Algorithmic Aspects of Symbolic Switch Network Analysis*

Randal E. Bryant
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

February, 1987

Abstract

A network of switches controlled by Boolean variables can be represented as a system of Boolean equations. The solution of this system gives a symbolic description of the conducting paths in the network. Gaussian elimination provides an efficient technique for solving sparse systems of Boolean equations. For the class of networks that arise when analyzing digital metal-oxide semiconductor (MOS) circuits, a simple pivot selection rule guarantees that most s switch networks encountered in practice can be solved with $O(s)$ operations. When represented by a directed acyclic graph, the set of Boolean formulas generated by the analysis has total size bounded by the number of operations required by the Gaussian elimination. This paper presents the mathematical basis for systems of Boolean equations, their solution by Gaussian elimination, and data structures and algorithms for representing and manipulating Boolean formulas.

Keywords and phrases: switch networks, symbolic analysis, MOS circuit analysis, Gaussian elimination, series-parallel graphs, Boolean manipulation.

1 Introduction

The advent of metal-oxide semiconductor (MOS) circuit technology has revived interest in analyzing networks of switches. This field originated when digital circuits were constructed with electromechanical relays. Shannon, in the first application of Boolean algebra to digital systems, developed several techniques for analyzing a switch network symbolically[1]. For a network of switches, each of which is either open or closed depending on the value of some Boolean variable, the goal of symbolic analysis is to derive formulas expressing the conditions under which conducting paths will exist between specified pairs of terminals.[1]

*This research was supported in part by the Defense Advanced Research Projects Agency, ARPA Order Number 4976, and in part by the Semiconductor Research Corporation under Contract 86-01-068.

[1]Shannon characterized a network by its "hindrance" function, with logic value 1 indicating the absence of any path. This paper adopts the more conventional view of 1 indicating the presence of a path.

A MOS transistor can often be abstracted as a switch—it conditionally forms a connection between its source and drain nodes depending on the voltage on its gate node. Several models of static logic gates in MOS treat transistors as simple switches and define the behavior of a gate in terms of the conditions under which a conducting path is formed from the supply or ground to the gate output[2, 3, 4]. More complex MOS models take into account such effects as resistance ratios, dynamic memory, and invalid or uninitialized logic values[5, 6]. A companion paper [7] shows that even with these more elaborate models, the behavior of a MOS circuit can be determined by analyzing a series of switch networks. Thus the symbolic analysis of switch networks remains as a key problem in deriving an abstract representation of the function computed by a digital circuit.

The demands placed upon symbolic analysis have changed greatly since the days of relay circuits. These circuits were relatively small, and the analysis was performed manually. Under these conditions, the asymptotic performance of the method matters less than its conceptual simplicity. For example, Shannon describes a method that involves enumerating every possible simple path between two terminals, forming the Boolean product of the switch labels in each path, and then forming the Boolean sum of these path formulas. In general, the number of simple paths in a network can grow exponentially with the number of switches (e.g., the parity ladder shown in Figure 7.) Consequently, path analysis cannot be applied to networks of significant size. Furthermore, there was no concern about data structures and algorithms for representing and manipulating Boolean formulas. Even more recent methods based on matrix representations[8] do not address these algorithmic issues.

Today symbolic analysis methods are to be executed by computers on networks containing thousands of switches. To implement an analyzer, every detail of representation and algorithm must be specified. Success must be measured by worst or average case complexity rather than by performance on small examples. Unfortunately, the state of the art in symbolic analysis has not kept up with these demands. For example, most published symbolic analysis methods for MOS circuits start by enumerating all possible simple paths in the network[9, 10, 11]. A second method involves enumerating the possible sets of connected components formed in the switch network for different values of the control variables [12]. This approach can also produce a description of size exponential in the number of transistors. These accounts indicate little progress since Shannon's day.

In general, a MOS circuit can be partitioned into smaller subnetworks and each subnetwork analyzed separately. Most of these subnetworks are small—containing no more than 10 transistors. Hence, one can argue that even an algorithm of exponential complexity can work well in practice. However, subnetworks containing over 1000 transistors commonly occur in large pass transistor and datapath circuits. A program for general use cannot rely so heavily on a particular circuit style to achieve tolerable efficiency.

This paper proposes a far more exacting standard for symbolic analysis: that the size of the symbolic description should be comparable to the size of the original network. That is, a network of s switches should be represented by a set of formulas containing, in total, $O(s)$ Boolean operations. This paper shows how to achieve this goal for most networks arising in the analysis of MOS circuits. Even for the dense pass transistor circuits that lead to a nonlinear complexity, the method produces a description of size $O(s^{3/2})$. This performance results from a combination of several techniques:

- A network is represented by a system of Boolean equations. This system expresses the effects of all paths in the network but lends itself to solution methods of polynomial complexity.

- The system of equation is solved symbolically by Gaussian elimination. A simple heuristic for selecting pivots guarantees that most practical networks of s switches can be solved with $O(s)$ algebraic operations.

- The set of Boolean formulas is represented by a directed acyclic graph (DAG), with each DAG node specifying a Boolean operation to be applied to its children, and with each leaf denoting a variable or constant. This representation naturally allows sharing of common subexpressions. The number of nodes in the DAG is bounded by the number of algebraic operations required during Gaussian elimination.

- Expression simplification techniques are applied to the DAG but only in ways that reduce the size of the DAG even further.

The algorithm presented is efficient in terms of execution speed as well as the size of the result produced.

