Massachusetts Institute of Technology

Artificial Intelligence Laboratory

Working Paper 292 January 1987

Knowledge-Based Schematics Drafting:

Aesthetic Configuration as a Design Task

Raul E. Valdes-Perez

Abstract

Depicting an electrical circuit by a schematic is a tedious task that is a good candidate for automation. Programs that draft schematics with the usual algorithmic approach do not fully exploit available knowledge of circuit function, relying mainly on the circuit topology. The extra-topological circuit characteristics are what an engineer uses to understand a schematic; human drafters take these characteristics into account when drawing a schematic.

This document presents a knowledge base and an architecture for drafting arithmetic digital circuits having a single theme. The relevance and limitations of this architecture and knowledge base for other types of circuit are explored.

It is argued that the task of schematics drafting is one of aesthetic design. The affect of aesthetic criteria on the program architecture is discussed. The circuit layout constraint language, the program's search regimen, and the backtracking scheme are highlighted and explained in detail.

A.I. Laboratory Working Papers are intended to circulate internally; they may contain information that is, for example, too preliminary or too detailed for formal publication. It is not intended that they be considered papers which published articles may cite.

Copyright (C) Massachusetts Institute of Technology (1987)

Table of Contents

1. Introduction 1
  1.1. Relevance to AI 1
  1.2. The Theme 1
  1.3. Salient Techniques 2
  1.4. Organization of This Paper 3
2. The Drafting Task 4
  2.1. What is a Schematic? 4
  2.2. Role of Schematic Layout in Engineering Design 4
  2.3. Evaluation of a Schematic 4
  2.4. Alternative Methods 6
  2.5. Why Traditional Methods Need Improvement 7
  2.6. Related Research 8
3. Characteristics of the Domain and Problem-Solver 9
  3.1. What Characteristics are Inherent? 9
  3.2. Task Characteristics 9
  3.3. Problem-Solver Characteristics 10
  3.4. Characteristics of Human Problem-Solving 11
  3.5. Other Domains for Which Architecture is Promising 12
4. Two Drafting Examples 12
  4.1. Binary Full Adder 13
  4.2. Four-Bit Incrementer 15
  4.3. Execution Time 15
5. Program Architecture 15
  5.1. The Input 15
  5.2. The Output 18
  5.3. Where Is The Domain Knowledge? 18
  5.4. Representation Of The Domain Knowledge 18
  5.5. Types of Contradiction 20
  5.6. The Search Regimen 21
  5.7. Fixes: Resolving Contradictions 22
  5.8. System Diagram 23
  5.9. The Backtracking Scheme 25
  5.10. Summary 28
6. Performance Analysis 28
  6.1. Why Does The Program Work? 28
  6.2. Program Competence - Knowledge 29
  6.3. Program Competence - Architecture 29
  6.4. Work To Be Done 31
7. Conclusion 31
  7.1. Achievements Of This Research 31
  7.2. A Computational Theory of Drafting 31
  7.3. Notable Aspects 31
I. The Features 33
I.1. Feature Definitions 33

List of Figures

Figure 3-1: Task Characteristics 10
Figure 3-2: Problem-Solver Characteristics 11
Figure 3-3: Human Problem-Solving Characteristics 12
Figure 4-1: Sample Layout of Tree and Bifeed Features 13
Figure 4-2: Full Adder 14
Figure 4-3: Full Adder Drafted by Program 14
Figure 4-4: Four-Bit Incrementer 16
Figure 4-5: Four-Bit Incrementer Drawn By Hand 17
Figure 5-1: Graph Rendering of a Simple Linear Inequality 19
Figure 5-2: Longest Path Example 20
Figure 5-3: Fixing A Conflicting Crown and Tree 23
Figure 5-4: System Block Diagram 24
Figure 5-5: The Backtracking Scheme 27
Figure 6-1: 4 Bit Arithmetic and Logic Unit 30
Figure I-1: Example Feature Configurations 35

Knowledge-Based Schematics Drafting:

Aesthetic Configuration as a Design Task

1. Introduction

This document reports on the design of a program to draft digital circuits in a schematic. The purpose of a schematic is to record in readable form an account of the connections and function of a circuit. Drafting a schematic is tedious, so automation of this task is desirable. Because schematics are meant to be viewed by people, a drafter should consider layout conventions as well as general rules of aesthetics. We have therefore designed a schematics drafter that incorporates knowledge of layout conventions and aesthetics, and borrows techniques from artificial intelligence.

1.1. Relevance to AI

Schematic layout is appropriate as a domain for artificial intelligence research. Previous work has reported on programs to automate drafting, but they use a conventionally algorithmic approach that produces correct schematics but lack the visual appeal of those made by a human. People are better drafters than these programs in part because they are able to relax intelligently desired features of a schematic when they conflict with other more important features.

Schematic layout is characteristically a problem of *design*. Design is not well understood, so a program that does it successfully is a contribution to progress. The evaluation of the quality of a schematic depends mostly on aesthetic factors that are little understood. Aesthetic quality is generally thought of as holistic, in the sense that conjoining the quality of subparts does not reliably predict the quality of the whole. However, our understanding of aesthetic quality is not mature enough to permit the formulation of an evaluation function.

Aesthetic design by machine is attempted by proceeding constructively and locally, despite the lack of a guiding evaluation function. As such, this procedure is heuristic, with no guarantee of success. Therefore any satisfactory result is a contribution to technique in aesthetic design.

1.2. The Theme

The theme that underlies our approach to schematic drafting is the following.

> **A program can competently draft a schematic by first identifying in a circuit the presence of known features that constrain the possible layouts in several disjunctive ways. The disjunctions suggested by the features are then searched using an iterative constraint regimen, fixing conflicts as they arise by effectively introducing a new uncatalogued disjunct in one of the features responsible for the conflict.**

These features are collected by hand, and are independent in the sense that each alone represents a comprehensible chunk of knowledge about layout. However they do interact and often the constraints suggested by one feature contradict those of another feature. Each feature makes a disjunctive statement about the layout, by saying that any of several ways to place the gates involved in the feature

is acceptable.

The search regimen of the program is *iterative constraint*, as defined in [13]. Features are successively selected according to priority and the constraints associated with the next of its disjuncts are *posted* [40]. In general, these disjuncts are a function of the state of the layout so far, instead of being statically catalogued. So the program needs to access the current partial layout.

Upon reaching a contradiction, the program attempts an immediate fix to the cause by invoking a group of rules specialized to this task. Fixing a contradiction is possible because there is much information available for an intelligent fix at the moment of the contradiction. A fix of the inconsistent interaction among features is an easily articulated piece of knowledge with a cognitive appeal. These pieces of knowledge are by nature *fixes* to problems, so there is no obvious way they could be incorporated into the program's *generator* of candidate solutions, instead of the *tester* of solutions.

If the fixes repair the contradiction, then the iterative constraint proceeds with the next feature according to priority. Otherwise the program backtracks intelligently to a previous state, exploiting information from the current and previous contradictions. This information is typically called a *nogood set*, or simply a *nogood*.[1]

1.3. Salient Techniques

There are several aspects of this work that contribute to the storehouse of AI techniques, beyond the domain at hand. These are:

1. The constraint language of simple linear inequalities can express constraints that mention only one of the two axes of the layout, and which specify maximal and minimal distances between gates. Moreover, there are algorithms that determine efficiently the consistency of a group of such inequalities, and as a by-product permit detecting the violation of disjointness and view constraints.

   Disjointness refers to impossibility of two gates sharing both x and y coordinates in the planar layout. A *view* between two gates in some axis, say x, means that there exists no third gate such that all three gates share the same y coordinate, and the third gate lies between the other two gates in the x axis. Our program has not exploited view constraints, but other space-planning domains require them [30].

2. The iteratively constrained search proceeds until meeting an inconsistency. At this point, a set of contradiction-handling rules is invoked to try to fix the problem by changing the constraints belonging to one of the features involved in the nogood set of features. A fix has preconditions that must be met before carrying out the action. These preconditions refer to the partial layout, the nogood set, and other information about the contradiction.

   Necessarily, the last feature whose constraints were posted accounts for the contradiction, because consistency is checked after each such posting. Often this feature is the one targeted for change, but a feature whose constraints were previously posted could be selected instead. Regardless, the fix inserts a new choice for a feature "on the fly", and all proceeds as if that choice had been the original one.

---

[1]The term *nogood* by convention refers to a set of assertions choices that conflict whenever they occur simultaneously. For example, the assertions "x=1" and "x>3" conflict, so the set {x=1, x>3} is a nogood. As a propositional statement, a nogood can be viewed as the denial of the conjunction of its elements, as well as a disjunction of the denial of each element. This propositional interpretation of nogoods is exploited during the discussion of backtracking in a later section.

Is it possible to anticipate the problem by cataloguing that choice in order to avoid the contradiction in the first place? Our answer is no, because the knowledge of fixes is not easily translated into knowledge about features and the statements that they make about layout, because the nature of the knowledge is distinct. These fixes are general, and treat the problems that arise when various competing subgoals (features) conflict, not just when known feature-1 conflicts with known feature-2.

3. When no fix applies, or fails to fix the contradiction, then the search tries another choice at some choice point (i.e. feature). The iterative constraint regimen seemed to degrade to chronological backtracking, because the last feature-choice posted always contributes to the conflict.

   The solution was to always try a new choice at the last choice point C, as in chronological backtracking, but to jump intelligently farther back in the search tree when the choices at C are exhausted. The information that allows this intelligent backtracking results from the nogoods accumulated by the contradictions at C. We call this the *backtrack-list* at C.

   When backtracking from C to a node N, the backtrack-list of N is updated using the backtrack-list of node C. This method of updating lists of backtrack points is equivalent to the resolution of nogood sets with disjunctive assertions about the existence of a solution path in the search tree.

   The virtues of this backtracking scheme are threefold. A simple backtrack-list is stored at nodes in the search tree, and these lists are updated whenever a backtrack occurs, if we consider returning to the last feature posted a backtrack also.[2] There is a simple justification for the scheme, based on resolution. Finally, the scheme guarantees completeness, in the sense that if there is a solution, it won't be overlooked.

1.4. Organization of This Paper

The sections in the rest of this document continue the description of the automatic schematics drafter.

The next section discusses the drafting task and motivations for a knowledge-based approach. Section 3 describes the properties of the domain and the problem-solving architecture. The fourth section presents the performance of the program on two example arithmetic digital circuits. The fifth section treats the program architecture in detail, points out where the domain knowledge lies in the program, how inference is accomplished, and shows a functional block diagram. Section 6 tries to account for the performance of the program, and examines the boundaries of its competence due to both knowledge and architectural limitations. Section 7 concludes.

Appendix I lists the domain knowledge captured by features. Appendix II lists the contradiction-handling rules or fixes.

---

[2]This backtracking method is apt when the choice made for a given feature depends on the choices made for previous features. Given this dependency, the usual assumption of simple dependency-directed backtracking (DDB) is invalid. When backtracking due to a conflict, DDB assumes that a new choice at some choice-point does not invalidate any other choice. This assumptions is void in our domain, because in general the choice for a feature depends on the partial layout, and therefore on all the previous choices. Therefore, when backtracking to some feature, all features skipped over for backtracking must be "reset". This could be avoided by introducing the overhead of some TMS mechanism.

2. The Drafting Task

2.1. What is a Schematic?

Schematics exist as documents for the purpose of signal tracing and functional understanding. A schematic is not concerned with a faithful depiction of those physical characteristics of a circuit such as size, shape, and location.

Diagrams that do include the physical arrangement of components are commonly called connection or wiring diagrams.

2.2. Role of Schematic Layout in Engineering Design

There are several uses for a functional schematic in engineering design. Among these are breadboarding, timing studies, archives, and optimization by an engineer of an automatically designed circuit.

A typical use is as a guide to breadboarding a circuit, in order to test its correctness and other properties such as noise immunity. Breadboarding refers to the hardware assembly of a designed circuit prototype. This practice is common even where circuit simulators are available, because often these simulators are complex to use and have other disadvantages that are time-consuming.[3] Moreover, circuit simulators are *logic simulators*, but there are many circuit hazards that are not predictable from the circuit logic.

A functional schematic is also useful for finding the critical path in a circuit, i.e. that signal path that determines the overall circuit delay.

When circuits are designed automatically by a program, an engineer checks the design for possible simple improvements that are evident by looking at a schematic, but are difficult to incorporate in programs.

A final purpose for schematics arises in the following common situation. A circuit is designed and used, and a year later changes must be made or the design is adapted for another purpose. An engineer is assigned to this task who was not previously involved with the design. His first step is to understand the circuit, and this is done through a schematic.

2.3. Evaluation of a Schematic

The purpose of a schematic is to highlight to the reader those aspects such as signal flow, the primary inputs and outputs of the circuit, connectivity, module identities, and the function of the circuit. This information is communicated by text, gate symbols, wires, placement of the gates and the primary signals, and finally by conventions such as left-to-right signal flow.

