# LAYOUT RULE SPACING OF SYMBOLIC INTEGRATED CIRCUIT ARTWORK

by

M. W. Bales

Memorandum No. UCB/ERL M82/72

4 May 1982

(cover)

# LAYOUT RULE SPACING OF SYMBOLIC INTEGRATED CIRCUIT ARTWORK

by

Mark W. Bales

Memorandum No. UCB/ERL M82/72

4 May 1982

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

**Mark W. Bales**

Author

# Layout Rule Spacing of Symbolic Integrated Circuit Artwork

Title

RESEARCH PROJECT

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Committee: Richard Newton Research Advisor

5/4/82 Date

5/6/82 Date

# CHAPTER 1

# Introduction

## 1.1. Layout Rule Spacing of Symbolic Integrated Circuit Designs

The increasing complexity of Very Large Scale Integrated (VLSI) circuits has made the use of computer aids for design, analysis, and data management a necessity.

The huge volume of data associated with the integrated circuit (IC) design process can only be managed effectively if the *structure* of the circuit is exploited by the designer and the CAD programs. Some form of hierarchical technique must be used to reduce the number of objects the designer must work with at any one time. A hierarchy can also reduce the amount of data storage required by the CAD system if the IC designs include replications of structures. Designs use replicated structures if they contain very regular structures, such as arrays. Cell-based design also uses replication to make use of a hierarchy. Efficient tools are needed to manage this hierarchy.

The use of abstractions, or *symbols*, to represent components of a design method (transistors, contacts, cells, *etc.*) can help reduce the complexity of the design process. Symbolic IC design has been in use since the early 1970s [Larsen71]. The use of symbols to represent IC devices is a logical mapping between the electrical schematic of circuits and the semiconductor process for creating the devices in silicon. In the circuit schematic, each symbol represents a complete device on a functional level. The electrical properties of each device-type are well modeled and the circuit-level device models are used as a basis for the electrical design of a circuit. In the semiconductor IC process, several photographic masks are used to create the circuit devices on the silicon through a series of chemical diffusions, oxidations, and implantations. Each mask is composed of a pattern which can be constructed from many rectangles. An individual mask contains the information used in one processing step for all of the devices in the IC. Each separate electrical device requires many mask steps, and hence, the mask representation of a single device includes many rectangles on many mask layers. In a symbolic design, all this information is coalesced into a single symbol that represents the circuit-level *device*.

Symbols contain information that is normally lost in present-day rectangle-based IC layout systems. The electrical connectivity of the circuit, as well as the placement of the components themselves, is lost when a design is entered in a rectangle-based artwork system. This information must be extracted from the mask rectangles if, for example, a circuit simulation or testability analysis is to be performed. Layout rule and electrical rule checking have more difficult tasks because they must reconstruct the circuit devices from the mask artwork to identify their context correctly. When symbols are used to represent individual devices, more information is available for use by auxiliary design tools. The symbols and their interconnections are strictly defined and the transformation of the symbols into geometric mask shapes can be an error-free process. Thus, the design and maintenance of IC's with symbols permits the design system to capture more of the design intent than placing individual rectangles on multiple mask layers.

In addition to providing a layout capability, symbolic layout systems may be used to guarantee that the spacing between elements in the design satisfies the layout rules for the IC process. Even further, the system can be used to compact the elements together to achieve a minimum size layout that still satisfies all layout rules and maintains circuit connections. The interconnection lines between invariant, primitive symbols (such as transistors, contacts, or circuit cells) are stretched or shrunk to change the size of the IC layout, while the dimensions of the symbols themselves remain unchanged. Thus a symbolic layout system may be used in a number of ways. It may merely provide a layout capability which allows IC mask artwork to be generated from the symbols. It may also provide a *spacing program* which is used to insure that the spacing between elements is correct. In addition, the symbolic design system may provide a *compaction program* which may be used not only to insure that the spacing between elements is correct but also to attempt to minimize the final area of the layout.

The following section of this chapter presents a review of existing symbolic design systems with a focus on spacing and compaction programs. Chapter 2 of this report describes the *CABBAGE* system, its limitations, and the additional capabilities that are useful in a compaction program. CABBAGE is designed for an NMOS polysilicon gate, single-layer metallization process. Details of primitive circuit elements in the NMOS technology as well as layout rules are hard-coded into the program. The concepts of *protection frames* and *terminal frames* are introduced to extend the CABBAGE algorithms to make them technology and process independent. The *Python* program, an implementation of these algorithms, is presented and the features of polygonal protection frames, box terminal frames, and *maximum* as well as minimum constraints are described.

Chapter 3 describes the specific algorithmic details of Python.

Chapter 4 presents two examples, a D-type flip/flop and a latch cell. For both examples, a comparison is made between CABBAGE and Python with respect to run-time, memory usage, and compaction efficiency. Python produces a layout which is 15% smaller than the CABBAGE result for the D-F/F example and the Python layout is 15% larger than the CABBAGE layout for the latch example. The reasons for these differences are explained. Overall, Python runs approximately 2.5 times faster than CABBAGE and uses approximately 2.5 times the amount of memory. Protection frames are generated for the D-F/F example and the D-F/F is used to construct a shift register cell. This shift register cell is compacted with Python to demonstrate the use of hierarchical compaction. Order dependencies for the algorithms in Python are reported.

Chapter 5 provides a summary of the results. The problems associated with necessary enhancements to Python are presented with recommendations for their solution.

## 1.2. Existing Symbolic Design Systems

### 1.2.1. Fixed Grid Systems

Fixed grid symbolic IC design systems have been in use within industry since the early 1970s [Larsen71]. The fixed grid represents the allowed locations of geometry for the IC layout. Quantizing the representation of the circuit in this manner allows efficient representation of the IC layout grid with the array constructs found in high-level computer languages. Early programs only allowed layout of highly regular structures, such as programmable logic arrays (PLAs). The *SLIC* program at AMI [Gibson76] allows arbitrary layout of IC cells. Each grid point may contain zero or more symbols. If two symbols overlap or are adjacent, simple rules assume the geometries the symbols represent are connected. In SLIC, a circuit is created by drawing the symbols representing the circuit on grid paper (See Fig. 1.1). This hand-drawn symbolic form is digitized, and the SLIC system generates the actual mask data from the symbols (See Fig. 1.2). The *MASKS* system from Rockwell International [Larsen78] has a similar input format, but the layout symbols are placed in a 'discrete topological schematic', which are entered directly as program data. Figure 1.3 shows the symbolic representation of a 2 input NAND gate using this approach.

Neither of these fixed-grid symbolic layout systems provides the capability to dynamically adjust the spacing between symbols to satisfy spacing rules. SLIC provides design rule checking and interactive editing to correct any design errors. Both systems provide for generation of actual mask data from the symbolic layout. Simple spacing rules are enforced through the

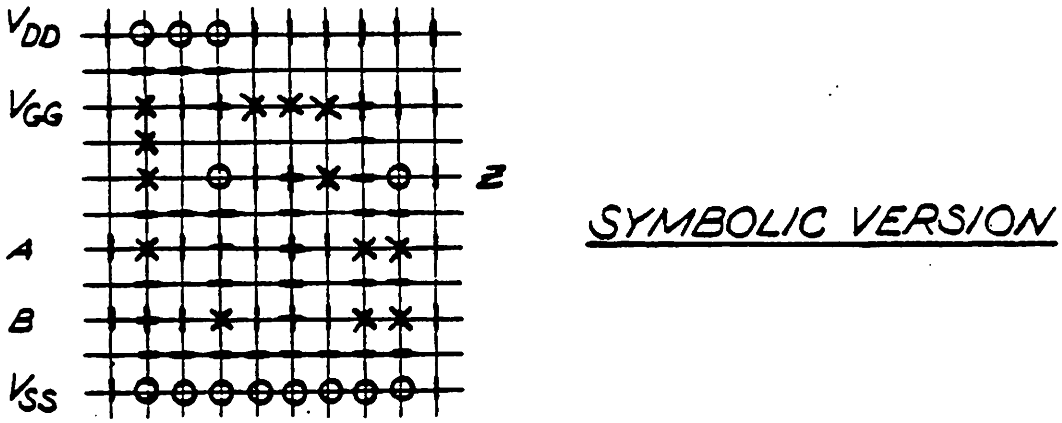


Fig. 1.1 Symbolic Input for the SLIC system ([Gibson76])

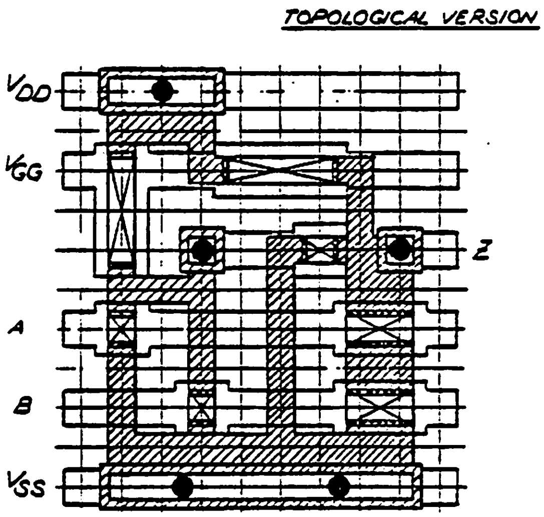
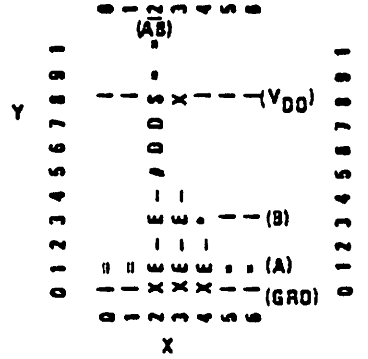


Fig. 1.2 Mask Layer Output from the SLIC system ([Gibson78])



NOTE: TOPOLOGICAL SCHEMATIC SYMBOLS ARE ROTATED 90° COUNTERCLOCKWISE

Fig. 1.3 MASKS System Input ([Larsen78])

choice of the spacing between grid points.

The basic limitation of fixed-grid layout systems is the forced choice of the worst-case design rule for the grid spacing. For example, any two interconnection lines one grid spacing apart must satisfy the spacing rules regardless of their type. In a typical NMOS IC process, the metal to metal spacing rule is the largest and will determine the grid spacing. Lines of another type (polysilicon, for example), might not require the largest spacing rule, but they must be at least one grid spacing apart to guarantee that the spacing rules are satisfied. Some area will be wasted as a result. A system with a finer grid, with the grid spacing the greatest common divisor of all of the spacing rules, would overcome this limitation. The symbols used would no longer be simple, since the would occupy many grid locations. Also, the data storage required would increase at least an order of magnitude. Current research at AMI and Rockwell International is being performed with this method.

### 1.2.2. Relative Grid Systems

Relative grid systems only use the grid-based symbolic layout of the IC cells to indicate the relative placement of symbols and to determine the electrical connectivity of the circuit. The locations of symbols are represented as some fraction of the greatest common divisor of the spacing rules. Although the representation of the elements is usually not a grid, the choice of the minimum resolution for symbol location makes the relative-grid approach similar in most respects to the fixed-grid method using a finer grid.

#### 1.2.2.1. Systems Previous to Cabbage I

One of the first programs to use the relative-grid method is the *FLOSS* program from RCA [Cho77]. FLOSS is similar to the SLIC program in that it reads a digitized sketch of the layout. The spacing between objects is determined by the construction of the objects and the individual spacing rules between the mask layers of the components that make up each object. Only orthogonal interconnection lines are allowed and the interconnections to cells must be points. While these restrictions are implicit in the nature of fixed-grid layout systems, they are imposed on relative-grid systems only through implementation difficulties.

FLOSS supports a hierarchy for circuit design, and results have been published for the compaction of entire ICs. The IC shown in Figure 1.4 is 32% smaller than the original symbolic sketch and 19% larger than a hand-drawn layout of the same IC.

Another symbolic layout aid which uses a relative grid and provides compaction is the *STICKS* program, developed at the Hewlett Packard Co [Williams78]. This program compacts the symbolic layout by starting from one side of the layout and sequentially placing elements as far to the side as is possible, given the positions of the previously placed elements and the spacing rules objects on different mask layers. The electrical connectivity of the circuit is also a determining factor in the positioning of elements during

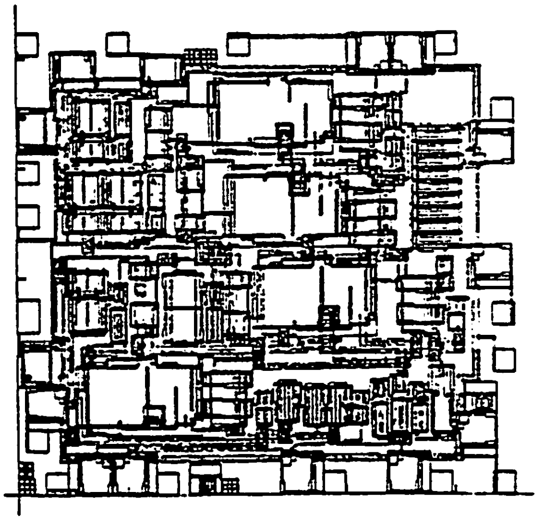


Fig. 1.4 Symbolic IC Layout After FLOSS Compaction ([Cho77])

Cabbage I has been used to design a complete digital filter IC at the Katholieke Universiteit Leuven in Belgium [Hurt82]. This IC has over 1500 NMOS transistors, included in over 9000 layout symbols, and has been successfully fabricated ( Layout shown in Fig. 1.6). The computer time required to compact this example was over 4 CPU hours and demonstrates the need for hierarchical compaction and layout.

### 1.2.2.3. Other Systems Since Cabbage I

The *SLIM* system [Dunlop80] combines a shear line algorithm used by Akers [Akers70] with the graph representation of the IC symbols. SLIM is a successor to the *SLIP* system [Dunlop79]. In this program, multiple spacing methods are used to optimize both speed and compaction efficiency. The IC symbolic data is partitioned automatically into optimal size groups calculated by the program [Dunlop79]. A loose initial placement guarantees a layout-rule correct (although not optimal) relative placement of the partitions. Critical path analysis, similar to the method used in CABBAGE, is

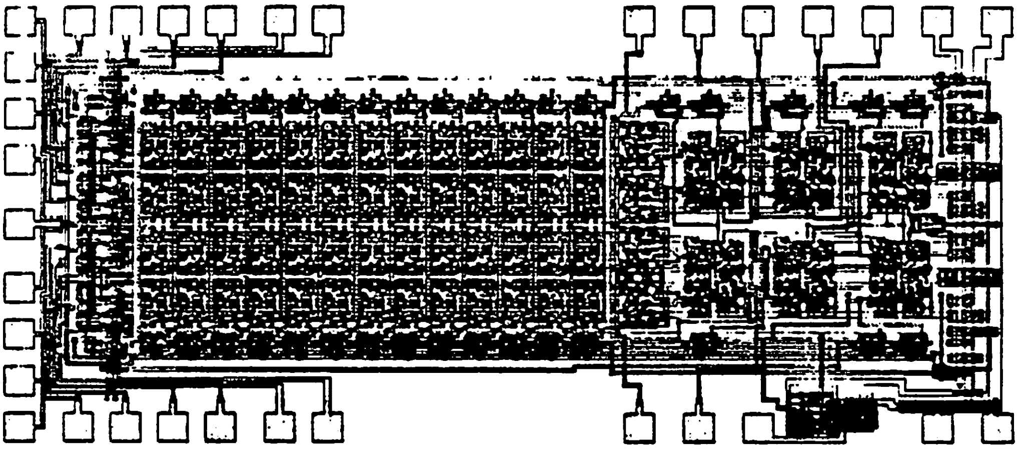


Fig. 1.6 KUL Digital Filter IC [DeMan82]

coupled with a local-compaction method. This reduces the total computer time required for solution of the compaction of each partition while maintaining an efficient compaction result. The local compaction procedure clusters together objects on the critical path. Jogs are inserted as zero-length lines perpendicular to the direction of the line into which they are inserted. They allow the objects connected to the top and bottom parts of a line to move independently in the direction parallel to the jog. In SLIM, jogs are only inserted at contact locations. Global rift line compaction [Akers70] removes the excess space between the locally compacted partitions. The order dependency for the execution time of the composite algorithm is approximately $O(n^{3/2})$.

### 1.2.2.4. MULGA

The *MULGA* system from Bell Telephone Laboratories provides a compaction capability which differs from other relative-grid layout systems. It uses a *virtual grid* to perform compaction of the symbolic layout. This virtual grid combines the ideas of both the fixed and relative-grid approaches to IC layout. Symbols are placed at grid locations, as in fixed-grid layout. The spacing between grid rows and columns is adjusted and takes on a real value, dependent on the actual spacing required by objects on each row. Individual rows and columns may have a unique spacing, so the grid spacing is non-uniform. Given the non-uniformity of the grid, there is no area penalty when two adjacent objects have a required spacing smaller than the worst-case layout rule. The resulting compacted layout is more dense than could be attained with fixed-grid layout but generally not as dense as in a compaction with a true relative-grid compaction method. Objects that are originally placed on a single row or column in the grid will remain on that row or column throughout the compaction, whether they are physically connected or not. In a relative-grid system, this restriction is not made and objects that are not physically connected may move relative to one another to achieve a more compact final result.

## 1.3. Program Characteristics

Python can perform hierarchical compaction of IC designs, through use of a hierarchical database for storage of the symbolic IC layouts and the use of abstractions of cell layouts. Spacing rules between mask layers in the IC process are specified in an ASCII file. The program uses these spacing rules to determine the minimum allowable distance between elements. All interconnections must be orthogonal. To allow technology and IC family independence the compaction is performed on interconnected protection frames. Protection frames are an abstraction of a cell. They reduce the amount of data needed at each level in the hierarchy. These frames may be of arbitrary orthogonal polygon shape. For each symbol, a set of frames is allowed on each mask layer in the IC process.

### 1.3.1. MFB - An Exercise in Terminal Independent Graphics

In preparation for the work on Python, a terminal-independent graphics package was co-written with the author of *Hawk*. *MFB*, a *M*odel *F*rame *B*uffer, is a database approach to terminal independent graphics. An ASCII file contains descriptions of the capabilities of graphics terminals which implement a predefined standard set of graphical functions. An example capability definition is the string 'GCS=E*dA', which defines the sequence for *G*raphics *C*lear *S*creen as '^[*dA'. The qualification of a terminal includes many of these definitions, and the predefined capabilities are a part of the definition of the model frame buffer. Not all video terminals will have a full set of capabilities for example, black and white graphics terminals do not have a video lookup table for color mapping.

There are two levels within the program package. A set of low level routines communicates with the ASCII database file and retrieves and parses the capabilities for a desired terminal specified within the file. A second level of routines executes primitive operations such as *SetColor*, *DrawLine*, *etc.*, and provides a graphical interface to the high-level applications program which is using MFB.

MFB requires that the high-level application program make its requests to the graphics interface based on the presence or absence of certain capabilities. For example, setting the color to blue on a black and white terminal would produce unpredictable results! If the high-level application program is to be truly graphics terminal independent, it must map its function onto the capabilities present in the many graphics terminals on which it may be used.

MFB is used by both the KIC [Keller81] and the Hawk graphics editors and fulfills its purpose in providing terminal independent graphics editing. See Appendices B and C for more detailed descriptions of the routines used in MFB.

# CHAPTER 2

# Compaction Algorithms

## 2.1. Cabbage

### 2.1.1. Introduction

The *CABBAGE* program was designed to work with an NMOS polysilicon gate integrated circuit process and is described in detail in [Hsueh79]. After a brief overview of how the program is used, this chapter describes the capabilities and limitations of CABBAGE. Enhancements necessary to make a useful production system are described, and the concepts of *protection frames* and *terminal frames* are introduced in order to accomplish the implementation of the *Python* program.

In a typical design session the engineer can lay out an NMOS cell using the *GRLIC* graphics editor and the symbolic layout is saved in an intermediate file. The *PRSLI* compactor reads this symbolic intermediate format and represents the physical topology of the cell with vertices and edges of a graph. The vertices of the graph represent the positions of electrically connected subgroups of objects within the cell and the edges of the graph represent the minimum required separation between groups that is imposed by the integrated circuit layout rules. Solution of the *longest path* through this graph using the Critical Path Method (CPM) [Thesen78] yields a minimal area for the entire cell while satisfying all of the layout rules. The compacted form is saved in the same intermediate form that was produced by the graphics editor GRLIC. Thus the designer is able to re-edit the compacted version of his cell, make topological chaages which allow another compaction step with the PRSLI program and perhaps obtain a better result. Interaction between the designer and the computer in this fashion minimizes much of the tedium involved in IC layout. The computer programs allow the designer to focus on the higher-level topological placement considerations of the layout process without requiring explicit attention to the exact spacing requirements between objects being placed.

### 2.1.2. Capabilities and Limitations of the System

The primitive objects, or symbols, available in the CABBAGE program are:

- Enhancement transistors
- Depletion transistors
- Diffusion-metal contacts
- Polysilicon-metal contacts
- Buried contacts
- Butting contacts.

The transistors are polysilicon-gate and there is a single layer of metal available for interconnections. An active-area mask is defined to complete the implementation of approximately 60 layout rules A single bounding rectangular polygon surrounds the cell. This rectangle is user-defined on another defined mask called *RUNX*. Each point structure (transistor, contact, *etc.*) is allowed only one interconnection point per side.

The construction of the NMOS primitives that CABBAGE uses are encoded directly into the program. This means that there is a special-purpose subroutine for transistors, one for contacts, and one for lines. In order to change the characteristics of the primitives, such as the polysilicon or diffusion extensions in a transistor, it is necessary to change some numbers in these special-purpose subroutines and to recompile the program. To extend CABBAGE to other IC technologies, it is necessary to write these special-purpose subroutines that understand the construction of each primitive in the target technology. This requires an intimate knowledge of the CABBAGE program itself.

The spacing analysis in CABBAGE is performed separately in both the X and Y directions. This separation is used for ease of implementation and for efficiency of the compaction algorithms since most IC geometry is orthogonal. Since the analysis is decoupled, the compaction process consists of alternating compactions in each axis direction. Iteration is required since spacing rules in the direction perpendicular to compaction are ignored and design-rule violations may be generated in this direction. In order to guarantee a legally spaced layout, successive iterations in the two axis directions are required until convergence is acheived (no .elements change position relative to one another) in *both* directions. Only then is the resulting layout guaranteed to meet all spacing rules.

In order to preserve electrical connectivity and in order to minimize the amount of memory required by the program, CABBAGE recognizes *groups* of topologically connected elements which share a common centerline in one or the other directions of compaction. These groups move as a unit and this keeps the electrical (connectivity) properties of the circuit correct.

CABBAGE uses the vertices of a graph to represent the locations of the groups of primitives and the locations of the lines interconnecting them. Edges are added to the graph to represent the minimum spacing requirement between groups. This minimum required spacing is found by tracing the right side of a primary group (P) and comparing it with the left side of a neighbor group (N) which is to the right of the primary group (See Fig. 2.1).

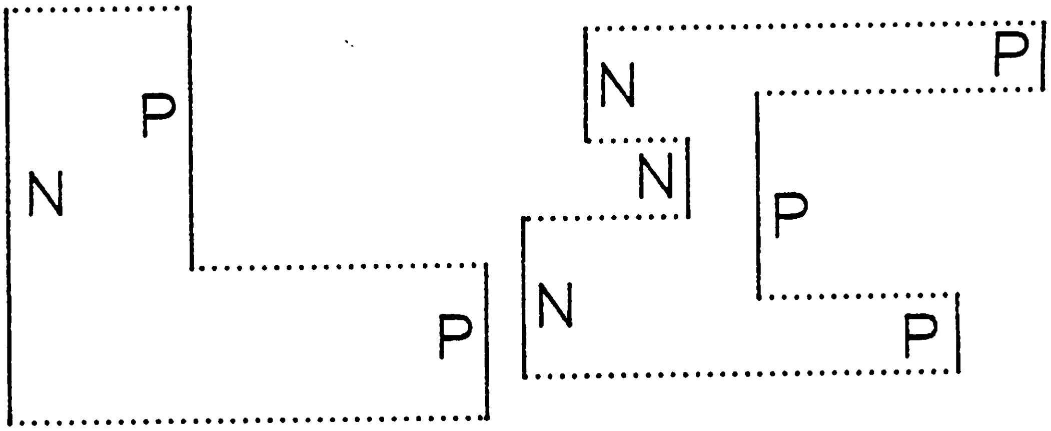


Fig. 2.1 X Graph Primary and Neighbor Edges of Two Elements

Each primary edge segment, or *interval*, is compared to each neighbor interval. The two intervals which generate the maximum spacing requirement determine the minimum allowable spacing between the two groups. The reference points for each group and the analysis routines are designed to ensure that the generated graph is a single-source, single-sink, acyclic digraph [Hsueh79].

Once the entire constraint graph with edges representing the spacing requirements has been generated, the Critical Path Method [Thesen78] (also referred to as the PERT method) is used to solve for the longest path from the graph source to graph sink. This step determines the positions of the centerlines of all groups.

An additional capability in CABBAGE is the ability to automatically insert jog points into appropriate interconnect lines in the IC layout. A byproduct of the longest path analysis can be used to determine a 'torque' on each interconnect line perpendicular to the direction of compaction. The lines which have greatest torque are then jogged. To maximize the effectiveness of the inserted jog, it is necessary to examine the geometry surrounding the line which is to be jogged and insert the jog where it will allow the greatest area savings upon subsequent compactions. CABBAGE does not do this.

CABBAGE was one of the first IC layout programs to use a relative-grid symbolic approach. Although the general concepts of symbolic layout have been in use for many years, *e. g.* [Larsen71], CABBAGE was one of the first layout compaction programs to use symbolic representations of the primitive devices available in a semicondutor process rather than symbolic representations for separate mask geometries. It is this difference which allows CABBAGE to compact layouts efficiently.

**2.2. Desired Enhancements** A major limitation of the CABBAGE system is the absence of a true hierarchy. With the order dependencies of the algorithms used in CABBAGE greater than linear, the computation time necessary for compaction of cells soon becomes impractical as the size of the cells increases. Although CABBAGE is extremely fast for the compaction of small, lower-level cells, the compaction of an entire IC design is almost impossible. An example of a 1500 transistor IC designed with CABBAGE [Hurt82] took several CPU hours for the complete compaction of the entire design on a VAX 11/780 32 bit minicomputer running the VAX/VMS operating system. Since the design was composed of only 20-30 cells, the use of true hierarchy might have reduced substantially the total compaction time.

A second desired enhancement is the ability to have more than one interconnect line per side of an object. CABBAGE requires each interconnect line to terminate on the exact center of the object to which it is connected for purposes of determining electrical connectivity. This has the effect of limiting the interconnections to primitives to a single connection per side. If the use of a hierarchy is to be effective, it is necessary to allow many interconnection lines per side of a higher level cell, such as an ALU or register file.

A compaction program should also store connectivity information in the description of the IC layout. This facilitates having multiple interconnections per side of each object, since they are no longer required to terminate on the exact center to specify the electrical connectivity.

Since lines will no longer be required to terminate on the center of the objects to which they connect, *sliding contacts* are a natural extension of the concept of terminals. If a line is allowed to connect to an object within a specified *range* along the side of the cell, the compaction program has more flexibility to arrange the interconnect and can potentially obtain a more compact final result (See Fig. 2.2).

Another logical extension to the hierarchy of cells is to allow cells to have complex shapes. To keep the program efficient, these shapes can be

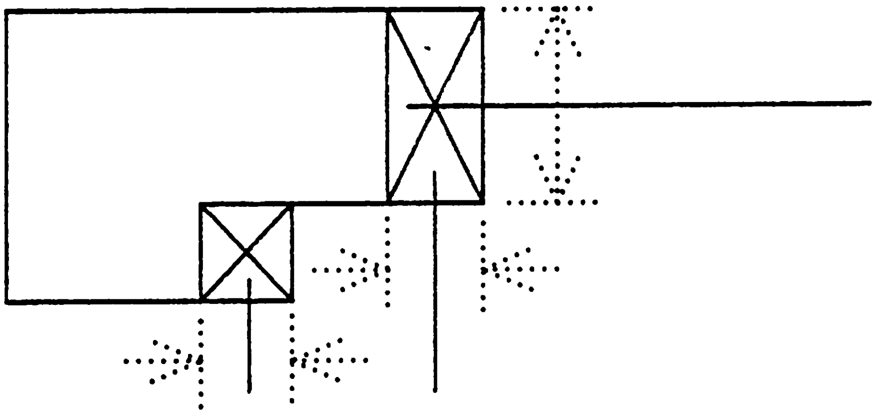


Fig. 2.2 Sliding Contacts on the Boundary of a Cell

restricted to rectangular polygons. Polygons are more general than bounding boxes, and allow compaction of odd shaped cells to obtain more optimal compaction results. At the same time, they can provide a method for routing over or through cells providing the geometry within the cell permits it.

Technology and process independence is also an important requirement for a general purpose compaction program. Technology independence implies that the program is easy to extend to new IC technologies, rather than being limited to a single technology. This calls for a general model for compaction which pays little or no regard to geometric construction of the devices which are primitives in a specific IC technology. Process independence implies abstraction of the specification of the spacing rules used by the program. Over the life of an IC technology, the spacing rules associated with the process will change many times and the compaction program must allow such updates in spacing rules to be made easily, with a minimal impact on the IC designs already entered into symbolic form.

It is also necessary to allow user constraints, both fixed and relative. Fixed constraints fix the size of an object, such as an interconnect line. Relative constraints fix the relative positions of two or more objects. An example of the need for these capabilities is found in standard cell design, where the pitch of cells and the location of busses within the cells are rigidly specified.

The increased extent of the information necessary for representation of the IC layout and related symbolic information calls for a comprehensive and efficient unified IC database. This database must be capable of storing logic information, such as electrical connectivity, as well as physical information, such as the geometric properties of a given cell. There must be a mechanism for storing information of a higher level, *e.g.* simulator modelling data, to make the database general enough to be used by a large suite of IC CAD tools. At the same time, the database must be specific enough to the problems associated with the storage of IC data to be more efficient than a general purpose database molded for use with ICs.

The last requirement for a useful compaction program is a powerful graphics input editor. This editor must be capable of quick layout and editing of the symbolic designs, as well as providing an interface to the auxiliary tools, such as the spacing program.

### 2.3. New Concepts in Python

In order to enhance the capabilities of CABBAGE, it is necessary to introduce some new concepts. These concepts are *protection frames* and *terminal frames* and they are required to augment the symbolic representation in order to make it process and technology independent. Once the symbolic model is sufficient to allow a more general spacing process, the spacing algorithms must be modified to accomodate more general shapes and to allow more flexible constraints to be used in the spacing process. The scope of these changes is described at the end of this chapter. Some important issues which have been bypassed in Python are described in detail.

### 2.3.1. Protection Frames

Protection frames are used to define the limits of the geometry contained within a cell - to 'protect' all of the geometry contained inside them. The only rule concerning protection frames as used here is that no geometry appear outside its protection frame. Given this single restriction, there are many valid interpretations of frames.

On the simplest level, a single bounding box could be used (See Figs 2.3 and 2.4 for illustrations of bounding box protection frames). Although the derivation of this box is easy and efficient, it does not provide a true representation of the internal geometries it bounds. Quite large unused areas within the cell are possibly 'invisible' outside the cell (the area dep-

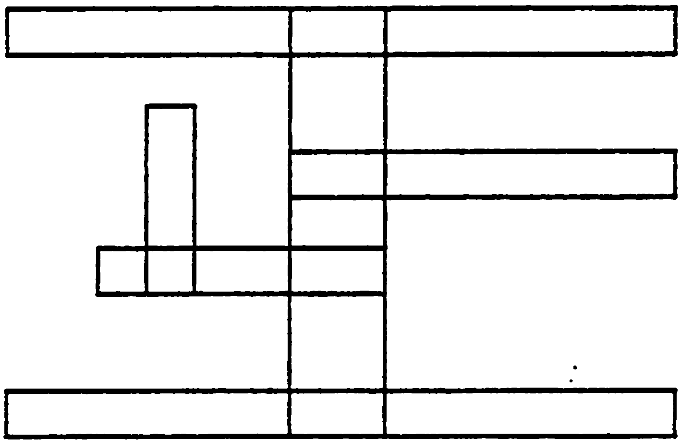


Fig. 2.3 Geometry of Cell

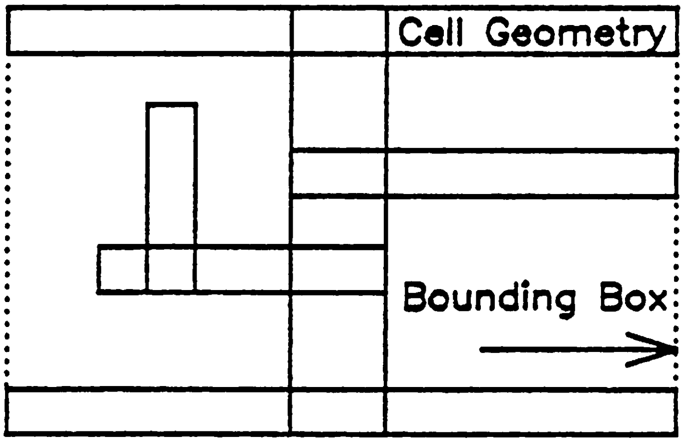


Fig. 2.4 Geometry of Cell with Simple Bounding Box

icted in Fig. 2.5).

The bounding box model of protection frames can be extended by making a separate bounding box for each mask layer in the IC process (Fig 2.6). Since the limits of each layer are most likely different, less unused area within the cell is 'invisible' outside the cell.

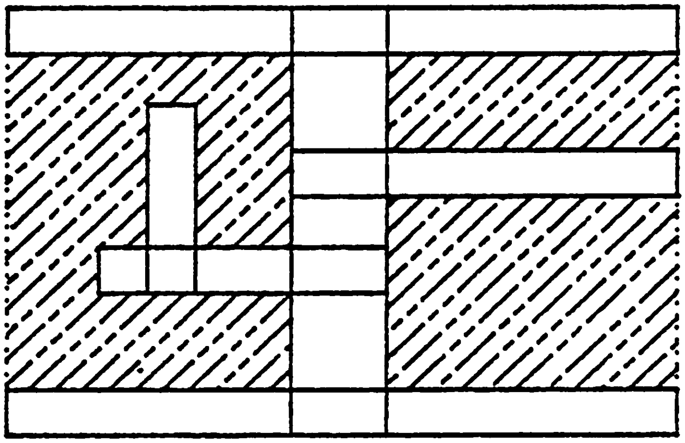


Fig. 2.5 'Invisible' Areas Within Bounding Box

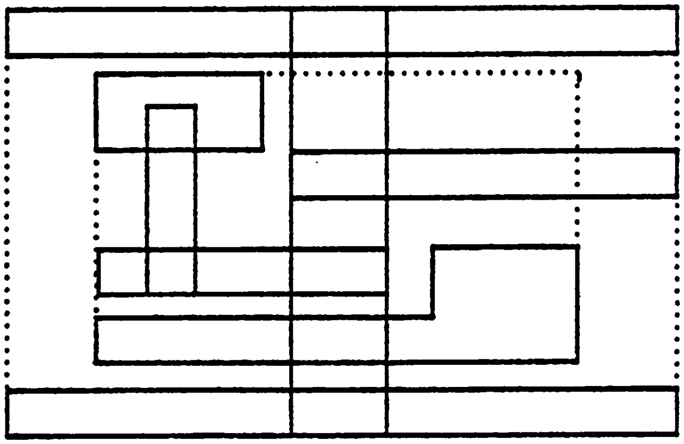


Fig. 2.6 Cell Geometry with Per-layer Bounding Rectangles

Per-layer sets of bounding rectangular polygons form the basis of protection frames as used in Python (Fig. 2.7). These allow an arbitrary tradeoff between spacing efficency and computational efficency. The closer a frame approximates a bounding box, the smaller the amount of information necessary to process to correctly space that frame. This will mean less computer time for spacing the frame. It will also mean the greatest area loss (barring a bounding box bigger than the actual cell geometry). On the other extreme, merging the geometry internal to the cell to form the protection frames will provide the greatest area efficiency, since all of the unused area within the cell may be used at the next level in the hierarchy. This approach reduces the effectiveness of the hierarchy however, since the amount of information contained in these merged protection frames is roughly comparable to examining all of the interior geomtries individually. The only gain remaining at this point would be due to multiple instances of the same cell.

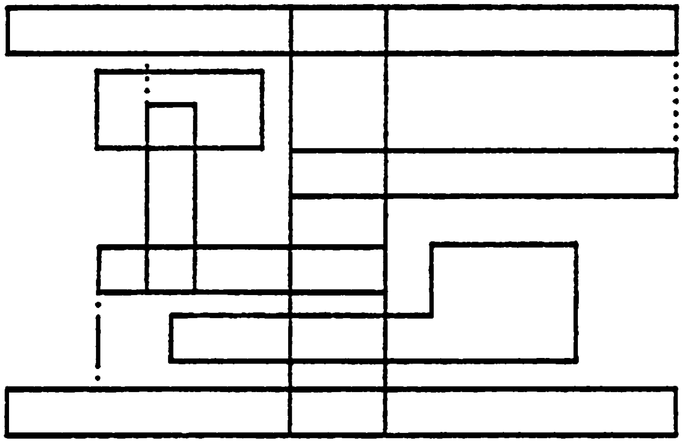


Fig. 2.7 Protection frames as used in Python

Protection frames define 'inviolate' regions. Thus it makes no sense to allow geometries to be larger than their protection frames. The reverse argument is not as obvious. The frames may be larger than the geometries within them.

Layout rules may be included in the protection frames [Lock82]. It is not only guaranteed that no geometry appears outside the frame but it is also guaranteed that geometry will not appear within one half of the maximum rule inside of the frame as well (Fig. 2.8). This has a few advantages, such as reducing the computation necessary to detect situations where there is no possibility of conflict between cells. This method taken as a whole, however, is quite cumbersome.

If protection frames were generated for each combination of mask layers with a spacing rule, there could be as many as $\frac{(m^2 - m)}{2}$ protection frames, where $m$ is the number of masks. Multiplied over the number of

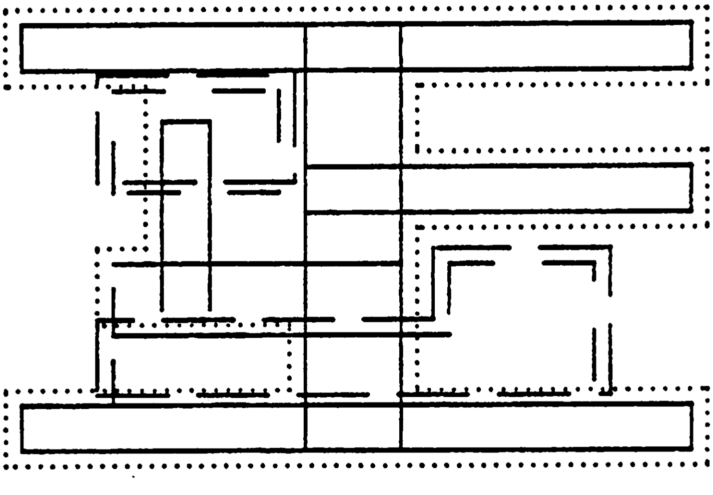


Fig. 2.8 Protection Frames With Rule

cells likely to appear in even a modest chip design, the amount of data storage required becomes very large.

The alternate approach is to include the worst case design rule (the maximum of the layer-to-layer spacing rules) and have only one protection frame for the geometry on each layer. Thus, we would have a maximum of *m* protection frames for each cell, with a corresponding reduction in data storage. Having only one such protection frame will not produce the best compaction unless all of the spacing rules are equal.

An additional problem with including spacing rules in the protection frames occurs when the spacing rules must be changed. All protection frames of all cells must then be recomputed. If the spacing rules had not been included, a simple spacing rule matrix could have been changed and the same information would now be available to use with all of the cells.

Thus the interpretation of protection frames *without* spacing rule is better. Only one set of frames is required for each mask layer in the IC process. An auxiliary set of spacing rules is used to determine conflicts between cells. If the spacing rules change, the protection frames remain unchanged.

### 2.3.2. Terminal Frames

These complement protection frames and define allowable areas of interconnection within the protection frames. One of the key differences between Python and CABBAGE is this idea of terminal connections that are *areas* instead of *points*. Rather than requiring a connection at the exact center of a point structure such as a contact, the entire contact area is suitable for termination of the interconnecting line. The connection is limited so that the line always remains within the boundaries of the terminal frame. Thus if a line is the same width as its terminal frame, the effect is the same as the point terminals in CABBAGE. If, however, the terminal frame is *wider* than the interconnect line, the interconnect can move between two constraints. This movement allows a spacing program to take maximum advantage of the instances and interconnections on the most constraining path through the IC geometry to obtain the most area efficient spacing solution.

There are two major restrictions on terminal frames. First, they must have at least one edge coincident with a protection frame edge. With the definition of protection frames presented above, it is not possible for a terminal frame to exist outside a protection frame (See Fig 2.9). To permit this would allow connection to a terminal at a point where there could not possibly exist geometries since frames define 'inviolate' areas on mask layers. On the other hand, having the terminal frame entirely within the boundaries of its protection frame would require interconnect to cross the 'inviolate' area of the frame (See Fig. 2.10). This is in violation of the restrictions placed on protection frames. So terminal frames must share at least part of one edge

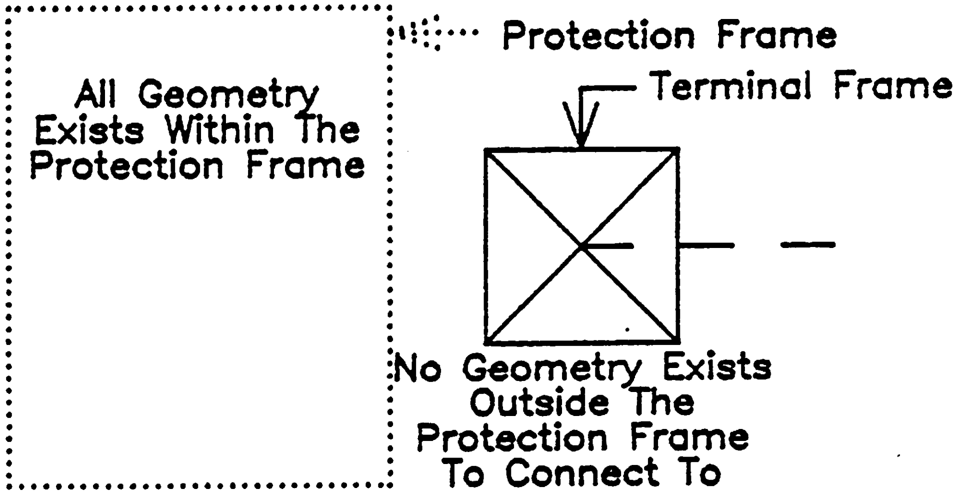

Protection Frame
Terminal Frame
All Geometry Exists Within The Protection Frame
No Geometry Exists Outside The Protection Frame To Connect To

Fig. 2.9 Terminal Frame Outside Protection Frame

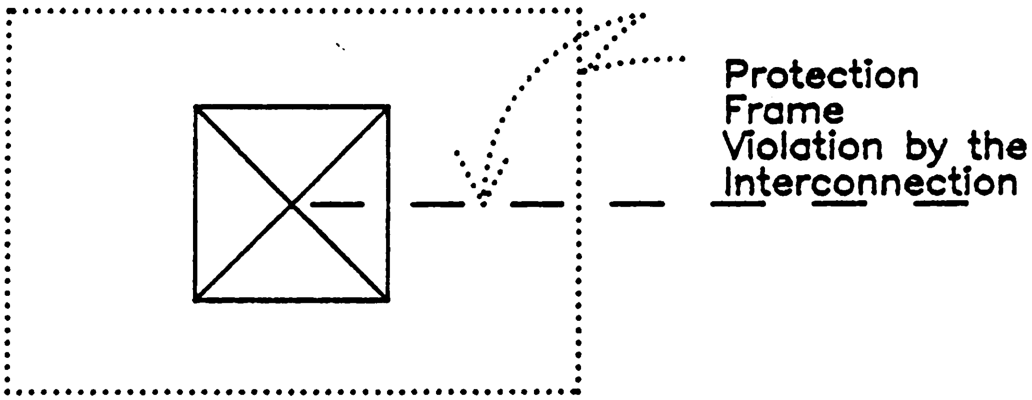


Fig. 2.10 Routing to an Isolated Terminal Frame

with a protection frame to permit external connections to the cell. They may have many edges in common. Objects such as contacts have protection frames and terminal frames that are entirely coincident. An interconnect line may connect to a contact from any one of four sides.

More than one interconnection line may terminate on the same terminal frame. There is no restriction to the total number of interconnections on each terminal frame or even the number of interconnections per side of the terminal frame. However, each interconnect line must end in a terminal frame. This is a convention used to simplify the treatment of interconnections, and is not directly related to their nature.

### 2.3.3. Hierarchical Spacing Using Protection and Terminal Frames

These two simple concepts of protection and terminal frames together provide the basis for a true hierarchical spacing. As each cell is designed, the elements within the cell are spaced according to the layout spacing rules by Python. After each cell is properly spaced, protection frames are automatically generated from the geometry internal to the cell. This is done *only once*, and the protection frames are stored with the cell. The terminal frames are defined explicitly by the user, who labels the signals which are to be exported to the next level in the hierarchy. The geometry that implements these electrical nets defines the physical implementation of the terminal frames. Alternatively, the user can define *local terminals* when laying out his cell. These terminals can automatically be used as terminal frames.

After the terminal frames and protection frames have been established for a cell, the cell can be placed at the next level in the hierarchy. The spacing program only need look at the protection frames of the cell at that level and need never look at the geometries contained within the cell.