The formulation of the switch network analysis problem used in this paper is tailored to the particular needs of the MOS circuit analysis method presented in the companion paper. It differs from the classic formulation in the following respects:

- The control signals of switches can be arbitrary Boolean formulas, not just variables or their complements.

- Switches are directed—the conduction conditions from one point to another can differ from those in the reverse direction.

- The analysis does not compute the conduction conditions between specified pairs of terminals. Instead, each node is given an initial value represented by a Boolean formula. A path is viewed as having an "effect" on its destination node equal to the Boolean product of formulas representing the initial value on the source node and the control signals of the switches. Symbolic analysis derives a formula for each node describing the Boolean sum of the effects of all paths to the node.

- The analysis may need to characterize the absence rather than the presence of conducting paths.

Each of these differences represents only a slight generalization of the original problem without increasing its complexity. The requirement for directed switches may seem counterintuitive given that MOS transistors are fully bidirectional devices. However, the MOS analyzer accounts for signals of varying strength (representing different driving admittances) by assigning different labels to the two directed edges representing a transistor.

This paper presents some new results on the efficiency of Gaussian elimination for solving systems of equations defined over general series-parallel graphs. Otherwise, it contains little that has not been presented in some form elsewhere. However, material has been drawn from a diversity of disciplines, including switching theory, graph theory and algorithms, linear systems, optimizing compilers, and symbolic manipulation. In many cases, ideas or techniques are applied in ways much different from those conceived by their developers. Few practitioners in the field of computer-aided design are well versed in all of these disciplines. Furthermore, other presentations of methods for solving path problems in graphs have been in terms far more general, abstract, and harder to understand. The main contribution of this paper is to synthesize a collection of ideas into a single framework for solving an important problem.

Sections 2–4 present a mathematical description of the symbolic analysis problem in terms of systems of equations defined over a Boolean algebra. It parallels previous work on the symbolic analysis of contact networks [8], and more general algebraic formulations of path problems [13, 14, 15, 16]. The presentation differs from previous ones in several respects. First, Boolean algebra is selected as the domain of interest. This gives properties that more general presentations cannot assume, including a finite, partially ordered domain, and an idempotent sum operation. In addition, systems of equations are expressed in terms of labeled graphs rather than with matrix algebras. Graphs more clearly capture the sparse structure of the problem to be solved and directly map into data structures for efficient algorithms. Section 5 describes how a system of Boolean equations can be solved by Gaussian elimination. A combination of formal and empirical arguments shows that most networks arising when analyzing MOS circuits can be solved with a linear number of operations using a simple pivot selection rule. This result could prove useful for other circuit analysis programs such as circuit-level simulators. Section 6 describes data structures and algorithms for representing and simplifying Boolean formulas. Section 7 presents an example showing some strengths and weaknesses of the method, and Section 8 summarizes the results.

2 Symbolic Algebra

Symbolic analysis derives formulas that express conditions under which conducting paths are formed in a network of switches. Such formulas are concrete, syntactic representations of Boolean functions. Several formulas may represent a single function. In mathematical terms, the analyzer computes over a domain consisting of the set of all functions mapping a set of p variables (describing the control signals on the switches) to the set $\{0,1\}$, i.e.,

$$\mathcal{B} \;=\; \left\{ f\colon \{0,1\}^p \rightarrow \{0,1\} \right\}$$

In the Boolean algebra of the analysis $\langle \mathcal{B}, \wedge, \vee, \neg, \mathbf{0}, \mathbf{1} \rangle$ the operations $\wedge$, $\vee$, and $\neg$ denote Boolean AND, OR, and NOT, respectively, applied to *functions*. The distinguished elements $\mathbf{0}$ and $\mathbf{1}$ denote the constant functions that yield 0 and 1, respectively, for all argument values. This process of *abstracting* from a primitive domain of Boolean values to one of functions over these values forms the basis of symbolic analysis. Most algebraic properties carry over from the original domain to the abstract one.

The Boolean product of the elements in a set A is denoted $\bigwedge_{a \in A} a$. The product of an empty set is defined to equal $\mathbf{1}$. Similarly, the Boolean sum of the elements in a set A is denoted $\bigvee_{a \in A} a$. The sum of an empty set is defined to equal $\mathbf{0}$.

Elements of $\mathcal{B}$ are partially ordered as $b \leq a$ when $b \vee a = a$, i.e., by their lattice ordering [2, 17]. This partial ordering obeys the following properties, as can easily be derived from the laws of Boolean algebra:

Proposition 1 *For any $b \in A$*

$$b \;\leq\; \bigvee_{a \in A} a$$

Proposition 2 *If $b \geq a$ for all $a \in A$ then*

$$b \;\geq\; \bigvee_{a \in A} a.$$

Proposition 3 *If $a \leq b$, then $\neg b \leq \neg a$.*

Proposition 4 *$b \leq a$ if and only if $b \wedge \neg a = \mathbf{0}$.*

3 Systems of Boolean Equations

Just as a resistor network can be represented by a system of linear equations, so a switch network can be represented by a system of equations in which $\vee$ and $\wedge$ replace addition and multiplication, respectively. This section develops the theory of such systems in terms of labeled graphs and then shows how the switch network analysis problem can be formulated in these terms. In this discussion, (V, E) is a finite, directed graph with vertices V and edges $E \subseteq V \times V$, where $|V| = n$.