To ease comprehension, schematics must be *correct* and they should be agreeable to the eye. The

---

[3]For example, simulators may require the selection of device models (e.g. for transistors) to carry out the simulation, forcing the engineer to recall the relative merits of the hybrid-pi over (or under) other transistor models.

assumption is that an agreeable layout eases the task of extracting information, at the very least because it does not *annoy* and therefore *distract* the viewer.[4]

As a guide to agreeable layouts, one can appeal to established notions of aesthetics. However, much of what is written on the subject is not computationally operational, because the distance from what is said to its axiomatization is vast. The next few paragraphs discuss what insights, formal and otherwise, are available from some of the pertinent writings.

The mathematician Birkhoff in the 1930's tried, by introspection with polygons, to quantify elementary notions of proportion and shape [6]. As simple geometric drawings, these polygons avoided the complexity of taking into account the aesthetic appeal of objects that draw on our life experience. Birkhoff attempted in his experiment to exclude semantics; the difficulty of this was illustrated in his collection of polygons by the ordinary rating assigned to the swastika, which at the time of his writing was already notorious.

His major idea relevant to our task is the strong antipathy caused by the presence of ambiguity in his geometric figures. Small distances between parallel lines that make the viewer strain to distinguish them are undesirable, as are angles that are near 0 or 180 degrees. Birkhoff also noted the favorable impression left by vertical symmetry in the polygons, versus the indifference caused by horizontal symmetry.

The schematics domain is more *connotative* (Birkhoff's term) than simple polygons, in the sense of invoking other experiences distinct from the syntactic etchings on the schematic. The overriding extra-syntactic notion is that of signal flow. There is also the idea of inputs to the circuit which are fed externally, and of outputs also having an external function.[5] Another source of ideas on aesthetics is a book by another mathematician Weyl [48]. He treats the presence of symmetry both in nature and in human artefacts, and discusses the feelings that they suggest. The best insight is that symmetry conveys a sense of permanence and stability, and that is why town halls and millenary cathedrals possess symmetry, often with respect to a central vertical line. Instead, to suggest fluidity and dynamicism, asymmetry is preferable, which is why Greek togas cover only one shoulder and slant across the upper breast.

An opinion of this author is that one should purposely, where appropriate, introduce asymmetries in objects designed by programs in order to give them a "human touch." Mindless symmetry is associated with contraptions, so a program that emulates human design should avoid it.

Aestheticians have in the past tried to use "golden rectangles" as a primitive aesthetic construct [32]. Golden rectangles have a ratio of sides equal to $(\sqrt{5}-1)/2$; this ratio has been discussed since Greek antiquity, and arises from the statement:

Side A is to side B as side B is to the sum of both.

The idea is that rectangles of this size have a pleasing appearance, and one author has exploited this fact to design a window system whose windows have this proportion [15]. The relevance to schematics is evident.

To summarize, the available insights concern ambiguity and symmetry. To reduce ambiguity, one can:

---

[4]For example, most people are distracted by flagrant grammatical mistakes or misspellings in text, and understandability can only decrease as a result.

[5]For a criticism of the results, but not of the approach, see [14].

1. Ensure that parallel wires are evenly spaced without being too close. The routing channels must be large enough to allow this spacing.
2. Distinguish clearly between wires that cross in the schematic without sharing the same signal, and fanouts. The easier way is to draw a large dot in the case of fanouts.
3. Avoid drawing wires contrary to the convention adopted for signal flow (here signals flow toward the right). The author has observed one schematics drafter used in industry that occasionally placed gates such that a backwards wire was needed to complete the routing. This is very undesirable because it suggests feedback where it does not exist.
4. When possible, distinguish gates not by adjacent text, but by using the conventional symbols for them (e.g. NANDs, ANDs, ORs).
5. Use different colors, for example for primary signals and internal gates.

Our program designs schematics by reconciling features that possibly conflict. Many of these features only concern connectivity of a few gates and are therefore quite localized. One can insert asymmetries into these local features by choosing the constraints that they introduce accordingly. For example, a "tree" feature occurs when a gate is fed only by two other gates, and each of the latter feeds only the former. Instead of choosing the constraints such that the tree is layed out like an isosceles triangle, we prefer to configure the three gates asymmetrically like a right triangle, with a head, center, and wing. This scheme suggests a dynamic flow, in agreement with the direction of actual signal flow, and is also visually appealing.

Finally, Herbert Simon identified three aspects of a mechanical designer that determine its *style* [36], and offered therefore a framework in which to analyse style. The following list is my interpretation of these three style determinants:

1. The order in which equally plausible but competing choices are made.
2. Catalogued solutions to specific task subproblems that are applied when the presence of the subproblems is recognized.
3. Constraints imposed on the task solution by virtue of the implements (read algorithms) that carry out the design. The constraints are not necessarily deliberate design decisions on the part of the programmer.

An example of the third style determinant above is the algorithm that is used here to assign components to final locations on the layout. When the search stage ends with an abstract and consistent description of the layout, there are many layouts that are instances of the abstract layout. To create one instance, the components are placed as far to the left and top as the abstract description allows, i.e. they are left- and top-justified. This *justification* is not deliberate on our part, rather they result from the somewhat complicated algorithm needed to assign the locations in a manner consistent with the abstract description.

2.4. Alternative Methods

People have built and reported on automatic schematics drafters at least since 1963 [21], as university theses [43, 24], as spinoffs from larger research projects [31], and as industrial projects [45]. These programs were all algorithmic and therefore one expects that they shared common virtues and drawbacks.

A typical algorithmic drafter likely has the following characteristics:

1. The programs, some to a greater degree than others, essentially lay out directed graphs

and not circuits, because they concentrate on the circuit topology. They do give some attention to the character of the nodes in this digraph, e.g. a multiplexer is treated differently from a latch. However, other information that relates groups of devices is ignored.

2. Knowledge about layout is not clearly separable from the procedural statements of the algorithm.
3. The modularity is questionable, in the sense that adding a new piece of knowledge about layout is not straightforward.
4. The program architecture makes little appeal to cognitive plausibility.
5. Placement across rows is largely determined by signal flow. Placement in columns proceeds typically by an evaluation function that selects an initial gate, followed by successive placements of gates that "are closest" in some sense to those already placed. Hill climbing is used by testing whether the permutation of two gates would improve the layout by reducing routing crossovers.
6. Execution is swift, often because programs do not backtrack; the knowledge they have is applied come what may. Since execution time is minimal if one randomly assigns gates to columns, evidently there is a trade-off between speed and layout clarity due to the ease of routing resulting from careful placement.

Another traditional method is to draft a schematic by hand. This is less appropriate now that circuits are designed by computer, because the circuit representation is freely available to a drafter program.

Stochastic methods for accomplishing VLSI layout or floorplanning have recently become popular [16, 51]. These approaches rely on optimization by simulated annealing. The goal is to minimize some mathematical function that represents the goodness of a solution, by searching the range of the function nondeterministically in order to avoid settling in local minima. The appeal of the approach lies in its theoretical grounding, and in the ease with which new aspects of goodness may be integrated by adding or subtracting new terms to the mathematical function.

The drawback for schematics drafting is that the quality of a schematic, being principally aesthetic, is not well understood and not currently formalizable. Some components of schematic quality, such as wire lengths and number of crossovers, are easily represented by a number. However, because these two aspects contribute only partly to overall quality, we believe that this stochastic approach is not adequate for our domain.

A recent paper reports on a rule-based schematics drafter [1, 2], written in OPS5 but with substantial computation external to the rule-based system. This system was experimental and never progressed beyond prototype. It is unclear whether the procedure that this program followed is algorithmic, and consequently OPS5 was used merely as a programming language, or whether the production system architecture meaningfully contributed to and was consonant with the approach tried.

2.5. Why Traditional Methods Need Improvement

A knowledge-based architecture has known advantages and disadvantages over algorithmic programs.[6] A worthy program built in a KBS approach should have a good separation between its domain knowledge and its inference or search procedure. From this property flows the software engineering virtues of extensibility both within a domain and across domains.

---

[6]The phrase *algorithmic program* should be interpreted as an idiom used colloquially by the AI community to refer to a certain program architecture or style.

Extensibility within a domain refers to the relative ease of adding new bits of knowledge to a program with at most localized changes to the initial domain knowledge, and no change to the inference/search mechanism. To extend across domains means to employ the same programmatic framework for a different task, by substituting the new domain knowledge for the old. In these two senses, rule-based systems are extensible.

Algorithmic schematics generators fail both tests of extensibility, and possibly this is the reason why papers on the subject have continually appeared, for each slightly different domain.

The program and approach described here are modular because they ease the insertion of new knowledge. The search framework is separable from the domain knowledge and is readily understood. In this way it is extensible to like domains, to be discussed shortly.

2.6. Related Research

As part of the general effort in automatic space-planning of a decade ago [17], the Design Problem Solver was an early attempt to configure equipment or furniture in a fixed planar space [30]. The input was a sorted list of objects to be layed out and constraints that must hold between objects in any solution. The program proceeded by placing the next object in the list in some corner of the free space, and a list of constraints was tested for possible violations. For example, if a clear view between a computer console and a tape drive were needed, then after both were placed this condition was checked.

Contradictions were handled by either relaxing the constraint that was violated, if that is permissible, or by backtracking. There was no systematic backtracking scheme similar to dependency-directed backtracking [38], because the latter was developed a few years afterwards.

The approach taken by DPS is that of *generate and test* [28]: generate specific candidate layouts and discard those that fail a list of tests. Therefore there was no notion of least-commitment, whereby a program repeatedly generates *classes* of solutions and applies to these classes those tests that are not incorporable into the generator. The *constraint-posting* technique of least-commitment search is an idea available to modern attempts at layout programs, and is exploited in our schematics drafter.

There are two current research expert systems that bear similarity with schematics drafting [20, 22]. TALIB is a rule-based system that lays out integrated circuits given their components and interconnections. WEAVER is a VLSI routing program, based on a blackboard architecture, that interconnects the pins on the surface of an IC chip.

The similarity to TALIB stems from the common need to allocate spatial resources. The differences arise from the performance evaluation criteria. In VLSI layout the overriding measure of quality is the compactness of the layout. The compactness is easily expressed as a number, whereas the quality of a circuit schematic is not.

WEAVER uses a blackboard architecture because of the diverse types of knowledge that bear on the routing task. Also some of these types of knowledge work from different problem representations. The need to integrate and reconcile this knowledge led to the adoption of the blackboard architecture. For drafting, whether in the placement or routing stage, the sources of knowledge are not as diverse. Therefore a more complicated blackboard architecture is not justified in this author's view.

3. Characteristics of the Domain and Problem-Solver

3.1. What Characteristics are Inherent?

Besides reporting on the abstract computations that a problem-solver carries out, one should highlight those properties whose presence would suggest the use of the same approach on another problem. The nature of *both* the task domain and the problem-solving strategy are needed, because what may appear inherent in the task is, on the contrary, merely characteristic of the chosen strategy.

Whether the act of design is a characteristic of the task or the problem-solver is controversial. Clancey [7] argues that design is better regarded as an architecture for problem-solving, as follows:

> **A common misconception is that the description "classification problem" is an inherent property of a problem, opposing, for example, classification with design ... However, classification problem-solving, as defined here, is** ***a description of how a problem is solved.*** **If the problem solver has a priori knowledge of solutions and can relate them to the problem description by data abstraction, heuristic association, and refinement, then the problem can be solved by classification. For example, if it were practical to enumerate all of the computer configurations R1 might select, or if the solutions were restricted to a predetermined set of designs, the program could be reconfigured to solve its problem by classification.**

To resolve this controversy about the nature of design, we could use the word "synthesis" to mean the creation of what does not exist, in contrast with "analysis." Then Clancey's distinctions are useful because they contrast the different AI techniques used by designers and classifiers:

> **Whether the solution is taken off the shelf or is pieced together has important computational implications for choosing a representation. In particular, construction problem-solving methods such as constraint propagation and dependency-directed backtracking have data structure requirements that may not be easily satisfied by a given representation language.**

So let us proceed to list the properties inherent to drafting, and those of the problem-solver and of a human performing the task.[7].

3.2. Task Characteristics

In Figure 3-1 appear the inherent characteristics of the task of schematics drafting. The following discusses them briefly.

As discussed in the previous section, the task *synthesizes* a solution.

The quality of a schematic is largely aesthetic and therefore based on the coherence of a whole that is not reliably predicted by conjoining the aesthetic quality of the parts. Since no one has formalized aesthetic appreciation successfully, there is no evaluation function available that captures our sense of the goodness of a schematic.

The scarcity of resources is not an important constraint, because the space used can be enlarged (or the magnification of the schematic decreased) without much trouble. The only exception to resource scarcity is that no two gates can occupy the same location in the plane (i.e. our schematics are not expandable in the third dimension).

---

[7]A checklist helps; ours is from [8]

1. Synthesis.
2. No evaluation function available that captures our appreciation of the quality of a solution.
3. Resource scarcity not a problem.
4. Combinatoric number of solutions.
5. Narrow and closed-ended domain.
6. There is freedom to fail.
7. Not time critical.
8. Incremental progress is possible.

Figure 3-1: Task Characteristics

The plausible layouts for a given circuit can be generated by permutations of assignments, so the number of layouts is evidently combinatoric.

The domain is certainly narrow and isolated enough so that there is little appeal to human common sense, except as concerns spatial relationships in the plane.

