SCHEMATIC GENERATION

by

Godfried M. Swinkels

**PhD. Queen's University, Kingston,
Ontario, 1971**

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
December 1987

Approval

Name: Godfried M. Swinkels

Degree: Master of Science

Title: Schematic Generation

Examining Committee:

Jia-Wei Han
Chairman

Louis Hafer
Senior Supervisor

Patrick St.Dizier
Supervisor

Ramesh Krishnamurti
External Examiner

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

SCHEMATIC GENERATION

Author:

(signature)

G.M. SWINKELS

(name)

JAN 4 1988

(date)

Abstract

A schematic is a graphical representation of relationships among parts of a system. Examples are functional block diagrams, logic diagrams, flow sheets of chemical processes and schedules.

Schematics are often the principal means of conveying a design and to that end they must demonstrate properties of the design such as direction of flow and logic clusters and obey graphical criteria such as few turns and crossings.

Their manual generation requires strong combinatorial skills. Algorithmic generation does not deal well with the multiple objectives or the required recognition of patterns. A successfull project in VLSI routing, a problem of a similar combinatorial nature, through an artificial intelligence approach suggested schematic generation with an expert system.

The present thesis examines the knowledge representation and the control of the reasoning in schematic generation with an expert system.

The generation is partitioned into subproblems to be tractable. Subproblems include placement of symbols and the placement of progressively more detailed parts of nets. Placements try to achieve desirable configurations specified by rules; placements may generate constraints which are propagated forward.

The placement of symbols in a schematic with feedback is related to the problem of the selection of tear variables and equation ordering in the solution of systems of simultaneous equations by iterative methods.

The theoretical considerations led to a compact implementation in Prolog, which handles examples of the kind and complexity mentioned in the literature.

Acknowledgements

I owe a debt of gratitude to my supervisor, Dr L.Hafer. He gave guidance, he truly listened to problems, he helped with ideas. In fact the weekly discussions were something I looked forward to. His painstaking perusal of the various drafts made the thesis a polished document which it would not have been otherwise.

Thanks to Patrick St.Dizier with whom I shared some problems in making Prolog do my bidding, and Ramesh Krishnamurti who gave his attention as the outside examiner.

I acknowledge the financial support of the University and the Centre for System Science.

The work is dedicated to my wife who once more and presumably for the last time had to put up with me pursuing studies.

Table of Contents

List of Figures

Chapter 1
Summary and Highlights

a picture is worth a thousand words

1.1. Background and Motivation

Schematics are graphical representations of relationships among parts of a system. Examples are functional block diagrams, logic diagrams and circuit schematics in electronics, flowcharts of computer programs, flowsheets of chemical processes and schedules of many kinds. One such example is given in Figure 1-1.

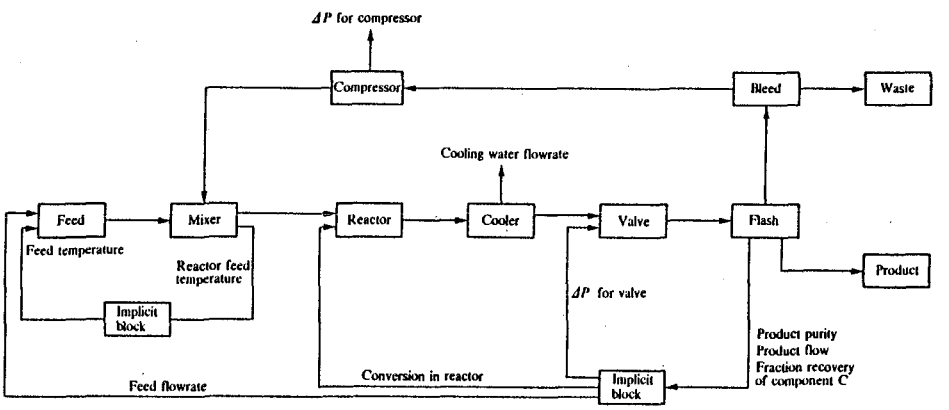

Figure 1-1: Functional Block Diagram

Schematics are often the principal means of conveying designs or at least the overview. To carry out this task schematics must satisfy certain criteria:

- maintain direction of flow
- reflect the existence of logical clusters
- minimize the number of intersections
- minimize the number of sharp turns
- utilize and minimize the drawing area

In our wide definition of a schematic, flow can stand for signals in VLSI, information, time or a commodity. This thesis is concerned mostly with application to functional block diagrams and logic diagrams in VLSI.

The first two criteria reflect the need for a suitable functional representation, the others help and guide the eye to recognize features. There are other desirable characteristics which enhance fast comprehension of a schematic, but for which no compact verbal descriptions exist (such as 'colinearity' in a later chapter).

From a combinatorial viewpoint, the generation of a schematic satisfying the above criteria is difficult. A much-simplified statement of schematic generation is equivalent to weighted matching on a tripartite graph. A subproblem of the complete schematic generation can be solved by channel routing; this problem in itself is NP- hard. These points are important in that they suggest the nature of algorithms for the generation: one can satisfy criteria, but one cannot find an optimum for large problems. Algorithms must make use of heuristics and iterative improvement.

The manual construction of a good schematic is a difficult job. Its combinatorial nature demands special skills in perceiving and manipulating spatial relations among symbols, a skill akin to that required in chess and go. Such skills are in limited supply. By the same token the job is time-consuming to the point that it often is on the critical path, and error-prone because of its iterative nature.

The problems of manual schematic generation suggest a computer solution. CAD systems take over some of the tasks such as redrawing which might (re-)introduce errors. They do not assist with the combinatorial problems or the consistency checks which make generation difficult in the first place.

The algorithmic generation of schematics does not deal easily either with multiple objectives or the type of pattern recognition which invokes special treatment. Algorithmic generation requires large programs; by their very nature these programs are difficult to modify. Thus addition and subtraction of special cases or a quick adaptation to a new style of schematic is almost impossible.

The present project starts from the following observation: The algorithmic routing of channels and switch boxes is difficult. Joobbani [Joob 86, Joob 85] demonstrated that a rule-based expert system can produce high quality routings and resolve some difficult problems on which algorithmic routers fail. Expansion and modification of the collection of patterns, special treatments and 'style' is possible; this so-called knowledge base is available in the form of a set of explicitly stated and independently accessible rules.

The generation of schematics has combinatorial characteristics similar to the routing problem; in addition

schematics have more ill-defined (perhaps not yet recognized) good and bad configurations. This strongly suggests that the generation of schematics with AI techniques is a project of merit.

1.2. Objectives

The present project applies an expert system to the generation of schematics with the specific goal of establishing whether the paradigm of knowledge-based programming can make a contribution. This includes questions on knowledge representation and control.

The thesis examines the theoretical aspects of schematic generation.

The project includes the implementation of a schematic generation system.

The program is given a netlist and a list of symbols as input; alternatively the symbols are already placed. Interaction is allowed on placement. The program returns a placement of the symbols and a routing of the nets which satisfies the criteria. Depending on the actual computer used, the result is displayed on a screen or presented on a printer.

In view of problems encountered with schematic routing in OPS5, the project examines the application of Prolog to the complete schematic generation problem. Prolog because of its dual nature should be able to deal with both the procedural aspects and the embedded reasoning systems.

Part of the reasoning consists of solving a constraint satisfaction problem, which normally calls for forward chaining data-driven inference. Efficient implementation in Prolog with its native goal-driven strategy is of particular interest.

1.3. Design Decisions

The thesis proper discusses and justifies the design decisions and the implementation details, both in the light of the previous work[1] and our own theoretical work.

Data structures are selected to impose a desirable representation of a net as shown in figure 1-2; this figure also includes the names of parts of the representation of a net. The whole schematic is mapped on a grid,

[1] Underlined words point to specific chapters of that name

which limits the reasoning process to a finite set of possibilities; figure 1-3 demonstrates the grid structure. The grid structure allows empty symbol positions and it is thus not restricting.

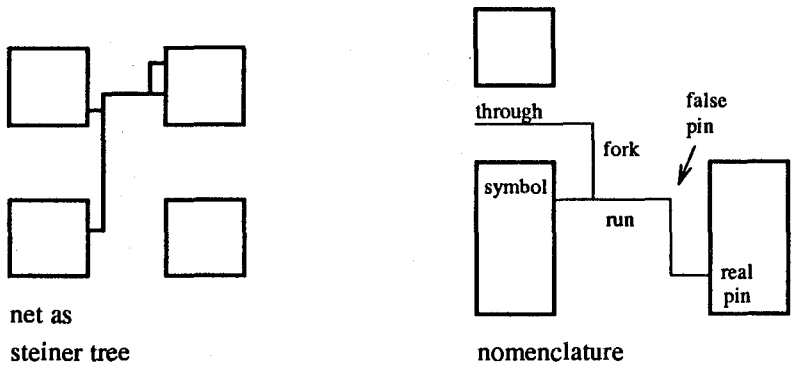

Figure 1-2: Simple Net as a Steiner Tree and the Parts of a Net

Following the literature on algorithmic schematic generation we observe a partitioning of the problem;

- Horizontal ordering of symbols into columns.
- Vertical ordering (within columns)
- Assignment of throughs
- Refinement of symbol positioning and through location
- Positioning of runs within a vertical channel
- Relative positioning of forks
- Drawing the schematic

Placement of the symbols is supposed to demonstrate the direction of information flow, minimize intersections and show logical clusters. It is a quadratic assignment problem, for which heuristic iterative schemes exist.

A specific method by May to carry out the complete placement in one step consists of repeated improvements of the ordering of the individual rows by a sort on a distance metric followed by the same process on the columns. The scheme was found wanting on any but the simplest examples; it got stuck on local optima. A method for the reduction of the total number of throughs failed in the same manner. Because of these failures the problem was partitioned in horizontal and vertical ordering.

Horizontal ordering is treated weakly in the literature; most authors do not address feedback explicitly. We

relate the problem of horizontal ordering to the solution of a set of simultaneous equations; a specific iterative solution method, direct substitution, requires the determination of a good set of feedback variables (tear variables, which tear the cycles). In fact a good set is not unique, it belongs to an equivalence class of sets; the desirable ordering is a member of that class (of which our ordering is an approximation).

We implemented a heuristic approximation which labels the nodes in precedence order through a breadth-first search and selects a desirable arc for elimination when a cycle is found to be present. Because of the analogy of the method to the critical path method, we obtain a rational method to select arcs for compaction.

Vertical ordering is carried out by the method of May referred to above applied to the columns only. The method remains a heuristic but in the limited context its power seems satisfactory.

Assignment of throughs to a specific gap between two symbols corresponds to logical routing, the choice of channels for a net if a choice exists. A full model based on avoiding crossings is too complex to compute; a model based on avoiding crossings at the ends is too weak a heuristic. The implementation builds up a vertical constraint graph by the applying a set of increasingly stronger ordering rules.

Refinement of the positioning of symbols and the actual positioning of throughs corresponds to detailed or physical placement; its prime purpose is to reduce the number of turns in nets. It is implemented as a greedy local improvement scheme (which guarantees that turns are not obviously redundant but not a minimum number of redundant turns).

Positioning of Runs is carried out as a search with backtracking on an ordered list of lists of preferred positions for a net. The list is ordered on the number of options available to a net, nets with few options first.

Relative Ordering of Forks is strictly based on the recognition of patterns and the application of a desirable ordering based on that pattern. This is a data-driven rule application: its efficient implementation in Prolog was of some interest.

Drawing the schematic through the mapping of abstract data structures to graphic primitives poses no interesting problems. The examples of figures 1-3 and 1-4 were generated directly through the inclusion of Postscript[2] files in the manuscript. The same representation can be seen on a screen so that an immediate check on the suitability of a schematic is obtained.

---

[2]Copyright by Adobe Systems

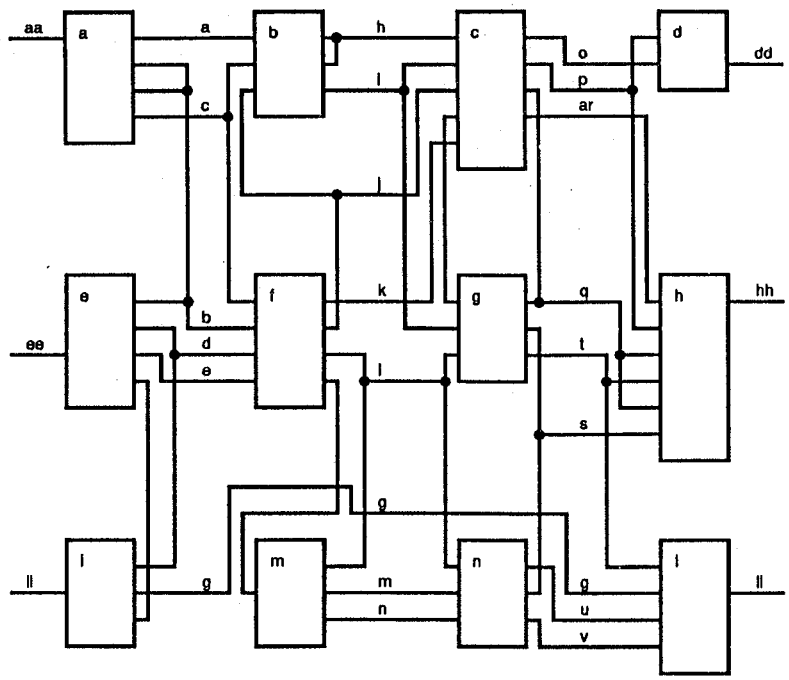

Figure 1-3: Schematic

This aspect is of importance to the interface between user and program. We have taken the position that the knowledge on schematic generation contained in the algorithms and rules cannot possibly cope with all possible cases, exceptions, personal preferences, *etc.* Thus there should be simple means of seeing and modifying (by addition or subtraction of constraints) intermediate results.

The design of the program in itself deserves some notice. The program is a mixture of algorithmic and pattern-based heuristic programming. If -as in the case of ordering- proper graph-theoretical algorithms exist, it would be foolish to insist on rule-based programming.

The rule-based parts follow a pattern, which consists of the mapping of the feature of interest to an abstraction, followed by the application of rules in some ordered manner. Backtracking is avoided except in the case of runs, which all have to be placed. Instead of backtracking we search over rules to avoid obnoxious features and include desirable ones.

1.4. Conclusions and Suggestions for Further Work

We draw the following conclusions (based on our own considerations and experience during program development and on direct comparison with other implementations of schematic generation):

1. The knowledge based paradigm is appropriate for schematic generation only in parts. It serves local planning such as placement of forks and runs very well. It is less effective in the global decisions such as placement.
2. Regardless of the previous observation, the power of relational programming and the interactive environment are most helpful in the development phase.
3. The correspondence between horizontal ordering and the selection of tear variables in the iterative solution of sets of equations by direct substitution gives a novel approach to horizontal ordering.
4. Prolog as opposed to, for example, OPS5 is an efficient vehicle for the combination of declarative and procedural knowledge in schematic generation.

The concepts contained in this thesis and specifically in the implementation can be improved and extended. In particular:

1. For logic gates pin placement should be manipulated to minimize turns and crossings.
2. Larger schematics would need quadratic run placement.
3. Implementation of the full cycle recognition package would allow folding, a common method of compaction in process flowsheets.
4. We were not able to come up with a reasonable local improvement scheme. It may require substantially different knowledge representation. An updating scheme remains a desirable goal.
5. The splitting of very large schematics by clustering techniques specific to directed graphs is unexplored territory.

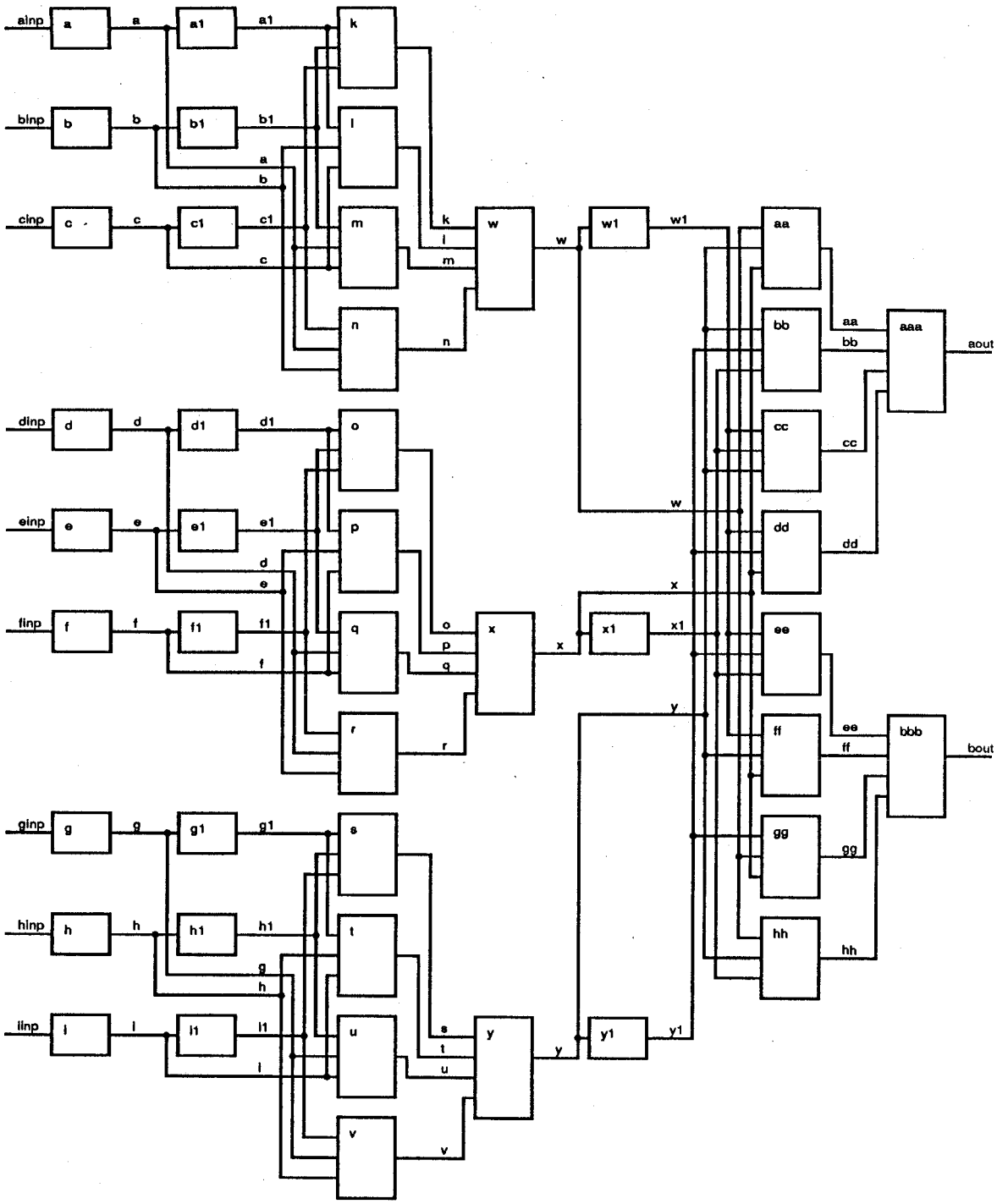

Figure 1-4: Gate-level Logic

Chapter 2
Previous Work

2.1. Introduction

In this chapter we discuss existing work on the generation of schematics. A few key papers deal with the complete problem from the definition of objectives through placement to routing, others deal with just one aspect. It is convenient to mention each paper with its particular context and then review the technology aspect by aspect.

The assumption is that the whole problem of schematic generation has to be partitioned into some form of placement followed by global and detailed (channel and switchbox) routing. Later on we will defend the necessity of partitioning and consider the different ways.

Arya [Arya 85] is the most recent work known to us which purports to discuss the whole schematic generation problem. The paper is the only one to mention the definition of a bus in the context of schematics.

May and co-workers [May 83, May 84, May 85] state a number of schematics such as functional block diagrams, flow charts, logic diagrams, *etc.*, as multi-row schematics. The multi-row schematic is a collection of symbols of similar sizes located on a grid. The particular contribution is a simple and powerful heuristic to minimize crossings; the heuristic hinges on the definition of the multi-row schematic.

Aoudja [Aoudja 86] deals with diagrams such as are used for the description of the electrical system of a plane, car or electromechanical device. Thus there is a notion of panels or junction boxes with a distinct geometrical position. Of interest to us is the global router, which uses a search algorithm over a channel adjacency graph.

Additional papers on algorithmic schematic generators will be quoted on minor points.

The main reference on the application of an expert system in routing is the work of Joobbani [Joob 86, Joob 85]. Its context is purely a limited aspect of VLSI design.

The P1 placement and interconnect system of Rivest [Rivest 82] is intended for VLSI design. It is included in the literature survey because it contains a noteworthy global router based on the construction of Steiner trees and an interesting heuristic for the placement of the locations where nets cross from channel to channel.

2.2. Requirements

The papers quoted above develop specific criteria for a good schematic. In quick repeat, these are:

1. maintain signal flow.
2. reflect the existence of logical clusters.
3. minimize the number of intersections and turns.
4. utilize and minimize the area.

Other requirements mentioned less prominently are:

5. buses are to be laid out as recognizable entities (*e.g.*, straight lines [Arya 85]).
6. functional blocks and nets are to be labeled.
7. inputs and outputs come out to the borders of the schematic
8. the schematic should reflect standards.
9. a schematic too large for one sheet is partitioned logically over different sheets.
10. a schematic should be stable under small changes.

All papers invoke a further undefined concept of aesthetics.

A few definitions are in order.

A logical cluster is a group of symbols used for the same task; it is usually distinguished by a high connectivity among the symbols relative to the connectivity of the group to the outside.

A net is a group of connected pins and the connection itself. There usually is at least one source and at least one sink in a net.

A bus is a net with many sources and/or many sinks. In an alternative definition it is a group of nets which carry related signals, *e.g.*, the individual bits of a signal carried in parallel.

A functional block or a symbol takes one or more inputs and transforms these into one or more outputs.

Requirement 1) is usually achieved through the imposition of a preferred direction of flow and the formation of the longest possible chains (see figure 2-1).

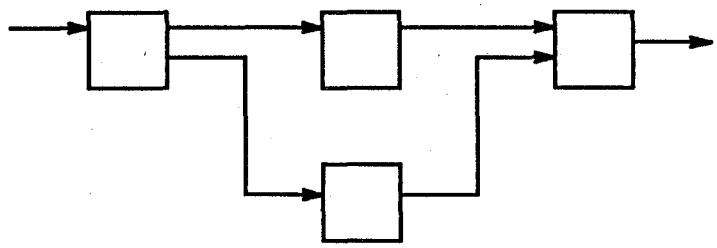

Figure 2-1: Left-to-Right Information Flow

The whole purpose of a schematic is to facilitate comprehension and acquisition of an overview. Requirement 1) stresses the sequence of operations on a signal or whatever commodity the arcs of the schematic represent. Requirements 1) and 2) aid cognition by making functional relations clear through proximity or connectivity (a recognizable bus).

Requirements 3), 4) and 6) are visual aids which have a particular function in guiding the eye in its examination of a particular connection or net. Recognition of a net or some other component is a goal, the achievement of which can be measured if need be. In general, this type of pattern recognition is the subject of ergonomics, where one might look for further guidance on requirements. The invocation of aesthetics seems particularly useless in comparison: it almost says that further inquiry will be futile.

The requirement on the visibility of a bus is interesting. Arya defines a bus as a net with "many" branches. She suggests the imposition of a particular representation (*e.g.*, a straight vertical line). In the usual partitioning of the schematic generation problem, this would call for a special placement of functional blocks, from which a routing of the bus would follow!

Other definitions of a bus might emphasize multiplicity of similar or parallel paths. I am not impressed with the representation of a "bus" by n lines as a strong way of conveying meaning. Perhaps we should make a distinction between a schematic and a wiring diagram.

This latter point has some merit, in that in a wiring diagram with a pin-by-pin representation of the chip the notion of direction of information flow from left to right is completely lost as well.

Leaving the discussion on the necessity of a bus, one can distinguish a need or requirement in a schematic for

lines, which go in a similar direction from the same block, to do so in parallel. For this I like to coin the word colinearity (if somebody has not already done so). This desirable concept is independent of intersections or turns as figure 2-2 can demonstrate. It is in the same class as crossings in that it is an aid in recognition.