The use of protection frames with terminal frames is intended for a bottom-up implementation style with Python. A top-down design style may require additional tools such as interconnect routers. If a cell is placed in its unspaced (original input) form and later compacted, the old terminal locations of the cell may not be at the same locations as the new terminals. The interconnections to the instances of the cell may require patching to physically connect to the now smaller cell (See Fig. 2.11). In the example shown, spacing of the cell yields a cell that is much smaller than the original. Most of the interconnect lines that terminated on the cell now are left unconnected. Three possibilities exist to maintain the physical implementation of the lines connecting to the instances. Since the order of the terminals along any given side of an instance is unchanged (Python does not re-arrange the topology of the layout) a simple program can be used to patch the old terminal locations to the new ones, adding jogs in the interconnection lines if necessary. This tool does not have to concern itself with proper spacing of

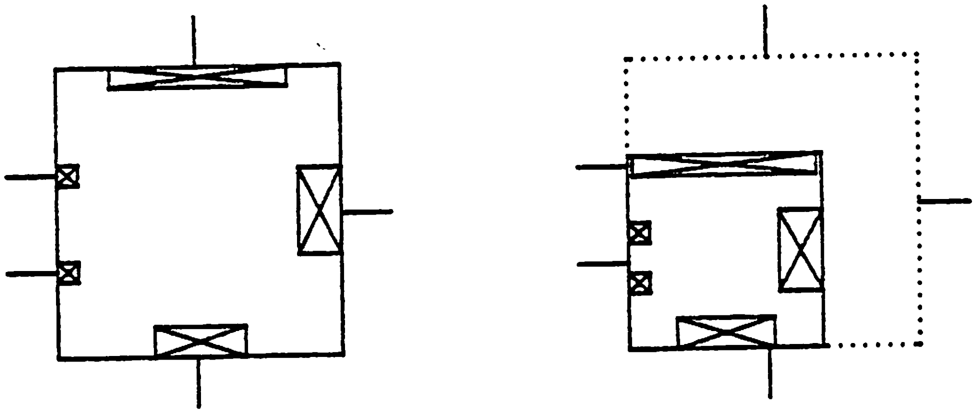


Fig. 2.11 Top-Down Implementation Before and After Spacing

the new lines; Python can resolve any design-rule violations when compacting the cell containing the newly compacted instance. Alternatively, a machine-based routing program or a human designer can patch the interconnections back to the proper terminals in the cell to restore the electrical connections (See Fig. 2.12). Over multiple levels of the hierarchy of a complete IC design,

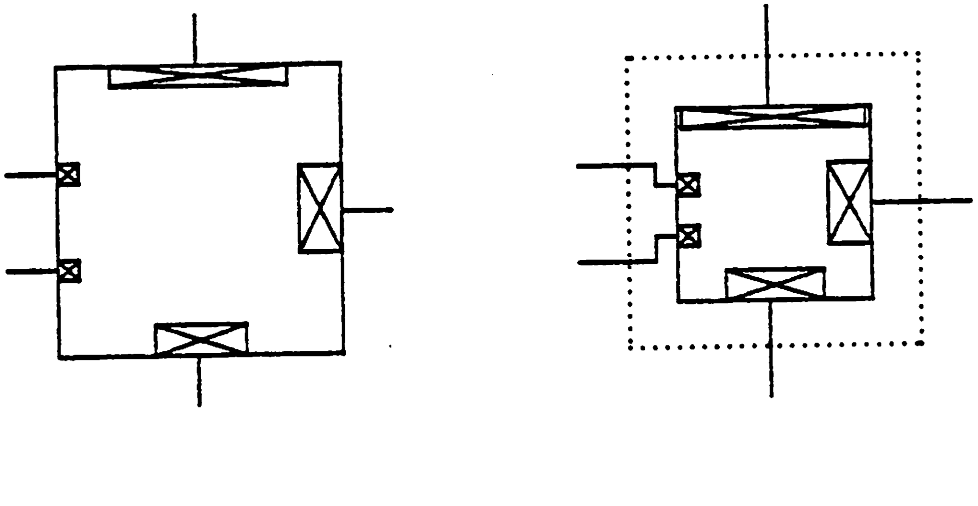


Fig. 2.12 Interconnect Patched to Cell of Fig. 2.11

hand correction of this problem would make the top down design approach intractable. Allowing Python to correct for any design-rule violations produced removes any difficulty associated with a top-down implementation.

An even more serious problem exists if the result of the spacing of a cell instance is *larger* than the original cell (See Fig. 2.13). At this point, the interconnect lines are violating the protection frames of the cell and must be moved out to properly connect to the cell. There may not be enough space to accomodate the expanded cell. A spacing of the current level in the hierarchy may be necessary in order to satisfy all design rules. Since this will expand the current level of the hierarchy, this single cell expansion could ripple up the entire hierarchy to the top chip level. Conceivably, change in a single low level inverter cell could require the re-spacing of the entire IC unless precautions are taken during the top-down design phase.

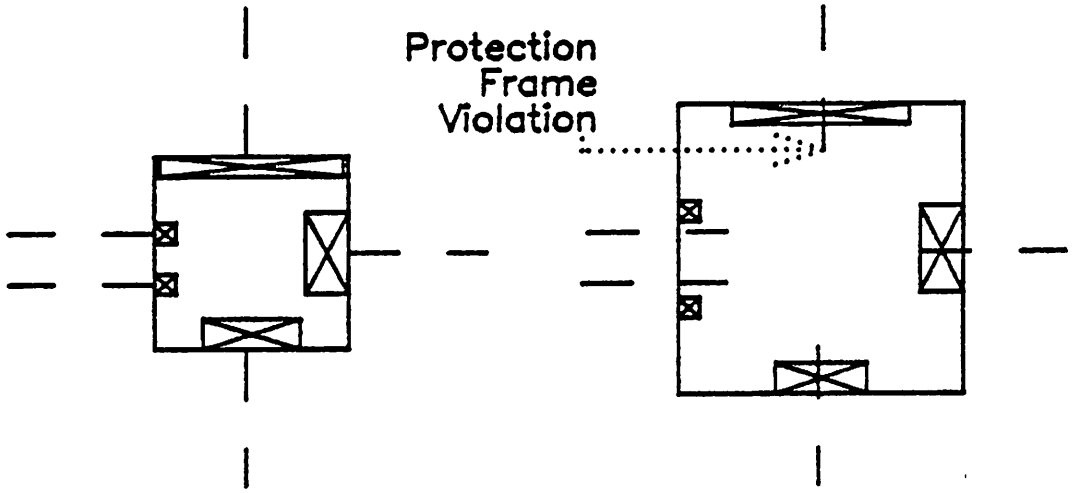


Fig. 2.13 Cell Which Grows - Before and After Spacing

### 2.3.4. Overview of Python

Python is based on the original CABBAGE program. It includes most of the enhancements described above. It uses the general concepts of representing the circuit topology with a graph and solving a longest path problem through the graph to space the IC elements. However, the mapping of the IC topology onto the graph and the algorithm used to solve the longest path through the graph are quite different from the CPM method used in CABBAGE. Some items specific to the Python program are:

- Arbitrary Complexity Polygon Protection Frames (A set per mask layer)
- Terminal Areas (Frames)
- Box (only) Terminal Frames
- Constraint Edges with Upper and Lower Bounds

The specific implementation of the Python algorithms is described in the following chapter.

# CHAPTER 3

# Python

## 3.1. Python as a Part of an IC Design System

While *Python* can be used as a stand-alone utility, it is designed to be incorporated as an integral part of a complete design system. To communicate with other tools in this design system, a comprehensive database is necessary, capable of efficiently representing and managing the information required for the symbolic design of integrated circuit layouts. Several unique types of information are required for the spacing process performed by Python. Electrical circuit connectivity must be known to distinguish between objects that are physically tied together (expressing an electrical connection) and objects that are merely touching or overlapping (perhaps in violation of a layout spacing rule). The geometric construction of cells is also required and provides the means for determining the minimum allowable separation between objects with a given set of layout spacing rules. The physical placement of the cells relative to one another establishes a precedence in the topology of the circuit. The remainder of this section describes the environment in which Python resides and the other programs that it uses.

### 3.1.1. The Squid Database

The *Squid* database [Keller82] provides a general framework to support the requirements of Python as well as other layout programs. At the same time, Squid is tailored for the representation of integrated circuit data so as to allow efficient management of this information.

Squid provides a procedural interface to a general-purpose file system where different *views*, or representations, of a circuit are stored. It allows an application program, such as Python, to create, alter, and delete these views, as well as create, alter, and delete logic and geometric information within individual views. The many Squid operations used by Python are described in Section 3.2.2. This section describes the transformation of the symbolic data from the database into the Python internal format.

### 3.1.2. The Fang Manhattan Polygon Package

As described earlier, *protection frames* and *terminal frames* provide the basis for a true hierarchy in the spacing system. It is important that these frames can be generated automatically. If the user were required to specify these frames, iterations on the design of a single cell would require respecification *by the designer* of the protection and terminal frames and such a process would be error-prone as well as tedious. The *Fang* program [Moore82] is a Manhattan polygon package which can be used to generate automatically protection and terminal frames through a sequence of grow, merge, and shrink operations on the individual mask geometries. Python also uses Fang to remove any overlaps which might be present in protection frames retrieved from the Squid database. Such overlaps would prevent a legal circuit spacing from being generated.

### 3.1.3. The Hawk Graphics Editor

With the ability to represent comprehensive symbolic information in the Squid database, this information must be entered into the database graphically or under program control. In *CABBAGE*, the *GRLIC* program is a graphics editor which provides rudimentary graphics entry and editing capabilities of the symbolic primitives. For Python, a more powerful entry system is required, both to provide the basis for a *system* that is more useful in a production sense, and to allow the designer to exploit the added enhancements in Python. In addition, this entry system should provide a clean, simple interface to the spacing program, shielding the designer from the tedious parts of the spacing process.

The *Hawk* graphics editor [Keller82] is the front-end for Python. It has many powerful features which ease layout for designers.

## 3.2. Algorithms Used in Python

### 3.2.1. General Overview

Python properly spaces interconnected cells which represent the mask topology of an IC. This spacing is a process of shrinking or expanding the lengths of the interconnection lines between these cell instances to obtain a minimum area for the entire layout. Electrical connectivity is preserved at all times. The semiconductor process defines a set of spacing rules (minimum spacings between mask layers) which are used to determine the minimum allowable separation between cells. Each cell instance is represented by a set of protection frames and terminal frames on many mask layers. At the lowest level, the actual mask geometries of the primitives will define their protection and terminal frames. Additional frames may be introduced to express more complex design rules (such as transistor active area to active area spacing in an NMOS process). At higher levels in the hierarchy, the protection frames and terminal frames are generated to produce the optimum computation time/area saving tradeoffs. The widths of interconnections are also taken into account during the spacing process.

Taken as a whole, this guarantees a compacted layout which satisfies all spacing rules for a given process, and which occupies a minimal amount of area given the latter constraint.

The spacing process is decoupled into separate problems for the X and Y directions. This allows the program to use the orthogonal structure of the instances and lines to best advantage. One point worth noting is that this decoupling introduces the possibility of the minimal area derived by the program being a local minimum, and larger than the global minimum area. In CABBAGE, this effect was often observed. Also, the initial direction of spacing has a large effect on the aspect ratio of the final layout. That is, an initial X spacing might yield a tall narrow cell whereas an initial Y spacing would yield a short wide cell.

For each spacing direction, each instance and interconnect line is represented by two vertices in a graph, which represent the coordinate positions of the lower and upper sides of the bounding box surrounding each instance and line. This bounding box is for use as a reference only and has no bearing on the complexity or number of protection frames used within each instance. Edges in the graph have upper and lower bounds, which represent the smallest and largest distances (orthogonal distances because of the decoupling into X and Y graphs) respectively between the locations represented by its source and sink vertices. Edges are added to each graph to preserve the shape of the instances and width of the lines. Additional edges are added to preserve the electrical connectivity. Next, edges are added between vertices in each graph to indicate the spacing requirements between two objects. The result is a complete constraint graph, with vertices corresponding to the physical locations of the instances and lines, and with edges corresponding to the spacing constraints necessary to preserve shape and electrical connectivity, as well as keeping objects spaced apart by the proper spacing rule. Figure 3.1 shows the edges and vertices for the graph that is generated for a single object. The source and the sink of the graph are global.

This graph is now processed using a modified Critical Path Method (CPM) algorithm. Finding the longest path from the source of the graph (lower edge of the topology) to the sink (upper edge of topology) yields the spacing rule correct positions of the vertices of the graph (and hence the instances and lines they represent). This method itself is iterative at two levels. The modified CPM algorithm is iterative, requiring as many iterations as the number of vertices in the graph in the worst case. Since the spacing process is decoupled into separate X and Y spacings, layout rules may be violated during a spacing in the direction perpendicular to the spacing. Thus, it is also necessary to iterate between X and Y spacings until no instance moves

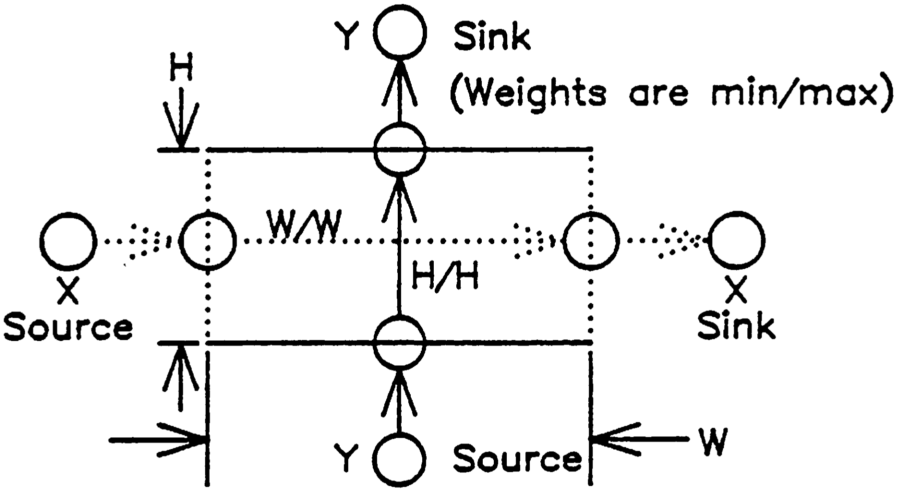


Fig. 3.1 Edges and Vertices for a Single Cell

for both an X and Y spacing. Only then is a completely error-free layout guaranteed. Changes during a spacing iteration may change the spacing requirements in the perpendicular direction. This requires that the edges used to express spacing constraints be recomputed during each iteration.

The algorithms in Python are derived from those used in CABBAGE. The graph representation of the topology is slightly different, due to the introduction of sliding contacts and polygonal protection frames. The spacing rule analysis is performed on an edge-segment by edge-segment basis, rather than on an object-by-object basis, as is done in CABBAGE. This is necessary because of the complex shapes that protection frames can have. The longest path analysis is modified to allow *maximum* constraints, as well as the minimum constraints generated in CABBAGE. The minimum constraints represent the minimum allowable spacing imposed by the spacing rules. The maximum constraints are used to preserve electrical connectivity, and to preserve the shape of objects. The modifications to the longest path algorithm make it iterative in nature, in contrast to the Critical Path Method used in CABBAGE which is a single-pass algorithm.

The specific algorithms used in each subsection of the program are described below. The Python program is divided into 5 major subsections. They are *Readin*, *Buildgraph*, *Sranalyze*, *Lngpth*, and *Update* subsections.

### 3.2.2. Transforming Data into Internal Form

The *Readin* phase of the program transforms the symbolic IC data from the Squid database into the internal data structures used by Python. Each cell within Squid is described by several *views*, or representations of the cell. The view from which the layout geometric information is read is termed the input view, and the view where the spaced layout is stored is termed the output view. The input view is opened for reading only, and copied to the output view. The output view is then opened for reading, and the input data is read from the output view. This is necessary to facilitate update at the end of the spacing process. The only coordinates that will be updated are the locations of instances and the paths of interconnect lines. An instance *generator* returns a different instance each time it is envoked, until all instances have been returned. Each instance is opened in turn for reading. Squid reads each cell type *only once*, and successive opens of the same cell return immediately.

Next, a *special generator* is invoked to read in all of the interconnect local to the cell being spaced. The only legal geometry for representing interconnect is the geometry-type *line*. Each line has a mask layer associated with it. If the mask layer of a line has no spacing rules to any other layer (including itself), the line is ignored, since no constraints will ever be generated because of it. The orientation of the line is considered next. The line must be either horizontal or vertical to work with the spacing algorithms used in Python. Also, Python requires each interconnect line to be a single segment joining two instances. This requires that special instances be created to jog wires. Each line has four *intervals* created for it. Intervals are edge segments which represent the bounding-box edges of the line. One interval is created for each edge of the bounding box. These intervals contain the mask layer of the line, as well as a *net-id* which defines the electrical net to which the interconnect belongs. They are used for spacing rule analysis. The vertical intervals are used in the X spacing analysis and the horizontal intervals are used in the Y spacing analysis.

The same special generator used above to generate the interconnect is now invoked to generate protection frames and terminal frames both local to the cell being spaced and local to each instance within the cell. Instances are also specified with a bounding box and, like lines, they have intervals for each side of the bounding box. The difference is that instances contain polygonal protection frames on many mask layers. Each interval represents an edge of one of these polygons. Squid currently maintains the polygon protection frames as rectangles within the database. Each rectangle is decomposed into four intervals, which are labelled with the mask number and given a NULL *net-id* (which indicates no connection to any net). As for the case of lines above, the horizontal and vertical intervals are kept separate for use in the decoupled spacing analyses.

Terminals are generated in the same step as protection frames. They too are represented with a bounding box, but have no intervals created since they are not used directly in the spacing analyses. Each terminal has an associated mask layer and *net-id* and points back to the instance to which the terminal belongs. If the *net-id* of a terminal is NULL, it means the terminal is 'floating' (not connected to any line) and it is ignored. As the terminals and protection frames are generated by Squid in a depth-first fashion, transitions from instance to instance are detected. It is at this point that a new internal form is generated for the instance and all successive protection frames and terminal frames are owned by this instance until the next transition occurs.

After all instances and lines have been read, some conditioning is necessary before the analysis can proceed. There may be overlaps between the polygonal protection frame edges since they are represented in Squid with rectangles. The Fang [Moore82] polygon package is used to remove any such overlaps, generating true polygons for the protection frames. At the same time, Fang determines the contour of the corners of the intervals. This information is necessary to properly handle corner-to-corner constraints in the spacing rule analysis.

Each interval has a *net-id* associated with it. For each interval, all associated terminals are checked to see if they share a common edge. If so, the part of the interval which overlaps the terminal is split from the rest of the interval and labelled with the *net-id* of the terminal. Using *net-ids* with intervals allows the spacing rule analyzer to ignore rules between intervals on the same mask layer with the same *net-id*. Although a constraint must be added to keep the topology of the circuit from changing, the value of the constraint is adjusted to allow the two intervals to touch.

Intervals must have their coordinates specified relative to the bounding box edges of the instance to which they belong. This removes the need to update the interval coordinates since they will always be correct. Even lines, which change length, will always have the correct values for their interval coordinates since the endpoints of the intervals along the length of a line are expressed relative to the endpoints of the line itself.

### 3.2.3. X and Y Graph Construction

During the *Buildgraph* phase of the program, the 'permanent' parts of the X and Y graphs are constructed. These are the parts which are static through the iterative X and Y spacings. The vertices which represent the locations of the 4 edges of the bounding box for each instance or line do not change. The edges added to preserve shape, width, and electrical connectivity, are also invariant. These parts provide the basis for the constraint graphs.

The *Readin* phase created lists of instances, lines, and terminals. Terminals contribute no vertices to the graph but are used to determine the upper and lower bounds of the edges added between instances and the lines connecting to them. Since the terminals are areas instead of points, any line whose width is smaller than the terminal to which it connects generates an edge with different upper and lower bounds. The provision of these 'sliding interconnects', which can connect to a terminal within a range of positions, allows the spacing program to take advantage of non-critical connection points and achieve a smaller overall size for the integrated circuit. For each instance, two vertices are allocated in each of the graphs. In the X graph, an edge is added between the right and left vertices with the same upper and lower bound, which is the width of the bounding box. This also occurs in the Y graph, between the bottom and top vertices. Now the bounding box of the instance will maintain its shape during the spacing process. Note that a single vertex in each graph would have been sufficient to represent the instance, since its shape will not change. Two vertices were added since they are necessary for lines, and it was desired to treat lines and instances in a consist manner in the constraint graph.

A line requires two vertices in each of the X and Y graphs because its endpoints can move independent of one another. The shrinking and growing of interconnect lines along their length provides the compaction - instances only change their location, they do not change their shape. Note that three vertices spread between two graphs would be sufficient to represent lines since either the X values are always equal (vertical lines) or the Y values are always equal (horizontal lines). Two vertices were used in each graph, instead of two in one and one in the other, so that the algorithms need not consider the orientation of lines. The edges of a line are expanded to include the width of the line by the time spacing rule analysis has been performed. To maintain the orthogonal nature of the lines, an edge is added between the vertices of the line in the graph perendicular to the orientation of the line. This edge has fixed and equal upper and lower bounds equal to the width of the line. It would not be necessary if three vertices were used to represent the line instead of four but simplicity of the algorithms was deemed more important than saving a small amount of space.

Each line has an additional edge added between its vertices in the graph parallel to its orientation. This edge has a lower bound of zero and an upper bound of infinity and serves to keep the upper coordinate (in the axis parallel to its orientation) greater than or equal to the lower coordinate. The idea of *coverage* (described in Section 3.2.4) will not work if the topology of the symbolic IC can change. One possible change occurs when a line 'changes polarity', *i. e.*, the length of the line defined as the upper coordinate minus the lower coordinate becomes negative. Since this would destroy the integrity of the spacing process it is forbidden. Addition of this edge with zero lower bound and infinite upper bound keeps the line lengths greater than or equal to zero.

Edges are now added to the graphs to preserve electrical connectivity between instances and the lines which interconnect them. Terminals, although not directly represented in the constraint graphs, determine the upper and lower bounds of the edges which bind instances and their lines together. The terminals and lines are individually sorted into their respective lists by *net-id*. Each terminal is compared only with lines of the same *net-id* to prevent the $O(n^2)$ time complexity that would occur if each terminal were checked against each other line. There will still be a problem with global signals, which are likely to have many terminals and lines in the same net. Examples of this are *VDD*, *GND*, and *CLK* signals in an integrated circuit.

Terminals which are checked against lines are compared to see if either of the line's endpoints is contained within the terminal. If one or the other endpoint *is* contained within the terminal, two constraint edges are added, one to each of the graphs. This fixes the vertices representing the instance of the terminal to the vertices representing the endpoint of the line contained within the terminal. Since the terminals are area frames rather than points, the edge added to the graph perpendicular to the line orientation is given some 'slop', by means of different, but finite, lower and upper bounds on the constraint. These bounds are fixed to allow the lower edge of the line to drop as far as to coincide with the lower edge of the terminal frame and to allow the upper edge of the line to rise far enough as to coincide with the upper edge of the terminal frame. Obviously, the smallest width a terminal frame can have is the width of the line that is connected to it. In this case, these upper and lower bounds would be equal and the interconnect line would be coupled rigidly with the instance of the terminal.

The graph parallel to the orientation of the line has an edge added to it to fix the endpoint of the line to the edge of the terminal frame with which it intersects. This edge is added to prevent terminals frames from merging. Objects should be allowed to merge, but this is a subject which needs extensive algorithmic development for the compaction model used in Python before a suitable method can be determined.

Two additional points should be noted about terminals. First, more than one line may end on a single terminal. This will not cause any problems *unless* the sum of the widths of the wires is greater than the width of the terminal. In the latter case, an overconstraining condition will be generated and the layout cannot be spaced correctly. The second point worthy of note is that the two or more lines ending on a single terminal do not have to enter from a single side if the protection frames will permit entry from multiple sides. A clear example of this is a contact between two mask layers. In a contact, the protection frame and the terminal frame coincide on all four edges. Assuming the two mask layers may overlap in an arbitrary fashion without violating layout rules (such as for polysilicon and metal in an NMOS process), there may be lines entering the terminal from all four sides on each mask layer.

At this point, all of the vertices for instances and lines have been created and all of the edges for preserving instance shape, line width, line length (greater than or equal to zero), and electrical connectivity have been added to these vertices. This constitutes the complete 'permanent' portion of the graph. All of these parts will remain unchanged throughout the iterations of spacing rule analyses and the determination of the longest path. The only parts to be added are the edges which express minimum spacing requirements between objects. These edges must be generated before each iteration. The reason for this is that movement of an instance or line in one direction disturbs the edges added for spacing *in the perpendicular direction*. Therefore, before each iteration, the graphs must be returned to the 'permanent' state and the spacing rule analysis will add edges for the current iteration. Notation is made of this 'permanent' state immediately after the graph is constructed. This makes the process of returning the graph to the proper state very fast.

The last thing done in the *Buildgraph* subsection is to generate the two interval queues that will be used in the spacing rule analysis. Since the interval coordinates are expressed relative to the appropriate bounding box edges of their instance or line, it is not necessary during the iterative loop of spacing rule analyses and longest path solutions to update the intervals coordinates. This is true for instances, which only change location, and it is also true for lines, which change shape (length) as well as changing location.

Now the information has been generated to allow the iterative loop of spacing rule analysis followed by longest path solution to be executed.

### 3.2.4. Element Spacing Rule Analysis

In Python, each of the four sides of an object has a per-layer set of protection frame edges associated with it. These edges, called *intervals*, are developed from the protection frames as the symbolic data is read from the Squid database. They consist of the edges of the polygons that form the individual protection frames on each mask layer for the cell. They are formed during part of the *Buildgraph* phase, and are used in the spacing rule analysis.

The idea in the spacing analysis is to compare all right-side edges against all overlapping left-side edges and determine the magnitudes of the constraints necessary between objects. Each object has only two vertices to represent it in each coordinate axis direction, but has many intervals. Only a single constraint is generated between any two objects. Each *primary* (right-side) edge (P) of an instance or line is compared with each *neighbor* (left-side) edge (N) of an instance to the right of it. The interval constraint which forces the maximum separation between the two instances defines the minimum allowed spacing.

First, the intervals are sorted by center coordinate, with conflicts in center coordinate being further sorted by lower coordinate. This list is then traversed in reverse order. Left-side edges are ignored in favor of *primary* edges, or right-side edges. When a primary edge is found, the interval list is scanned forward from this point, in search of a left-side edge, or *neighbor* edge that overlaps the primary edge. Several features make this search efficient. If the neighbor edge is below the primary edge a distance greater than the maximum spacing rule, there cannot be any interaction between the neighbor edge and the primary edge. The next neighbor edge is then found. If the neighbor edge is above the primary edge a distance greater than the maximum spacing rule, there can be no interaction between the primary edge and the neighbor edge. Furthermore, since the intervals are sorted in two directions, no further neighbor edge with the same center coordinate can possibly overlap the primary edge. Thus, these neighbor intervals can be skipped at substantial computational savings.

Once it has been determined that a primary edge and neighbor edge might overlap, there are several cases to ignore. The first is when both the primary edge and neighbor edge belong to the same instance. Since the primary edge is a right-side edge, and the neighbor edge is a left-side edge, overlapping intervals may belong to the same instance only if they are interior to a concavity in a protection frame of if they belong to protection frames on different mask layers as illustrated in Figs. 3.2 and 3.3. These cases of overlapping intervals should be ignored, since it is assumed the construction of the instance protection frames is correct.

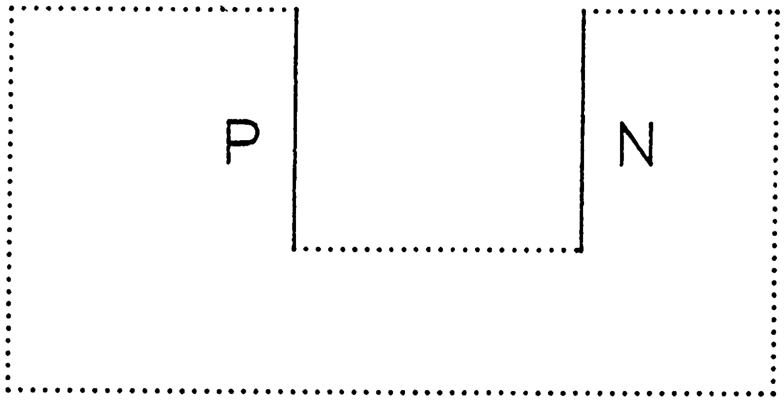


Fig. 3.2 Primary and Neighbor from Same Instance (Same Mask Layer)

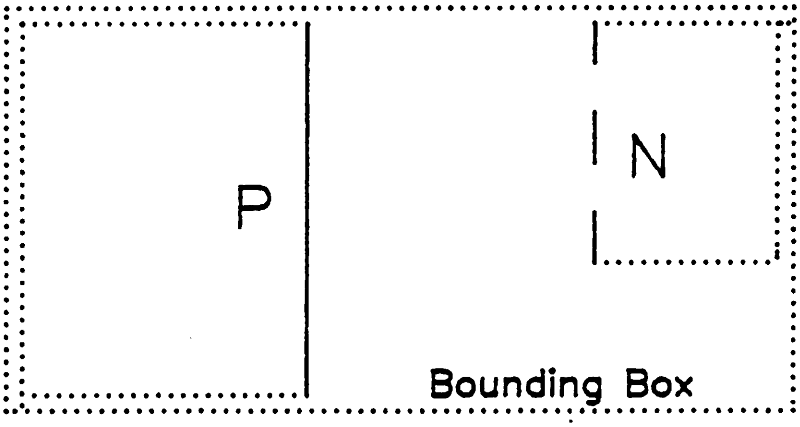


Fig. 3.3 Primary and Neighbor from Same Instance (Different Mask Layers)

Another case to ignore occurs when there exists no rule between the mask layers of the primary and neighbor intervals. Since there is no rule, the two intervals are not constrained relative to one another. They may even overlap if the rest of the circuit topology permits.

If the primary and neighbor instances have a 'permanent' edge (one preserving electrical connectivity or instance shape), the spacing rule between them should be ignored. Otherwise, overconstraints are easily introduced into the constraint graph, and no solution can be attained.

If the overlap between primary and neighbor edges is not ignored for one of the above-mentioned reasons, the two intervals are checked more closely for overlap. The two types of overlap are solid overlap and corner overlap and are illustrated in Figs. 3.4 and 3.5. Solid overlap occurs when the themselves overlap. Corner overlap occurs when the intervals do not directly overlap, but the distance by which they miss is less than the spacing rule between the mask layers of the two terminals. Corner overlap is necessary to prevent overconstraining conditions from being generated. Since the spacing rule analysis is decoupled, the spacing rules in the direction perpendicular to the direction of spacing are ignored. This means that relative movement of instances and lines during a spacing may violate design rules in

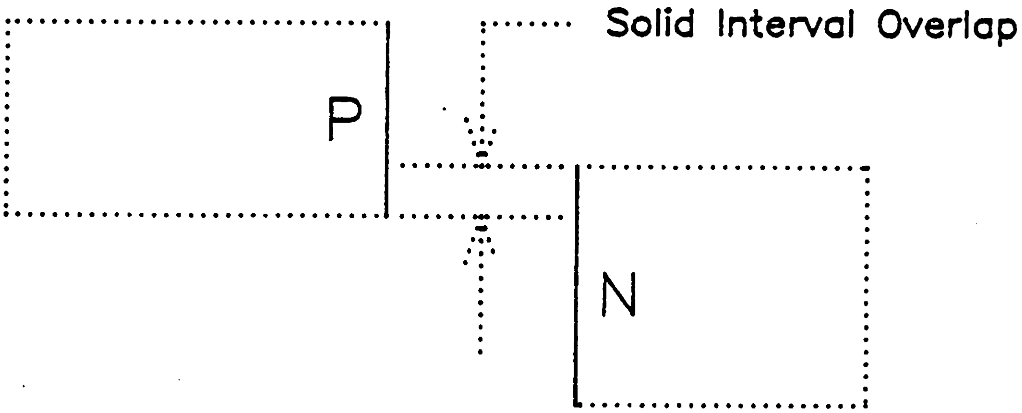


Fig. 3.4 Solid Interval Overlap

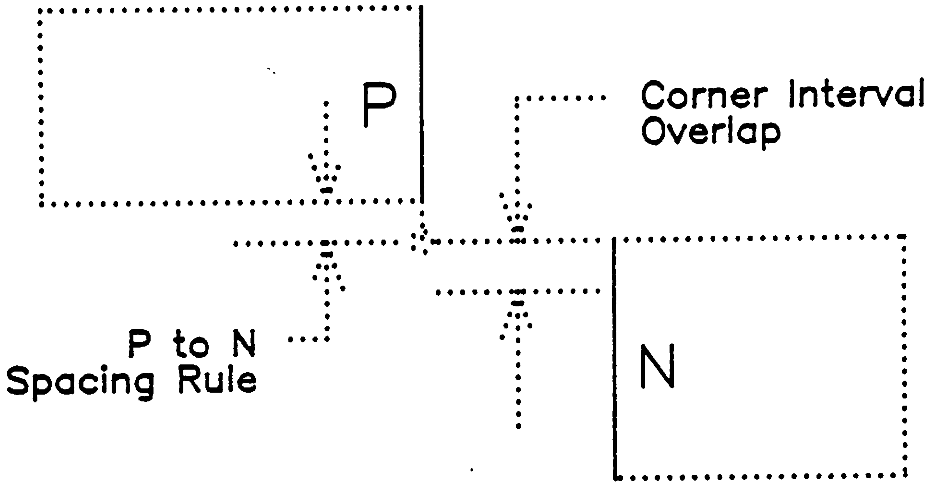


Fig. 3.5 Corner Interval Overlap

in the perpendicular direction. Also, if corner constraints are ignored, it is possible to generate a compacted layout that cannot be legally solved in the perpendicular direction, as shown in Fig. 3.6. The situation in Fig. 3.6 is corrected if corner constraints are applied, as illustrated in Fig. 3.7.

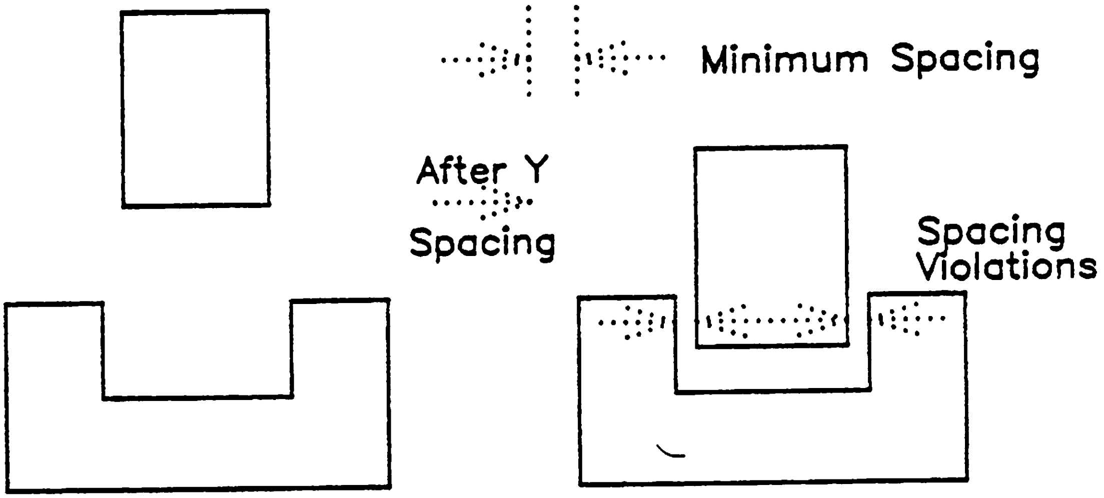


Fig. 3.6 Overconstraint Introduced When Corner Constraints Ignored

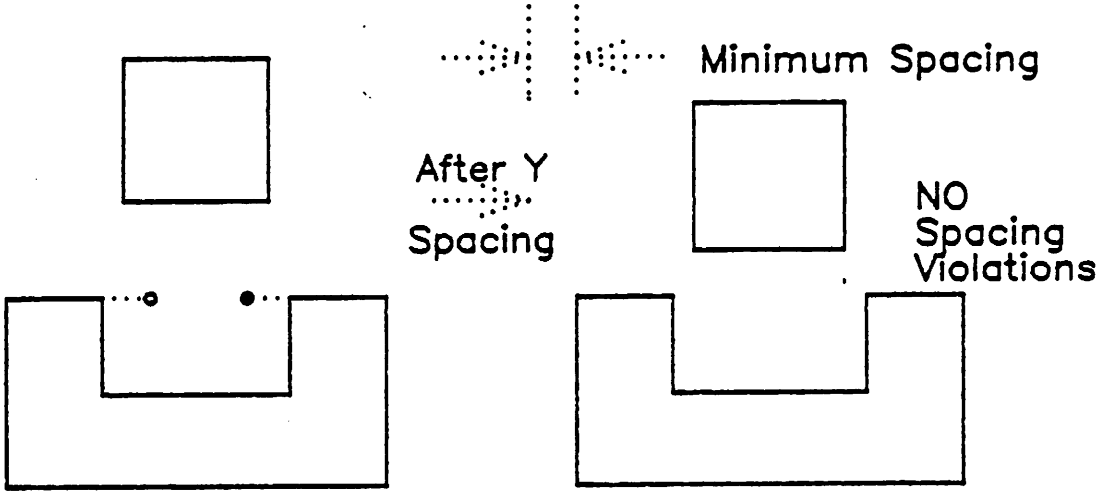


Fig. 3.7 Overconstraint Avoided When Corner Constraints Checked

If either a solid or corner constraint is necessary, the vertices representing the center positions of the primary and neighbor intervals are checked. If a constraint is already present between the instances which contain the primary and neighbor edges, the value of the constraint is updated to the maximum of its current value and the new required spacing imposed with the current primary and neighbor edges. If no constraint is present, one is added.

The concept of *coverage* is used to limit further the interval search. Give $m^2$ objects arranged in an $m*m$ matrix, in the worst case, where there is a spacing requirement between each object and every other object, there are $\frac{m^2(m-1)}{2}$ constraints required [Hsueh79]. Given $n$ objects, the average case would then generate $n^{1.5}$ constraints and the worst case, with all $n$ objects in a line, would generate $n^2$ constraints. The idea of coverage is that a constraint generated between a primary edge and a neighbor edge shields the overlapping interval of the primary edge from successive neighbor edges with higher center coordinates. Implicit in the concept of coverage is the assumption that a right-side edge of the neighbor's instance will generate a constraint with successive neighbor edges (See Fig. 3.8). Thus in Python, each interval is covered by the solid overlap of a neighboring interval that causes a constraint to be generated. This works properly for NMOS and Bipolar IC processes. However, there can be a problem, as illustrated in Fig. 3.9. If there are three objects, on mask layers A, B, and C, and there are spacing rules between mask layers A and B, mask layers A and C, but *not* mask layers B and C, an improper cover is generated, as seen in the figure. The primary edge in object 1 is covered by its neighbor edge in object 2, and there is no constraint generated between objects 1 and 3. Since there is no spacing rule between mask layers B and C, no constraint is generated between objects 2 and 3. Thus, object 3 is free to move relative to object 1. The subsequent spacing process may overlap objects 1 and 3, in violation of the spacing rule between mask layers A and C. Thus it is important to verify for new IC technologies that the spacing rules are defined to prevent this case. Not using

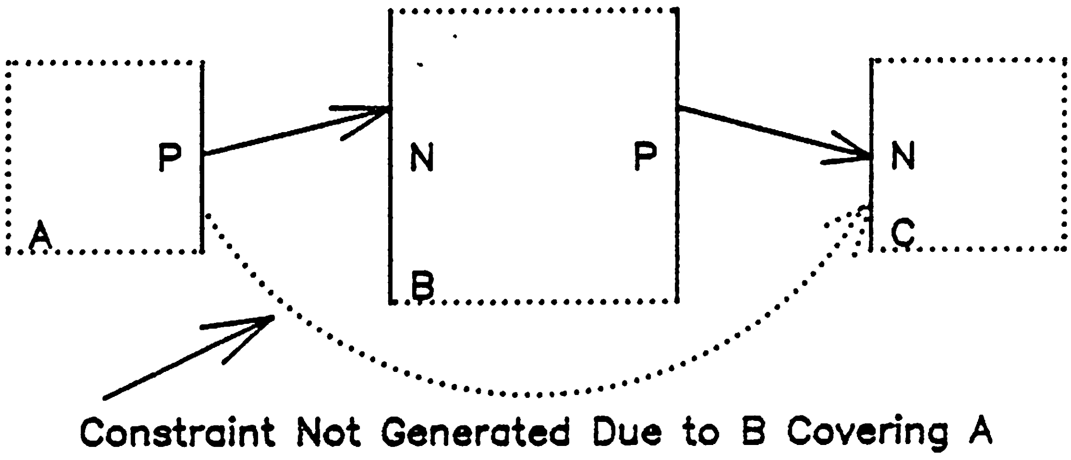


Fig. 3.8 Coverage of a Primary Interval

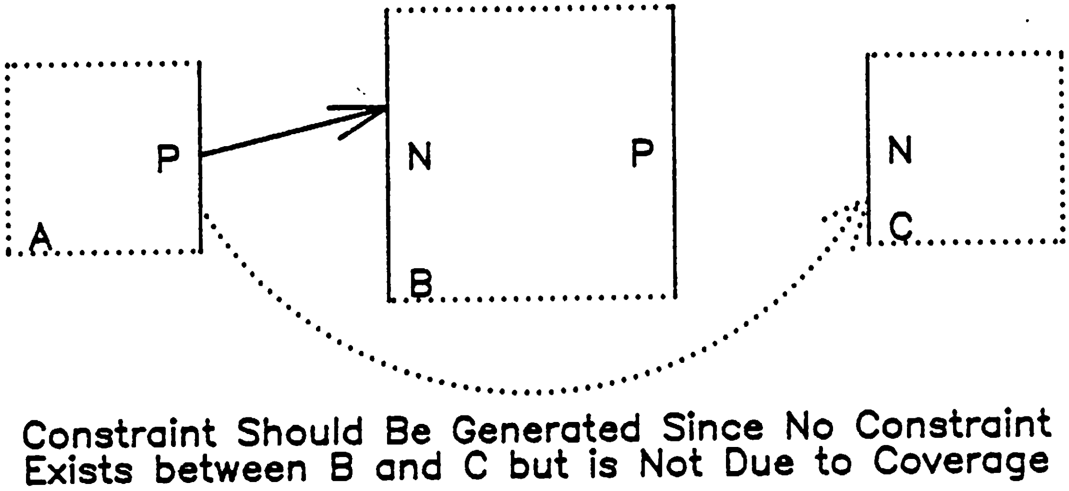
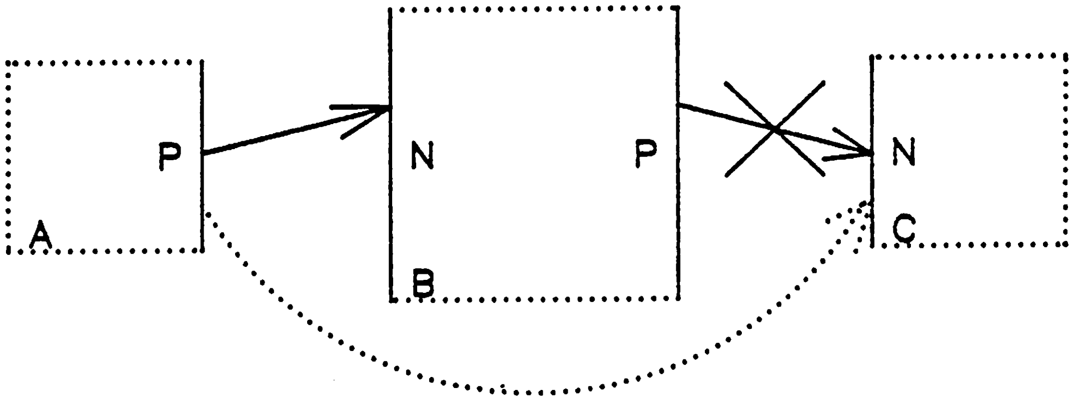

P
A
N
P
B
N
C
Constraint Should Be Generated Since No Constraint Exists between B and C but is Not Due to Coverage

Fig. 3.9 Improper Coverage of a Primary Interval

coverage would change the order dependencies of the spacing rule analysis from $O(n)$ to average $O(n^{1.5})$, and worst case $O(n^2)$. These super-linear order dependencies quickly increase analysis time to an impractical size for real cells.

A concept which also increases the order dependency of the analysis, but one which is necessary for area efficient spacing, is the concept of mergability. Objects are not permitted to merge in Python. If two intervals overlap and a spacing requirement exists between them, the spacing is allowed to drop to zero provided the two intervals are in the same electrical net and on the same layer. Thus two lines connected to a single terminal could have zero spacing without violating any spacing rules. Under these conditions, the possible overlap resulting from the inaccuracies of the IC process (the basis for the spacing rules) would not damage the electrical performance of the circuit. If the lines were on two different mask layers, polysilicon and diffusion for example, the results of ignoring spacing rules could be disastrous even if the two lines were electrically equivalent. In a similar way, ignoring the spacing rules between object on the same mask layer but in different electrical nets could quite easily destroy the electrical properties of the circuit. Therefore, only the spacing rules between intervals on the same mask layer that are electrically equivalent are ignored, by setting them to zero.

Rather than set the spacing requirement to zero for objects that are in the same net, they should be allowed to *merge*, which means that no spacing requirement should be added to the constraint graph in the first place. There are several problems with this. Given three objects A, B, and C (See Fig. 3.10) if B, and C can be merged, then no constraint is added between them. There is a spacing requirement between objects A and B which keeps them properly spaced. There should also be a constraint between objects A and C, but object B covers object C, so this constraint is not generated. During the subsequent spacing process, object C overlaps object A, in violation of spacing rules (See Fig. 3.11). The program should recognize that objects B



Constraint Between A & C Should Be Generated Since B & C Can Merge and Have Rejected Constraint

Fig. 3.10 Objects B and C can Merge

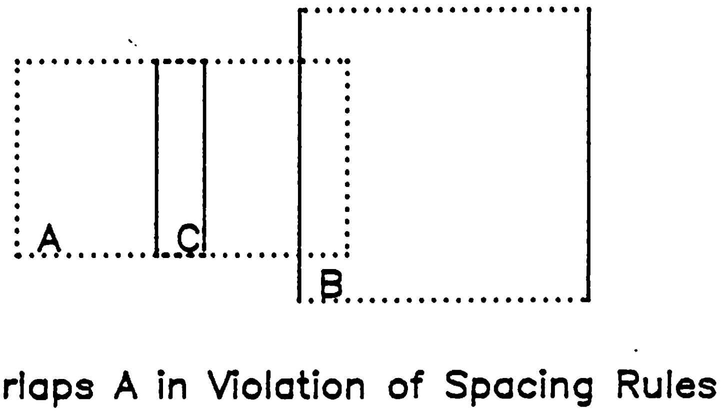


Fig. 3.11 Possible Result of Spacing Fig. 3.10

and C cab be merged, and hence could change relative placement during the spacing process, *when the spacing requirement between objects A and B is determined*. Although this is trivial for the case with simple rectangles, it is not generally trivial for the case when arbitrary rectangular polygons are used as protection frames, as shown in Fig. 312. What is necessary is to store

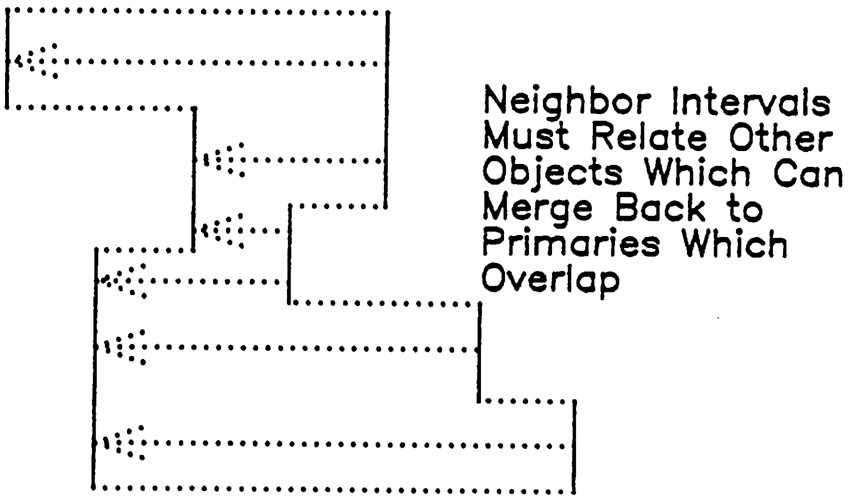

Neighbor Intervals Must Relate Other Objects Which Can Merge Back to Primaries Which Overlap

Fig. 3.12 Problems with Merging of Arbitrary Rectagons

the information as to the mergability of a right-side interval with it corresponding left-side interval(s). This is not easy to do in a general fashion. Thus, Python does not allow objects to merge, with some loss in the area efficiency of the program.