The freedom to fail is important because the general brittleness of expert systems (rule-based or otherwise) makes them susceptible to outright failure on any given problem, if the knowledge base is incomplete. Time criticality is undesirable because the architecture of the classic rule-based system does not admit time constraints. These two task characteristics are actually unimportant for the architecture adopted for the schematics drafter, because a random solution can be extracted from the currently contemplated class of solutions at any time. The more time that is available, the more that this class of solutions is narrowed.

Finally incremental progress is possible because with little knowledge one can still obtain *some* schematic; with more knowledge performance should improve.

Note that all the characteristics but the first two, regarding resource scarcity and lack of an evaluation function, are desirable task attributes for a knowledge-based approach to make sense.

3.3. Problem-Solver Characteristics

The properties of our program problem-solver, as distinct from those inherent in the drafting task, are listed in Figure 3-2. Our program *designs*, which means, following Simon [37], that it searches through a space of prospective solutions that it constructs dynamically. There is not a list of canned solutions which

1. Does not use classification approach (does design).
2. Satisfices instead of optimizing.
3. Formulates solution as set of disjunctive subgoals, where for each subgoal an optional disjunct is the null goal (i.e. subgoals are relaxable).
4. Conflicting subgoals are occasionally reconciled by adjusting one (or more) of them to remove the conflict.
5. Demands a task where it is possible to write intelligible rules to fix conflicting subgoals.

Figure 3-2: Problem-Solver Characteristics

are heuristically matched against the specific case, as is typical of the classification approach [7].

Like most practical design procedures, our problem-solver "satisfices" (Simon's term) instead of optimizing. For schematics drafting, reliance on optimization, or suboptimization through hill-climbing to local maxima, is somewhat contrived because the function to be maximized does not capture our appreciation of what makes a schematic good. At best it concerns one aspect, e.g. wire lengths, that contribute only partially to overall quality.

A solution to the drafting task is ideally the selection of single disjuncts from each of the disjunctions contributed by features that are present in the input circuit. Each feature suggests a set of constraints that it desires the solution to satisfy, but states a few of these sets in case some of them are not satisfiable. These disjuncts are equally acceptable to the feature. If the feature is relaxable, and perhaps difficult to satisfy in practice, then one could include a null disjunct that asserts nothing at all.

Inevitably these features will conflict, so our problem-solver has a mechanism that allows the embedding of knowledge to fix contradictions as they arise. It should be possible to recognize those features that are responsible for the conflict, and the circumstances should be rich in information to enable an intelligent correction of that conflict.

3.4. Characteristics of Human Problem-Solving

Some methods of the program are common to a human drafter also (please refer to Figure 3-3. People *design* schematics, instead of selecting one from a small catalogue of pre-selected schematics. They also satisfice, because the cost of optimizing is not worth the effort, and because an optimal schematic is a vague notion anyway.

1. Does design.
2. Satisfices.
3. Uses both cognitive and perceptual skills.
4. Performance takes a few minutes.

Figure 3-3: Human Problem-Solving Characteristics

A human drafts by employing both cognitive and perceptual faculties, just as a chess player does. The greater the role of perception in the problem-solving task, the harder it is for current AI techniques to perform well.[8] Our program emulates cognitive aspects of drafting, and is therefore limited by the extent to which it fails to capture essential feedback provided by human perception.

3.5. Other Domains for Which Architecture Is Promising

The techniques developed here have an obvious application to the problems treated by the space-planning research of a decade ago. Space planning concerns the physical arrangement of objects in two or three dimensional space. Interest in those problems within AI diminished due to lack of success [17], but the evolution of dependency-directed backtracking and constraint-posting promises better results.

Our architecture also seems relevant to planning problems, in which the steps that constitute a course of action are delineated before carrying out any of the steps. If knowledge that reconciles conflicting aspects of the plan steps is readily available and intelligible, then an architecture that instantiates plan steps by priority and fixes problems as they arise, backtracking when needed, may be appropriate.

4. Two Drafting Examples

Two drafting examples appear in this section. The reader should keep in mind that our program does *placement*. It is expected that a good placement greatly eases the routing task, so the major task is to place the gates well. However, in order to evaluate the quality of the placement, the routing between gates should also be shown.

The output of our program is a pair of integers for each object (gates plus primary signals) to be

[8]Chess programs do better at tournament speeds than at lightning chess, where only a few seconds per move is possible, and where humans rely more on pattern-matching than on conscious search and evaluation.

placed, representing the grid points along the x and y axes. However, much of the routing is determined by the features in the circuit that serve to guide the placement. For example, a *tree* feature indicates that the center of a tree should directly feed the head, without any bends in the connection, while the wing approaches the head from the side, with two bends, as shown in Figure 4-1a.[9] Figure 4-1b shows a typical layout of a *bifeed* feature.

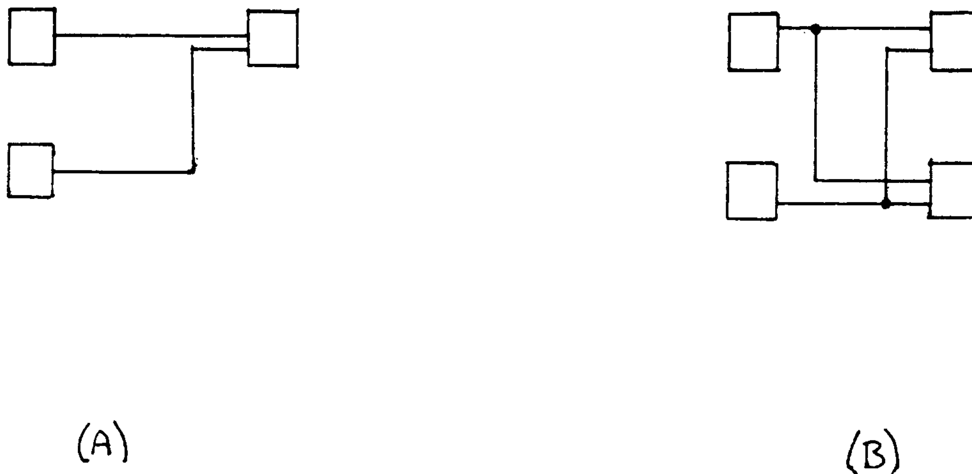


Figure 4-1: Sample Layout of Tree and Bifeed Features

Note that both a tree and bifeed can be layed out in any of four equivalent ways, by permuting positions of the gates. A routing program would have access to the features detected in the circuit, so much of the wiring pattern is already determined before the routing stage.

4.1. Binary Full Adder

The first circuit is a binary full-adder that uses only NAND gates and an inverter, taken from a textbook on digital circuits [27]. There are fifteen items to be arranged, counting gates and primary signals. Figure 4-3a is the schematic as found in the textbook.

Figure 4-3b is the schematic drafted by the program.

---

[9]This is a default way to configure a tree; often this arrangement conflicts with other features, so the center or wing connection of the tree is stretched.

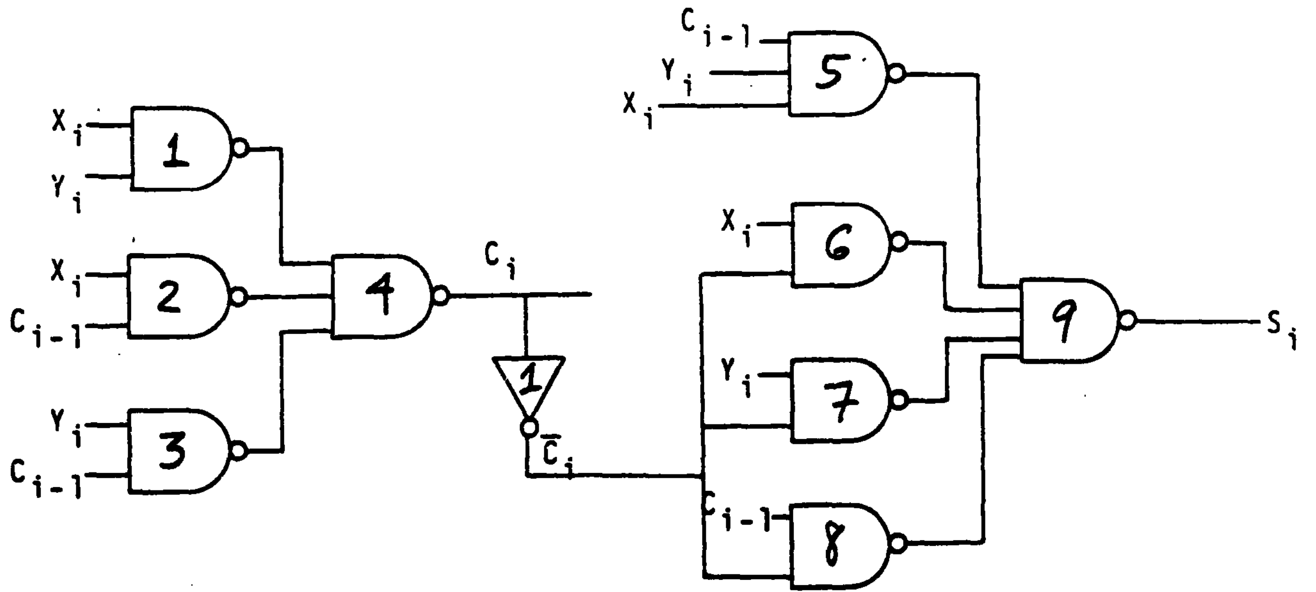


Figure 4-2: Full Adder

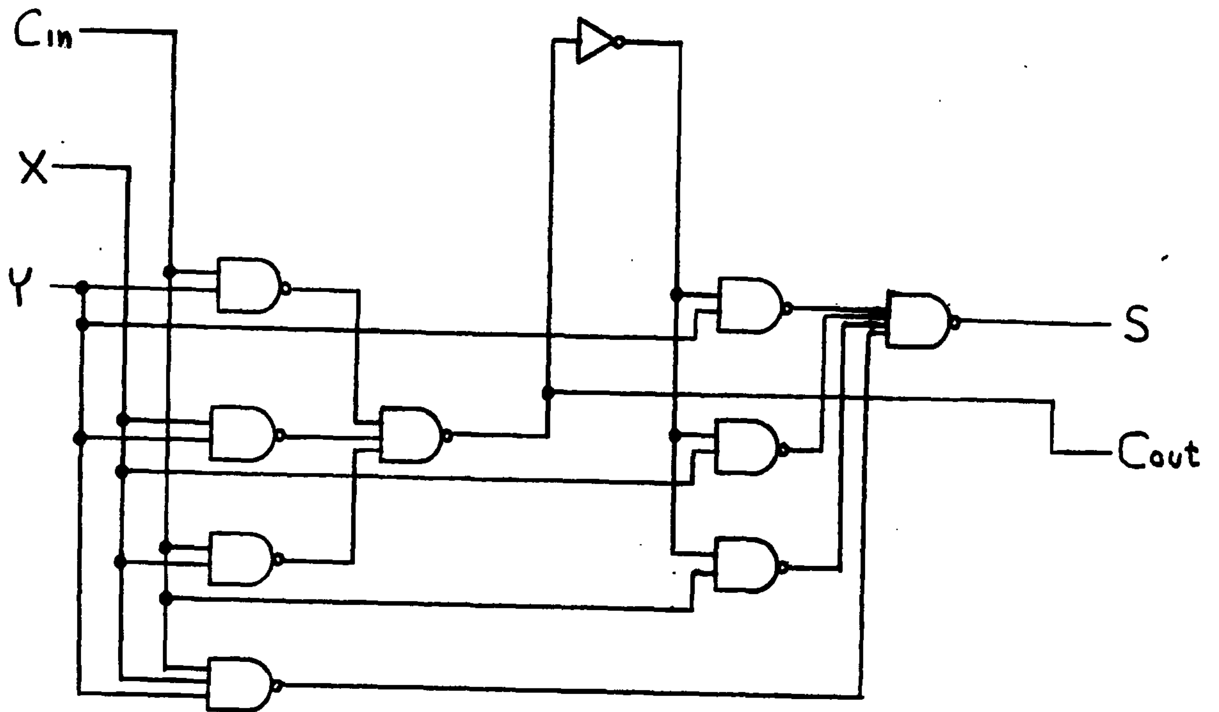


Figure 4-3: Full Adder Drafted by Program

4.2. Four-Bit Incrementer

The second circuit, shown in Figure 4-4, increments a four-bit integer, and outputs a four-bit integer and a carry bit. There are 18 gates and 9 primary signals in the circuit, giving 27 placements.

This incrementer was originally made for the purpose of experimenting with the diagnostic algorithms of the Hardware Troubleshooting group at the MIT AI Lab. This author drafted by hand the schematic shown in Figure 4-5, and relied on it for signal tracing and verification of the diagnoses produced. Although this hand schematic was not done carefully, it demonstrates, by contrast with the machine schematic, the advantage of automatic schematics drafting.

4.3. Execution Time

The following table gives the execution times for various arithmetic circuits, including the two above. The number of objects refers to gates plus primary signals.

