2:-nextsheet(NONE);
            WHILE s2=/=NONE DO BEGIN
                REF(sheet)ss;
                ss:-nextsheet(s2);
                IF s2.wrap(pti)\=0 THEN BEGIN
                    IF pout==NONE THEN pout:-NEW polygon;
                    pout.addsheet(extractsheet(s2));
                END;
                s2:-ss;
            END;
        END;
        ExtractMatching:-pout;
    END;
END of ExtractMatching;


REF(sheet) PROCEDURE ExtractSheet(s); REF(sheet)s;
! This procedure removes sheet s from this polygon and returns it;
BEGIN
    IF s=/=NONE THEN BEGIN
        IF down=/=NONE THEN BEGIN
```

```
            IF down==s THEN down:-s.out ELSE s.out;
        END;
        ExtractSheet:-s;
    END;
END of ExtractSheet;


REF(polygon)PROCEDURE ExtractHoles;
! This procedure removes those sheets this polygon that;
! enclose negative area and returns them;
BEGIN
    REF(sheet)s;
    REF(polygon)newp;
    newp:-NEW polygon;
    s:-nextsheet(NONE);
    WHILE s=/=NONE DO BEGIN
        REF(sheet)ss;
        ss:-nextsheet(s);
        IF s.area(=0 THEN
        newp.addsheet(extractsheet(s));
        s:-ss;
    END;
    ExtractHoles:-newp;
END of ExtractHoles;


INTEGER PROCEDURE NumSheets;
! This procedure returns the number sheets owned by this polygon;
BEGIN
    INTEGER count;
    REF(sheet)s;
    count:=0;
    FOR s:-p.nextsheet(s) WHILE s=/=NONE DO count:=count+1;
    NumSheets:=count;
END of NumSheets;


REF(polygon) PROCEDURE Trim(wc); INTEGER wc;
! This procedure extracts those sheets of this polygon which ;
! separate areas of wrap=wc from those with wraps of wc+1 or wc-1;
BEGIN
    REF(vect)v;
    REF(polygon)pout;
    REF(sheet)s;

    thing CLASS sheetentry(s,wi,wo);
    ! This makes a an object that can be vectorized;
    REF(sheet)s; INTEGER wi,wo;
    BEGIN
    END;
```

```
    v:-NEW vector;
    s:-NONE;
    FOR s:-nextsheet(s) WHILE s=/=NONE DO BEGIN
        INTEGER wi,wo,w;
        REF(edge)e;
        REF(point)pti,pto;
        e:-NONE;
        w:=2;
        FOR e:-s.nextedge(e)
        WHILE e=/=NONE AND (w)1 OR w(-1) DO BEGIN
            straddlepoints(e,pti,pto);
            wi:=wrap(pti);
            wo:=wrap(pto);
            w:=wi-wo;
        END;
        IF e==NONE THEN BEGIN
            outtext("cannot find non-multitrace edge");
            outimage;
        END
        ELSE v.append(NEW sheetentry(s,wi,wo));
    END;
    IF v.length)0 THEN BEGIN
        INTEGER i,l;
        l:=v.length;
        FOR i:=1 STEP 1 UNTIL l DO BEGIN
            REF(sheetentry)se;
            REF(sheet)ss;
            INTEGER wi,wo;
            se:-v.a.a(i) QUA sheetentry;
            ss:-se.s; wi:=se.wi; wo:=se.wo;
            IF NOT((wi=wc AND wo=wc-1) OR
                   (wo=wc AND wi=wc-1)) THEN BEGIN
                IF pout==NONE THEN pout:-NEW polygon;
                pout.addsheet(extractsheet(ss));
            END;
        END;
    END;
    Trim:-pout;
    outint(numsheets(pout),3); outtext(" trimmed off");
    outimage;
END of Trim;


REF(polygon) PROCEDURE Copy;
! This procedure returns a new copy of this sheet;
BEGIN
    REF(polygon)p;
    REF(sheet)s;
    p:-NEW polygon;
    FOR s:-nextsheet(s) WHILE s=/=NONE DO BEGIN
        REF(edge)e;
        REF(sheet)ns;
```

```
        ns:-NEW sheet;
        e:-NONE;
        FOR e:-s.nextedge(e) WHILE e=/=NONE DO BEGIN
            REF(edge)ned;
            REF(vertex)nv;
            REF(point)v;
            REF(line)ln;
            REF(circle)circ;
            IF e.equ IS line THEN BEGIN
                ln:-e.equ QUA line;
                ned:-NEW edge(NEW line(ln.a,ln.b,ln.c));
            END
            ELSE IF e.equ IS circle THEN BEGIN
                circ:-e.equ QUA circle;
                ned:-NEW edge(NEW circle(circ.x,circ.y,
                circ.r));
            END;
            v:-e.suc QUA vertex.pt;
            nv:-NEW vertex(NEW point(v.x,v.y));
            ned.putinto(ns);
            nv.putafter(ned);
            ned.intobox(p);
        END;
        ns.putinto(p);
    END;
    Copy:-p;
END of Copy;


REF(polygon) PROCEDURE Combine(pb); REF(polygon)pb;
! This proceddure  takes the sheets of pb and puts them;
! in this polygon.;
BEGIN
    REF(sheet)s;
    s:-pb.nextsheet(NONE);
    WHILE s=/=NONE DO BEGIN
        REF(sheet)ss;
        ss:-pb.nextsheet(s);
        addsheet(pb.extractsheet(s));
        s:-ss;
    END;
    Combine:-p;
END of Combine;


REF(polygon) PROCEDURE Deflate(size); REAL size;
! This procedure is merely the inverse of inflate;
Deflate:-inflate(-size);