### 3.2.5. Element Placement

At this point, the entire graph representing the locations of the instances, with constraints for spacing rule separation, preservation of electrical connectivity, and preservation of instance shape and line width, has been constructed. Next, to insure that the graph is connected and to provide a starting point and ending point, a graph source and sink are defined and added to the graph. Each vertex with no predecessors has an edge added to it directed from the graph source to the vertex, with lower bound of zero and upper bound of infinity. Each vertex with no successors has a directed edge added to it directed from the vertex to the defined graph sink, also with lower bound of zero and upper bound of infinity. The graph is now connected.

CABBAGE requires the constraint graph to be acyclic. This requirement is necessary for solution with the Critical Path Method. Python only requires that cycles be of non-positive weight. Given the Polygonal protection frames of Python, cycles in the constraint graph are legal (Shown in Fig. 3.13). In this case, the two interior intervals of the U-shaped polygon A generate constraints between the exterior intervals of the box B. Since the right-side intervals are referenced to the right-side vertices and *vice versa*, a legal cycle is generated for the X directed graph. The cycle runs from the left side of polygon A to the right side of polygon A (an edge added to preserve the width of polygon A), from the right side of polygon A to the left side of box B

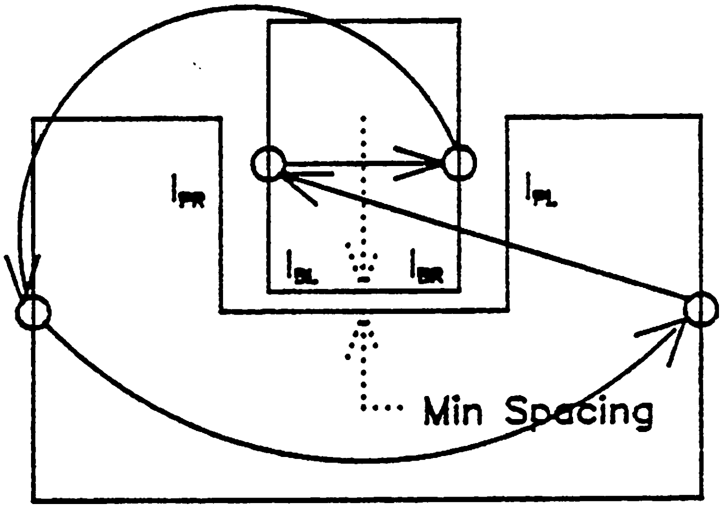


Fig. 3.13 Legal Cycle in Constraint Graph

(an edge added to express the spacing constraint between interval $I_{pr}$ and interval $I_{bl}$), from the left side of the box B to the right side of the box B (an edge added to preserve the width of box B), and finally from the right side of box B to the left side of polygon A (an edge added to express the spacing constraint between interval $I_{br}$ and interval $I_{pl}$). Thus, the algorithm used in Python does not require an acyclic graph.

The solution of the Python constraint graph with upper and lower bounds on the lengths of the edges is derived from the longest path problem in graph theory. What is needed is the minimum value of the positions of all instances and interconnect lines (vertices in the graph) subject to satisfying the minimum spacing requirement between objects and preserving the electrical connectivity of the circuit as well as preserving the shape of instances and the width of lines. (Lines are allowed to stretch and shrink - this is how the final spacing is performed.) See Appendix D for a derivation of the algorithm used in Python. It is an iterative algorithm, and is presented in a psuedo-programming language which follows the control structures of the C programming language [Kernighan78].

```
1)      Schedule all Vertices to be Examined.

2)      Make the position of each scheduled vertex the maximum of:
                Its current position
                The maximum of its predecessors current positions
                 plus the lower weight of the edge joining them
                The maximum of its successors current positions
                 minus the upper weight of the edge joining them
        If a Vertex Changes:
                For Each of Its Predecessors and Successors
                        If that Vertex Is Not Scheduled
                                Schedule for Current Iteration
                        Else If that Vertex WAS Scheduled
                                Schedule for Next Iteration
                        Else If that Vertex IS Scheduled
                                Do Nothing

3)      If the Next Iteration Queue is Empty
                Converged (=> DONE)
        Else If the Iteration Count is Greater than the Number of Vertices
                An Over-constraining Condition has been Encountered
        Else
                Goto Step 2) and Continue Iterating
```

This algorithm is guaranteed to converge in $v$ steps, where $v$ is the number of vertices in the graph. If an overconstraining condition (positive cycle) exists in the graph, the existence of this overconstraint is detected when convergence is not reached on the $(v + 1)st$ iteration. Unfortunately, detection of the *location* of the positive cycle is an $O(2^v)$ problem. The time required for detection of the positive cycles becomes intractable for even a small number of vertices. For this reason, Python only reports the existence of overconstraining conditions. The program makes no attempt to discover their location. A related point is that the solution of the graphs generated for representing IC topology in Python is well behaved, *i.e.*, the number of iterations for convergence is typically a small fraction of $v$, the number of vertices in the graph. If an error is present in the layout, Python takes substantially longer to detect its presence than the program does to solve the equivalent graph without the error. Because the solution of the constraint graph generally takes fewer steps than the maximum, Python provides a maximum iteration count. If the iteration count rises above this value, the spacing process is terminated. At this point, the program may be run again with this maximum count disabled to determine if the input actually contains an overconstraint. Alternatively, the user may examine the input topology in a graphics editor to apply his heuristic knowledge in search of the overconstraint.

Once the longest path analysis has been performed, all spacing rules are satisfied. There is one (or many equivalent) *critical paths* through the graph. A critical path is defined as a path from the graph source to the sink where the difference in the positions of each pair of vertices on the path is equal to the lower bound of the edge connecting them. Note that although the spacing between two vertices in the graph is at the lower bound of the edge connecting them, they will not be on the critical path unless they lie on a path from.source to sink where *all* of the edges are at their lower bound length.

At this point, the IC topology has the minimum size. The problem with the solution is that all objects are at their minimum coordinate location possible while still satisfying design rules. This has the observed effect of 'pulling' objects to the left or bottom side of the circuit. While this effect is not a problem for the objects on the critical path (these objects could not have there positions changed without increasing the overall size of the cell or violating a spacing rule) there is 'slack' in the spacing among objects not on the critical path. This slack is distributed on the upper side of groups of connected objects not on the critical path. If the slack were distributed more evenly, the geometric yield of the IC would be improved since fewer objects would be at their minimum required spacing apart. To accomplish a more equitable slack distribution, a *reverse pass* is made through the graph to determine the maximum coordinate locations of all objects while still satisfying all spacing rules. The critical path objects can now be determined easily by noting which have the equal lower and upper positions. The rest of the objects have different lower and upper positions, which define the 'slack' around each object. Note that this 'slack' may be shared among other objects not on the critical path; if more than one object is on a path not on the critical path, the slack belongs to both of them.

If the average position is taken for each object, it can be shown that all spacing rules are still satisfied, if the spacing rules are satisfied when all objects are at their lower positions and when all objects are at their upper positions. If there is a single object between two objects on the critical path, the non-critical object is properly spaced in the middle of the slack space (See Fig. 3.14). If there is more than one group between two groups on the critical path, this averaging technique has the effect of bunching the non-critical path groups together in the center of the slack space (See Fig. 3.15). Although this is not optimum, the averaging technique is more desirable than placing all of the slack space on the upper side of a non-critical path group. The optimum solution would be to apportion the slack between the non-critical path groups evenly. Figure 16 shows this phenomenon on a simple example. What is necessary is to first determine the non-critical subgraphs of the constraint graph. Next, the sum of the excess distances between all groups in the subgraph should be maximized. An alternative approach (to minimize total line length) is to move each object to a lower or upper position to minimize the total line length. Note that the lower and upper

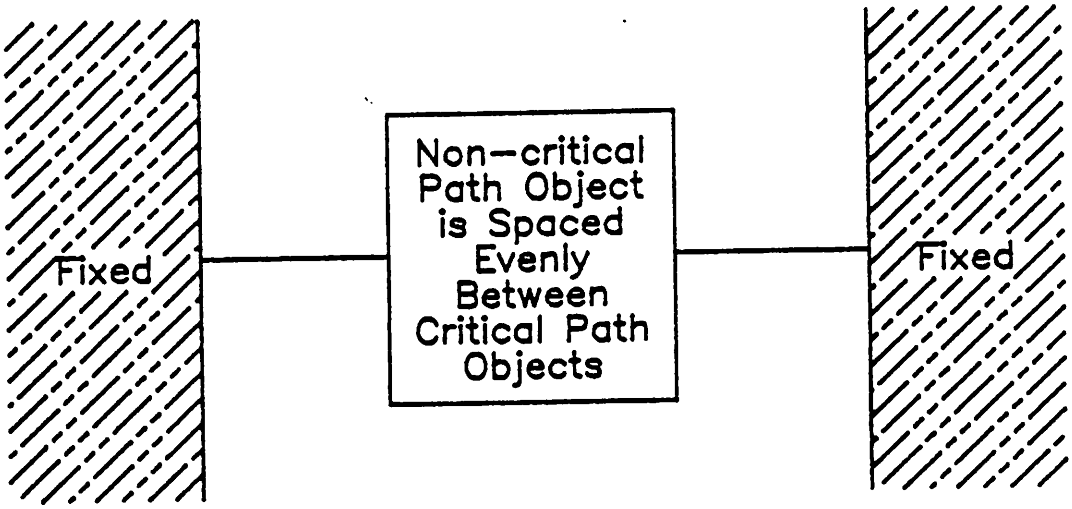

Fixed
Non–critical Path Object is Spaced Evenly Between Critical Path Objects
Fixed

Fig. 3.14 Non-critical Path Object Spacing

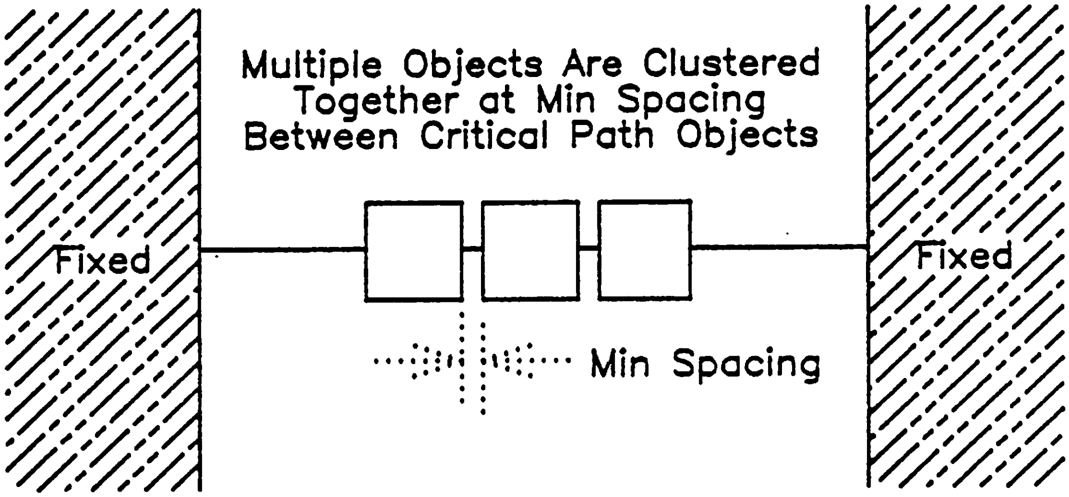

Fixed
Multiple Objects Are Clustered Together at Min Spacing Between Critical Path Objects
Min Spacing
Fixed

Fig. 3.15 Multiple Objects Not on the Critical Path

positions in this case may be different from the lower and upper positions computed in the forward and reverse passes of the longest path algorithm; since the slack is shared among possibly many non-critical path groups, placement of one group at its upper position followed by placement of an adjacent group at its lower position would most likely result in design rule violations.

In Python, the current approach to this problem is to average the lower and upper coordinate locations of each group. The critical path groups have equal upper and lower bounds so they do not move. The non-critcal-path groups are bunched together in the center of their available slack space. Although this is not the best possible solution to non-critical group placement, it is efficient and simple from computational and conceptual viewpoints.

### 3.2.6. Updating the Symbolic Data

After the spacing process, the instances and lines have left, bottom, right, and top location which reflect the instances new locations and the lines new paths. All that is necessary is to update this information in the Squid database in the output view which was copied from the input view during the *Readin* phase. The only possible problem is caused by local terminals. They are stored as instances but their *instance-id* actually refers to a terminal.

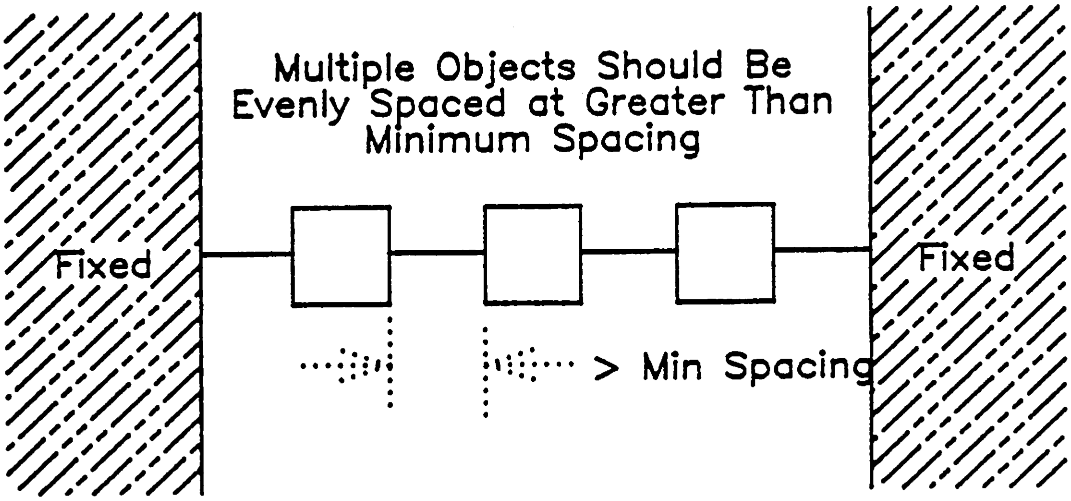


Fig. 3.16 Even Placement of Non-critical Path Objects

Attempting to retrieve an instance with the *terminal-id* will result in an error being returned from Squid.

Each instance in turn is retrieved from the database. Instances have a transformation matrix [Newman80] which determines the translation and rotations and mirrorings of the instance. Instances are only translated in Python. Thus to update the $T_x$ and $T_y$ values of the matrix (locations matrix[2][0] and matrix[2][1]), incremental distances are computed by subtracting the old from the new values of the left and bottom bounding box and adding these differences to the $T_x$ and $T_y$. The instance is then updated in Squid. If the retrieval fails, the instance is assumed to be a local terminal and the terminal is retrieved from the database. Terminals just have a rectangle which can be directly updated from the bounding box of the pseudo-instance of the terminal. All local terminals are then updated in Squid.

Lines are retrieved from the database using their *geoid* field. They return a path in an auxiliary array variable. This variable is updated with the new path of the line. This new path is derived by removing the width from the bounding box of the line and again by representing each line as an orthogonal path. Each line is then updated in the Squid database.

Finally, the newly updated output view is saved on disk and the entire spacing process is complete. The end result is a version of the symbolic layout of an integrated circuit which is of minimal area while satisfying all layout rules. The spacing maintains the absolute locations of the left and bottom portions of the cell as constants.

## 3.3. Implementation Issues

### 3.3.1. Data Structures

The data structures of Python are quite complex. This choice was made so that the program would run efficiently after the setup phase with very few patch-up steps. Each of the data structures is listed and then followed by a brief description. They are grouped by the major program subsection in which they are most heavily used. The C programming language [Kernighan78] **struct** is similar to the Pascal **record** construct. The Glossary of Terms (Appendix E) explains unfamiliar terms.

**Readin**

```
struct INSTANCE {
        ctype               l;       /* left (lower x) coordinate */
        ctype               b;       /* bottom (lower y) coordinate */
        ctype               r;       /* right (upper x) coord */
        ctype               t;       /* top (upper y) coord */
        instype             type;    /* true instance or local terminal */
        int          instid;         /* used to id instance in Squid */
        struct INSTANCE     *next;   /* pointer to next INSTANCE */
        struct INTVLHD      *xfe;    /* left and right frame edges */
        struct INTVLHD      *yfe;    /* bottom and top frame edges */
        struct VERTEX       *vl;     /* left coordinate vertex */
        struct VERTEX       *vb;     /* bottom coordinate vertex */
        struct VERTEX       *vr;     /* right coordinate vertex */
        struct VERTEX       *vt;     /* top coordinate vertex */
};
```

Each Squid instance has a corresponding INSTANCE **struct** in Python. The bounding box *l*, *b*, *r*, and *t* values are taken directly from the Squid instance. This bounding box is used to perform a rough check for intersection in spacing rule analysis. Also, it defines the location of the instance and is updated after each longest path calculation. Any other data structure that references a bounding box does it indirectly through an INSTANCE **struct**, so that updating this information in one place will guarantee that all routines will use the correct values.

Local terminals have instances created for them so Python can treat them in a consistent manner with instances. The *type* field distinguishes the two cases and is used when the spaced layout is updated in the database.

The *instid* is used to identify the particular instance in Squid. This field is used to retrieve the Squid instance when its location is to be updated after spacing.

Instances are linked together, so a *next* member is included to provide the link.

Two sets of frame edges are provided for the X and Y sides of the instance. The X sides (vertical lines) are used in the X spacing rule analysis and the Y sides (horizontal lines) are used in the Y spacing rule analysis. These edges are the edges of the polygons which form the protection frames for the instance. They consist of one set of edges per active mask layer. See the definitions of the INTVLHD and INTRVL structures for more details.

A vertex in the constraint graph is associated with each bounding box edge of an instance. The left and right vertices are in the x-directed graph and the bottom and top vertices are in the y-directed graph. Edges are added from the lower vertex to the upper vertex in each graph to fix the distance between lower and upper edges since instances change only their location during the spacing process. Each vertex has a pointer to the coordinate to which it is linked and so has access to the correct bounding box information at all times. See the definition of the VERTEX structure for more details.

```
struct LINE {
        ctype           l;      /*left (lower x) coordinate */
        ctype           b;      /*bottom (lower y) coordinate */
        ctype           r;      /*right (upper x) coord */
        ctype           t;      /*top (upper y) coord */
        ntype           netid;  /*integer net id */
```

```
    int                     geoid;  /*geometry id used with database */
    otype             orientation;  /* HORIZONTAL or VERTICAL */
    struct LINE             *next;  /*pointer to next LINE */
    ctype                   width;  /*width of the line */
    int                      mask;  /*integer mask number */
    struct VERTEX           *vl;    /*left coordinate vertex */
    struct VERTEX           *vb;    /*bottom coordinate vertex */
    struct VERTEX           *vr;    /*right coordinate vertex */
    struct VERTEX           *vt;    /*top coordinate vertex */
};
```

The LINE **struct** represents a line, or piece of interconnect, in the Python data structure. Although it also has a bounding box, this box must be derived from the endpoints and width retrieved from the Squid database. This derivation is performed later on in the construction of the constraint graph so the *width* of a line is explicitly stored along with the bounding box. The *geoid* identifies the line to the Squid database and is used to retrieve each particular line when updating path information after spacing. Since a LINE is part of the physical implementation of a net, which connects two terminals, it has an integer *net-id* associated with it. There is also a *next* member to link the lines together.

Lines also contain an orientation which is one of HORIZONTAL or VERTICAL. Different than instances, all of the geometry for a line is contained on a single mask layer. Thus, lines contain an integer *mask* number. This number is related to an actual mask name by the Squid database.

Lines are similar to instances since they also contain a vertex for each of the bounding box edges. In the direction perpendicular to the line's orientation, a fixed-length edge is added to keep the two sides of the line spaced at the line width. In the direction parallel to the line's orientation, an edge is added to keep the length of the line from becoming negative. As in the case of instances, each vertex has a pointer to the coordinate to which it is linked.

```
struct TERMINAL {
        ctype              l;          /* left (lower x) coordinate */
        ctype              b;          /* bottom (lower y) coordinate */
        ctype              r;          /* right (upper x) coord */
        ctype              t;          /* top (upper y) coord */
        ntype              netid;      /* integer net id */
        struct TERMINAL    *next;      /* pointer to next TERMINAL */
        int                mask;       /* integer mask number */
        struct INSTANCE    *owner;     /* element that 'owns' term */
};
```

The bounding box information for terminals is read directly from the Squid database. Terminals contribute no vertices to either of the constraint graphs and only serve to generate the proper constraint between an instance and the lines that connect to it. Local terminals, *i.e.* terminals that exist outside of any instance in the cell being spaced, are treated specially in the readin phase. They have instances constructed to represent them although there are not really such instances in their Squid representation. Creation of these special instances is necessary to treat local terminals in a manner consistent with the treatment of instances.

Terminals, like lines, are also part of the physical implementation of a net so they also contain *net-ids*. Since their geometry is entirely on one mask layer, they contain an integer mask number as well.

**Buildgraph**

```
struct INTRVL {            /* interval list type */
        ctype              high;       /* high coordinate */
        ctype              *hbase;     /* high coord base origin */
        ctype              low;        /* low coordinate */
        ctype              *lbase;     /* low coord base origin */
        ctype              center;     /* centerline of interval */
        ctype              *cbase;     /* center coord base origin */
        itype              type;       /* type of interval */
        ntype              netid;      /* 0 unless over term or line */
        int                mask;       /* integer mask number */
        struct INTRVL      *next;      /* pointer to next interval */
        struct VERTEX      *owner;     /* owner of interval */
};
```

INTRVL **structs** are kept in lists headed by INTVLQ **structs**. They are sorted from high to low by the absolute location of the center coordinate, which is the center coordinate plus the center base origin. They are then sorted by the absolute location of the lower coordinate, which is the low coordinate plus the low base origin. The *high*, *low*, and *center* members are specified relative to the corresponding base origins of their parent instance. Therefore, lines, which change dimension during the course of the spacing process, can be treated in the same manner as instances, which do not change dimension, without the need for special case processing.

The *type* field is a bit field and contains information about the convexity of the intervals endpoints (used in spacing rule analysis) as well as defining whether the interval is an upper or lower interval. This is necessary since intervals are kept in lists according to their orientation.

If an interval overlaps a terminal frame edge (in an instance) or if it overlaps the edge of a line, it carries the *net-id* of the geometry it overlaps. This information is used in spacing rule analysis to obtain more optimal results by ignoring spacing requirements between intervals of the same *net-id.*

Intervals are associated with a single mask so they carry an integer *mask* number.

Interval lists are linked so they have a *next* entry.

Since intervals are sorted in a global list, they have a *owner* pointer to the vertex which represents the bounding box edge corresponding to the *cbase* coordinate.

### Spacing Rule Analysis

```
struct INTVLQ {                 /*protection frame edge list head */
        struct INTRVL       *intrvl;        /*pointer to first interval */
        struct INTVLQ       *next;          /*pointer to next list head */
        struct INTVLQ       *prev;          /*pointer to previous list head */
};
```

These **structs** head the lists of projected frame edges for the spacing analysis queues. They are doubly linked together, and point to their respective *intrvls*. One set of lists is initially provided for each of the two sets (X (right and left) and Y (top and bottom)) of frame edges for each instance. The frame edges of all instances and lines are linked together, sorted, and used in the spacing rule analysis to determine the minimum spacing allowed between objects.

### Longest Path Calculation

```
struct VERTEX {
        ctype               loc;            /* current real location */
        ctype               newloc;         /*new loc during iterations */
        ctype               loloc;          /*used for forward pass 'loc' */
        ctype               *coordinate;    /* coord. for vertex to update */
        struct EDGE         *pred;          /*predecessor edges pointer */
        struct EDGE         *permpred;      /*'permanent' edges pointer */
        struct EDGE         *succ;          /*successor edges pointer */
        struct EDGE         *permsucc;      /*'permanent' edges pointer */
        int                 refcnt;         /*used for pred. count in CPM */
        struct VERTEX       *next;          /*ptr used to link vertices */
        struct VERTEX       *nextq;                 /*ptr used to schedule vertic*/
        sstat               schedule        /*used for scheduling queues */
};
```

The VERTEX **struct** and the EDGE **struct** (below) are used to space the elements properly. Each VERTEX contains a *loc* and *newloc* to compute the longest path to each vertex in the forward and backward passes. There is also a *loloc* to store the forward pass value during the backward pass calculations. As stated above, each vertex points to a coordinate. There are lists of predecessor and successor edges as well as the 'permanent' successor and predecessor edges (those preserving shape of instances, width of lines, and electrical connectivity between lines and terminals). A reference count serves in the first iteration of the forward and backward passes. It counts the number of predecessors or successors. A *schedule* status word is used during the forward and backward passes to record the scheduling information of each vertex.

The vertices are scheduled for the event-driven longest-path solution in queues by the *nextq* member. Queues are maintained for the current and next iteration queues during the course of the solution. As the position of a vertex changes, it causes its predecessors and successors to be scheduled, since they may change. Using two queues maintains the notion of iterations, while at the same time minimizing the amount of data to be examined. The notion of iterations must be kept so that over-constraining conditions can be detected.

```
struct EDGE {
        ctype              lobnd;       /* lower bound */
        ctype              hibnd;       /* upper bound */
        struct VERTEX      *pred;       /* predecessor vertex pointer */
        struct VERTEX      *succ;       /* successor vertex pointer */
        struct EDGE        *nextpred;   /*ptr to next pred edge */
        struct EDGE        *nextsucc;   /*ptr to next succ edge */
};
```

The EDGE **struct** links two vertices together in the constraint graph. It contains the *lobnd* and *hibnd*, which indicate the lower and upper magnitudes of the constraints imposed by the edge. Each EDGE contains pointers to its source and sink vertices, under the names *pred* and *succ*. The lists are doubly threaded, once from the source vertex and once from the sink vertex through the *nextpred* and *nextsucc* members.

**Global Structures**

```
struct ERROR {
        pstat  errnum;
        char   *errmsg;
};
```

### 3.3.2. Squid/Python Conventions

Python expects the following restrictions on its input from Hawk through Squid:

- All lines are single segment (two endpoints) ending on terminal frames
- Intersecting lines must have a pseudo-instance at the intersection point
- All protection and terminal frames must be in the instance bounding box
- Orthogonal Edges (integral multiples of 90 degrees on all edges)
- No overconstraining conditions may occur
- Electrical connectivity must be explicitly expressed in the net ids
- Protection frames of objects must not overlap

These restrictions were made for several reasons. All lines must be a single line segment in order to keep the model of the symbolic integrated circuit data simple. From this restriction follows the next point, that is, lines which intersect must have a psuedo-instance (which can be viewed as a layer-to-same-layer contact) added at their intersection point to keep the model consistent. By keeping the model simple, the Python program can be made more efficient. By treating lines and instances in a consistent manner, implementation of the algorithms is simpler and the program can be made more compact and easier to maintain.

The algorithms for spacing analysis and subsequent longest path solution are decoupled into separate X and Y spacings and hence are are an order of magnitude less complex when used with orthogonal geometries.

If overconstraints are present in the graph, the analysis will take much longer ($v$ iterations) to determine an error is present and the algorithms have no way to determine exactly where the error is. The order of time complexity to find the objects contributing to the overconstraint is $O(2^v)$, where v is the number of vertices in the graph (the solution is non-deterministic polynomially bounded). Although it will always be possible for the user to enter a layout which is overconstrained, proper conditioning of the input as it is entered into the Squid database (with Hawk) can detect many of these errors as they are introduced. Chapter 4 describes the types of overconstraints which can occur in Python.

Explicit electrical connectivity is absolutely essential for a proper spacing. As the levels in the hierarchy increase, the shapes of protection frames and terminal frames become more complex. This makes it impossible for the program to derive the connectivity of the circuit directly from the layout, as was done in CABBAGE. CABBAGE required all interconnect to terminate at the exact center of the instance it was connected to. This was not an unreasonable restriction since there were no sliding contacts and each instance was allowed only one interconnect to terminate per side of the instance. Python cannot make this restriction so it becomes necessary to add the physical connectivity information to the Squid database.

Because the spacing rule interval analysis is performed scanning from left to right, constraints may not be generated between instances if their protection frames overlap. This is an artifact of the implementation of the spacing rule analysis algorithm and should be fixed.

### 3.3.3. Constraint Graph Construction

The constraint graph used in Python has the values of vertices represent the physical coordinate locations of bounding box edges of the instances and lines that make up the IC cell. The analyses are decoupled so there are separate graphs for both X and Y directions. The X graph contains vertices to represent the right and left sides of the bounding boxes and the Y graph contains vertices to represent the top and bottom sides of the bounding boxes. Each vertex has a pointer back to the coordinate which caused the vertex to be allocated. After each longest path analysis, these coordinates are easily updated from the new locations of the objects stored in the graph vertices. Since the intervals used in the spacing rule analysis have their dimensions specified relative to specific coordinates of the bounding box of the instance or line to which they belong, the intervals belonging to lines (which grow and shrink) are properly adjusted automatically, since the bounding box locations are updated at the end of each longest path analysis.

Three locations are required in each vertex. During the longest path analysis, the location at the past and present iterations is necessary to determine when a vertex changes location. The third location is used to store the minimum possible coordinate location while doing the reverse pass through the graph (to determine the *maximum* possible coordinate locations).

Each vertex has a pointer to its predecessors and successors. To keep track of the 'permanent' part of the graph (which remains static throughout the entire series of analyses) pointers are kept to the permanent predecessors and successors. Since new edges are added at the beginning of the edge lists, returning the graph to its permanent state is done merely by copying the permanent pointer over the temporary one.

Edges are added between vertices and have lower and upper bounds which represent the minimum and maximum differences in position between the two vertices each edge connects together. They also contain pointers to their predecessor vertex (source vertex) and successor vertex (sink vertex). They are linked together and they have two link fields since they appear in both the successor list of their predecessor vertex and the predecessor list of their successor vertex.

Edges are added for a number of reasons. The first is to preserve the shape of the instances. Specifically, this means keeping a constant spacing between the two bounding box edges which appear in each graph of each instance. These edges have fixed upper and lower bounds which are equal, and are merely the difference in location between the upper and lower bounding box edges.

Edges are also added to preserve the width of lines. These are only added between vertices of lines which are perpendicular to the direction of spacing. These edges are also fixed, and their equal lower and upper bounds are just the width of the line.

In the direction parallel to the direction of spacing, lines can stretch and shrink. However, because of the problems with the merging of elements, it is not possible to let the length of lines drop below zero. Thus an edge with zero lower bound and infinite upper bound is added between the vertices of lines parallel to the spacing direction. A line can be as long as is necessary allow the surrounding geometry to satisfy the spacing rules, but it can never drop below zero length.

Edges are next added between instances and lines to preserve the electrical connectivity of the circuit. The size of the terminal frames each line terminates on determine the magnitude of the lower and upper bounds of the edge. If the terminal is wider than the line that terminates in it, the lower and upper bounds are unequal and provide the 'slop' in the sliding terminals. If the width of the line and its terminal are equal, the lower and upper bcunds of the edge are equal and fix that endpoint of the line rigidly to the instance which owns the terminal.

Edges are added to express the minimum allowable spacing requirements between objects. This reason is perhaps the most important of all. Each edge is the maximum of all of the spacing requirements generated by overlaps of the intervals of two objects. With the addition of these spacing requirements, the constraint graph is now complete and represents the topology of the IC cell.

# CHAPTER 4

# Results

## 4.1. Examples

### 4.1.1. CABBAGE I Latch Example

As a comparison between the *CABBAGE* and *Python* programs, the latch block example from [Hsueh79] was compacted with both programs. The gate-level schematic for this block is shown in figure 4.1. The program *cab-*

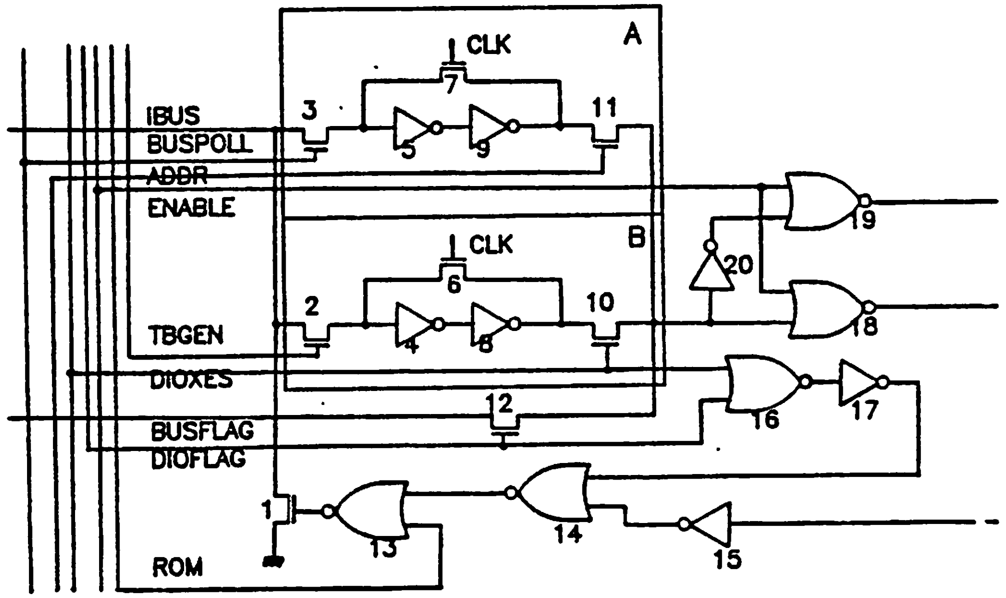


Fig. 4.1 Gate-level Schematic for Latch-Driver Block

*tosquid* (Appendix G) was used to translate the CABBAGE symbolic intermediate file and enter the data into the *Squid* database. Both programs used the same design rules. The input for both programs is shown in Figure 4.2. There are two observable differences in compaction method. Python does not permit objects to merge but uses sliding contacts. CABBAGE permits objects to merge but uses point contacts for terminals. Objects are permitted to merge in CABBAGE when the merging will not affect the circuit electrical performance and will help decrease the size of the layout. Since CABBAGE is written for a specific technology, the rules defining permissible merging are well known. Python is technology independent, and no general purpose method for expressing or determining permissible merging has been implemented. Sliding contacts imply that each endpoint an interconnect line must terminate within the area of its terminal frame, but it can terminate anywhere within that area. The compaction program can place the endpoint anywhere within its terminal frame to obtain minimum size for the total layout. The results of the compactions are shown in Figures 4.3 and 4.4.

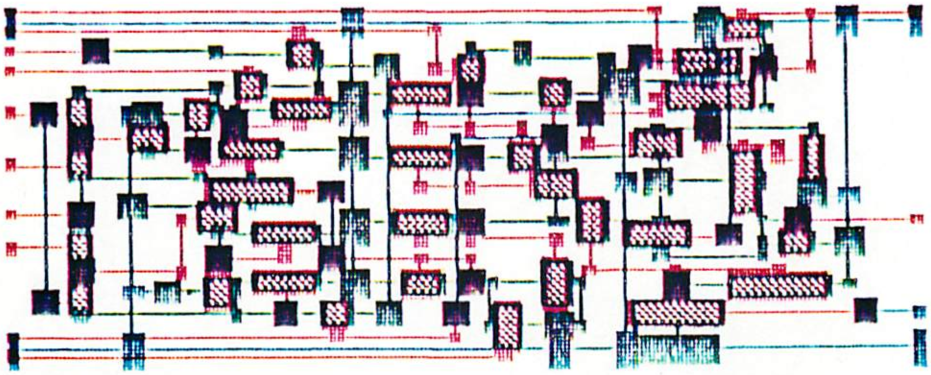


Fig. 4.2 Latch-Driver Block Input Symbolic Layout

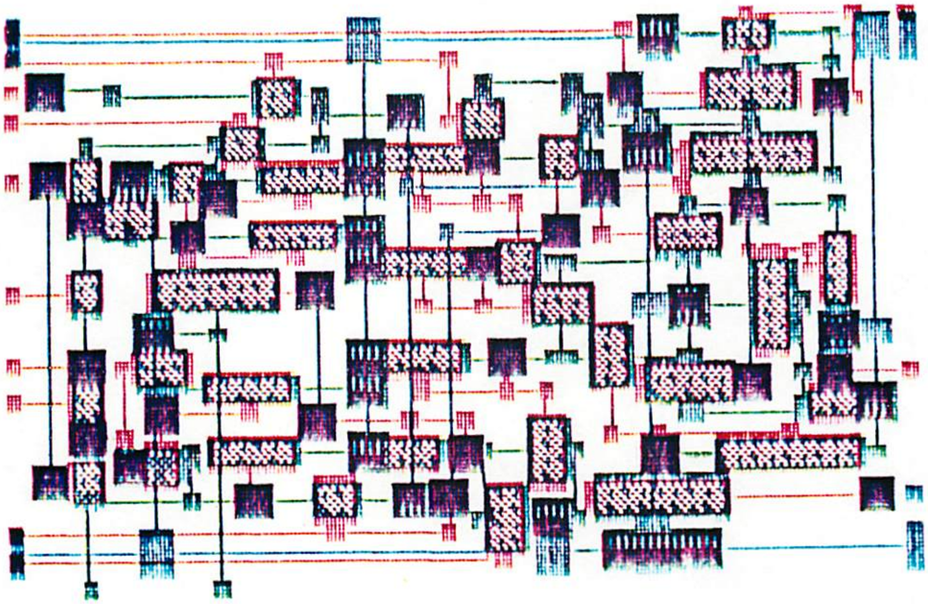


Fig. 4.3 Latch-Driver Block Compacted with the CABBAGE Program

CABBAGE compacts the cell to a size approximately 15% smaller than the size of the cell as compacted with the Python program. The difference in size in this example can be attributed to the ability to merge objects within CABBAGE.

### 4.1.2. A Low Level NMOS Example

A further comparison of the CABBAGE and Python programs was made for the example of an NMOS D-type flip/flop, taken from [Hsueh79]. The schematic diagram for the circuit is shown in Figure 4.5. *Cabtosquid* was used to translate the CABBAGE ASCII symbolic intermediate file and enter it into the Squid database. The conventions of Python were observed, so intersections of interconnection lines had layer-to-same-layer terminals added. This is apparent in Figure 4.6, which shows the input to both compaction pro-

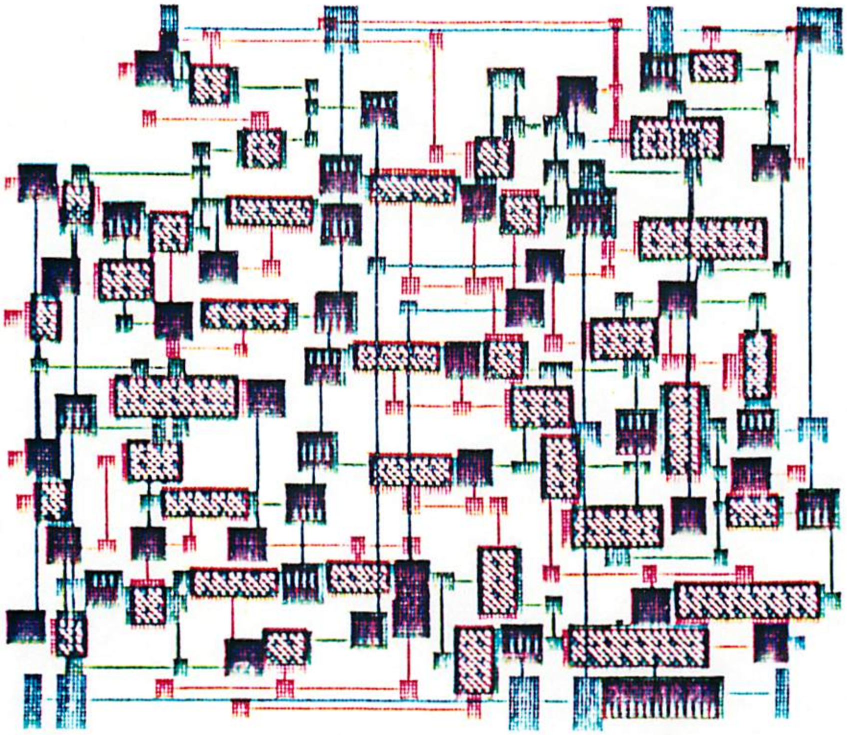


Fig. 4.4 Latch-Driver Block Compacted with the Python Program

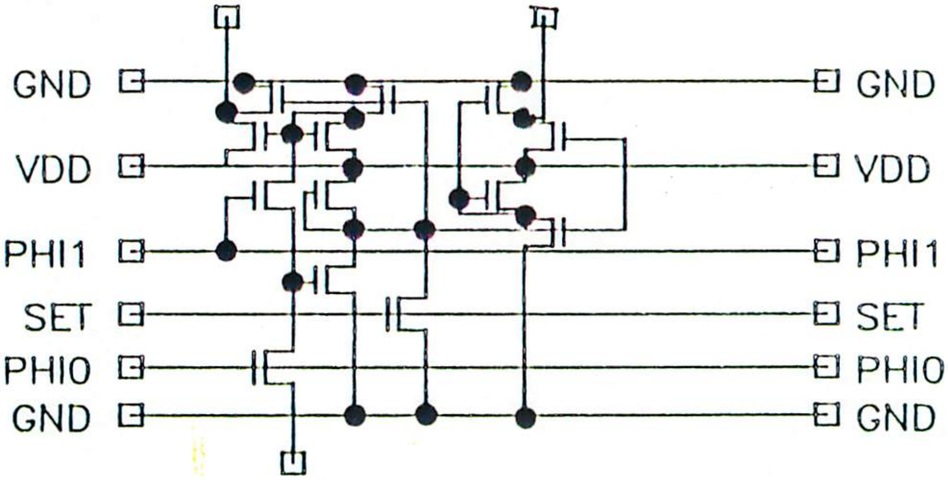


Fig. 4.5 Schematic for D-type Flip/Flop

grams, displayed in the input format for Python. This same input was compacted with equal spacing rules in both programs.

The outputs from CABBAGE and Python are shown in Figures 4.7 and 4.8. The result obtained with Python is approximately 20% smaller than the results obtained with CABBAGE. This difference in size can be directly attributed to the use of the sliding contacts. If merging were allowed in Python, the size difference might have been even larger.

### 4.1.3 Hierarchical Spacing of the Low Level NMOS Example

To demonstrate the use of the hierarchy, the D-type flip flop example of the previous section is used in a shift-register cell. The Squid cell for the compacted version of the flip/flop has protection frames generated for it using the program *frame* (Appendix H). The protection frames are used to place multiple instances of the cell within a shift register cell and the terminals of the flip/flops are interconnected. The resulting cell is shown in

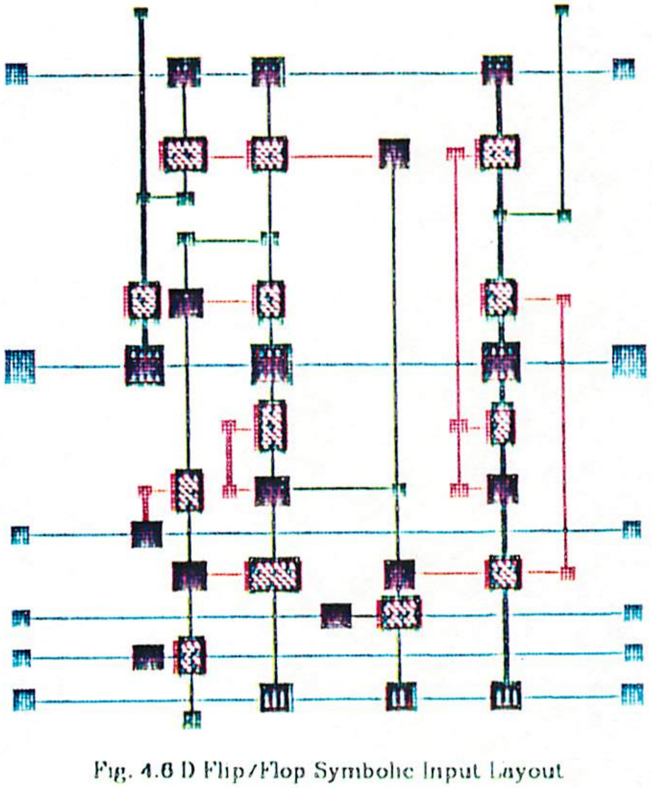


Fig. 4.6 D Flip/Flop Symbolic Input Layout

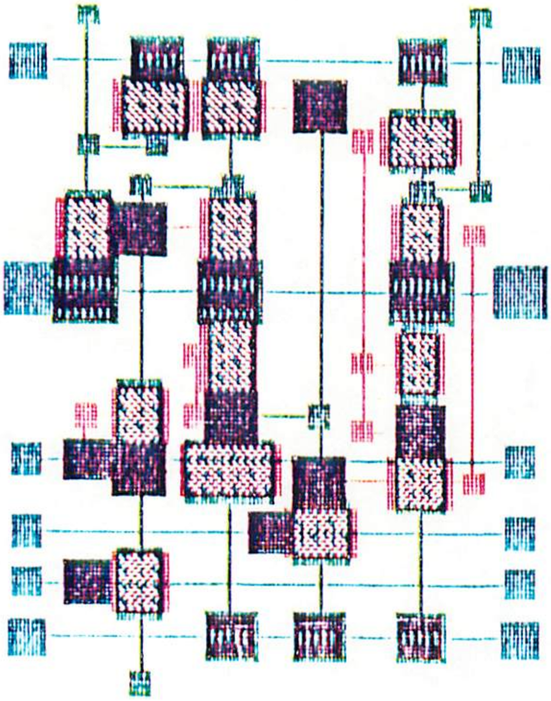


Fig 4.7 Compacted Output from the CABBAGE Program

Figure 4.9 This register cell is compacted with Python, and the output is shown in Figure 4.10 Using hierarchy in this manner greatly reduces the

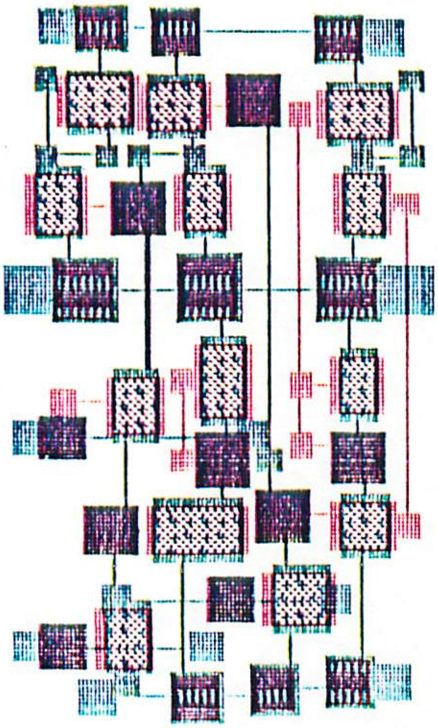


Fig 4.8 Compacted Output from the Python Program

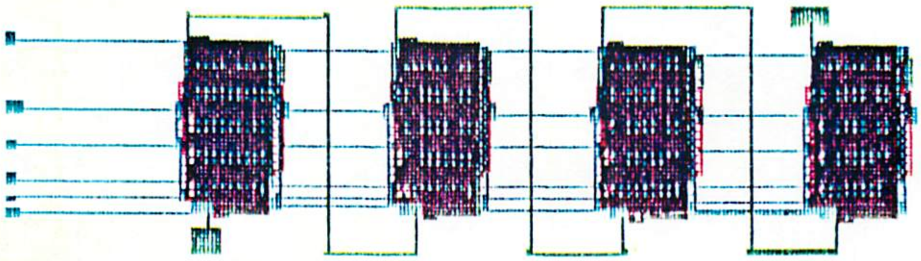


Fig. 4.9 Shift Register Cell Made From D-Type Flip/Flops

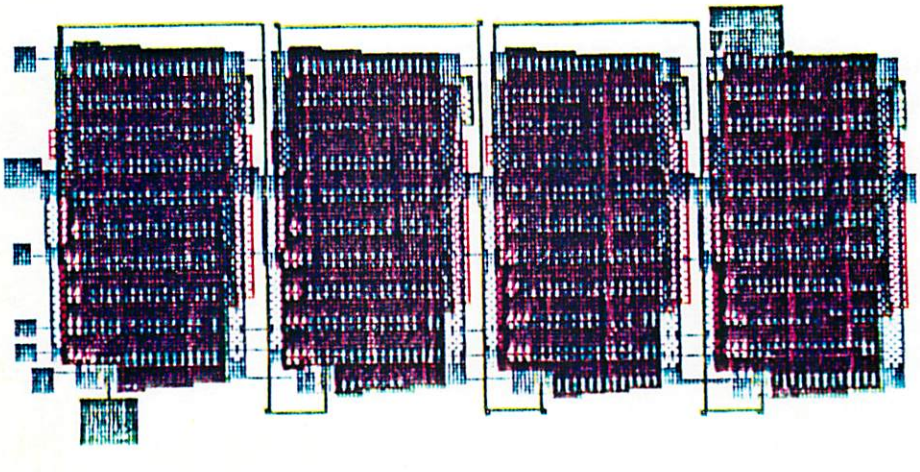


Fig. 4.10 Compacted Result of Figure 4.9

time required for the compaction of a complete circuit. The greater the regularity, or repetition, of cells used in the layout, the greater the savings in the overall compaction time.