TYPE	NUMBER OF OBJECTS	TIME
Full Single-Bit Adder	15	20 seconds
Full Single-Bit Adder Using Only NAND Gates & Inverter	13	34 seconds
4-Bit Incrementer	27	6 minutes
2-Bit Carry Lookahead Adder	24	18 minutes
2-Bit Multiplier	20	4 minutes

The reader is cautioned not to extrapolate a relationship between the number of objects and the execution time. The execution time depends much more on the amount of backtracking. The backtracking is less when fixes succeed in resolving contradictions, and when the order in which the disjuncts of a feature are searched is coincidentally a good one for the circuit at hand.

5. Program Architecture

5.1. The Input

One input to the drafter is naturally the circuit of interest. This circuit is represented in a much simplified version of **TDL**, a circuit description language developed by the Hardware Troubleshooting Group at the MIT AI Lab. By *circuit* is meant the identities of gates together with their input and output ports, and the wires between these ports.

For our purpose, we have dispensed with the hierarchical circuit description of **TDL**, and have demoted wires, fanouts, and their "ports" by grouping them into signalnets, which are fed by one signal and feed any number of signals.

The other input to the drafter is assertions about the function of the circuit and the nature of signals.

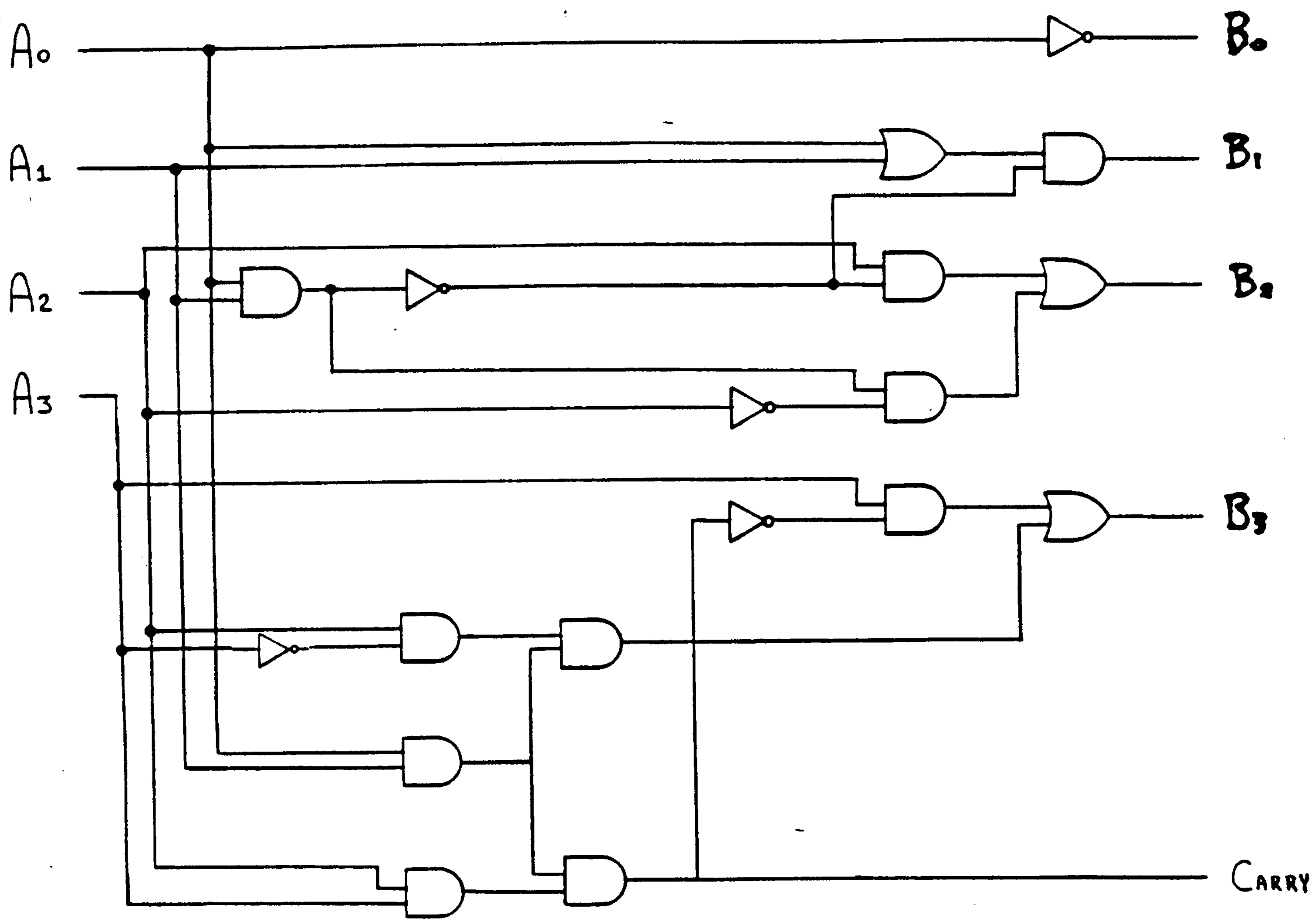
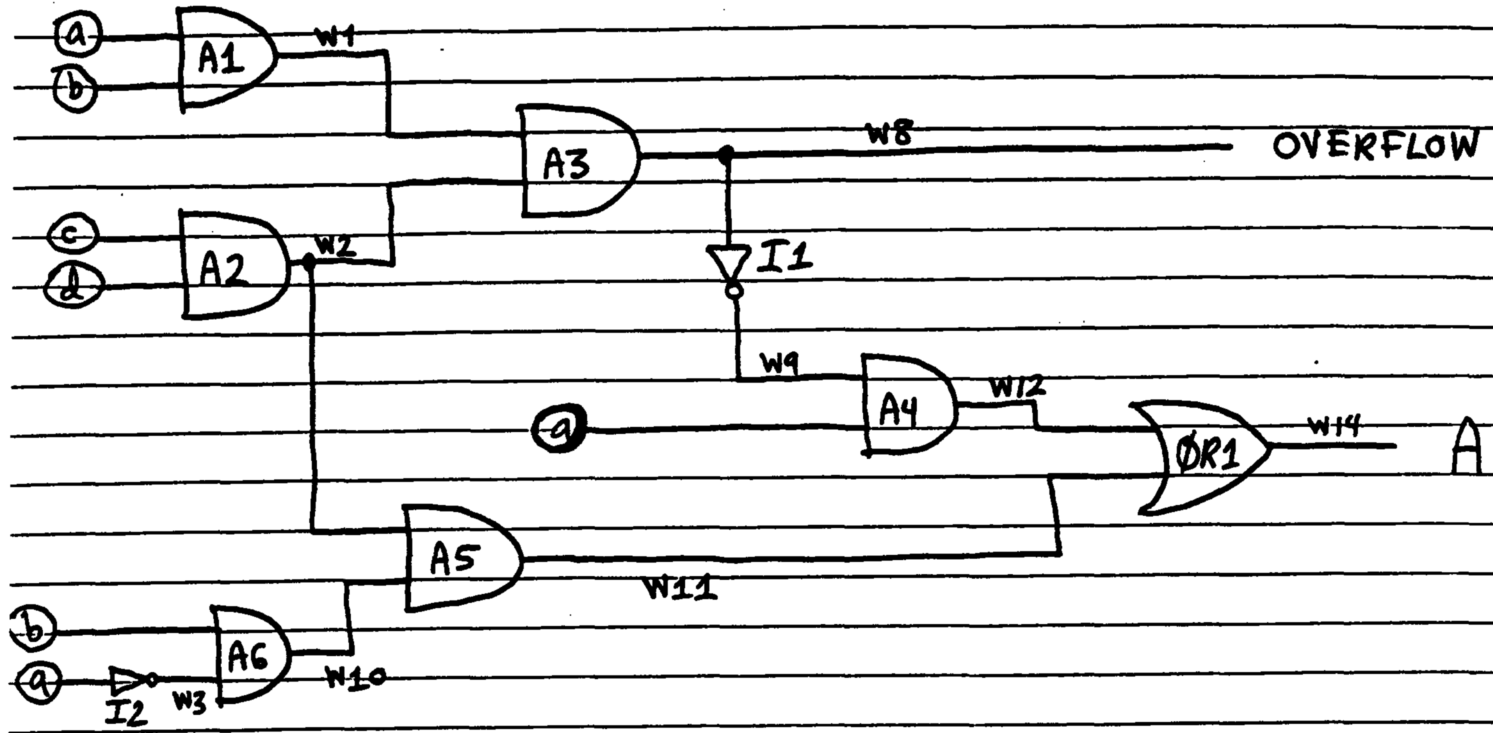


Figure 4-4: Four-Bit Incrementer

Figure 4-5: Four-Bit Incrementer Drawn By Hand



For example, the statement that a set of primary input signals constitutes the bits of an integer is useful information, as is the fact that two such integers are added to produce a third.

This functional circuit knowledge is used by human drafters, but is typically ignored by algorithmic drafters, which concentrate on the circuit topology. Characteristic of the knowledge-based systems approach is that such domain knowledge should be made available to a program and exploited.

5.2. The Output

The drafter outputs a pair of grid coordinates for each object of the circuit. Grid coordinates are integers that begin at zero and range to whatever size the program finds necessary to draft the circuit; no limit is imposed.

5.3. Where Is The Domain Knowledge?

There are two places where domain knowledge resides: in the features and in the fixes. The sources of this knowledge are introspection while drafting a schematic, the study of schematics appearing in books, and experimental observation of deficiencies while running the program on circuits.

Features make disjunctive statements about the layout of a circuit. These features are a catalogue of cues that guide the drafting. There is no limitation on the number of occurrences of each feature in a circuit. Features take into account information about connectivity and assertions regarding circuit function and the nature of signals. The features currently used by our program are listed in Appendix I.

Features are found by simply iterating over all the signalnets in the circuit, trying to classify each, singly or in combination. Hence, to find whether net N_1 is involved in a tree, one necessary condition is that the single device fed by N also be fed by another signalnet N_2 and so on. This approach appears better than describing circuits using propositional statements, and finding properties with conjunctive pattern matches over these propositions, as in [42].

Features are sorted by priority, and this order is followed when applying feature constraints to the current partial layout. Therefore, this order is also a repository of domain knowledge.

Introspection and study accounted for the features themselves, and occasionally the features' disjunctive statements were changed after experiment showed that they led too often to contradictory interaction with other features.

The fixes were collected mostly by experiment. As backtrack search through the feature disjuncts proceeded, information about contradictions was collected and examined later. Whenever a contradiction was easy to correct, its correction was added to the fixes if the correction seemed general rather than a special case for this instance.

5.4. Representation Of The Domain Knowledge

Each disjunct of the disjunctive feature statement is a set of constraints on the layout. These constraints are expressed in a constraint language that should satisfy at least these criteria:

1. Need to express the constraints of interest.
2. A contradictory set of constraints should be detectable.

3. Need to infer spatial relationships between two objects that are implicit in a set of constraints.
4. The language should allow useful information (e.g. a nogood set) to be extracted when a *contradiction occurs.*

We will assume a that a schematic is layed out in a tessellated (checkered) plane or grid, and that distances on the grid are in steps along the x and y axes only (Manhattan distances). For example, the distance between diagonally opposite points of a unit square is 2, *not* the square root of 2.

Our language must permit the expression of maximal and minimal distances between pairs of objects. As a tradeoff, we shall only allow distance constraints that refer to one of the axes, so that each constraint contains only two variables (distance constraints along a diagonal need to mention four variables). An inequality of the form

$$OBJ1_x - OBJ2_x \geq C_{1,2,x} \qquad (1)$$

where $C_{1,2,x}$ can be negative, captures constraints of maximal and minimal distances along the x axis. We note that the equality of two quantities can be expressed by the conjunction of two inequalities.

It is known that a set of *simple linear inequalities* such as (1) maps onto a graph, where the nodes are the variables and the directed edges are weighted by the inequality constant. Figure 5-1 shows a graph representation of (1).

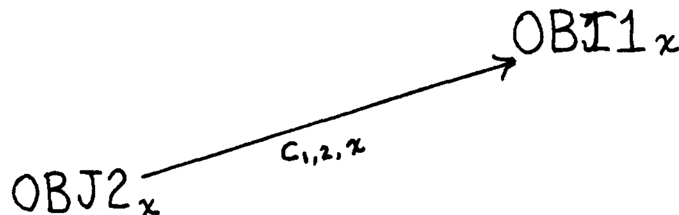


Figure 5-1: Graph Rendering of a Simple Linear Inequality

A set of inequalities is consistent if numbers (integers in this case) can be assigned to the variables such that all the inequalities hold. The single-source longest-path algorithm (SSLP)[10] determines consistency of a graph by assigning integers; if after a certain number of iterations there remains some violated inequality, then the set of inequalities is unsatisfiable.

However, we do not use this algorithm because its output does not permit the query of spatial relationships between pairs of objects. Since the assignments of integers to variables is the only output, we can not ask whether AND-1 is necessarily to the right of AND-2 in any layout resulting from the current

[10]The usual name is the single-source *shortest-path* algorithm, where the goal is to find the shortest path from a designated "floor" variable to all other variables. By reversing the directions of the inequalities in the algorithm we find the longest path, instead of the shortest.

partial layout. If the SSLP assigns 5 to the x coordinate of AND-1 and 4 to the x coordinate of AND-2, *at most* we know that AND-2 is not necessarily to the right of AND-1.