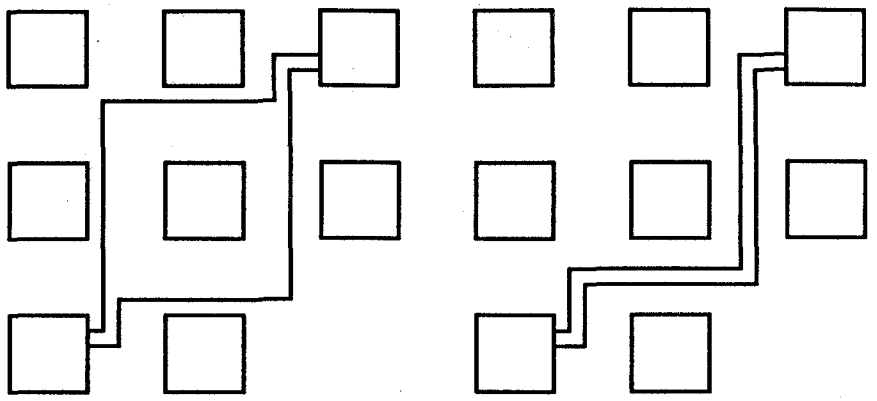

Figure 2-2: Colinear and Non-Colinear Nets

Requirement 4) could be captured in the word 'compactness'. This requirement interacts and puts limits on how requirement 8) is satisfied.

Requirement 9) on stability is an addition of the author. A schematic, however good it is, still takes a certain amount of time to absorb. Some image of the schematic is retained by the (frequent) user, *i.e.*, certain feautures are known to be in certain places. A small change in or addition to the schematic should not cause a major reshuffle of symbols or nets, thus forcing the user into a new recognition process and a new image. This somewhat imitates manual correction of prints: local features are erased and redone.

A schematic does not have to obey certain requirements imposed on VLSI placement and routing algorithms. Requirements such as limitations on number of tracks or the configuration of vias relate to economy or the physical realization. In fact, some of these requirements (minimum wirelength, minimum number of tracks) or rather the means by which they are achieved, would be detrimental to the purpose of a schematic. Thus a dogleg has no place in a schematic. This very much limits the applicability of conventional routers to schematics. A similar statement can be made on placement.

May enumerates the reasons for generating a schematic by computer:

1. To facilitate data entry (from a data base or design program). The unstated idea is that the schematic is an integral part of a CAD DBS.

2. To eliminate human error.
3. To shorten the drafting cycle.
4. To enforce standards.

We will later make the point that these goals are better served and in a more flexible manner by knowledge-based systems than by the algorithmic approach.

2.3. Algorithmic Schematic Generators

2.3.1. Partitioning of the Problem

According to May [May 85], it is necessary to split the total problem of schematic generation into a number of carefully chosen subproblems to keep it tractable. The meaning of this is that the schematic generation is carried out as a sequence of subproblems without backtracking on the intermediate results.

In support of this we note that any reasonable partition of the problem results in at least one sub-problem, which is (now) known to be NP-complete. An attempt at a simultaneous complete solution would obviously be that much more complex than the sequential solution.

As a specific example, the minimum number of crossings was given as the strongest visual criterion. Minimum crossing number is NP-complete even for bipartite graphs [Garey 83] This latter fact precludes the multi-row graph from being simpler than an arbitrary digraph.

Most papers refer to minimizing the number of feedback nets in a schematic: again the minimum feedback arc set on a digraph is NP-complete (problem GT8 in [Garey 79]).

Shing and Hu [Shing 86] give a complete survey of NP-complete problems in VLSI design. Most of these apply to schematics as well.

The schematic routing problem (as opposed to placement) is of a fundamentally lesser complexity because we have the freedom to add extra tracks in both directions. As a quick lookahead, the knowledge-based channel router of Joobbani tackles a NP-complete problem [Shing 86].

Assuming then that we accept the need for a partitioning of the total problem, the above remarks are still pertinent for the choice of algorithms (approximation *vs.* exhaustive search, iterative improvement).

We make these points here rather than in the discussion to get a proper appreciation of the literature.

A normal partitioning is as follows:

1. Logical Placement or Ordering of Symbols
    a. Precedence Ordering of Symbols left-to-right
    b. Ordering of Symbols within Columns
2. Global Routing, the assignment of nets to horizontal channels
3. Improvement of Symbol Placement
4. Physical or Detailed Routing of Runs and Forks

The above terminology is the one used by the authors quoted below; it is somewhat non-standard relative to the VLSI literature.

Arya precedes ordering of symbols by identification of buses; she forces a bus into one vertical column.

Smith and May see precedence and column ordering as an iterative loop, Arya implements them as two sequential processes.

May lumps global and detailed routing (to be carried out by a conventional router) and uses the physical placement and an expansion process to straighten out doglegs and turns.

Marathe *et.al.* make no mention of partitioning at all. They place symbols in trees according to connectivity; placement of one symbol in more than one tree is normal. After this phase, duplicate branches among trees are lopped off and the trees reordered. Growing complete trees and finding duplicate branches are combinatorially expensive.

The literature mentions the need for splitting a schematic, if the schematic is too large for one sheet. The partitioning of digraphs or the criteria for a good partition are then treated with neglect.

The graph partitioning of Kernighan [Ker 70] really pertains to undirected graphs. The problem is not identical to partitioning in VLSI , where wire length is the sole criterion; in this case the partition has to have some logical shape. Nor is the problem a matter of constructing a schematic the size of a bed sheet and cutting it into drawing size pieces (as suggested by May); the inter-sheet connections do not have to add up to a planar or near-planar representation.

Smith recommends single-seed partitioning. This may form one coherent sheet, but the symbols rejected for this one sheet do not necessarily add up to something reasonable.

I believe there is a form of clustering needed (topological, logical?) which is not treated in the literature. Heller [Heller 82] discusses a clustering technique based on topology exclusively, but not enough detail is given.

2.3.2. Placement

Assume that the schematic is given as a list of symbols and a net list and that the direction of information flow is known.

Placement of course determines the eventual routing. The literature generally splits placement into two steps.

Assignment of Columns

Step 1 is the ordering of symbols into columns. The principal objective of the stage is to allow the demonstration of the sequential nature of the signal flow. This implies minimizing the number of feedback arcs in the schematic.

There is an intimation, that this can be achieved by topological ordering, starting at the inputs and with elimination of cycles. Aoudja and May mention the minimization of the number of feedback arcs by this ordering, but this is an unsupportable claim.

It is true, that after a feedback set has been assigned (and removed from the graph) one can assign vertices to columns on the basis of the longest distance from the origin to a vertex. A set of feedback arcs can be generated by a breadth first search; this set gives long chains of connected blocks from the input as will be shown in the thesis.

Smith obtains a preplacement as a result of his single-seed partitioning, which is refined with a pair-wise interchange scheme based on an objective function which accounts for interconnection lengths and back arcs (not further defined).

Ordering within Columns

Step 2, the ordering of symbols within a column, is carried out to minimize the number of crossings. This makes the relations among strongly connected units more obvious.

Smith has obtained a vertical order from the preplacement. It is improved with pair-wise interchange with an objective function containing adjacency and interconnection length. The algorithm includes iterative improvement as expected.

Arya does her vertical ordering column-by-column in one pass. Rows are created and inserted as need be. The effect of columns of symbols already in place is modeled by minimizing a sum of path lengths to these symbols. The effect of yet unplaced symbols is represented by an affinity function (which counts direct connections between two modules as well as connections to common modules). The first is an assignment problem, the second a quadratic assignment problem. It is solved by an approximation method. Arya could have quoted May [May 83, May 84].

The most extensive (and most recent) theoretical discussion of the minimization of the crossing number of a bipartite graph is contained in [May 84]. The argument hinges on the relation between a bipartite graph and the adjacency matrix describing the graph as demonstrated in Figure 2-4. The bipartite graph is mapped to the matrix in the following manner. The vertices on the left of the graph are identified with the rows of the matrix. The vertices on the right are identified with the columns of the matrix. An entry in the matrix at row i column j means that there is an arc in the graph from vertex i on the left to vertex j on the right.

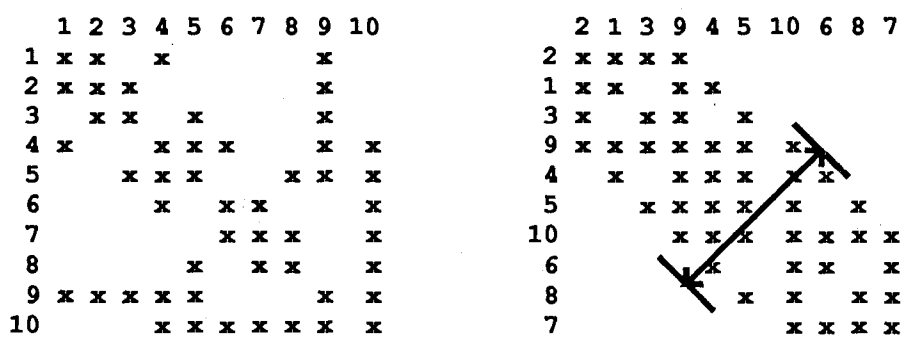

Figure 2-3: Matrix before and after Bandwidth Minimization

May works out some upper bounds on the number of crossings in the bipartite graph on the basis of its structure (number of cycles of uneven length). The most important of these is a bound stated in terms of the bandwidth of the matrix describing the bipartite graph. The concept of bandwidth is demonstrated in the right-hand matrix of Figure 2-3

May now argues as follows: if one can find an algorithm which reduces the bandwidth and thus the upper bound on the number of crossings, the actual number of crossings will likely decrease.

The relationship can be made intuitively clear (although not by mathematical proof) by the following consideration. The number of crossings is equal to the number of submatrices of the form

```
a   1
1   b
```

in the adjacency matrix. In the submatrix a and b can be 0 or 1. By bringing off-diagonal elements of the adjacency matrix closer to the diagonal, fewer submatrices of that form can be constructed. Off-diagonal "1" elements are brought closer to the diagonal through row and column permutations of the matrix, which corresponds to vertex interchanges of the bipartite graph.

In Figure 2-4 we show a bipartite graph and its adjacency matrix. We observe in the marked-up adjacency matrix that the distance from an entry to the diagonal in the same row (the lines O----□) is precisely the (vertical) distance between the two nodes of the bipartite graph connected by this edge. From this we conclude that an algorithm which decreases edge length improves crossing number!

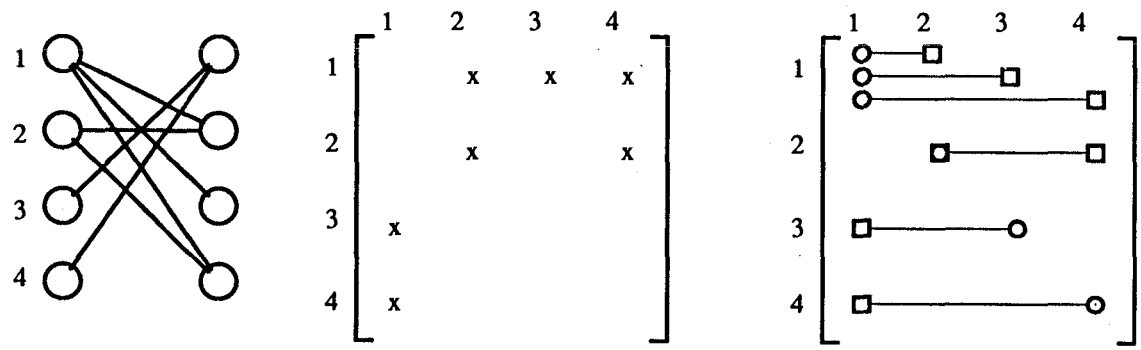

Figure 2-4: Digraph and its Matrix

The novel algorithm operates on a representation of the schematic as a multi-partite graph . This is the point where the definition of multi-row schematics as nodes on a grid is used. I will demonstrate May's algorithm with two figures.

In the algorithm the nodes on one side of a component bipartite graph of the multi-partite graph (see Figure

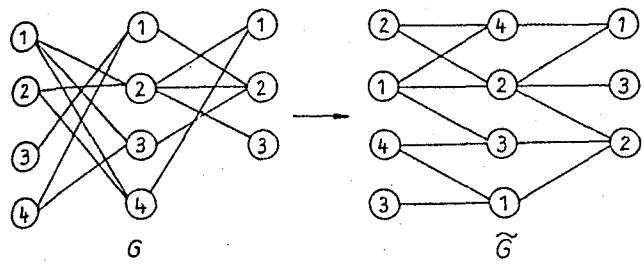

Figure 2-5: Sorting on a Tri-Partite Graph

2-5) are sorted and placed according to the average of the coordinates of the nodes they are connected to on the other side; this procedure reduces the (vertical) edge lengths. Any ties are resolved by making a swap; this is to keep the algorithm from getting stuck. This simple metric gives surprisingly good performance in an iterative improvement scheme both with respect to edge length and the goal of lowering the number of crossings.

The algorithm generalizes in an obvious manner to a multi-partite graph by application to the first two columns, then sets of three columns and finally a set of two columns. In the particular case of nets which cover more than one column, the representation of the net is forced to a format suitable for the multi-partite version by the introduction of dummy vertices as shown in Figure 2-6.

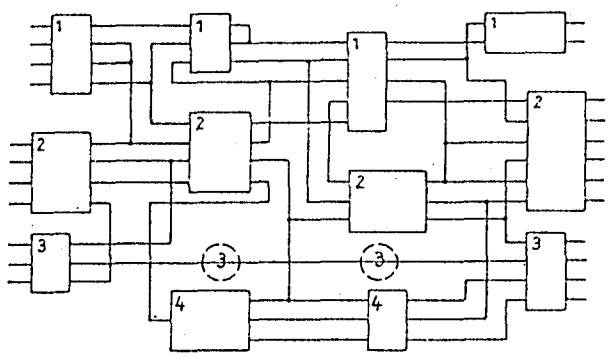

Figure 2-6: Multirow Graph with dummy Nodes

The author shows cases where it performs better than pair-wise exchange. This is of course because each step considers and possibly repositions all nodes of a column simultaneously.

Figure 2-7 shows the application of the same algorithm to a general graph on a grid. The algorithm is applied alternately to the partite graphs supported by rows and columns.

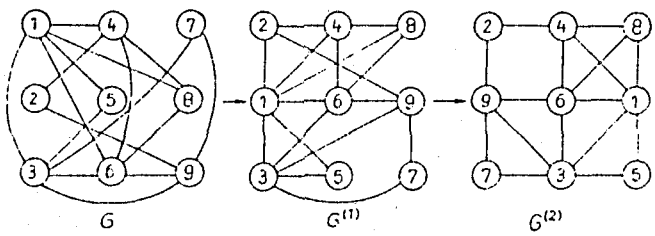

Figure 2-7: The Algorithm on a general Graph

While bemoaning the iterative nature of the algorithm (which I think is part and parcel of any algorithm for this kind of problem), he suggests the placement of at least one column by a constructive method. The algorithm is based on computing affinities among elements in the unplaced columns.

From this we can clearly identify Arya's algorithm as a one-pass combination of the two algorithms of May. We prefer May because of its simplicity and the iteration.

2.3.3. Routing

Global Routing

Global routing is an issue in systems which do not have a topological placement[3]. Aoudja places symbols in relation to their physical location in some system.

Given a set of placed symbols Aoudja must determine through which channel (see Figure 2-8) a given net will pass. He plans to use an A* algorithm (a branch-and-bound-like search technique [Nilsson 80]) to accomplish a global routing which results in an eventual detailed routing with few crossings. It is rather a trade-off between doing the work in a topological placement or in a search on the channel graph.

Detailed Routing

Most authors use some modification (single layer, intersections allowed) of a normal channel router. If the channel router operates in a wide channel[4], an acceptable routing may or may not result; it may still contain a Steiner tree representation with the Steiner points in a wrong orientation.

---

[3]Topological placement is a placement based on the connectivity

[4]a channel with more tracks than nets

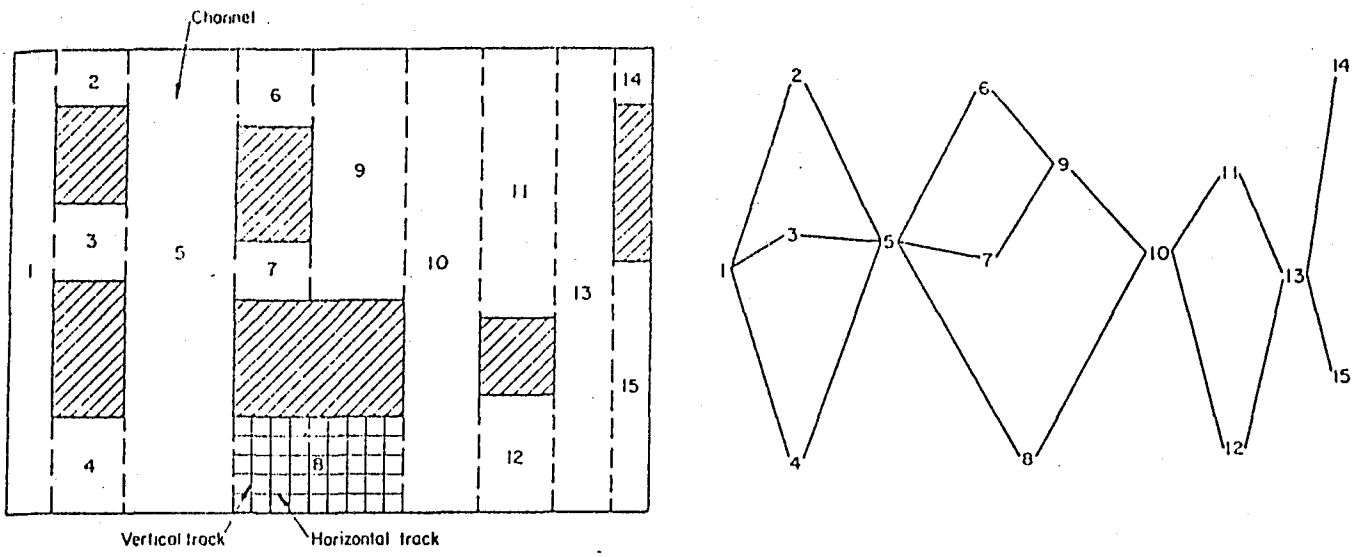

Figure 2-8: The Definition of Channels and a Channel Graph

May elects to run with a dense routing [Yoshi 82]. He is then forced to another processing step. The shifting of the symbols to straighten runs has a major impact on the quality of the schematic, as demonstrated below. This processing step seems non-trivial. Again we have to add a processing step to undo the effect of using an unsuitable algorithm before.

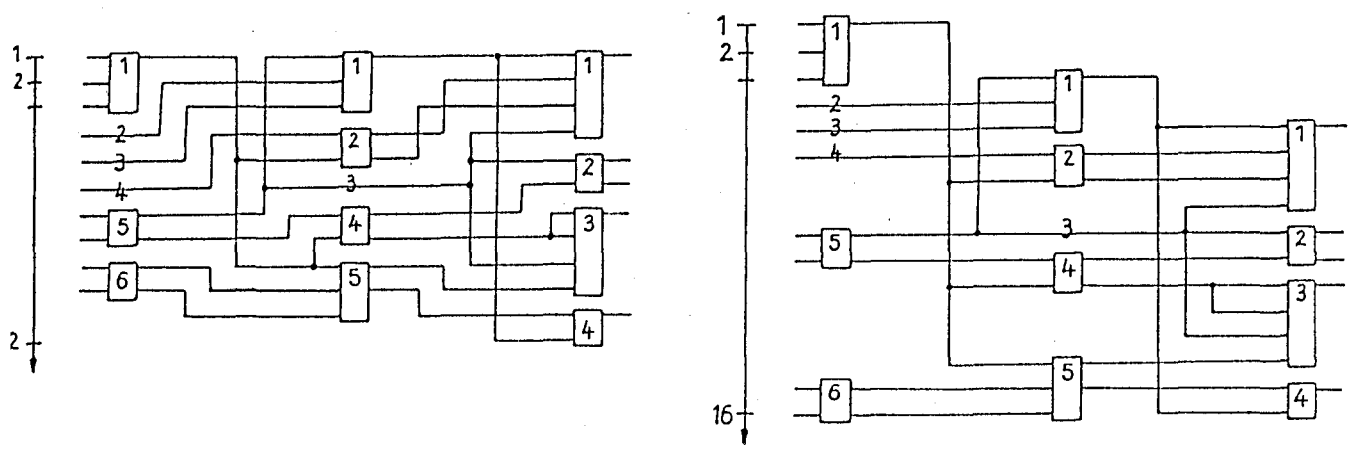

Figure 2-9: A Multirow Graph before and after Shifting

2.4. A Knowledge-Based Router

The concept of a knowledge-based schematic generator is derived from original work of Joobbani [Joob 86] on VLSI routing. It is appropriate to review his contribution.

Joobbani notes that algorithmic VLSI routers produce routings fast, but that the quality is inferior to routings done by hand. This lack of quality is blamed on the inflexibility of algorithmic routing and the difficulty of describing and using in conventional languages the many diverse kinds of knowledge the human designer brings to bear on a problem. These are precisely the problems which are addressed by expert systems.

Joobbani implements a router, the architecture of which we will discuss. The knowledge of routing is embodied in the system in the classical expert system paradigm of rules (if-conditions-then-action pairs).

The knowledge is partitioned in a number of rule sets called experts. The experts act as planners, which suggest (generate) candidate placements, and/or consultants, which criticize, modify and give priorities.

A quick rundown of the experts and their functions follows. The ones marked * will reappear in our work in appropriately modified form.

1. Wire Length Expert. Selects nets or net fragments on the basis of criteria such as the number of pins on the given side.
2. Merging Expert. Selects combinations of nets for routing on the same track with the purpose of reducing the number of tracks .
3. Congestion Expert. Observes and comments on the positioning of a net fragment with the goal of reducing maximum congestion.
4. Vertical/Horizontal Constraint Expert. Select or generate candidates on the basis of the constraint graph.*
5. Via Expert. Modifies candidates to eliminate vias by changing layer for a net segment.
6. Common Sense Expert. Suggest candidates on the basis of heuristics, if no candidate is generated by the formal system or no candidate is left after the critique.*
7. Pattern Router. Direct routing of a net on the basis of a characteristic pattern (without benefit of critique).*
8. Constraint Propagation Expert. Directly extends a net if it has only one legal move left.*
9. Minimal Rectilinear Steiner Tree Expert. Ranks candidates on the basis of pre-calculated MRST's.

The experts contain an average of 50 rules. The particular implementation language often has some rule proliferation to cover right and left-handed cases, set-up of needed intermediate values and such like; the fifty rules likely contain fewer than 10-15 ideas.

Cooperation among experts is handled by a focus-of-attention module and communication through a blackboard. The blackboard contains a representation of the state of the routing, candidates for routing and the decisions currently under consideration.

The problem representation consists of the current placement of net fragments, specifically the "moving pins", pins on the border between free tracks and the routed area.

In addition to this, the abstract representations of the net as a Steiner tree and the interactions among nets in the constraint graph make a reasoning process possible. I believe that the sophistication of the abstraction determines the power of an expert system.

A routing is built up incrementally by a move of a "moving pin" or fragment. The typical decision cycle is given below:

1. pattern router
2. wire length with consultants
    a. MRST
    b. Constraint Graph
    c. Merging
    d. Congestion
3. Constraint Graph
4. Common Sense

The constraint propagation expert is called after each move, wherever the move originated.

The total decision process is a sequence of locally good and monotone decisions. There is no backtracking or formal look-ahead. It is the function of the experts to pick a move among the candidates which does the least damage to the other nets. Note that this is a particular approach to a quadratic assignment problem.

The system is implemented in OPS5, a forward-chaining data-driven rule-based expert system shell, developed by Forgy at CMU. Its characteristic is the RETE network, a network of partially and completely instantiated condition parts of rules. It maintains information on rule satisfaction so that it does not have to be recalculated after each move, *i.e.*, if a potential move is not selected on one round, it is kept for the next round. Thus OPS5 is supposed to support the fine-grained decision process mentioned above.

However, because of the very same device the system does not switch graciously among experts. With each switch in focus instantiations have to be made and unmade, because the focus is controlled by the leading condition of a rule. This is a particular problem of OPS5.

As an aside, it is explicitly stated that some experts are used both as generators and consultants. The wording leaves it open whether one and the same rule set can be used in two ways. The fact that the generator has to operate on the problem state and the consultant on the candidate set makes this unlikely. Perhaps a small rather dumb generator provides all kinds of candidates, which are then filtered through the consultant (a sort of brainstorm and a common way of using consultants).

There are various ways of judging the merit of the WEAVER, an appropriate name for this program as the example below shows. The example is a difficult switchbox which gives many algorithmic routers problem. WEAVER finds a solution for this and some other difficult channels.