## 4.2. Results

In spite of the greater complexity of the spacing rule analysis and longest path algorithm, Python is approximately 2.5 times faster than CABBAGE on the same examples. Even more important, the percentage difference in run times for small and larger input cells is much smaller for Python which implies that the overall order dependencies in Python are lower than those in CABBAGE.

The time/memory tradeoffs between the two programs seem almost linear; Python uses approximately 2.5 times more memory than CABBAGE. The larger memory usage of Python stems from the graph representation for the IC topology which has many more vertices and edges than the corresponding graph in CABBAGE. Here is a comparison between CABBAGE and Python in tabular form.

### 4.2.1. Run-times and Order Dependencies of Algorithms

All times reported are for a Digital Equipment Corporation VAX 11/780 32 bit minicomputer running the 4.1BSD version of the VAX/Virtual UNIX† operating system [Fabry82][Ritchie78]. CABBAGE was compiled from the *ratfor* programming language [Kernighan76] using the UNIX *f77* compiler[Feldman78]. Python is written entirely in the *C* programming language [Kernighan78], and was compiled with the UNIX cc compiler [Johnson80].

The *readin* phase is $O(n)$, or linear with time, with number of objects. A greater number of different master cells called within the cell being spaced

†UNIX is a Trademark of Bell Laboratories.

tends to increase the readin time above linear order and a larger number of calls to the same master cells within the cell being spaced tends to decrease the time below linear order. The two factors roughly cancel for the IC cell examples described here resulting in $O(n)$ linear order dependence.

The *buildgraph* phase is also approximately $O(n)$. The one exception lies in the routine *P_cktermline*, which derives the connectivity of terminal frames and interconnect lines. The terminals and nets are sorted by net id, and then compared on an object by object basis. This comparison is $O(m^2)$, but $m$, the number of terminals and lines in each net, is generally small. Exceptions to this include global signals, such as *VDD*, *GND*, and *CLK* signals. Fortunately, the hierarchy helps to reduce the number of terminals and lines in global nets, and the observed order dependence is only slightly above linear.

The *sranalyze* has the worst order dependence of any subsection, observed to be approximately $O(n^{1.7})$. Comparison of each primary (right-side edge) with every other neighbor (left-side edge) is an $O(n^2)$ operation. The savings in the current implementation comes from sorting the intervals prior to the sranalyze phase. This sorting is $O(n \log n)$ dependent, and lowers the order dependency of the analysis through the ability to ignore many comparisons when there is no possibility of interaction between sets of intervals.

The *lngpth* phase has a *proven* order dependence of $O(v^3)$ [Lawler76]. This is fortunately a worst case order dependence, and only is true for completely connected graphs. The graphs generated by IC layouts seems to have a constant regularity somewhere between 3 and 5. Thus, the observed order dependence is only slightly above linear.

The *update* portion of the program is also linear, $O(n)$. A single pass is required through all of the instance and line data structures, and only the root Squid cell requires updating.

The following table gives CPU times (broken down into true cpu and system times) for the D Flip/flop and Latch examples presented in the previous section.

| *Run Times and Memory Usage* | | | | | | | |
|---|---|---|---|---|---|---|---|
| Example | Elements | CABBAGE | | | Python | | |
| | | User | System | Memory | User | System | Memory |
| dff | 98 | 20.3 | 1.5 | 125544 | 11.9 | 2.8 | 417764 |
| intlk | 290 | 197.7 | 5.4 | 166664 | 72.4 | 7.7 | 750636 |

System time is mainly disk I/O time, and User time is the CPU time. Again, the difference in memory utilization stems from the more flexible graph representation of the IC layout used in Python.

### 4.2.2. Program Status

Python consists of ~4000 lines of C code, with ~1000 of the 4000 lines being comments. In addition, the Squid database has ~2000 lines of C code.

### 4.2.3. Non-Optimal Results of Python

There are still algorithmic improvements that could be made to Python that would improve its performance and/or improve the area efficiency of the spacing process. The ability to merge elements is the single most important feature lacking from Python. Components of elements may safely merge if they are on the same mask layer and in the same electrical net. Python cannot permit merging, since topology changes might occur which could cause unexpected design-rule violations. The related problem of line lengths changing sign is a less severe problem, but will need to be introduced for jog generation to work properly. When an interconnect line is in the critical path, and the surrounding objects are also in the critical path, the spacing efficiency will benefit from the insertion of jogs, or zero length lines inserted perpendicular to the line somewhere along its length. Aside from the problem of determining where the jog should be inserted to permit the smallest area result after a subsequent series of compactions, if the length of interconnect lines cannot drop below zero length, the orientation of the wire will have to determined by which way the surrounding geometry will move. Several overconstraining conditions can occur which cannot be detected easily. The data structuring for the spacing rule analysis imposes an $O(n^{1.7})$ dependency on the analysis, when it is possible to reorganize the data, obtaining an $O(n)$ dependency on both the sorting of intervals and the spacing rule analysis itself.

As is readily apparent in the A-B comparison of the latch-driver example, CABBAGE gains a great deal of area efficiency through the merging of instances during the spacing process. Given the more general nature of Python, it is difficult to implement a general merging strategy. Therefore, although the sliding contacts provide a good deal of area efficiency, they are not alone sufficient to produce a well-minimized layout.

Related to the problems with the ability to merge is the need to keep the lengths of lines from dropping below zero. This problem is shared between CABBAGE and Python. It stems from the fact that if two edges can merge, then they can change relative position. Figure 4.11 shows how a third edge can be covered when it should not be covered, resulting in a spacing rule violation after the spacing process. In addition, a problem not encountered in the normal merging problems is the change in sign of the length of the lines. Top becomes bottom, and vice versa, which requires some adjust-

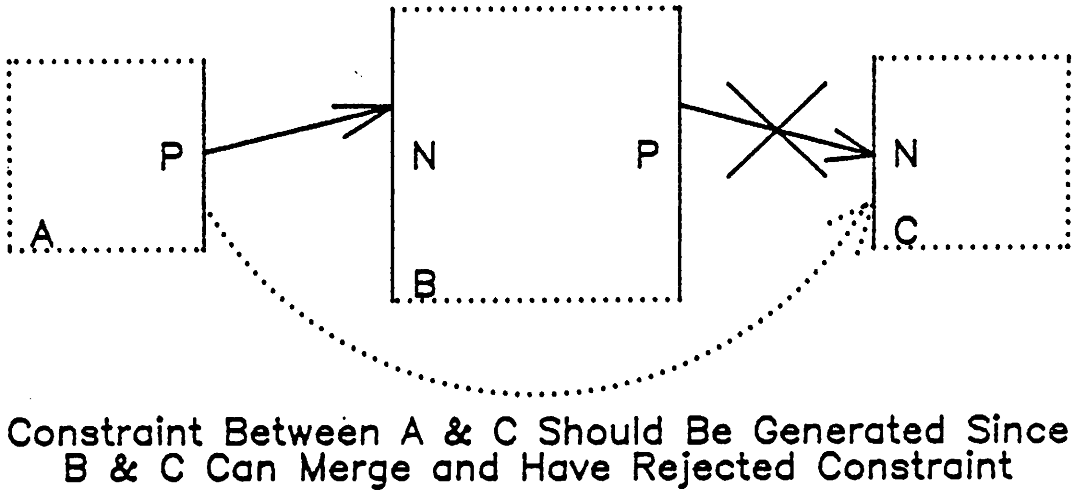

P
N
P
N
A
B
C
Constraint Between A & C Should Be Generated Since
B & C Can Merge and Have Rejected Constraint

Fig. 4.11 Incorrectly Covered Edge

ment of the bounding box parameters and vertices associated with lines. This patch-up step would have to be done after each spacing iteration.

If the sum of the widths of several different lines terminating on a single terminal frame is greater than the width of the terminal frame, an overconstraining condition will be generated, and the spacing analysis will fail. Since the general detection of overconstraining conditions in the graph is an NP-complete problem [Lawler76], this analysis is not performed in Python. All that is reported is the existence of an error somewhere in the graph. Also, if an object is initially placed within a concavity that is too small for it and the wrong initial direction of compaction is chosen, an overconstraining condition will be discovered, even though it would not have occurred if the initial direction of compaction were chosen differently (See Fig. 4.12).

The intervals used in the spacing rule analysis could be organized into a two-dimensional bin data structure. This would make the sorting time linear with a slightly worse coefficient than the current version due to the overhead of computing the bin locations. It would also make the interval analysis

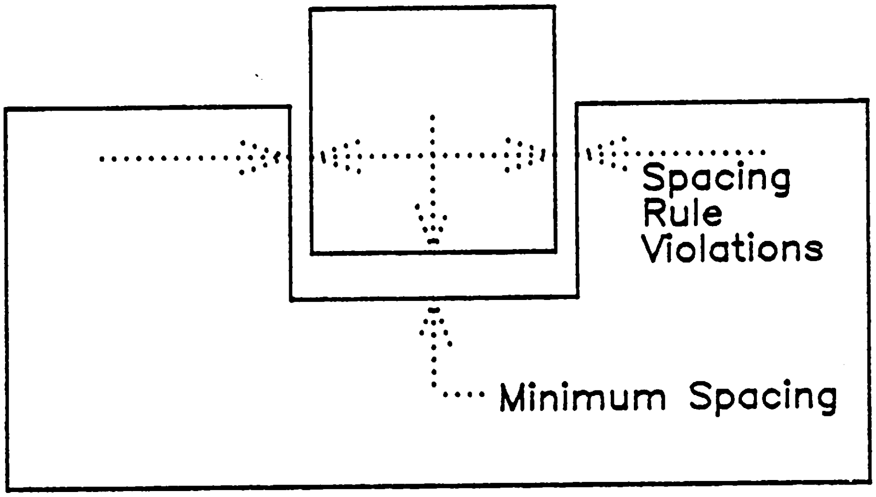


Fig. 4.12 Overconstraining Initial Condition

almost linear. The current method effectively has bins in one dimension only.

# CHAPTER 5

# Summary

## 5.1. Summary of Python Characteristics

*Python* is symbolic IC layout spacing aid based on the *CABBAGE* program. The increasing complexity of integrated circuit designs makes the use of traditional, non-hierarchical design-aids expensive, both in design time and in actual computer cost. Python reduces tedium involved in IC layout. The use of symbols that need only be placed relative to one another makes the layout process very similar to the initial layout sketching that is part of most IC design methods.

The algorithms used in Python have been described and their derivation from the CABBAGE approach has been explained. The major extensions to CABBAGE are in the areas of the longest path solution and the spacing rule analysis.

The implementation of the algorithms in Python is completely different from the implementation of the CABBAGE algorithms. Python is written in the *C* programming language [Kernighan78], uses the *Squid* [Keller82] database to store the intermediate symbolic format, and has the *Hawk* viewport manager [Keller82] as its input editor.

A detailed comparison of Python and CABBAGE has illustrated the speed/memory tradeoffs possible. While Python is approximately 2.5 times faster than CABBAGE for large circuits, the additional capabilities of Python, including hierarchy, sliding contacts, technology, and process independence make it a more powerful aid for symbolic IC design.

### 5.2. Open Research Questions

Although the Python program provides a solid basis for future work, many important questions remain unanswered. Some of the more interesting research questions are presented in this section. Most deal with making the program useful as a production program and with extensions to the program which will allow it to track the ever-changing state of the art in IC design and layout.

#### 5.2.1. Error Detection in Over-Constraining Conditions

The most important shortcoming of Python is the inability to determine the *location* within the graph of positive cycles which represent over-constraining conditions. The number of iterations of the longest path algorithm to even determine the *existence* of an overconstraint is equal to the worst case number, *i.e.* the number of vertices in the graph. Since there are two vertices for each instance and each interconnect line in each of the X and Y graphs, the number of vertices grows linearly. The time for each iteration grows as a small fraction of the number of vertices in the graph. Thus, although the total time for solution of the longest path algorithm has an order dependence only slightly above linear, the actual time can grow prohibitively large for complex layouts.

The complexity of any algorithm to determine the locations within the graph of all positive cycles is at least $O(2^v)$ where $v$ is again the number of vertices in the graph. This time becomes totally impractical for very small cells.

At present, there are very few ways to generate overconstraints in the layout topology since there is no way to enter user-defined constraints. If more than one interconnect line terminates through the same side of a terminal frame and the total width of the lines is greater than the width of the terminal frame, an overconstraining situation will be created. The spacing between the interconnect lines is not counted since they will be of the same net and their spacing requirements will thus drop to zero. Also, if an overconstrained condition such as is illustrated in Fig. 4.12 is created in the initial layout and the wrong initial direction for spacing is chosen (X in the case of the figure), an overconstraining condition will be entered into the graph and the spacing will fail. If the other spacing direction had been chosen first, or if the original layout had not included the box in the notch, the overconstraint never would have been generated because the corner constraint checking in the Y direction would have prevented it.

What is necessary for future work is to develop fast heuristics to discover the location of overconstraints in a layout.

### 5.2.2. Jog Generation and Placement

Jog generation was not added to Python because the jog insertion algorithms in CABBAGE are not sufficient and no better solution could be found. The algorithms to find the proper lines to insert jogs on are very sound. The process consists of finding all interconnect lines perpendicular to the spacing direction which are on the critical path (*i.e.*, have their endpoints connected to two vertices which have equal minimum and maximum allowed positions for the minimum area). CABBAGE places the jog at the middle of each such line. Figure 5.1 shows that this form of jog insertion requires a large number of jog insertions to have any real effect. The line segment is

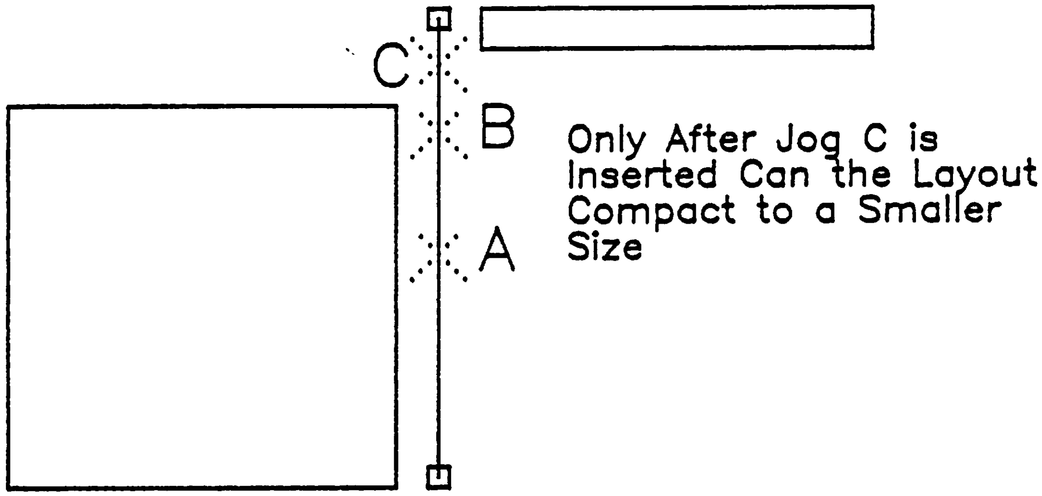

C
B
A
Only After Jog C is Inserted Can the Layout Compact to a Smaller Size

Fig. 5.1 Worst Case Jog Insertion

split in half by the first jog. The second jog splits the top half of the line, and so on until finally the jog point is high enough to allow the right side to slip past the left side to yield the desired, compact result. A human designer, with his knowledge of the circuit topology, could properly place the jog the first time to allow the compaction to gain the most benefit from the jog. What is required is to examine the geometry surrounding each interconnect line on the critical path to determine where jogs should be inserted. There does not seem to be a rigorous algorithm for determining the optimum locations for jog insertion; the algorithms developed will have to be heuristic in nature.

### 5.2.3. More Complex Design Rules

Python only satisfies spacing rules. The types of layout rules found in industry semiconductor processes include:

- Reflection Design Rules
- Mutual Capacitance Rules
- Minimum Width Rules
- Minimum Area Rules

- Minimum Enclosure Rules
- Minimum Overlap Rules
- Contact Design Rules

The reflection design rules state that when certain line types cross certain other line types at right angles (*e.g.* metal lines crossing diffusion lines in an NMOS process), the line which is applied second in the semiconductor process must be *shrunk* where the two lines overlap. These rules stem from an unfortunate byproduct of the photolithographic process used to build mask layers on the IC. The three-dimensional effects of patterning devices on the silicon cause blooming to occur when the photoresist for a line type is run at right angles over the valley created by the processing for a previous layer. If the crossing line is narrowed down over the valley, the blooming and the narrowed line offset, and the net result is a line of almost constant width over the valley.

Mutual capacitance rules require long parallel runs of interconnect lines to be further apart than the minimum spacing. This extra space is required to minimize the mutual capacitance that might induce crosstalk between the signals on the two lines. Implementation of this design rule could be added easily to Python. When overlaps are determined in the spacing rule analysis, the magnitude of the overlap could be used to index into a three dimensional spacing rule table. The first two dimensions relate the spacing rules for the various mask layers as in the normal case. The third additional dimension relates length of overlap to spacing rule. Thus, when comparing the spacing requirement between two adjacent metal lines for example, the length of the overlap would be used to determine an appropriate spacing requirement which keeps this capacitance within acceptable limits.

The minimum width and minimum area layout rules can be processed during the input phase, before the spacing program is invoked. When a user requests a line or geometry of less than minimum width, the input editor can either refuse the request (with an appropriate error message), or increase the width to the minimum width allowed and warn the user. Minimum area violations would be detected when the graphical specification of each geometry was completed; only when the exact dimensions of the geometry are known is it possible to discover all minimum area violations.

The minimum enclosure rules and the minimum overlap rules are used most heavily in systems that use the *sticks* form of symbolic layout, where devices are implied by the crossing of the appropriate line types. Minimum enclosure rules define the distance one mask layer (such as contact cut) must be contained with the mask layer surrounding it (such as metal or polysilicon). Minimum overlap rules define the minimum extension of two line types on either side of an intended intersection. This rule is used most often for transistors in the particular example of an NMOS process. Both the minimum enclosure rules and the minimum overlap rules are part of the construction of primitives in the IC technology and are included as a part of the leaf cells in the input editor. Since Python (after CABBAGE) models the symbolic data with *explicitly* placed primitives, such as transistors and contacts, including these layout rules in the design of the primitives is the obvious thing to do.

There are several different types of layout rules associated with contacts between interconnect lines on different mask layers. Aside from the minimum enclosure rule mentioned above, there are two cases which require changing the shape of the contact to ensure correct electrical operation.

The first case concerns polysilicon-metal and diffusion-metal contacts. At the stage where the metal layer is added to the IC, the polysilicon or diffusion is opened to the metal via the contact cut. The vertical depth of the contact cut is very great; it is on the order of one half the minimum geometry. The walls of the depression for the contact are very steep. To insure proper metal coverage of the step, the size of the metal cap of the contact is often increased. The second case concerns a polysilicon-diffusion buried contact. In order to prevent the formation of an active transistor should the contact window misalign, the contact window is lengthened along the direction of the diffusion line.

Again, both of these layout rules can be dealt with during the input phase. No absolute topology changes are made in Python, so the contacts will retain their original shape throughout the complete spacing process. Addition of the dynamic determination of the contact sizes, as done in CABBAGE, is a complication that is best left to the input editor.

### 5.2.4. Complex Terminal Frames

Python only allows rectangles as terminal frames which are the allowable interconnection areas between cells and a higher-level representation. A slightly more powerful data structure can allow arbitrary Manhattan polygons as terminal frames. This can have an advantage as seen in Fig. 5.2. During the first Y spacing, the interconnect and attached fixed geometry can slide to the top of the first segment of boxes. During the successive X spacing, the endpoint of the interconnect is free to slide so that the following Y spacing will allow the interconnect to slide up further, yielding a final result that is more compact than if the terminals had been restricted to boxes.

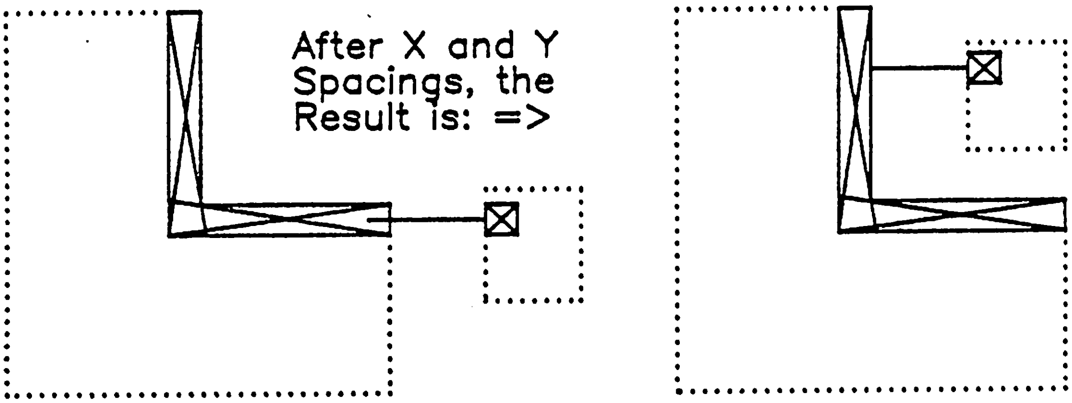


Fig. 5.2 Spacing Savings Resulting From Complex Terminal Frame

Note that although more than one terminal can have the same name (and hence be electrically equivalent) in the definition of a cell, factoring polygonal terminal frames does not allow the same flexibility as having true polygons for terminals. This happens because each interconnect line is bound to the terminal it ends on and cannot change terminals during the spacing analyses, even if the new terminals are electrically equivalent.

The terminal frame edges are used to label the appropriate protection frame intervals with the correct, non-zero net. If the edges for terminals are managed in much the same way as the polygon edges for protection frames, the example of Fig. 5.2 is possible. Since polygons are stored in the database as boxes, the *Fang* polygon package must be used to merge the boxes into true polygons and to develop the individual edges of the polygon. These edges may then be used to compare against the protection frame edges, as the edges of the box terminals are compared now.

### 5.2.5. Constraints (Fixed and Relative)

No user-defined constraints are allowed by Python. This restriction is primarily due to the lack of adequate heuristic algorithms to determine where overconstrained situations exist in the graph. It is possible to intentionally generate overconstraints if user-defined constrains are permitted. It is almost as easy to introduce *unintentional* problems. Adding a mechanism to allow user specified constraints is simple. Lines could be labeled as fixed by the input editor. Normally, an edge is added for each line between the two vertices that are in the graph parallel to the direction of the line. This edge has zero lower bound, and infinite upper bound, and serves to keep the length of the line greater than zero. In fixed lines, the lower and upper bounds of the edge are set equal to the length of the line as read in from the input editor. The length of the line is thus fixed at the input value, and remains at the same length throughout all spacing iterations.

Given this capability of fixed length lines, relative constraints between two objects can be expressed as follows. Define a new mask layer for constraints only; a mask layer with no physical meaning; similar to the *RUNX* mask layer used in CABBAGE. Add a terminal on this new mask layer for each instance. The geometries which can be added on this new mask layer allow the specification of many types of relative constraints. Connecting two (or more) instances with fixed length lines on the constraint layer fixes the two (or more) objects rigidly relative to one another. They may move anywhere within the IC layout, subject to the spacing constraints. They must move with one another, however. Figure 5.3 displays how two objects would be fixed relative to one another.

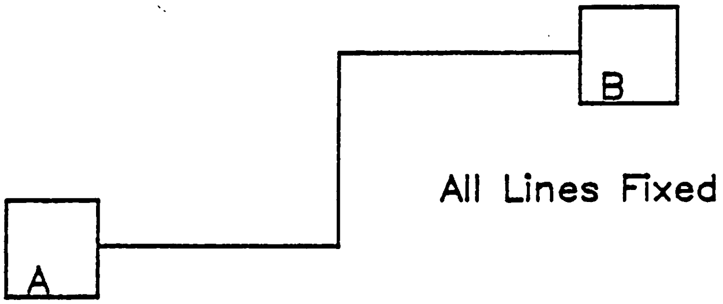


Fig. 5.3 Rigid Relative Constraints

Constraints in one direction only can be made by connecting two objects together with a fixed line in that direction and an unconstrained line in the other direction. Figures 5.4 and 5.5 show objects constrained in the X and Y directions only, respectively.

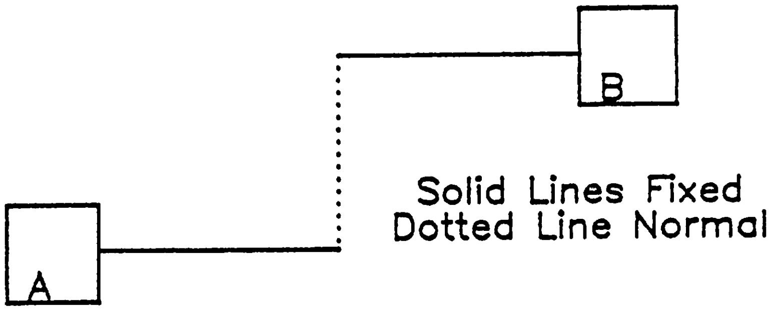


Fig. 5.4 X Only Relative Constraint

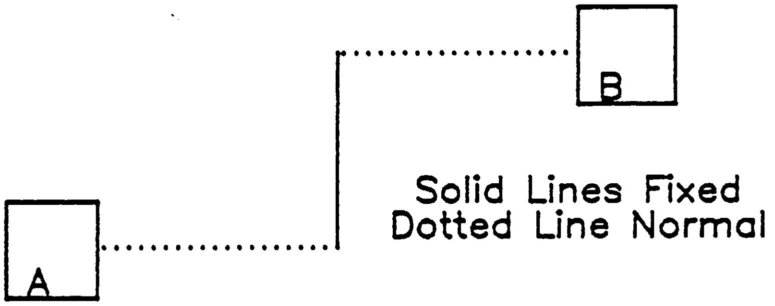


Fig. 5.5 Y Only Relative Constraint

### 5.2.6. 45-Degree Interconnect

45-degree interconnections are not allowed in Python, however they are used heavily in many IC technologies. They are almost impossible to add to Python because of the separation of the X and Y spacings. Stretching and shrinking 45 degree lines necessitates movement in both axis directions (See Fig. 5.6). The decoupling of the spacing process introduces a high probability of an oscillatory condition that will never converge if 45 degree interconnect lines are allowed. Figure 5.7 illustrates one such example. During the first X compaction, object B moves to the left, because of the spacing requirement between it and object C. This *horizontal* movement implies a *vertical* movement, in order to keep the angle of the line at 45 degrees. In the example shown, the required vertical movement violates the spacing rule between objects B and D in the Y direction. During the subsequent Y spacing, B and D are spaced farther apart. This *vertical* movement implies a necessary *horizontal* movement, which moves B back to its original position. The next X spacing moves B to the left (and hence *down*) and the resulting oscillatory

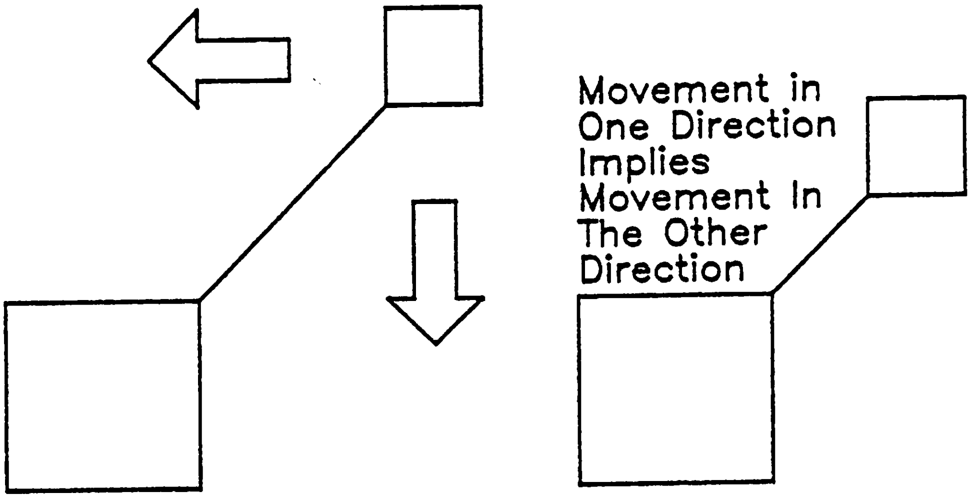


Fig. 5.6 Shrinking a 45 Degree Line

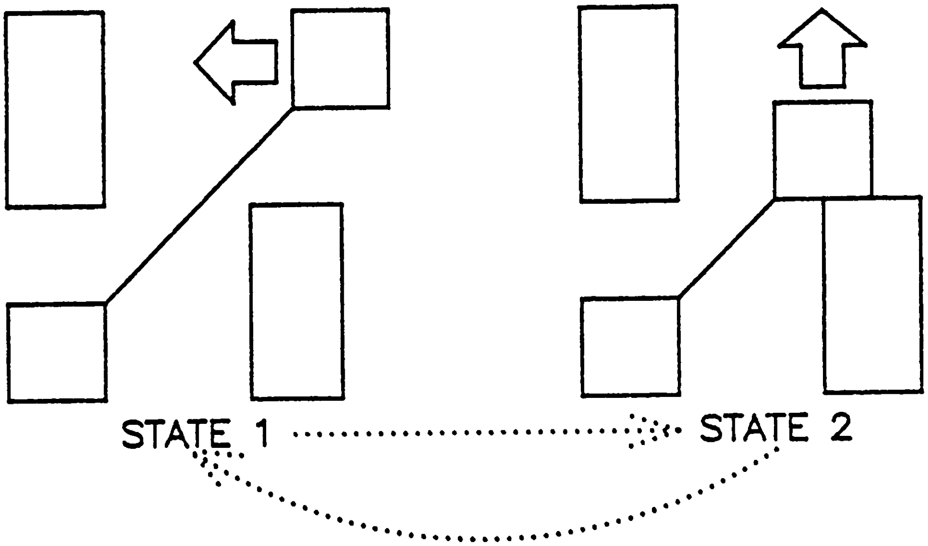


Fig. 5.7 Oscillatory 45 Degree Example

condition will never converge. What is required to determine the proper location for objects connected with 45 degree lines is the surrounding geometry *in both directions at the same time.*

### 5.2.7. Incremental (Interactive) Spacing

This is an interesting idea from the point of view of program implementation. Algorithms for incremental graph update and solution must be developed before this idea could be exploited. Given the use of a true hierarchy, interactive spacing may be of debatable usefulness. In a limited way however, it might find use in technologies where the size of devices is very closely related to the actual parasitics of the circuit. Only when the circuit is compacted can the parasitics be accurately extracted. These values might then be used to compute the size of devices yet to be added.

## 5.3. Acknowledgements

Several people have contributed to Python, both by making direct technical contributions to the ideas and concepts implemented in the program, and by contributing related software utilities which allow Python to fit in well as part of an IC design *system*. Ken Keller provided the Squid database, and the graphical input capabilities of Hawk. Peter Moore provided the Fang polygon package. Dan Fitzpatrick provided the *CADRC* package. Professor Karp pointed to the proper references which contained the information necessary to derive the longest path algorithm for Python. Numerous discussions with Ken Keller, Professor Richard Newton, Professor Alberto Sangiovanni-Vincentelli, Peter Moore, Tom Quarles, Jim Kleckner, and other members of the IC CAD group at Berkeley were invaluable. Discussions with Min-Yu Hsueh, the author of the CABBAGE program, with Jack Hurt, Ian Getreu, and John Crawford of Tektronix, and with Wayne Wolf, and Bill McCalla of Hewlett Packard were very helpful. Discussions with Professor John Ousterhaut and Dave Wallace of Computer Science at Berkeley stimulated thought as to new approaches to spacing. Finally, special thanks are due to

Professor Richard Newton, who provided the encouragement and motivation which prompted this work. This work was supported in part by Hewlett-Packard Co., Corporate Design Aids, Tektronix, Inc., and the Digital Equipment Corporation. Their assistance is gratefully acknowledged.

# APPENDIX A

## User Manual

This appendix contains the UNIX† manual pages for the stand-alone version of *Python*.

†UNIX is a Trademark of Bell Laboratories.

## NAME

python – A spacing program for symbolic integrated circuit layouts

## SYNOPSIS

**python** [options] squidcell [ [options] squidcell ] ...

## DESCRIPTION

*Python* is a successor to the *CABBAGE I* program. It takes as input the symbolic layout of an integrated circuit cell, and adjusts the locations of objects within the cell to obtain as output the minimal area representation of the cell which satisfies all spacing rules. The interconnect between instances is stretched or shrunk to satisfy the spacing rules.

The available options are:

**-i <inview>** Change input view from default 'layout' to '<inview>'.

**-o <outview>** Change output view from default 'spaced' to '<outview>'.

**-r <rulename>** Change rules name (in .cadrc file) from default 'python' to '<rulename>'.

**-e <errorfile>** Change error file from default 'pythonerr' to '<errorfile>'.

**-m#** Specify a number # limit to the number of iterations. Default is 40. Specifying no number, or a number less than zero means to iterate until the solution is reached or the existence of an overconstraint is discovered.

**-R** Do not perform a reverse pass spacing. Normally, the minimum and maximum positions of all objects are calculated. Each object is then placed at the average of its minimum and maximum allowable locations. Using this option prevents the determination of the maximum allowable positions of the objects. All objects are placed at their minimum location. This has the visual effect of pulling all objects toward the lower left corner of the layout as much as is possible. Using this option saves time for the spacing process.

**-y** Perform the initial spacing in the y (vertical) direction. Normally, the compaction is performed as successive iterations in the x and y directions. Using this option causes the first iteration to start with the y direction. The initial direction of spacing has a great deal to do with the final aspect ratio of the spaced cell.

## FILES

| | |
|---|---|
| ~cad/.cadrc | – Read first for spacing rules |
| ~/.cadrc | – Read last for spacing rules |
| pythonerr | – Default error ( and statistics ) file |
| <squidcell>/layout | – Default input view |
| <squidcell>/spaced | – Default output view |

## SEE ALSO

hawk(cad), squid(3cad), fang(3cad), cabtosquid(cad)

## AUTHOR

Mark Bales

## DIAGNOSTICS

All error messages are self explanatory. They appear in the error file, which is 'pythonerr' by default.

## BUGS

It is possible to generate overconstraining conditions by ending too many lines in a single terminal frame. The program does not detect this.

Detection of *the existence* of overconstraining conditions takes $O( v )$ iterations, where $v$ is approximately twice the number of objects in the cell being spaced. To space a cell with no overconstraining condition takes < 40 iterations. Thus, detection of *the existence* of overconstraining conditions in large cells takes much longer than spacing of the same cell without the overconstraint. Detection of where the overconstraints are takes $O( 2\sim v )$ time, and is hence unfeasible for large cells.

Merging is not currently allowed. This reduces greatly the area efficiency the program can obtain when spacing a layout.

The interfacing between *python* and other programs in the design system is still a bit primitive.

# APPENDIX B

# GRAPHCAP

This appendix contains the section 3 and 5 manuals from UNIX† for *graphcap*, a low level database interface for terminal independent graphics.

†UNIX is a Trademark of Bell Laboratories.

**NAME**

ggetent, ggetnum, ggetflag, ggetstr, gencod, gdecod, gputs – terminal independent graphics operation routines

**SYNOPSIS**

```
char PC;
char *BC;
char *UP;
short ospeed;

ggetent(bp, name)
char *bp, *name;

ggetnum(id)
char *id;

ggetflag(id)
char *id;

char *
ggetstr(id, area)
char *id, **area;

char *
gencod(pm, x, y, z, t)
char *pm;

gdecod(pm, x, y, z, t)
char *pm;
int *x, *y, *z, *t;

gputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

**DESCRIPTION**

These functions extract and use capabilities stored in the terminal graphics capability data base file *graphcap*(5). These are low level routines; see *mfb*(3) for a higher level package.

*Ggetent* extracts the entry for terminal *name* into the buffer at *bp*. *Bp* should be a character buffer of length 4096 and must be retained through all subsequent calls to *ggetnum*, *ggetflag*, and *ggetstr*. *Ggetent* will look in the environment for a GRAPHCAP variable. If found, and the value does not begin with a slash, and the terminal type name is the same as the environment string TERM, the GRAPHCAP string is used as the entry, instead of reading the graphcap file. If it does begin with a slash, the string is used as a path name rather than */cad/etc/graphcap*. This can speed up entry into programs that call *ggetent*, as well as to help debug new terminal descriptions or to make one for your terminal if you don't have write permission on the file */cad/etc/graphcap*. *Ggetent* returns −1 if it cannot open the graphcap file, 0 if the terminal name given does not have an entry, and 1 if all goes well.

*Ggetnum* gets the numeric value of capability *id*, returning 0 if is not given for the terminal. *Ggetflag* returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. *Ggetstr* gets the string value of capability *id*, placing it in the buffer at *area*, advancing the *area* pointer. It decodes the abbreviations for this field described in *GRAPHCAP*(5), except for sequences beginning with ",which are dynamically interpreted by the routines *gencod* and *gdecod*.

*Gencod* returns a pointer to a formatted string using the values x, y, z, and t and using the string *pm* as a formatting template. See *GRAPHCAP*(5), for a description of the formatting conventions. (Note that all programs using *graphcap* should turn off XTABS, since gencod may now output a tab. Note that programs using graphcap should in general turn off XTABS anyway since some terminals use control I for other functions, such as nondestructive space.) If a % sequence is given which is not understood, then *gencod* returns OOPS.

*Gdecod* returns the values *x*, *y*, *z*, and *t* having decoded them from the input stream using the format string *pm*. The same formatting conventions are followed as in *gencod.*

*Gputs* decodes the leading padding information of the string *cp; affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine which is called with each character in turn. The external variable **ospeed** should contain the output speed of the terminal as encoded by *stty (2)*. The external variable PC should contain a pad character to be used (from the **pc** capability) if a null (^@) is inappropriate.

**FILES**

/cad/etc/graphcap data base

**SEE ALSO**

kic(1), mfb(3), graphcap(5)

**AUTHOR**

Mark Bales (Much of it stolen from *termcap(3).)* Giles Billingsly is taking over development

**BUGS**

The pointer returned by *gencod* points to a static buffer area which is overwritten upon each call.

## NAME

graphcap – graphics terminal capability data base

## SYNOPSIS

/cad/etc/graphcap

## DESCRIPTION

*Graphcap* is a data base describing graphics terminals, used, *e.g.*, by *kic*(1) and *mfb*(3). Terminals are described in *graphcap* by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in *graphcap*.

Entries in *graphcap* consist of a number of ':' separated fields. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name is always 2 characters long and is used by older version 6 systems which store the terminal type in a 16 bit word in a systemwide data base. The second name given is the most common abbreviation for the terminal, and the last name given should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may well contain blanks for readability.

## CAPABILITIES

The Parms column indicates which of the four possible paramters are used in the encoding of string variable.

| **Name** | | **Type** | **ParmsDescription** |
|---|---|---|---|
| ACS | string | | Alphanumeric Clear Screen |
| APT | boolean | | Accurately Positionable Text |
| BDE | string | X | Box Pattern Define End |
| BDF | string | X | Box pattern Define Format |
| BDH | numeric | | Box Definition Height ( number of rows ) |
| BDR | boolean | | Box Definition Row major |
| BDS | string | X | Box pattern Define Start |
| BDW | numeric | | Box Definition Width ( number of columns ) |
| BLD | boolean | | BLinkers Defineable |
| BLE | string | | BLinkers End |
| BLS | string | XYZT | BLinkers Start |
| BX0 | string | | BoX type 0 ( solid ) |
| BX1 | string | | BoX type 1 |
| BX2 | string | | BoX type 2 |
| BX3 | string | | BoX type 3 |
| BX4 | string | | BoX type 4 |
| BX5 | string | | BoX type 5 |
| BX6 | string | | BoX type 6 |
| BX7 | string | | BoX type 7 |
| CBK | string | | set to Color BlacK |
| CBU | string | | set to Color BlUe |
| CCY | string | | set to Color CYan |
| CGN | string | | set to Color GreeN |
| CHO | numeric | | Character Height Offset |
| CMD | boolean | | Character Mode Destructive |
| CMG | string | | set to Color MaGenta |
| CMN | boolean | | Character Mode Non-destructive |
| CRD | string | | set to Color ReD |
| CWH | string | | set to Color WHite |

| | | | |
|---|---|---|---|
| CWO | numeric | | Character Width Offset |
| CYL | string | | set to Color YeLlow |
| DBP | boolean | | Defineable Box Patterns |
| DBS | string | XYZT | Draw Box Sequence |
| DLP | boolean | | Defineable Line Patterns |
| DLS | string | XY | Draw Line Sequence |
| DMO | string | | Destructive Mode On |
| GCD | numeric | | Graphcis Clear screen Delay |
| GCH | numeric | | Graphics Character Height |
| GCS | string | X | Graphics Clear Screen (in current color) |
| GCW | numeric | | Graphics Character Width |
| GFD | numeric | | Graphics Finish Delay |
| GFE | string | | Graphics Finish End |
| GFS | string | | Graphics Finish Start |
| GID | numeric | | Graphics Initialization Delay |
| GIE | string | | Graphics Initialization at End |
| GIS | string | | Graphics Initialization at Start |
| GME | string | | Graphics Mode End |
| GMS | string | | Graphics Mode Start |
| GPC | string | | Graphics Pad Character (default is NULL) |
| GTE | string | | Graphics Text End |
| GTS | string | | Graphics Text Start |
| IBS | string | | Initialize predefined Box Styles |
| ICS | string | | Initialize predefined Color Styles |
| ILS | string | | Initialize predefined Line Styles |
| LD3 | string | | Line type dot dot dashed |
| LDD | string | | Line type long Dot Dashed |
| LDO | string | | Line type DOtted |
| LLD | string | | Line type Long Dashed |
| LPD | string | | Line Pattern Define |
| LPW | numeric | | Line Pattern Width (in bits) |
| LSO | string | | Line type SOlid |
| Ld3 | string | | Line type dot dot dot |
| Ldd | string | | Line type short dot dashed |
| Lsd | string | | Line type short dashed |
| MPS | string | XY | Move Pen Sequence |
| MXC | numeric | | Maximum X Coordinate |
| MYC | numeric | | Maximum Y Coordinate |
| NBL | numeric | | Number of BLinkers |
| NBS | numeric | | Number of Box Styles |
| NCS | numeric | | Number of Color Styles |
| NLS | numeric | | Number of Line Styles |
| NMO | string | | Non-destructive Mode On |
| NPB | numeric | | Number of Pointing device Buttons |
| PDB | boolean | | Pointing Device has Buttons |
| PDE | string | | Pointing Device End |
| PDF | string | XYZT | Pointing Device coordinate Format |
| PDR | string | | Pointing Device initiate Read |
| PDS | string | | Pointing Device Start |
| PRI | boolean | | Pointing Read Immediately returns coordinates |
| RLS | boolean | | Reissue Line Style before each line |
| SBS | string | X | Set Box Style |

| | | | |
|---|---|---|---|
| SCS | string | X | Set Color Style |
| SLS | string | X | Set Line Style |
| TOH | string | | Text Orientation Horizontal |
| TOV | string | | Text Orientation Vertical (read from bottom) |
| VLT | boolean | | Video Lookup Table present |
| VTE | string | XYZT | Video Table Entry |
| VTI | numeric | | Video Table maximum Intensity |
| VTL | numeric | | Video lookup Table Length |
| VWM | string | X | Video Write Mask |

### A Sample Entry

The following entry, which describes the Aed 512, is among the more complex entries in the *graphcap* file as of this writing. (This particular aed entry may be outdated, and is used as an example only.)