To inquire of the relationship between any pair of objects, we need to compute the longest paths between all pairs of nodes, not just from a single source. The All-Pairs Longest-Path (APLP) algorithm computes the transitive closure [3] of the graph; it makes explicit all the relations between pairs of nodes that hold implicitly through transitivity. With these longest paths, one can answer queries about the spatial relationship between any pair of objects. For example, the path in Figure 5-2 implies the inequality

AND-1 - AND-3 ≥ 2

by the inferential step

If $w\text{-}x \geq c_1$ and $x\text{-}y \geq c_2$ then $w\text{-}y \geq c_1+c_2$.

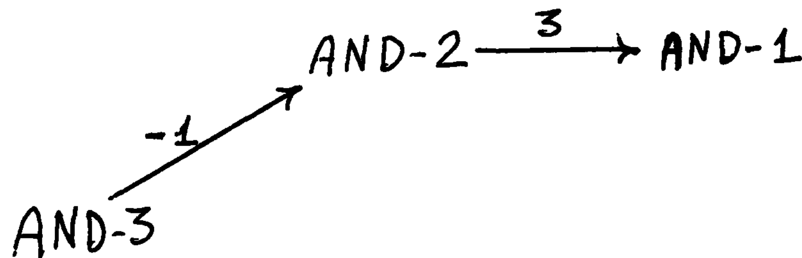


Figure 5-2: Longest Path Example

The APLP algorithm does not assign integers to each of the objects as the SSLP does, so one cannot verify consistency by checking whether the inequalities hold under the assignments. Instead a set of inequalities is consistent if, after performing the APLP, there exists no pair of variables (object coordinates) such that the sum of the longest paths in either direction is positive. The time complexity of this check is quadratic in the number of objects.

Before discussing the information reported when contradictions occur, we need to define the nature and physical interpretation of a contradiction, in the next section.

5.5. Types of Contradiction

If one asserts the two statements $a\text{-}b \geq 1$ and $b\text{-}a \geq 1$, evidently there follows a contradiction. This contradiction is manifested by the appearance in the graph of a closed loop with a positive sum of edge weights. Physically, "a" is required to be both above (or to the right of) and below (left of) "b." Another interpretation is that "a" is required to be above itself. The set of constraints, or nogood, responsible for this *transitive* contradiction is the inequalities along the closed path.

However, there is another type of inconsistency which is characteristic of layout problems: no two objects can share the same coordinates in the relevant dimensions. In schematics, we designate by the term *clash* the forced sharing of the same position in the plane.

The transitive closure, as computed by the APLP algorithm, reveals whether two objects clash. Two

objects *align* along an axis if they forcibly share the same position on that axis. Two objects *A* and *B* forcibly share the same position if and only if there is a zero-sum path in each direction between them, because these two paths assert that

$$A\text{-}B \geq 0 \text{ and } B\text{-}A \geq 0,$$

meaning that *A* equals *B*. If two objects align along both x and y, then they clash.

To find the clashes, one first performs the APLP algorithm. Then one collects the alignments in both dimensions and intersects the two sets. For N the number of objects, the worst case complexity for the alignment computation is $N^2 \log N$, but if no clashes are found, the complexity is N.[11] For a clash, the nogood set is the set of inequalities along the four zero-sum paths covering both axes.

There is a third type of constraint whose violation the transitive closure helps to detect, but was not exploited by our program. We call this the *view* constraint, as defined in the Design Problem Solver of a decade ago [30]. A view constraint between a pair of objects in, say, the x direction prohibits the presence of a third object between the pair along the x direction. These constraints were not incorporated into our drafter, but a case could be made for their inclusion.

The violation of the example view constraint mentioned is checked by finding whether a third object shares the same y coordinate as the pair of objects, and whether its x coordinate is constrained to be between the pair. For a single view constraint, this is done in linear time. The extraction of a nogood set that accounts for a violation is straightforward.

To summarize, there are two types of contradiction in our drafter: a transitive contradiction (or transit), and a clash. A transitive contradiction is a logical absurdity which can be interpreted to mean that, by virtue of the constraints, an object is above itself. A transit is associated with an axis, either x or y. This is manifested in the graph representation of constraints by a closed loop having a positive sum of edge weights. The nogood reported for this contradiction is the set of constraints along this closed loop.

A clash occurs when two objects forcibly share the same location in the plane. A clash is associated with both axes, and is manifested in the graph representation by a zero-sum path from each object to the other along both axes. The nogood for a clash consists of the inequalities along all four zero-sum paths.

A fuller discussion of the algorithms used in our program, together with theorems about their incremental properties, is in MIT AI Lab Memo 875 written by this author [44].

5.6. The Search Regimen

The program applies each of the features detected in the circuit, in order to constrain the possible layouts. The features are sorted by their perceived importance, using a static assignment of priority developed by hand. The search regimen adopted here selects a feature at each turn, and posts one of the sets of constraints, out of several possible, by adding the inequalities to the collection. There are several sets of constraints to choose from because each feature makes a disjunctive statement about the partial layout.

After a set of constraints is posted, we check for the presence of a contradiction (a clash or a transit).

[11] If there is no transitive contradiction, and if the longest path from any object to itself along a dimension is negative, then there are no clashes.

If there is none, then search proceeds with the next feature in the sorted list. When all the features have been applied to the partial layout, a single schematic is extracted from the class of schematics determined by the constraints, by assigning integers such that the constraints are fulfilled.

We use this search regimen of *iterative constraint* [13] for two reasons. First, in general, the constraints suggested by a feature depend on the current partial layout, so that it makes little sense to apply all the features together en masse. Moreover, the fixes of contradictions (see next section) also depend on the current partial layout.

Second, it is desirable to notice a conflict between two features when the partial layout is as unconstrained as possible, i.e. when a small number of features have already been applied. The reason is that fixes to these conflicts are applicable only when the rather localized pre-conditions of these fixes are satisfied. For example, a fix may have some knowledge of how to correct a typical conflict between a **tree** feature and a **crown**. If a conflict among the constraints contributed by these features is obscured by *other conflicts, then the preconditions of the fix may not be fulfilled.*

5.7. Fixes: Resolving Contradictions

Domain knowledge is also embedded in the fixes to conflicts between features. This knowledge is not incorporable into the generator of partial layouts because it is by nature a repair to a problem. Moreover, it is not desirable to transform the fixes to a conflict between features into precatalogued choices of these features because doing so would enlarge greatly the number of disjuncts attached to a feature, thereby increasing greatly the search tree.

Fixes exploit the richness of information available when a conflict occurs. In our program, the information available is the following:

1. **The type of contradiction (whether clash or transit).** The type of contradiction matters because removing a clash is different than removing a transit, because their physical interpretations differ.
2. **If a transit, whether it occurred along the x or y axis.** Whether a transit occurs in the x or y axis also changes the way a conflict is handled. One gate is to the right of another because the signal path to the first gate is longer. Often a transit in the x direction is repaired by stretching a wire i.e. allowing a greater distance between two gates that originally were required to be a unit distance apart.

   In our experience, transits along the y axis are sometimes repaired by a fix that concerns a stereotypical conflict between two features. More often, the conflict is somewhat complex, involving highly incompatible feature choices, so backtracking must be done.
3. **The identity of the last feature applied.** This information is also useful to the writer of the fixes. By focusing on the conflict introduced by the last feature, one can write a specialized fix that handles typical conflicts caused by certain features.
4. **The set (nogood) of features responsible for the conflict (includes the last feature applied).** An example of the use of a nogood set to fix a conflict is the case where a conflict only involves a crown and a tree, as shown in Figure 5-3a. It is possible to remove this transit in the y direction by reconfiguring the crown as in Figure 5-3b.[12] **If a transit, the sum of the edge weights along the closed path.** If the sum is 1, then one can eliminate

[12]We remind the reader that the three configurations shown in Figure 5-3 are chosen by the programmer, and reflects his taste. The components could be arranged differently by changing the constraints associated with the features and the fixes.

this transit by decrementing one of the negative edge weights along the path, if one exists and the action is permissible.[13] This corresponds to incrementing the permissible maximal distance between two gates.

5. **The partial layout before the conflict occurred.** The partial layout serves, for example, to choose within a fix one option over another. The difference between the two options may be that one works better when some GATE-1 is above GATE-2, and the partial solution may be queried for this information.

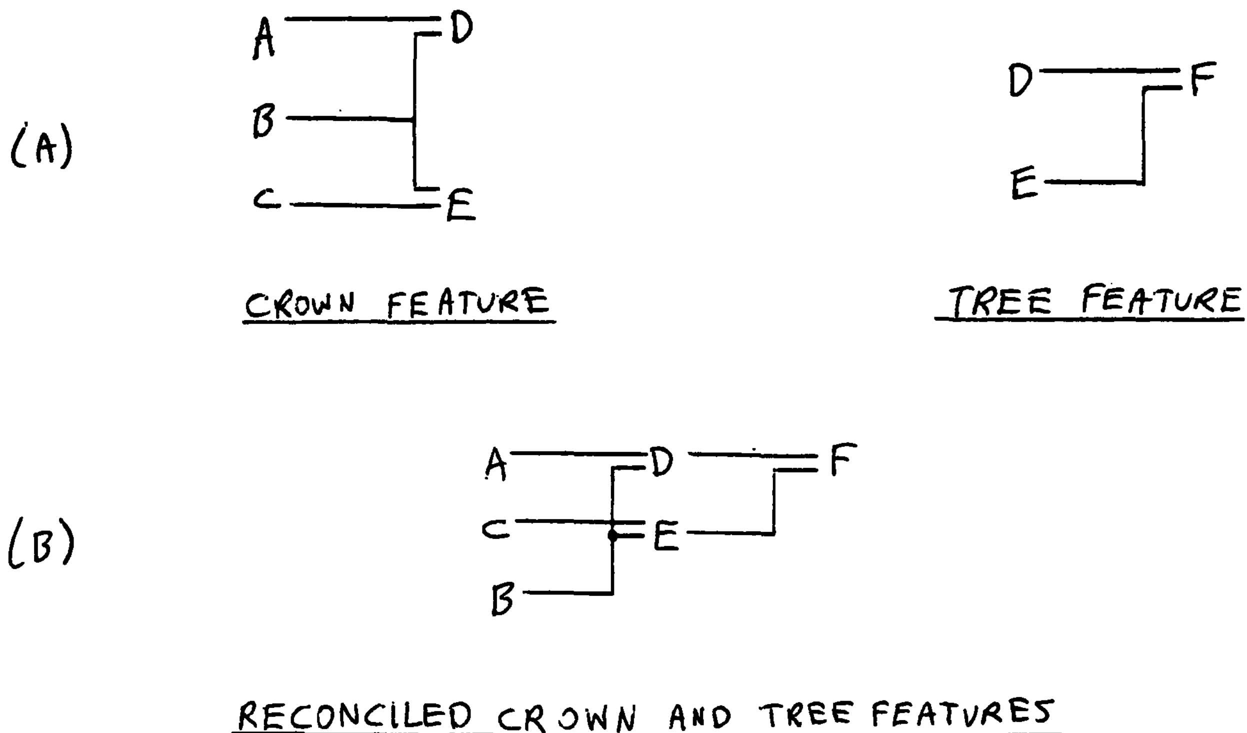


Figure 5-3: Fixing A Conflicting Crown and Tree

5.8. System Diagram

Figure 5-4 shows the block diagram of the program architecture. The key aspects of the diagram are as follows.

Success occurs when there are no more features to apply to the partial layout. At that point, the set of constraints constituting the solution determine several possible layouts. From these, a single layout is extracted by assigning, along each axis, the smallest integers to the gates such that all the constraints are fulfilled.

The search fails when for some feature each of its choices, whether catalogued or inserted by a fix, leads eventually to an insoluble contradiction. Speaking algorithmically, the program backtracks to a feature which has exhausted its choices. Currently, the program simply signals failure and stops, and at that point one looks at the traces of contradictions to identify what knowledge can be added to the

[13] An assertion attached to a constraint says whether it is permissible to decrement the constraint weight. This is almost always for constraints that stipulate a maximal distance between two gates of one unit.