Definition 1 *A* vertex labeling *x is an assignment $x(v) \in \mathcal{B}$ to each vertex $v \in V$.*

Two vertex labelings x and y are partially ordered $x \leq y$ when $x(v) \leq y(v)$ for all $v \in V$.

Definition 2 *An* edge labeling *A is an assignment $A(u, v) \in \mathcal{B}$ to each edge $(u, v) \in E$.*

Definition 3 *A* system of Boolean equations *$[A, b]$ consists of an edge labeling A and a vertex labeling b.*

Definition 4 *A vertex labeling x* satisfies *the system $[A, b]$ when*

$$x(v) \;=\; b(v) \vee \bigvee_{(u,v) \in E} [x(u) \wedge A(u, v)] \tag{1}$$

for every $v \in V$.

Observe that unlike the usual formulation of systems of linear equations ($Ax = b$), the unknown x appears on both the left and right hand side of Equation 1. In a matrix notation, this equation would have the form $x = b \vee Ax$. This departure from convention is forced by the fact that the domain has no inverses under $\vee$.

The following property follows directly from the above definition and Proposition 1.

Proposition 5 *If x satisfies a system $[A, b]$, then $x \geq b$.*

In general, many labelings can satisfy a system of equations. For example, any vertex labeling satisfies the system $[I, z]$ defined over a graph with all edges of the form (v, v), where $I(v, v)$ equals $\mathbf{1}$ and $z(v)$ equals $\mathbf{0}$ for all v. We focus our attention on a particular one, considered the "solution" of the system.

Definition 5 *Vertex labeling x is a* solution *of the system $[A, b]$, when*

1. *it satisfies the system, and*
2. *$x \leq y$ for any labeling y satisfying the system.*

By this definition, a system can have at most one solution. In fact, we can show that any system has exactly one solution.

Theorem 1 *Any Boolean system $[A, b]$ has a unique solution x given by the limit of the sequence x^i, where $x^0(v) = \mathbf{0}$ for all v, and $x^i(v)$ is defined for all $i > 0$ and all v as:*

$$x^i(v) \quad = \quad b(v) \vee \bigvee_{(u,v)\in E} [x^{i-1}(u) \wedge A(u, v)]. \tag{2}$$

A proof of this theorem is given in Appendix A, based on the following series of arguments. First, the sequence satisfies $x^i \leq x^{i+1}$ for all i and therefore converges. Second, the value to which the sequence converges satisfies the system. Finally, for any labeling y satisfying the system, $x^i \leq y$ for all i. Therefore, the sequence converges to the minimum labeling satisfying the system.

A system of Boolean equations defines a path problem as follows. Let $P_{u,v}$ denote the set of paths from vertex u to vertex v. A path contributes an "effect" to its destination vertex equal to the Boolean product of labels on its source vertex and edges. The net effect at a vertex is given by the Boolean sum of the effects of all paths having this vertex as destination. The solution of a Boolean system yields the vertex labeling giving the net effect at each vertex as is expressed in the following theorem.

Theorem 2 *If $x(v)$ is defined for all $v \in V$ as*

$$x(v) \quad = \quad \bigvee_{u\in V} \; \bigvee_{p\in P_{u,v}} \left[b(u) \wedge \bigwedge_{(s,t)\in p} A(s, t) \right] \tag{3}$$

then x is the solution of the system $[A, b]$.

A proof of this theorem is shown in Appendix A based on the following series of arguments. First, x satisfies the system $[A, b]$. Second, for any path p to vertex v and any labeling y satisfying the system, the effect of path p is less than or equal to $y(v)$. Therefore, the combined effects of the paths to every vertex must equal the minimum labeling satisfying the system.

Theorem 2 shows how to formulate the classical switch network analysis problem as one of solving a system of Boolean equations. For example, consider a switch network with a designated source terminal s. Let the graph (V, E) have the nodes of the network as vertices, and have an edge (m, n) for each pair of nodes m and n connected by a switch. Let edge labeling $A(m, n)$ equal the Boolean sum of all signals controlling switches connecting nodes m and n. Define the vertex labeling b as $b(s) = \mathbf{1}$, and $b(n) = \mathbf{0}$ otherwise. Then the solution of the system $[A, b]$ is a vertex labeling x where $x(n)$ describes the conditions under which a conducting path will form from s to node n.

4 Dual Systems

Some applications require formulas expressing conditions for the *absence* of any conducting path between terminals of a switch network. Such formulas could be obtained by first solving a system expressing conditions for the presence of paths and then complementing the solution values. These negation operations, however, complicate the task of simplifying the formulas. Alternatively, the formulas can be derived directly by solving a *dual system* in which the roles of $\vee$ and $\wedge$ are interchanged. The mathematical basis for this technique stems from DeMorgan's Laws.

Definition 6 *A* dual system of Boolean equations $[A, b]^D$ *consists of an edge labeling A and a vertex labeling b.*

Definition 7 *Vertex labeling x* satisfies *the dual system $[A, b]^D$ when*

$$x(v) \quad = \quad b(v) \wedge \bigwedge_{(u,v)\in E} [x(u) \vee A(u, v)]$$

for every $v \in V$.

As before, only one vertex labeling is considered to solve a dual system, but this time the maximum one is selected.