```
a1|aed|aedj|aed512|Advanced Electronics Design Model 512:\
        :APT:CHO#3:CWO#1:CMN:MXC#511:MYC#482:GCH#9:GCW#6:NCS#256:\
        :NLS#1:NBS#16:VLT:VTI#255:VTL#256:VTE=K%X%h201%Y%h2%Z%h2%T%h2:\
        :SCS=C%X%h2[%X%h2:VWM=L%X%h2:GMS=\E:GME=^A:GCS='\120~L\E'\160:\
        :GTS=^A:GTE=\E:GIS=\E\E\E\E\E\E\E\E\E\E0:GID#2:\
        :GIE=\E'\160\~15\06\011LG1HHH.:\
        :GFS=\E\E\E\E\E\E\E\E\E\E0:GFE=\E'\100~L^A:\
        :GFD#2:GCD#0:BLD:NBL#8:BLS=\E4%X%h2%Y%h2%Z%h2%T%h21D1D:\
        :BLE=\E4%X%h2%Y%h2%Z%h2%T%h21D00:\
        :DLS=Q%X%>>#6%&#0xC%R%Y%>>#8%|%R%h1%X%h2%Y%h2A%Z%>>#6%&#0xC%
X%>>#6%&#0xC%R%Y%>>#8%|%R%h1%X%h2%Y%h2o%Z%>>#6%&#0xC%R%T%>>#8%|%R%h1%Z%h2%T%h2:\
        :MPS=Q%X%>>#6%&#0xC%R%Y%>>#8%|%R%h1%X%h2%Y%h2:\
        :ICS=K0008000000000FF00FFFF00FF00FF00FFFF0000FFFFFF00FFFF:ILS=~@:\
        :CBK=C00[00:CBU=C01[01:CCY=C02[02:CGN=C03[03:\
        :CMG=C04[04:CRD=C05[05:CWH=C06[06:CYL=C07[07:\
        :EBK='\120~L\E'\160:EBU='\120~L\E'\160:ECY='\120~L\E'\160:\
        :EGN='\120~L\E'\160:EMG='\120~L\E'\160:ERD='\120~L\E'\160:\
        :EWH='\120~L\E'\160:EYL='\120~L\E'\160:\
        :LD3=01AE55:LDD=01BE11:LDO=01AA55:LLD=01FC11:\
        :LSO=01FF55:Ld3=01A855:Ldd=01BE55:Lsd=01FC55:\
        :PDS=c%X%h2%Y%h202U:PDE=d:PDR=j:PRI:\
        :PDF=%h1%<<#8%R%>>#2%&#0x300%X%h2%|%X%X%h2%|%R%&#0x3FF%Y:\
        :BDE=~@:BDF=0%Y%h2:BDH#8:BDR:BDS=,%X%h2:BDW#8:BX0="00:\
        :BX1="01:BX2="02:BX3="03:BX4="04:BX5="05:BX6="06:BX7="07:\
        :DBP:DLP:IBS=,00FFFFFFFFFFFFFFFF,01FE82BAAAA2BE80FF,021824428181422
        :LPW#8:LPD=01%X%h255:SBS="%X%h2:SLS=~@:
```

Entries may continue onto multiple lines by giving a \ as the last character of a line, and that empty fields may be included for readability (here between the last field on a line and the first field on the next). Capabilities in *graphcap* are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

### Types of Capabilities

All capabilities have three letter codes. For instance, the fact that the Aed has accurately positionable text ( *i.e.*, graphics text may be positioned with lower left corner at any pixel on the screen ) is indicated by the capability **APT**. Hence the description of the Aed includes **APT**. Numeric capabilities are followed by the character '#' and then the value. Thus **MXC** which indicates the maximum value of the X coordinate on the terminal screen gives the value '511' for the Aed. **Formatting in String Capabilities**

Most of the string variables have a primitive formatting capability to be used in encoding numbers into ASCII strings and decoding ASCII strings into numbers. An example of the former si the capability **DBS** ( for Draw Box Sequence ), which takes four numbers (X, Y, Z, and T) and generates the proper sequence to draw a box from lower left corner (X,Y) to upper right corner (Z,T). An example of a string decode is the capability **PDF** ( for Pointing Device Format ), which takes an ASCII string from the input stream and extracts from it an x and y coordinate, a key (if one was pushed) and a buttonmask (if a cursor button was pushed).

**String Formatting**

Most of the string variables have a primitive formatting capability which uses four variables (X, Y, Z, and T) to generate a formatted string (with *gencod),* or generates four variables (X, Y, Z, and T) from a formatted string (with *gdecod).* All operations begin with a percent sign '', and they are listed below:

| Com | Command Description encod/(decod) |
|---|---|
| %X | set value/(X variable) to the X variable/(value) |
| %Y | set value/(Y variable) to the Y variable/(value) |
| %Z | set value/(Z variable) to the Z variable/(value) |
| %T | set value/(T variable) to the T variable/(value) |
| %d | output/(input) value in variable length decimal format |
| %d2 | output/(input) value converting to/(from) two decimal digits |
| %d3 | output/(input) value converting to/(from) three decimal digits |
| %. | output/(input) least significant byte of value withoutconversions |
| %h1 | output/(input) least significant four bits converting to/(from) one ASCII hex character |
| %h2 | output/(input) least significant byte converting to/(from) two ASCII hex characters |
| %h3 | output/(input) least significant twelve bits converting to/(from) three ASCII hex characters |
| %h4 | output/(input) least significant sixteen bits converting to/(from) four ASCII hex characters |
| %o1 | output/(input) least significant three bits converting to/(from) one ASCII octal character |
| %o2 | output/(input) least significant six bits converting to/(from) two ASCII octal characters |
| %o3 | output/(input) least significant nine bits converting to/(from) three ASCII octal characters |
| %o4 | output/(input) least significant twelve bits converting to/(from) four ASCII octal characters |
| %o5 | output/(input) least significant fifteen bits converting to/(from) five ASCII octal characters |
| %o6 | output/(input) least significant sixteen bits converting to/(from) six ASCII octal characters |
| %R | store/(retrieve) value in/(from) a temporary register |
| %+x | add x to value |

| | |
|---|---|
| %-x | subtract x from value |
| %>>x | shift value right by x bits |
| %<<x | shift value left by x bits |
| %\|x | OR x with value |
| %&x | AND x with value |
| %^x | XOR x with value |
| %~ | Complement value ( 1's complement ) |
| %% | gives % |
| %B | BCD (2 decimal digits encoded in one byte) |
| %D | Delta Data (backwards bcd) |

Where x can be:

(1) One byte - the numeric value of this byte is used as x

(2) The character "#" followed by an ASCII number which is
Hex if the first character sequence is '0x' or '0X',
octal if the first digit is 0, and decimal otherwise.

(3) The character "%" followed by X, Y, Z, T, or R - the value
of X, Y, Z, T, or R respectively is used as x.

These formatting commands are very similar to those found in *termcap(5)*, but are more numerous due to the more rigorous requirements of graphics terminals.

**Miscellaneous** See *mfb(5) for a description of how*

**FILES**

/cad/etc/graphcap file containing terminal descriptions

**SEE ALSO**

kic(1), mfb(3), graphcap(3)

**AUTHOR**

Mark Bales ( Much of it stolen from termcap(3) ) Giles Billingsly is taking over development

**BUGS**

# APPENDIX C

# MFB

This appendix contains the section 3 and 5 manuals from UNIX† for *mfb*, a medium level *M*odel *F*rame *B*uffer which interfaces between a high-level application program and the low-level *graphcap* routines.

†UNIX is a Trademark of Bell Laboratories.

## NAME

mfb – a Model Frame Buffer graphics package

## SYNOPSIS

**MFBBegin(displayName)**
char *displayName;

**MFBEnd()**

**MFBSetNaiveMode()**

**MFBUpdate()**

**MFBDefineColor(colorId,r,g,b)**
int colorId, r, g, b;

**MFBDefineLineStyle(styleId,styleDefinition)**
int styleId, styleDefinition;

**MFBDefineBoxStyle(styleId,styleDefinition)**
int styleId, *styleDefinition;

**MFBSetColor(colorId)**
int colorId;

**MFBSetLineStyle(styleId)**
int styleId;

**MFBSetBoxStyle(styleId)**
int styleId;

**MFBSetChannelMask(channelMask)**
int channelMask;

**MFBLine(x1,y1,x2,y2)**
int x1, y1, x2, y2;

**MFBBox(l,b,r,t)**
int l, b, r, t;

**MFBText(text,x,y)**
char *text;
int x, y;

**MFBSetTextMode(destructiveP,colorId)**
int destructiveP, colorId;

**MFBPoint(x,y,key,buttonMask)**
int *x, *y, *buttonMask;
char *key;

**MFBSetCursor(colorId);**
int colorId;

**MFBError(errnum,termname)**
int errnum;
char *termname;

**MFBBlinker(colorId,r,g,b,onP)**
int colorId, r, g, b, onP;

**MFBPutchar(c)**
char c;

**char MFBGetchar()**

**MFBUngetchar()**

**MFBTrap()**

**MFBFlood()**

**DESCRIPTION**

These functions form the core set of a medium-level terminal independent graphics package. They use the low level routines in *graphcap(3)* to achieve terminal independence, and provide a basis for writing high-level graphics packages. See *mfb(5)* for a detailed introduction to the Model Frame Buffer.

*MFBBegin* is called to initialize the package for terminal *displayName*. It reads the set of capabilities from the terminal database, turns off user messages, sets the terminal in cbreak mode, and diverts all signals to call the routine *MFBTrap* if a termination signal is encountered. *MFBTrap* calls *MFBEnd*, and then calls *exit(2)*.

*MFBEnd* must be the last routine called before program exit, and resets the terminal parameters to their state at run time.

*MFBSetNaiveMode* establishes a simple mode of operation with eight colors, eight line types, and eight box styles predefined ( within the limitations of the terminal ).

*MFBUpdate* flushes the buffer used to improve system efficiency. It should be called whenever a sequence of commands is considered complete, such as at the end of a plot or before a read.

*MFBDefineColor* sets the color identified by *colorId* to the value defined by the intensity triple *r, g, b*. The intensities are normalized to 1000. *MFBDefineLineStyle* sets the line style identified by *styleId* to the bit pattern contained in *styleDefinition*. The pattern is taken from the low order bits. The length of the pattern depends upon the terminal used. *MFBDefineBoxSlyle* sets the box style identified by *styleId* to the bit pattern pointed to by *styleDefinition*. The pattern is an array of integers which provide bit patterns for the individual rows (columns) of the pattern. The pattern may be row or column major, depending on the terminal, and it may be of different bit widths and heights.

*MFBSetColor*, *MFBSetLineStyle*, and *MFBSetBoxStyle* set the current color, linestyle, and boxstyle respectively. These attributes are used when drawing lines, boxes, and text.

*MFBLine* draws a line from point *(x1,y1)* to point *(x2,y2)*. *MFBBox* draws a box from the point *(left, bottom)* to the point *(right, top)*. Note that the points must be given in the proper order. This is due to the idiosyncrasies of some graphics terminals.

*MFBSetTextMode* sets the text writing mode to either *destructive*, with the background bits in each character cell of the text being erased, or *nondestructive*, where these background bits are not changed. Not all terminals will have both capabilities. *MFBText* is the routine used to output the string *text* with the lower left hand corner at the point *(x,y)*.

*MFBPoint* is used to read a pointing device associated with the terminal ( a tablet, joystick, mouse, etc. ). It turns the graphics cursor on and decodes the cursor position, returning it in *x* and *y*. The routine also determines if a button on the cursor was pressed ( if such buttons exist ), or if a key on the terminal keyboard was depressed. This information is returned in *key* and *buttonMask*. After the cursor position has been entered, the cursor itself is turned off.

*MFBSetCursor sets the color of the* in *MFBPoint* to *colorId*.

*MFBBlinker* is for those few color graphics terminals which allow certain colors to blink between two colors. The color identified by *colorId* is made to blink between the color it is set to and the color defined by the triple *r*, *g*, *b*. Again, the intensities are normalized to 1000. The flag *onP* determines whether the blinker is being set or cleared.

*MFBFlood* floods the screen with the current color as defined in *MFBDefineColor*.

*MFBError* is used with the error indications returned by the routines *MFBBegin*, *MFBEnd*, *MFBSetColor*, *MFBSetLineStyle*, *MFBSetBoxStyle*, *MFBPoint*, and *MFBBlinker*. Passing the returned error status *errnum* along with *displayName* yields a pointer to a formatted string containing an error message which explains the problem. If no error is encountered, the indication MFBOK is returned. See mfb(5) for a more detailed explanation of the errors that can occur.

*MFBPutchar*, *MFBGetchar*, and *MFBUngetchar* replace the familiar I/O functions for use with this package.

**FILES**

/cad/etc/graphcap terminal database<br>
/cad/include/mfb.h file defining the MFB structure

**SEE ALSO**

kic(1), graphcap(3), mfb(5)

**AUTHOR**

Mark Bales and Ken Keller Giles Billingsly is taking over development

**DIAGNOSTICS**

See the description of *MFBError* above.

**BUGS**

## NAME

mfb – a Model Frame Buffer intermediate-level graphics package

## SYNOPSIS

```
MFBBegin(displayName)
char *displayName;

MFBEnd()

MFBSetNaiveMode()

MFBUpdate()

MFBDefineColor(colorId,r,g,b)
int colorId, r, g, b;

MFBDefineLineStyle(styleId,styleDefinition)
int styleId, styleDefinition;

MFBDefineBoxStyle(styleId,styleDefinition)
int styleId, *styleDefinition;

MFBSetColor(colorId)
int colorId;

MFBSetLineStyle(styleId)
int styleId;

MFBSetBoxStyle(styleId)
int styleId;

MFBSetChannelMask(channelMask)
int channelMask;

MFBLine(x1,y1,x2,y2)
int x1, y1, x2, y2;

MFBBox(l,b,r,t)
int l, b, r, t;

MFBText(text,x,y)
char *text;
int x, y;

MFBSetTextMode(destructiveP,colorId)
int destructiveP, colorId;

MFBPoint(x,y,key,buttonMask)
int *x, *y, *buttonMask;
char *key;

MFBSetCursor(colorId);
int colorId;

MFBError(errnum,termname)
int errnum;
char *termname;

MFBBlinker(colorId,r,g,b,onP)
int colorId, r, g, b, onP;

MFBPutchar(c)
char c;
```

**char MFBGetchar()**

**MFBUngetchar()**

**MFBTrap()**

**MFBFlood()**

## DESCRIPTION

*Mfb* is a medium level terminal independent graphics package intended as a basis for high level graphics packages. It uses the graphics terminal database management routines in *graphcap(3)* and provides for the high level user an unintelligent interface to a predefined set of operations, as well as the relevant additional information necessary to make the high level package truly terminal independent.

This document describes the data structure that defines the Model Frame Buffer. For a shorter description of the *mfb* routines, see *mfb(3)*.

### The MFB Data Structure

Here is the *mfb* data structure, which will be broken up and discussed in separate components.

```
/* MFB structure definition. */

struct MFB
 {
 int initializedP;

 int naiveModeP;

 /*Initialization*/
 char *startSequence1, *startSequence2, *endSequence1, *endSequence2;
 unsigned startDelayTime, endDelayTime;        /* In Seconds */

 /*Resolution.*/
 int maxX,maxY;

 /*Text font.*/
 int textPositionableP;
 int fontHeight,fontWidth,fontXOffset,fontYOffset;
 char *graphicsTextStart, *graphicsTextEnd;
 int destructiveP,orP;
 char *destructiveON,*orON;

 int numberOfColors;               /* number of color styles */
 int numberOfLineStyles;  /* number of line styles */
 int numberOfBoxStyles;

 /* Naive user mode parameters */
 int naiveColors, naiveLineStyles, naiveBoxStyles, naiveEraseStyles;
 char *naiveLineInit, *naiveColorInit, *naiveBoxInit;
 char *naiveLineSet[8], *naiveColorSet[8], *naiveBoxSet[8], *naiveErase[8];

 char *displayName;
 char *screenFlood;
```

```
unsigned floodDelayTime;
char *graphicsON;
char *graphicsOFF;

/*true if display has a VLT.*/
int vltP;
/*Max value of red, green, or blue intensity.*/
int maxIntensity;
int widthOfVLT;      /* = Ceiling( log base 2 ( maxIntensity ) * 3 ) */
int lengthOfVLT;
char *VLTentry;
char *setForegroundColor;

/*true if display has a channel mask:  also known as memory
  plane write enable mask.*/
int channelMaskP;
char *channelMaskSet;

/*true if pointing device has buttons.*/
int buttonsP;
int numberOfButtons;
char *enablePointingDevice;
char *disablePointingDevice;
int readImmediateP;
char *readPointingDevice;
char *formatPointingDevice;

/*true if has blinkers.*/
int blinkersP;
int numberOfBlinkers;
char *blinkerON, *blinkerOFF;

char *pointingDeviceName;
int pointingDeviceId;

/* Line drawing parameters. */
int linePatternDefineP;
char *linePatternDefine;
int linePatternWidth;
char *setLineStyle;
int *linePatterns;
char *movePenSequence;
char *drawLineSequence;

/* Box drawing parameters. */
int boxPatternDefineP;
int boxRowMajorP;
int boxDefineHeight;
int boxDefineWidth;
char *boxDefineStart;
char *boxDefineFormat;
char *boxDefineEnd;
```

```
 char *setBoxStyle;
 int *boxPatterns;
 char *drawBoxSequence;

 /* Section of 'current' variables */
 int ForegroundColorId;
 int CursorColor1Id;
 int CursorColor2Id;
 int BoxStyle;
 int LineStyle;
 int *VLTcopy;
 int channelMask;
 int textMode;

 /* Kludge section (sigh!) */
 int reissueLineStyleP;
 };
```

**Naive User Mode**

```
/*  Define Macros for MFB naive user mode. */

/*  Macros for Line Styles. */
#define DOTDOTDASHED      0
#define LONGDOTDASHED     1
#define DOTTED            2
#define LONGDASHED        3
#define SOLID             4
#define DOTDOTDOT 5
#define SHORTDOTDASHED 6
#define SHORTDASHED       7

/*  Macros for Color Styles. */
#define BLACK       0
#define BLUE 1
#define CYAN 2
#define GREEN       3
#define MAGENTA     4
#define RED  5
#define WHITE       6
#define YELLOW      7

/*  Macros for Box Styles. */
#define BOXTYPE0   0
#define BOXTYPE1   1
#define BOXTYPE2   2
#define BOXTYPE3   3
#define BOXTYPE4   4
#define BOXTYPE5   5
#define BOXTYPE6   6
#define BOXTYPE7   7
```

**Error Diagnostics**

```
#define MFBOK 1
#define MFBBADENT -10
#define MFBBADTTY -20
#define MFBBADNLN -30
#define MFBNODFLP -40
#define MFBNODFBP -50
#define MFBBADNCO -60
#define MFBNODFCO -70
#define MFBNOBLNK -80
#define MFBTMBLNK -90
#define MFBBADNBX -100
#define MFBBADSIG -110
#define MFBBADSTT -120
#define MFBBADCHM -130
#define MFBBADWRT -140
#define MFBPNTERR -150
#define MFBNOPTFT -160
```

## FILES

/cad/etc/graphcap terminal database
/cad/include/mfb.h file defining the MFB structure

## SEE ALSO

kic(1), graphcap(3), mfb(3), graphcap(5)

## AUTHOR

Mark Bales and Ken Keller Giles Billingsly is taking over development

## DIAGNOSTICS

## BUGS

# APPENDIX D

## A CPM Algorithm for Acyclic Digraphs with Lower/Upper Bounds

This appendix describes the derivation and implementation of the critical path method (CPM) algorithm used in *Python*. A simple CPM was used in *CABBAGE* [Hsueh79] to solve the critical path problem in a single-source, single-sink (or multiple equivalent sinks) acyclic directed graph with lower bounds on the lengths of the edges [Thesen78]. The constraint graph in Python is different in that upper bounds are placed on the lengths of some edges, in addition to the lower bounds placed on all edges. The new CPM algorithm is an iterative one based on the shortest path problem, and is guaranteed to converge in $O(v^3)$ time and within $O(v)$ iterations. Since this convergence rate is for a completely connected graph, and the graphs generated by integrated circuit mask artwork are far from complete, the expected convergence time is approximately $O(v^{1.5})$.

The spacing problem is decoupled into X and Y graphs, corresponding to the placement of the elements in the X and Y directions. A typical X-graph generated in the spacing process performed by Python has the form shown in Fig. D.1 (although a real graph would be much larger). In order to space the elements (corresponding to vertices in the graph) in a layout-rule correct fashion, it is necessary to find the longest path to each vertex subject to the constraints of the lower and upper bounds on each of the edges. This can be expressed as the following linear program:

Given:

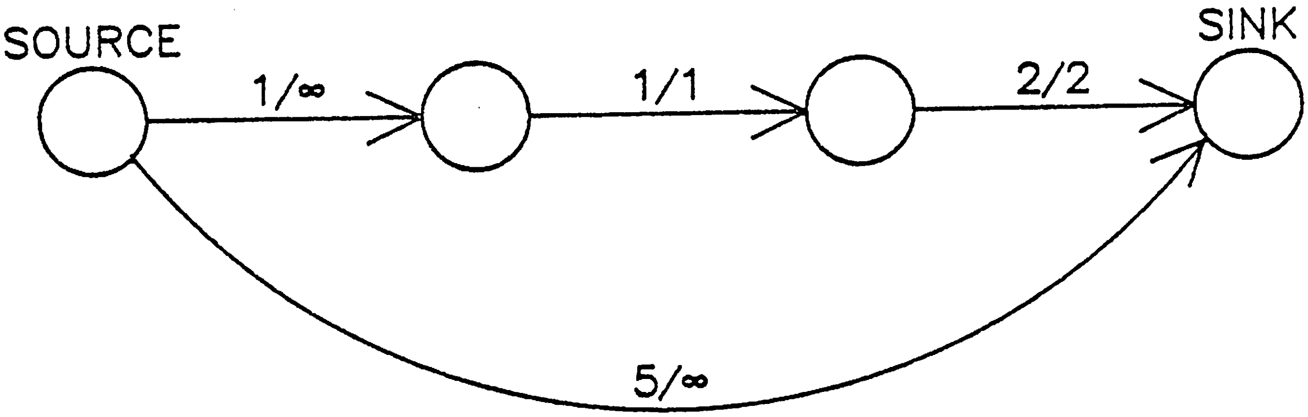


Fig. D.1 A Typical Python Constraint Graph

$U_j$ - linear position of the $j$th vertex
$e_{kj}$ - length of the edge joining vertices $k$ (source) and $j$ (sink)
$a_{kj}$ - lower bound of the $kj$th edge
$b_{kj}$ - upper bound of the $kj$th edge
$\left\{E\right\}$ - set of edges of the graph

Minimize the total length of the edges in the graph:

$$\sum e_{kj} \quad \forall e_{kj} \in \left\{E\right\} \qquad \text{D.1}$$

subject to the length constraints $a_{kj} \le e_{kj} \le b_{kj}$, expressed in linear program form as:

$$e_{kj} \ge a_{kj} \qquad \text{D.2}$$

$$-e_{kj} \ge -b_{kj} \ \dagger \qquad \text{D.3}$$

Physically, this corresponds to the addition of the extra edges in Fig. D.2. The shortest path problem is very similar to the linear program described above. The solution [Lawler76] is an iterative one, guaranteed to converge in $O(v)$ steps, with a maximum $O(v^2)$ calculations at each iteration (for a

† Some of the $b_{kj}$ may actually be ∞ (i.e., no upper bound) which implies that condition D.3 above has no meaning.

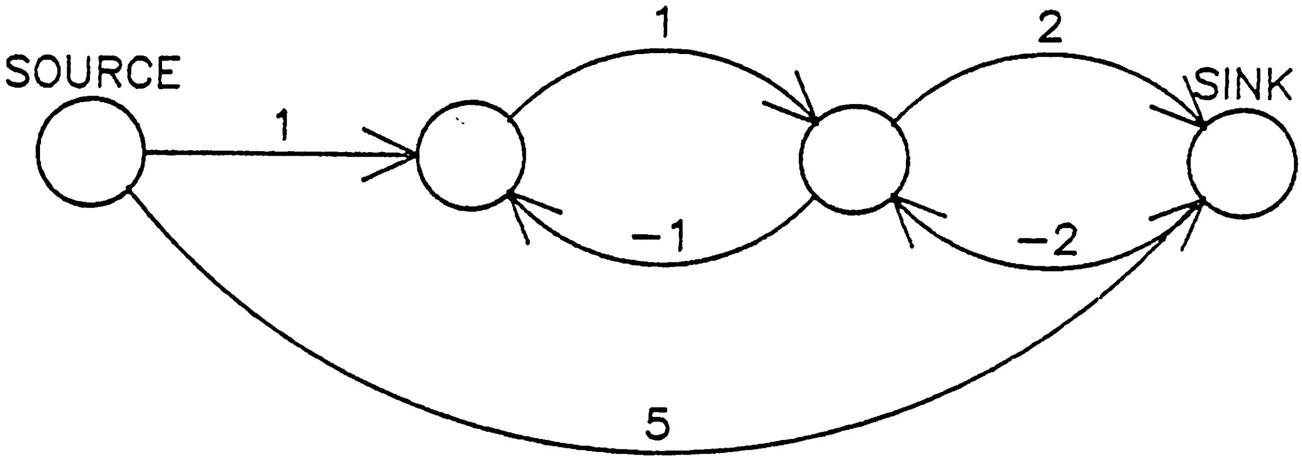


Fig. D.2 Physical Correspondence of Lower and Upper Bounds

complete graph) and thus $O(v^3)$ time complexity. Given $U_j^m$ as the position of the $j$th vertex at the $m$th iteration, the recursion formula is:

$$U_j^{(m+1)} = min(\ U_j^m, \min_{k!=j}(\ U_k^m + a_{kj}\ )\ ) \qquad \text{D.4}$$

The shortest path problem can be turned into a longest path problem by inverting the signs of all of the edge weights and inverting the signs of the resulting positions. This transformation can be reflected in the recursion formula above as:

$$U_j^{(m+1)} = max(\ U_j^m, \max_{k!=j}(\ U_k^m + a_{kj}\ )\ ) \qquad \text{D.5}$$

Applying this formula to the graph of Fig. D.2 yields the correct final spacings. In physical terms, during each iteration, each vertex is assigned the maximum of its current position and the current position of each of its predecessors plus the weight of the edge connecting the two vertices. Thus, when convergence is achieved, the position of each vertex is equal to the longest path from the source to that vertex.

Each edge with a finite upper bound in Fig. D.1 corresponds to a reverse edge with negative branch weight in Fig. D.2. Therefore, finding the maximum of the values of the predecessors plus the edge weight in Fig. D.2 can be related back to Fig. D.1 by taking the maximum of all the values of each of the predecessors plus the lower bound and the values of the successors minus the upper bound. Changing the algorithm in this way to work with graphs of the form D.1 yields the following formula:

$$U_j^{(m+1)} = max\left( U_j^m, \max_{k|=j}( U_k^m + a_{kj} ), \max_{l|=j}( U_l^m - b_{jl} ) \right) \qquad \text{D.6}$$

Equation D.6 is the basis for the critical path method used in Python.

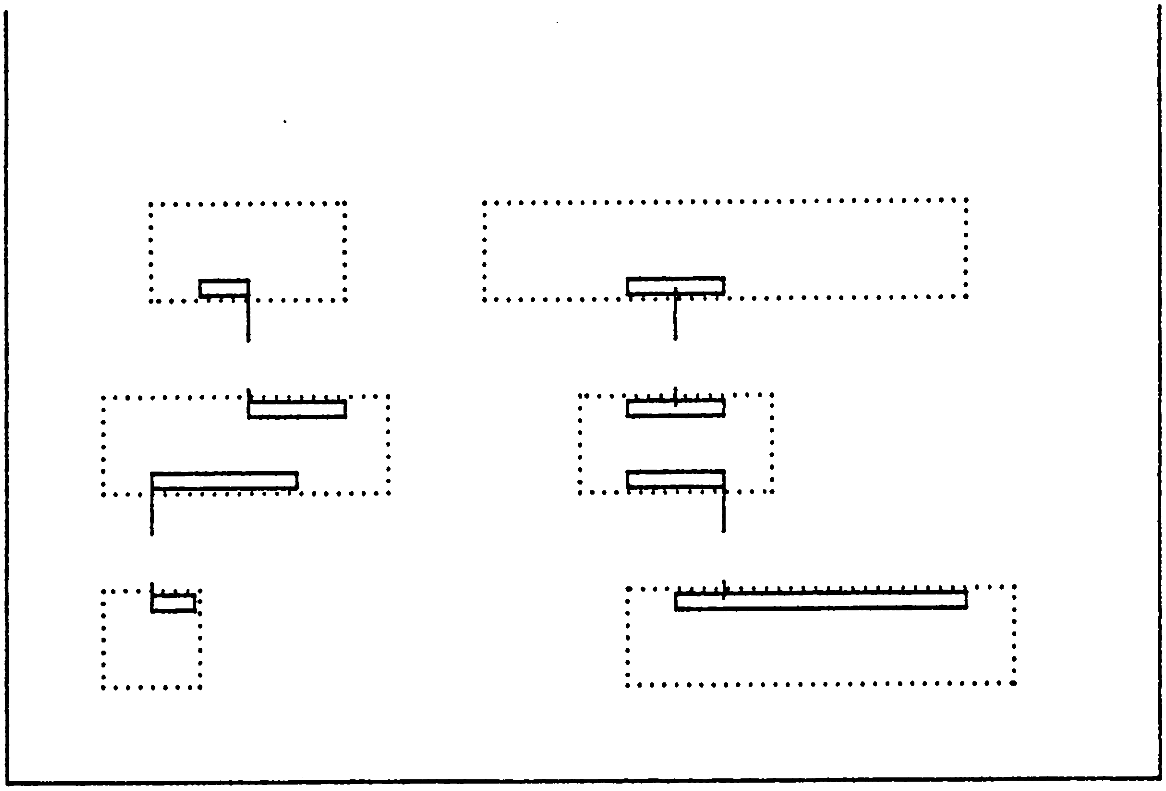


Fig. D.3 Sample Symbolic Layout

Since this algorithm is derived directly from the shortest path problem, it too is guaranteed to converge in at most $n$ steps. This provides a useful indication of negative cycles, which will cause the number of iterations to extend beyond the maximum allowed.

Fig. D.3 shows a simple symbolic layout with protection frames, terminal frames, and interconnect all of a single layer. Assume a spacing rule of 1 between objects, consider interconnect lines to be of zero width, and choose the reference point for each object as the lower left corner (to assure a digraph with no negative cycles). Examining the geometry for a horizontal compaction, we obtain the graph shown in Fig. D.4. Using equation D.6 iteratively on this graph, we find the vertex positions converge to their proper minimum allowed spacings in eight iterations, much lower than the upper bound of 19 guaranteed by the shortest path algorithm. The values at each iteration are illustrated in Table D.1, where each entry is a triple of the form:

$$U_j^m,\ \max_{k \neq j}(\ U_k + a_{kj}\ ),\ \max_{l \neq j}(\ U_l - b_{jl}\ )$$

Boldface indicates which member of the triple is maximum at each iteration, and when the remaining values change to italics, this indicates the maximum has reached its convergence value.

Note the number of vertices whose values change at each timepoint. This number is never greater than 50% of the number of vertices. It suggests that a different approach, processing only the vertices which would change, would greatly improve the order dependencies of the algorithm. Two modifications were made along these lines.

The first modification was to make the algorithm 'event-driven' at each iteration. Note that with positive upper bounds on the lengths of the graph

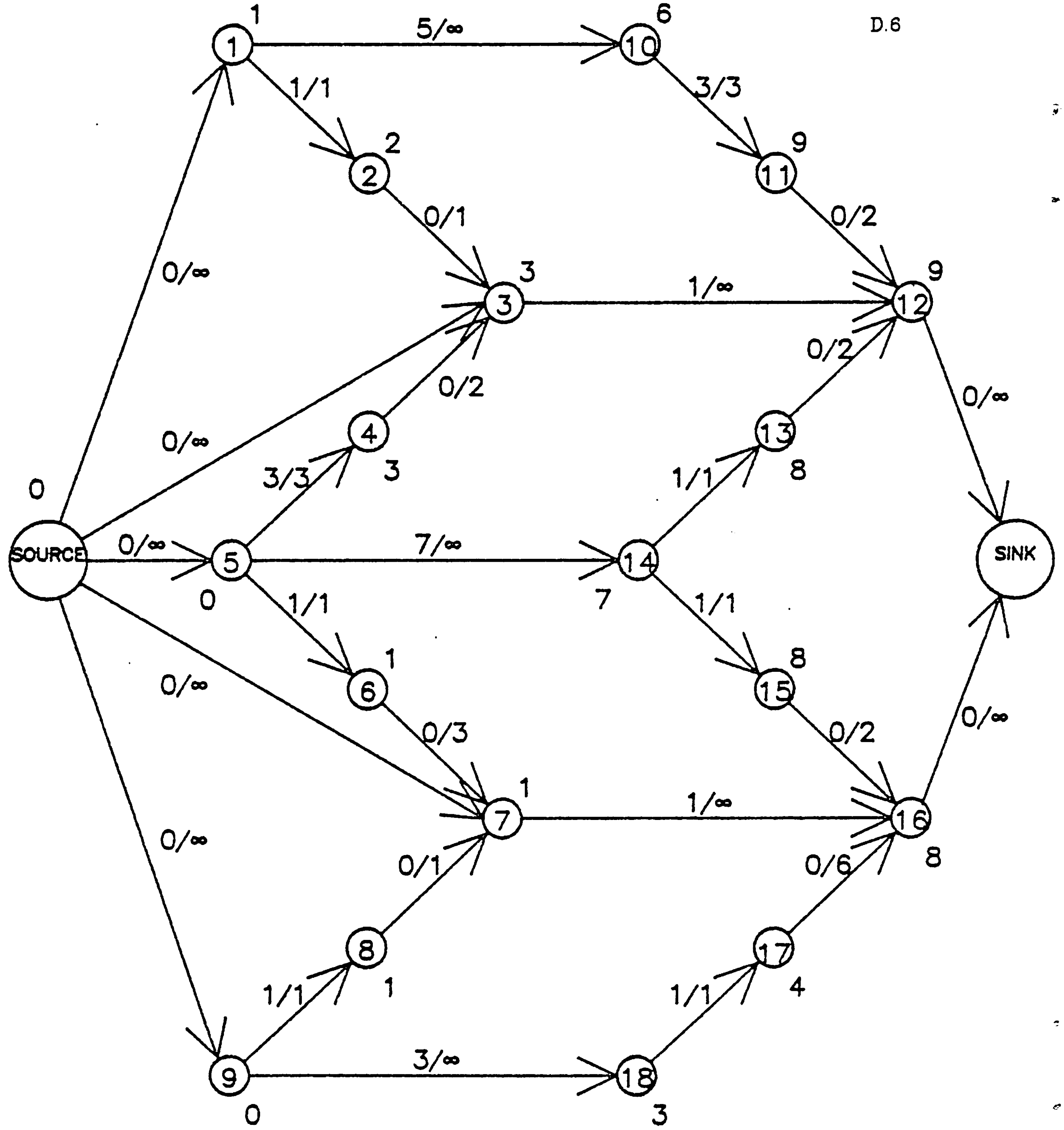


Fig. D.4 Horizontal Graph from Geometry in Fig. D.3

| VERTEX | ITERATION | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0,0,-1 | .0,0,0 | 0,0,0 | 0,0,1 | 1,0,1 | 1,0,1 | 1,0,1 | 1,0,1 |
| 2 | 0 | 0,1,-1 | 1,1,-1 | 1,1,2 | 2,1,2 | 2,2,2 | 2,2,2 | 2,2,2 | 2,2,2 |
| 3 | 0 | 0,0,-∞ | 0,3,-∞ | 3,3,-∞ | 3,3,-∞ | 3,3,-∞ | 3,3,-∞ | 3,3,-∞ | 3,3,-∞ |
| 4 | 0 | 0,3,-2 | 3,3,-2 | 3,3,1 | 3,3,1 | 3,3,1 | 3,3,1 | 3,3,1 | 3,3,1 |
| 5 | 0 | 0,0,-1 | 0,0,0 | 0,0,0 | 0,0,0 | 0,0,0 | 0,0,0 | 0,0,0 | 0,0,0 |
| 6 | 0 | 0,1,-3 | 1,1,-3 | 1,1,-2 | 1,1,-2 | 1,1,-2 | 1,1,-2 | 1,1,-2 | 1,1,-2 |
| 7 | 0 | 0,0,-∞ | 0,1,-∞ | 1,1,-∞ | 1,1,-∞ | 1,1,-∞ | 1,1,-∞ | 1,1,-∞ | 1,1,-∞ |
| 8 | 0 | 0,1,-1 | 1,1,-1 | 1,1,0 | 1,1,0 | 1,1,0 | 1,1,0 | 1,1,0 | 1,1,0 |
| 9 | 0 | 0,0,-1 | 0,0,0 | 0,0,0 | 0,0,0 | 0,0,0 | 0,0,0 | 0,0,0 | 0,0,0 |
| 10 | 0 | 0,5,-3 | 5,5,0 | 5,5,5 | 5,5,5 | 5,6,5 | 6,6,5 | 6,6,6 | 6,6,6 |
| 11 | 0 | 0,3,-2 | 3,8,-1 | 8,8,1 | 8,8,6 | 8,8,6 | 8,9,6 | 9,9,6 | 9,9,7 |
| 12 | 0 | 0,1,† | 1,3,† | 3,8,† | 8,8,† | 8,8,† | 8,8,† | 8,9,† | 9,9,† |
| 13 | 0 | 0,1,-2 | 1,8,-1 | 8,8,1 | 8,8,6 | 8,8,6 | 8,8,6 | 8,8,6 | 8,8,7 |
| 14 | 0 | 0,7,-1 | 7,7,0 | 7,7,7 | 7,7,7 | 7,7,7 | 7,7,7 | 7,7,7 | 7,7,7 |
| 15 | 0 | 0,1,-2 | 1,8,-1 | 8,8,1 | 8,8,6 | 8,8,6 | 8,8,6 | 8,8,6 | 8,8,6 |
| 16 | 0 | 0,1,† | 1,1,† | 1,8,† | 8,8,† | 8,8,† | 8,8,† | 8,8,† | 8,8,† |
| 17 | 0 | 0,1,-6 | 1,4,-5 | 4,4,-5 | 4,4,2 | 4,4,2 | 4,4,2 | 4,4,2 | 4,4,2 |
| 18 | 0 | 0,3,-1 | 3,3,0 | 3,3,3 | 3,3,3 | 3,3,3 | 3,3,3 | 3,3,3 | 3,3,3 |
| CHANGED | | 8 | 5 | 2 | 1 | 1 | 1 | 1 | 0 |

Table D.1 Iterations of Eqn. D.6 Applied to the Graph of Fig. D.4

edges, the first iteration of equation D.6 is equivalent to the unmodified CPM, with lower bounds only. At each iteration, a vertex can change position only if one of its predecessors or successors has changed position at the previous iteration. Thus, an event queue is maintained, and when the position of a vertex changes, all of its predecessors and successors are scheduled to be examined.

The second modification was to allow the changed positions of vertices to be used during the current iteration. The logic of the 'event-driven' algorithm is shown in Section 3.2.5. This modified CPM algorithm performs less than one quarter the number of operations as equation D.6.

† These vertices have no successors.

# APPENDIX E

## Glossary of Terms

This appendix contains definitions of the terms used in this report. Also summarized here are the descriptions of several other layout spacing aids which are compared and contrasted with *CABBAGE* in Chapter 1. In addition, many of the auxiliary programs and utilities that are used by or use *Python* are described. Terms that appear in boldface begin definitions of that particular term. Words that appear in italics during the definitions are themselves defined.

**Actual Terminals**

As contrasted with *formal terminals*, *actual terminals* are represented by physical geometries. This geometry is part of the *instance* of a master cell, or can also be a local *terminal* in the definition of the cell. It binds the formal *terminal*, which contains the *net-id* and name of the *terminal*, to the geometry which implements a particular *instance* of a *terminal*.

**Bounding Box**

This is defined as the box surrounding a cell such that no geometry local to the cell or contained within any *instance* in the cell extends beyond the box boundaries. In common usage the term *bounding box* refers to the *minimum* size box that encapsulates the cell. *Bounding boxes* are used to do quick checks to determine that an object is definitely not of interest. One example of this usage is checking to see whether or not a particular cell intersects with an area of interest that is being displayed on a graphics device. Another usage is found in Python, where the upper and lower coordinates of *intervals* are used to quickly determine if there is a possible overlap between the *intervals*.

**Bounding Polygon**

An extension of the *bounding box* idea, this is a polygon which completely encapsulates a cell, including all instances and geometries within the cell. Using a *bounding polygon* rather than a *bounding box* allows a closer representation of the limits of the geometries internal to a cell.

**CABBAGE**The predecessor of the Python program, CABBAGE is a layout compacter for symbolic integrated circuit layouts, consisting of a suite of two separate programs which communicate through a symbolic intermediate file. The program PRSLI takes a layout entered through the graphics editor GRLIC and compacts the primitives according to a set of *spacing rules*, which define the minimum allowed spacing between objects on the different mask layers in the semiconductor process. CABBAGE was one of the first spacing programs to use the idea of *coverage* to limit the order complexity of the spacing rule analysis. Doing so drops the observed order from the worst case value of $O(n^2)$ to $O(n^{1.2})$. The ideas contained in CABBAGE were extended and built upon to form the Python system.

**Cabtosquid**

This is a computer program that interprets the symbolic intermediate file used by CABBAGE and enters the data into the *Squid* database, obeying the conventions required by Python (such as explicitly labeling connectivity, etc.).

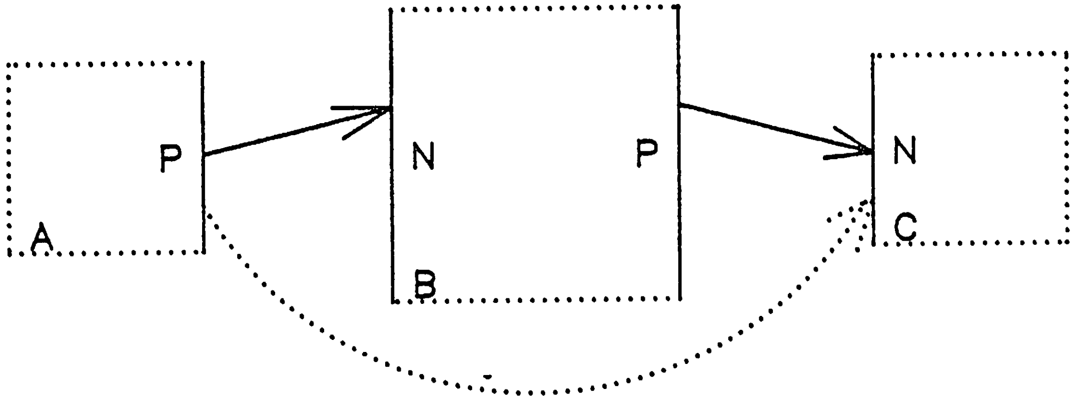
**Constraint**

A *constraint* is the spacing requirement between two objects. It is represented in Python by an *edge* in a *graph* which represents the topology and spacing requirements of the IC layout. *Constraints* may have minimum and maximum values in Python, and can be used to express spacing requirements between objects imposed by the *spacing rules*, or to express fixed spacing requirements between objects for preservation of electrical connectivity. Also, *constraints* may be used to preserve the shape of objects, *e.g.*, to keep the width of interconnect *lines* constant.

**Constraint Graph**

This is the *graph* containing the *constraints*. It represents the topology of the IC. While the *edges* in the *constraint graph* are interpreted as lengths indicating minimum and maximum spacings between objects, the vertices in the *constraint graph* are interpreted as the locations of the reference points for the geometries in the IC layout. Solution of the longest path problem from the source of the *graph* to the sink of the *graph* determines the locations of the primitives in the IC that require a minimal area for the entire layout.

**Cover** When checking *intervals* for overlap, if a *primary* and a *neighbor* overlap, the *neighbor* is said to *cover* or shield the *primary*. It is no longer necessary to check for design rule constraints between the *primary* and any other *neighbor* over the overlapping part of the *interval*. Thus, this limits the search required in the spacing rule analysis from a worst case $O(n^2)$ time complexity to an observed complexity of only $O(n^{1.1})$. As the size of cells becomes larger, this savings in computation can be extremely large. The idea of *coverage* assumes that given two objects that have a spacing rule between them, and object to the right of the second will be covered by the second over its *interval* and may not require a *constraint* (See Fig. E.1). Put in another way, if there is a rule between objects A and B, there is no need for a rule between objects A and C, since the rule between objects B and C will keep objects A and C from coming too close. Note that this premise falls short when there exists three mask layers and one mask layer has a spacing rule to both of the other two but the other two have no spacing rule between them. In this case, a spacing *constraint* is ignored that may cause a layout rule violation. The concept of *coverage* also must be modified if objects are allowed to merge. When *merging* two objects, the *intervals* in the two objects should not be used as



Constraint Not Generated Due to B Covering A

Fig. E.1 Coverage of a Primary *Interval*

*covers.*

Critical Path

A path from the source of the *constraint graph* to the sink where the difference between the location of every pair of vertices on the path is equal to the lower bound of the *constraint edge* joining the two vertices is a *critical path.* There may be more than one equivalent *critical path* in a given topology. The *critical path* fixes the minimal area required for the IC layout. If any object is moved in a manner such that the distance from source to sink in the *constraint graph* is lessened, a layout rule *must* be violated.

Decoupled Spacing Analysis

In order to take advantage of the rectilinear nature of the geometries that are spaced by Python, the spacing analysis is *decoupled* into separate passes for the X and Y dimensions of the IC layout. *Layout rules* are ignored in the direction perpendicular to the direction of spacing. Hence, *layout rules* can be violated in the direction perpendicular to the spacing direction. This fact requires that the spacing process be iterated between the X and Y directions until neither spacing causes the cell bounds to change and all constraints are satisfied. Only then is a layout-rule-correct layout guaranteed.

**Edge** In Python, an *edge* is the *constraint edge* added to the *graph* which represents the IC topology. Vertices in the *graph* represent the locations of reference points of the geometries within a cell. Each *edge* has a lower and an upper bound, which are the minimum and maximum allowed distances between the two vertices the *edge* joins.

**Edge Tracing**

A term used in CABBAGE, *edge tracing* is used to develop all of the *edges* (really called *intervals*) that occur on a particular side of a group of interconnected elements. CABBAGE grouped interconnected elements together to save memory space. Since these groups contained interconnect *lines*, which change shape, it was necessary to dynamically develop these *edges* during each spacing rule analysis. Since each group is represented individually in Python and since the *interval edges* are expressed relative to the *bounding box* coordinates of the *instance* that owns them, it is only necessary to do the *edge tracing* in Python once, before the first analysis.

**Fang** This is a rectilinear polygon package written by Peter Moore. It allows arbitrary logical operations on rectilinear mask data, as well as providing a grow/shrink capability. This package is used in Python for developing the *edges* of the *protection frames* from the box representation of polygons stored in the Squid database. It is used by the frame generation program to do the grow/merge/shrink operation used for removing the holes in the geometry and producing the frames.

**Fixed Constraint**

An *edge* in the *constraint graph* that has the same lower and upper bound on its length, and hence fixes the distance between two vertices. This is the major cause of overconstraining conditions. When a *fixed constraint* exists between two vertices and the upper bound is less than the sum of the lower bounds on another path between the two vertices, this creates an overconstraining condition.

**Fixed Grid**

In both the *SLIC* and *MULGA* systems, an equally spaced grid is used to represent the allowed positions. In SLIC, the grid spacing is determined by the largest of the *spacing rules*. The designer places geometry in the grid cells, and this grid is then translated into the actual IC layout. Since the *fixed grid* spacing is determined by the worst case spacing rule, the resulting layout is not as compact as the *spacing rules* allow. In MULGA, the *fixed grid* is later mapped into real values, and the spacings between grid points are not necessarily uniform. This overcomes the disadvantage of the simpler *fixed grid* systems. *Fixed grid* methods are contrasted with the *relative grid* methods used in *FLOSS*, CABBAGE, *STICKS*, and Python.

**FLOSS** One of the earliest spacing programs to use a *relative grid* and a *graph* representation of the IC topology, FLOSS is from RCA. The *F*inished *L*ay*O*ut *S*tarting from *S*ketch program makes use of hierarchy, and is designed for the COS/MOS technology. It relies on the initial relative placement of the objects in the same manner as CABBAGE and Python. FLOSS is written in PL/I and consists of ~50 procedures. The major difference between the FLOSS and CABBAGE programs is the input format. FLOSS digitizes a hand-drawn sketch and takes information about the cells from either a standard library or a previously defined cell. The results show a layout ~19% larger than the equivalent hand-drawn layout.

**Formal Terminals**

As contrasted with *actual terminals*, *formal terminals* are the definitions of a *terminal* associated with the master cell of an *instance*. An example would be the drain, gate, and source *terminals* of an NMOS transistor. While they are specified in the definition of the transistor (the master cell) as *formal terminals*, they do not have a net or physical location associated with them until the master cell is instanced. At this point, *actual terminals* in each *instance* are created for each of the *formal terminals* in the master cell.

**Graph** According to [Bondy76], a *graph* defined as:

> ... an ordered triple ( $V(G)$, $E(G)$, $\varphi_G$ ) consisting of a nonempty set $V(G)$ of *vertices*, a set $E(G)$, disjoint from $V(G)$, of *edges*, and an *incidence function* $\varphi_G$ that associates with each *edge* of G an unordered pair of (not necessarily distinct) vertices of G.

In Python, a *graph* is used to represent the physical topology of the IC mask data. The locations of the vertices represent the locations of the reference points of the primitives, and the lengths of the *edges* represent the minimum and maximum allowable spacing between the two vertices each *edge* joins.

**Graphcap**

A terminal independent graphics package developed initially by the author and Ken Keller, with ongoing development by Giles Billingsly and Ken Keller. This package is used by *Hawk* to provide a graphics terminal independent editor, which, in addition to its many other capabilities, is the input editor for Python. The *graphcap* routines themselves are a low level package for dealing with an ASCII database representation of a defined standard set of capabilities of a graphics terminal. The *Model Frame Buffer*, or *MFB*, is the set of terminal independent graphics routines that interface a high-level applications program, such as Hawk, to the low level *graphcap* routines.

**Hawk** Hawk is the graphics editor which is the input editor for Python, and it stores the IC designs in the Squid database. Hawk is very powerful, and has the intelligence necessary to allow symbolic layout. It also provides a clean user interface to the Python program itself. See [Keller82] for a description of Hawk.

**Hierarchical Design**

If a designer encapsulates functional blocks of his design and uses these blocks in other blocks, it is called hierarchical design. There are two forms of *hierarchical design*: top down and bottom up. In the top down design the system level considerations are given first priority and the design is narrowed along functional *lines* until a particular implementation is achieved. In the bottom up approach, the low level cells are designed first and used to build up more and more complex cells until finally the entire system is designed. Python is designed for use in a *hierarchical design* style. This type of design style has several advantages for a spacing program. If only the master cell of each different subcell is spaced instead of each *instance* of each master cell, great computational savings can be realized in the spacing of a regular structure. In addition, the greater-than-linear order dependencies of the algorithms in the Python imply that merely dividing the spacing process up into $m$ spacings each of $\frac{n}{m}$ elements will improve the computational performance. Another very important point is that very large circuits will not fit into virtual memory on the computer. This requires reducing the size of larger circuits through the use of hierarchy.

**Instance** When a cell is designed and laid out, this cell is called a *master cell.* When the master cell is used in another cell, it is called an *instance* of the master cell, and has all of the characteristics of the master cell, in addition to its own unique name, translation, and optional mirrorings and rotations about the coordinate axes. Instances are physically placed and then connected with *lines*. They do not change form during the spacing process; they only change translation. *Lines* grow or shrink to space the instances according to the *layout rules*.

**Interval** In Python, an *interval* is an *edge* of a *protection frame*. These *interval edges* are used in the spacing rule analysis to determine the minimum spacing required between any pair of objects. All primaries (right-side *intervals*) are compared against all *neighbors* (left-side *intervals*) to check for the spacing requirement. *Coverage* is used to limit the depth of the search. Since the *intervals* are compared in sorted order, *lines* may be routed through cells, and concave and convex objects may 'fit' together, if doing so does not violate any *spacing rules* (Fig. E.2).

**Jog** A *jog* is a *line* of one orientation that connects two *lines* in the perpendicular orientation (See Fig. E.C). *Jogs* introduce great latitude in the spacing analysis for Python. Two objects joined by a straight *line* are fixed relative to one another in the dimension perpendicular to the length of the *line*. Two objects joined by a jogged *line* are

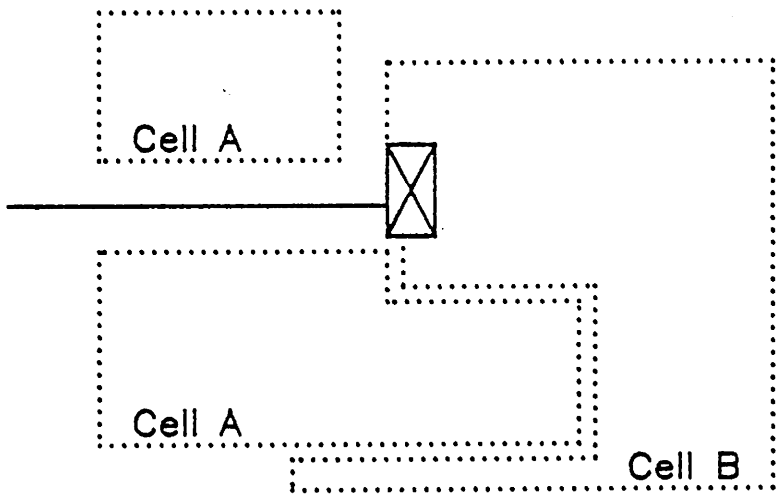


Fig. E.2 Routing through Cells and Concave/Convex Interaction

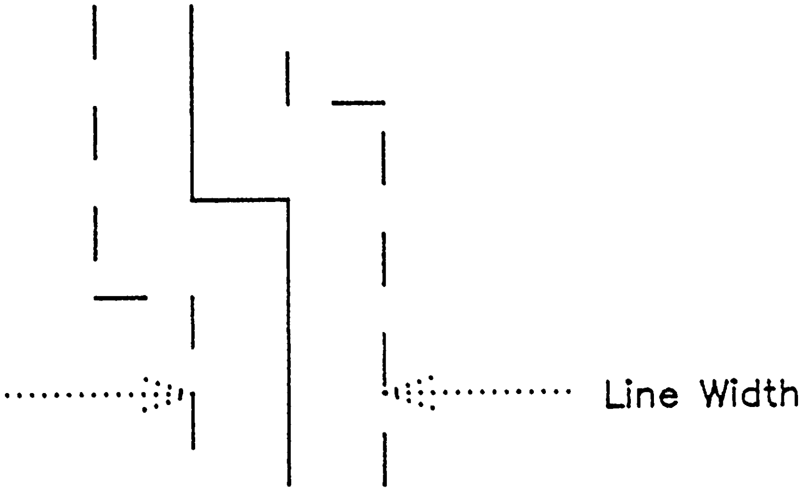


Fig. E.3 A Jog

not fixed at all relative to one another.

### Jog Generation

A property of the CABBAGE program was the ability to determine which *lines* on the *critical path* would benefit from insertion of *jogs*. The idea was that splitting *critical path lines* would allow a more compact final result. Determining which interconnection *lines* are on the *critical path* is not very difficult. Determining where on the *line* to place the *jog* requires examination of the surrounding geometry. CABBAGE placed the *jog* in the middle. In the worst case, this could mean $O(n \log n)$ *jog* insertions to find the proper location. In Fig. E.4,

**Layout Rules**

These include the minimum allowable spacings between objects on the same and different mask layers in the semiconductor IC process. They take their values from considerations of the possible misalignment between mask layers during processing of the IC. They are designed to preserve the electrical properties of the IC even under worst-case conditions. Not all mask layers have rules for every other mask layer. Here is an example table of the Mead/Conway NMOS *spacing rules* which are used by Python:

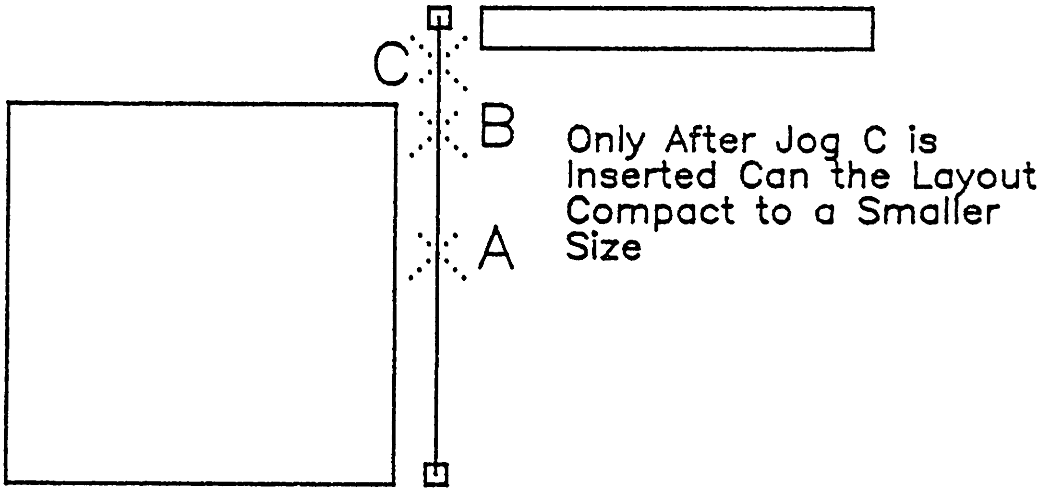

C
B
A
Only After Jog C is Inserted Can the Layout Compact to a Smaller Size

Fig. E.4 Binary Search Jog Insertion