REF(polygon) PROCEDURE Intersect(a); REF(polygon)a;
! This procedure returns the intersection of this polygon;
```

```
! and a.  Both this polygon and a are modified.;
BEGIN
    IF a=/=NONE THEN BEGIN
        combine(a.copy);
        clean.selfintersect.retrace.trim(2);
        trim(p,2);
    END ELSE BEGIN
        REF(sheet)s;
        FOR s:-nextsheet(s) WHILE s=/=NONE DO extractsheet(s);
    END;
    Intersect:-THIS polygon;
END of Intersect;


REF(polygon) PROCEDURE Retrace;
! This procedure reassigns the edges of sheets into new;
! sheets which do not intersect each other at more than a line;
BEGIN
    fixedgrefs;
    fixsheets;
    Retrace:-THIS polygon;
END of Retrace;


REF(polygon) PROCEDURE Union(a); REF(polygon)a;
! This procedure returns the union of this polygon and a.;
! Both this polygon and a are modified;
BEGIN
    combine(a).selfintersect.retrace.trim(1);
    Union:-THIS polygon;
END of Union;


REF(polygon) PROCEDURE Subtract(b); REF(polygon)b;
! This procedure returns the difference of this polygon from;
! b.  Both this polygon and b are modified.;
BEGIN
    combine(b.inverted).selfintersect.retrace.trim(1);
    Subtract:-THIS polygon;
END of Subtract;


REF(polygon) PROCEDURE ExclusiveOr(b); REF(polygon)b;
! This procedure returns the exclusive or of this polygon and b;
! Both this polygon and b are modified.;
BEGIN
    REF(polygon)temp;
    temp:-combine(b).selfintersect.retrace.trim(1);
    ExclusiveOr:-intersect(temp.inverted);
END of ExclusiveOr;
```

```
REF(polygon) PROCEDURE TakeVectorAsSheet(v); REF(vector)v;
! This procedure takes a vector of points and builds a ;
! straight line connected sheet and combines it with ;
! this polygon.;
BEGIN
    REF(sheet)s;
    REF(point)pt,bp,tp,sp;
    INTEGER length;
    s:-NEW sheet;
    IF v.length)1 THEN BEGIN
        sp:-bp:-v.val(1) QUA point;
        length:=v.length;
        FOR i:=2 STEP 1 UNTIL length DO BEGIN
            tp:-v.val(i) QUA point;
            s.makedge(bp.lineto(tp),tp);
            bp:-tp;
        END;
        s.makedge(tp.lineto(sp),sp);
        IF s.area<0 THEN s:-s.inverted;
    END;
    addsheet(s);
    TakeVectorAsSheet:-THIS polygon;
END of TakeVectorAsSheet;


!************************************************;
!                                                ;
!    The following procedures can be added to    ;
!    CLASS sheet in the polygon package.         ;
!                                                ;
!************************************************;


REF(sheet) PROCEDURE Inverted;
! This procedure builds a new sheet with reversed sense edges;
BEGIN


    !this has to pass by the edges;
    REF(ringer) PROCEDURE lastringer(v); REF(ringer)v;
    BEGIN
        lastringer:-IF v==NONE THEN sht.down
        ELSE IF v.pred=/=down THEN v.pred ELSE NONE;
    END of lastringer;

    REF(ringer)lr;
    REF(sheet)s;
    s:-NEW sheet;
    FOR lr:-lastringer(lr) WHILE lr=/=NONE DO BEGIN
        IF lr IS vertex THEN BEGIN
            REF(point)pt;
```

```
            pt:-lr QUA vertex.pt;
            NEW vertex(NEW point(pt.x,pt.y)).putinto(s);
        END
        ELSE IF lr IS edge THEN BEGIN
            REF(equation)e;
            e:-lr QUA edge.equ;
            IF e IS line THEN BEGIN
                REF(line)ln;
                ln:-e QUA line;
                NEW edge(NEW line(-ln.a,-ln.b,-ln.c)).putinto(s);
            END
            ELSE IF e IS circle THEN BEGIN
                REF(circle)c;
                c:-e QUA circle;
                NEW edge(NEW circle(c.x,c.y,-c.r)).putinto(s);
            END;
        END;
    END;
    Inverted:-s;
END of Inverted;



REF(sheet)PROCEDURE Xform(tr); REF(transform)tr;
! This procedure applies transformation tr to this sheet;
BEGIN
    REF(ringer)lr;
    REF(sheet)s;
    s:-NEW sheet;
    FOR lr:-next(lr) WHILE lr=/=NONE DO BEGIN
        IF lr IS vertex THEN BEGIN
            REF(point)pt;
            pt:-lr QUA vertex.pt;
            NEW vertex(tr.pt(pt)).putinto(s);
        END
        ELSE IF lr IS edge THEN BEGIN
            REF(equation)e;
            e:-lr QUA edge.equ;
            IF e IS line THEN BEGIN
                REF(point)pt1,pt2;
                pt2:-tr.pt(lr.suc QUA vertex.pt);
                pt1:-tr.pt(lr.pred QUA vertex.pt);
                NEW edge(pt1.lineto(pt2)).putinto(s);
            END
            ELSE IF e IS circle THEN BEGIN
                REF(circle)c;
                REAL x,y;
                c:-e QUA circle;
                x:=t11*c.x+t21*c.y+tx;
                y:=t12*c.x+t22*c.y+ty;
                NEW edge(NEW circle(x,y,c.r)).putinto(s);
```

```
            END;
        END;
    END;
    Xform:=s;
END of Xform;
```