Definition 8 *Vertex labeling x is a* solution *of the dual system $[A, b]^D$, when*

1. *it satisfies the system, and*

2. *$x \geq y$ for any labeling y satisfying the system.*

Definition 9 *The* complement *of vertex labeling b, denoted $\overline{b}$, is a vertex labeling with $\overline{b}(v) = \neg b(v)$ for all $v \in V$.*

Definition 10 *The* complement *of edge labeling A, denoted $\overline{A}$, is an edge labeling with $\overline{A}(u, v) = \neg A(u, v)$ for all $(u, v) \in E$.*

Theorem 3 *x is the solution of the system $[A, b]$ if and only if $\overline{x}$ is the solution of the dual system $[\overline{A}, \overline{b}]^D$.*

The proof of this theorem is given in Appendix A. It involves a straightforward application of DeMorgan's Laws.

From this theorem, the role of dual systems in expressing the absence of paths becomes clear.

Corollary 1 *If x is the solution of the dual system $[\overline{A}, \overline{b}]^D$, then*

$$x(v) \;=\; \neg \bigvee_{u \in V} \bigvee_{p \in P_{u,v}} \left[b(u) \wedge \bigwedge_{(s,t) \in p} A(s,t) \right]$$

5 Equation Solution

The equations of the preceding section give an implicit representation of the network function. Symbolic analysis derives explicit formulas by solving these equations. The following presentation describes the solution of normal systems. Dual systems are solved similarly by interchanging the roles of $\wedge$ and $\vee$, as well as those of $\mathbf{0}$ and $\mathbf{1}$.

As Equation 2 suggests, a simple, iterative method can solve a Boolean system $[A, b]$. Although this method lacks efficiency, it aids in understanding more advanced methods. Starting with $x(v) = b(v)$ for each vertex v, the iterative method "propagates" values from a vertex v through an edge (v, u) to the adjacent vertex u, and combines this value into the value already on vertex u. That is, each step involves a computation of the form

$$x(u) \;\leftarrow\; x(u) \,\vee\, [x(v) \wedge A(v,u)].$$

Ultimately, this process converges to a solution. However, an iterative method must either test the solution for convergence, i.e., that $[x(v) \wedge A(v,u)] \leq x(u)$ for all $(v,u) \in E$, or it must perform enough iteration steps to guarantee that for every simple path in the graph, the value on the source vertex has been propagated through the edges of the path to the destination vertex. The first method requires solving the NP-hard problem of testing Boolean formulas for equivalence [18], while the second requires $\Theta(|V| \cdot |E|)$ steps, except for restricted graph structures [19].

5.1 Gaussian Elimination

Gaussian elimination provides the most efficient known method for solving sparse Boolean systems, where Boolean operations replace the real arithmetic used when solving linear systems [13, 20]. Figure 1 shows a sketch of the Gaussian elimination algorithm. The code has two parts: forward elimination and back substitution. Forward elimination successively modifies the system structure, each time eliminating a vertex and all incident edges, and possibly adding edges between some remaining vertices. These structural modifications give Gaussian elimination its performance advantage over iterative methods. Eliminating a vertex v_i (termed the "pivot") involves updating the value of $b(v)$ for each uneliminated neighbor v of v_i. Then for each pair of uneliminated neighbors u and v, the value of $A(u,v)$ is updated. This may require adding a new "fill-in" edge to the graph if (u,v) does not already exist. During back substitution, the vertices are processed in the reverse of their elimination

```
{ Forward elimination }
V_0 ← V; { The uneliminated vertices }
E_0 ← E; { The original edges plus fill-in's}
for i ← 1 to n do
begin
    choose vertex from V_{i-1} and call it v_i; { Select pivot }
    V_i ← V_{i-1} − {v_i};
    E_i ← E_{i-1} ∩ [V_i × V_i];
    for each v ∈ V_i such that (v_i, v) ∈ E_{i-1} do
    begin
        b(v) ← b(v) ∨ [b(v_i) ∧ A(v_i, v)];
        for each u ∈ V_i such that (u, v_i) ∈ E_{i-1} and u ≠ v do
        begin
            if (u, v) ∈ E_i
            then A(u, v) ← A(u, v) ∨ [A(u, v_i) ∧ A(v_i, v)]
            else begin
                { Create fill-in edge. }
                E_i ← E_i ∪ {(u, v)};
                A(u, v) ← A(u, v_i) ∧ A(v_i, v)
            end
        end
    end
end;

{ Back Substitution }
for i ← n step −1 to 1 do
begin
    x(v_i) ← b(v_i);
    for each u ∈ V_i such that (u, v_i) ∈ E_{i-1} do
        x(v_i) ← x(v_i) ∨ [x(u) ∧ A(u, v_i)];
end
```

Figure 1: **Gaussian Elimination Algorithm.** The code differs from the conventional presentation in that it uses graph notation and substitutes Boolean for numerical operations.

ordering. For each vertex v_i, the value of $x(v_i)$ is computed in terms of $b(v_i)$ and the value of $x(u)$ for every neighboring vertex u eliminated after v_i.

The code of Figure 1 also defines some notation to assist the proof of correctness and the performance analysis. To summarize, the vertices of V are labeled $v_1, \ldots, v_n$ in the order they are eliminated. The set $V_i \subseteq V$ is defined as the set of all vertices eliminated after v_i. The set E_i is defined as the set of all edges (both original and fill-in) connecting vertices in V_i. In the actual implementation, little of this information need be stored explicitly. Edges can be represented by adjacency lists with fill-in edges appended to the lists. A stack can keep track of the elimination ordering for use in back substitution. The set of uneliminated vertices can be represented by a priority data structure to implement the desired pivot selection rule.

Theorem 4 *The Gaussian elimination algorithm of Figure 1 solves the system* $[A, b]$.