```
spacerule ND ND 3
spacerule ND NP 1
spacerule ND NC 3
spacerule ND NB 2
spacerule NP NP 2
spacerule NP NC 3
spacerule NP NB 2
spacerule NM NM 3
spacerule NC NC 2
spacerule NC NB 1
spacerule NB NB 2
spacerule NT NC 2
```

ND is diffusion, NP is polysilicon, NC is contact cut, NM is metal for interconnections, NB is the buried contact window cut, and NT is a special mask for the active area of transistors (necessary for implementation of the full Mead/Conway layout rule set). *Layout rules* also include additional rules, such as:

Minimum Area Rules
Minimum Width Rules
Minimum Enclosure Rules

These rules are not directly checked by as they are easier to check in the input phase.

**Leaf Cell** A *leaf cell* is a cell that simply contains no instances. It can be looked at as the leaf of a tree; the smallest unit which cannot 'branch' into anything smaller. It is the atom of *hierarchical design*.

**Line** In Python, a *line* is a path on one of the IC mask layers that is used for interconnecting *terminals* on instances. *Lines* grow and shrink along their length, and this allows the spacing program to change the size of the layout to satisfy the design rules.

**Local Interconnect**

This is interconnect local to an *instance*. In the hierarchy, if the level of the instances within a cell is 0, the level of the *local interconnect* is also 0. *Local interconnect*, *instance protection frames*, and local and *instance terminals* are the only geometries used to do the spacing in Python.

**Local Terminals**

*Local terminals* are at the same level in the hierarchy as the local interconnect described above. They are used to communicate the signals between the cell and the next level in the hierarchy where the *instance* in which the *local terminals* are used is placed.

**Longest Path**

The *longest path* through the *graph* is defined as the *longest path* from the source of the *constraint graph* to the sink of the *constraint graph* while satisfying the minimum constraints on all of the *edges* joining vertices on the path. Solution of this *longest path* determines the locations of the primitives that requires the smallest area while satisfying all *spacing rules*. The algorithm to solve this problem is an iterative one, guaranteed to converge in $v$ iterations, if there are no overconstraining conditions ($v$ is the number of vertices in the *graph*). See Appendix D for a discussion of the algorithm.

**Merging** When two similar objects are interconnected, often they may 'merge', or overlap, in a certain fashion. An example of this is two NMOS transistors connected by their gates. The polysilicon extension past the end of the transistor is 2λ. When the gates of the two transistors are connected, if no *merging* is allowed, the spacing between the source and drain regions of the two transistors is 4λ.

The required spacing is only 3λ, and 1λ of space is being wasted. If the primitives are allowed to merge, the diffusion to diffusion spacing of 3λ can be met. CABBAGE allows *merging*, since the rules for *merging* the specific NMOS primitives known to the program are well understood. Python is a more general program, and does not allow *merging*, since there is not yet a general convention for how and when arbitrary objects can merge.

**MFB** See the definition of *Graphcap*.

**Model Frame Buffer**

See the definition of *Graphcap*.

**MULGA** MULGA is one of the most recent IC spacing programs. It uses the concept of a *virtual grid* to obtain an efficient compaction result without sacrifice of computation time. Integer grids are used to represent the IC layout in a matrix form. The spacing analysis maps the integer grid spacing into a real value which satisfies the spacing rules while minimizing total area required for the layout. Corner constraints are checked, and a form of recursion is used to check constraints that extend over multiple grid rows or columns. *Jog generation* is intentionally avoided. The author feels that such drastic topological changes as can be introduced with *jog generation* are better introduced by the human designer. MULGA has been used to design several NMOS and CMOS ICs, which have been fabricated.

**Neighbor**A *neighbor* is a left-hand-side *interval*, and all *neighbors* to the right of a *primary* are compared against it for possible overlap, which would require a *constraint* between the objects which the *primary* and *neighbor* belong to.

**Overconstraint**

An *overconstraint* in the *constraint graph* is generated when two paths exist between a pair of vertices and the sum of the lower bounds of the *edges* in the first path is greater than the sum of the upper bounds of the *edges* in the second path. In this case, there can be no location for the vertices on the two paths that will satisfy all of the constraints. The *graph* is said to be overconstrained. The nature of the *longest path* algorithm that solves for the locations of all of the vertices is such that $v$ iterations are required to detect the *existence* of an *overconstraint*, where $v$ is the number of vertices in the *graph*. Detection of the *location* of an *overconstraint* is an $O(2^v)$ time complexity problem. For even very small graphs, this time complexity becomes prohibitively large. For this reason, no detection of the *location* of overconstraining conditions is done in Python.

**Primary** A *primary* is a right-side-edge *interval*, and is compared in the left-to-right scan of *neighbors* for possible overlap.

**Protection Frame**

A *protection frame*, in the implementation of Python, is used to surround the geometry within a cell, to reduce the amount of data it is necessary to examine in order to space the cell when it is used at a greater level in the hierarchy.

**Python** The program developed for this report, Python is a layout rule spacing program for symbolically designed IC layouts, and is based on the ideas contained within the CABBAGE program. Python extends the *critical path* algorithm used to solve for the minimum area layout to include *upper* as well as lower constraints on the *edges* in the *graph*. With these upper bounds, each object can be represented individually, rather than as groups of connected objects, as was done in the CABBAGE program. Fixed bound *edges* preserve electrical connectivity. *Edges* with different lower and upper bounds but with a finite upper bound can be used for *sliding contacts*, or *terminal frames*.

Rectangular Polygon

A polygon where the angle between the *edge* segments at each of its vertices is some integral multiple of 90 degrees. These polygons are used in Python as *protection frames* for cells. This allows almost any arbitrary shaped object to appear within a *leaf cell*. As long as the *protection frame* is a rectangular polygon, Python can space instances of the cell.

Relative Constraint

A *relative constraint* relates two (or more) objects to one another, while making no restrictions on the movement of the group as a whole.

Relative Grid

Contrasted to the *fixed grid* approach, the *relative grid* serves only to indicate the initial relative placement and interconnection of the objects in the layout. The spacing between objects is determined through analysis of the spacing requirements between objects. This approach is used by the FLOSS, CABBAGE, STICKS, and Python programs.

**SLIC** The SLIC (*S*ymbolic *L*ayout of *I*ntegrated *C*ircuits) system is a *fixed grid* symbolic layout aid from AMI which provides an interactive environment for detection and correction of errors in the symbolic layout. No automatic spacing is done. The types of checking that are available to the designer are:

- Design Rule Checking (Boolean Equation Input)
- Network Comparison (Between Logic Deck and Layout)
- Netlist Trace (Includes Device Sizes)
- Parasitic Extraction (Capacitances and Resistances)

Once the layout is error free, masks are generated and the circuit can be fabricated. Examples are presented which show circuit sizes within 10% of hand-drawn circuit size.

**Sliding Contacts**

See *Terminal Frames*.

**SLIP/SLIM**

The *SLIM* program [Dunlop80], and previously the *SLIP* program [Dunlop78], are from Bell Laboratories. A novel use of multiple spacing methods makes SLIM an interesting example. The IC symbolic data is partitioned automatically, into an optimal size calculated by the author[Dunlop79]. A loose initial placement which is design rule correct (although not optimal) is generated. *Critical path* detection, similar to the method used in CABBAGE, is coupled with a local compaction method to reduce total run-time required, while maintaining an efficient compaction result. The local compaction procedure clusters together groups of objects on the *critical path*. *Jogs* are inserted in the layout, but only at contact locations. Global rift *line* compaction [Akers70] removes the excess space between the locally compacted partitions. The order dependency for the composite algorithm is approximately $O(n^{3/2})$.

**Spacing Rules**

See *layout rules* (which are a superset of *spacing rules*).

**Squid** This is the database used by Python to store and retrieve the symbolic IC information. It stores logic information, such as electrical connectivity, as well as geometric information, relating to the physical layout of the circuit.

**STICKS** This program is a high-level layout aid from Hewlett-Packard Co. [Williams78]. As in CABBAGE and FLOSS, STICKS uses a *relative grid* approach, with the designer's initial placement of the circuit elements determining to a great degree the possible efficiency of the compaction. STICKS takes a stick diagram as input, with the elements represented by symbols, and the interconnection *lines* represented by zero-width sticks. The spacing process is separated into X and Y compactions. The spacing algorithm works by taking groups of elements and placing them as far to one side as is possible, given the fixed topology of the previously placed elements and the spacing requirements between the elements. Interactive optimization is possible. The designer may perform a *local compaction* by identifying an area to be compacted and a direction to compact the elements. This helps alleviate the problems that can occur when compaction in one direction prevents efficient compaction in the perpendicular direction. A second capability with the interactive program is the user introduction of *jog* points. Cells may be encapsulated, so there is a form of hierarchy in STICKS.

TerminalA *terminal* is used to connect between levels in the hierarchy. When a cell is designed, all of the signals that are required at the next level (VDD, GND, inputs, and outputs) are terminated on *local terminals* in the cell. When this newly created master cell is placed, each *instance* has only the *protection frames* and *terminals* (which are also frames) visible for connection. All of the interior components of the cell are no longer visible. This partitioning of the design and implementation assists both the human designer and the computer tools that work on the layout. The designer sees less information at any one time and is less likely to be confused or overwhelmed. He can match the logic partitioning to the logical partitioning present in his mind at the time of the design. At the same time, the computer tools that work on his layout must work with less information at a time, and this greatly reduces the computation time required for the tasks of design rule checking or layout rule spacing.

Terminal Frame

The *terminal frame* is the normal geometric implementation of a *terminal*. Instead of point connections, as are made in CABBAGE, the *terminal frames* only require that the *lines* interconnecting them terminate somewhere within the *terminal frame*. With this 'slop', the spacing program may yield a better compaction result, since it may place each *terminal* to the best advantage.

Vertex In the Python *constraint graph*, the vertices represent the locations of the reference points of each *instance* and interconnection *line* in the layout. Specifically, the *bounding box edges* of the instances and interconnection *lines* are represented. The spacing analysis is separated in the X and Y directions, so there are two graphs. The vertices for the top and bottom bounding box *edges* are in the Y *graph*, and the vertices for the right and left bounding box *edges* are in the X *graph*. Each *vertex* is responsible for updating its related coordinate. Storing what is actually the transformational information for each *instance* in only one place insures the integrity of the data.

# APPENDIX F

## Python Listing

This appendix contains a listing of the *Python* program. The program is approximately 4000 lines of the *C* programming language. Approximately 1000 of the 4000 lines are comments. The size of the compiled program is 74752 bytes of instructions, 13312 bytes of data, and 10820 bytes of common storage on a VAX 11/780 32 bit minicomputer running the 4.1BSD version of the UNIX† operating system.

†UNIX is a Trademark of Bell Laboratories.

Appendix F can be obtained from Pamela Bostelman
c/o Industrial Liaison Program
499 Cory Hall
University of California
Berkeley, CA 94720
(415) 642-4370

# APPENDIX G

## Cabtosquid Program Listing

This appendix contains the manual entry and program listing for the *cabtosquid* program, a translator from the *CABBAGE* intermediate format into the *Squid* database. The input conventions of *Python* are observed.

**NAME**

cabtosquid – A translator from *CABBAGE I* format into the *squid* database

**SYNOPSIS**

**cabtosquid** [-e#] [-NO] cabbage1file ...

**DESCRIPTION**

*Cabtosquid* provides a means to translate cells generated with the *CABBAGE I* system into the *squid* database format. The conventions for interfacing to *python* are observed, so translated cells may be spaced with *python*. The set of spacing rules used is the Mead/Conway rules with lambda equal to two microns.

The program creates *squid* master cells for each different sized *CABBAGE I* primitive. Then, a master cell for the top level *CABBAGE I* cell is created, with the lower level master cells instanciated. *Squid* requires a separate directory for each master cell, so be in the directory where you want the output before invoking *cabtosquid*.

The *-e#* option takes a number and expands the *CABBAGE I* file before translation. This expansion multiplies the center locations of all point structures and the endpoints of all wires by the expansion factor. The width of all wires and size of all devices is preserved.

The *-O* and *-N* options allow translation of intermediate files generated by older and newer versions of the *CABBAGE I* program respectively. The default version is 1B. These options are additive - typing '-N -N -N -O -N' would set up for using version 1E files.

**FILES**

| | |
|---|---|
| c2s.out | – Root *squid* cell of *CABBAGE I* instance |
| [0-9]*by[0-9]*.* | – *Squid* master cells for *CABBAGE I* primitives |

**SEE ALSO**

cabbage(cad), hawk(cad), python(cad), squid(3cad), cadrc(3cad)

**AUTHOR**

Mark Bales

**DIAGNOSTICS**

Various self explanatory error diagnostics. Most report problems with queries to the *squid* database. A few report errors in the *CABBAGE I* input file, such as the wrong version, or overlapping transistors, etc.

**BUGS**

The change from point terminals in *CABBAGE I* to terminal *areas* in *python* creates a problem. It is possible for a line segment to have both endpoints contained within a single terminal frame. In *CABBAGE I*, this presented no difficulty, since only the endpoint terminating at the center of the terminal was considered connected to the terminal. With the terminal frames used in *python*, it is no longer possible to determine which of the two possible endpoints should be bound to the terminal. This is the reason for the *-e* option. Expanding the layout will eventually move one of the endpoints out of such a terminal, and the connectivity can be uniquely determined.

The RUNX line type is currently ignored. It should be passed through to allow circuits spaced with *python* to retain a rectilinear aspect ratio.

```
#include <stdio.h>
#include "drules.h"

/*
 * these are some global defines
 */
#define TRUE 1
#define FALSE 0
#define OK 1
/* Here are some orientation defines for the CABBAGE I file */
#define HORIZONTAL 0            /* Orientations for the CABBAGE I file */
#define VERTICAL 2              /* Orientations for the CABBAGE I file */
#define NORTH 0                 /* Orientations for the CABBAGE I file */
#define EAST 2                  /* Orientations for the CABBAGE I file */
#define SOUTH 4                 /* Orientations for the CABBAGE I file */
#define WEST 6                  /* Orientations for the CABBAGE I file */
/* Here are some type ids used with the `point' data structure */
#define ULINE 1                 /* Upper Line endpoint for point struct */
#define LLINE 10                /* Upper Line endpoint for point struct */
#define DRAIN 2                 /* Tran or load drain terminal for point */
#define GATE 3                  /* Tran or load gate terminal for point */
#define SOURCE 4                /* Tran or load source terminal for point */
#define POINT 5                 /* Point structure id for point struct */
/*
 * These are the types of elements in the cabbage file
 */
/* Variable length lines (may stretch or shrink during compaction) */
#define DIFF 1                  /* Diffusion interconnection line */
#define POLY 2                  /* Polysilicon interconnection line */
#define METAL 3                 /* Metal interconnection line */
#define RUNX 4                  /* Runx (bounding box) interconnection line */
/* Fixed length lines (will only change location during compaction) */
#define FDIFF 5                 /* Diffusion interconnection line */
#define FPOLY 6                 /* Polysilicon interconnection line */
#define FMETAL 7                /* Metal interconnection line */
#define FRUNX 8                 /* Runx (bounding box) interconnection line */
/* Transistor devices */
#define TRAN 9                  /* Enhancement transistor */
#define LOAD 10                 /* Depletion transistor (load) */
#define D_M_CNT 11              /* Diffusion-Metal contact */
#define P_M_CNT 12              /* Polysilicon-Metal contact */
#define BUR_CNT 13              /* Buried (Polysilicon-Diffusion) contact */
#define BUT_CNT 14              /* Butting (Polysilicon-Diffusion) contact */
#define TERM 15                 /* Line terminator (local terminal to Squid) */
/*  Here are a few extra types needed for the CABBAGE/Python conventions */
#define D_D_CNT 16              /* Diffusion-Diffusion contact */
#define P_P_CNT 17              /* Polysilicon-Polysilicon contact */
#define M_M_CNT 18              /* Metal-Metal contact */
/*
 * These are the layer definitions in cabbage
 */
#define DIFFLEVEL 0             /* Diffusion layer */
#define POLYLEVEL 1             /* Polysilicon layer */
#define METALLEVEL 2            /* Metal layer */
#define RESERVEDLEVEL 3         /* Runx (bounding box) layer */
#define ACTIVELEVEL 4           /* Active area (Diffusion & Polysilicon) */
#define CONTACTLEVEL 5          /* Contact window cut */
#define BURIEDLEVEL 6           /* Buried contact window cut */
#define IMPLANTLEVEL 7          /* Implant layer (Depletion loads only) */
#define EXTRALEVEL 8            /* Unused layer */
/*
 * This is the structure for the data from the cabbage intermediate file
 */
struct lsg {
        int type;                  /* Integer type of element (Values 1-15) */
        int orientation;           /* Integer orientation ( 0 or 2 ) */
        int x_center;              /* Integer element x center; -1 => Horiz line */
        int x_left_offset;         /* Integer offset to left of center; +ive # */
        int x_right_offset;        /* Integer offset to right of center; +ive # */
        int y_center;              /* Integer element y center; -1 => Vert line */
        int y_bottom_offset;       /* Integer offset below center; +ive # */
        int y_top_offset;          /* Integer offset above center; +ive # */
        char *name;                /* Pointer to name derived from [xy]_center */
        struct lsg *next;          /* Pointer to next like-type element in array */
        int     instanciated;      /* Flag indicating if element been instanced */
        int     instid;            /* Id of instance in Squid database */
};

/*
 * This is the structure for `points'. Each line has two, each transistor
 * has 3, and each contact has 1 only. These points are sorted and used
 * to determine electrical connectivity.
 */
struct point {
        int     x;                 /* X coordinate of `point' */
        int     y;                 /* Y coordinate of `point' */
        struct point *next;        /* Next `point' in linked list */
        struct point *nextinnet; /* Next `point' in the same electrical net */
        struct point *nextatpoint; /* Next `point' at the same X & Y */
        struct nethd *netid;       /* Pointer to nethd net id structure */
        int     type;              /* Type of `point' (See above) */
        struct lsg *element;       /* Pointer to the element owning the `point' */
};

/*
 * This structure is in a doubly linked list of nets and points to
 * a list of `points' in the same net linked by the `nextinnet' field.
 */
struct nethd {
        struct nethd *next;        /* Pointer to next nethd */
        struct nethd *prev;        /* Pointer to previous nethd */
        struct point *points;      /* Pointer to list of `points' in net */
        int     net;               /* Squid net id (Used upon readout) */
};

/*
 * This structure is used by the routines which generate primitives for
 * each different type of device. The names for the devices are generated
 * from their size and type. Each different primitive is created only once.
 */
struct sqcell {
        struct sqcell *next;       /* Pointer to next sqcell in list */
        char *name;                /* Pointer to generated name */
};

/*
 * this is an in-line max(a,b) macro
 */
#define max( a, b ) ( ( a ) > ( b ) ? ( a ) : ( b ) )
/* Here are the external variables and global type declarations */
char *sprintf();
extern struct lsg *elements;
extern struct point *points;
extern struct nethd nethead;
extern unsigned int num_elements;
extern int versflg;
extern char version[3];
extern int expfactor;
```

```
/*
 *  These are the global variables
 */
struct lsg *elements;                              /* Pointer to CABBAGE I elements */
struct point *points;                              /* Pointer to `points' list */
struct nethd nethead = { NULL, NULL, NULL };       /* Nethd head struct */
unsigned int num_elements;                         /* Number of CABBAGE I elements */
int versflg = 1;                                   /* Default version "1B" */
char version[3] = "1A";
int expfactor = 1;                                 /* Expansion factor */
```

```
/*
 * these are the design rules the translation program must know about.
 * the information consists of the size of overlaps from the 'bounding
 * box' of each element to one of the edges of a component of the
 * element.  For example, the variable D_L_P_TRAN represents the
 * 'Delta-Length-of Polysilicon-in a TRANsistor' and represents the
 * change in length required for a VERTICAL transistor to accurately
 * depict the polysilicon gate as a box. They are expressed in lambda
 * * 2, since the default units in Squid are 2 * user lambda.
 */
#define VERSION "(Mead/Conway Design Rules 3/21/82);\n"
#define D_W_D_DMCNT 0   /* Delta-Width in Diffusion in a Diff-Metal Contact */
#define D_L_D_DMCNT 0   /* Delta-Length in Diffusion in a Diff-Metal Contact */
#define D_W_D_BUCNT 0   /* Delta-Width in Diffusion in a Buried Contact */
#define D_L_D_BUCNT 0   /* Delta-Length in Diffusion in a Buried Contact */
#define D_W_D_TRAN 4    /* Delta-Width in Diffusion in a Transistor */
#define D_L_D_TRAN 0    /* Delta-Length in Diffusion in a Transistor */
#define D_W_P_PMCNT 0   /* Delta-Width in Polysilicon
#define D_L_P_PMCNT 0   /* Delta-Length in Polysilicon
#define D_W_P_BUCNT 0   /* Delta-Width in Polysilicon in a Buried Contact */
#define D_L_P_BUCNT 0   /* Delta-Length in Polysilicon in a Buried Contact */
#define D_W_P_TRAN 0    /* Delta-Width in Polysilicon in a Transistor */
#define D_L_P_TRAN 4    /* Delta-Length in Polysilicon in a Transistor */
#define D_W_M_DMCNT 0   /* Delta-Width in Metal in a Diff-Metal Contact */
#define D_L_M_DMCNT 0   /* Delta-Length in Metal in a Diff-Metal Contact */
#define D_W_M_PMCNT 0   /* Delta-Width in Metal in a Poly-Metal Contact */
#define D_L_M_PMCNT 0   /* Delta-Length in Metal in a Poly-Metal Contact */
#define D_W_C_DMCNT 2   /* Delta-Width in Contact Cut in a Diff-Metal Contact */
#define D_L_C_DMCNT 2   /* Delta-Length in Contact Ct in a Diff-Metal Contact */
#define D_W_C_PMCNT 2   /* Delta-Width in Contact Cut in a Poly-Metal Contact */
#define D_L_C_PMCNT 2   /* Delta-Length in Contact Ct in a Poly-Metal Contact */
#define D_W_I_TRAN 1    /* Delta-Width in Implant in a Transistor */
#define D_L_I_TRAN 1    /* Delta-Length in Implant in a Transistor */
#define D_W_B_BUCNT 2   /* Delta-Width in Buried Cut in a Buried Contact */
#define D_L_B_BUCNT 2   /* Delta-Length in Buried Cut in a Buried Contact */
#define D_W_M_BTCNT 0   /* Delta-Width in Metal in a Butting Contact */
#define D_L_M_BTCNT 0   /* Delta-Length in Metal in a Butting Contact */
#define D_W_C_BTCNT 2   /* Delta-Width in Contact in a Butting Contact */
#define D_L_C_BTCNT 2   /* Delta-Length in Contact in a Butting Contact */
#define D_O_D_BTCNT 2   /* Delta-Offset in Diffusion in a Butting Contact */
```

```
#include "cabtosquid.h"

struct point *
allocpoints( ele )                                              allocpoints
struct lsg  *ele;
{
/*
 *  Macro for insert sorting point into element list.
 */
#define INSERT(point,elem,xcoord,ycoord,ttype) \
        point = P_getpoint(); \
        point->element = elem; \
        point->type = ttype;\
        point->nextinnet = NULL; \
        point->netid = NULL; \
        point->x = xcoord; \
        point->y = ycoord; \
        for(tmppnt = &pointhead; tmppnt->next != NULL; tmppnt = tmppnt->next) {\
                if( point->x <= tmppnt->next->x ) \
                        break; \
        } \
        for( ; tmppnt->next != NULL && tmppnt->next->x == point->x; tmppnt = \
            tmppnt->next ) { \
                if( point->y <= tmppnt->next->y ) \
                        break; \
        } \
        if( tmppnt->next->x == point->x && tmppnt->next->y == point->y ) { \
                point->nextatpoint = tmppnt->next->nextatpoint; \
                tmppnt->next->nextatpoint = point; \
        } \
        else { \
                point->next = tmppnt->next; \
                tmppnt->next = point; \
        }

/*
 *  Macro for adding a new net to the netlist.
 */
#define ADDNET(point1,point2) \
        tmpnet = P_getnethd(); \
        point1->nextinnet = point2; \
        point2->nextinnet = (struct point  *)NULL; \
        point1->netid = tmpnet; \
        point2->netid = tmpnet; \
        tmpnet->points = point1; \
        tmpnet->prev = NULL; \
        tmpnet->next = nethead.next; \
        nethead.next = tmpnet; \
        if( tmpnet->next != NULL ) \
                tmpnet->next->prev = tmpnet; \

/*
 *  This routine allocates `point' data structures for each of the elements.
 *  One `point' is allocated for each terminal of the element.  Thus, lines
 *  have two points, transistors and loads have three points each, and
 *  the plethora of contacts each have a single point.
 */
struct point pointhead, *P_getpoint();
struct point *tmppnt, *pnt, *pnt1;
struct nethd *tmpnet, *P_getnethd();
        /* Initialize the head of the point list */
        pointhead.next = (struct point  *)NULL;
        /* For all of the elements in the element list ... */
        for( ; ele != NULL; ele = ele->next ) {
                /* If the element is RUNX, ignore it. */
                if( ele->type == FRUNX || ele->type == RUNX )
                        continue;
                /* If the element is a horizontal line ... */
                if( ele->x_center < 0 ) {                  /* =>HORIZONTAL line */
                        /* Insert lower point */
                        INSERT(pnt1,ele,ele->x_left_offset,ele->y_center,LLINE);
                        /* Insert upper point */
                        INSERT(pnt,ele,ele->x_right_offset,ele->y_center,ULINE);
                        /* Add a new net with the two points */
                        ADDNET( pnt1, pnt );
                }
                /* Else if the element is a vertical line ... */
                else if( ele->y_center < 0 ) {    /* =>Vertical Line */
                        /* Insert lower point */
                        INSERT(pnt,ele,ele->x_center,
                            ele->y_bottom_offset,LLINE);
                        /* Insert upper point */
                        INSERT(pnt1,ele,ele->x_center,ele->y_top_offset,ULINE);
                        /* Add a new net with the two points */
                        ADDNET( pnt, pnt1 );
                }
                /* Else if the element is some sort of transistor ... */
                else if( ele->type == TRAN || ele->type == LOAD ) {
                        /* Insert a point for the drain terminal */
                        INSERT(pnt,ele,ele->x_center,ele->y_center,DRAIN);
                        /* Insert a point for the gate terminal */
                        INSERT(pnt,ele,ele->x_center,ele->y_center,GATE);
                        /* Insert a point for the source terminal */
                        INSERT(pnt,ele,ele->x_center,ele->y_center,SOURCE);
                }
                else {  /* Some sort of contact */
                        /* Insert a single point for the structure */
                        INSERT(pnt,ele,ele->x_center,ele->y_center,POINT);
                }
        }
        return( pointhead.next );          /* Return a pointer to sorted list */
}
```

```
/* LINTLIBRARY (to shut lint up about unused functions) */

#include "cabtosquid.h"

/* global free list pointers and initializers */
struct lsg *P_lsgfree = (struct lsg *)NULL;
struct lsg P_lsginit;
struct point *P_pointfree = (struct point *)NULL;
struct point P_pointinit;
struct nethd *P_nethdfree = (struct nethd *)NULL;
struct nethd P_nethdinit;
struct sqcell *P_sqcellfree = (struct sqcell *)NULL;
struct sqcell P_sqcellinit;

#ifdef MDEBUG
/* memory management stat variables*/
int P_lsgget = 0;
int P_lsgret = 0;
int P_lsgsys = 0;
int P_pointget = 0;
int P_pointret = 0;
int P_pointsys = 0;
int P_nethdget = 0;
int P_nethdret = 0;
int P_nethdsys = 0;
int P_sqcellget = 0;
int P_sqcellret = 0;
int P_sqcellsys = 0;
#endif

#define MEMBLKSIZ 1024


/* lsg management */
```

P_getlsg

```
struct lsg *
P_getlsg()
{
struct lsg *tmplsg;
int i;
char *malloc();
#ifdef MDEBUG
        P_lsgget++;
#endif
        if( P_lsgfree == (struct lsg *)NULL ) {
#ifdef MDEBUG
                P_lsgsys++;
                printf("P_getlsg: Calling malloc\n");
#endif
        /* (=> Have to get a new block */
                P_lsgfree = (struct lsg *) malloc( MEMBLKSIZ * sizeof( struct lsg) );
                /* link the block together */
                for( i = 0; i < MEMBLKSIZ; i++ )
                        (P_lsgfree + 1)->next = P_lsgfree + i + 1;
                (P_lsgfree + MEMBLKSIZ - 1)->next = (struct lsg *)NULL;
        }
        tmplsg = P_lsgfree;
        P_lsgfree = P_lsgfree->next;
        *tmplsg = P_lsginit;
        return( tmplsg );
}
```

P_retlsg

```
P_retlsg( lsg )
struct lsg *lsg;
```

```
{
        if( lsg != (struct lsg *)NULL ) {
#ifdef MDEBUG
                P_lsgret++;
#endif
                lsg->next = P_lsgfree;
                P_lsgfree = lsg;
        }
}

/* point management */

struct point *
P_getpoint()                                                    P_getpoint
{
struct point *tmppoint;
int i;
char *malloc();
#ifdef MDEBUG
        P_pointget++;
#endif
        if( P_pointfree == (struct point *)NULL ) {
#ifdef MDEBUG
                P_pointsys++;
                printf("P_getpoint: Calling malloc\n");
#endif
        /* (=> Have to get a new block */
                P_pointfree = (struct point *) malloc( MEMBLKSIZ * sizeof( struct point) )
                /* link the block together */
                for( i = 0; i < MEMBLKSIZ; i++ )
                        (P_pointfree + i)->next = P_pointfree + i + 1;
                (P_pointfree + MEMBLKSIZ - 1)->next = (struct point *)NULL;
        }
        tmppoint = P_pointfree;
        P_pointfree = P_pointfree->next;
        *tmppoint = P_pointinit;
        return( tmppoint );
}

P_retpoint( point )                                             P_retpoint
struct point *point;
{
        if( point != (struct point *)NULL ) {
#ifdef MDEBUG
                P_pointret++;
#endif
                point->next = P_pointfree;
                P_pointfree = point;
        }
}

/* nethd management */

struct nethd *
P_getnethd()                                                    P_getnethd
{
struct nethd *tmpnethd;
int i;
char *malloc();
#ifdef MDEBUG
        P_nethdget++;
#endif
        if( P_nethdfree == (struct nethd *)NULL ) {
#ifdef MDEBUG
```

```
                P_nethdsys++;
                printf("P_getnethd: Calling malloc\n");
#endif
        /* (=> Have to get a new block */
                P_nethdfree = (struct nethd *) malloc( MEMBLKSIZ * sizeof( struct nethd);
                /* link the block together */
                for( i = 0; i < MEMBLKSIZ; i++ )
                        (P_nethdfree + i)->next = P_nethdfree + i + 1;
                (P_nethdfree + MEMBLKSIZ - 1)->next = (struct nethd *)NULL;
        }
        tmpnethd = P_nethdfree;
        P_nethdfree = P_nethdfree->next;
        *tmpnethd = P_nethdinit;
        return( tmpnethd );
}
```

P_retnethd

```
P_retnethd( nethd )
struct nethd *nethd;
{
        if( nethd != (struct nethd *)NULL ) {
#ifdef MDEBUG
                P_nethdret++;
#endif
                nethd->next = P_nethdfree;
                P_nethdfree = nethd;
        }
}

/* sqcell management */
```

P_getsqcell

```
struct sqcell *
P_getsqcell()
{
struct sqcell *tmpsqcell;
int i;
char *malloc();
#ifdef MDEBUG
        P_sqcellget++;
#endif
        if( P_sqcellfree == (struct sqcell *)NULL ) {
#ifdef MDEBUG
                P_sqcellsys++;
                printf("P_getsqcell: Calling malloc\n");
#endif
        /* (=> Have to get a new block */
                P_sqcellfree = (struct sqcell *) malloc( MEMBLKSIZ * sizeof( struct sqcell);
                /* link the block together */
                for( i = 0; i < MEMBLKSIZ; i++ )
                        (P_sqcellfree + i)->next = P_sqcellfree + i + 1;
                (P_sqcellfree + MEMBLKSIZ - 1)->next = (struct sqcell *)NULL;
        }
        tmpsqcell = P_sqcellfree;
        P_sqcellfree = P_sqcellfree->next;
        *tmpsqcell = P_sqcellinit;
        return( tmpsqcell );
}
```

P_retsqcell

```
P_retsqcell( sqcell )
struct sqcell *sqcell;
{
        if( sqcell != (struct sqcell *)NULL ) {
#ifdef MDEBUG
                P_sqcellret++;
#endif
```

```
                        sqcell->next = P_sqcellfree;
                        P_sqcellfree = sqcell;
                }
}

#ifdef MDEBUG
P_memstat()
{
                printf( "getlsg called %d times\n", P_lsgget );
                printf( "malloc called %d times\n", P_lsgsys );
                printf( "retlsg called %d times\n", P_lsgret );
                printf( "%d remain alloc\n", P_lsgget - P_lsgret );
                printf( "getpoint called %d times\n", P_pointget );
                printf( "malloc called %d times\n", P_pointsys );
                printf( "retpoint called %d times\n", P_pointret );
                printf( "%d remain alloc\n", P_pointget - P_pointret );
                printf( "getnethd called %d times\n", P_nethdget );
                printf( "malloc called %d times\n", P_nethdsys );
                printf( "retnethd called %d times\n", P_nethdret );
                printf( "%d remain alloc\n", P_nethdget - P_nethdret );
                printf( "getsqcell called %d times\n", P_sqcellget );
                printf( "malloc called %d times\n", P_sqcellsys );
                printf( "retsqcell called %d times\n", P_sqcellret );
                printf( "%d remain alloc\n", P_sqcellget - P_sqcellret );
}
#endif
```

*P_memstat*

checkversion

```
#include "cabtosquid.h"

checkversion(fp)
FILE *fp;
/*
 *  This routine checks the version number of CABBAGE I as
 *  stored in the intermediate file against the version
 *  number specififed on the run-line of cabtosquid.
 *  If they are not equal, or the formats are incorrect, the
 *  routine returns an error status.
 */
{
long int first_byte_count,second_byte_count;
int i;
struct lsg *lsg, *P_getlsg();
short int getshortint();
int status;
long int getlongint();
char *calloc();
        if( versflg == 0 ) {          /* If the original binary version ... */
                /* Binary byte count preceeds and follows each record */
                if((first_byte_count = getlongint(fp)) != 4)
                        return(NULL);
                version[0] = getc( fp );            /* Read 1st byte of version */
                version[1] = getc( fp );            /* Read 2nd byte of version */
                version[2] = '\0';          /* Terminate version string with NULL */
                if( strcmp( version, "1A" ) != 0 )          /* Version must be 1A */
                        return(NULL);
                else {  /* Read number of elements and second byte count */
                        num_elements = (unsigned)getshortint(fp);
                        second_byte_count = getlongint(fp);
                        if(first_byte_count != second_byte_count)
                                return(NULL);
                }
        }
        else if( versflg == 1 ) {               /* If the ASCII version */
                /* Use scanf to read in version number and number of elements */
                status = fscanf( fp, "%2s %d", version, &num_elements );
                if( status != 2 )           /* If there aren't two things scanned */
                        return( NULL );
                /* This version must be 1B */
                else if( strcmp( version, "1B" ) != 0 )
                        return( NULL );
        }
        else
                return( NULL );
        /* For each element in the file ... */
        for( i = num_elements; i > 0; i-- ) {
                lsg = P_getlsg();           /* Get element data structure */
                lsg->next = elements;       /* Link it in to list */
                elements = lsg;
        }
        return(OK);
}
```

connet

```
#include "cabtcsquid.h"

connet( point )
struct point  *point;
/*
 *  This routine connects nets at a point.  When properly sorted,
 *  the `nextinnet' field will link points at the same X and Y
 *  coordinates together.  Simple rules are followed to equivalence
 *  nets of the points that coincide.  Transistors are respective
 *  of the orientation of the lines intersecting them, and will only
 *  equivalence nets in a proper fashion.  The source of a transistor
 *  is defined to be the top or right side of the transistor, depending
 *  on its orientation.  All contacts are considered to equivalence
 *  all nets that coincide with their reference point.  If lines
 *  intersect without a contact, a same-layer-contact is added.
 *  This is necessary for the python data model (all interconnect
 *  lines are two segments only).
 */
{
register int trans = 0, lines = 0, conts = 0, terms = 0;
register int W = 0, L = 0, type;
register struct point *left, *bottom, *right, *top, *tmppnt;
register struct lsg *transistor, *contact, *terminal;
register int errflg = FALSE;
struct point  *P_getpoint();
struct lsg  *P_getlsg();
int delta;
        /* For each point at this coordinate ... */
        for( tmppnt = point; tmppnt != NULL; tmppnt = tmppnt->nextatpoint ) {
                /* If the element type is a transistor or load ... */
                if( tmppnt->element->type == TRAN || tmppnt->element->type ==
                    LOAD ) {
                        /* Record the presence of a transistor */
                        trans++;
                        /* Record a pointer to the transistor */
                        transistor = tmppnt->element;
                        /* Switch on point type */
                        switch( tmppnt->type ) {
                                case DRAIN:     /* If point for drain ... */
                                        /* If HORIZONTAL orientation ... */
                                        if( tmppnt->element->orientation ==
                                            HORIZONTAL ) {
                                                /* Bottom represents drain */
                                                bottom = tmppnt;
                                        }
                                        /* Else ... */
                                        else {
                                                /* Left represents drain */
                                                left = tmppnt;
                                        }
                                        break;
                                case GATE:      /* If point for gate ... */
                                        /* If HORIZONTAL orientation ... */
                                        if( tmppnt->element->orientation ==
                                            HORIZONTAL ) {
                                                /* Left represents gate */
                                                left = tmppnt;
                                                /* Right represents gate */
                                                right = tmppnt;
                                        }
                                        /* Else ... */
                                        else {
```