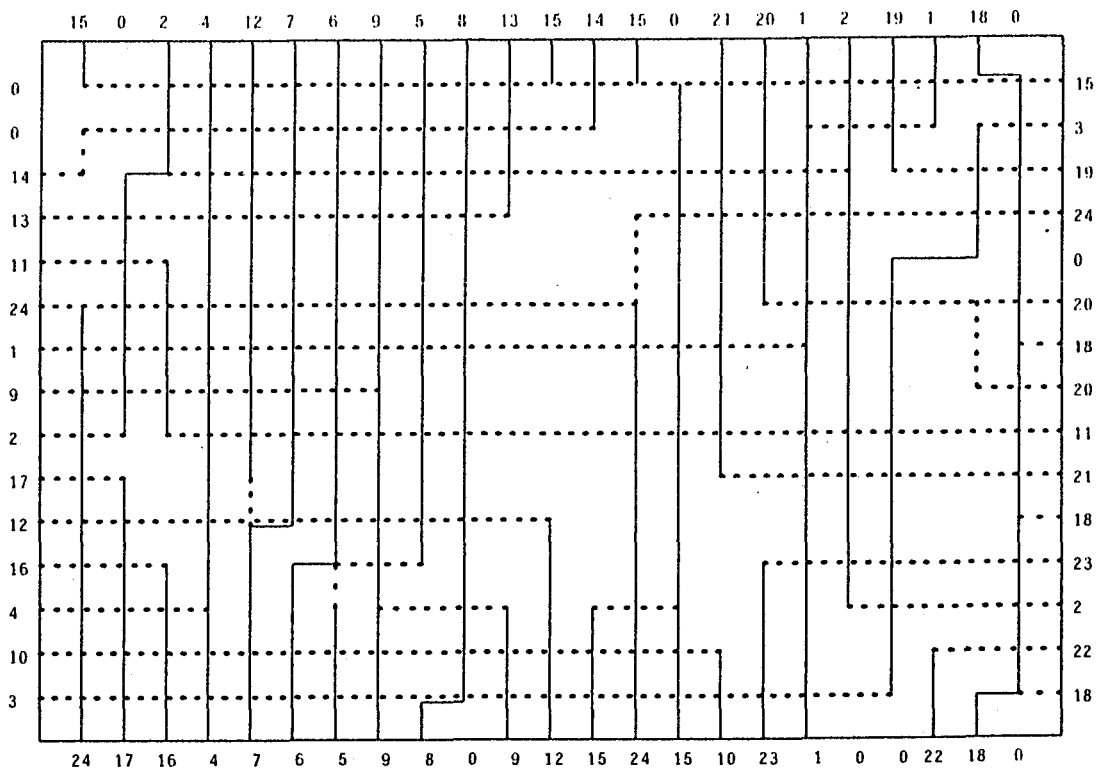

Figure 2-10: Example of Weaver: a difficult switch box

The program is reported to be very slow. One might inquire into the overhead of the RETE network. A disturbing remark of Joobbani is the need for very careful tuning of the rules (in terms of the order of the conditions and the the order of the specific tests within a condition) to gain acceptable operation. This tuning

relates to operational detail of the RETE network which is not normally visible or obvious. What is the use of working with a shell rather than a specific application program, if one has to dig back into the guts anyway?

2.5. Other Pertinent Work

Placement of the throughs, the connections between columns, is a particular problem in our system: at the time not enough information is available to note and avoid specific crossings. Thus a good heuristic is needed which does not prejudice future placement.

Rivest in his paper on the P1 System [Rivest 82] deals with this precise problem. After the division of the space between chips into channels, one is faced with determining where multi-channel nets cross from one channel to another.

A common approach is to route the channels in some favorable order. Thus the routing of one channel will determine the crossings for the adjacent channel without taking the topology of that channel into account and with potential dire consequences. In Rivest's view,"the problem is too important to leave to a lowly channel router".

Rivest introduces a crossing placer, which organizes the crossings in a global manner that maximizes the probability of a good routing on individual channels.

The crossing placer uses an iterative relaxation method where every edge is considered in turn. Crossings are moved on the basis of weighted influence of pins (closer is heavier) and the crossings facing the edge. Figure 2-11 illustrates the preferred positions induced by pins and crossings.

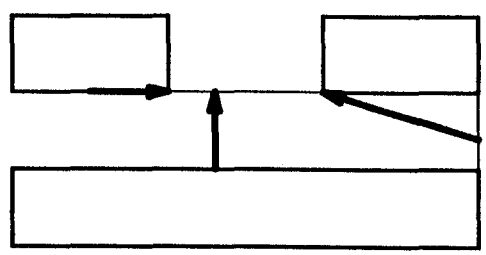

Figure 2-11: The Projection of Pins on the Divider

Rivest's global router (the step before the crossing placement) does not reach the same simplicity. It uses an explicit graph representation of the topological structure of channels, pins and a net (figure 2-12).

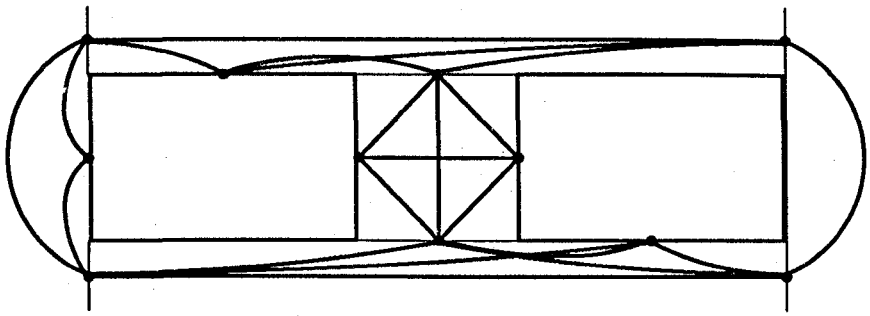

Figure 2-12: The Graph for a Five Pin Net

A weight is attached to each arc in the graph, taking into account distance plus penalties for turns and congestion on traversed edges. This latter measure makes the problem into a quadratic assignment, which seems to be not properly accounted for.

The router finds a shortest path in this graph for a two-pin net or a Steiner tree for multi-pin nets.

Chapter 3
Representation and Data Structures

The data structures which describe the representation of a schematic have two distinct aspects. On the one hand they describe (or declare) what can be represented. The design of the structure is used to limit what can be represented to what we consider desirable by some measure.

On the other hand the data structures have an operational significance in that they determine how the rules can look at, reason about and construct the components of the schematic. The rules consider features of a net in relation to other nets and the environment. The rules can be phrased parsimoniously and the rule set can become smaller to the extent that nets can be discussed in terms higher than a collection of points at such-and-such a location. Thus we need a sufficient set of abstractions beyond pins and interconnections.

These two aspects fit in well with the declarative and procedural interpretation of Prolog.

In the following sections we show a set of abstractions, which describe the objects as we want to see them. The reader must judge whether the rules constructed with these abstractions are indeed parsimonious.

3.1. Grid

The schematic is constructed by a combination of symbols and nets placed on a grid with a finite number of discrete positions (as opposed to a continuum). Clearly, reasoning about the feasibility of placing a line in a given location in a continuum requires checking an adjacent interval for the presence of other features, a feat considerably more difficult than a check for the presence of something on the very same location.

The unit spacing of the grid is chosen identical to the spacing of pins or connections on the side of functional blocks. No particular purpose seemed to be served by allowing closer spacing elsewhere.

The grid has unlimited extent. In fact during most of the construction process the reasoning is on relative positions and extent is not an issue. During the drawing of the schematic only a limited number of symbols

and interconnections can be represented with clarity on a given display area and the schematic might have to be split over a number of displays or observed through a window. That issue will be addressed in the chapter on drawing.

3.2. Symbols

For the purpose of this project a functional block has a name, a location on the grid, a vertical offset from this position and a height (which is given as the number of connection positions along the left or right edges).

For the purposes of this program a functional block is a rectangular box with connections to its left and right sides. The connections have to be at grid coordinates.

Depending on the power and the resolution of the display device, one could allow the representation of some symbol within the limits of the box as shown in figure 3-1; this would have nothing to do with the program logic. The symbol would have to include connections from the inside to the bounding box.

Figure 3-1: Block and Possible Internal Structure

The offset allows small shifts to opportunistically eliminate turns in the routing of the nets.

The system is built around the set of assumptions on the extent and the location of a symbol given in figure 3-2 right.

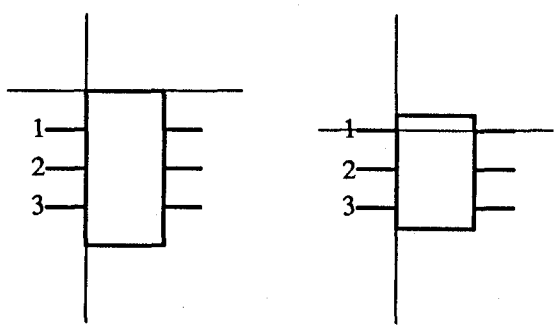

Figure 3-2: Coordinate System and Extent of a Block

This is logical arrangement for displays with control at the pixel level. The original implementation on a PC (where the schematic was drawn with graphic characters on a dot-matrix printer) required the set of assumptions of figure 3-2 left. This design choice influences the possible closeness of approach of two blocks. Moreover (in what might be the only invocation of aesthetics) the present choice is more pleasing to the eye.

Figure 3-2 defines the positive directions of the coordinate system and the numbering of pins. This choice causes the top of a symbol to have a smaller y-coordinate than the bottom; thus it is compatible with the screen coordinates of the SUN. The notions left and right refer to the location of a feature relative to the symbol and not to the channel!

The symbol or functional block could be given an attribute "function" as distinct from "name". The potential use would be in detection of flip-flops, interchange of pins, *etc.* The latter kind of facility would point at application-specific rules.

By the interchange of pins we mean a facility to interchange the connections of distinct nets to the same side of one symbol as shown in figure 3-3. The incentive would be to eliminate crossings. Such a facility could be obtained by a generalization independent of a specific function (*e.g.*, declare a group of connections to be interchangeable).

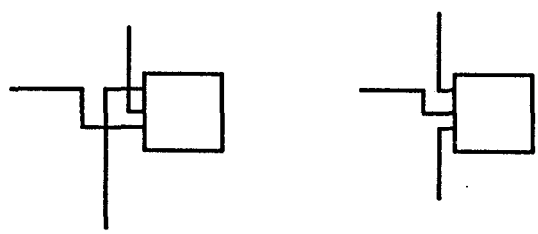

Figure 3-3: Interchanging Pins

3.3. Nets

In the present section we describe and determine explicitly the representation of nets. We have not found an equivalent to this in any of the papers. Specifically May has no control over the lay-out of a net: he has to accept whatever a channel router happens to give (except for the straightening of turns).

In routing a net for the purpose of VLSI design, the net usually becomes a Steiner tree; there is no inherent desirable position of the Steiner points with respect to the channel sides. The only criteria for merit are overall wiring length, total area, *etc.*

A schematic has to give visual clues to exhibit and highlight a feature, for example the fact that information in a given net is coming from more than one output or is going to more than one input pin. In this instance, we decided to collect outputs and distribute inputs from separate points; in other words, we locate Steiner points always on opposite sides of a channel. By example, in figure 3-4 we do not construct a net as shown on the right, but take the construct on the left.

Figure 3-4: The Representation of a Net as a Steiner Tree

Given this decision it becomes natural to collect all pins belonging to one net in one column and on one side into one structure called a fork. A fork is defined by the net it belongs to, the column and side. The major attributes are the y-coordinates of the top and bottom pin. The formal definition (in terms of Prolog) will be given in the next chapter together with the other data structures.

Forks belonging to one net but on different sides of a channel are connected by a run.

Runs were selected as another primary data structure. A run is defined by the forks which it connects and the location given by column number and the y-coordinate in that column (often called "track" in the code). If the run does not meet a fork at one end, a dummy pin is added to that fork at the given y coordinate. Given that the forks are associated with a side, a run is the horizontal line which crosses the middle of the channel.

We need some degrees of freedom in the representation of a net to fit it into the schematic. By choosing the run as a main abstraction we commit ourselves to fitting the net by adjusting the y-coordinate of the run as demonstrated by the thin lines in figure 3-5 right.

As indicated in figure 3-5 left, we could have chosen the cross fork as the main representation of the connection between two forks with a different set of alternative placements and a totally different visual effect, which is more reminiscent of channel routing.

In general one likes to use the extreme positions for either data construct because they give the minimum number of sharp turns; turns are in some cases more objectionable to the eye than crossings. Figure 3-6 demonstrates the disquieting effect of more involved representations (with more turns).

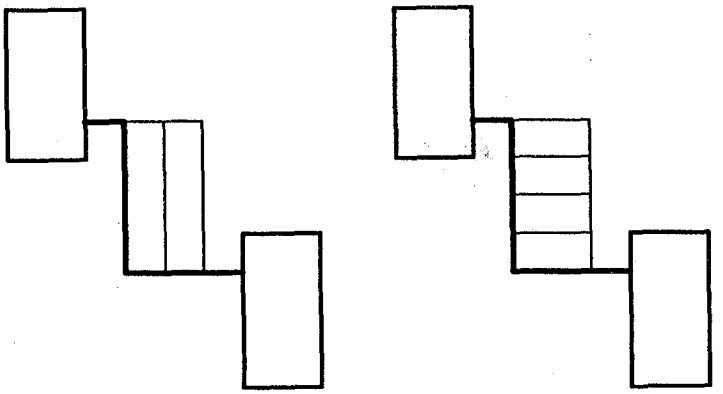

Figure 3-5: Different Layouts for Runs

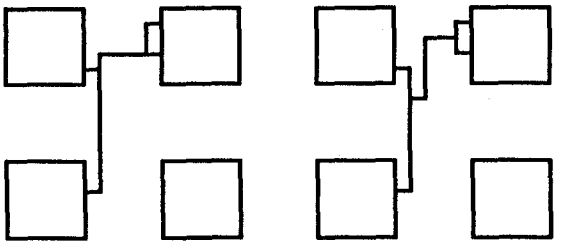

Figure 3-6: A Simple *vs.* a Complex Representation

One more concept is needed to complete the construction of all possible nets. Nets may extend across different columns and connections have to be made from column to column. These connections are referred to as "throughs". Because their end-points are aligned vertically with the symbol pins, they are treated as pins and joined to the nets.

Pins belonging to a through come in pairs, one on the left of a column, one on the right of the next column; we call them by the same unique name both as an indication of their special status and to help find sibling pins.

If a through covers more than one column, we elect to represent it as one straight line (figure 3-7). This decision precludes the treatment of throughs as objects on par with symbols for May's ordering over columns.

Figure 3-7 demonstrates another convention. The nets a and b are composed of a run, a through and a run (and some other things). In net a the runs are lined up with the through, in run b they are not. We accept the lay-out of net a as a convention: the runs adjacent to throughs are lined up with the throughs.

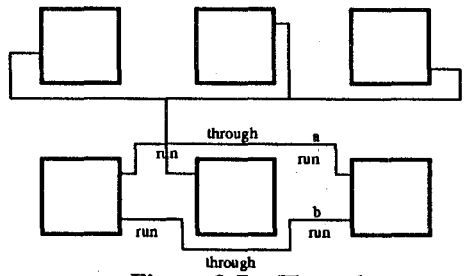

Figure 3-7: Throughs

The concepts symbol, net, pin, fork, through and run, with appropriate attributes, are sufficient for the description of a schematic with little redundancy. Figure 3-8 summarizes the terminology.

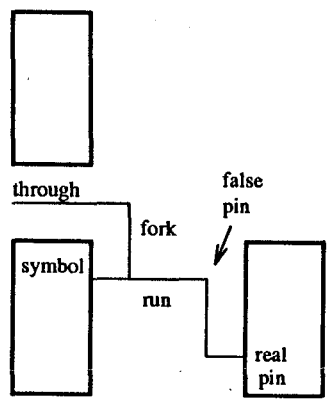

Figure 3-8: The Parts of a Net

Chapter 4
Partitioning and Intermediate Data Structures

4.1. The Need for Partitioning

"Schematic generation needs to be partitioned into subproblems to become mathematically tractable". May [May 85] made this explicit statement in the context of algorithmic schematic generation and all other authors work with it implicitly.

We want to inquire what partitioning means and whether it is still needed in the context of another programming paradigm. Partitioning is the conversion of a problem in which a number of (interacting) decisions have to be taken simultaneously, into a series of subproblems in which (groups of) decisions are made sequentially. In the strong sense of partitioning often used in VLSI design and schematic generation, the later decisions are not allowed to have an effect on the earlier ones: there is no backtracking.

The fact that the subproblems which result from any reasonable factoring of schematic generation are NP-hard is alluded to as a justification for partitioning. If two subproblems each grow exponentially with size, an attempt at simultaneous solution would result in some super-exponential behaviour.

Clearly, a switch from an algorithmic approach to some form of knowledge or rule based programming cannot change the complexity of a problem and thus the presumed reason for partitioning.

Joobbani [Joob 86] while working on a problem of the same complexity and in the paradigm of rule-based programming did not have the same need for partitioning. We have to examine why this is so.

In fact it is very difficult to imagine an undifferentiated collection of rules or declarations which allow placement of some symbols, then some pins, then some runs, *etc.* The knowledge could be embodied in two ways:

- A very complex rule can place a preferred symbol or net taking into account both global and local constraints.

- Simpler rules place symbols or nets. Some measure of quality is used to detect undesirable features, which are then removed in an iterative scheme.

In the former case the whole idea of small independent easily maintainable chunks of knowledge falls by the wayside. In the latter, searching over trees of uncontrolled depth seems to result (*e.g.*, if net a has to move to location b, we should move symbol c, which requires.....).

The critical distinction between needing or not needing partitioning seems to be homogeneity of the decisions. Joobbani deals with homogeneous decisions (one-pin moves) while schematic generation does not. We will use partitioning and whatever that entails in terms of modularizing and splitting knowledge into small identifiable packages.

The general concept of partitioning in the absence of a natural hierarchy or partition is the following. Each step works on some aspect of the whole problem. Because the result of subsequent steps is not yet known, the step works on an abstraction which models the features of interest. For example, the schematic is abstracted as a directed graph in which the nodes are the symbols and the arcs model the nets. The nodes are shuffled to minimize the number of crossings in the digraph; this should make it possible to construct nets with few crossings in the schematic. The abstractions are successful if they do not put unnecessary constraints on future placements and allow good solutions.

Each step of a partition is supported by or constructs data structures which emphasize the feature under consideration. These data structures are handed over from step to step. They are the logical points for monitoring the progress of schematic generation and for interaction (adding constraints, expressing preferences).

4.2. The Actual Partitioning

Our particular partitioning is as follows:

- Horizontal ordering of symbols into columns.
- Vertical ordering (within columns)
- Assignment of throughs
- Refinement of symbols positioning and through location
- Positioning of runs within a vertical channel
- Relative positioning of forks

- Drawing the schematic

The need for placement of symbols prior to routing is obvious. The location of the symbols and the pins demands the construction of set of paths. We cannot determine a set of paths and then hope to find a placement of symbols which satisfies the set of paths.

Nets are constructed in steps from building blocks. Of these throughs are defined as multi-column features and runs and forks as within-column features.

The positioning of the throughs leaves the ordering of the vertical channels as distinct and separate problems. This separation is of course very desirable; Rivest used precisely the same device ([Rivest 82]).

After the relative positioning of the throughs, the positioning of both throughs and symbols is refined by shifting to eliminate turns. In this process throughs and symbols are first order objects.

The order in which runs and forks are placed is a function of our representation of forks. The placement of a run determines the extent of (some) forks. The positioning of the runs imposes constraints on the ordering of the forks.

We note in passing that the placement of throughs corresponds to global routing, the shifting of symbols to improvement of the initial placement on the basis of global routing information and the placement of runs and forks to detailed routing. The fact that we have a schematic router thus allows a factoring of the problem different from [May 85] or the other authors.

4.3. Initial and Intermediate Data Structures

The initial and intermediate data structures used in the various steps are defined in the following section. Structures internal to a step or algorithm will be mentioned there.

The schematic generation can start with different sets of input data depending on the existence of prior knowledge or preferences.

4.3.1. Initial Data

Symbols or functional blocks are given by

```
node(Chip,Col,Y-coor,No)
```

in which

Chip is the unique name of the symbol

Col is the number of the column in which node is presently placed

Y-coor is the number of the row (numbered from top to bottom)

No is the number of potential connections on the side of the symbol

This latter No is not a necessary attribute of a node (as an abstraction of a real symbol). It could have been carried separately as in height_symbol(Chip,No).

Note that we are following the convention that variable names begin with a capital. An instance of a node in the database:

```
node(a,2,3,5)
```

for the node a in column 2 row 3 and a symbol with 5 pins/side.

A dummy input module is defined for column 0 as is an output module for column n+1.

Information on connectivity is given in the form

```
raw(Net,Chip,Side,No)
```

in which a raw pin is defined by

Net a unique name for the net

Chip, Side and No for the location of the pin on the chip

A collection of pins with the same net name defines a net.

Safeguards are built in to check for pins in the same location.

4.3.2. Placement of Symbols

Within placement the schematic is mapped as a digraph. The digraph models the schematic for the purpose of sorting nodes; it is defined by x_arcs and x_connections.

```
x_arc(Net,Chip1,Chip2)
```

in which an arc (or connection) is defined by

Net is the net name and the arc leads from

Chip1 to Chip2

When all nodes are placed, nodes are mapped to chips which have the definition:

```
chip(chip,col,y_coor,no)
```

which is identical to node except for

y_coor a real coordinate (as opposed to an ordinal row number)

4.3.3. Placement of Runs

At this point we have definitions for pins and forks:

```
pin(Net,Chip,Column,Side,Coor,Type)
```

in which

Net is the net the pin belongs to

Chip is the chip the pin is located on

Column is the column the chip is located on

Side is the side

Coor is a real coordinate and

Type is a real pin, through or false pin

Note that some of this information is redundant; column could be obtained through chip, coor could be computed from chip coordinate and pin number. With the many comparisons being made there is a saving in the redundancy.

```
fork(Net,Col,Side,Top,Bottom)
```

in which

Top is the coordinate of the top pin and

Bottom is the coordinate of the bottom pin.

There is no further reference to which forks and pins belong together; the information is not often used and can be regenerated.

The placement of runs generates a quite complex term for each column:

```
rl(Col,Rconstraints,Lconstraints,Runs)
```

in which

Rconstraints is a list of constraints imposed by the placement of the runs on the forks on the right hand side of the channel. The list is of the form [X-Y,W-Z,...] in which each ordered pair X-Y indicates the constraint that the fork of net X should be inside the fork of net Y.

Lconstraints is similarly defined for the left hand side

Runs is a list [X-Y,W-Z,...] in which an ordered pair X-Y means that the run of net X is located in track Y.

Note that the semantics of the list is positionally determined. The construct X-Y makes use of the Prolog operator - . It is merely a connective which would acquire an operational meaning only if we gave it one. It happens to be a convenient connective, because it is used in a number of system predicates like *keysort*. To clarify this point, we could have written the list of constraints and the list of runs in the following way with virtually no change in the program.

```
[inside(X,Y),inside(), ...]
[ run(X,Y),run(),....]
```

4.3.4. Placement of Forks

The ordering of the forks results in other complex data structures, one for each side of a column.

```
rr(Col,Side,Extents,Labeling)
```

in which

Extents is a list [(Net,Top,Bottom),(),(),....] of all forks (in that Col and on that Side). The three entries between a pair of brackets describe a fork of net Net with top Top and bottom Bottom; it is a package of information.

Labeling is a list [X-Y, ...] in which X is the label of a fork (of net) Y, which indicates how far the fork is from the side.

Note that the forks in Extents are the old forks extended with the connection to the run, if that was needed.

Chapter 5
Ordering the Symbols

5.1. Introduction

The first criterion mentioned for a good schematic is maintaining signal flow. The word "maintaining" is to be interpreted as "demonstrating" or "making apparent" the continuity of the flow. It is tacitly implied that the schematic is a digraph.

The criterion translates in practical terms into showing signal flows as strings or sequences of units and arcs laid out in one general direction. By convention this direction is left-to-right. The convention is so strong that the direction is usually not shown at all. If the (information) flow is in the other direction some strong clue must be left behind. It is desirable that the strings are as long as possible in the given topology.

The sequences have entangled tree structures[5].

If the schematic is constructed on a grid, this means that the assignment of units to columns determines the lengths of the strings. The assignment of a unit to a (vertical) position within a column determines the extent to which the sequences are entangled, *i.e.*, the number of crossings and turns.

Note that because of the requirement to demonstrate the continuity of the information flow and the desired direction we are treating horizontal and vertical positions and flows differently. We are introducing an asymmetry, which does not exist in VLSI layout.

In the following sections we will examine two approaches to placement, an iterative heuristic and an (approximation of a) graph-theoretical algorithm. The current implementation uses the latter. However an inordinate amount of the research went into the iterative heuristic. Why the pre-occupation with this approach?

---

[5]This observation was perhaps the origin of Marathe's algorithm, which builds all the trees and then tries to combine them [Mar 82].