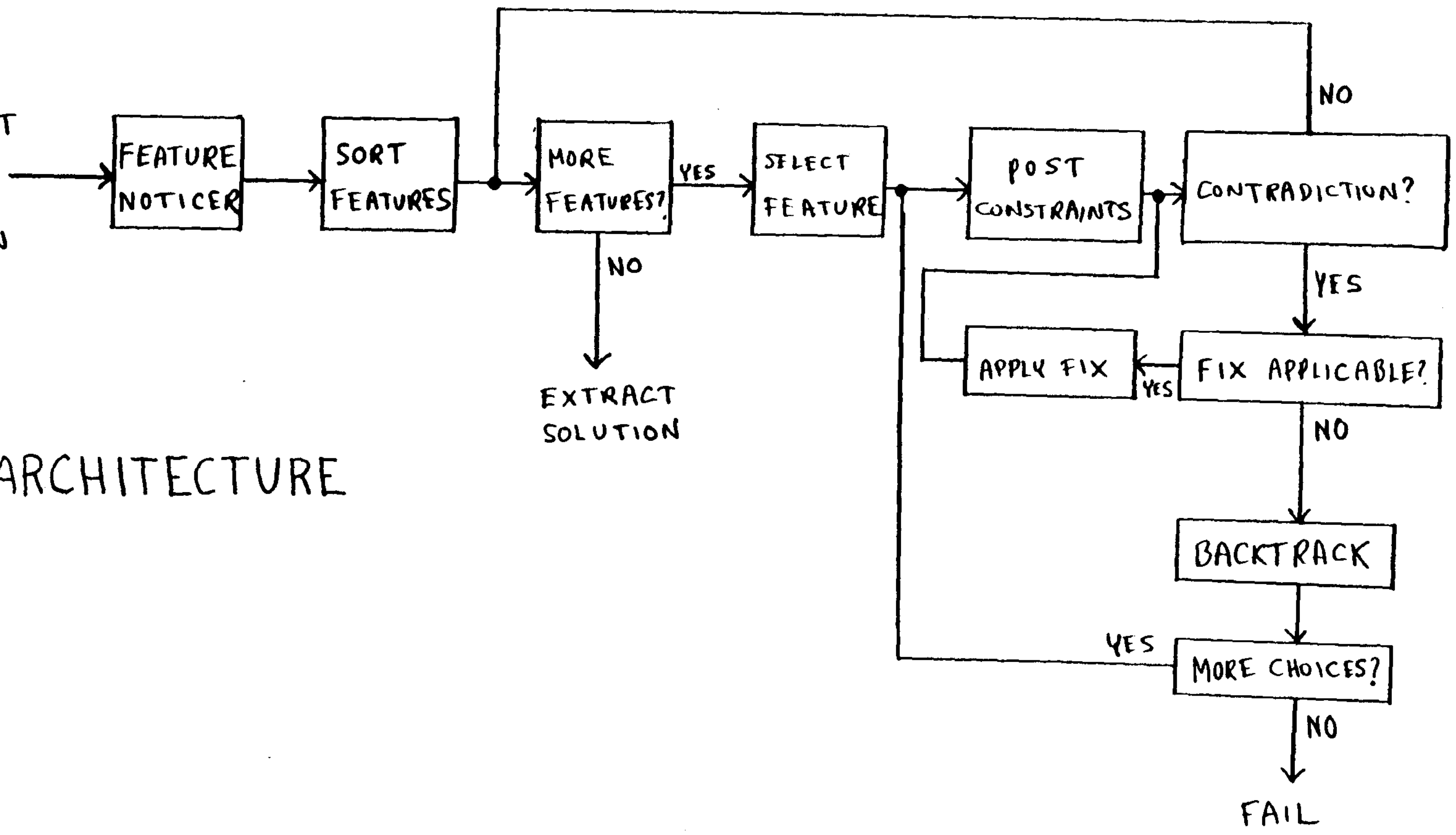


Figure 5-4: System Block Diagram

features or fixes to correct that type of failure in the future. In a deliverable system, one could maintain the "best" partial layout seen so far, and return that upon "failure."

The first block notices the presence of features in the circuit and assertions about function, and makes a list of them. Note that there is no restriction that a feature occur only once in the circuit. Next the features are sorted according to fixed priorities.

There are three simple cycles in the diagram. One is the smooth-sailing cycle where features are repeatedly selected and applied to the partial layout. This cyclic behavior stops when there are no more features to apply.

Another cycle is where a fix corrects the contradiction but introduces another conflict, whereupon the fixes are re-applied thus re-introducing another conflict ad infinitum. In theory, this cycle might never end, and can be interpreted as a drawback to our architecture. This has never happened in our experience, but certainly a commercial program should attend to this. However, forward chaining rule-based systems are also subject to this cycle, and that has not impeded their practical use.

The last cycle involves backtracking, where after each backtrack a new conflict occurs, no fixes apply, and backtracking occurs again.[14] This cycle must end because the number of features and catalogued choices for each feature is finite.

5.9. The Backtracking Scheme

A notable aspect of this research is the backtracking scheme. To understand the scheme, we first highlight the relevant aspects of the search regimen.

There is a prioritized list of features detected in the circuit to be drafted. The application of a feature to the partial layout involves the selection of one of the choices associated with that feature. In general, these choices depend in turn on the partial layout that is current when the feature is applied. The consistency of the partial layout is tested immediately after the application of a feature.

Given this search strategy, whenever an insoluble conflict arises, the obvious remedy is to "backtrack" to the last feature applied and try another choice. The remaining question is what to do when the choices at this feature are exhausted. Evidently, the search must at that point backtrack to a previous feature.

The only information available to guide the backtracking to previous features is the record of contradictions recorded at the node whose choices have been exhausted. A nogood is recorded at this node each time that its then current choice was involved in a contradiction.

We propose to use this information from nogoods optimally, thus avoiding fruitless search resulting from overly conservative backtracking. In the following, we shall: 1. state the algorithm; 2. explain the algorithm; and 3. justify its correctness.

Assume that associated with every feature there is a variable *backtrack-list* that suggests backtrack destinations from the feature.

1. Let **current-backtrack-list** denote the backtrack-list of the current feature. Let **nearest** be the feature in **current-backtrack-list** that is temporally nearest the current feature (i.e. that

[14] Here we consider backtracking to include the case where a new choice is tried at the last feature applied.

feature that was applied latest).

2. Reset the choices at each feature that was applied after **nearest**.
3. For feature **nearest**, set its backtrack-list to the union of itself with the **current-backtrack-list**, after first removing **nearest** from the latter.
4. Now proceed normally with the search from feature **nearest**.

All of the features skipped over during the backtrack must be reset as indicated, because these choices depended on the partial layout which will no longer exist. The above is one way that the backtrack-list of a feature is updated.

The other way to update the backtrack-list is through the trivial "backtracking" after a contradiction that occurs when the search simply tries another choice for the feature most recently applied (the current feature). The algorithm for this is as follows.

1. Obtain the set of conflicting features; this set includes the current feature.
2. Delete the current feature from this set, and perform the union of the result with the backtrack-list for the current feature. This union becomes the new backtrack-list for the current feature.

Note that, although backtracking to the same current-feature could be handled similarly to backtracking to previous features, we treat them differently in the code for clarity. While examining the current feature, when a choice conflicts with the current partial layout, the search simply updates the backtrack-list of the current feature and the next choice is tried.

If this backtracking scheme is correct, it should guarantee completeness in the sense that if a solution exists, it won't be missed somehow. If the scheme is good, it should make maximal use of the information that contradictions yield.

We can justify the scheme by appealing to resolution [50]. Two insights are necessary. First, any nogood set {A B C} implies the propositional statement:

$\neg A \vee \neg B \vee \neg C.$

The second insight is that if there are *k* choices for a node N in a search tree, and there exists a solution to the search, then the following disjunction is true:

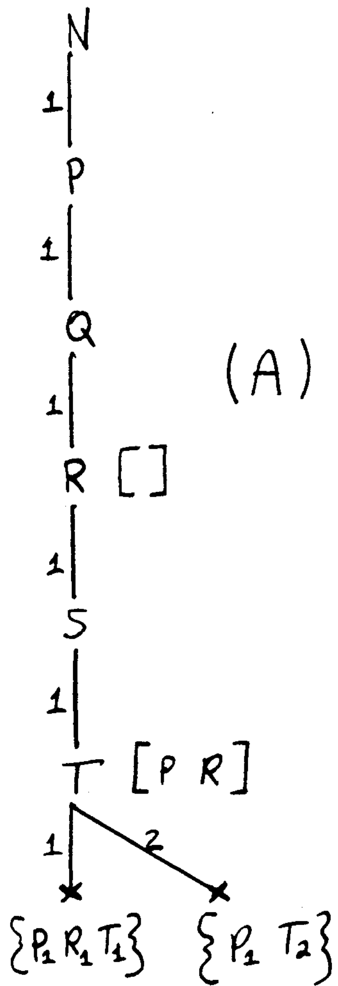
$N_1 \vee N_2 \vee \ldots N_k.$

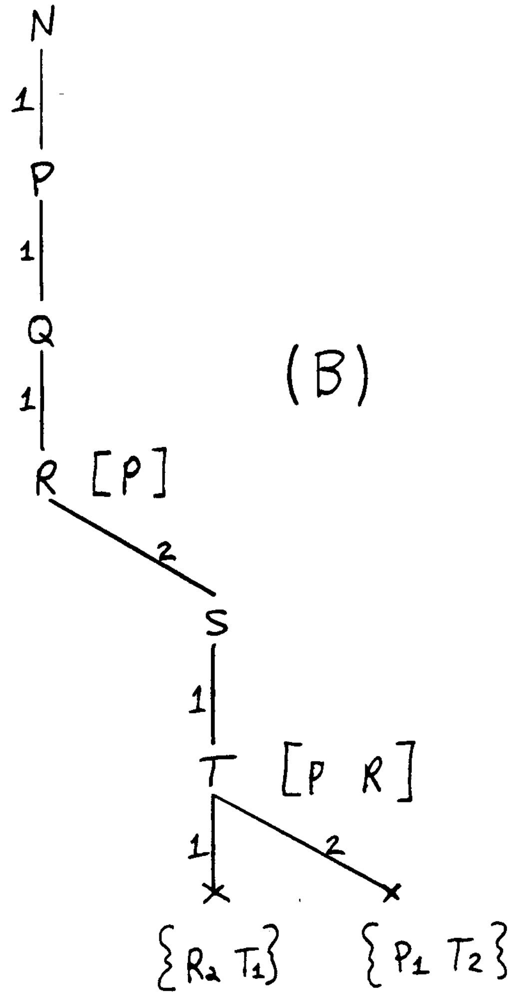
Figure 5-5 illustrates the iterative constraint search where a contradiction occurs each time that a choice for *T* is made. In squiggly brackets are shown the nogood sets that arise from each contradiction at a node marked with an X, and in square brackets are the backtrack-lists.

In Figure 5-5a, the choices at node *T* have been exhausted, so the search will backtrack to *R*, the nearer of features *P* and *R*. At *R* the backtrack-list is updated from *[]* to *[P]*. Another choice R_2 is made and search proceeds until, for example, both T_1 and T_2 fail again. The backtrack-list at that point is *[P R]*, so the search backtracks to *R* once more, and *R*'s backtrack-list is updated by the union operation, but left unchanged this time. By resolving the various expressions, one obtains the proposition $\neg P_1$, so the search backtracks from *R*, which has exhausted its choices, to *P*, in order to try *P*'s second choice.

Evidently the search based on this backtracking scheme will completely cover the search space, in the sense that if a solution exists it will not be missed. Also there seems no way to further prune the search space using the information from the nogoods, unless one resorts to a truth-maintenance

Figure 5-5: The Backtracking Scheme





$\overline{P_1} \vee \overline{R_1} \vee \overline{T_1}$	; first nogood	$\overline{R_2} \vee \overline{T_1}$	; first nogood
$\overline{P_1} \vee \overline{T_2}$	; second nogood	$\overline{P_1} \vee \overline{T_2}$	; second nogood
$\overline{T_1} \vee \overline{T_2}$	; T has two choices	$\overline{T_1} \vee \overline{T_2}$	; T has two choices
(I) $\overline{P_1} \vee \overline{R_1}$	; by resolution	(II) $\overline{P_1} \vee \overline{R_2}$	; by resolution

Resolving (I) and (II) with $R_1 \vee R_2$

gives: $\overline{P_1}$

approach and its attendant complexities [10].

The formalized justification presented here does not precisely reflect our architecture, because of the fixes that can insert choices spontaneously. Disjunctions of the type $T_1 \vee T_2$ have in theory an unbounded number of disjuncts, obtained by composing the fixes. To give a formal justification of the backtracking scheme, we would need first-order predicate calculus statements. Instead we shall rely on the intuitive correctness of the approach for this more complex case.

5.10. Summary

The program described here takes as input the topology of a circuit plus assertions regarding function and the nature of signals. The output is the placement of each gate and primary signal on a tessellated grid, by means of a pair of x and y coordinates.

As befits a knowledge-based program, the domain knowledge is distinct from the control of the search. Domain knowledge resides in two places. Features are topological patterns or functional relationships that assert several possible sets of constraints. One of these sets is applied to the partial layout, and the others are tried when the preceding attempts fail.

The first stage detects the presence of features in the input circuit. The drafting proceeds by successively choosing a feature and applying one of its choices to a partial layout. When a contradiction arises, other specialized domain knowledge is invoked to attempt a fix to the conflict by changing the constraints suggested by one or more features. This effectively introduces new choices for features spontaneously.

When an insoluble conflict remains, the search backtracks using the information collected during the search. This backtracking scheme is dependency-directed in the simple sense of being guided by those decisions that contribute to a conflict. The backtracking scheme is very simple and well-suited to an iterative constraint regimen, where the choice at each node depends on the previous choices.