```
#include "cabtosquid.h"

equivalence( point1, point2 )                                    equivalence
struct point *point1, *point2;
/*
 * This routine equivalences the nets between two points.  If
 * neither of them is yet in a net, a new net is created and
 * the two points are placed in it.  If one or the other point
 * is in a net, the one not in a net is added to the net of the
 * other one.  If both are in a net, the two nets are merged.
 */
{
struct nethd *tmpnet, *P_getnethd();
struct point *tmppnt;
        /* If they are already in the same net, ignore them */
        if( point1->netid != NULL && point1->netid == point2->netid )
                /* Already in same net */
                return;
        /* If the first point is not yet in a net ... */
        if( point1->netid == NULL ) {     /* Not in a net yet */
                /* If the second point is not yet in a net ... */
                if( point2->netid == NULL ) { /* Not in a net yet */
                        /* Create a new net */
                        tmpnet = P_getnethd();
                        /* And point to the new net */
                        tmpnet->points = point1;
                        point1->nextinnet = point2;
                        point2->nextinnet = (struct point *)NULL;
                        /* Set the netid to point to the net head */
                        point1->netid = tmpnet;
                        point2->netid = tmpnet;
                        /* And link the new net into the netlist */
                        tmpnet->next = nethead.next;
                        tmpnet->prev = NULL;
                        nethead.next = tmpnet;
                        if( tmpnet->next != NULL )
                                tmpnet->next->prev = tmpnet;
                }
                else {                  /* Add point1 into point2's net */
                        point1->netid = point2->netid;
                        point1->nextinnet = point2->netid->points;
                        point2->netid->points = point1;
                }
        }
        else if( point2->netid == NULL ) {
        /* Add point2 into poin1's net */
                /* Put point2 into point1's net */
                point2->netid = point1->netid;
                point2->nextinnet = point1->netid->points;
                point1->netid->points = point2;
        }
        else {          /* Merge two nets together */
                /* Add point2 list to point1 list */
                /* Save pointer to point2 net head */
                tmpnet = point2->netid;
                /* For all points in point2 net list ... */
                for( tmppnt = tmpnet->points; tmppnt != NULL; tmppnt =
                    tmppnt->nextinnet ) {
                        /* Reset net id to point to point1 net id */
                        tmppnt->netid = point1->netid;  /* Reset Netid */
                        /* Break immediately before end of point2 list */
```

```
				if( tmppnt->nextinnet == NULL ) /* End of point2 list */
					break;
			}
			/* End of point2 list gets head of point1 list */
			tmppnt->nextinnet = point1->netid->points;
			/* Head of point1 list gets head of point2 list */
			point1->netid->points = tmpnet->points;
			/* Unlink tmpnet from net list */
			if( tmpnet->prev != NULL ) {
				tmpnet->prev->next = tmpnet->next;
			}
			else {	/* Unlink tmpnet from net list head */
				nethead.next = tmpnet->next;
			}
			/* Patch next net list head around tmpnet */
			if( tmpnet->next != NULL )
				tmpnet->next->prev = tmpnet->prev;
			/* Return the removed net list head to memory management */
			P_retnethd( tmpnet );
		}
}
```

```
                                                /* Bottom represents gate */
                                                bottom = tmppnt;
                                                /* Top represents gate */
                                                top = tmppnt;
                                        }
                                        break;
                        case SOURCE:            /* If point for source ... */
                                        /* If HORIZONTAL orientation ... */
                                        if( tmppnt->element->orientation ==
                                            HORIZONTAL ) {
                                                /* Top represents source */
                                                top = tmppnt;
                                        }
                                        /* Else ... */
                                        else {
                                                /* Right represents source */
                                                right = tmppnt;
                                        }
                                        break;
                }
        }
        /* Else if the element is a line ... */
        else if( tmppnt->element->type < FRUNX ) {
                lines++;                /* Record that a line is present */
        }
        /* Else if the element is a terminal ... */
        else if( tmppnt->element->type == TERM ) {
                terms++;                /* Record that a terminal is present */
                terminal = tmppnt->element;     /* Record the term */
        }
        /* Else ... (=> a contact) */
        else {
                conts++;                /* Record that a contact is present */
                contact = tmppnt->element;      /* Record the contact */
        }
}
/* There can only be one transistor or one contact at any coordinate */
if( (trans / 3 + conts + terms ) > 1 ) {        /* Illegal condition */
        fprintf( stderr, "Multiple point structures overlapping.\n" );
        fprintf( stderr, "X = %d, Y = %d.\n", point->x, point->y );
        errflg++;                       /* Prepare to bomb after this routine */
}
else if( trans == 0 ) {                 /* No transistors => easy time */
        /* Go through lines - compute contact width, height */
        for( tmppnt = point; tmppnt != NULL; tmppnt =
            tmppnt->nextatpoint ) {
                /* Choose a line type from the line endpoints */
                if( tmppnt->element->type < FRUNX )
                        type = tmppnt->element->type % 4;
                /* Choose the width and height from the data */
                if( tmppnt->element->orientation == VERTICAL ) {
                        delta = tmppnt->element->x_left_offset +
                            tmppnt->element->x_right_offset;
                        W = max( delta, W );
                }
                else {
                        delta = tmppnt->element->y_top_offset +
                            tmppnt->element->y_bottom_offset;
                        L = max( delta, L );
                }
        }
        /* If no contacts or terminals either ... */
        if( conts == 0 && terms == 0 ) {
                /* Must put a same-layer contact */
                /* Set up default width and height if necessary */
                if( L == 0 )
                        L = W;
                else if( W == 0 )
                        W = L;
                /* Get a point from memory management */
                tmppnt = P_getpoint();
                /* Link into point list */
                tmppnt->nextatpoint = point->nextatpoint;
                point->nextatpoint = tmppnt;
                /* Initialize coordinate values */
                tmppnt->x = point->x;
                tmppnt->y = point->y;
                /* Initialize net pointer */
                tmppnt->nextinnet = NULL;
                /* Initialize type */
                tmppnt->type = POINT;
                /* Get an element from memory management */
                contact = P_getlsg();
                /* Create contact from line intersection */
                contact->x_center = tmppnt->x;
                contact->x_left_offset = W / 2;
                contact->x_right_offset = W / 2;
                contact->y_center = tmppnt->y;
                contact->y_bottom_offset = L / 2;
                contact->y_top_offset = L / 2;
                /* Un-rotated orientation */
                contact->orientation = HORIZONTAL;
                /* Link into elements list */
                contact->next = elements;
                elements = contact;
                tmppnt->element = contact;
                /* Switch on the type of contact */
                switch( type ) {
                        case DIFF:      /* Generate a D-D contact */
                                contact->type = D_D_CNT;
                                break;
                        case POLY:      /* Generate a P-P contact */
                                contact->type = P_P_CNT;
                                break;
                        case METAL:     /* Generate a M-M contact */
                                contact->type = M_M_CNT;
                                break;
                }
                gencont( contact );
        }
        else if( terms != 0 ) {         /* Line terminator present */
                switch( type ) {        /* Figure out line type */
                        case DIFF:
                                terminal->name = "ND";
                                break;
                        case METAL:
                                terminal->name = "NM";
                                break;
                        case POLY:
                                terminal->name = "NP";
                                break;
                        default:
                                fprintf( stderr, "Bad type %d.\n",
                                    type );
                                break;
                }
        }
        /* For all points at this coordinate */
        for( tmppnt = point; tmppnt != NULL; tmppnt =
            tmppnt->nextatpoint ) {
                /* Equivalence the nets of the points */
                equivalence( point, tmppnt );
        }
}
else {          /* A transistor */
        /* For all of the coordinate points ... */
        for( tmppnt = point; tmppnt != NULL; tmppnt =
            tmppnt->nextatpoint ) {
                /* Skip over the ONE transistor at this coordinate */
                if( tmppnt->element == transistor ) {
                        continue;
                }
                /* If the line found is a vertical line */
                if( tmppnt->element->orientation == VERTICAL ) {
                        /* If the point is the upper endpoint */
                        if( tmppnt->type == ULINE ) {
                                /* Peel line back into terminal frame */
                                tmppnt->element->y_top_offset =
                                    bottom->element->y_center -
                                    bottom->element->y_bottom_offset;
                                /* Equivalence the nets of the points */
                                equivalence( bottom, tmppnt );
                        }
                        /* Else, if the point is the lower endpoint */
                        else {
                                /* Peel line back into terminal frame */
                                tmppnt->element->y_bottom_offset =
                                    top->element->y_center +
                                    top->element->y_top_offset;
                                /* Equivalence the nets of the points */
                                equivalence( top, tmppnt );
                        }
                }
                /* Else, the line is HORIZONTAL ... */
                else {
                        /* If the point is the upper endpoint */
                        if( tmppnt->type == ULINE ) {
                                /* Peel line back into terminal frame */
                                tmppnt->element->x_right_offset =
                                    left->element->x_center -
                                    left->element->x_left_offset;
                                /* Equivalence the nets of the points */
                                equivalence( left, tmppnt );
                        }
                        /* Else, if the point is the lower endpoint */
                        else {
                                /* Peel line back into terminal frame */
                                tmppnt->element->x_left_offset =
                                    right->element->x_center +
                                    right->element->x_right_offset;
                                /* Equivalence the nets of the points */
                                equivalence( right, tmppnt );
                        }
                }
        }
}
```

```
        return( errflg );           /* Die if overlapping point structures exist */
}
```

```
#include "cabtosquid.h"
#include "sq.h"
```

*genprimitives*

```
genprimitives()
/*
 * This routine supervises generation of Squid master cells
 * from the point structure elements read from the CABBAGE I
 * file.  Separate routines are used for transistors and
 * contacts.  Each routine keeps track of the master cells
 * which have already been generated, so that the same
 * cells are not generated more than once.
 */
{
register struct lsg *tmpele;
        /* For all of the CABBAGE I elements */
        for( tmpele = elements; tmpele != NULL; tmpele = tmpele->next ) {
                /* Switch on the type of element */
                switch( tmpele->type ) {
                        case TRAN:      /* Enhancement transistor */
                                gentran( tmpele, FALSE );
                                break;
                        case LOAD:      /* Depletion load */
                                gentran( tmpele, TRUE );
                                break;
                        case D_M_CNT:   /* Diffusion-Metal contact */
                                gencont( tmpele );
                                break;
                        case P_M_CNT:   /* Polysilicon-Metal contact */
                                gencont( tmpele );
                                break;
                        case BUR_CNT:   /* Buried (Poly-Diffusion) contact */
                                gencont( tmpele );
                                break;
                        case BUT_CNT:   /* Butting contact */
                                genbutcnt( tmpele );
                                break;
                }
        }
        return(OK);
}
```

*gentran*

```
gentran( element, loadflg )
struct lsg *element;
int loadflg;
/*
 * This routine generates master cells for transistor (enhancement
 * and depletion) CABBAGE I elements.  Each different size of
 * master cell is created only once.  A transistor master cell
 * consists of a polysilicon protection frame coincident with
 * the transistor gate, a diffusion protection frame coincident
 * with the diffusion source-channel-drain, an active area
 * protection frame coincident with the active area of the
 * transistor (defined as overlap of polysilicon and diffusion),
 * two terminal frames at the ends of the gate protection
 * frame (defined as polysilicon and (NOT diffusion)), and one
 * terminal frame at each end of the diffusion protection
 * frame for the source and drain regions (defined as
 * diffusion and (NOT polysilicon)).  A depletion transistor
 * also has an implant protection frame surrounding the active
 * area of the device.
 *
 * NOTE:  The W and L generated for identification of the device
 * are bounding box values, *NOT* active area values.
 */
```

...gentran

```
{
char name[BUFSIZ], *strsav();
static struct sqcell transistors = { NULL };
struct sqcell *tmptran, *P_getsqcell();
int W, L;
SQView view;
FILE *stream;
SQStatus status;
SQGeo geo;
SQTerm term;
        /* If the element is horizontal ... */
        if( element->orientation == HORIZONTAL ) {
                /* Get W and L directly */
                W = element->x_right_offset + element->x_left_offset;
                L = element->y_top_offset + element->y_bottom_offset;
        }
        else {  /* Element is vertically oriented */
                /* Get W and L from opposite coordinates */
                W = element->y_top_offset + element->y_bottom_offset;
                L = element->x_right_offset + element->x_left_offset;
        }
        /* Generate the name of the device from W and L */
        if( loadflg ) {
                sprintf( name, "%dby%dl", W, L );
        }
        else {
                sprintf( name, "%dby%dt", W, L );
        }
        /* Check to see if this transistor has already been generated */
        for( tmptran = &transistors; tmptran->next != NULL; tmptran =
            tmptran->next ) {
                /* If it has been generated ... */
                if( strcmp( name, tmptran->next->name ) == 0 ) {
                        /* Set name pointer to saved name */
                        element->name = tmptran->next->name;
                        return;         /* Return happy */
                }
        }
        /* Get a Squid cell from memory management */
        tmptran->next = P_getsqcell();
        tmptran = tmptran->next;
        /* Save the name */
        tmptran->name = strsav( name );
        element->name = tmptran->name;
        /* Set up to open a view with the masterCell's name */
        view.cell = tmptran->name;
        view.view = "layout";
        view.mode = "w";
        /* If the view cannot be created, assume it already exists ... */
        if( (status = SQ(sqCreate, sqView, view, &stream)) <= 0 ) {
                /* ... and open the already existing view ... */
                if( (status = SQ(sqOpen, sqView, view, &stream)) <= 0 ) {
                        fprintf( stderr, "Couldn't open view %s/layout.%d\n",
                            name, status);
                        exit(2);
                }
                /* ... and delete it. (Show no mercy!) */
                SQ(sqRm,sqView);
                /* Now if the view cannot be created ... */
                if( (status = SQ(sqCreate, sqView, view, &stream)) <= 0 ) {
                        /* Holler like mad! */
                        fprintf( stderr, "Couldn't create view %s/layout.%d\n",
                            name, status);
                        exit(3);
                }
        }
        /* Prepare to create terminals on the device */
        term.instID = NULL;
        term.netID = NULL;
        term.name = "d";
        /* Create the drain terminal */
        if( SQ(sqCreate,sqTerm,term) <= 0 ) {
                fprintf( stderr, "No master term.\n" );
        }
        term.name = "g";
        /* Create the gate terminal */
        if( SQ(sqCreate,sqTerm,term) <= 0 ) {
                fprintf( stderr, "No master term.\n" );
        }
        term.name = "s";
        /* Create the source terminal */
        if( SQ(sqCreate,sqTerm,term) <= 0 ) {
                fprintf( stderr, "No master term.\n" );
        }
        /* Prepare to generate the diffusion protection frame ... */
        geo.layer = "ND";
        geo.manhattanP = sqTrue;
        geo.geoType = sqRect;
        /* Add in the offsets to properly define the diffusion rectangle */
        geo.def.rect.l = -W / 2 + D_W_D_TRAN;
        geo.def.rect.b = -L / 2 + D_L_D_TRAN;
        geo.def.rect.r = W / 2 - D_W_D_TRAN;
        geo.def.rect.t = L / 2 - D_L_D_TRAN;
        geo.function = sqFrame;
        /* Create the frame */
        if( (status = SQ(sqCreate, sqGeo, &geo)) <= 0 ) {
                fprintf( stderr, "Couldn't create geometry! %d\n", status );
                exit(4);
        }
        /* Prepare to generate the diffusion source terminal frame ... */
        geo.function = sqTermArea;
        geo.implements.term = "s";
        geo.def.rect.b = L / 2 - D_L_P_TRAN;
        if( (status = SQ(sqCreate, sqGec, &geo)) <= 0 ) {
                fprintf( stderr, "Couldn't create geometry! %d\n", status );
                exit(5);
        }
        /* Prepare to generate the diffusion drain terminal frame ... */
        geo.implements.term = "d";
        geo.def.rect.b = -L / 2 + D_L_D_TRAN;
        geo.def.rect.t = -L / 2 + D_L_P_TRAN;
        if( (status = SQ(sqCreate, sqGeo, &geo)) <= 0 ) {
                fprintf( stderr, "Couldn't create geometry! %d\n", status );
                exit(6);
        }
        /* Prepare to generate the polysilicon protection frame ... */
        geo.layer = "NP";
        /* Add in the offsets to properly define the polysilicon rectangle */
        geo.def.rect.l = -W / 2 + D_W_P_TRAN;
        geo.def.rect.b = -L / 2 + D_L_P_TRAN;
        geo.def.rect.r = W / 2 - D_W_P_TRAN;
        geo.def.rect.t = L / 2 - D_L_P_TRAN;
        geo.function = sqFrame;
        if( (status = SQ(sqCreate, sqGeo, &geo)) <= 0 ) {
                fprintf( stderr, "Couldn't create geometry! %d\n", status );
                exit(7);
        }
```

*...gentran*

```
        /* Prepare to generate the polysilicon gate terminal frames ...  */
        geo.function = sqTermArea;
        geo.implements.term = "g";
        geo.def.rect.r = -W / 2 + D_W_D_TRAN;
        if( (status = SQ(sqCreate, sqGeo, &geo)) <= 0 ) {
                fprintf( stderr, "Couldn't create geometry! %d\n", status );
                exit(8);
        }
        geo.def.rect.r = W / 2 - D_W_P_TRAN;
        geo.def.rect.l = W / 2 - D_W_D_TRAN;
        if( (status = SQ(sqCreate, sqGeo, &geo)) <= 0 ) {
                fprintf( stderr, "Couldn't create geometry! %d\n", status );
                exit(9);
        }
        /* If this is depletion device, generate implant protection frame */
        if( loadflg ) {
                geo.layer = "NI";
                geo.function = sqFrame;
                geo.def.rect.l = -W / 2 + D_W_I_TRAN;
                geo.def.rect.b = -L / 2 + D_L_I_TRAN;
                geo.def.rect.r = W / 2 - D_W_I_TRAN;
                geo.def.rect.t = L / 2 - D_L_I_TRAN;
                if( (status = SQ(sqCreate, sqGeo, &geo)) <= 0 ) {
                        fprintf( stderr, "Couldn't create geometry! %d\n",
                            status );
                        exit(10);
                }
        }
        /* Generate an active area protection frame */
        geo.layer = "NT";
        geo.function = sqFrame;
        geo.def.rect.l = -W / 2 + D_W_D_TRAN;
        geo.def.rect.b = -L / 2 + D_L_P_TRAN;
        geo.def.rect.r = W / 2 - D_W_D_TRAN;
        geo.def.rect.t = L / 2 - D_L_P_TRAN;
        if( (status = SQ(sqCreate, sqGeo, &geo)) <= 0 ) {
                fprintf( stderr, "Couldn't create geometry! %d\n", status );
                exit(11);
        }
        /* Save the view of the transistor */
        if( SQ(sqSave, sqView) <= 0 ) {
                fprintf( stderr, "Couldn't save view %s.\n", name );
                exit(12);
        }
}
```

*gencont*

```
gencont( element )
struct lsg *element;
/*
 * This routine generates diffusion-metal, polysilicon-metal, and
 * polysilicon-diffusion (buried) contacts, as well as all
 * same-layer contacts for intersections of lines.  Contacts
 * are single terminal devices and have the same name 'i' for
 * all terminals on all layers.  Each layer has a protection
 * frame with a coincident terminal frame.
 */
{
char name[BUFSIZ], *strsav();
static struct sqcell contacts = { NULL };
struct sqcell *tmpcnt, *P_getsqcell();
int D_W_L1, D_L_L1, D_W_L2, D_L_L2, D_W_LC, D_L_LC;
char *layer1, *layer2, *layerc;
int W, L;
SQView view;
```

...gencont

```
FILE *stream;
SQStatus status;
SQGeo geo;
SQTerm term;
        /* Choose W and L from orientation (HORIZONTAL default) */
        if( element->orientation == HORIZONTAL ) {
                W = element->x_right_offset + element->x_left_offset;
                L = element->y_top_offset + element->y_bottom_offset;
        }
        else {
                W = element->y_top_offset + element->y_bottom_offset;
                L = element->x_right_offset + element->x_left_offset;
        }
        /* Switch on type of contact */
        switch( element->type ) {
                case D_M_CNT:  /* Diffusion-Metal contact */
                        /* Generate contact name */
                        sprintf( name, "%dby%ddmc", W, L );
                        /* Define the layers for the contact */
                        layer1 = "ND";
                        layer2 = "NM";
                        layerc = "NC";
                        /* Set the proper offsets for the contact */
                        D_W_L1 = D_W_D_DMCNT;
                        D_L_L1 = D_L_D_DMCNT;
                        D_W_L2 = D_W_M_DMCNT;
                        D_L_L2 = D_L_M_DMCNT;
                        D_W_LC = D_W_C_DMCNT;
                        D_L_LC = D_L_C_DMCNT;
                        break;
                case P_M_CNT:  /* Polysilicon-Metal contact */
                        /* Generate contact name */
                        sprintf( name, "%dby%dpmc", W, L );
                        /* Define the layers for the contact */
                        layer1 = "NP";
                        layer2 = "NM";
                        layerc = "NC";
                        /* Set the proper offsets for the contact */
                        D_W_L1 = D_W_P_PMCNT;
                        D_L_L1 = D_L_P_PMCNT;
                        D_W_L2 = D_W_M_PMCNT;
                        D_L_L2 = D_L_M_PMCNT;
                        D_W_LC = D_W_C_PMCNT;
                        D_L_LC = D_L_C_PMCNT;
                        break;
                case BUR_CNT:  /* Polysilicon-Diffusion (Buried) contact */
                        /* Generate contact name */
                        sprintf( name, "%dby%dburc", W, L );
                        /* Define the layers for the contact */
                        layer1 = "ND";
                        layer2 = "NP";
                        layerc = "NB";
                        /* Set the proper offsets for the contact */
                        D_W_L1 = D_W_D_BUCNT;
                        D_L_L1 = D_L_D_BUCNT;
                        D_W_L2 = D_W_P_BUCNT;
                        D_L_L2 = D_L_P_BUCNT;
                        D_W_LC = D_W_B_BUCNT;
                        D_L_LC = D_L_B_BUCNT;
                        break;
                case D_D_CNT:  /* Diffusion-diffusion (same-layer) contact */
```

```
                        /* Generate contact name */
                        sprintf( name, "%dby%ddterm", W, L );
                        layer1 = "ND";
                        layer2 = NULL;
                        layerc = NULL;
                        /* All offsets are zero for this case */
                        D_W_L1 = 0;
                        D_L_L1 = 0;
                        break;
                case P_P_CNT:  /* Poly-poly (same-layer) contact */
                        /* Generate contact name */
                        sprintf( name, "%dby%dpterm", W, L );
                        layer1 = "NP";
                        layer2 = NULL;
                        layerc = NULL;
                        /* All offsets are zero for this case */
                        D_W_L1 = 0;
                        D_L_L1 = 0;
                        break;
                case M_M_CNT:  /* Metal-metal (same-layer) contact */
                        /* Generate contact name */
                        sprintf( name, "%dby%dmterm", W, L );
                        layer1 = "NM";
                        layer2 = NULL;
                        layerc = NULL;
                        /* All offsets are zero for this case */
                        D_W_L1 = 0;
                        D_L_L1 = 0;
                        break;
        }
        /* Check to see if this contact has been generated already */
        for( tmpcnt = &contacts; tmpcnt->next != NULL; tmpcnt =
            tmpcnt->next ) {
                /* If it has already been generated ... */
                if( strcmp( name, tmpcnt->next->name ) == 0 ) {
                        element->name = tmpcnt->next->name;
                        return;         /* Return happy */
                }
        }
        /* Get a Squid cell from memory management */
        tmpcnt->next = P_getsqcell();
        tmpcnt = tmpcnt->next;
        /* Save the name of the master cell */
        tmpcnt->name = strsav( name );
        element->name = tmpcnt->name;
        /* Prepare to create a view of the new master cell */
        view.cell = tmpcnt->name;
        view.view = "layout";
        view.mode = "w";
        /* If the view cannot be created, assume it already exists ... */
        if( (status = SQ(sqCreate, sqView, view, &stream)) <= 0 ) {
                /* ... and open the already existing view ... */
                if( (status = SQ(sqOpen, sqView, view, &stream)) <= 0 ) {
                        fprintf( stderr, "Couldn't open view %s/layout.%d\n",
                            name, status);
                        exit(13);
                }
                /* ... and delete it. (Show no mercy!) */
                SQ(sqRm,sqView);
                /* Now if the view cannot be created ... */
                if( (status = SQ(sqCreate, sqView, view, &stream)) <= 0 ) {
```

...gencont

```
                        /* Holler like mad! */
                        fprintf( stderr, "Couldn't create view %s/layout.%d\n",
                            name, status);
                        exit(14);
                }
        }
        /* Generate a single terminal 'i' for the contact */
        term.instID = NULL;
        term.netID = NULL;
        term.name = "i";
        if( SQ(sqCreate,sqTerm,term) <= 0 ) {
                fprintf( stderr, "No master term.\n" );
        }
        /* Generate the first layer of the contact */
        geo.manhattanP = sqTrue;
        geo.geoType = sqRect;
        geo.layer = layer1;
        geo.function = sqFrame;
        geo.def.rect.l = -W / 2 + D_W_L1;
        geo.def.rect.b = -L / 2 + D_L_L1;
        geo.def.rect.r = W / 2 - D_W_L1;
        geo.def.rect.t = L / 2 - D_L_L1;
        if( (status = SQ(sqCreate, sqGeo, &geo)) <= 0 ) {
                fprintf( stderr, "Couldn't create geometry! %d\n", status );
                exit(15);
        }
        /* Generate the terminal of the first layer of the contact */
        geo.function = sqTermArea;
        geo.implements.term = "i";
        geo.def.rect.l = -W / 2 + D_W_L1;
        geo.def.rect.b = -L / 2 + D_L_L1;
        geo.def.rect.r = W / 2 - D_W_L1;
        geo.def.rect.t = L / 2 - D_L_L1;
        if( (status = SQ(sqCreate, sqGeo, &geo)) <= 0 ) {
                fprintf( stderr, "Couldn't create geometry! %d\n", status );
                exit(16);
        }
        if( (geo.layer = layer2) != NULL ) {
                /* Generate the second layer of the contact */
                geo.function = sqFrame;
                geo.def.rect.l = -W / 2 + D_W_L2;
                geo.def.rect.b = -L / 2 + D_L_L2;
                geo.def.rect.r = W / 2 - D_W_L2;
                geo.def.rect.t = L / 2 - D_L_L2;
                if( (status = SQ(sqCreate, sqGeo, &geo)) <= 0 ) {
                        fprintf( stderr, "Couldn't create geometry! %d\n",
                            status );
                        exit(18);
                }
                /* Generate the terminal of the second layer of the contact */
                geo.function = sqTermArea;
                geo.implements.term = "i";
                geo.def.rect.l = -W / 2 + D_W_L2;
                geo.def.rect.b = -L / 2 + D_L_L2;
                geo.def.rect.r = W / 2 - D_W_L2;
                geo.def.rect.t = L / 2 - D_L_L2;
                if( (status = SQ(sqCreate, sqGeo, &geo)) <= 0 ) {
                        fprintf( stderr, "Couldn't create geometry! %d\n",
                            status );
                        exit(19);
                }
        }
        if( (geo.layer = layerc) != NULL ) {
                /* Generate the contact hole layer of the contact */
                geo.function = sqFrame;
                geo.def.rect.l = -W / 2 + D_W_LC;
                geo.def.rect.b = -L / 2 + D_L_LC;
                geo.def.rect.r = W / 2 - D_W_LC;
                geo.def.rect.t = L / 2 - D_L_LC;
                if( (status = SQ(sqCreate, sqGeo, &geo)) <= 0 ) {
                        fprintf( stderr, "Couldn't create geometry! %d\n",
                            status );
                        exit(20);
                }
                /* Generate the contact hole terminal of the contact */
                geo.function = sqTermArea;
                geo.implements.term = "i";
                geo.def.rect.l = -W / 2 + D_W_LC;
                geo.def.rect.b = -L / 2 + D_L_LC;
                geo.def.rect.r = W / 2 - D_W_LC;
                geo.def.rect.t = L / 2 - D_L_LC;
                if( (status = SQ(sqCreate, sqGeo, &geo)) <= 0 ) {
                        fprintf( stderr, "Couldn't create geometry! %d\n",
                            status );
                        exit(120);
                }
        }
        /* Save the master cell view */
        if( SQ(sqSave, sqView) <= 0 ) {
                fprintf( stderr, "Couldn't save view %s.\n", name );
                exit(21);
        }
}
```

*genbutcnt*

```
genbutcnt( element )
struct lsg *element;
/*
 * This routine handles butting contacts.  A special routine
 * is required since butting contacts are not symmetric.
 * It is not implemented at this time.
 */
{
char name[BUFSIZ], *strsav();
static struct sqcell contacts = { NULL };
struct sqcell *tmpcnt, *P_getsqcell();
int W, L;
SQView view;
FILE *stream;
SQStatus status;
SQGeo geo;
SQTerm term;
        if( element->orientation == HORIZONTAL ) {
                W = element->x_right_offset + element->x_left_offset;
                L = element->y_top_offset + element->y_bottom_offset;
        }
        else {
                W = element->y_top_offset + element->y_bottom_offset;
                L = element->x_right_offset + element->x_left_offset;
        }
        sprintf( name, "%dby%dbutc", W, L );
        for( tmpcnt = &contacts; tmpcnt->next != NULL; tmpcnt =
            tmpcnt->next ) {
                if( strcmp( name, tmpcnt->next->name ) == 0 ) {
                        element->name = tmpcnt->next->name;
                        return;
                }
        }
        tmpcnt->next = P_getsqcell();
	tmpcnt = tmpcnt->next;
	tmpcnt->name = strsav( name );
	element->name = tmpcnt->name;
	view.cell = tmpcnt->name;
	view.view = "layout";
	view.mode = "w";
	if( (status = SQ(sqCreate, sqView, view, &stream)) <= 0 ) {
		if( (status = SQ(sqOpen, sqView, view, &stream)) <= 0 ) {
			fprintf( stderr, "Couldn't open view %s/layout.%d\n",
			    name, status);
			exit(22);
		}
		SQ(sqRm,sqView);
		if( (status = SQ(sqCreate, sqView, view, &stream)) <= 0 ) {
			fprintf( stderr, "Couldn't create view %s/layout.%d\n",
			    name, status);
			exit(23);
		}
	}
	term.instID = NULL;
	term.netID = NULL;
	term.name = "i";
	if( SQ(sqCreate,sqTerm,term) <= 0 ) {
		fprintf( stderr, "No master term.\n" );
	}
	printf( "Sorry, genbutcnt not finished yet.\n" );
}
```

init

```
#include "cabtosquid.h"

FILE *
init(argc,argv)
int argc;
char *argv[];
/*
 * This routine parses the run line arguments and set options
 * accordingly. It allows the following options: `e' followed
 * immediately by a number allows an expansion factor, which
 * changes the locations of all elements without changing their
 * size (other than line length). This is necessary in some
 * cases. Python allows terminal AREAS, whereas CABBAGE I
 * requires terminals to connect at specific points. Since
 * the translation of contacts into Squid format changes
 * these terminal points on contacts to terminal areas, problems
 * can occur. One such problem occurs when BOTH endpoints of
 * a line end up within a terminal frame. Since there is no
 * explicit connectivity within the Squid database, there is
 * no way to determine which endpoint is connected to the
 * terminal frame. Expanding the layout pushes contacts farther
 * apart, so only one endpoint is contained within each terminal
 * frame.
 *      The second options allow newer or older versions of
 * the CABBAGE I format to be translated.
 */
{
char    *s;
FILE *fp, *fopen();

        fp = (FILE *)NULL;
        while(--argc > 0 && (*++argv)[0] == '-'){
                for(s = argv[0] + 1; *s != '\0';s++){
                        switch( *s){

                        case 'e':       /* Expansion factor */
                                expfactor = atoi( ++s );
                                if( expfactor < 1 )
                                        expfactor = 1;
                                *s-- = '\0';
                                printf( "Exp factor is %d.\n", expfactor );
                                break;

                        case 'N':       /* Version newer than default */
                                versflg++;
                                break;

                        case 'O':       /* Version older than default */
                                versflg--;
                                break;

                        default:
                                fprintf(stderr,
                                    "cabtosquid: illegal option: %c\n", *s);
                                argc = -1;
                                break;
                        }
                }
        }
        if(argc < 0)
                fprintf(stderr,"usage: cabtosquid [-NOe#] inpfil\n");
        else if( argc == 0 )
                fp = stdin;
        else if((fp = fopen( *argv,"r")) == NULL)
```

```
                fprintf(stderr,"cabtosquid: can't open %s\n", *argv);
        return(fp);
}
```

main

```
#include "cabtosquid.h"
#include "private.h"

main(argc,argv)
/*
 *      Main checks call syntax issuing error messages if syntax or usage
 *  is incorrect.  it sets the flags for the options, and attempts to read
 *  in the file(s) specified, issuing an error message if the file does not
 *  exist.  It then reads in the cabbage file (from stdin if no input is
 *  specified) and writes it out again in Squid format, suitable for viewing
 *  with Hawk, or spacing with Python (CABBAGE II)
 */
int argc;
char *argv[];
{
FILE *fp, *init();

        if( ( fp = init( argc, argv ) ) == NULL )
                ; /* Do nothing - all error messages will have been written */
        else if( checkversion( fp ) == NULL )
                fprintf( stderr, "Cabbage file not created by version %.2s\n",
                    version );
        else if( readcabbage( fp ) == NULL )
                fprintf( stderr, "Cabbage file read failure\n" );
        else if( SQBegin() <= 0 )
                fprintf( stderr ,"Couldn't SQBegin.\n" );
        else if( genprimitives() == NULL )
                fprintf( stderr, "Squid primitives generation failure.\n" );
        else if( splitwires() == NULL )
                fprintf( stderr, "Problem Splitting Wires.\n" );
        else if( sortcabbage() == NULL )
                fprintf( stderr, "Cabbage file sort failure\n" );
        else if( writesquid() == NULL )
                fprintf( stderr, "Squid output file write failure\n" );
        else if( SQEnd() <= 0 )
                fprintf( stderr, "Couldn't SQEnd.\n" );
        exit(0);
}
```

```
#include "cabtosquid.h"

readcabbage(fp)                                              readcabbage
FILE *fp;
/*
 *  This routine reads in the elements from the CABBAGE I intermediate
 *  file, after `checkversion' has verified the version number and
 *  determined the number of elements contained within the file.
 */
{
struct lsg *ele;
long int first_byte_count, second_byte_count;
int junk, status;
short int getshortint();
long int getlongint();
        /* Scan through already allocated element list */
        for( ele = elements; ele != NULL; ele = ele->next ) {
                if( versflg == 0) {        /* This is older binary format */
                        if((first_byte_count = getlongint(fp)) != 18)
                                return(NULL);
/*
 * cabbage has 1 user unit = 1 integer, whereas Squid has 1 user unit
 * = 2 integers.  all numbers read in from the cabbage file are multiplied
 * by 2 to maintain compatibility with Squid and to prevent roundoff in integer
 * division.
 */
                        /* Read in the element's members */
                        ele->type = getshortint(fp);
                        ele->orientation = getshortint(fp);
                        ele->x_center = 2 * getshortint(fp);
                        ele->x_left_offset = 2 * getshortint(fp);
                        ele->x_right_offset = 2 * getshortint(fp);
                        ele->y_center = 2 * getshortint(fp);
                        ele->y_bottom_offset = 2 * getshortint(fp);
                        ele->y_top_offset = 2 * getshortint(fp);
                        junk = getshortint(fp);
                        second_byte_count = getlongint(fp);
                        if(first_byte_count != second_byte_count)
                                return(NULL);
                }
                else if( versflg == 1 ) {          /* The newer ASCII format */
                        /* Read in the element's members */
                        status = fscanf( fp, " %d %d %d %d %d %d %d %d %d",
                           &ele->type, &ele->orientation, &ele->x_center,
                           &ele->x_left_offset, &ele->x_right_offset,
                           &ele->y_center, &ele->y_bottom_offset,
                           &ele->y_top_offset, &junk );
                        if( status != 9 )          /* Complain if error occurs */
                                return( NULL );
                        ele->x_center *= 2 * expfactor;
                        if( ele->x_center < 0 ) {          /* => a line */
                                ele->x_left_offset *= 2 * expfactor;
                                ele->x_right_offset *= 2 * expfactor;
                        }
                        else {
                                ele->x_left_offset *= 2;
                                ele->x_right_offset *= 2;
                                /* In case of an odd size object */
                                ele->x_left_offset = ele->x_right_offset
                                   = ( ele->x_left_offset +
                                   ele->x_right_offset ) / 2;
                        }
                        ele->y_center *= 2 * expfactor;
                        if( ele->y_center < 0 ) {          /* => a line */
                                ele->y_bottom_offset *= 2 * expfactor;
                                ele->y_top_offset *= 2 * expfactor;
                        }
                        else {
                                ele->y_bottom_offset *= 2;
                                ele->y_top_offset *= 2;
                                /* In case of an odd size object */
                                ele->y_bottom_offset = ele->y_top_offset
                                   = ( ele->y_bottom_offset +
                                   ele->y_top_offset ) / 2;
                        }
                }
        }
        return(OK);
}
```

```
#include "cabtosquid.h"

sortcabbage()                                                      sortcabbage
/*
 * This routine sorts the element data read in from the CABBAE I
 * intermediate file into `points', suitable for generating
 * Squid instances, after the nets of connected objects are
 * made equal.
 */
{
struct point *tmppnt, *allocpoints();
register int errflg = FALSE;
        points = allocpoints( elements );          /* Allocate `points' */
        for( tmppnt = points; tmppnt != NULL; tmppnt = tmppnt->next ) {
                if( connet( tmppnt ) == TRUE ) { /* Connect nets at point */
                        errflg++;
                }
        }
        return(errflg);
}
```

```
#include "cabtosquid.h"

splitwires()                                                    splitwires
/*
 *  This routine is used after read in to split long wires that
 *  cross at contacts, other lines, etc.  It has a horrible
 *  n**2 sort of dependency. (What a hack!)
 */
{
struct lsg  *line;
struct lsg  *pstruct;
        /* For every element ... */
        for( line = elements; line != NULL; line = line->next ) {
                if( line->type > FRUNX )          /* ... that is a line */
                        continue;
                /* For every element ... */
                for( pstruct = elements; pstruct != NULL; pstruct =
                    pstruct->next ) {
                        /* Check to see if line intersects line */
                        if( pstruct->type <= FRUNX ) {
                                /* If the lines are not the same type ... */
                                if( line->type != pstruct->type )
                                        continue;         /* Ignore the line */
                                /* If the line is vertical */
                                if( pstruct->orientation == VERTICAL ) {
                                        /* Check the bottom endpoint */
                                        split1wire(line, pstruct->x_center,
                                            pstruct->y_bottom_offset );
                                        /* Check the top endpoint */
                                        split1wire(line, pstruct->x_center,
                                            pstruct->y_top_offset );
                                }
                                else {  /* => the line is horizontal */
                                        /* Check the left endpoint */
                                        split1wire(line, pstruct->x_left_offset,
                                            pstruct->y_center );
                                        /* Check the right endpoint */
                                        split1wire(line,pstruct->x_right_offset,
                                            pstruct->y_center );
                                }
                        }
                        /* Ignore metal lines crossing transistors */
                        else if( line->type == METAL && (pstruct->type ==
                            TRAN || pstruct->type == LOAD || pstruct->type
                            == BUR_CNT ) )
                                continue;
                        /* Ignore poly lines crossing diff-metal contacts */
                        else if(line->type == POLY && pstruct->type == D_M_CNT)
                                continue;
                        /* Ignore diff lines crossing poly-metal contacts */
                        else if(line->type == DIFF && pstruct->type == P_M_CNT)
                                continue;
                        else    /* See if the point structure splits the line */
                                split1wire( line, pstruct->x_center,
                                    pstruct->y_center );
                }
        }
        return(OK);
}

split1wire( line, x_center, y_center )                          split1wire
struct lsg  *line;
int x_center, y_center;
```

```
/*
 * This routine handles splitting one line at x_center, y_center.
 * It makes sure the point actually splits the line.
 */
{
struct lsg *tmpline, *P_getlsg();
        if( line->orientation == HORIZONTAL ) {
                if( y_center == line->y_center && x_center > line->x_left_offset
                    && x_center < line->x_right_offset ) {
                        tmpline = P_getlsg();
                        *tmpline = *line;
                        line->next = tmpline;
                        line->x_right_offset = tmpline->x_left_offset =
                            x_center;
                }
        }
        else {  /* => Vertical Orientation */
                if( x_center == line->x_center && y_center >
                    line->y_bottom_offset && y_center < line->y_top_offset ) {
                        tmpline = P_getlsg();
                        *tmpline = *line;
                        line->next = tmpline;
                        line->y_top_offset = tmpline->y_bottom_offset =
                            y_center;
                }
        }
}
```

```
#include "cabtosquid.h"
#include "sq.h"

long int getlongint(fp)
FILE *fp;
{
unsigned int byte1, byte2, byte3, byte4;
long int value;
/*
 * this function reads a 32-bit integer from a file where it is stored
*   in the format:
*                   MSB:   byte4;byte3;byte2;byte1   :LSB
 */
        byte1 = getc(fp);
        byte2 = getc(fp);
        byte3 = getc(fp);
        byte4 = getc(fp);
        value = (byte4 << 24) | (byte3 << 16) | (byte2 << 8) | byte1;
        return(value);
}
short int getshortint(fp)
FILE *fp;
{
unsigned int byte1, byte2;
short int value;
/*
 * this function reads a 16-bit integer from file fp.  it is used
 * to maintain portability between machines of different word size
 * and uses getc to do this.
 */
        byte1 = getc(fp);
        byte2 = getc(fp);
        value = (byte2 << 8) | byte1;
        return(value);
}
char *
strsav( string )                                                            strsav
char *string;
/*
 * This routine calls the virtual memory allocator to save a string.
 */
{
char *newstring, *malloc(), *strcpy();
        /* Get a pointer to the new string */
        newstring = malloc( (unsigned)(strlen( string ) + 1) );
        /* Copy the string into the new string */
        strcpy( newstring, string );
        /* Return the pointer to the new string */
        return( newstring );
}
```

```
#include "cabtosquid.h"
#include "sq.h"

writesquid()                                                   writesquid
/*
 *  This routine writes out the squid for the CABBAGE I cell.
 *  It instanciates primitives which have had master cells
 *  created for them previously.
 */
{
SQView view;
FILE *stream;
struct nethd *tmpnet;
SQNet sqnet;
SQTerm term;
struct point *tmppnt, *tmppnt1;
SQInst inst;
char cif[20];
char trmnam[20];
SQGeo geo;
SQIntegerPoint path[2];
SQProp ishorizontal, isvertical;
        /* Initialize some local variables */
        ishorizontal.name = "ishorizontal";
        ishorizontal.valueType = sqBool;
        ishorizontal.value.bool = sqTrue;
        isvertical.name = "ishorizontal";
        isvertical.valueType = sqBool;
        isvertical.value.bool = sqFalse;
        /* Prepare the master view of the top level instance */
        view.cell = "c2s.out";
        view.view = "layout";
        view.mode = "w";
        /* If the view cannot be created, assume it already exists ... */
        if( SQ(sqCreate, sqView, view, &stream) <= 0 ) {
                /* ... and open the already existing view ... */
                if( SQ(sqOpen, sqView, view, &stream) <= 0 ) {
                        fprintf( stderr, "Can't open cell %s.\n", view.cell );
                        exit(25);
                }
                /* ... and delete it. */
                SQ(sqRm, sqView);
                /* If the view still cannot be created ... */
                if( SQ(sqCreate, sqView, view, &stream) <= 0 ) {
                        /* Holler like mad! */
                        fprintf( stderr, "Can't create cell %s.\n", view.cell );
                        exit(26);
                }
        }
        sqnet.name = "";           /* Make sure no name is specified */
        /* And create nets for all of the top level nets in the circuit */
        for( tmpnet = nethead.next; tmpnet != NULL; tmpnet = tmpnet->next ) {
                if(SQ(sqCreate, sqNet, &sqnet) <= 0 ) {
                        fprintf( stderr, "Can't generate net!\n" );
                        exit(27);
                }
                tmpnet->net = sqnet.netID;
        }
        /* For each `point' data structure ... */
        for( tmppnt = points; tmppnt != NULL; tmppnt = tmppnt->next ) {
                /* For each element at this coordinate ... */
                for( tmppnt1 = tmppnt; tmppnt1 != NULL; tmppnt1 =
                     tmppnt1->nextatpoint ) {
```

...writesquid

```
/* If the element is a terminal ... */
if( tmppnt1->element->type == TERM ) {
        term.instID = NULL;
        term.netID = tmppnt1->netid->net;
        /* Generate a local terminal name */
        sprintf( trmnam, "T%dX%dY",
            tmppnt1->element->x_center,
            tmppnt1->element->y_center );
        term.name = trmnam;
        if(SQ(sqCreate,sqTerm,term) <= 0) {
                fprintf(stderr, "Bad term creation.\n");
                exit( 100 );
        }
        geo.layer = tmppnt1->element->name;
        geo.manhattanP = sqTrue;
        geo.geoType = sqRect;
        geo.def.rect.l = tmppnt1->element->x_center -
            tmppnt1->element->x_left_offset;
        geo.def.rect.b = tmppnt1->element->y_center -
            tmppnt1->element->y_bottom_offset;
        geo.def.rect.r = tmppnt1->element->x_center +
            tmppnt1->element->x_right_offset;
        geo.def.rect.t = tmppnt1->element->y_center +
            tmppnt1->element->y_top_offset;
        geo.function = sqTermArea;
        geo.implements.term = term.name;
        if(SQ(sqCreate, sqGeo, &geo) <= 0 ) {
                fprintf( stderr, "No geometry!\n" );
                exit(128);
        }
}
/* Else if the element is not a line ... */
else if( tmppnt1->element->type > FRUNX ) {
        /* Generate an instance for it */
        inst.name = tmppnt1->element->name;
        if( inst.name == NULL ) {
                continue;
        }
        inst.masterCell = tmppnt1->element->name;
        inst.masterView = "layout";
        /* Key transformation off of orientation */
        if( tmppnt1->element->orientation ==
            HORIZONTAL ) {
                sprintf( cif, "T %d %d", tmppnt1->x,
                    tmppnt1->y );
        }
        else {
                sprintf( cif, "R 0 -1 T %d %d",
                    tmppnt1->x, tmppnt1->y );
        }
        inst.cif = cif;
        if( !tmppnt1->element->instanciated ) {
                tmppnt1->element->instanciated++;
                if( SQ(sqCreate, sqInst, &inst) <= 0 ) {
                        fprintf( stderr, "No inst.\n" );
                }
                tmppnt1->element->instid = inst.instID;
        }
        term.instID = tmppnt1->element->instid;
        if( tmppnt1->netid != NULL ) {
                term.netID = tmppnt1->netid->net;
        }
        else {
                term.netID = NULL;
        }
        /* Bind the formal and actual terminals */
        switch( tmppnt1->type ) {
                case POINT:
                        term.name = "i";
                        if( SQ(sqUpdate,sqTerm,term) <=
                            0 ) {
                                fprintf( stderr,
                                    "No term.\n" );
                        }
                        break;
                case DRAIN:
                        term.name = "d";
                        if( SQ(sqUpdate,sqTerm,term) <=
                            0 ) {
                                fprintf( stderr,
                                    "No term.\n" );
                        }
                        break;
                case GATE:
                        term.name = "g";
                        if( SQ(sqUpdate,sqTerm,term) <=
                            0 ) {
                                fprintf( stderr,
                                    "No term.\n" );
                        }
                        break;
                case SOURCE:
                        term.name = "s";
                        if( SQ(sqUpdate,sqTerm,term) <=
                            0 ) {
                                fprintf( stderr,
                                    "No term.\n" );
                        }
                        break;
        }
}
/* Else, if the element is a line ... */
else if( tmppnt1->type == ULINE ) {
        /* Switch on the mask layer */
        switch( tmppnt1->element->type % 4 ) {
                case DIFF:
                        geo.layer = "ND";
                        break;
                case POLY:
                        geo.layer = "NP";
                        break;
                case METAL:
                        geo.layer = "NM";
                        break;
        }
        /* Create a geometry for the line */
        geo.manhattanP = sqTrue;
        geo.geoType = sqLine;
        if(tmppnt1->element->orientation == VERTICAL) {
                /* Don't output 0 length lines */
                if( tmppnt1->element->y_bottom_offset ==
                    tmppnt1->element->y_top_offset )
                        continue;
                path[0].x =
                    tmppnt1->element->x_center;
                path[0].y =
                    tmppnt1->element->y_bottom_offset;
                path[1].x =
                                            tmppnt1->element->x_center;
                                        path[1].y =
                                            tmppnt1->element->y_top_offset;
                                        geo.def.line.width =
                                            tmppnt1->element->x_right_offset +
                                            tmppnt1->element->x_left_offset;
                                }
                                else {
                                        /* Don't output 0 length lines */
                                        if( tmppnt1->element->x_left_offset ==
                                            tmppnt1->element->x_right_offset )
                                                continue;
                                        path[0].x =
                                            tmppnt1->element->x_left_offset;
                                        path[0].y =
                                            tmppnt1->element->y_center;
                                        path[1].x =
                                            tmppnt1->element->x_right_offset;
                                        path[1].y =
                                            tmppnt1->element->y_center;
                                        geo.def.line.width =
                                            tmppnt1->element->y_top_offset +
                                            tmppnt1->element->y_bottom_offset;
                                        geo.prop = ishorizontal;
                                }
                                geo.def.line.path = path;
                                geo.def.line.nPath = 2;
                                geo.function = sqInterconnect;
                                geo.implements.net = tmppnt1->netid->net;
                                if(SQ(sqCreate, sqGeo, &geo) <= 0 ) {
                                        fprintf( stderr, "No geometry!\n" );
                                        exit(28);
                                }
                                if(tmppnt1->element->orientation == VERTICAL) {
                                        geo.prop = isvertical;
                                }
                                else {
                                        geo.prop = ishorizontal;
                                }
                                if(SQ(sqPutProp, sqGeo, geo) <= 0 ) {
                                        fprintf( stderr, "No prop.\n" );
                                        exit(29);
                                }
                        }
                }
        }
        /* Save the created view */
        if( SQ(sqSave, sqView) <= 0 ) {
                fprintf( stderr, "Couldn't save cell/view %s/%s.\n",
                    view.cell, view.view );
        }
        return(OK);
}
```

```
/*
Public types for Squid DBMS.