There are three reasons, which we will work out in some detail below:

1. The placement problem can be phrased as an optimization problem, which usually are solved by iterative improvement.
2. Iterative improvement does not require the identification of cycles and the explicit selection of back edges.
3. Compactness and control of aspect ratio are easier in the heuristic scheme.

One possible statement of the objective of placement is minimum edge length which induces a low crossing number. This problem is known to be NP-hard. Solutions to an optimization of this nature are usually of an iterative nature (or some form of search with a controlled depth).

A strict precedence ordering, which precisely maintains the signal flow, introduces forward connections or forward arcs which skip over a column of symbols. This creates throughs. Throughs are more difficult to assimilate and to follow for the eye than short one-column connections. Thus the minimization of the number of throughs is desirable. Minimization of throughs may conflict with maintaining the direction of signal flow. The elimination of some forward arcs may force (a smaller number of) other arcs to become or be represented as back arcs (see figure 5-1).

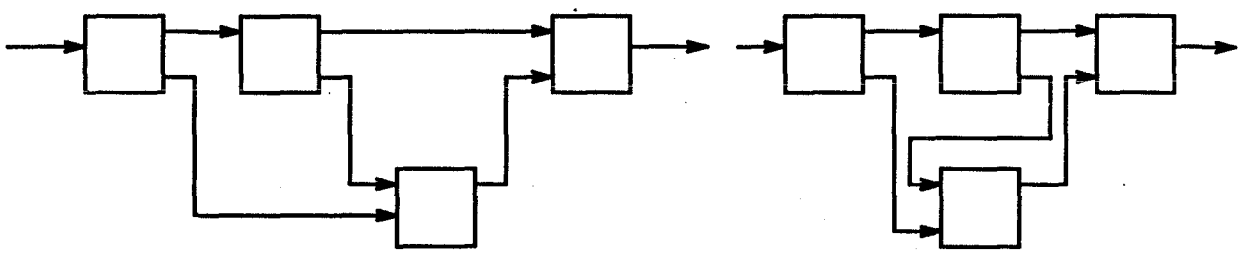

Figure 5-1: Forward Arcs and Induced Back Arcs

In fact it may be impossible to find a strict precedence order because of the existence of cycles in the schematic or flowsheet. Each cycle induces at least one back arc, an arc which goes against the left-to-right information flow. Thus back arcs arise naturally.

Graph-theoretical approaches require a formal identification of the cycles and an explicit selection of back arcs, a non-trivial problem.

Another statement of the objective of placement is the minimization of all forms of throughs (both the ones formed by forward arcs and backward arcs). Again the minimization would lead one to expect an iterative solution.

The minimization of throughs does not require formal identification of the cycles. The actual back arcs are selected implicitly.

Minimization of throughs supports the lesser goal of compactness. A schematic should use the display (screen, sheet of paper) properly. For example a long and skinny display may become difficult to comprehend because of this shape.

Both compactness and the trade-offs among forward and backward arcs can be controlled through an imposed aspect ratio. It was felt that this form of control was easy to achieve in the heuristic of the next section and would not be compatible with the rigorous graph techniques.

5.2. Simultaneous Horizontal and Vertical Placement

Figure 2-7 taken from [May 84] summarizes the contention that iterative sorts of the rows and columns in turn constitute a suitable method to obtain a placement for a general graph and a schematic.

The application requires a mapping from the nets to a graph and a metric for counting the arc length. In the first instance we tried the mapping of a multi-point net to a complete graph over the same points. Inputs and outputs are mapped as arcs to fictitious nodes. The sorting of the nodes in one row proceeds precisely as suggested by May. In the sorting of the nodes in one column we omit counting the arcs from a node to the input or output nodes; connections to an input or output node should only influence the horizontal position.

The double sort with this mapping and metric will correctly position the nodes of our main example (see figure 1-3 on page 6):

```
a    b    c    d
e    f    g    h
i    m    n    l
```

However, schematics are not necessarily rectangular; the example of figure 1-4 on page 8 is obviously displayed to most advantage in the triangular shape in which there are many empty positions in a grid. May did not touch upon the treatment of empty positions.

We developed a small routine which sorts a number of nodes with a preferred position over an equal or larger number of available positions; details are given in 5.5.

A double sort with this routine was applied to the following example:

```
        b       c       d
e       f       g       h
i       m       n       l
```

which was obtained from the main example by removal of symbol a and all its arcs. The actual case submitted to the double sort was obtained by random permutations of rows and columns.

The solution was always

```
i       b       c       d
e       f       g       h
m       n       l
```

or its vertical inverse.

This solution is inferior to the statement of the example but it is a local optimum. The sort cannot get from this solution to the proper one by sorts of rows and columns alone. We would need a diagonal move!

We argued that this was due to the fact that the mapping does not properly recognize the desired direction of the arcs, *i.e.*, it does not recognize that case a of figure 5-2 is inferior to case b.

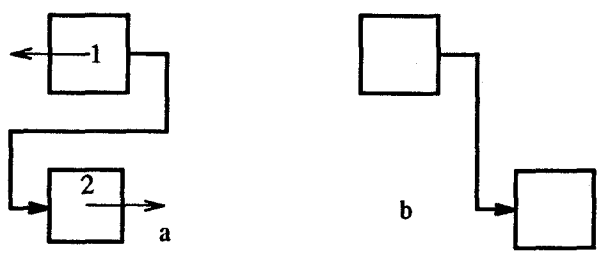

Figure 5-2: Incorrect and Correct Mapping of a Net

We changed the mapping accordingly. Thus the arc of case a would induce a pull to left on symbol 1 and a pull to the right on symbol 2, while case b would be stable. In this new mapping there is no rationale for the introduction of directed arcs between nodes on the same side of the net (input side, output side); figure 5-3 compares the mapping of a net as directed and undirected arcs. The arcs marked a of the undirected case are omitted in the directed case.

The example contains a few nets which connect two symbols on the same side and only that (for example in

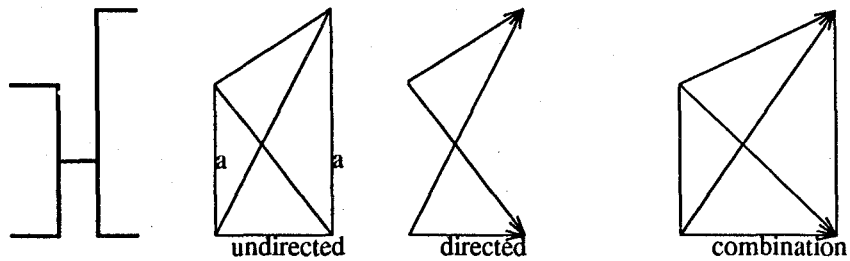

Figure 5-3: Mapping to Undirected and Directed Graph

figure 1-3 the unmarked net on the right between e and i and the unmarked net on the left of c and g). The mapping of these nets causes a particular problem. In the situation of figure 5-4, case a should induce the pull demonstrated by the arrows, but what about case b. We resolved this for the horizontal direction in an *ad hoc* manner by assigning desirable positions halfway between the two symbols. For the vertical direction one can only make the unhelpful statement that they should not be on the same track!

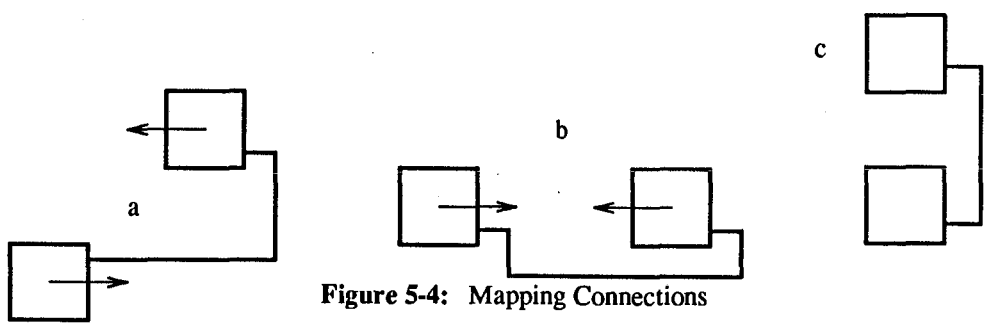

Figure 5-4: Mapping Connections

This mapping did correctly interpret the constrained case where all positions are filled. Performance on the case with the missing symbol depended on the starting position. For example the starting position

```
     h     b     d
f    i     c     e
g    n     m     l
```

resulted in the schematic of figure 5-5. This schematic should have no more throughs, turns and crossings than figure 1-3 on page 6!

A remaining question is whether the net of figure 5-3 should have been mapped as a combination of directed arcs and connections as shown in that figure.

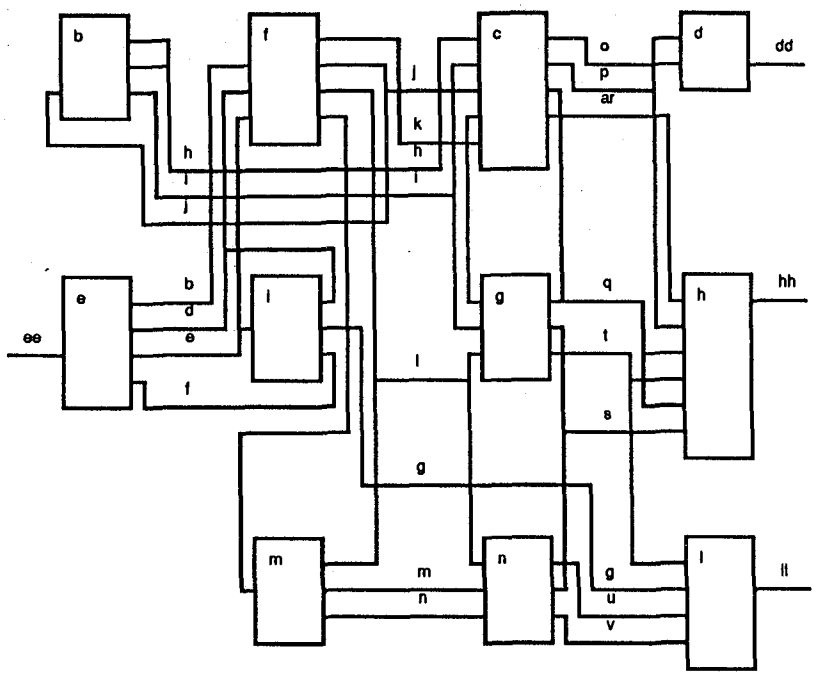

Figure 5-5: Improper Placement

We have to conclude that the heuristic of the double sort is not sufficiently powerful. The claim was of course that because it places and replaces a whole string of symbols at once, it is more powerful than -say- pairwise interchange.

The presumed reason for the failure resides in the problem being quadratic especially in the horizontal direction. The original concept was generated for the ordering of the nodes of a bipartite graph where there is no interaction amongst the nodes of one column other than the fact that two nodes cannot be in the same place.

5.3. Horizontal Ordering: Heuristic Approach

The algorithm of the previous section tried to place all symbols within a constrained space of a given aspect ratio. It failed because symbols could not interchange positions within the limitations of row and column moves. We searched for a heuristic method which would not restrict motion in this way.

A natural objective then is the minimization of the number of throughs (both forward and backward) under the constraint of a fixed distance between input and output. If we do not restrict the vertical extent of the columns, symbols on different rows could move to and from arbitrary columns and thus presumably past one another.

This type of heuristic would still not require explicit identification of cycles and feedback arcs.

The attraction of the particular objective is that the presence and extent of the throughs has to be known anyway in a later stage of the generation.

In the first instance we determine for each net the extent of the net and the nodes at either end. If a net extents precisely over one column, we define the extent as zero.

A net with extent larger than zero has a through and a movement of the node(s) at either end could eliminate a through. Of course that node is connected to other nets. Only if the node is an extreme of another net, would that net be influenced.

The influence is easiest to see in an example (see figure 5-6). Node i needs to move to the right to shorten a net c (right pull). Node i is the right end of the zero-extent net a (right stop) and the net d (left pull); both net a and d would be extended by the right move of node i (unfavorable). Node i is the left end of the zero-extent net b (left stop) and the right move of i would cause a through in the net b as well. Another quite common situation is demonstrated by node j: it is part of the left end of net c. Net c can only get shorter through a joint move of i and j (or a move at the other end).

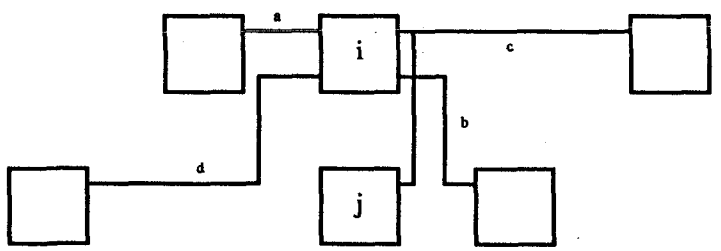

Figure 5-6: The Effect of Move of a Node on Adjacent Nets

The pull on each node can be calculated by counting the adjacent nets of a particular type (right stop, left stop, right pull, left pull). The pull to the right is the right pull minus the other three counts, *etc.*

We implemented an iterative improvement heuristic based on a greedy choice of the node to be moved. The heuristic performed well on the main example and the example with the missing chip from all starting positions.

However on the example of figure 1-4 it misplaced the group comprised of the symbols a1, b1, c1, k, l, m, n and w by one column. A quick perusal of the figure will show that these symbols are strongly connected among themselves and they have few connections to the remainder of the schematic.

This error is unacceptable and disqualifies the heuristic. The eye catches the misplacement of a group immediately, precisely because it is a group and the group is adjacent to similar groups in a different column.

The only way in which further improvement could be made in this schematic would be through the movement of groups of symbols. It is believed that the identification of such groups is a combinatorially explosive process in itself.

It is ironic that the method suffers in schematics which have a large degree of internal organization. A good clustering algorithm operating independently from the through minimization would be very useful.

The utility of the heuristic as an refinement scheme given a reasonable starting position will be discussed in the appropriate section.

5.4. Horizontal Ordering: Graph-Theoretical Approach

5.4.1. Exposition of the Problem

A unit the output of which goes to the input of another unit should be to the left of that unit. In a first approximation this suggests a realization by precedence ordering such as is used in PERT networks. Of course precedence ordering assumes that the directed graph is acyclical, which is not generally the case for our intended applications.

Some authors loosely refer to applying ordering to a "decyclisized" graph and leave it at that. Apart from the horrid word which suggests some form of dry cleaning of graphs, the implication is that one ignores a set of edges which caused the graph to be cyclical. Such a set of edges is not unique and the question becomes whether one set is as good as the other.

Other authors [Arya 85] specify the removal of a minimum set of edges which break all cycles. The idea seems to be that if a general direction of information flow is set, then back edges which run counter to that flow are bad and should be reduced to the minimum number. Ignoring for the moment that determining the minimum feedback arc set is NP-hard and that the minimum set is not necessarily unique, some examples will demonstrate that the minimum set is visually (or functionally) not always the most desirable.

Figure 5-7 gives a functional block diagram with two cycles. We already made a choice of a set of feedback edges which cut the cycles: flows 7 and 8, {7,8}.

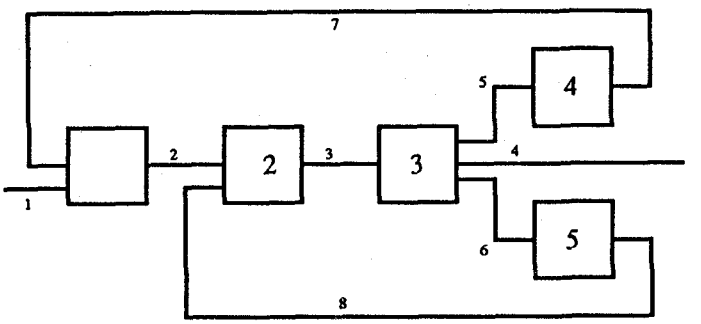

Figure 5-7: Visual Effect of Choice of Back Edges: Normal Case

To make it clear what the choice of a feedback arc set means, consider figure 5-8, in which flows 5 and 6 are the feedback flows. It seems to imply a different functionality.

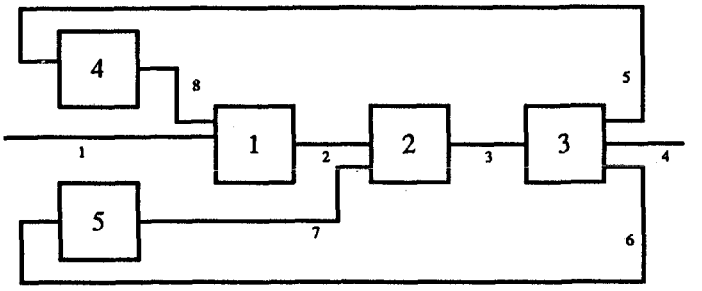

Figure 5-8: A Reasonable Mapping

We now observe that the minimum feedback arc set of the schematic under consideration is {3} with the representation of the figure 5-9.

This representation is clearly unacceptable, but the question is what makes it so in concrete terms. The worst visual aspect is the lack of direction of signal flow; the input and output are in fact overlapping. In addition, we note that the horizontal track length (counting every feedback and every extra feedforward segment as noted in figure 5-10) of figure 5-9 is larger than that of figure 5-7.

The various representations of circuits of two functional blocks and a minimum feedback arc set of size one

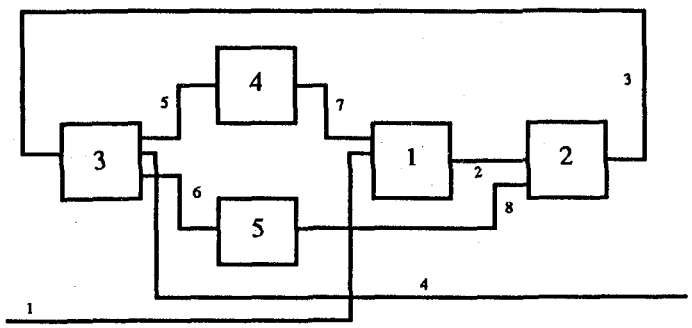

Figure 5-9: Unacceptable Mapping of Back Edges

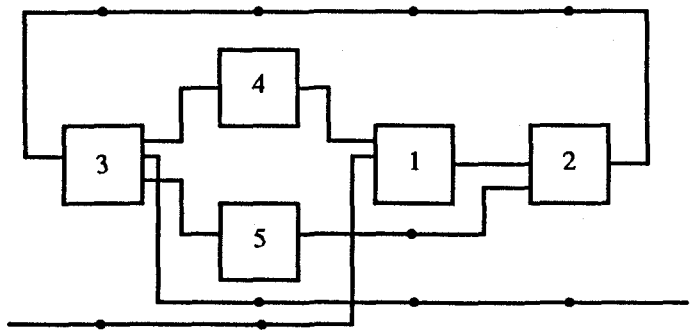

Figure 5-10: Counting Horizontal Track Length

adds to the confusion of what is the right criterion (see figure 5-11). A minimum horizontal track length criterion rather than any imposed precedence gives the accepted solution for a flip flop; this solution has two feedback arcs. Of course, the same solution applied to two non-identical units would be strange indeed. Neither would it be easy to think of a situation in which figure 5-12 would make sense.

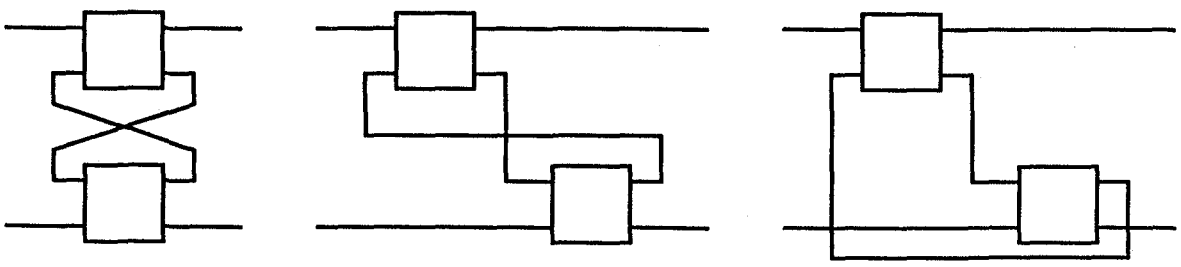

Figure 5-11: Representations of Flip Flops

A flip flop is a very special case which requires the units involved to be of precisely the same type and to have the same function in a symmetrical hook-up. In fact, in ladder diagrams circuits with identical function as a flip-flop (bistable circuits) are shown in sequence (switch and latch) rather than in parallel.

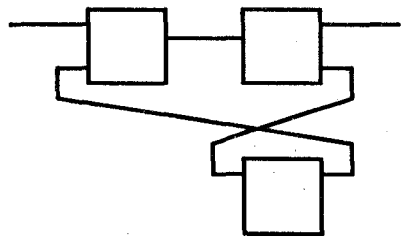

Figure 5-12: Strange Feedback Representation

Hafer in personal communication argues that a flip flop should be considered as a complex symbol. The persuasive argument is that by convention one never routes another horizontal connection between the two constituent symbols! Chun in a very recent paper [Chun 87] follows up on this contention by searching the graph for flip flops and giving them special treatment.

Note that the accepted flip flop rendering shows a symmetrical crossing. Our particular data abstraction cannot represent symmetry. The sequential rendering shows an ugly crossing which is not needed if the "through" is located on the proper side.

On the balance, it seems that the safe-guarding of signal flow is the stronger guide.

The more likely problem in schematic generation is one in which a loop can be represented in two ways and we want to force a particular choice. We would not want such a choice to be a function of the order of input data or some such accident. If we assume that the system knows nothing more than the topology, then the option of making an explicit statement of preference would be desirable (see below).

An ordering of units into columns which has long sequences can be achieved by applying the following algorithms to the digraph augmented with a dummy unit from which originate all inputs and a similar unit for the output.

1. Depth First Search finds a set of feedback arcs as far away from the origin as possible.

2. Assign columns in a process analogous to the Critical Path Method ([Even 79] p139).

The DFS of Tarjan [Tarjan 74] produces a set of feedback edges and a topological ordering of the (acyclical) graph which is left over after the removal of the back edges. This topological order is really all we need to go

to the second step; if we assign columns in the topological order we precisely ignore the back edges. Note that the set of back edges is not a minimum set. The set is not unique and depends on the order of the input data. An example of an unfortunate ordering of the data for the DFS is given in figure 5-13.

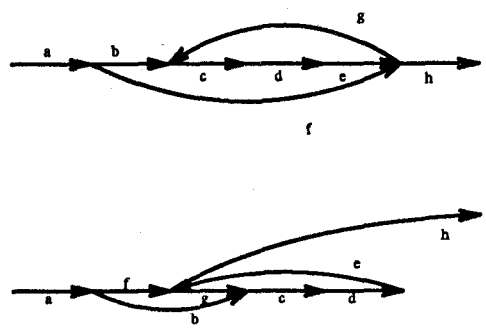

Figure 5-13: Orderings by different Sequences of the same Data

5.4.2. Remarks on Minimum Feedback Arc Sets.

It was somewhat surprising that the utility of a minimum set of back edges disappeared with one counter example. At the same time the DFS solution suggested above cannot be satisfactory on larger examples. Is there no concept of merit which can guide the search for a better order?

Some clarity can be brought to this matter by considering a related problem in the chemical engineering literature. Let the schematic of figure 5-14 represent a process. For the purpose of design or control, one may want to compute a mass or energy balance for the circuit. The complexity of the equations describing each unit is often such that simultaneous solution is impossible. Sequential solution of the equations is made impossible by the feedback edges.

The common device to get around this problem is an iterative solution with direct substitution. One assumes values for the feedback edges (which have to cover all cycles) , for example streams 4 ,5 and 6, {4,5,6}. It is now possible to calculate sequentially through the flowsheet; of course one obtains new values for the assumed values of the feedback edges. These are substituted for the old assumptions and the calculations are repeated till convergence is achieved. The feedback edges are often referred to as tear edges, the edges one has to tear to get a cycle-free calculation.

The question arises what sets of feedback edges are the best for the purpose of convergence of the calculations. For the remaining argument we need the new concept of a family of sets of feedback edges. In

figure 5-14 Upadhye and Grens [Upadhye 75] argue that the sets {4,5,6}, {1,6} and {2,6} as tear streams have identical convergence behaviour. In general any set which can be obtained from another set by the replacement of tears on all input streams of a unit by tears on all output streams, are in one equivalence class of convergence (which they call a family).