6. Performance Analysis

6.1. Why Does The Program Work?

The program drafts arithmetic digital circuits competently. This author contends that the program works because it captures a part of the reasoning that people do as they draft a circuit.[15] People use more than topological information while drafting; they also employ information about function and heuristics, such as threading the bits of two integers that are summed. Moreover, people are good at reconciling conflicting desires, and they seem not to do it by ordinary backtracking and search. Elsewhere, protocols on architectural design have yielded the following observations:

> **The real difficulty of integrating solutions lies in resolution of conflicts. ... conflicts are resolved either by remodifying the physical description or by modifying the problem criteria violated ...**

although this problem-solving behavior, concerned with fixes to conflicts, was not highlighted by the

[15]The section on program competence discusses those drafting skills that are not emulated.

author [4].

A recent on frameworks for knowledge-based design [26] identifies four kinds of knowledge used by the process of design. Their observations result from the attempt to build a program that designs the paper transports of photocopiers.

> **A major piece of knowledge that expert designers seem to use when the design fails some acceptability condition (constraint) is how to modify the design. Consider a dependency-directed backtracking problem solver in contrast. It knows enough to back up to a relevant decision point but does not have any way of deciding how to modify its decision. Good designers, on the other hand, not only know where the relevant prior decision points are but also analyse the failure to decide how to modify their past decisions. Being able to advise a prior decision point (and a problem solver in general) is crucial in reducing the search. In the best case, the advice would enable a previous decision to be modified in exactly the way needed to fix the current constraint failure.**

Our framework integrates naturally repair knowledge of this kind.

It is advantageous, whenever possible, for a program to search over a small space of choices and fix conflicts, rather than search over a larger space of choices and backtrack.

6.2. Program Competence - Knowledge

The program drafts arithmetic digital circuits that have a single purpose, and that contain no feedback loops.[16] We claim that the terms *arithmetic* and *digital* reflect the knowledge that the program *has*, not what its limitations are. We see no reason why drafting analog circuits or even general block diagrams cannot be drawn with this approach, because the domain knowledge in the program is explicit and therefore removable. For example, knowledge of conventional left-to-right signal flow is explicit in the flow feature, which states that if AND-1 feeds AND-2, then AND-2 should be to the right of AND-1.

6.3. Program Competence - Architecture

Circuits that have multiple purposes are beyond the competence of this program, for fundamental architectural reasons. Such circuits contain subcircuits that perform individually meaningful functions. In order to highlight such meaningful units, and present coherently their interaction, some global schematic *plan* is needed, and this is beyond the task tackled by our program.

Note that the problem is not of *finding* the abstractions and corresponding subcircuits. Even assuming that these are given by hand, the problem of the formulation of a plan remains. The 4-bit ALU in Figure 6-1, taken from a textbook [18] and drawn presumably by hand, hints at the need for such a capability to plan.

Conceivably the program developed here can serve as a "subroutine" for a larger drafter, which identifies meaningful subcircuits, plans the overall schematic by assigning regions, and selects the knowledge base pertinent to each subcircuit.

---

[16]One way to draft circuits with feedback is to break all closed loops by ignoring one of their connections, and computing the gate placement of the resulting circuit. These connections are included during routing. However, it is not obvious in general which connection to consider feedback.

Figure 6-1: 4 Bit Arithmetic and Logic Unit

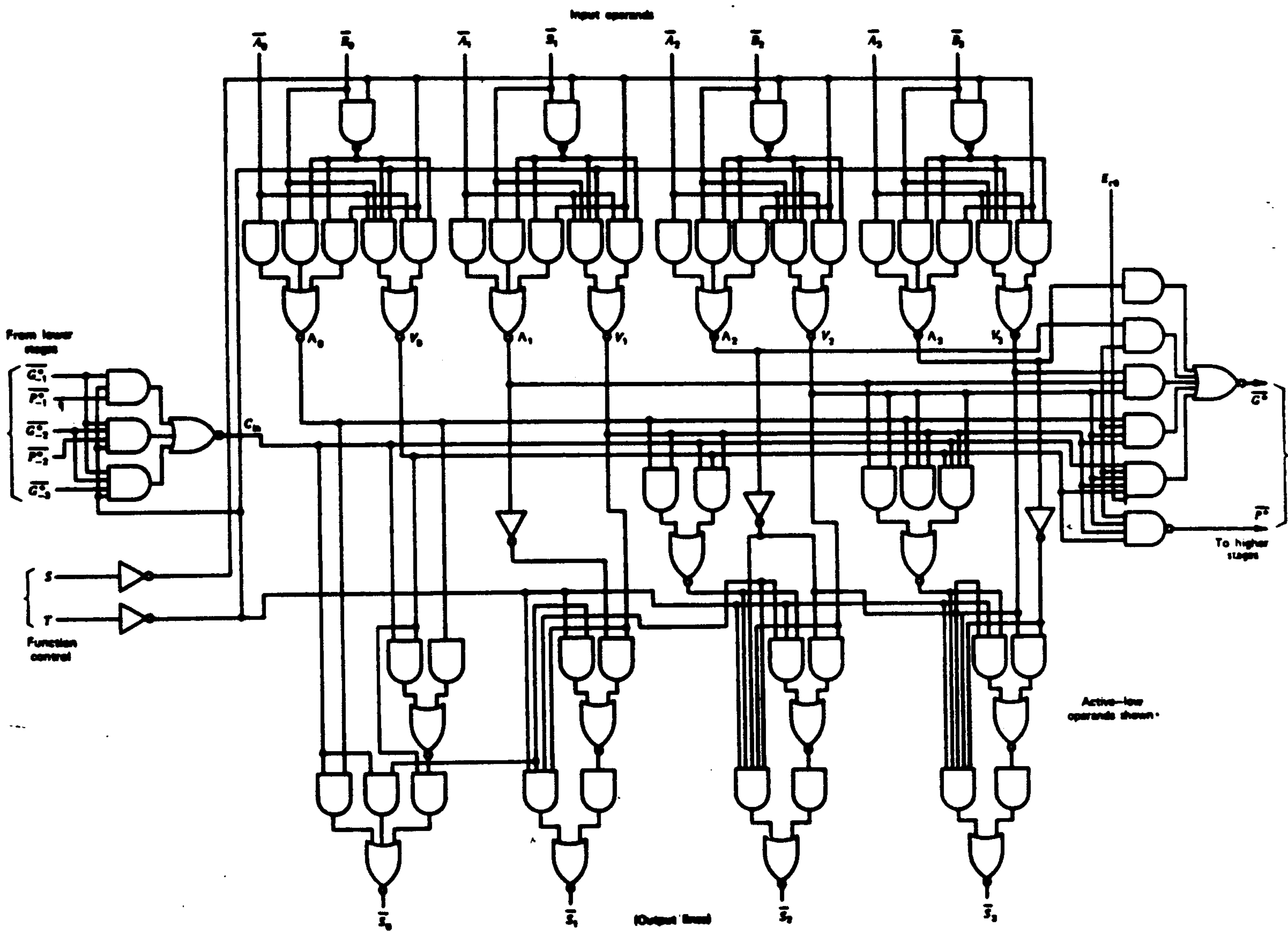


Figure 4.9 The schematic logic diagram of a 4-bit ALU with carry lookahead as specified in Figure 4.8.

6.4. Work To Be Done

The immediate need is for some graphics programming and a simple router to complete the drafting task, and the author expects to finish this.

Other work of a research nature would be the application of this architecture to another problem in space-planning, or even design in general.

Finally, as discussed in the previous section, the task of planning the schematic for a circuit having multiple meaningful subcircuits, or even a hierarchy of subcircuits, is challenging and needs attention.

7. Conclusion

7.1. Achievements Of This Research

The original purpose of this work was to apply the lessons available from knowledge-based systems to the task of schematics drafting. Drafting was approached as a problem of design, and more specifically as a configuration or space-planning problem.

The outcome is a program that has given evidence, in the form of the layouts described in this report, of drafting competently within the domain of arithmetic digital circuits. The layouts presented here are For this outcome to be generally useful, the architecture should be sufficiently understandable so that it may be applied again for other types of schematics, or even for other layout problems.

Accordingly, we next extract from our program an abstract description of its behavior, independent of the algorithms that implement the behavior.

7.2. A Computational Theory of Drafting

Given a catalogue of schematics features, a program can draft a schematic competently by first detecting the possible presence of these features in the circuit to be drafted.

Each feature constrains the possible layouts in any of several disjunctive ways. These disjunctions constitute a search space that is an AND-OR tree of depth two. This search space is not static at the OR level, because in general the number and character of the disjuncts depend on the state of the search (partial layout) so far.

The program then searches through this search space by *iterative constraint*, fixes eventual conflicts by the use of knowledge specialized for this task, and backtracks intelligently when the disjuncts from features conflict insolubly. The repair of conflicts can be thought of as the spontaneous insertion of disjuncts at the OR level of the tree.

7.3. Notable Aspects

There are several aspects of this research that contribute notably to the storehouse of AI techniques. These are:

1. A knowledge-based system that designs schematic layout using an architecture that maintains a distinction between its domain knowledge and the search framework.
2. A modern approach to space planning that exploits the concepts of dependency-directed backtracking [38] and constraint posting [40] or least commitment search.[17]
3. The idea of contradiction-fixing rules that exploit the richness of available information when a conflict occurs.
4. A simple backtracking approach that is suitable for an iterative constraint search regimen.
5. The study of a inequality-based constraint language and its algorithmic properties, including expressivity and incremental behavior. This topic is more fully explored in a previous paper [44].

Acknowledgment

I wish to thank Professor Randall Davis for his guidance, and for encouraging me to work on this project. I am grateful to all the members of his Hardware Troubleshooting group at the MIT Artificial Intelligence Laboratory for all the excellent discussions.

Thanks for insight on some of the algorithms used here go to Professor Charles Leiserson and to Cindy Phillips, both of the Laboratory for Computer Science at MIT.

Clint Bissel of the Silicon Systems Engineering group at Digital Equipment Corporation in Hudson, Massachusetts graciously discussed the role of schematics in circuit design and showed a working program used for that purpose at Digital.

[17]There is much confusion about exactly what is dependency-directed backtracking. Stallman & Sussman presented a scheme which provides both intelligent backtracking and intelligent forward search. The backtracking is intelligent because it returns to a choice-point which contributes a constraint that participates in the contradiction. The forward search is intelligent in the sense that those contexts will not be explored that are provably fruitless using the information obtained previously from contradictions. The designation "dependency-directed backtracking" hints at something about backward behavior, not forward, whence the confusion.

Here is one interpretation of dependency-directed backtracking [46]:

> **An alternative to chronological backtracking where the backtrack point (the choice point that control is passed back to on failure) is determined by the nature of the failure. That is, the choice that caused the failure is undone whereas in chronological backtracking it is simply the last choice that is reconsidered.**

This only speaks of the backward behavior. The AI Handbook [5] implies that dependency-directed backtracking is simply synonymous with nonchronological backtracking.

I. The Features

The features used by our program are listed here:

`addition`	`crown`	`mapping`
`arrow`	`fanout`	`primary-sigs`
`bifeed`	`fanout2`	`thread`
`candelabrum`	`flow`	`tree`
`carry`	`line-fanin`	`tripod`
`composite`		

Next we define each feature verbally, and afterwards show graphically the disjunctive constraints that each feature suggests.

I.1. Feature Definitions

There are currently 16 features used by the program. Of these, 11 concern only the connections or topology of the circuit, and can be viewed as general features of a directed graph. The remaining 5 features concern information about function and the nature of signals.

First we itemize the extra-topological features. Each item defines the feature, and then describes the constraints that the feature suggests.

1. **addition**. Two (abstract) primary input signals are summed by the circuit. By abstract is meant that each addend signal can be composed of several subsignals. For example signals A and B are added, but each is composed of 4 bits.

   Each bit of the sum is placed vertically between the corresponding bits of the two summands.

2. **carry**. Signal C is the carry or overflow of signals $\{X_1,X_2,...,X_n\}$.

   The carry is vertically above (or below) all of the bits X_i.

3. **composite**. The signals $\{X_1,X_2,...,X_n\}$ constitute a meaningful ordered abstract signal e.g. an integer.

   The signals are constrained to be vertically in either increasing or decreasing order.

4. **mapping**. The signals $\{X_1,X_2,...,X_n\}$ are "transformed" into $\{Y_1,Y_2,...,Y_n\}$ by some operations. For example, an incrementer circuit transforms an integer into another integer of approximately the same size.

   The vertical order of the input and output signals should agree.

5. **thread**. Two abstract signals X and Y, each composed of subsignals, are combined arithmetically in the circuit.

   The subsignals of X and Y should be vertically threaded thus: $x_0\ y_0\ x_1\ y_1\ x_2$ ... There are four ways to thread the signals, depending on whether the order is decreasing or increasing, and on which signal goes first in the thread.

Next are the topological features, which refer only to the (directed) graph properties of the circuit. "Gates" are interpreted to include both devices and primary signals.

1. **arrow**. Gate A feeds only gate B. Gate B is fed only by gate A.

Gates A and B share the same vertical coordinate, and gate B is one unit ahead of gate A.

2. **bifeed.** This feature involves 4 gates: A,B,C,D. Each of gates A and B feeds only and both of gates C and D. Gates C and D are fed by no other gates.

   These four gates are layed out as shown in Figure I-1a. There are 4 ways to configure a bifeed, obtained by permuting the gates.