A proof of this theorem is given in Appendix A. It involves showing that the elimination of a vertex does not change the solution for the remaining vertices. The final system involves only one vertex and is solved trivially. Each back substitution step then adds back a vertex, computing its solution in terms of those for the other vertices.

5.2 Pivot Selection

The efficiency of Gaussian elimination depends largely on the number of uneliminated neighbors each vertex has as it is eliminated. Consider a graph with n vertices. Assume for simplicity that the graph is *structurally symmetric*, that is $(u, v) \in E$ whenever $(v, u) \in E$. This requirement can be met by adding edges to the graph with labels equal to $\mathbf{0}$.[2] With this simplification, an undirected graph describes the structure of the system to be solved. Referring to Figure 1, define the *elimination degree* of vertex v, denoted $d(v)$, as

$$d(v_i) = \Big|\{u \in V_i | (u, v_i) \in E_{i-1}\}\Big|.$$

The number of algebraic operations ($\wedge$ and $\vee$) for elimination is at most

$$2 \sum_{1 \leq i \leq n} [d(v_i) + d(v_i)^2].$$

This formula is highly sensitive to the values of $d(v)$. For example, if the degrees are all bounded by a constant, then only $O(n)$ operations are required. On the other hand, in the worst case $d(v_i)$ can equal $|V_i|$ for all vertices and $O(n^3)$ operations are required.

The vertex elimination degrees depend greatly on the elimination order. Consider, for example a "star" graph, such as the one shown in Figure 2. If the center vertex is eliminated first, fill-in edges are added between every pair of remaining remaining vertices, and the algorithm requires $O(n^3)$ operations. If, on the hand, this vertex is eliminated last, we would have $d(v) \leq 1$ for all v, requiring only $O(n)$ operations.

[2] In fact such edges are often added to simplify the data structures, as it eliminates the need to store explicit pointers in the reverse direction of the edges.

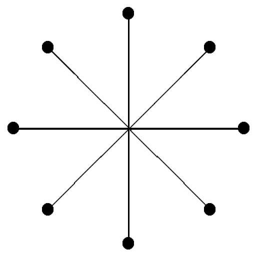

Figure 2: **Star Graph Example.** The elimination complexity of this class of graphs ranges between linear and cubic, depending on the elimination ordering.

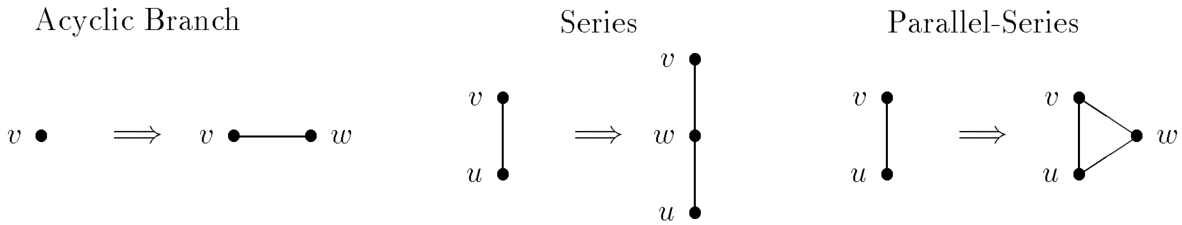

Figure 3: **General Series-Parallel Production Rules.** Any GSP graph can be generated by starting with a single vertex and applying a sequence of these rules.

Much has been written on strategies for choosing elimination orderings, including both empirical [21, 22] and theoretical [23] results. In general, the problem of selecting an *optimal* ordering is NP-complete [18]. However, we would be satisfied with a "good", but not necessarily optimal, ordering, and we can exploit properties of the graphs that arise in MOS circuit analysis.

The following heuristic strategy is often cited [21, 22] for deciding which vertex to select as the next pivot during Gaussian elimination:

Rule M: Choose a vertex that minimizes $d(v)$.

This rule is an example of a "greedy" strategy. That is, it selects a pivot to minimize the computational effort of the next step without regard to future eliminations. For MOS circuits, this strategy works quite well—with only a few exceptions the resulting elimination requires only $O(n)$ operations. A MOS circuit maps into a "channel graph" for symbolic analysis[7]. This graph contains a vertex for each storage (i.e., non-input) node n, and an edge (m, n) for each pair of storage nodes m, n forming the source and drain of a transistor. In general, this graph will contain many components, and each component is analyzed separately.

Most channel graphs describing digital MOS circuits fall into a restricted class that we shall term "general series-parallel" (GSP). This class extends conventional series-parallel graphs[24] to include those containing acyclic branches. GSP graphs can be defined inductively starting with a single vertex as the basis, and applying the production rules illustrated

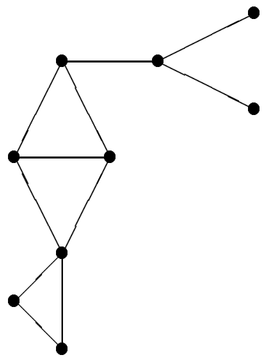

Figure 4: **Channel Graph for Complex nMOS Gate.** Even though the gate has a bridge in its pulldown network, the graph is general series-parallel.