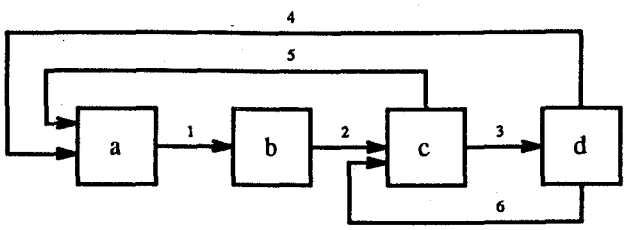

Figure 5-14: Funtional Block Diagram

Not too surprisingly a minimum set of feed back edges gives the best convergence but so does the family related to that set. The proof hinges on the demonstration that all edges in the minimum set are essential (in the sense that without a given edge in the set at least one cycle would not be cut) or non-redundant. Similarly one is guaranteed that all family (set) members have a minimum number of (edge) members. The latter statement means that if a set is derived from the minimum set by replacements then all members of that set are essential.

To extend the example, another family on the same graph is {2,3}, {2,4,6}, {3,4,5} and the bizarre member {4,4,5,6} which is obtained directly by the replacement rule. A redundancy exists in that a cycle is unnecessarily cut more than once. In complex cases the redundancy is not that obvious.

Remembering that the selection of a set of (non-redundant) back edges allows a unique precedence ordering of the remaining edges, we now equate the utility of the non-redundant family of tear sets for computational purposes with utility for representation. For the representation we want at least a member of the non-redundant family. This statement is based on the idea that feedback edges are more difficult to assimilate visually and that their number should be limited. Thus among the members of a family we have to choose one set , the edges of which are desirably located with respect to input and output or -in different words- we have to choose that set which locates the units which receive input or give output close to that side of the schematic.

This is precisely what happened in our examples. Figure 5-9 represents the schematic generated from the minimum set, figure 5-7 the one generated from a more desirable family member.

5.4.3. Computational Approach

While the above section may clarify the function of the minimum set of back edges in horizontal ordering, it is of no computational use. Researchers interested in the convergence problem are satisfied with a minimum set solution or actually any member of the non-redundant family. We would have to carry out some form of search among the members for the one with (for example) the minimum sum of input and output edges. It is unknown whether such a search has to be exhaustive (by generating the members by application of the replacement rule) or it can be guided by a metric or a heuristic (and thus generate only a few members).

A possible clue to a computational approach is contained in the following theorem from Motard and Westerberg [Motard 81]. If and only if there exists a set which tears every cycle only once, is the family the unique non-redundant family. Under exceptional circumstances (those involving cascades of feedbacks) there may be more than one family of non-redundant sets.

A recent minimum tear set algorithm based on this idea is a branch-and-bound technique with a heuristic to determine a lower bound on a given search path; it must be exponential in the number of nodes.

The authors of the algorithm (who are not computer scientists) claim the algorithm to be very efficient on the basis of experimental work on a list of examples. The particular list of several hundred flowsheets (*i.e.*, digraphs) contains common chemical processes with convergence problems. The interesting point is that only one out of the many examples had more than one non-redundant family and that the larger computational effort was associated with this example.

Would a similar observation hold for electrical or electronic circuits? If such were the case, one can almost trivially determine if a desirable set of back edges is a member of the proper family. Desirability would be determined in terms of proximity of a potential feedback edge to the input or output.

It would even be reasonable to choose a small if not minimal set from the edge/cycle matrix (for example by choosing output edges greedily from those units which have system output edges and similarly for inputs). Expressing a preference for one edge over another as a back edge would simply mean removing the non-preferred edge from the list.

Heuristics of this kind (as opposed to optimization techniques) are of course most suitable to the expert system style of this project.

Formally, the horizontal order would be determined by the following sequence of steps:

1. A DFS determines whether the graph is cyclical at all.
2. If it is cyclical, obtain a list of all cycles. A cycle is given as a list of its edges [Tarjan 73].
3. Two breadth-first searches, starting at the dummy input and output nodes and running side by side, search for edges which are part of a cycle, which do not cut a cycle a second time and which are close to the input or output. When such an edge is found for each cycle, the search for an acceptable back edge set is finished.
4. Do a precedence ordering on the graph remaining after the removal of the back edges.

It is important to realize that this sequence does not guarantee an optimal solution (unless we accept a possibly exhaustive search). The problem is in the search for an edge which cuts a cycle and does not cut another one.

An interesting by-product of the sequence relates to the problem of compaction. In a schematic with many feedforward edges we may not want to accept the full path length given by the precedence ordering. In the next section we will discuss the form of compaction demonstrated by figure 5-1. Another form of compaction is folding as demonstrated in figure 5-15.

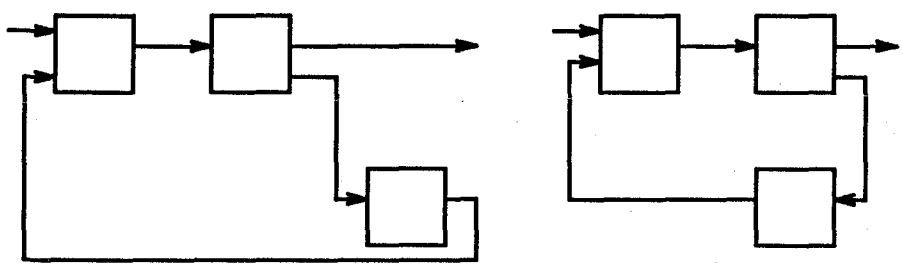

Figure 5-15: Compaction by Folding

The figure on the right is clearly more acceptable than the somewhat bizarre configuration of figure 5-12 ; it has fewer throughs as well! The mechanics are shown in figure 5-16. In a cycle (which has a direction) some node is close to the output and some node is close to the input. The candidates for folding are in the string of symbols from the output node to the input node. The actual reversal is implemented by interchanging input and output pins on the symbols of the string. Folding has more meaning in process flowsheets than in electronic or logical schematics.

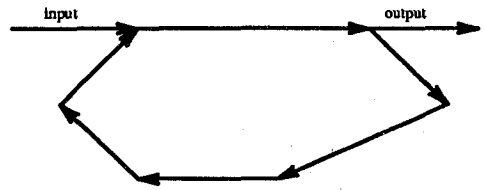

Figure 5-16: Mechanics of Folding

5.4.4. Implementation

We implemented a one-pass approximation to the sequence of the previous section, which derives from the remark there on the choice of a small but not necessarily minimal set of edges.

A breadth-first search starting at either the input node or the output establishes precedence for a number of nodes. The algorithm starts with a node without inputs and removes it and all its edges from the graph. This operation uncovers other nodes; it is characteristic of breadth-first search that these nodes are placed on a queue and that the next node for removal is taken from the front of this queue.

If at some point the queue becomes empty without all nodes of the graph having been placed we have discovered the presence of a cycle (without knowing what this cycle is or whether there is indeed only one).

We now choose a node adjacent to a placed node close to the input and with few remaining input edges and declare these input edges as back arcs. If we remove these arcs (or ignore them) the chosen node can be placed and other nodes will become uncovered, *etc.*

The code is quite typical of Prolog. The following clauses contain the essence of the modified breadth-first search. The first line (:- mode ...) indicate what is input and output in the clauses; in this case there is only one direction of use!

The first clause bfs is the terminal clause of the recursion.

The second clause treats the special case that the queue of bare nodes is empty. pick_node picks a node; its input arc(s) become back arcs.

The third clause is the workhorse. It takes the node H of the queue, extracts all arcs starting with H, marks all

nodes downstream of H with the NextLabel (and grabs the newly bare nodes in the list L), puts the newly bare nodes at the tail of the Queue and calls bfs recursively.

```
/*
bfs/5
   arg1: list of nodes to be placed
   arg2: list of arcs still to accounted for
   arg3: stack of things already bare and marked with level
      level-node pairs
   arg4: level-node pairs already placed.
   arg5: final list
*/

:- mode bfs(i,i,i,i,o) .

bfs([],_,[],Placed,Placed) .

bfs(Nodes,Arcs,[],Placed,Final) :-
   pick_node(Nodes,Label-Node),
   extractr(Y-Node,Arcs,RArcs,NArcs),
   del(Label-Node,Nodes,NNodes),
   bfs(NNodes,NArcs,[Label-Node],Placed,Final).
   /* note how the selected node is put in the queue  */
   /* NNodes stands for NewNodes and so forth        */

bfs(Nodes,Arcs,[Label-H|Queue],Placed,Final) :-
   extract(H-Y,Arcs,RArcs,NArcs),
   /* strip arcs coming from H; uncovers new nodes */
   NextLabel is Label + 1,
   mark_nodes(Nodes,RArcs,NArcs,NextLabel,L,NNodes),
   append(Queue,L,NewQueue),!,
   bfs(NNodes,NArcs,NewQueue,[Label-H|Placed],Final).
```

The heuristic undoubtedly will cut all cycles. It differs from the method in the previous section in that it does not have the help of the cycle/edge list to make the choice of back edge and thus may make a worse choice.

A breadth-first search over the nodes of the main example gave the following result:

```
a          b      c      d
e    f            g      h
i          m      n      l
```

Inspection of figure 1-3 shows that this is in fact the correct solution. The result on the example with the missing chip was as above with the symbol a removed.

The search over the example of figure 1-4 placed the symbol aa in the same column as the symbols x1, y1

and w1. This is technically correct because aa is not connected to either of these symbols. However it is not the desired result. One pass of the through minimization routine would correct this. A through location routine is needed anyway.

A small point on the treatment of the strange connections shown in figure 5-4. The only sensible treatment in the context of precedence ordering is to let them enforce placement in the same column, a graphical equal sign!

It is my conviction that no placement routine can be perfect if only because it does not understand the functionality. Thus interaction is needed. Interaction with the precedence sort is easier than with a minimization. One can add or subtract an edge with perfectly predictable results.

The precedence sort emphasizes the left-to-right signal flow at the expense of more feedforward edges. Earlier on we claimed as a unique advantage for heuristic methods that we could minimize the number of throughs and compact the schematic. It turns out that the precedence sort has a good rationale for compaction as well.

If we ignore the feedback edges the remaining graph resembles a critical path network as known in operations research. In this network we can define a critical path, a sequence of edges which determine the total distance from input to output (or the number of columns in our case). In compaction we have to remove edges from the critical path; it becomes a matter of identifying on the critical path a normal edge adjacent to forward edges. The removal of this edge would allow shortening the forward edges (see figure 5-1).

The major facility not offered by the precedence sort as opposed to the sequence of the previous section is the ability to detect opportunity for folding a cycle; that requires knowledge of the cycle structure.

5.5. Vertical Ordering

Once the symbols are ordered into columns, the sorting heuristic of [May 84] performs satisfactorily in limiting the number of crossings. Satisfactory is defined as not leaving crossings which the eye recognizes as redundant. We do not have a better (but more costly) algorithm against which to test the present algorithm.

The column-by-column sort is implemented in the following manner. All nets are mapped as directed arcs according to figure 5-3 and the connections mentioned in the context of figure 5-4.

The actual sort is performed by the predicate:

```
sort_col(Col,L) :-
        setof(Ave-Chip,Chip^ave_y(Chip,Col,Ave),L).
```

ave_y(Chip,Col,Ave) computes the coordinate where Chip should be placed by looking at all adjacent chips.

The setof built-in function conveniently sorts the Chips on the average Ave. As an aside setof needs an indication on which space to search for ave_y; the combination Chip^ indicates existensional quantification of Chip (there exist Chips such that ....).

The sorted collection L of Ave-Chip now has to be mapped to actual locations. Of course if there are as many symbols as places, this is trivial.

Mapping of symbols to a larger set of locations is carried out by the following predicate:

```
holes([List of TargetLocations and Symbols],
        [List of Locations],
        [List of Placements]).
```

All lists are ordered low to high. The first list contains desired location/symbol pairs, the third list location/symbol pairs.

The algorithm starts with the assumption that all symbols are placed consecutively in the first available positions of the list of positions. The algorithm proceeds by testing whether the whole current list of symbols can be moved advantageously one position over from the current position.

If the whole list can be moved, the list of positions is shortened by one and the routine is called recursively with the same symbol list and the shortened place list.

If a suffix cannot shift over (without incurring a penalty in added distance from the more desirable current position for that group) then that group is placed in the current position and the routine is called recursively with the remaining symbols and places. This is a recursive statement of a linear assignment.

The number of available locations is set by finding the maximum number of symbols in any one column. Sorting then proceeds starting at the column and proceeding towards the input and the output.

In the next section we will find that the placement of throughs relative to symbols and other throughs is a non-trivial problem. We gave some thought to including the throughs as one-pin symbols amongst the symbols and sorting them together with the symbols in this stage. However in the case of multi-column throughs, there is no way in the sort to guarantee that the throughs remain lined up. We originally stated that this was desirable.

Chapter 6
Locating Throughs

6.1. Statement of the Problem

If the placement of symbols creates the potential for a low crossing number, the placement of throughs defines the boundary conditions (in a literal sense) of the subproblems which settle the actual crossings. It is the last time routing is considered and can be influenced on a global level.

This step has been the subject of much research and experimentation. The final form was settled late in the project[6].

The task falls into four subtasks.

- identify throughs
- assign throughs to channel
- order throughs within a channel
- treat throughs which can justifiably be placed in more than one channel.

Identification is trivial.

Assignment can be performed easily by finding that channel which minimizes the total fork length of a net. For multi-row nets we only consider the extreme rows as options.

The ordering task constitutes the substance of this chapter.

It is not at all clear how the fourth subtask is to be fitted in. As we will see later the proper positioning of the long multi-row nets can convey good properties to the schematic. The question is when and how the choice should be made.

---

[6]The project was carried out in the inverse order of the reporting. The reason for starting at the end was to get visible output at the earliest opportunity, a proper goal in schematic generation.

One can conceive of more and more complex rules to force a through to one channel exclusively. In that case the through becomes a local feature of that channel.

Perhaps the decision on a channel can only be made after most of the ordering is done. A trial placement of a through in an existing ordering could determine the number of crossings caused by that placement in that row. Assignment would be to the row with the least number of crossings. The literature points very helpfully in both directions.

6.2. Heuristic Linear Placement

Early attempts proceeded from the misconception stated in the literature survey Section 2.5: "Not enough information is available for this step to avoid specific crossings. A good heuristic is needed which does not prejudice future placement." Thus the search was for a heuristic.

These heuristics were of the form: find a location for a through as the weighted average of the pins of the net, *etc*. Such heuristics leave ugly crossings.

Another line of research emphasized utilization of local features. Two examples of rules are:

> if a through can be located to hit one of its pins without a turn, do so.
>
> if a through can be located to make a tight turn, do so.

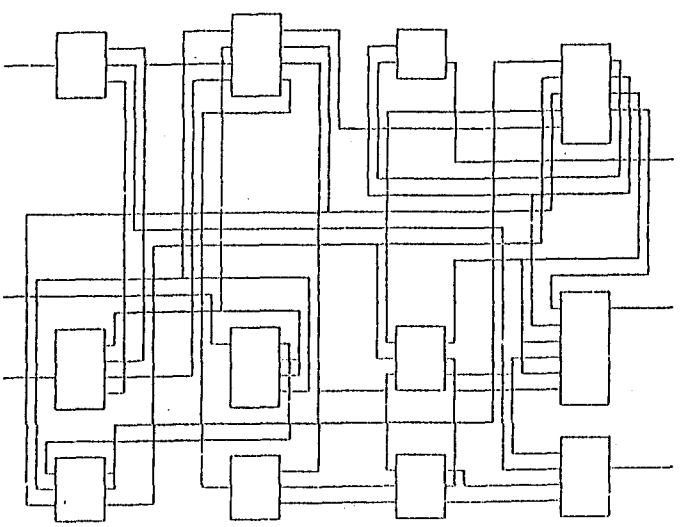

Figure 6-1: Through Placement on Local Features

These rules tried to find a favorable location for a through. For example (see the inner turn in figure 6-2) a net in a tight turn cannot get involved in a crossing! A method of this nature is used with success in later chapters. In this case it still leaves ugly crossings and these are more prominent than an adroit placement. In figure 6-1 lower left corner the short through makes a nice tight turn and so incurs two crossings on the long through. Similarly in the upper right corner, the through which is right on the proper pin incurs three extra crossings. The method does make good use of the geometry of the channel which is described by local features. It did not identify possible colinear routings (the three long nets from upper right to lower left).

This focuses the attention on the avoidance of crossings between throughs, a quadratic placement problem. The question is whether we can place forks correctly relative to one another without having access to the complete forks (which are created later). It is demonstrated in fig 6-2: can we make the correct choice amongst the three cases without the forks? It turns out that the question can be answered in the affirmative.

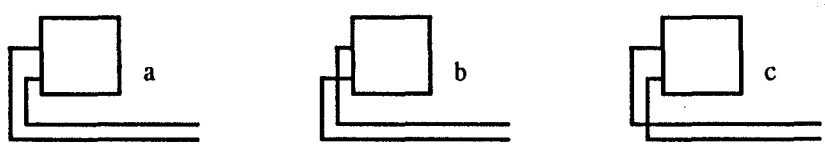

Figure 6-2: Choice among Through Positions

6.3. Quadratic Placement: Positioning Two Nets

A particular attempt at the quadratic placement of throughs is demonstrated by figure 6-3.

In the figure we give two instances of pairs of nets which have to be placed relative to one another. A pair of small circles or squares represents the endpoints of a net to be connected Manhattan-style. The bounding boxes supported by the endpoints describe the interactions among the pair. In case a one has the choice of the following combinations:

- route 1 for the circle net and route 2 for the square net
- route 3 for the circle and 4 for the square
- route 1 for the circle and 4 for the square

The first two have colinearity. The 2-3 combination leads to two crossings.

In case b the circle net is forced to position 1, regardless of where the square net goes and the square net is free.

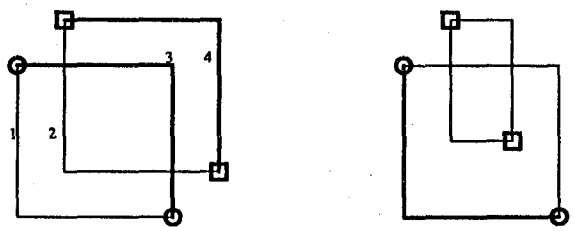

Figure 6-3: Bounding Box Model of Through Placement

A word on the specific interest in making the method of figure 6-3 succeed. Particularly in the case of a pair of nets which both cover more than one column, it has the potential of achieving the desired property of colinearity demonstrated in the left part of the figure by either the pair of light or the pair of heavy edges.

The problem with this approach is the lack of a natural way of dealing with extended forks. In addition if the endpoints of the two nets are on the same side of the same symbol, the bounding boxes coincide in that corner and degenerate cases result. There is an absolute need for a way to distinguish among such points.

A natural manner for this is suggested in figure 6-4. The pins are projected on the wall of the horizontal channel in a way which is reminiscent of figure 2-11 taken from [Rivest 82]. It is clear that this abstraction can make the correct choice in figure 6-2.

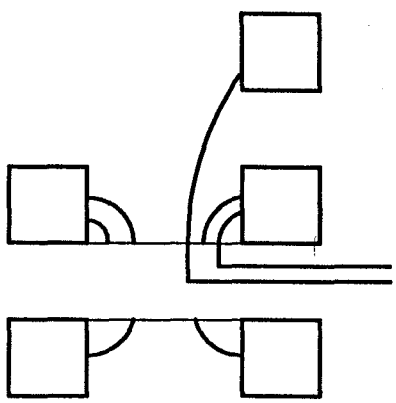

Figure 6-4: Projection of Pins on Channel Wall

Note that the relative position of the projection of a pin depends on the particular channel it is referred to. Thus a pair of throughs has to be assigned to a channel beforehand. Note also that the abstraction deals

correctly with symbols in more remote rows. The figure suggests projection of pins; the implementation uses a minimum crossings concept, which is almost projection of forks.

Given the idea of pins on the channel wall it becomes more straightforward to approach the problem in the same way as our fork placement or conventional channel routing and abandon the bounding box model.

The problem resembles the placement of forks and runs in the vertical channels. It differs from this by the restriction of the representation of a net in the vertical channel to Steiner trees. Figure 6-5 demonstrates the difference. For ease of comparison the horizontal channel is turned 90 degrees.

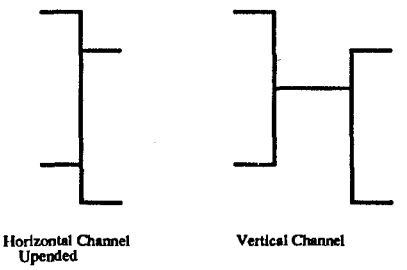

Figure 6-5: Horizontal and Vertical Channels Routing

It differs from the commonly accepted notion of channel routing in three aspects:

- common channel routing aims for minimum number of tracks
- the notion of conflicting pins is completely different
- we do not allow doglegs

If two nets have pins to opposite sides of the channel (see figure 6-6 A a), that introduces a hard constraint (A b in figure). If a prior constraint exist, the nets will be unroutable (without a dogleg) (A c in figure). In schematic generation the same situation is resolved by moving a pin (B in figure)!

The goal of placing the throughs is definitely not minimizing the number of tracks. In view of our observations above the goal seems to be the avoidance of undesirable features (such as bad crossings) and in the second instance the addition of desirable things such as colinearity, *i.e.*, it is a multiple-objective problem.

We designed a program which tries to place two throughs relative to one another. The preferences built into it are expressed in figures 6-2 and 6-7; in both figures a is the favored situation. It is important to notice that a correct placement of the throughs implies a needed relation among the forks to realize the benefit of the placement.

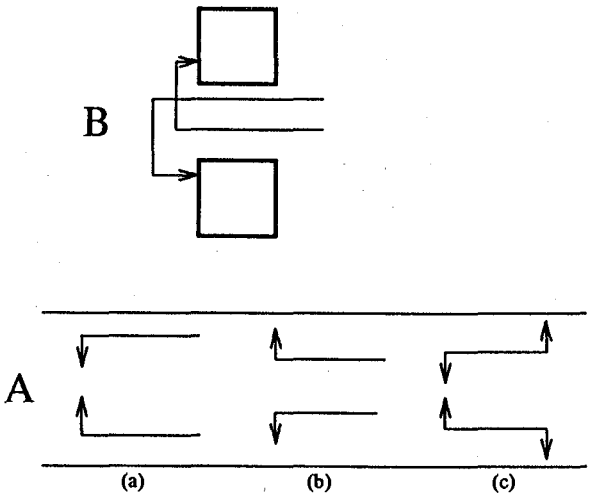

Figure 6-6: Treatment of Opposing Pins

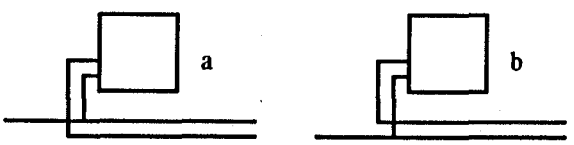

Figure 6-7: Case of One Fork Continuing

The situation of figure 6-8 is not counted in. Regardless of the placement of the throughs, one incurs one crossing (a and b in figure). Of course if one does not obey the implied fork order two crossings result (c and d in figure).

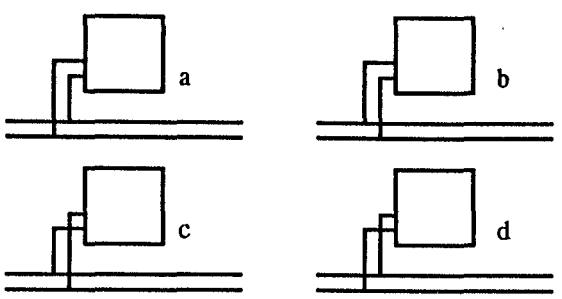

Figure 6-8: Competing Forks in the Middle of a Through

The program can deal with branches going up and down simultaneously. The case of tangled forks is dealt with as in figure 6-9: the fork with the most pins in the interval is put on the inside.

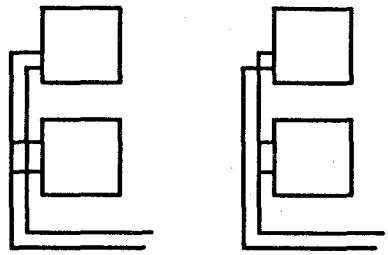

Figure 6-9: Forks with Many Interacting Pins

The right end and the left end are considered separately and the result is combined with the three-valued logic:

through x:		up:	down:	open:
through y:	up:	up	open	up
	down:	open	down	down
	open:	up	down	open

This extensive logic applied to the example of figure 6-1 left so many pairs of forks open that there was not enough information for a vertical constraint graph!

The above procedure can give a partial order on pairs of throughs based on the configuration of the pair at the extremes of the overlap. To our surprise and dismay this preference did not create enough relations for a complete ordering. A stronger partial order on pairs can be obtained with a metric such as the number of crossings incurred by a wrong placement. Before we rush to use those, a discussion of how the partial order induces a complete placement, is in order.