3. **candelabrum**. One gate G is fed by N>3 gates, and these N gates feed no other gates besides G.

   There are 2N ways to configure the gates involved in this feature. One of the N gates is chosen as central, and its vertical position is aligned with that of G. The rest of the N-1 gates go either all vertically above or all vertically below the central gate. This is illustrated in Figure I-1b.

4. **crown**. Two gates at the head are fed by three gates at the tail. The three tail gates feed no other gates, and the head gates are fed by no other gates. One of the three tail gates (the center) feeds both of the heads, and the other two tail gates (the wings) feed only one of the heads.

   Please refer to Figure I-1c for an example configuration. By permuting the heads and wings, one obtains two ways to configure a crown.

5. **fanout**. Gate A feeds N>2 gates, which are fed by no other gate.

   The N gates are all either vertically above (or equal to) gate A, or they are vertically below (or equal to) gate A.

6. **fanout2**. Gate A feeds exactly 2 gates, neither of which is fed by any other gate.

   There are 4 ways to configure this feature. One of the 2 fed gates is selected to align vertically with A, and the remaining gate is placed either above or below the other two.

7. **flow**. A signal originating from gate A feeds N gates.

   Each of the N gates is placed horizontally to the right of gate A.

8. **line-fanin**. Gate A feeds only gate B, which could be fed by other gates.

   This feature vertically aligns A and B, and places B to the right horizontally one unit.

9. **primary-sigs**. The signals involved in this feature are the primary inputs and outputs of the circuit.

   The input primary signals share the same horizontal coordinate, as do the output primary signals.

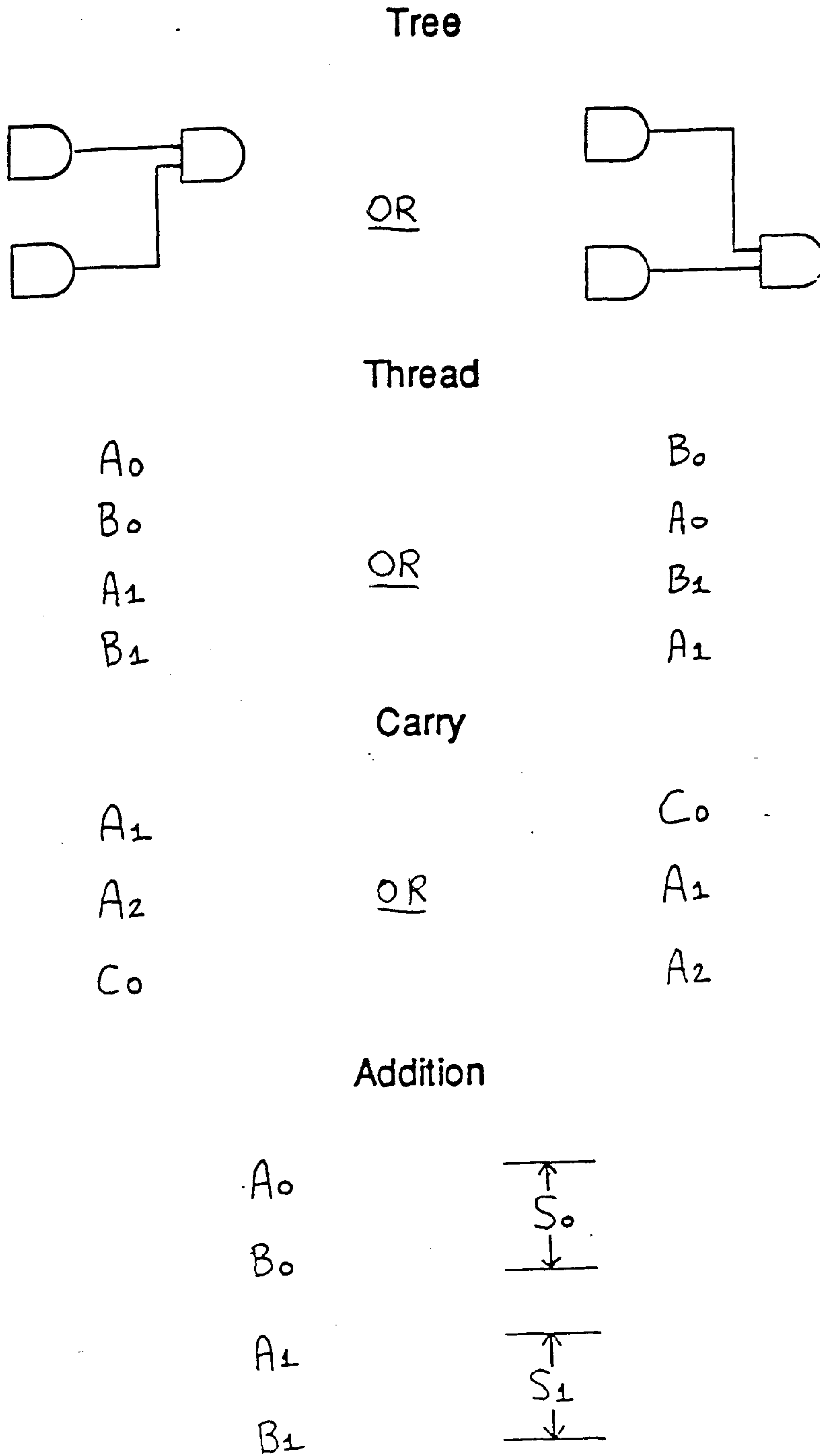
10. **tree**. Gates A and B feed only gate C, which is fed by no other gate.

    We draw trees asymmetrically by vertically aligning one of the two feeding gates with the head. The other gate is then placed vertically above or below the centered gate. This gives 4 ways to configure a tree. One way is shown in Figure I-1e.

11. **tripod**. A head gate G is fed only by 3 gates, which in turn feed no other gate besides G.

    This feature is draw symmetrically, by centering one of the 3 feeders and placing on each wing one of the remaining two gates. There are 6 ways to configure a tripod.

Figure I-1: Example Feature Configurations



References

1. M.L. Ahlstrom, G.D. Hadden, and G.R. Stroick. An Expert System for the Generation of Schematics. Proceedings IEEE Int. Conf. on Computer Design, 1984, pp. 720-725.

2. M.L. Ahlstrom, G.D. Hadden, and G.R. Stroick. HAL: A Heuristic Approach to Schematic Generation. IEEE Int. Conf. on Computer Aided Design, 1984, pp. 83-86.

3. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

4. O. Akin. How Do Architects Design? In *Artificial Intelligence and Pattern Recognition in Computer Aided Design*, J.C. Latombe, Ed., North Holland, 1978.

5. A. Barr and E.A. Feigenbaum. *The Handbook of Artificial Intelligence.* William Kaufmann Inc., 1982. Top of page 74.

6. G.D. Birkhoff. *Aesthetic Measure.* Harvard University Press, 1933.

7. W.J. Clancey. Classification Problem Solving. Proceedings of AAAI, 1984, pp. 49-55.

8. R. Davis. Problem Selection Criteria. Distributed as part of course on Knowledge-Based Applications Systems, March 1985.

9. R. Davis. Expert Systems: Where Are We? And Where Do We Go From Here? Memo 665, MIT AI Lab, 1982.

10. J. de Kleer. "An Assumption-based TMS". *Artificial Intelligence 28* (1986), 127-162.

11. N. Deo. *Graph Theory with Applications to Engineering and Computer Science.* Prentice Hall, 1974.

12. Department of Defense. Military Standard Engineering Drawing Practices. DOD-STD-100C, 22 December 1978.

13. Y. Descotte and J.C. Latombe. "Making Compromises among Antagonist Constraints in a Planner". *Artificial Intelligence 27* (1985), 183-217.

14. H.J. Eysenck. *Sense and Nonsense in Psychology.* Penguin, 1972. Chapter 8.

15. J. Gait. "An Aspect of Aesthetics in Human-Computer Communications: Pretty Windows". *IEEE Transactions on Software Engineering SE-11* (August 1985), 714-717.

16. J.G. Gay, R. Richter, and B.J. Berne. "Component Placement in VLSI Circuits Using a Constant Pressure Monte Carlo Method". *Integration, The VLSI Journal 3* (1985), 271-282.

17. M. Henrion. Automatic Space-Planning: A Post-Mortem? In *Artificial Intelligence and Pattern Recognition in Computer Aided Design*, J.C. Latombe, Ed., North Holland, 1978.

18. K. Hwang. *Computer Arithmetic.* John Wiley & Sons, 1979.

19. D.B. Johnson. "Efficient Algorithms for Shortest Paths in Sparse Networks". *Journal of the ACM 24*, 1 (Jan 1977), 1-13.

20. R. Joobbani and D.P. Siewiorek. "Weaver: A Knowledge-Based Routing Expert". *IEEE Design & Test 3*, 1 (February 1986), 12-23.

21. H.M. Kalish. "Machine-Aided Preparation of Electrical Diagrams". *Bell Laboratories Record 41*, 9 (1963), 338-345.

22. J.H. Kim, J. McDermott, and D.P. Siewiorek. "Exploiting Domain Knowledge in IC Cell Layout". *IEEE Design & Test 1*, 3 (August 1984), 52-64.

23. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. "Optimization by Simulated Annealing". *Science 220* (1983), 671.

24. A. Kumar, A. Arya, V.V. Swaminathan, and A. Misra. "Automatic Generation of Digital System Schematic Diagrams". *IEEE Design & Test 3*, 1 (February 1986), 58-65.

25. Y.Z. Liao and C.K. Wong. "An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD-2*, 2 (April 1983), 62-69.

26. S. Mittal and A. Araya. A Knowledge-Based Framework for Design. Proceedings of AAAI, 1986, pp. 856-865.

27. F.J. Mowle. *A Systematic Approach to Digital Logic Design.* Addison Wesley, 1976. Page 236, circuit (a).

28. A. Newell. Artificial Intelligence and the Concept of Mind. In *Computer Models of Thought and Language*, R. Schank and K. Colby, Ed., Freeman, 1973.

29. M. Palczewski. Performance of Algorithms for Initial Placement. 21st Design Automation Conference, 1984, pp. 399-404.

30. C. Pfefferkorn. The Design Problem Solver. In *Spatial Synthesis in Computer-Aided Building Design*, C. Eastman, Ed., Halsted Press, 1975.

31. L.B. Protsko, P.G. Sorenson, and J.P. Tremblay. A System for the Automatic Generation of Data Flow Diagrams. Research Report 84-8, Department of Computational Science, University of Saskatchewan, 1984.

32. S.E. Rasmussen. *Experiencing Architecture.* MIT Press, 1982.

33. B.D. Richard. A Standard Cell Initial Placement Strategy. 21st Design Automation Conference, 1984, pp. 392-398.

34. J.A. Roach. The Rectangle Placement Language. 21st Design Automation Conference, 1984, pp. 405-411.

35. J.A. Roach. Managing the Constraints and Restraints found in Symbolic Layout. Ph.D. Thesis Proposal, Rutgers, July 1985.

36. H.A. Simon. Style in Design. In *Spatial Synthesis in Computer-Aided Building Design*, C. Eastman, Ed., Halsted Press, 1975.

37. H.A. Simon. *The Sciences of the Artificial.* MIT Press, 1980. Chapter 3: The Science of Design.

38. R.M. Stallman and G.J. Sussman. "Forward Reasoning and Dependency-directed Backtracking in a System for Computer-Aided Circuit Analysis". *Artificial Intelligence 9* (1977), 135-196.

39. G.L. Steele Jr. The Definition and Implementation of a Computer Programming Language Based on Constraints. AI-TR 595, MIT AI Lab, 1980.

40. M. Stefik. "Planning with Constraints (MOLGEN: Part 1)". *Artificial Intelligence 16* (1981), 111-140.

41. G. Stiny and J. Gips. Shape Grammars and the Generative Specification of Painting and Sculpture. IFIP Congress, 1971.

42. T. Tanaka. Representation and Analysis of Electrical Circuits in a Deductive System. Proceedings of IJCAI-8, 1983, pp. 263-267.

43. I.O. Tou. Automatic Formatting of Logic Schematics. Master Th., Massachusetts Institute of Technology,1984.

44. R.E. Valdes-Perez. Spatio-Temporal Reasoning and Linear Inequalities. Memo 875, MIT AI Lab, 1986.

45. V.V. Venkataraman and C.D. Wilcox. GEMS: An Automatic Layout Tool for Mimola Schematics. 23rd Design Automation Conference, 1986, pp. 131-137.

46. L. Wallen (Contributor). *Catalogue of Artificial Intelligence Tools.* Springer Verlag, 1984.

47. T.M. Weinrich. A Structured Approach to Building Schematic Display Programs. Unpublished manuscript, Rutgers Computer Science Department, 1985.

48. H. Weyl. *Symmetry.* Princeton University Press, 1982.

49. K.E. Wieckert. Automated Drafting of Schematic and Block Diagrams. M.S. Thesis Proposal, MIT AI Lab, April 1982.

50. Brian C. Williams. . Personal Communication.

51. D.F. Wong and C.L. Liu. A New Algorithm For Floorplan Design. 23rd Design Automation Conference, 1986, pp. 101-107.