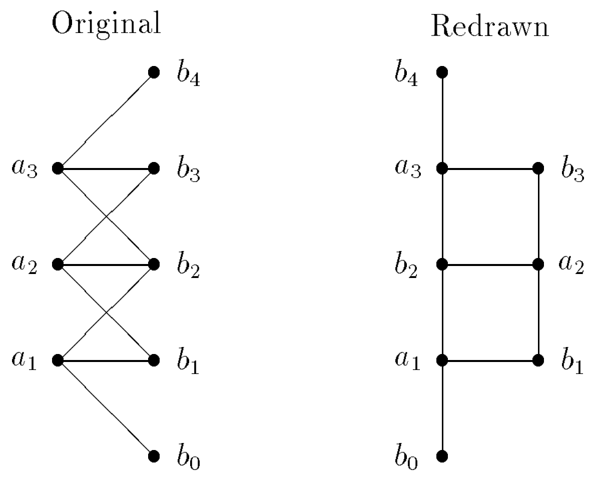

Figure 5: **Channel Graph for Shift Network.** Redrawing it shows the general series-parallel structure more clearly.

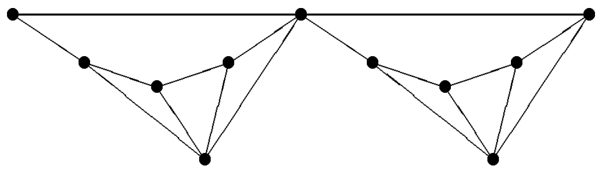

Figure 6: **Channel Graph for Section of Carry Chain.** Although not GSP, no vertex has elimination greater greater than 3.

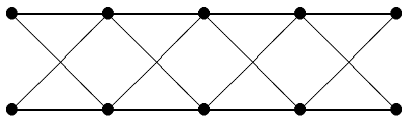

Figure 7: **Channel Graph for Parity Ladder**. Although not GSP, no vertex has elimination degree greater than 3.

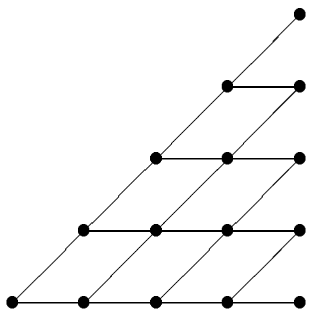

Figure 8: **Channel Graph for Tally Circuit**. This class of graphs has $O(n^2)$ elimination complexity when pivots are selected by Rule M.

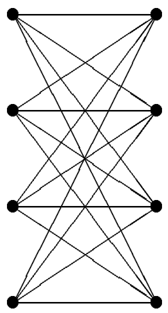

Figure 9: **Channel Graph for Barrel Shifter**. This class of graphs has $O(n^3)$ elimination complexity regardless of pivot sequence.

in Figure 3. That is, given a GSP graph containing a vertex v, a new vertex w and edge (v, w) can be added to give a new GSP graph. Similarly, given a GSP graph containing vertices u and v and edge (u, v), a new vertex w along with edges (v, w) and (u, w) can be added. The edge (u, v) is either deleted for the Series production rule or retained for the Parallel-Series rule. We do not define a pure Parallel rule to avoid creating multigraphs. It can easily be seen that this class of graphs has significance for MOS circuits. Most MOS circuits involve transistors connected series-parallel to implement logic functions and acyclically to implement information transfer.

Figures 4 and 5 show examples of GSP channel graphs. Figure 4 is typical of those that arise when analyzing complex nMOS logic gates with connected pass transistors. Note that the graph contains no edge corresponding to the pullup transistor, since this transistor is connected directly to an input node. The pulldown network contains a "bridge", and hence many would not consider this a series-parallel graph. Most presentations of series-parallel networks assume a "virtual edge" between the top and bottom terminals (to represent the power supply.) Switch graphs need not include such an edge, and hence the channel graph is GSP. CMOS logic gates usually have GSP channel graphs as well. Figure 5 is typical of those that arise when analyzing pass transistor shift networks. This network transfers a set of inputs on the left to the outputs on the right shifted either -1, 0, or 1 positions. As normally drawn, the graph appears quite complex. However, it can be redrawn as shown on the right, making it easier to see the GSP structure. Experience has shown that many seemingly complex circuits have simple channel graphs.

Theorem 5 *A system of equations defined on a graph can be solved by Gaussian elimination such that no vertex has elimination degree greater than 2 if and only if the graph is general series-parallel.*

A proof of this theorem is given in Appendix A. It follows from the observation that the production rules of Figure 3 and the graph transformations caused by eliminating a vertex of degree less than or equal to 2 are inverses of each other. Our application requires only the "if" part of the theorem. The "only if" part is included for intellectual interest. It shows that only GSP graphs satisfy this bound on the maximum elimination degree.

Corollary 2 *Gaussian elimination applied to an n vertex GSP graph with pivots selected by Rule M requires at most $12n$ algebraic operations.*

An n vertex GSP graph must have between $n - 1$ and $2n - 1$ edges. Hence, this result shows that an the analysis of an s switch network requires $O(s)$ operations when the network has a GSP structure.

A survey of 4 books on VLSI [25, 26, 27, 28], plus a direct analysis of many circuit designs has uncovered only a handful of non-GSP channel graphs, as illustrated in Figures 6, 7, 8 and 9. Figure 6 shows the graph for a section of the carry chain circuit from the MIPS-X processor [29]. Even when repeated for a number of stages, systems with this graph have linear elimination complexity, because no vertex has elimination degree greater than 3. The same holds for pass transistor parity ladders based on a well known relay contact network [1], as illustrated in Figure 7. In contrast, path enumeration over this graph gives a result of exponential complexity, while iterative methods have quadratic worst case complexity. Graphs that arise when a circuit is created by repeating a structure in one dimension generally have some constant upper bound on elimination degree and hence linear elimination complexity.