6.4. Quadratic Placement: Global Aspect

The expectation was that the pair-wise ordering of throughs would result in an overspecified system or, in terms of the classical paper of Yoshimura [Yoshi 82], a cyclical vertical constraint graph.

Yoshimura removes the cycles by the introduction of doglegs. We do not have this option and a cycle means taking an ugly crossing or extra crossings (in the most graceful manner).

Given the fact that we are dealing with a quadratic problem, taking some not obviously necessary crossing earlier may obviate the need for more crossings later. Classical methods such as branch-and-bound and A* search aim to locate an optimal or good design with the help of some heuristic predictor.

Such a complex technique did not seem necessary in this case. There is a strong ordering on the through pairs

as shown in figure 6-10. The small overlap on the left is more conspicuous than the one in the middle and so forth. The unnecessary two crossing in the large overlap on the right is almost invisible. Thus the inclusion of all (or many) constraints pertaining to small overlaps in favor of larger ones is a good heuristic.

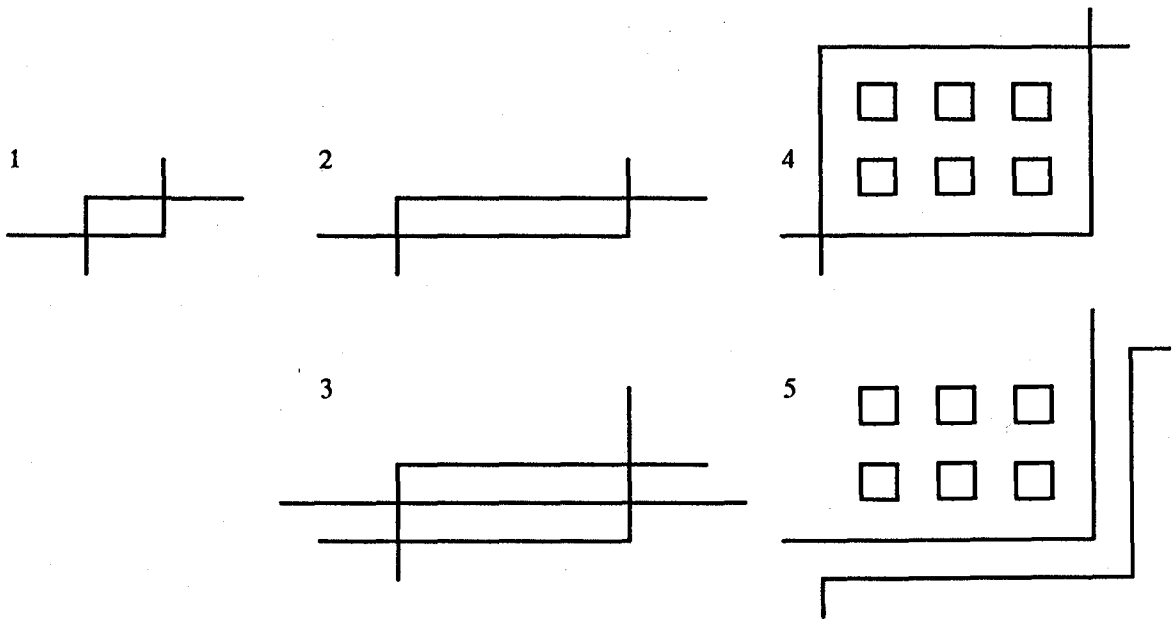

Figure 6-10: Through Crossings with Increasing Quality

Consider now an ordering on the basis of crossings. We can establish for each pair of throughs the number of crossings caused by positioning this pair correctly as opposed to positioning it incorrectly. The precedences for all pairs form a (usually) cyclical graph with weighted arcs.

The minimization of the number of crossings by partial ordering is the feedback arc set problem on the graph. The feedback arcs correspond precisely to the pair-wise precedences which we ignore to obtain a consistent partial ordering! Unfortunately the feedback arcs set problem is known to be NP complete (problem GT8 of [Garey 79]). A quick scan of the literature offered very little by way of approximations to the unattainable exact solution.

It is not even clear that minimization of the number of crossings is desirable. Figure 6-11 a shows the solution picked by crossing minimization, part b shows what I believe to be the better solution. The minimum crossing solution is very harsh.

I note that the choice of the solution in this figure and in figure 6-9 is almost a matter of taste. An expert system can enforce consistency in application and a quick way of changing the standard by changing a rule.

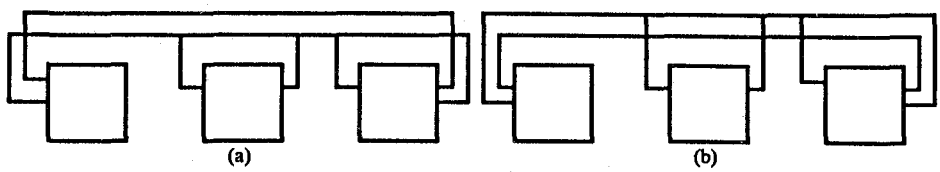

Figure 6-11: Ordering of Throughs on Crossings

Some thought was given to a heuristic placement based on local improvement of a linear placement. The linear placement could be obtained as follows: a preferred position is calculated for each through independent of where the other throughs are (center of gravity). The throughs are sorted on the preferred positions. Local improvements are computed on the basis of the endpoint criterion of the previous section. Two throughs can be swapped without any impact on the rest of the ordering. However this technique has a no reasonable way of putting two short throughs on one track if that is possible.

This latter remark explains our interest in constructing the complete vertical constraint graph. It identifies the possible location of two throughs on one track by the absence of a precedence among these two throughs.

The implementation applies ordering on the basis of preferred endpoints and number of crossings to each pair in turn. A marker can be left behind indicating which criterion was used in ordering.

The partial ordering for the graph is constructed from preferences taken from a list of preferred orientations based on endpoints and sorted by size of overlap, followed by the list based on minimum crossings. This strategy guarantees (within the bounds of the achievable with greedy techniques) a low number of first order errors. A first order error is an obnoxious feature the removal of which is obvious to the eye. It often is a pair of crossings which can be removed by a simple interchange of, for example, throughs or runs .

It is noted that the strategy is an approximation to the quadratic assignment problem. The quadratic assignment problem has exponential running time. The approximation searches for a solution over the space of all pair interactions. This space is constructed in time $O(n^2)$ in the number of throughs.

6.5. Multi-Row Throughs

In a first approximation to the placement of multi-row throughs we consider their effect on crossings. Referring to figure 6-1 which contains four obvious multi-row throughs, we note the lack of balance in the filling of the top and bottom row. This is visually quite unsettling.

However there are quantifiable reasons to strive for a balanced filling. The number of crossings is at least proportional to the square of the number of throughs. On this basis alone distribution over the tracks is desirable.

In addition the complexity of the placement is an exponential function of the number of throughs in a row. For a given computational effort the quality of the solutions will be better on balanced assignment.

This reasoning permits almost arbitrary assignment to rows. Our implementation uses this policy, but user interaction on this specific point is allowed.

The schematic can be much enhanced by a more intelligent allocation. Again taking figure 6-1 as the example three major nets run from the upper right to the lower left. Running these three in parallel would make the tracing of the corresponding nets easier. This is true even as it costs some extra crossings.

Following is a sketch for a search for nets which can be run together. This is not implemented.

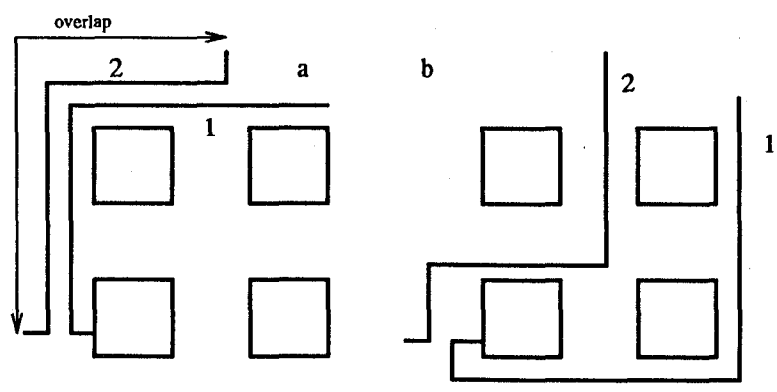

Figure 6-12: Measuring Colinearity

Two nets are compatible if they share row and track. Figure 6-12 demonstrates the concept. Net 2 shares row

and track with net 1 if both are oriented the right way (part a as opposed to b). Part b shows the colinearity of net 1 and a horizontal through; it does not have the strength of two nets sharing a turn.

A metric is needed to rank the possibilities. Length of overlap is a strong metric: its definition relies on Manhattan distance.

The search over conflicting options is preferrably carried out by some sequential process. In such a process the early colinear placement of two nets would tend to attract other nets.

The figure illustrates a problem in the search for compatible nets. The originating symbols do not have to be in the same row!

The question on when the choice of channels for multi-row throughs should be made can be answered unambiguously. The information needed to construct them is available independent of the through ordering. Multi-row throughs should be placed before through ordering so that they can be ordered with the rest.

6.6. Fork Precedences from Through Placement

The first criterion for through ordering was based on pin location relative to the channel wall. Once the throughs are placed favorably, the fork placement still has to realize the advantage of the placement.

Even when two throughs cannot be located correctly with respect to one another because of conflicting requirements at either end, then at least the fork placement at one end can be correct.

The point is that the correct fork placement for a particular relative placement of a through pair is known when the criterion for throughs is computed.

We will see in the fork placement that forks are partially ordered by rules in a recursive routine. A recursive call places a new fork subject to the constraints of the placement of previous forks.

Because of the recursive nature of the call, it would be straightforward to collect the fork precedences from the through placement and pass them to the fork placement as a preplacement.

Later on we will observe that run placements generate fork precedences as well. In fact the run placement inspects the already-established fork precedences to check that a particular run placement is possible (see

figure 8-6). So here we see a subtle way of communication of the through placement with the run placement through the fork precedences. The precedences assume the role of integrity constraints!

Chapter 7
Refinement of Placement

For the following discussion we need the concept of a track as a line on which a run or through can be routed. The tracks have a minimum distance from one another and a set of adjacent tracks make up a channel.

In schematics one has the freedom to use many more tracks than are needed to accommodate the throughs and runs. We already made use of this option by running throughs as straight lines.

The freedom to choose the number of tracks allows another form of optimization with considerable visual impact. It pertains particularly to a schematic with symbols of different sizes and numbers of pins and not so much to logic diagrams.

A compact arrangement of the throughs leaves odd spaces as in figure 7-1(a). In (b) the space is used to advantage by shifting a symbol and thus eliminating a turn.

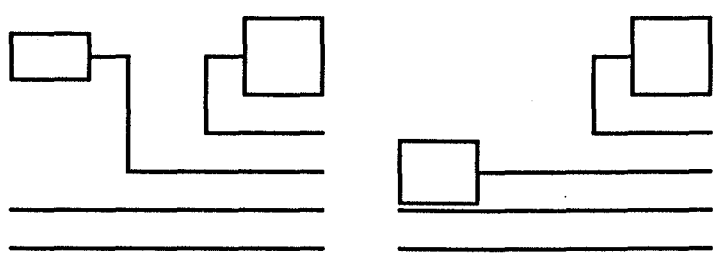

Figure 7-1: Elimination of a Turn by Shifting a Symbol

Odd spaces can be left as well by symbols. In figure 7-2 the through is moved to a gap to eliminate a turn; another symbol shift is added.

We showed both examples to make a particular point; we need the capability to move both type of objects in the same turn elimination process. The way to achieve this is demonstrated in figure 7-3. Throughs are

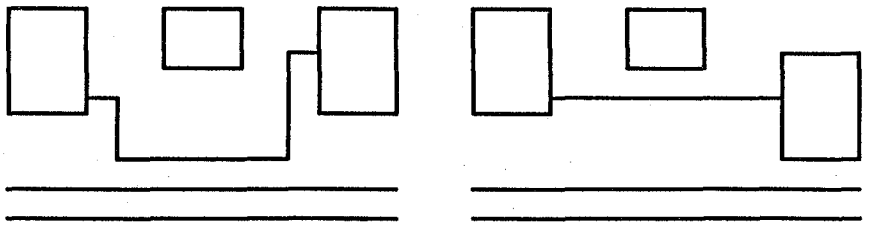

Figure 7-2: Elimination of a Turn by Shifting a Symbol

promoted to a kind of one-pin symbol (outlined with a light line) which can be treated identically to regular symbols.

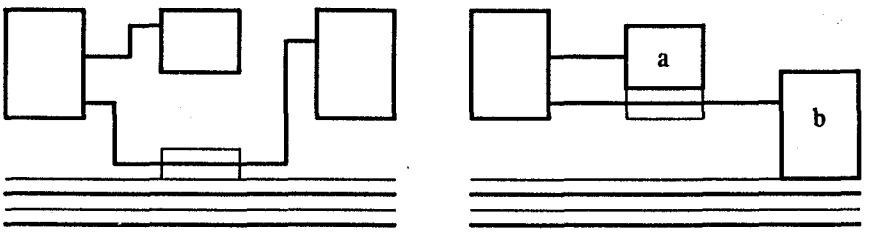

Figure 7-3: Throughs as One-Pin Symbols

A metric is needed to measure the incentive for a symbol to move. The worthwhile moves are all of the type indicated in figure 7-4. One can define the concept of a 'gap', the space between the forks on either side of a vertical channel. A gap is easy to compute.

A move has to remove the whole gap. The problem of reducing the number of turns seen as a constrained optimization problem has a rather strange objective function: a number of gaps rather than something like a sum of gaps. Moreover it is clear that the problem is quadratic. The constraints are quite normal: symbols and throughs have to stay a certain distance apart and they have a certain height.

The present implementation does not have turn elimination. A previous version used a greedy approach: move the symbol with the most incentive. This strategy does not give an optimum result. It has the clear advantage that it leaves no obvious possibilities for improvement. Obvious is taken to mean opportunity on one symbol or two neighbouring ones.

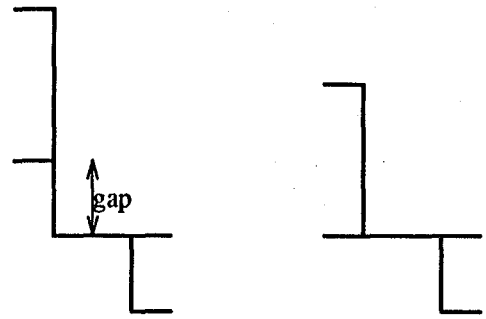

Figure 7-4: Gap as Metric for Turn Elimination

The problem is somewhat analogous to the sorting of throughs in an earlier chapter: some form of backtrack search or block moves are needed to get farther.

Chapter 8
Positioning Runs

8.1. Function in Schematic

The placement of the throughs splits the problem of schematic generation into subproblems the size of the channel between columns of symbols. We could refer to the following step as schematic channel routing.

The particular geometric constraints imposed in the chapter on design choices makes it possible to subdivide schematic routing into positioning runs and ordering forks. The fact that runs can extend forks makes it mandatory to place runs before forks.

The function of the present step is to find a feasible assignment for all runs which minimizes[7] turns and intersections. Figure 8-1 gives an example of a run placement which minimizes turns. The incorrect solution is on the left.

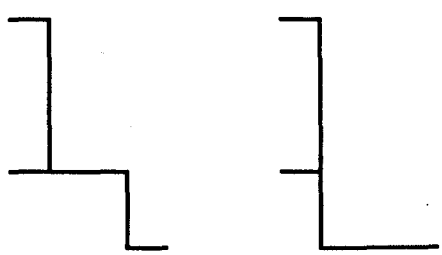

Figure 8-1: Run Placement and Turns

Figure 8-2 shows the effect of run placement on crossings and colinearity. Note that minimizing turns is a function of the net itself only; avoiding crossings depends on the placement of at least two nets, a quadratic problem.

In addition the location of the run determines which fork of the pair is extended; this can determine the level to which forks pile up on one side or another (see figure 8-3).

[7]The word is used in a loose sense. In fact we obtain an approximation.

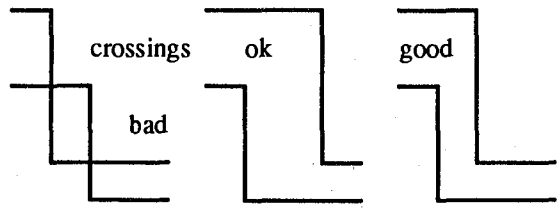

Figure 8-2: Run Placement and Crossings

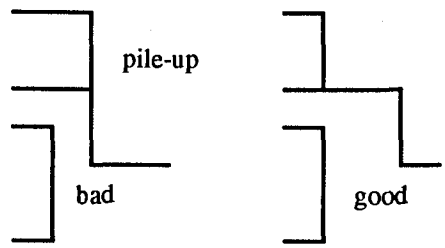

Figure 8-3: Run Placement and Pile-up

8.2. Turns

It is possible to design desirable low turn configurations for all possible combinations of two forks (see figure 8-4). In this figure one should note that placing a run on a degenerate fork instead of on the opposing fork incurs one less turn.

The sketches of this figure are a graphic representation of a set of rules. In these rules a net is described by two forks. The first of these forks is the fork drawn on the left regardless of where this fork is in the real picture. If a pattern for a pair of forks cannot be found in this list, the pair is submitted again in the inverse order.

Some cases which one would expect to be present are not because they are caught correctly by a more general case. Thus the (vertical) mirror image of 'degenerate above' is correctly interpreted after inversion by 'above'.

Some cases like 'degenerate concluded' and 'degenerate start' seem to be redundant. Unfortunately, because of the use of comparison operators, a rule and the description of its vertical mirror image have to be given. If a notion like mirror image could be included, the number of rules could be cut in half!

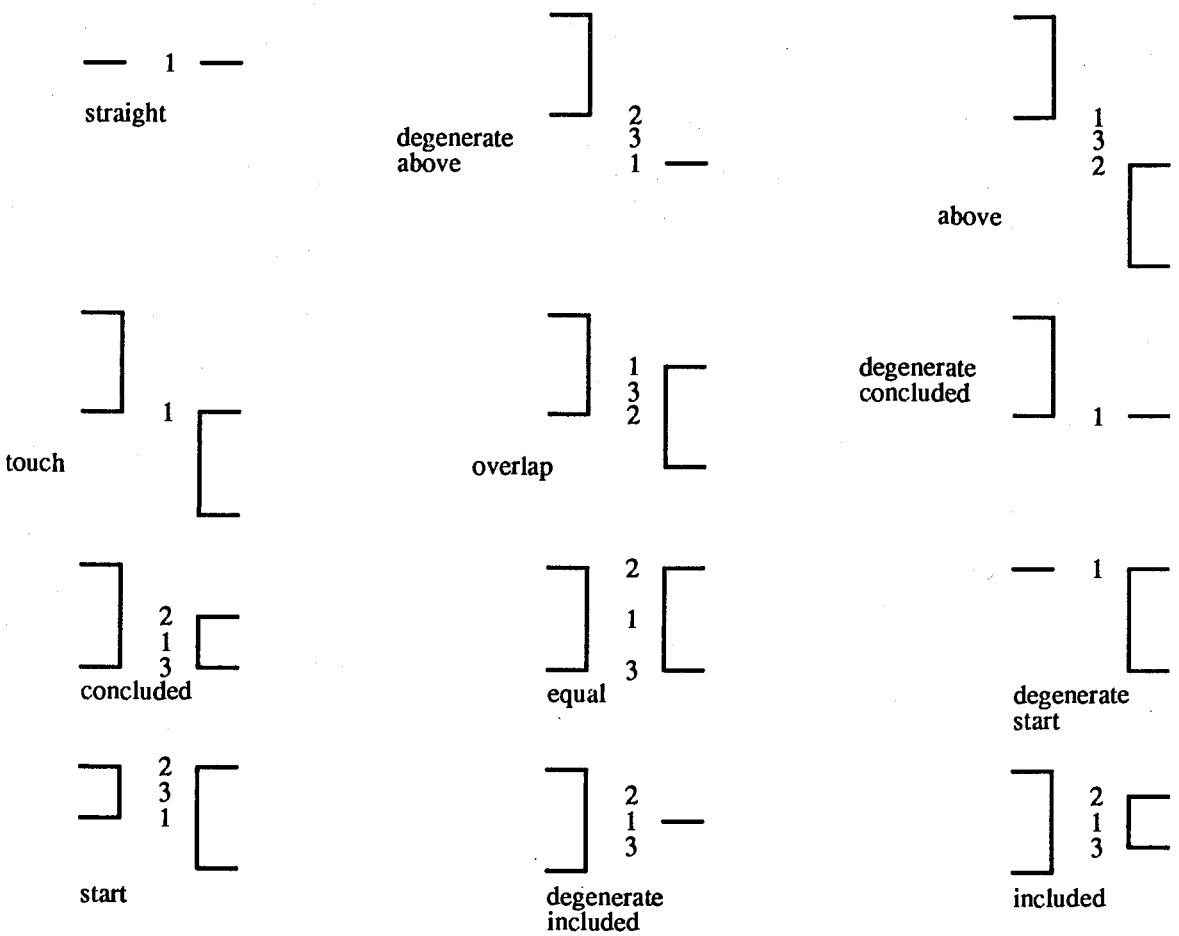

Figure 8-4: All Configurations of Nets

The placement of figure 8-4 could be improved upon, for example by putting runs preferably on pins within the given range (figure 8-5) or by ranking the members of a range in some desirable manner (symmetrically around the center).

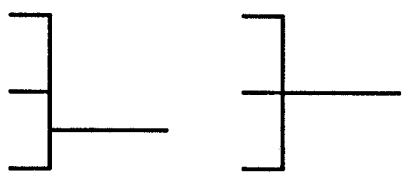

Figure 8-5: Favorable Placement of a Run in a Range

The problem is of course that we may not be able to assign the most desirable configuration to each pair (because another pair took it) and in fact we may have to settle for second best for some nets.

The assignment may be seen as a search problem in which each level in the search tree represents the assignment of a run to one of its (remaining) options. Can the assignment of forks be ranked in a manner which supports the visual impact or improves the efficiency of the search process? A heuristic which seems to achieve both is a ranking by the number of options. The code implements an algorithm of this nature.

The rational for this ranking is clear. An undesirable placement is more conspicuous on a narrow fork than on a wider fork. Given the fact that all runs have to be placed and we have to backtrack if we fail to place a run, putting the more restricted placements up front improves the search.

When a run is positioned in a particular track and there is a pin of another net on that track then that net has to stay inside the first net (figure 8-6). These constraints become known as the placement proceeds and they are needed to check subsequent placements.

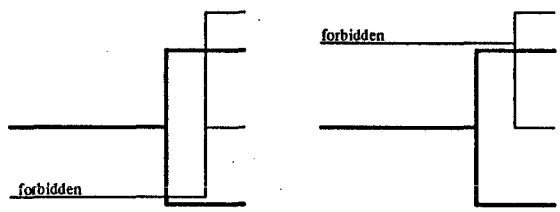

Figure 8-6: Constraints imposed by Runs

Presumably the algorithm could be made better by adjusting the ranking dynamically as runs become more restricted. Specifically if a run has only one place left, it should be placed there. This is very much in the style of the solution to Constraint Satisfaction Problems of van Hentenryck [Dincbas 87].

8.3. Implementation

The driving predicate is the following

```
test_place(Col) :-
   mark(1),  plist(Col,LL),
   place(Col,LL,[],[],[],Rconstraints,Lconstraints,Runs),
   assert(r1(Col,Rconstraints,Lconstraints,Runs)) , cut(1), fail.
```

in which **plist** generates the list of potential positions for each net and **place** does the actual positioning of the runs.

Turning our attention first to the generation of the potential positions, the key predicate shows the power of high level constructs in Prolog:

```
plist(Col,LL) :-
    setof(Ll-Net-L, (Net^cand(Col,Net,L,Ll)),LL).
```

which asks for the ordered list LL of all items of the form Ll-Net-L, where the items satisfy the predicate cand(idates). The predicate **cand** is of course designed to produce the required list for one net.

The package of information Ll-Net-L gives the potential positions for the run of Net[8] in the list L and the list L is of length Ll. The predicate **setof** uses the value of the very first item (the one in the position of Ll) to sort the items before putting them in the list LL. As stated in the previous section we use the length of the lists to guide our search, most constrained items first.

There are three versions of **cand**, one for ordinary columns and one each for the first and last column because the nets in those columns do not come in pairs.

We will consider the most general version:

```
cand(Col,Net,LL,LL1) :-
    fork(Net,Col,Side,Top,Bottom),
    fork(Net,Col,OppS,OTop,OBtm),
    Side =\= OppS,
    options(Top,Bottom,OTop,OBtm,L),
    (thru(Net,Tr) ->
        (LL = [Tr|L];
        LL = L )),
        cnt_list(LL,LL1).
```

The first three lines establish that we are dealing with a net with a fork on both sides (and pick up the information where these forks are). The middle line **options** uses this information to detect which case of figure 8-4 we are dealing with and create the list of options L. The three lines starting with **thru** constitute the Prolog if..then...else statement. If there is a through in the given Net, then put that through up front as the most important option. The final line counts the number of items in the list.