Copyright Ken Keller 1981
*/

#define SQMAXLAYERS 20
#define SQMAXDEPTH 100

typedef int SQStatus;
#define SQOUTOFVM -1
#define SQUNKNOWNLAYER -2
#define SQENDGEN -3
#define SQUNKNOWNCURRENTVIEW -4
#define SQTOOMANYLAYERS -5
#define SQUNKNOWNDEMON -6
#define SQUNKNOWNOPERATION -7
#define SQUNKNOWNOBJECT -8
#define SQHIERARCHYISTOODEEP -10
#define SQRECURSIVEHIERARCHY -11
#define SQCANNOTPARSETRANSFORMATION -20
#define SQNONMANHATTANTRANSFORMATION -21
#define SQUNTYPEDVALUE -30
#define SQUNKNOWNPARM -31
#define SQUNKNOWNTERM -40
#define SQUNKNOWNPROP -41
#define SQUNTYPEDGEO -50
#define SQCANNOTCREATEVIEW -62
#define SQCANNOTCREATECELL -63
#define SQCELLDOESNOTEXIST -64
#define SQCANNOTOPENVIEW -65
#define SQVIEWEXISTS -67
#define SQVIEWDOESNOTEXIST -68
#define SQNOTAVIEW -69
#define SQCORRUPTVIEW -70
#define SQCANNOTSAVEVIEW -71
#define SQCANNOTRMVIEW -72
#define SQTRIVIALGEN -73
#define SQCANNOTCPVIEW -74
#define SQDEGENERATEPATH -75

typedef enum {sqGeo,sqTerm,sqNet,sqView,sqInst,sqParm} SQObjectType;

typedef enum {sqCreate,sqUpdate,sqGet,sqDelete,sqBeginGen,sqGen,
  sqBeginPropGen,sqGenProp,sqPutProp,sqGetProp,sqRmProp,
  sqSave,sqOpen,sqClose,sqRm,sqCp} SQOperationType;

typedef enum {sqFalse,sqTrue} SQBool;

typedef struct SQBB SQBB;
struct SQBB {
  int l,b,r,t; };

typedef struct SQRealPoint SQRealPoint;
struct SQRealPoint {
  float x,y; };

typedef struct SQIntegerPoint SQIntegerPoint;
struct SQIntegerPoint {
  int x,y; };

typedef enum {sqInteger,sqReal,sqString,sqBool} SQValueType;

typedef struct SQParm SQParm;
```

```
struct SQParm {
  char *name;
  int instID;
  SQValueType valueType;
  union {
    int integer;
    float real;
    char *string;
    SQBool bool; }
  value; };

typedef struct SQProp SQProp;
struct SQProp {
  char *name;
  SQValueType valueType;
  union {
    int integer;
    float real;
    char *string;
    SQBool bool; }
  value; };

typedef struct SQView SQView;
struct SQView {
  char *cell, *view, *mode;
  SQBB bb;
  SQProp prop; };

typedef enum {sqFrame,sqActiveArea,sqInterconnect,sqTermArea} SQFunction;

typedef enum {sqPlot,sqRect,sqLine,sqPolygon,sqCircle,sqLabel} SQGeoType;

typedef struct SQGeo SQGeo;
struct SQGeo {
  char *layer;
  SQBB bb;
  SQBool manhattanP,filledP;
  int geoID;
  SQGeoType geoType;
  union {
    SQBB rect;
    struct {
      int nPath;
      SQRealPoint *path; }
    plot;
    struct {
      int nPath;
      SQIntegerPoint *path; }
    polygon;
    struct {
      int width;
      int nPath;
      SQIntegerPoint *path; }
    line;
    struct {
      SQIntegerPoint *center;
      SQIntegerPoint *beginAngle, *endAngle;
      SQIntegerPoint *innerRadius, *outerRadius; }
    circle;
    struct {
      SQIntegerPoint position;
      int height;
      int angle;
      char *justification;
      char *label;
      char *font; }
    label; }
  def;
  SQFunction function;
  union {
    char *term;
    int net; }
  implements;
  SQProp prop; };

typedef struct SQTerm SQTerm;
struct SQTerm {
  int instID;
  char *name;
  int netID;
  SQProp prop; };

typedef struct SQNet SQNet;
struct SQNet {
  char *name;
  int netID;
  SQProp prop; };

typedef struct SQInst SQInst;
struct SQInst {
  char *name;
  char *masterCell, *masterView;
  char *cif;
  int matrix[3][3];
  int instID;
  SQBB bb;
  SQProp prop;
  SQFunction function;
  union {
    char *term;
    int net; }
  implements; };

extern SQStatus
  SQ(),
  SQAttachDemon(),
  SQDetachDemon(),
  SQSpecialGen(),
  SQGenNetTerm(),
  SQBeginLayerGen(),
  SQGenLayer(),
  SQEnd(),
  SQCurrentView(),
  SQBegin(),
  SQPopSpecialGen();
extern int SQSpecialBeginGen(),
  SQBeginNetTermGen(),
  SQLayerNameToNumber();
extern char *SQLayerNumberToName();
```

# APPENDIX H

## Frame Program Listing

This appendix contains the manual entry and program listing for the *frame* program, which generates protection frames for *Squid* cells from the geometry contained within the cell. It uses the *Fang* Manhattan polygon package [Moore82] to generate the frames.

## NAME

frame – Generates protection frames for squid cells.

## SYNOPSIS

**frame** [-i <inview>] [-o <outview>] [-e <errfil>] [-a#] cell ...

## DESCRIPTION

*Frame* generates protection frames for symbolic IC cells stored in the *squid* database. The *fang* Manhattan polygon package is used to generate the frames with a grow/merge/shrink algorithm. This algorithm take the boxes which comprise the geometries contained within a cell ( polygons are factored into boxes ), and expands each box about its center. The individual boxes are merged together, and the resulting polygon(s) are shrunk by the expansion factor to obtain a set of protection frames for the cell. The operations are performed on a per/layer basis, so there is a set of protection frames for each mask layer.

By adjusting the magnitude of the grow/shrink factor, tradeoffs can be made in the complexity of the protection frames. A zero grow/shrink factor would have the effect of simply merging the geometries on each mask layer. While this would allow all of the unused area within the cell to be used for routing at greater levels in the design hierarchy, the number of boxes required to represent these complex frames is almost as great as in the original cell itself. An infinite grow/shrink factor, on the other hand, would remove all interior 'holes' from the cell, leaving a set of bounding polygons for the protection frames. These polygons would require fewer boxes to represent, but would not allow routing in areas that are actually unused within the cell.

The options available with *frame* are:

-i <inview>    Change the *squid* input view from the default 'layout' to '<inview>'.

-o <outview>
    Change the *squid* output view from the default 'framed' to '<outview>'.

-e <errfil>    Change the error reporting file from the default 'framerr' to '<errfil>'.

-a#    Change the amount of the grow/shrink from the default amount of 200 to number #.

## FILES

| | |
|---|---|
| <cell>/layout | – Default input view |
| <cell>/framed | – Default output view |

## SEE ALSO

hawk(cad), python(cad), fang(3cad), squid(3cad)

## AUTHOR

| | |
|---|---|
| Mark Bales | (Supervisory code) |
| Ken Keller | (Squid DBMS) |
| Peter Moore | (Fang polygon package) |

## DIAGNOSTICS

Error messages are self explanatory and detail errors encountered in the *squid* database procedural interface.

## BUGS

The objects used to define the protection frames are the terminals local to the cell, the interconnect local to the cell, and the protection frames of the instances contained within the cell. Local geometries are **NOT** included, so there may be some problems in generating protection frames for leaf cells (cells which contain no instances).

The grow/merge/shrink algorithm for generating the protection frames sometimes removes large unused area which should be left unprotected.

```
/* Copyright -C- 1982 Mark W. Bales  All Rights Reserved */
#include "sq.h"
#include "fang.h"
#include <setjmp.h>
#include <stdio.h>

/* Typedefs */
typedef enum { FOK, FBADSQOPEN, FBADSQGET, FBADSQGEN,
     FBADSQPROP, FBADSQLINE, FBADSQTERM, FNOLINEORIENT,
     FBADSQFRAME, FLEAFCELL, FBADSQSAVE, FBADSQCOPY, FBADSQDEL,
     FBADSQUPDATE, F_FA_FRAME, F_FA_TO_BOX, F_FA_ADD_BOX,
     FBADSQCREATE } fstat;

/****** #defines ******/
#define EVER ;;
#define TRUE 1
#define FALSE 0
#define FATAL 0
#define NONFATAL 1

/* Some macro definitions */
#define F_max(a,b) ((a) > (b) ? (a) : (b))
#define F_min(a,b) ((a) < (b) ? (a) : (b))
/*
 * ALIGN - compute smallest number >= x which is exactly
 *  divisible by size.
 */
#define  ALIGN(x, size)    ( (x) % (size) ?  (x) + (size) - (x) % (size) : (x) )
#define malloc P_vmalloc

extern fa_geometry F_ingeo[SQMAXLAYERS];
extern fa_geometry F_outgeo[SQMAXLAYERS];
extern fa_box_list F_outbox[SQMAXLAYERS];
extern char *F_inview;
extern char *F_outview;
extern char *F_erfilnam;
extern char *F_version;
extern FILE *F_errfil;
extern fa_coord F_amount;
extern jmp_buf F_errorenv;
```

```
fa_geometry F_ingeo[SQMAXLAYERS];
fa_geometry F_outgeo[SQMAXLAYERS];
fa_box_list F_outbox[SQMAXLAYERS];
char *F_inview = "layout";
char *F_outview = "framed";
char *F_erfilnam = NULL;
char *F_version = "Frames version 0.0";
FILE *F_errfil;
fa_coord F_amount = 200;
jmp_buf F_errorenv;
```

```
/* Copyright -C- 1982 Mark W. Bales  All Rights Reserved */
#include "frame.h"

main( argc, argv )                                                                main
int argc;
char *argv[];
/*
 *  This routine is the executive for using frames as a standalone
 *  program.  It allows a user to generate protection frames for
 *  many cells, changing the input and/or output views for each cell.
 */
{
char *s;
FILE *fopen();

        if( SQBegin() <= 0 )        /* Initialize the squid database */
                exit(100);
        while( --argc > 0 ) {
                if( (*++argv)[0] == '-' && (*argv)[0] != '\0' ) {
                        for( s = argv[0] + 1; *s != '\0'; s++ ) {
                                switch( *s) {

                                case 'i':         /* Change `inview' */
                                        F_inview = *++argv;
                                        argc--;
#ifdef RDEBUG

                                        printf( "Changing to input view `%s'\n",
                                            F_inview );

#endif

                                        break;

                                case 'o':         /* Change `outview' */
                                        F_outview = *++argv;
                                        argc--;
#ifdef RDEBUG

                                        printf( "Changing to out view `%s'\n",
                                            F_outview );

#endif

                                        break;

                                case 'e':         /* Change F_errfil */
                                        F_erfilnam = *++argv;
                                        argc--;
#ifdef RDEBUG

                                        printf( "Changing to error file `%s'\n",
                                            F_erfilnam );

#endif

                                        if((F_errfil = fopen(F_erfilnam,
                                            "w")) == NULL ) {
                                                fprintf( stderr,
                                                    "Couldn't open %s!\n",
                                                    F_erfilnam );
                                        }
                                        break;

                                case 'a':
                                        F_amount = atoi( ++s );
                                        if( F_amount < 0 )
                                                F_amount = 0;
                                        *s-- = '\0';
#ifdef RDEBUG

                                        printf( "Amount of grow/shrink %d.\n",
                                            F_amount );

#endif
```

```
                                        break;

                                default:
                                        fprintf( stderr,
                                            "%s: illegal option: %c\n",
                                                F_version, *s );
                                        argc = -1;
                                        break;
                                }
                        }
                }
                else {
                        if( F_errfil == NULL ) {
                                if( (F_errfil = fopen("framerr", "w")) ==
                                    NULL ) {
                                        fprintf( stderr,
                                "Couldn't open error file framerr!\n");
                                        F_errfil = stderr;
                                        F_erfilnam = "> stderr";
                                }
                                else {
                                        F_erfilnam = "framerr";
                                }
                        }
                        if( F_frame( *argv, F_inview, F_outview, F_amount )
                            != (int)FOK )
                                fprintf( stderr,
    "frame: Error occurred in file %s.  See file %s for details.\n",
                                    *argv, F_erfilnam );
                }
        }
        if(argc < 0) {                  /* => an illegal option was specified */
                fprintf(stderr,
    "usage: frame [-i inview] [-o outview] [-e errfil] file ...\n");
        }
        if( SQEnd() <= 0 )              /* Wrap up the squid database */
                exit(200);
        exit(0);
}
```

```
#include "frame.h"
#include "private.h"

F_frame( cell, inview, outview, amount )                        F_frame
char *cell, *inview, *outview;
fa_coord amount;
{
register int layer;
register int status;
        /* Set up for `longjmp' from down inside hierarchy */
        if( (status = setjmp( F_errorenv )) != 0 ) {       /* Error occurred! */
                return( status );
        }
        /* Initialize all of the input and output geometries */
        for( layer = 0; layer < SQMAXLAYERS; layer++ ) {
                fa_init( &(F_ingeo[layer]) );
                fa_init( &(F_outgeo[layer]) );
        }
        /* Read in squid view and generate fa_geometries */
        F_readin( cell, inview, outview );
        /* Generate box lists for merged frames */
        for( layer = 0; layer < SQMAXLAYERS; layer++ ) {
                /* Temporary hack */
                if( F_ingeo[layer].count <= 0 )
                        continue;
                if( fa_frame( F_ingeo[layer], amount, &(F_outgeo[layer]) ) !=
                    FA_OK ) {
                        F_error( FATAL, F_FA_FRAME, __FILE__, __LINE__,
                            "FA_ERROR: %s\n", fa_err_string );
                }
                if( fa_to_box( F_outgeo[layer], &(F_outbox[layer]) ) != FA_OK ){
                        F_error( FATAL, F_FA_TO_BOX, __FILE__, __LINE__,
                            "FA_ERROR: %s\n", fa_err_string );
                }
        }
        F_update( cell, outview );
        return( (int)FOK );
}

/* VARARGS5 */
F_error( fatalflag, errornum, filename, linenum, format, arguments )     F_error
int fatalflag;
fstat errornum;
char *filename;
int linenum;
char *format;
/*
 * This routine takes an error number, a filename, and a line number
 * and prints out an appropriate error message.  If no message
 * exists for a particular number, the number alone is printed.
 */
{
#ifndef HAWK
        fprintf( F_errfil, "%s(line %d): ", filename, linenum );
        _doprnt( format, &arguments, F_errfil );
#else
char errbuf[BUFSIZ];
register int i;
struct _iobuf _strbuf;

        sprintf( errbuf, "%s(line %d): ", filename, linenum );
        i = strlen( errbuf );
        _strbuf._flag = _IOWRT+_IOSTRG;
        _strbuf._ptr = &(errbuf[i]);
```

```
		_strbuf._cnt = BUFSIZ;
		_doprnt( format, &arguments, &_strbuf);
		putc('\0', &_strbuf);
#endif
		if( fatalflag ) {
			longjmp( F_errorenv, (int)errornum );
		}
}
```

```
/* Copyright -C- 1982 Mark W. Bales  All Rights Reserved */
/* LINTLIBRARY */
#include "frame.h"
#include <ctype.h>

F_readin( cell, inview, outview )
char *cell, *inview, *outview;
/*
 * This routine is responsible for reading the data from the
 * Squid database and generating the fa_geometries for fang.
 * The special generator in Squid is run twice, the first
 * time to get the local interconnect and the second time to
 * get instance protection frames, local terminals, and
 * terminals contained within instances.
 */
{
FILE *cellfil, *instfil;
SQStatus cstat, SQ();
SQView cellview, instview;
int cellgen, i;
int mask[4*SQMAXLAYERS+1][2];
int linepath[2], instids[2], level;
SQGeo geo;
SQInst inst;
SQStatus status;
#ifdef RDEBUG
        printf( "Reading cell %s, with inview %s and outview %s\n",
            cell, inview, outview );
#endif
        /* Initialize the number of instances to zero. */
        /* Set up for readin the `cell' with `inview' */
        cellview.mode = "r";
        cellview.cell = cell;
        cellview.view = inview;
        /* KLUDGE (necessary since Squid may not have proper bounding box) */
        cellview.bb.l = -1000000000;
        cellview.bb.b = -1000000000;
        cellview.bb.r = 1000000000;
        cellview.bb.t = 1000000000;
        /* Open the cell with current view `inview' */
        if((status = SQ(sqOpen, sqView, cellview, &cellfil)) <= 0) {
                F_error( FATAL, FBADSQOPEN, __FILE__, __LINE__,
                    "Couldn't open cell %s inview %s. Status = %d.\n",
                    cell, inview, status );
        }
        /* Copy the `inview' to `outview' */
        cellview.view = outview;
        if( SQ(sqCp, sqView, cellview) <= 0 ) {      /* If the copy fails */
                /* Open the `outview' */
                if((status = SQ(sqOpen, sqView, cellview, &cellfil)) <= 0) {
                        F_error( FATAL, FBADSQOPEN, __FILE__, __LINE__,
                            "Can't open cell %s to rm outview %s. Status = %d\n",
                            cell, outview, status );
                }
                /* And remove it */
                if((status = SQ(sqRm, sqView, cellview)) <= 0) {
                        F_error( FATAL, FBADSQCOPY, __FILE__, __LINE__,
                            "Can't remove cell %s view %s. Status = %d.\n",
                            cell, outview, status );
                }
                /* Set up to open the `inview' */
                cellview.view = inview;
```

*F_readin*

...F_readin

```
                /* Open the cell with current view `inview' */
                if((status = SQ(sqOpen, sqView, cellview, &cellfil)) <= 0) {
                        F_error( FATAL, FBADSQOPEN, __FILE__, __LINE__,
                        "Can't open cell %s view %s after rm. Status = %d.\n",
                            cell, inview, status );
                }
                /* Set up to open the `outview' */
                cellview.view = outview;
                /* And copy the `inview' to the `outview' */
                if((status = SQ(sqCp, sqView, cellview)) <= 0) {
                        F_error( FATAL, FBADSQCOPY, __FILE__, __LINE__,
                        "Can't copy cell %s view %s to view %s. Status = %d.\n",
                            cell, inview, outview, status );
                }
        }
        /* Open the cell with current view `outview' */
        cellview.mode = "w";        /* Necessary for updating later on */
        if((status = SQ(sqOpen, sqView, cellview, &cellfil)) <= 0) {
                F_error( FATAL, FBADSQOPEN, __FILE__, __LINE__,
                    "Can't open cell %s outview %s. Status = %d.\n",
                    cell, outview, status );
        }
#ifdef RDEBUG
        printf( "Opened Cell Froperly.\n" );
#endif
        /* Open all of the masters of the instances in `cell' */
        if((status = SQ(sqBeginGen, sqInst, &cellgen)) <= 0) {
                F_error( FATAL, FBADSQGEN, __FILE__, __LINE__,
                    "Can't Begin Instance Generator.  Status = %d.\n",
                    status );
        }
        /* Read in to VM all of the instances in the current view */
        while((cstat = SQ(sqGen, sqInst, cellgen, &inst )) > 0) {
                /* Fill in the holes in the instance */
                inst.cif = NULL;
                if((status = SQ(sqGet, sqInst, &inst)) <= 0) {
                        F_error( FATAL, FBADSQGET, __FILE__, __LINE__,
                            "Can't get instance ID = %d. Status = %d.\n",
                            inst.instID, status );
                }
                /* Fill in the instance `view' structure */
                instview.mode = "r";
                instview.cell = inst.masterCell;
                instview.view = inst.masterView;
#ifdef RDEBUG
                printf( "Opening master cell %s with view %s in instance.\n",
                    instview.cell, instview.view );
#endif
                /* Open the master cell of the instance in `cell' */
                if((status = SQ(sqOpen, sqView, instview, &instfil)) <= 0) {
                        F_error( FATAL, FBADSQOPEN, __FILE__, __LINE__,
                        "Can't open instance master %s view %s. Status = %d.\n",
                            inst.masterCell, inst.masterView, status );
                }
        }
        /* Check to make sure that exit status was `SQENDGEN' */
        if(cstat != SQENDGEN) {
                F_error( FATAL, FBADSQGEN, __FILE__, __LINE__,
                    "Squid returned bad status %d from instance generator.\n",
                    cstat );
        }
        /* Open the cell again to insure the `outview' is `current' */
        if((status = SQ(sqOpen, sqView, cellview, &cellfil)) <= 0) {
                F_error( FATAL, FBADSQOPEN, __FILE__, __LINE__,
                    "Can't reopen cell %s view %s after instgen. Status = %d.\n",
                     cell, outview, status );
        }
        /* Set up mask for special generator */
        for( i = 0; i < SQMAXLAYERS; i++ ) {
                mask[i][0] = i;
                mask[i][1] = (int)sqInterconnect;
        }
        mask[SQMAXLAYERS][0] = -1;
#ifdef RDEBUG
        printf( "Looking for local interconnect.\n" );
#endif
        /* Begin a generator to retrieve all interconnect local to the cell */
        if((status = SQSpecialBeginGen(cellview.bb, mask, 1, &cellgen)) <= 0) {
                F_error( FATAL, FBADSQGEN, __FILE__, __LINE__,
                     "Can't begin interconnect generator. Status = %d.\n",
                     status );
        }
#ifdef RDEBUG
        printf( "Generator started properly.\n" );
#endif
        level = 1;
        /* Read all of the interconnect local to the cell */
        while((cstat = SQSpecialGen(cellgen, &geo, linepath, 2, NULL, 0,
            instids, &level )) > 0) {
                /* LATER - Check to insure geo is Manhattan */
/*                ASSERT(geo.manhattanP, FNONMANHATTAN);*/
                /* Make sure the function is interconnect */
                if(geo.function != sqInterconnect) {
                        F_error( NONFATAL, FBADSQGEN, __FILE__, __LINE__,
                        "Intcon. local to cell %s view %s has function %d.\n",
                             cell, outview, geo.function );
                        continue;
                }
                /* Add the line */
                F_addline( &geo );
                level = 1;
        }
        /* Check to make sure that exit status was `SQENDGEN' */
        if(cstat != SQENDGEN) {
                F_error( FATAL, FBADSQGEN, __FILE__, __LINE__,
                     "Squid returned bad status %d from instance generator.\n",
                     cstat );
        }
        /* Set up mask for special generator */
        for( i = 0; i < SQMAXLAYERS; i++ ) {
                mask[i*2][0] = i;
                mask[i*2][1] = (int)sqFrame;
                mask[i*2+1][0] = i;
                mask[i*2+1][1] = (int)sqTermArea;
        }
        mask[2*SQMAXLAYERS][0] = -1;
#ifdef RDEBUG
        printf( "Looking for terminals and frames.\n" );
#endif
        /* Begin a generator to retrieve all terminals and frames */
        if((status = SQSpecialBeginGen(cellview.bb, mask, 2, &cellgen)) <= 0) {
                F_error( FATAL, FBADSQGEN, __FILE__, __LINE__,
                     "Can't begin frame and terminal generator. Status = %d.\n",
                     status );
        }
```

```
#ifdef RDEBUG
        printf( "Generator started properly.\n" );
#endif
        level = 2;
        /* Read all of the local terminals, and all of the terminals and
         * protection frames of the instances. */
        while((cstat = SQSpecialGen(cellgen, &geo, linepath, 2, NULL, 0,
            instids, &level )) > 0 ) {
                /* Add a frame if the function is `frame' */
                if( geo.function == sqFrame ) {
                        F_addframe( &geo, instids, level );
                }
                else {
                        F_addterm( &geo, instids, level );
                }
                level = 2;
        }
        /* Check to make sure that exit status was `SQENDGEN' */
        if(cstat != SQENDGEN) {
                F_error( FATAL, FBADSQGEN, __FILE__, __LINE__,
                    "Squid returned bad status %d from instance generator.\n",
                    cstat );
        }
}
```

F_addline

```
F_addline( geo )
SQGeo *geo;
/*
 *  This routine adds a line to the Fang data structures which was
 *  obtained from the Squid database.
 */
{
register int layer;
SQBB bb;
SQIntegerPoint *point1, *point2;
SQStatus status;
        /* All lines must be geometries with type `sqLine' */
        if(geo->geoType != sqLine) {
                F_error( NONFATAL, FBADSQLINE, __FILE__, __LINE__,
                    "Line on layer %s has type %d not `sqLine'. Ignored.\n",
                    geo->layer, geo->geoType );
                return;
        }
        layer = SQLayerNameToNumber( geo->layer );
#ifdef RDEBUG
        printf( "Starting a new line layer = %s.\n", geo->layer );
#endif
        /* This is how the orientation of lines is communicated */
        geo->prop.name = "ishorizontal";
        /* All lines must have this property to work with Fang */
        if((status = SQ(sqGetProp, sqGeo, geo)) <= 0) {
                F_error( FATAL, FNOLINEORIENT, __FILE__, __LINE__,
"Line: mask %s (%d,%d) to (%d,%d) No property `ishorizontal'. Status = %d.\n",
                  geo->layer, geo->def.line.path[0].x, geo->def.line.path[0].y,
                  geo->def.line.path[1].x, geo->def.line.path[1].y, status );
        }
        /* The property must be of type boolean */
        if(geo->prop.valueType != sqBool) {
                F_error( FATAL, FBADSQPROP, __FILE__, __LINE__,
"Line: mask %s (%d,%d) to (%d,%d) Bad property `ishorizontal'.\n",
                  geo->layer, geo->def.line.path[0].x, geo->def.line.path[0].y,
                  geo->def.line.path[1].x, geo->def.line.path[1].y );
        }
```

...F_addline

```
	/* Get the two endpoints of the line from the path */
	point1 = geo->def.line.path;
	point2 = &point1[1];
	bb.l = F_min( point1->x, point2->x );
	bb.r = F_max( point1->x, point2->x );
	bb.b = F_min( point1->y, point2->y );
	bb.t = F_max( point1->y, point2->y );
	/* Augment the coordinates of the line by the line width */
	if( geo->prop.value.bool ) {		/* => HORIZONTAL line */
		bb.b -= geo->def.line.width / 2;
		bb.t += geo->def.line.width / 2;
	}
	else {					/* => VERTICAL line */
		bb.l -= geo->def.line.width / 2;
		bb.r += geo->def.line.width / 2;
	}
	/* Add this box to the geometry */
	fa_add_box( &(F_ingeo[layer]), bb.l, bb.b, bb.r, bb.t );
}
```

F_addframe

```
F_addframe( geo, instids, level )
SQGeo *geo;
int instids[];
int level;
/*
 * This routine adds in protection frames (boxes) for the instances
 * in the cell.  Level 0 (local) frames are ignored.  Instance ids
 * are watched and the routine allocates a new instance structure
 * when the instance ID changes from its previous value.
 */
{
register int layer;
char *SQLayerNumberToName();
SQBB bb;
SQStatus status;
	/* Frames must be geometries of type `sqRect' */
	if(geo->geoType != sqRect) {
		F_error( NONFATAL, FBADSQFRAME, __FILE__, __LINE__,
		    "Frame on layer %s has type %d not `sqFrame'.\n",
		    geo->layer, geo->geoType );
		return;
	}
#ifdef RDEBUG
	printf( "Found a frame for instance %d.\n", instids[0] );
#endif
	if( level != 1 ) {		/* Delete local protection frames */
		if((status = SQ(sqDelete, sqGeo, geo)) <= 0) {
			F_error( FATAL, FBADSQDEL, __FILE__, __LINE__,
			  "Can't delete local protection frame. Status = %d.\n",
			    status );
		}
		return;
	}
	layer = SQLayerNameToNumber( geo->layer );
#ifdef RDEBUG
	printf( "Instanciating a frame for instance %d.\n", instids[0] );
#endif
	/* This min-max stuff is necessary since Squid confuses
	 * corners of bounding boxes upon rotational transforms */
	bb.l = F_min( geo->def.rect.l, geo->def.rect.r );
	bb.b = F_min( geo->def.rect.b, geo->def.rect.t );
	bb.r = F_max( geo->def.rect.l, geo->def.rect.r );
	bb.t = F_max( geo->def.rect.b, geo->def.rect.t );
```

...F_addframe

```
        /* Add the box to the appropriate fa_gemoetry */
        fa_add_box( &(F_ingeo[layer]), bb.l, bb.b, bb.r, bb.t );
}
```

F_addterm

```
F_addterm( geo, instids, level )
SQGeo *geo;
int instids[];
int level;
/*
 * This routine adds a terminal into the Fang data structures
 * read in from the Squid database.
 */
{
register int layer;
SQBB bb;
        /*  all terminals must be expressed with rectangular geometries */
        if(geo->geoType != sqRect) {
                F_error( NONFATAL, FBADSQTERM, __FILE__, __LINE__,
                    "Term on layer %s has type %d not `sqRect'.\n",
                    geo->layer, geo->geoType );
                return;
        }
        layer = SQLayerNameToNumber( geo->layer );
#ifdef RDEBUG
        printf( "Starting a new terminal %s, layer = %s.\n",
            geo->implements.term, geo->layer );
#endif
        /* If a terminal within an instance ... */
        if( level == 1 ) {
                return;         /* Ignore it. */
        }
        /* This min-max stuff is necessary since Squid confuses
         * corners of terminal boxes upon rotational transforms */
        bb.l = F_min( geo->def.rect.l, geo->def.rect.r );
        bb.b = F_min( geo->def.rect.b, geo->def.rect.t );
        bb.r = F_max( geo->def.rect.l, geo->def.rect.r );
        bb.t = F_max( geo->def.rect.b, geo->def.rect.t );
        /* Add the box to the appropriate fa_gemoetry */
        fa_add_box( &(F_ingeo[layer]), bb.l, bb.b, bb.r, bb.t );
}
```

*F_update*

```
#include "frame.h"

F_update( cell, outview )
char *cell, *outview;
{
SQStatus status;
SQGeo geo;
SQView view;
char *SQLayerNumberToName();
fa_box *boxptr;
register int layer, i, count;
#ifdef UDEBUG
        printf( "Updating cell %s view %s.\n", cell, outview );
#endif
        /* Make sure the current view is `outview' */
        view.mode = "w";
        view.cell = cell;
        view.view = outview;
        if((status = SQ(sqOpen, sqView, view)) <= 0) {
                F_error( FATAL, FBADSQOPEN, __FILE__, __LINE__,
                "Can't reopen `%s' as current view.  Squid returned %d.\n",
                outview, status );
        }
        for( layer = 0; layer < SQMAXLAYERS; layer++ ) {
                if( (count = F_outbox[layer].count) <= 0 )
                        continue;
                boxptr = F_outbox[layer].list;
                for( i = 0; i < count; i++ ) {
                        geo.layer = SQLayerNumberToName( layer );
                        geo.manhattanP = sqTrue;
                        geo.geoType = sqRect;
                        geo.def.rect.l = boxptr->left;
                        geo.def.rect.b = boxptr->bottom;
                        geo.def.rect.r = boxptr->right;
                        geo.def.rect.t = boxptr->top;
                        geo.function = sqFrame;
                        if((status = SQ(sqCreate, sqGeo, &geo)) <= 0) {
                                F_error(FATAL, FBADSQCREATE, __FILE__, __LINE__,
                                  "Can't create box mask %s (%d %d) (%d %d).\n",
                                   SQLayerNumberToName( layer ), geo.def.rect.l,
                                    geo.def.rect.b, geo.def.rect.r,
                                    geo.def.rect.t );
                        }
#ifdef UDEBUG
                        printf( "Frame box: layer %s: (%d,%d) (%d,%d).\n",
                             geo.layer, geo.def.rect.l, geo.def.rect.b,
                             geo.def.rect.r, geo.def.rect.t );
#endif
                        boxptr = boxptr->next;
                }
        }
        /* Save the contents of the view */
        if((status = SQ(sqSave, sqView, view)) <= 0) {
                F_error( FATAL, FBADSQSAVE, __FILE__, __LINE__,
                "Can't save `%s' as output view.  Squid returned %d.\n",
                outview, status );
        }
}
```

```
/*
Public types for Squid DBMS.

Copyright Ken Keller 1981
*/

#define SQMAXLAYERS 20
#define SQMAXDEPTH 100

typedef int SQStatus;
#define SQOUTOFVM -1
#define SQUNKNOWNLAYER -2
#define SQENDGEN -3
#define SQUNKNOWNCURRENTVIEW -4
#define SQTOOMANYLAYERS -5
#define SQUNKNOWNDEMON -6
#define SQUNKNOWNOPERATION -7
#define SQUNKNOWNOBJECT -8
#define SQHIERARCHYISTOODEEP -10
#define SQRECURSIVEHIERARCHY -11
#define SQCANNOTPARSETRANSFORMATION -20
#define SQNONMANHATTANTRANSFORMATION -21
#define SQUNTYPEDVALUE -30
#define SQUNKNOWNPARM -31
#define SQUNKNOWNTERM -40
#define SQUNKNOWNPROP -41
#define SQUNTYPEDGEO -50
#define SQCANNOTCREATEVIEW -62
#define SQCANNOTCREATECELL -63
#define SQCELLDOESNOTEXIST -64
#define SQCANNOTOPENVIEW -65 ·
#define SQVIEWEXISTS -67
#define SQVIEWDOESNOTEXIST -68
#define SQNOTAVIEW -69
#define SQCORRUPTVIEW -70
#define SQCANNOTSAVEVIEW -71
#define SQCANNOTRMVIEW -72
#define SQTRIVIALGEN -73
#define SQCANNOTCPVIEW -74
#define SQDEGENERATEPATH -75

typedef enum {sqGeo,sqTerm,sqNet,sqView,sqInst,sqParm} SQObjectType;

typedef enum {sqCreate,sqUpdate,sqGet,sqDelete,sqBeginGen,sqGen,
  sqBeginPropGen,sqGenProp,sqPutProp,sqGetProp,sqRmProp,
  sqSave,sqOpen,sqClose,sqRm,sqCp} SQOperationType;

typedef enum {sqFalse,sqTrue} SQBool;

typedef struct SQBB SQBB;
struct SQBB {
  int l,b,r,t; };

typedef struct SQRealPoint SQRealPoint;
struct SQRealPoint {
  float x,y; };

typedef struct SQIntegerPoint SQIntegerPoint;
struct SQIntegerPoint {
  int x,y; };

typedef enum {sqInteger,sqReal,sqString,sqBool} SQValueType;

typedef struct SQParm SQParm;
```

```
struct SQParm {
  char *name;
  int instID;
  SQValueType valueType;
  union {
    int integer;
    float real;
    char *string;
    SQBool bool; }
  value; };

typedef struct SQProp SQProp;
struct SQProp {
  char *name;
  SQValueType valueType;
  union {
    int integer;
    float real;
    char *string;
    SQBool bool; }
  value; };

typedef struct SQView SQView;
struct SQView {
  char *cell, *view, *mode;
  SQBB bb;
  SQProp prop; };

typedef enum {sqFrame,sqActiveArea,sqInterconnect,sqTermArea} SQFunction;

typedef enum {sqPlot,sqRect,sqLine,sqPolygon,sqCircle,sqLabel} SQGeoType;

typedef struct SQGeo SQGeo;
struct SQGeo {
  char *layer;
  SQBB bb;
  SQBool manhattanP,filledP;
  int geoID;
  SQGeoType geoType;
  union {
    SQBB rect;
    struct {
      int nPath;
      SQRealPoint *path; }
    plot;
    struct {
      int nPath;
      SQIntegerPoint *path; }
    polygon;
    struct {
      int width;
      int nPath;
      SQIntegerPoint *path; }
    line;
    struct {
      SQIntegerPoint *center;
      SQIntegerPoint *beginAngle, *endAngle;
      SQIntegerPoint *innerRadius, *outerRadius; }
    circle;
    struct {
      SQIntegerPoint position;
      int height;
      int angle;
      char *justification;
      char *label;
      char *font; }
    label; }
  def;
  SQFunction function;
  union {
    char *term;
    int net; }
  implements;
  SQProp prop; };

typedef struct SQTerm SQTerm;
struct SQTerm {
  int instID;
  char *name;
  int netID;
  SQProp prop; };

typedef struct SQNet SQNet;
struct SQNet {
  char *name;
  int netID;
  SQProp prop; };

typedef struct SQInst SQInst;
struct SQInst {
  char *name;
  char *masterCell, *masterView;
  char *cif;
  int matrix[3][3];
  int instID;
  SQBB bb;
  SQProp prop;
  SQFunction function;
  union {
    char *term;
    int net; }
  implements; };

extern SQStatus
  SQ(),
  SQAttachDemon(),
  SQDetachDemon(),
  SQSpecialGen(),
  SQGenNetTerm(),
  SQBeginLayerGen(),
  SQGenLayer(),
  SQEnd(),
  SQCurrentView(),
  SQBegin(),
  SQPopSpecialGen();
extern int SQSpecialBeginGen(),
  SQBeginNetTermGen(),
  SQLayerNameToNumber();
extern char *SQLayerNumberToName();
```

```
typedef int fa_coord;

typedef enum {FA_OK, FA_ERROR} fa_status;

typedef struct fa_box fa_box;

struct fa_box {
    fa_coord left, right, bottom, top;
    fa_box *next;
};

typedef struct fa_box_list fa_box_list;

struct fa_box_list {
    int count;
    fa_box *list;
};

        /* The eight different types of simple vertices. The shape
         * is described by the position of a equivelently shaped
         * corner in a square, while the sense is POS if vertex is
         * concave (as in a square) or NEG if convex.
         */

typedef enum {
    NO_VERTEX = 0,
    UPPER_RIGHT_NEG,
    UPPER_LEFT_POS,
    LOWER_RIGHT_POS,
    LOWER_LEFT_NEG,
    UPPER_RIGHT_POS,
    LOWER_LEFT_POS,
    UPPER_LEFT_NEG,
    LOWER_RIGHT_NEG,
    LEFT_DIAGONAL,      /* UPPER_LEFT_POS and LOWER_RIGHT_POS */
    RIGHT_DIAGONAL      /* UPPER_RIGHT_POS and LOWER_LEFT_POS */
} fa_vertex_type;

typedef struct fa_vertex fa_vertex;

struct fa_vertex {
    fa_coord x,y;
    fa_vertex_type type;
    fa_vertex *next;
};

typedef struct fa_geometry fa_geometry;

struct fa_geometry {
    int count;    /* number of vertices */
    int status;   /* indicates whether the geometry is sorted, merged,
                     etc. */
    fa_box bb;  /* the bounding box */
    fa_vertex *head, *tail;         /* pointers to the vertex list */
};

typedef struct fa_edge fa_edge;

    /* if the edge is horizontal, then low is the low x end,
       if the edge is vertical, then low is the low y end. */

struct fa_edge {
    fa_coord low, high;
    fa_coord center;
    struct {
         unsigned int edge : 2;           /* direction the edge points,
                                             either FA_LEFT, FA_RIGHT,
                                             FA_UP, or FA_DOWN */
         unsigned int low_corner : 1;     /* FA_CONCAVE or FA_CONVEX */
         unsigned int high_corner : 1;    /* FA_CONCAVE or FA_CONVEX */
         unsigned int
    } type;
    fa_edge  *next;
};

#define FA_LEFT 0
#define FA_RIGHT 1
#define FA_UP 2
#define FA_DOWN 3
#define FA_CONCAVE 0
#define FA_CONVEX 1

fa_status fa_merge();
fa_status fa_frame();
fa_status fa_to_edge();
fa_status fa_to_box();
fa_status fa_add_box();

extern char fa_err_string[];         /* String containing error messages
                                      * after a fang routine returns FA_ERROR
                                      */
```

# References

[Akers70] S. B. Akers, J. M. Geyer, and D. L. Roberts, "IC Mask Layout with a Single Conductor Layer", *Proceedings of the 7th Annual Design Automation Conference*, June 1970, pp 7-16.

[Bales80] M. W. Bales, "The CABTOCIF Program", CADMAN on Berkeley UNIX† System, University of California at Berkeley, Berkeley, CA, June 1980.

[Bondy76] J. A. Bondy, U. S. R. Murty, *Graph Theory with Applications*, North Holland, New York, 1976.

[Cho77] Y. E. Cho, A. J. Korenjak, and D. E. Stockton, "FLOSS: An Approach to Automated Layout for High-Volume Designs", *Proceedings of the 14th Annual Design Automation Conference*, June 1977, pp. 138-141.

[DeMan82] H. DeMan, Private Communication, Apr 1982.

[Dunlop78] A. E. Dunlop, "SLIP: Symbolic Layout of Integrated Circuits with Compaction", *Computer-Aided Design*, Vol. 10, No. 6, Nov 1978, pp. 387-391.

[Dunlop79] A. E. Dunlop, "Integrated Circuit Mask Compaction", PhD Thesis, Carnegie-Mellon University, Pittsburgh, PA, 17 Oct

†UNIX is a Trademark of Bell Laboratories.

1979.

[Dunlop80] A. E. Dunlop, "SLIM - The Translation of Symbolic Layouts into Mask Data", *Proceedings of the 17th Annual Design Automation Conference*, June 1980, pp. 595-602.

[Feldman78] S. I. Feldman and P. J. Weinberger, "A Portable Fortran 77 Compiler", UNIX System Documentation, August 1978.

[Gibson76] Dave Gibson and Scott Nance, "SLIC - Symbolic Layout of Integrated Circuits", *Proceedings of the 13th Annual Design Automation Conference*, June 1976, pp. 434-440.

[Hsueh79] M. Y. Hsueh and D. O. Pederson, "Computer-Aided Layout of LSI Circuit Building-Blocks", *Proceedings of the 1979 IEEE International Symposium on Circuits and Systems*, pp. 474-477.

[Hsueh79] M. Y. Hsueh, "Symbolic Layout and Compaction of Integrated Circuits", PhD Thesis, UCB/ERL M79/80, University of California at Berkeley, Berkeley, CA, 10 Dec 1979.

[Hurt82] J. Hurt, Private Communication, Mar 1982.

[Johnson80] S. C. Johnson, "A Tour Through the Portable C Compiler", UNIX System Documentation, 1980.

[Keller81] K. H. Keller, "KIC: A Graphics Editor for Integrated Circuits", MS Report, University of California at Berkeley, Berkeley, CA, June 1981.

[Keller81] K. H. Keller, "Squid: A Database System for Integrated Circuits", Preliminary Draft, University of California at Berkeley, Berkeley, CA, December 1981.

[Keller82] K. H. Keller, Private Communication, April 1982.

[Kernighan76] B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Reading, MA, 1976.

[Kernighan78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, London, 1978.

[Larsen71] R. P. Larsen, "Computer-Aided Preliminary Layout Design of Customized MOS Arrays", *IEEE Transactions on Computers*, Vol. C-20, No. 5, May 1971, pp. 512-523.

[Larsen78] R. P. Larsen, "Versatile Mask Generation Techniques for Custom Microelectronics Devices", *Proceedings of the 15th Annual Design Automation Conference*, June 1978, pp. 193-198.

[Lawler76] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, 1976.

[Lock81] E. Lock, "Techniques for the Construction of Parameterized Functional Modules", MS Report, University of California at Berkeley, Berkeley, CA, Dec 1981.

[Mead80] C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, 1980.

[Moore82] P. Moore, "The FANG Manhattan Polygon Package", CADMAN on Berkeley UNIX System, May 1982.

[Newman80] W. M. Newman and R. F. Sproul, *Principles of Interactive Computer Graphics*, McGraw-Hill, New York, 1980.

[Ritchie78] D. M. Ritchie and K. Thompson, "The UNIX Time-sharing System", *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, July - August 1978.

[Thesen78] A. Thesen, *Computer Methods in Operations Research*, Academic Press, 1978, Chapter V.

[Weste81] N. H. E. Weste, "Virtual Grid Symbolic Layout", *Proceedings of the 18th Annual Design Automation Conference", June 1981, pp. 225-233.*

[Weste81] N. H. E. Weste, "MULGA - An Interactive Symbolic Layout System for the Design of Integrated Circuits", *The Bell System Technical Journal*, Vol. 60, No. 6, July - August 1981, pp. 823-857.

[Williams78] J. D. Williams, "STICKS - A graphical complier for high level LSI design", *AFIPS Conference Proceedings*, Vol. 47, June 1978, pp. 289-295.