The Tally network of Mead and Conway [25], with graph illustrated in Figure 8 does not yield such favorable results. This network has the lower triangle of a square mesh as its channel graph. For such meshes, informal experiments indicate that selecting pivots by Rule M gives quadratic complexity. For a planar graph such as this, pivot selection by nested dissection can solve an s switch system with $O(s^{3/2})$ operations [23]. In practice, however, only small versions of this circuit are used, or restoring buffers are inserted for performance reasons. Either case yields small channel graphs, and Rule M handles small graphs well. For example, the four input tally circuit shown in the figure has maximum elimination degree 3.

A variety of pass transistor shift networks yield non-GSP channel graphs. A barrel shifter as shown in Figure 9 provides the most extreme case. The channel graph for this circuit is a complete bipartite graph. For solving such a dense system $O(n^3)$ operations are required for any elimination ordering. Given that the number of switches s grows quadratically with the number of nodes, however, the elimination complexity is a respectable $O(s^{3/2})$.

Other shift networks have complexity between those of Figures 5 and 9. For example, the Caltech Mosaic processor [30] has a network that passes the data either straight through, shifted 1 position, or shifted 4 positions, where shifts are circular. When following Rule M, experiments indicate that the elimination degree never exceeds 12 for such a graph, regardless of the shifter width. Although this bound yields a solution of linear complexity, the constant of proportionality becomes noticeably high. Fortunately, shift networks constitute only a small fraction of the total circuitry in a full scale VLSI chip. Even subnetworks with $O(n^3)$ elimination complexity should have little impact on the total result. Furthermore, this polynomial worst case complexity compares favorably to the exponential complexity of other methods.

As an aside, this analysis shows that Gaussian elimination would provide an efficient method for computing node voltages in a linear switch simulator such as RSIM [11]. On the other hand, the results do not carry over as well to circuit simulators such as SPICE [22]. When an implicit integration method is used in a circuit simulator, a conductance is inserted across the terminals of each capacitor. This effectively creates a connection between the gate, source, and drain of every MOS transistor. The resulting graphs can have far more complex structure than channel graphs.

6 Boolean Formula Representation and Manipulation

Up to this point, the presentation has intentionally remained vague as to how Boolean formulas are represented and manipulated. In fact, there are many possible representations offering a wide range of capabilities and limitations. As has been shown, most networks arising in MOS circuit analysis require a linear number of algebraic operations to analyze. Ideally, the Boolean formulas should be represented in such a way that the total size of the formulas for a network preserves this linear growth. A directed acyclic graph representation satisfies this requirement.

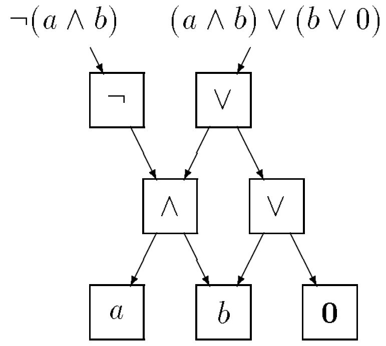

Figure 10: **DAG Representation of Two Formulas**. The leaves denote variables and constants, while the nodes denote Boolean operations. A formula is denoted by a pointer to a node.

6.1 DAG Representation of Formulas

A directed acyclic graph (DAG) [31, 32] resembles a parse tree, with leaves representing either variables or constants, and with internal nodes representing Boolean operations. In a DAG, however, a given subgraph may be shared by several branches, yielding a more compact representation. During the analysis of a switch network, the program constructs a single DAG having multiple roots. A formula is indicated by a pointer to some DAG node, where the formula denoted consists of the node and all of its descendants. Figure 10 shows a DAG representing two formulas.

During Gaussian Elimination, the program can perform an operation symbolically by simply adding a new node to the DAG with branches to the nodes representing the arguments. As observed by Tarjan [20], the DAG produced by this method can grow no larger than the total number of algebraic operations.

6.2 Formula Simplification

As the example of Figure 10 shows, formulas can often be simplified by applying laws of Boolean algebra. Fortunately, the DAG representation forms an ideal data structure for performing these simplifications and for detecting and eliminating common subexpressions [31]. In general, the problem of reducing a formula to its simplest form is NP-hard (proving tautology involves proving that a formula can be simplified to $\mathbf{1}$). However, a large class of simplifications can be expressed in terms of local transformation on the DAG, where no transformation increases the number of nodes. This paper presents only transformations appropriate for the formulas generated by switch network analysis. In particular, it includes only a limited set of negation rules, because negation can only occur within the control formulas for the switches.

The program can readily apply simplifying transformations each time an operation is requested. That is, if some operation *op* (either $\wedge$, $\vee$, or $\neg$) is to be applied to a list of arguments A, the procedure applies transformations to produce a new list A', possibly changing the operation to op' as well. Then the procedure checks a symbol table (e.g., a hash table) to see if a node with this operation and with these arguments already exists. If not it creates a new node and stores a pointer to it in the symbol table. This method avoids ever creating duplicate subexpressions.