---

[8]A capitalized word in this context stands for a variable in a clause or term, *e.g.*, Net stands for the net Net.

The predicate **options** checks the forks against a logical description of the cases of figure 8-4. It then constructs a list of options (hence the name) according to the ordinal numbers in that figure. Consider one example:

```
options(Top,Btm,Otop,Otop,[Otop,Btm|L]) :-
    Top < Otop,
    Btm < Otop,
    range(Btm,Otop,L),!.
```

The rule is the 'degenerate above' case. The check whether the given request from **cand** corresponds to this case is contained partially in the head of the clause (does the second fork have a top called OTop and a bottom called OTop, *i.e.*, are top and bottom the same) and partially in the body of the clause in the form of comparisons. Testing in the head is a very fast mechanism. If the tests are passed, then **range** constructs a list L which contains the positions between the two given points. The complete list is constructed right in the head of the clause fifth position: there is a list which has OTop as the first member, Btm as the second and L as its (possibly empty) tail.

Note that the case where the top left fork is degenerate itself is included in this case. The numbering of preferred run placements came out right, so there was no point in distinguishing the case.

The complete list of clauses named **options** reflects the list of cases in figure 8-4. It has to be in that precise order, so that special cases are trapped before the following general case.

Going back to the first clause in this section and in particular

```
place(Col,LL,[],[],[],Rconstraints,Lconstraints,Runs),
```

the predicate **place** is used to place recursively and with backtracking the individual runs of the list LL. The above line calls for an instance of the predicate

```
place(Col,[Ll-Net-List|Rest],Placed,Right,Left,
            FRight,FLeft,[Net-Track|FList]) :-
    member(Track,List),
    not member(Track,Placed),
    check(Col,Net,Track,r,Right,NRight),
    check(Col,Net,Track,l,Left,NLeft),
    place(Col,Rest,[Track|Placed],NRight,NLeft,FRight,FLeft,FList).
```

The general clause requests a placement (the list in the eighth position) and any constraints FRight and FLeft

resulting from that placement, starting with the unplaced runs in the list in the second position, the runs already Placed and constraints Right and Left already generated. The two lists are being referred to in such a complex manner, because in the second position the first element is isolated for treatment and in the eighth position a newly placed run is added as a suffix to the list being handed up from the recursive calls.

The instance quoted above was the initial call of place when no runs are placed yet and no prior constraints exist (three empty lists!).

The general clause proceeds by placing one item and calling itself recursively. It takes the List of positions for the Net and takes the next available position Track (member(Track,List)). This means it takes the first track the first time and the next one on backtracking. It checks that this track is not already used (not ...) and that there are no topological problems (check....).

The **check** predicates consider whether there is a pin on the same track as the intended position of the run. If it is one of the same net or there is no pin, the run is placed at that track. If it is the pin of another net then a check is made on a prior constraint from that net to the current net. If there is no prior constraint, this net establishes a constraint on that net.

The constraints are contained in lists for the left and the right side. The predicate **inside** checks for the presence of a particular relation in those lists. **Inside** is a specialized membership predicate , which knows about transitive relations.

8.4. Crossings

Like the barking of the Hounds of Baskerville the most curious thing about quadratic assignment is that it is not there! The present implementation does not have quadratic assignment and it has not resulted in obvious eye-catching errors.

Runs cannot intersect, thus they would not be part of overt errors. However wrong run placement can lead to unnecessary crossings and affect quality. It is of interest to consider what the criteria for (an approximation to) quadratic assignment might be and how it would be implemented.

Basically we look for a way to model the necessity or avoidability of crossings.

If the pattern of the nets suggests the possibility of avoidance, it is enforced by a particular ordering of the

runs and the list of desired orderings is made part of the constraints[9] of the placement.

Each net has a range in which the run is preferably located. Only when these ranges overlap is there reason to consider interactions among runs.

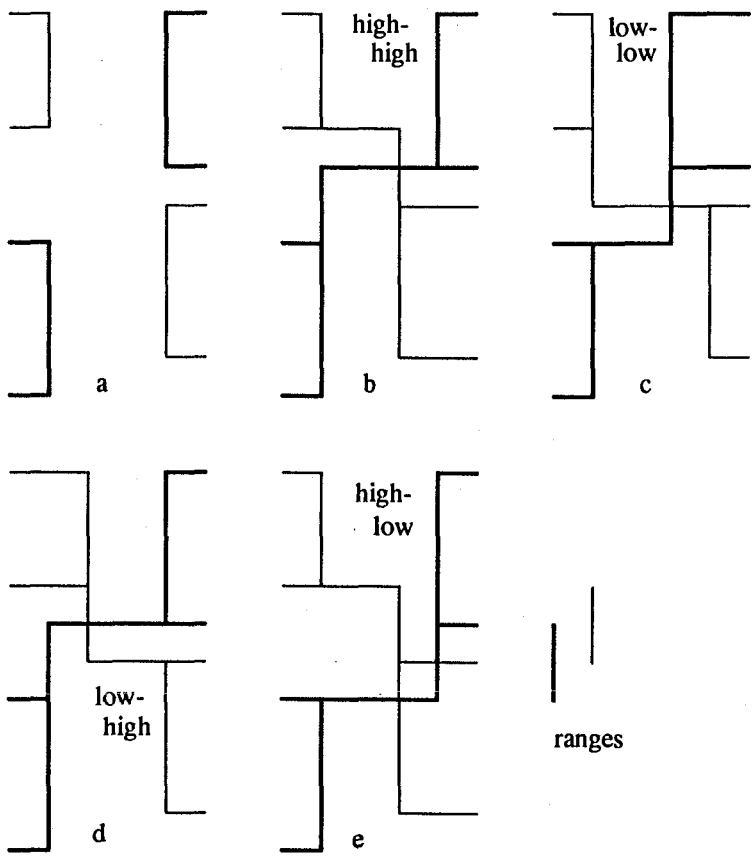

Figure 8-7: Combinations of Runs, Case I

Figure 8-7 shows an intuitively-likely heuristic which in fact did not give few crossings. The heuristic is given in the form of a rule: if the range 'a' overlaps range 'b' from above, then place the run of 'a' above the run of 'b'. Part a of the figure shows two sets of nets, one in light lines, the other in heavy lines. The light net is above the heavy net in the sense of the heuristic as demonstrated by the set of two vertical lines giving the respective ranges of the runs. Thus the run of the light net should be placed above the heavy net (this is the meaning of the caption high-high in the figure).

[9]Constraint is a convenient word. In fact we are prepared to accept the satisfaction of just a fraction of the constraints. The proper word desiderata is not well known and wishlist is descriptive but flippant.

Parts b and c show the runs placed in that relationship in their respective highest and lowest positions. The runs would be close to one another and show some colinearity. Part d shows the runs placed in the wrong relationship with no particular harmful effect! The surprise is contained in part e in which the top run is in its highest position and the bottom run in its lowest: there is an ugly intersection and it is unavoidable.

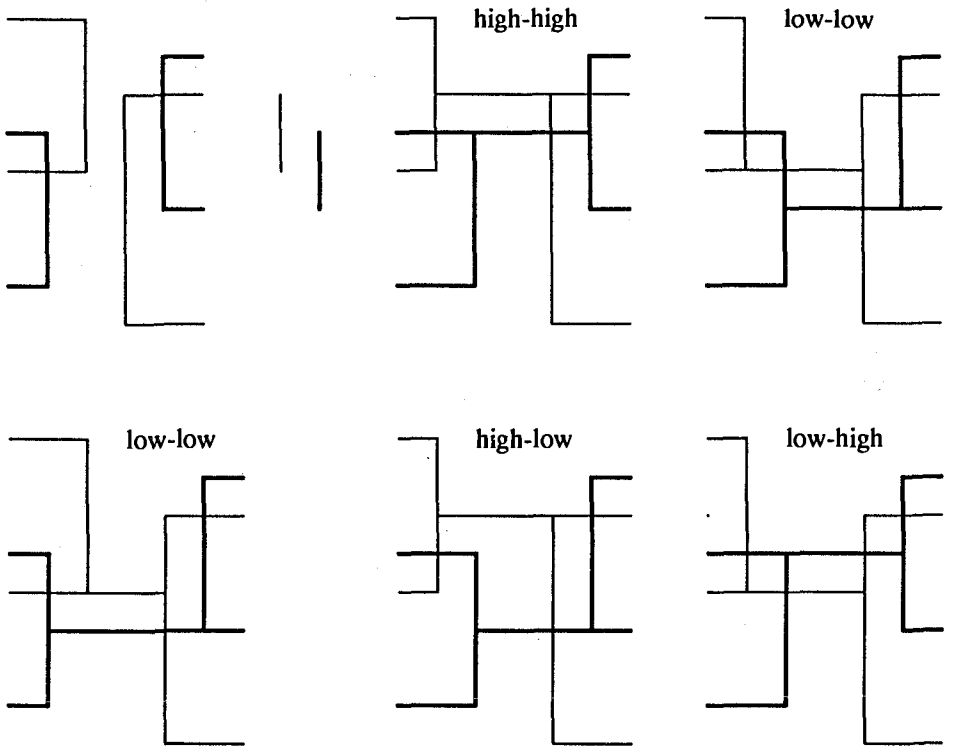

Figure 8-8: Combinations of Runs, Case II

Figure 8-8 makes the same point once more. The cases high-high, low-low and high-low give the lowest number of intersections.

There seems to be no generalization and one may have to resort to either working out all cases by hand and handing over the desirable placements in the form of a ranked list or doing a new search for each pair of nets. The latter assumes that we can count the number of crossings.

Assuming for the moment the availability of the ranked lists of choices for each pair of nets, a search for an assignment can proceed in the same way as above. We have not found a strong criterion for ordering the search. A possibility is the order of the intersections from top to bottom. In this way backtracking will be to relevant cases ([Yoshi 82]).

This line of research was not pursued because -as stated above- no particular need for quadratic placement was obvious.

Chapter 9
Ordering of Forks

9.1. Function in the Schematic

The final phase of the schematic generation deals with the remaining degrees of freedom in the representation of a net: the horizontal position of a fork relative to other forks in the same column and on the same side.

In the first instance the relative position has to obey geometric constraints of the kind demonstrated in the previous chapter: the run of one net cannot be in the same place as the pin of another net. Beyond that the relative positions can be chosen to give few crossings, control the width of the drawing as shown in figure 9-1 or serve colinearity. We have no other concept such as aesthetics.

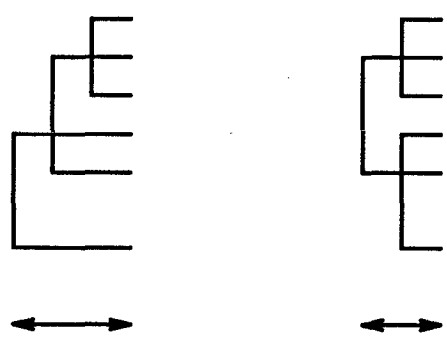

Figure 9-1: Ordering of Forks and Width

9.2. The Rules

The actual positioning is performed by rules which recognize a pattern and set a relative ordering. The rules are ordered so that more desirable and more specific rules are tried before less desirable and more general rules.

Each rule application induces an ordering on two forks, a > b. After ordering a number of pairs of forks (a > b, b > c ...) it would be possible for some rule to suggest the ordering (c > a) which is inconsistent with the

previous ordering. A check is made that the ordering remains acyclical. If a particular ordering would make the graph cyclical, the ordering is simply rejected.

The directed acyclical digraph defined by the orderings is eventually used to generate a precedence order of the forks relative to the side of a channel.

The rules are of a very simple design as shown by the example:

```
inclusion(X-Y, (Col,Side,Nets,Runs)) :-
    member((Y,Top,Btm),Nets),
    member((X,OTop,OBtm),Nets),
    OTop > Top,
    OBtm < Btm.
```

which ascertains that the fork of net X satisfies the conditions for it to be placed inside the fork of net Y as an included fork (see figure 9-2) and accepts the ordering accordingly. The actual ordering is no more than a placement of the pair X-Y on the list of ordered pairs.

In the above predicate the ordered pair X-Y is given the meaning X inside Y by definition. The symbol '-' is a PROLOG operator (which has a precedence, and association but no operational meaning); it is a convenient connective.

In this and the following code beware of > and < (because of the upside down coordinate system).

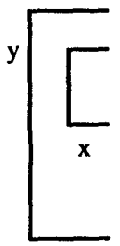

Figure 9-2: Ordering on Inclusion

The following few paragraphs show how a rule is developed and refined. The rule:

```
overlap(X-Y, (Col,Side,Nets,Runs)) :-
    member((X,Top,Btm),Nets),
    member((Y,OTop,OBtm),Nets),
    ((Top < OTop, OTop < Btm,
    member(Y-OTop,Runs));
```

```
(OTop < Top, Top  < OBtm,
member(Y-OBtm,Runs))).
```

was intended to cover the cases in figure 9-3a and b. In fact the rule gives an undesired result on the case depicted in figure 9-3c. Note that the one rule covers two cases with the ';' as an 'or' statement: case a corrsponds to the two lines starting with 'Top < OTop'.

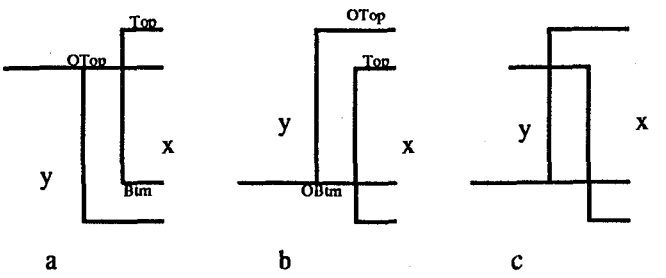

Figure 9-3: Overlap: wrong interpretation

The misguided attempt was to use weak rules and catch many cases; in fact the patterns of rules should be constrained precisely. Another reason to constrain patterns rather than rely on implicit screening by the ordering of rules is the maintainability of a rule set: interaction between rules should be limited.

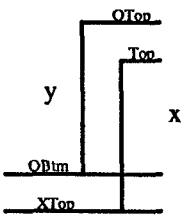

Figure 9-4: A Proper Overlap

The rule called overlap should have taken into account where the run of the other net is located as shown in figure 9-4. A corrected statement of the rule includes the extra lines which define the relative position of that run:

```
overlap(X-Y,(Col,Side,Nets,Runs)) :-
```

```
member((X,Top,Btm),Nets),
member((Y,OTop,OBtm),Nets),
((Top < OTop, OTop < Btm,
member(Y-OTop,Runs),
member(X-XTop,Runs),
XTop < OTop);
(OTop < Top, Top  < OBtm,
member(Y-OBtm,Runs),
member(X-XTop,Runs),
XTop > OBtm)).
```

Figure 9-4 corresponds to the second case in the rule. A new variable XTop is introduced so that the run of X does not have to coincide with the bottom of that fork; it just has to be below the bottom of Y.

The case of figure 9-3c is captured by the next rule

```
dummy_pin(X-Y,(Col,Side,Nets,Runs)) :-
    member((Y,Top,Btm),Nets),
    member(Y-Track,Runs),
    not pin(Y,_,Col,Side,Track,_).
```

which states that a fork with a dummy pin should be on the outside (see figure 9-5). Note the reading of this rule: X is accepted inside Y if Y has a false pin; all other interesting cases were already screened out beforehand.

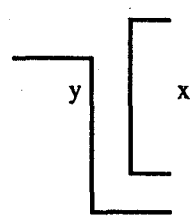

Figure 9-5: Ordering on a Dummy Pin

There is in fact a final default which accepts whatever the ordering happens to be for the remaining pairs of forks.

The ability to play with a set of rules and make incremental improvements is the strength of rule-based programming. The set of rules is an executable summary of the knowledge on the given subject. It is not too surprising that the development process leads to generalizations which catch the knowledge in a more concise form.

The generalizations may lead to a set of rules so small that programming in a conventional language becomes straight forward. All that this means is that the symbolic program helped find an (almost) algorithmic statement.

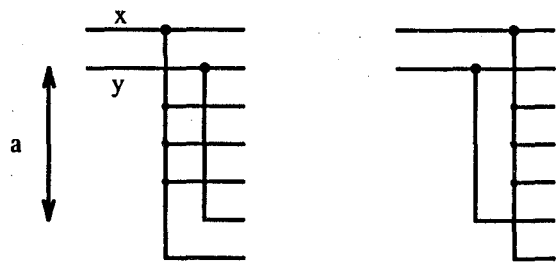

Figure 9-6: A General Rule

A generalization of this kind is a rule, which states that that fork which has the maximum number of pins in the intersection or overlap of the two forks should be on the inside. A run in the interval would be counted negative because it would intersect the other fork and thus give an incentive to move to the outside. In figure 9-6 the intersection is a. Fork x has three pins in the intersection, fork y 2 pins minus one fork. Fork y should be on the outside! This rule seems to subsume the three rules mentioned above. Experiments are to be carried out.

9.3. Implementation of Preferred Rules: an Issue in Prolog

The way in which the pairs of forks to be ordered are submitted to the rules requires some study.

Most expert system shells include some facility to rank the instantiations of various rules either by the priority of the rule or some measure of the importance of the data which instantiate the rule. For example in OPS5 rules or rule groups could be selected by context elements and data were selected by recency exclusively. Because we are working in a language we are free to design controls as desired; in fact we have to design the controls if we want anything besides depth-first left-to-right.

As a first attempt at submitting data to rules a naive and very Prolog-like structuring of the rules would be an ordered sequence

```
rule([X-Y|L],.....) :- .........,rule(L,.....) /*inclusion
rule([X-Y|L],.....) :- .........,rule(L,.....) /*overlap
```

....

in which the lines would represent the cases inclusion, overlap, dummy_pin .... A rule would consume a data item from the list and invoke a rule on the remainder of the list recursively. However if the rule fails on the pair X-Y, in this structure the pair is submitted to the rules in sequence. In short the above construct finds a rule for a pair. Given the fact that the application of a rule to a data item may influence future applications of rules to other data items (through the constraints which keep the graph acyclical) this is not the same as trying a desirable rule on all data items to be followed by the next rule.

In the next attempt we name the rules. A given rule "rule1" is invoked for each data item. If the rule1 succeeds on a data item, that item is ordered and rule1 is invoked on the remainder of the list of data. If it does not succeed, the data item is put on a list of holdovers (for the next rule) and the rule1 is tried on the remainder of the list as before. Only when the list of data is empty, is a new rule "rule2" invoked on the accumulated holdover list. We could call this 'rule-first' search as a take-off on breadth-first.

```
rule1([],H,....) :- rule2(H,[].....).
rule1([X-Y,L],H,....) :- ...success... ,rule1(L,H,.....).
rule1([X-Y,L],H,....) :- rule1(L,[X-Y|H].....).
```

In this code the second line is the case of the rule being applicable to data item X-Y, a pair of forks; the rule is invoked again on the list minus X-Y. The code which determines applicability is symbolized by the word success.

The third line represents the case of the rule not being applicable to X-Y; failure is indicated by control coming to this line at all. The data item X-Y is added to the holdover list H.

The first line is the continuation to another rule; the list of cases is empty [] and the new rule is invoked with the holdover list H. Note that the holdover position of the new call is given an empty list to begin with.

The last three paragraphs reflect the somewhat disconcerting Prolog idiom, in which order has a meaning. Thus exception and stopping conditions finish up ahead of the normal case.

While this is an efficient implementation of the concept, each real rule needs these three Prolog rules to implement the mechanism. The independent real rules are now connected through the mentioning of a following rule in the first line of Prolog. The Prolog rules are even more complex than shown because the mechanism to collect successful orderings and check for cycles is not mentioned. Such a scheme would be most difficult to maintain.

Thus I decided to isolate the mechanism of control completely from the real rule which embodies knowledge. A rule only has to answer whether a pattern holds or does not hold.

```
rule1(X-Y,...) :- ........
```

The control mechanism has an explicit list of the order in which to apply the rules (which makes it easy to add a rule, disable a rule or change the order of rules):

```
rules([rule1,rule2,.....]).
```

This is the rationale behind the hierarchy of

- applyrules(list_of_rules,list_of_cases), which invokes
- rule(rule_from_list,cases), which invokes a specific rule on a specific case with holdovers etc.

This control mechanism is applied twice in the schematic generation program (placement of throughs and ordering of forks). The current instance (forks) constructs a call to a specific rule for a specific case with the meta predicate =.. and a rule base of the form:

```
rule1(X-Y,...)
rule2(X-Y,...)
...
```

In the case of throughs we used a different form of rule base:

```
rule(rule1,X-Y,....) :- ...
rule(rule2,X-Y,....) :- ...
....
```

The call does not need to be constructed. This is the equivalent of a call with a procedure parameter in a conventional language. In most Prolog's the latter form would be more efficient.

I note that the use of recursive calls has the purpose of handing lists of cases and holdover cases to the next rule. No backtracking will occur, because there is no failure per se. If a fork pair cannot be ordered in a desirable way it is placed by default and if another track is needed, one is created. In some Prolog's these remarks could be used to improve efficiency by tail recursion optimization.

It is implied by these remarks that these concepts could have been implemented with explicit assertions of relations and retractions of the cases ordered by some rule. Such coding would be fairly obscure; it would

require total recoding if ever the need for backtracking became apparent (*i.e.*, we allow a quadratic interaction among orderings).

After similar considerations in other sections one can inquire into the need for or utility of ordering the fork pairs. I have found no metric for such ordering (size of intersection of fork pairs?) nor did I observe errors in the schematic generation which suggested a lack in that area.

9.3.1. Other Implementation Details

The real implementation contains some other procedural details, which have to be mentioned.

Up to this part of the schematic generation forks were considered in the narrow sense of extents over real pins and throughs. In the case of a net with a gap between the forks, the run always extends one fork or both. The extended forks are the ones to be ordered. The predicate extended_forks creates these real forks and collects them in a list.

Forks whose extent is one pin (skinny forks) do not need ordering and they are stripped out of the list of forks (and added back in for drawing).

Finally ordering is needed only for those forks (in a given column and side) which overlap. The predicates needed_relations check for overlap (with a brute force method which is $O(n^2)$ in the number of forks).

The ordering of the forks only creates a directed graph. This partial order still has to be translated into a precedence order. The preferred order is one in which a fork is placed close to its side of the channel.

The precedence ordering is done in two steps. In the first step a topologically sorted list is produced. This list is used in the second step to assign levels (distance from the side) to the forks. Little attention is paid to efficiency in this part. Specifically, by the addition of extra relations which position each fork between a side and the middle (*i.e.*, make each precedence graph single source single sink) the linear topological search of Tarjan would become applicable; a variant of Dykstra's algorithm for all single source shortest paths would have complexity O(relations) rather than $O(n^2)$ in the number of forks.

Chapter 10
Display of the Schematic

The present chapter is divided into two parts: one part dealing with the reduction of the schematic representation to graphic primitives such as boxes, lines and text, the other part dealing with the specifics of a particular type of display.

In displays with addressing of individual pixels, parts of a net can be drawn as they are discovered. In fact parts of a column could be drawn when they are generated!

In displays in which the graphics are imitated with a set of (graphic) characters (a fixed and limited set of groups of pixels) the output is often line-directed. Then the hierarchical drawing requires intermediate storage of characters for the eventual output as lines of characters.

Moreover in pixel displays features which fall on top of one another (crossings of forks and runs) or touch (intersection of run and fork) require no special notice. This does become an issue in character displays as shown in figure 10-1.

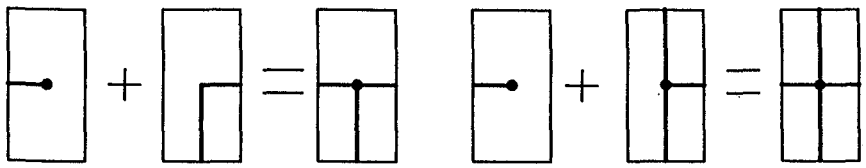

Figure 10-1: Combination of Features to form a Graphic Character

The run has to check in some manner whether there is a contact with a fork and what kind of junction or intersection it is (fork of own net, other net, false pin, *etc*).

10.1. Reduction to Graphic Primitives

There are no theoretical problems in the reduction process other than those related to specific display types.

The reduction takes place column by column. Within a column the necessary width of the channel is determined first by adding the maximum depth of forks on both sides and an extra allowance (of three spaces). Note that this is the first time that the actual horizontal extent of the schematic becomes known. Up to this point it was known only as a number of columns.

The allowance serves to emphasize the separation between forks on the left and forks on the right. Coincidentally, it provides a logical place for a run identifier.

The detailed order of the within-column reduction is discussed below. Eventually a box is drawn as a box or four lines (depending on the primitives available), forks as a combination of three lines or one path, pins as lines with a bullet at one end, runs as a line with two bullets, *etc.*

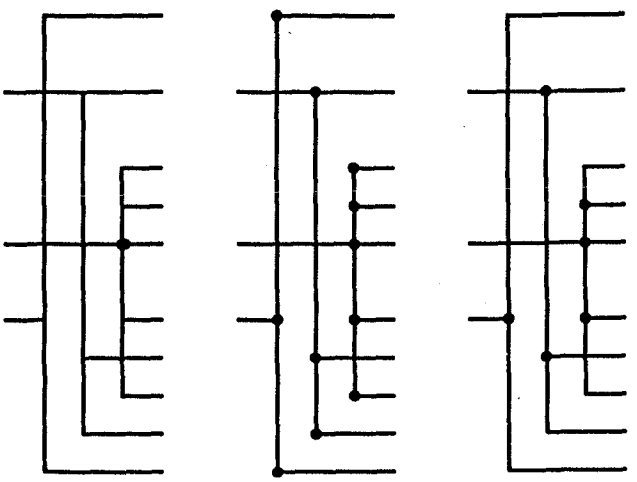

Figure 10-2: Bullets

A small point on visual syntax is the application of bullets on junctions of features (see figure 10-2). A rigorous approach would put bullets only in locations where they are needed to prevent ambiguity. Thus if a pin crosses some forks and then ends on its own fork with a "T", it belongs to that fork unambiguously. If the same pin was opposed by a run on the same track, it would not be clear which fork the run and pin belong to and a bullet is needed. The same argument holds for runs. This approach has a disquieting visual impact: it seems the eye wants to proof for each junction with a bullet that it is indeed special.

In a generous approach all junctions get bullets. However, bullets on turns are experienced as totally superfluous and disturbing.

The approach accepted is a bullet on each junction unless it is a turn.

10.2. Display in Character Graphics

A major part of the system was developed in Arity Prolog on a PC. At the time this combination only supported graphic characters both on the screen and on the printer. The (limited) screen allowed display and querying of individual characters at given coordinates; strings would of course go strictly from left to right. The printer allowed line directed output only. Thus we had the problems of intermediate storage and checking of previously determined characters.

As long as the drawings were small and fitted on the screen, the screen in fact served as the storage. When the schematic examples became larger we ran into the limitation of Prolog that it does not support arrays. An implementation of an array as simple facts in the database, binary trees and hash tables all took up too much space in memory and this became in effect the limit on the use of this version of the language[10]

In order to control the number of necessary checks the order of drawing is carefully determined: forks close to the chip before forks farther out, pins after their parent fork and forks before runs. In this way the vertical of a fork can never intersect anything, pins can intersect forks and runs can intersect forks. As well the burden of determining the type of junction (and the corresponding character) falls on the pins and runs. This particular order was maintained in the pixel graphics, simply because it was available.

In addition to the problems of overlaying and touching, the paradigm of characters introduces an asymmmetry in the treatment of essentially symmetrical features. Because the display is constructed strictly from left to right we now have to distinguish between a left pin and a right pin. This results in additional code.

For line-oriented character displays in the absence of an efficient array implementation another paradigm for drawing should be considered. A possible suggestion is a finite state automaton to construct strings of characters between two symbols (see figure 10-3). Examples of a transition would be:

---

[10]In fairness to Arity Prolog a more advanced version supports arrays implemented through a C language interface: due to budgetary constraints it was not available to us.

In a given track, if a run meets a fork of another net, then draw a crossing and remain in the run state.

If a run meets the fork of its own net and there is a pin of the same net in the same track and it is not a fork end, then draw a crossing with a bullet and go to the pin state.

If a run meets the fork of its own net and there is no pin and it is not a fork end then draw a T with a bullet and go to a blank state.

This kind of programming becomes quite tedious, but no storage of the drawing or checking for overlays would be needed.

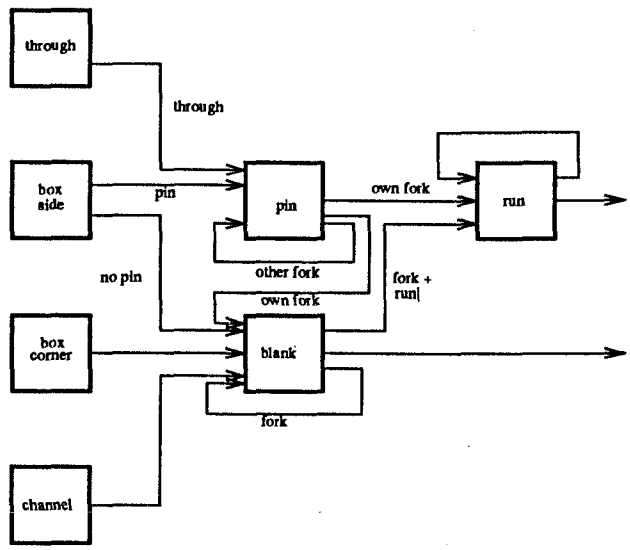

Figure 10-3: Drawing as a Finite State Machine

The issues of recognition of the type of intersection and junctions have an interest besides display. In an assessment of the quality of the display the number of crossings, junctions and turns are prime measures.

10.3. Display in Pixels

The implementation of the final drawing process in pixel displays is completely straightforward. It consists of initialization of a window or file, the actual drawing process described in the first section and the closing of the window or file.

It is noted that the need for the grid was created in part by the requirements of character graphic displays (and

in part by the need to reason over a limited set of possibilities). For pixel displays this limitation does not exist and at least in the final positioning we would be free to put a symbol at a fractional distance, for example to put it in a symmetrical position.

One version of the program writes a Postscript[11] file which can be displayed on a laser printer (the direct source of figure 1-3 and figure 1-4) or through psview under NeWS on the screen of a SUN. We note that the Postscript version is completely independent of any special features of BIM Prolog; it could be implemented in C Prolog.

Another version of the program constructs a schematic in a window under Suntools on a Sun. The two versions differ by some twenty lines of code. Part of the difference is a trivial change of coordinate system and scale factors and different names for the same primitives.

If speed of communication was a problem, objects could be handed over to the display system in a higher level form, *e.g.*, boxes instead of line segments.

[11]trademark of Adobe Systems

Chapter 11
Interface to User

I do not believe that a general schematic generator can be made perfect without at the same time becoming an unmaintainable behemoth[12]. Instead we aim for a relatively small system which gives a result in real time (*i.e.*, no more than a very few minutes) and counts on user interaction.

The central theme in the design of an interface is that the user is kept entertained with information, is aware of what goes on and is given the opportunity for interaction.

However, this interaction should be strictly controled. The heuristic which places throughs and runs produces constraints in addition to the placement itself. These constraints must be carried forward to the proper stage. Because the constraints cannot be identified one-to-one with a run, through or fork, interaction with both the placement and the constraints must be limited so as not to damage the data structures which contain them.

A general guiding principle is that the user can restrict or reorder options, *i.e.*, he can remove some positions from the list of positions available to a run (which will bring it closer to the head of the list at the same time).

The user can add restrictions but they would have to be checked for consistency with constraints already present.

It is impossible to let the user at the constraints, which protect consistency. The constraint may be many arcs deep inside a directed graph and it is not possible to answer the question of what the arc was trying to protect (aside from something in the immediate environment).

The user decides on a change of the input data or some addition or subtraction from an intermediate structure on the basis of an observed result or an inspection of intermediate data. For the latter it is important to know the order in which preferences were expressed and the rule which was active. In Prolog it is straightforward to extend a term describing a preference with an identifier of the rule.

---

[12]A schematic generator in a limited domain such as two-level logic might do very well

```
(p - q)  becomes (p - q by inclusion)
```

in which the word by is declared as a new operator!

The list in which the preferences are collected shows the required order; asserting in memory is order-preserving as well. Note that the information is in the preference list which the user does not normally see. The precedence-ordered graph cannot carry it.

The data structures of chapter 3 will need modification to make them accessible. At least the complex structure describing a whole column (runs, forks on both sides, constraints on both sides) should be taken apart for ease of access; it has no bearing on the reasoning process.

The program has attained neither the stability nor maturity which warrants a complete graphic interface with selection of objects by mouse, highlighting of nets and repositioning by dragging. For now it might be desirable to stay close to the level of Prolog, so that data structures can be examined. We have developed two ideas which work with typed-in commands.

11.1. States

The first interface is designed around the existence of named states in the schematic generation.

A state is a defined collection of data structures and the access routines. It describes the desired state of the memory after a particular successful action, *e.g.*, horizontal_order, vertical_order, throughs_placed, *etc.* Figure 11-1 clarifies what one can do in a state.

A state is the result of some (preceding) action.

Where appropriate the state can on request display characteristic information, such as the histogram-like representation of the horizontal ordering or the matrix of complete placement.

The program can be sent to the next action (*e.g.*, order_forks from runs_placed). It is important that a state is marked with its identity so that a next action command does not have to be specified.

Specific routines are provided to modify intermediate data structures with the caveats mentioned in the previous section.

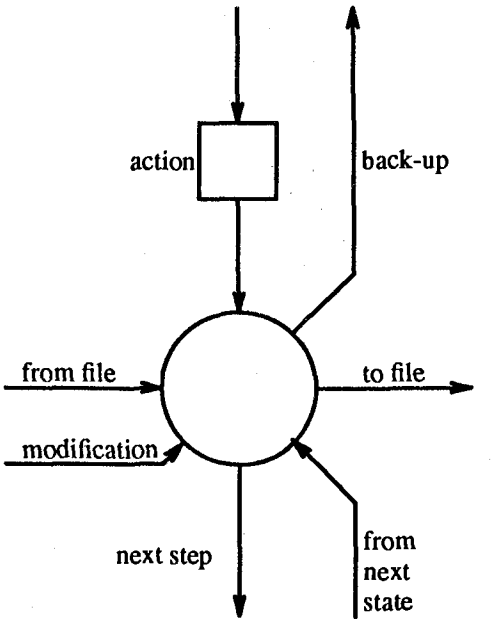

Figure 11-1: Concept of State in the Interface

The complete state can be dumped to a file so that one can keep a copy of a state while trying some modification. The complementary action is recovering a complete state from a file.

Finally one may not like the result of some action and the command clean returns to a previous state. Of course the modifications made to that state remain and would have to be undone as well.

A state can be defined in an interactive mode (as it is now). More desirably, it is all of the above and a menu program which encapsulates the access routines, enumerates command options and gives single keystroke access to them.

Figure 11-2 gives a description of the states and the transitions. In this description the states are named in capital letters. The middle column names the data which are presumed to be present in that state. The left hand column specifies interaction with the data.

The space between states gives the name of the action that leads from one state to another on the left and a corresponding 'clean' operation on the right. The 'clean' operations are implemented such that the invocation of one implies the following ones; basically one can leap over several states back to a much earlier state.

```
                                        clean_all
-------------------------------------------------------------------------
DATA
                        raw/4
-------------------------------------------------------------------------
                    prep_hor            clean_arcs
-------------------------------------------------------------------------
ARCS
                        x_arc/3
                        x_conn/3
                        nodes/1
                        nets/1
view_arcs
add_arc
add_conn
rm_arc
rm_conn
dump_arcs
-------------------------------------------------------------------------
                    bfs                 clean_hor
-------------------------------------------------------------------------
HOR
                        node/2
view_hor
change_node
dump_hor
-------------------------------------------------------------------------
                    prep_vert
                                        clean_vert
                    col_sort
-------------------------------------------------------------------------
VERT
                        node/3
                        rows/1
                        cols/1
                        max_row/2
                        max_col/1
view_graph
dump_graph
-------------------------------------------------------------------------
                  find_thrus          clean_thrus
-------------------------------------------------------------------------
THRUS
                        thru/4
change_thru
prefer_thru_over_thru
-------------------------------------------------------------------------
```

Figure 11-2: States for the Interface

```
                order_thrus        clean_thru_order
-------------------------------------------------------------------
THRUS1
                      thru/
remove_preferences
-------------------------------------------------------------------

.... placement refinement fits here

                make_forks         clean_run_options
                make_run_options
-------------------------------------------------------------------
RUNS
                      option_list
change_run_options
-------------------------------------------------------------------
                place_runs         clean_runs
-------------------------------------------------------------------
RUNS1
                      r1/4
add_fork_preferences
-------------------------------------------------------------------
                order_forks        clean_forks
-------------------------------------------------------------------
FORKS
                      rr/4
-------------------------------------------------------------------

view_forks
-------------------------------------------------------------------
                draw
```

Figure 11-2 continued

11.2. Script

The problem with the the previous interface is that modifications become invisible as the session proceeds. Thus one may have the proper schematic without remembering the particular modifications that lead to it. Dumping all the data and tracing through it is not adequate.

The Script idea gets around these objections by handing the program a series of commands in the form of a file including modifications of the data, movements of symbols or whatever. In a window environment the display and the command file are visible side by side and one can muse and ponder with all changes in clear view.

The commands remain the same as the commands for the 'state' interface. The concept of a state remains. It is only the way in which commands are entered and documented which changes. The encapsulation in menus becomes unnecessary.

The Script can be seen as a high level description of the schematic.

11.3. Points of Interaction

We now discuss some details of the user interaction with reference to figure 11-2.

The DATA state is of course reached through a file read. An extra clean_all operation is provided to re-initialize the memory.

No provision is made to change the data. That can be done by editing the raw data file.

In the ARCS state we operate on the arcs and connections of the graph. The removal of an arc means that that precedence will not be considered in the horizontal and vertical ordering: it will still be placed and drawn!

By the same token we can force a spatial relation amongst two symbols by adding arcs (forces a symbol to the following column) or conn's (forces symbols to the same column), but these will not be drawn. The strange connection which we called conn is discussed on page 56.

In the HOR state we can look at the ordering and move nodes from one column to another. In this case the responsibility for the choice of back arcs is no longer with the system.

VERT provides interchange and shifting of nodes. Again the responsibility for the number of crossings becomes the user's.

THRUS allows inspection of and changes to the assignment of all throughs to the horizontal channels. At this point as well we can add a consistent set of preferences for through ordering.

THRUS1 allows inspection,*etc.* We can remove some of the preferences induced by the through placement. Note that we will not be allowed to do this in the next state because there the constraints are hard!

RUNS allows inspection of the placement lists and changes to them.

RUNS1 allows inspection of the actual placement, storage and recovery. No changes are allowed because they would invalidate the fork constraints. If the placement is not acceptable, the change should be made in RUNS. Extra constraints for fork pairs can be added, if they are consistent with all constraints generated in Runs and Throughs.

FORKS only allows inspection, storage and recovery. It is the starting state for drawing. We note that control of pileup is achieved through changing run placement.

11.4. Addition of Rules and Program Modification

Because of the large procedural content of schematic generation and the necessary partitioning the system is best described as a procedure with embedded rule bases.

In consequence most of the program would be modified by normal procedures. An effort was made to keep the program modular for ease of maintenance.

The rules for forks and runs are written in a very stylized way, so that a minimum amount of knowledge of Prolog and the system is needed to make additions or alterations to this part.

Chapter 12
Conclusions and Suggestions for Further Work

12.1. Results

We want to answer and discuss a number of issues raised in section 1.4 in Summary and Highlights.

12.1.1. Schematic Generation in a Knowledge-Based Paradigm

The K(nowledge)B(ased) approach has to be contrasted with an algorithmic approach.

Description of knowledge in rules and patterns is very successful in local decision making such as the placement of forks and runs.

The treatment of global aspects such as placement and ordering of throughs is not done well by rules. In both instances the problem was mapped to a graph and the graph was ordered by some algorithm.

This does not take away from the fact that pattern matching was used extensively in the mapping from connectivity data to the graph (see figure 5-4) and in modifying the results.

Unquestionably the KB paradigm is very appropriate in a project of an experimental nature such as this one. It allows quick prototyping, the examination of diverse knowledge descriptions and the application of different algorithms as high level concepts. I believe this to be true even if the intended end product is algorithmic or is to be implemented in a procedural language.

12.1.2. Choice of Language

Prolog is a most appropriate language for schematic generation because it supports both the reasoning and the procedural aspects. Extensive use was made of the ability to define complex structures (to describe a fork, a net, *etc.*), pattern matching and lists. The heavy use of predicates such as **setof** and **findall** which create collections, came as a surprise to me.

Presumably an object-oriented Lisp and definitely Smalltalk could satisfy the same requirements.

An early version of the schematic generator was built in OPS5, a Lisp program variously billed as a programming language or an expert system shell. OPS5 was clumsy to use because of the lack of lists, structures and collections. The lack of control over rule precedence was particularly annoying; it forces a style of programming which mixes knowledge and controls, the very thing to avoid in KB systems.

OPS5 claims a particular advantage for data-driven reasoning through the use of the RETE network and the question is how Prolog handles such reasoning. In the given instance of schematic generation Prolog simulates data-driven reasoning through the submission of data to an ordered set of rules; it is considerably more efficient than OPS5 (10 minutes on a VAX *vs.* 5 seconds on a PC XT).

This conclusion would not hold in the environment of a large set of homogeneous decisions with continuous updating of the antecedents of partially instantiated rules. There is of course nothing that forbids implementation of a RETE network in Prolog.

12.1.3. Specific Contributions to Schematic Generation

The literature on schematic generation suggests *ad hoc* methods for placement of symbols and throughs; it is moreover intimated that exact optimal solutions can be obtained for the latter.

We show that both placement of symbols and throughs are equivalent to either quadratic assignment or the problem of obtaining a partial order on a cyclic directed graph by solving the feedback arc set problem. These latter problems fall in the class of NP-complete problems and one cannot obtain exact solutions except for small problems.

The minimum set of feedback arcs does not even give desirable placements. To describe these we borrow a concept from an ordering problem in the solution of large sets of simultaneous equations by iteration. In this problem the best solution is shown to be a member of an equivalence class of solutions with identical convergence behaviour: the same family contains the desirable choice of feedback arcs for display. We then develop an approximation to the best choice.

This is a recurring theme in schematic generation. A problem is combinatorially complex. An algorithmic approximation is made difficult by little exceptions and slight variations on the model problem. The implementation uses heuristic methods, which look for desirable patterns and avoid undesirable ones. Thus

we convert a problem with a single objective function into a problem with multiple objectives. The decision making on these is aided by casting the problem into the proper abstraction.

One can argue that such work on the structure of a problem is contrary to the concepts of expert systems, which try to capture unstructured knowledge as small independent chunks. I do not think that the advent of this style of programming absolves the scientist or engineer from attempting to state knowledge in as concise a manner as possible.

For that matter, if a diagnostic system or planning is best represented in one of the AI paradigms, it still has to address questions on completeness, termination, efficiency and other properties of programs.

12.2. Suggestions for Further Work

12.2.1. Improvements to the Implementation

The first set of suggestions deals with improvements on the current implementation. They will improve the quality of the schematic.

- Recognition of special features in horizontal ordering, *e.g.*, flip flops in logic circuits.
- Recognition of special features in vertical ordering of the symbols, *e.g.*, logical inverters in straight line with their source.
- Quadratic placement of the runs.
- The complete cycle recognition package will allow finding a possibly smaller set of back arcs.
- The implementation of the colinear placement of multi-row throughs.

The following item addresses efficiency:

- Run placement offers an opportunity for forward constraint enforcement, which will reduce backtracking in complex schematics.

The following suggestions are enhancements beyond the current capability of the program.

- The through placement can be generalized to accommodate pins on the top or bottom of the symbol
- Compaction can be carried out on the present horizontal ordering by methods known for Pert networks.
- The cycle recognition package allows folding as a means of compaction.
- For logic gates pin placement can be manipulated to minimize turns and crossings.

12.2.2. Other Issues

We rejected local improvement because our methodology is designed to avoid preventable "first-order" errors. It is still desirable to route a complete new net or add one symbol with nets to an existing schematic. Such an update will not be of the quality of a complete generation but it gives the user the benefit of stability and familiarity. Apart from routing a net in available tracks, opening up the forks or horizontal tracks could be allowed.

We did not address the issue of partitioning a large schematic. It seems that no proper method for partitioning a directed graph exists. Moreover schematic partitioning would impose some constraints on the shape of the resulting partitions (they have to be drawable in a rectangle).

The issue of partitioning showed up in a different guise in some of the iterative improvement schemes for horizontal ordering. A knowledge of logical clusters independent of the improvement routine, could have made such methods feasible.

References

[Aoudja 86] Aoudja, F.
CASE:automatic generation of electric diagrams.
Computer-Aided Design 18(7):356-60, Sept, 1986.

[Arya 85] Arya A., *et.al.*.
Automatic Generation of Digital System Schematic Diagrams.
In *22nd Design Automation Conference Proceedings*, pages 388-395. ACM SIGDA, IEEE Computer Society-DATC, June, 1985.

[Chun 87] Chun, R.K., Chag, K-J. and McNamee, L.P.
Vision: VHDL Induced Schematic Imaging on Net Lists.
In *24th Design Automation Conference Proceedings*, pages 436-442. ACM SIGDA, IEEE Computer Society-DATC, June, 1987.

[Dincbas 87] VanHentenryck, P. & Dincbas, M.
Forward Checking in Logic Programming.
In *Conf Logic Prog, Melbourne* , pages 674-680. 1987.

[Even 79] Even, Shimon.
Graph Algorithms.
Computer Science Press, Potomac,Md, 1979.

[Garey 79] Garey, M.R. & Johnson, D.S.
Computers and Intractability, A Guide to the Study of NP-Completeness.
Freeman, 1979.

[Garey 83] Garey, M.R. & Johnson, D.S.
Crossing Number is NPC .
SIAM Journal Algorithms in Discrete Mathematics 4:312-316, 1983.

[Heller 82] Heller W.R. *et.al.*
The Planar Package Planner for System Designers.
In *19th Design Automation Conference Proceedings*, pages 253-260. ACM SIGDA, IEEE Computer Society-DATC, June, 1982.

[Joob 85] Joobbani, R. *et.al.* .
WEAVER, A Knowledge-Based Routing Expert.
In *22nd Design Automation Conference Proceedings*, pages 266-272. ACM SIGDA, IEEE Computer Society-DATC, June, 1985.

[Joob 86] Joobbani R.
An Artificial Intelligence Approach to VLSI Routing.
Kluwer Academic Publ., 1986.

[Ker 70] Kernighan B.W. & Lin S.
An Efficient Heuristic Procedure for Partitioning Graphs.
Bell Sys Tech Jrnl :291-307, Feb, 1970.

[Mar 82] Marathe, S.S. & Joshi, R.R.
A Placement Algorithm for Logic Schematics.
In *CAD 82: 5th Int Conf & Exh on Comp in Design Engg.* 1982.

[May 83] May, M. *et.al.*,
Placement and routing for logic schematics.
Computer Aided Design 15(3):89-101, May, 1983.

[May 84] May, M. & Mennecke, P.
Layout of Schematic Drawings.
Syst. Anal. Model. Simul. 1(4):307-38, 1984.

[May 85] May, M.
Computer-generated multi-row schematics.
Computer- Aided Design 17(1):25-29, Jan, 1985.

[Motard 81] Motard, R.L. & Westerberg, A.W.
Exclusive Tear Sets for Flowsheets.
AIChE Journal 27(5):725-732, Sept, 1981.

[Nilsson 80] Nilsson, N.J.
Principles of Artificial Intelligence.
Tioga Publ Co, Palo Alto, 1980.

[Rivest 82] Rivest, R.L.
The P1 placement and Interconnect System.
In *19th Design Automation Conference Proceedings*, pages 475-481. ACM SIGDA, IEEE Computer Society-DATC, June, 1982.

[Shing 86] Shing, M.T. & Hu, T.C.
Computational Complexity of Layout Problems.
Advances in CAD for VLSI. Volume 4.*Layout Design and Verification.*
North-Holland, Amsterdam, 1986, Chapter 8.

[Tarjan 73] Tarjan, R.E.
Enumeration of the Elementary Circuits of a Graph.
SIAM J Comput 2(3):211-216, Sept, 1973.

[Tarjan 74] Tarjan, R.E.
Finding Dominators in Directed Graphs.
SIAM J Comput 3(1):62-89, 1974.

[Upadhye 75] Upadhye, R.S. & Grens, E.A.
Selection of Decompositions for Chemical Process Simulation.
AIChE Journal 21(1):136-143, Jan, 1975.

[Yoshi 82] Yoshimura, T. & Kuh, E.S. .
Efficient Algorithms for Channel Routing.
IEEE Trans Comp Aided Des Int Circ Sys 1(1):25-35, Jan, 1982.