6.3 Simplification Rules

This presentation utilizes the following node representation. Associated with each node is a $type \in \{\wedge, \vee, \neg, \mathbf{0}, \mathbf{1}, var\}$. Types $\wedge$, $\vee$, and $\neg$ represent operations. Types $\mathbf{0}$ and $\mathbf{1}$ represent constants, while type var represents a variable. Associated with a node x for which $type(x) \in \{\wedge,\ \vee,\ \text{or}\ \neg\}$ is a list of arguments $Args(x)$. Although the list is not technically a set (because it is ordered and contains duplicates), set notation is used to denote list operations. The set of all nodes is assumed totally ordered, as can be implemented by assigning a unique integer identifier to each node and ordering nodes by their identifiers. This total ordering serves only to permit a canonical listing of all children of a node.

The steps below outline a procedure to apply operation $\vee$ to a list of arguments A, where each argument is specified by a DAG node. The steps to apply $\wedge$ are similar, interchanging the roles of $\wedge$ and $\vee$, as well as those of $\mathbf{0}$ and $\mathbf{1}$. Each step indicates an algebraic identity and an associated set of transformations. The steps are ordered in such a way that the procedure need only sequence through the list once to implement the operation.

1. Associativity: $(a \vee b) \vee c = a \vee (b \vee c)$
   For each $x \in A$ such that $type(x) = \vee$, remove x from A and add $Args(x)$ to A. This transformation guarantees that no node in the DAG will ever have a child of the same type. This transformation may or may not be desirable as is discussed below.

2. Commutativity: $a \vee b = b \vee a$
   Sort the elements of A. This transformation guarantees that all nodes in the DAG will list their children in the same order.

3. Idempotency: $a \vee a = a$
   Remove any duplicate entries from A. Since the elements of A are sorted, duplicates must appear consecutively.

4. Identity: $a \vee \mathbf{0} = a$
   Remove from A any element x such that $type(x) = \mathbf{0}$.

5. Annihilator: $a \vee \mathbf{1} = \mathbf{1}$
   If A contains any element x such that $type(x) = \mathbf{1}$, then return x as the result of the evaluation.

6. Excluded Middle: $a \vee \neg a = \mathbf{1}$
   If A contains elements x and y such that $type(x) = \neg$ and $y \in Args(x)$, then return a node of type $\mathbf{1}$ as the result of the evaluation.

7. Redundancy: $b \leq a \Rightarrow b \vee a = a$.
   For each $x \in A$, label x with 1 and every $y \in A - \{x\}$ with 0. If a search with these labels leads to a contradiction, then remove x from A. The search procedure is described in detail below.

8. Degenerate Cases: $\bigvee_{a \in \{b\}} a = b$, $\bigvee_{a \in \emptyset} a = \mathbf{0}$

   (a) If A contains only a single element x, then return x as the result of the evaluation.

   (b) If A is empty, then return a node of type $\mathbf{0}$ as the result of the evaluation.

9. Common Subexpressions
   Look in the symbol table for an entry with key $\langle \vee, A \rangle$.

   (a) If an entry is found, then return it as the result of the evaluation.

   (b) If no entry is found, then create a new node x with $type(x) = \vee$ and $Args(x) = A$. Add an entry for x to the symbol table with key $\langle \vee, A \rangle$. Return x.

   By this method, we guarantee that duplicate nodes are never created.

6.4 Discussion of Transformations

Observe that this list of transformations does not include any for the two distributive laws:

$$\begin{array}{rcl} (a \vee b) \wedge c & = & (a \vee c) \wedge (b \vee c) \\ (a \wedge b) \vee c & = & (a \wedge c) \vee (b \wedge c). \end{array}$$

If we were to apply transformations that distribute one operation over the other, the size of the DAG would be increased. The DAG could even grow to exponential size, if for example, distributivity were applied to transform the formula into sum-of-products form. On the other hand, we could attempt to recognize opportunities to factor expressions. However, expressions such as $(a \wedge b) \vee (b \wedge c) \vee (a \wedge c)$ can be factored in more than one way, giving different degrees of simplification. Thus, the manipulator ignores the distributive laws altogether.

The associativity transformation (step 1) does not increase the number of nodes in the DAG, and hence incurs no added complexity under a *node cost* model, where the complexity of a DAG is expressed as the total number of nodes. However, it can create nodes with more children than the number of arguments in the original list. For example, the evaluation sequence

$$\begin{array}{rcl} x & \leftarrow & a \wedge b \\ y & \leftarrow & x \wedge c \\ z & \leftarrow & y \wedge d \end{array}$$

would create 3 nodes having 2, 3, and 4 children, respectively. The *binary cost* model expresses the DAG complexity as the sum of its node costs, where a node with n children has cost $n - 1$. This measure corresponds to the number of binary operations required to evaluate the resulting expression. Under this cost model, the associativity transformation can increase the complexity. The above example would yield a DAG of binary cost 6, whereas omitting the transformation would yield a DAG of cost 3.

The associativity transformation also interacts with the redundancy transformation (step 7), described in detail later. This transformation requires a search of the DAG for each element of the list A, incurring a significant computational effort. The associativity transformation can expand this list and, consequently, the number of searches.

Of course, omitting this transformation causes the manipulator to overlook some useful simplifications. It will fail to recognize the equivalence of some expressions. Furthermore, it will fail to eliminate some redundancies that would otherwise be found. For example, consider the DAG for the expression $a \vee [c \vee (a \wedge b)]$. The associativity transformation would create a list of arguments a, c, and $a \wedge b$. The redundancy transformation would then eliminate the third argument, yielding a simplified expression $a \vee c$. On the other hand, no simplification would occur if the associativity transformation were omitted, because neither the expression a nor $c \vee (a \wedge b)$ is redundant with respect to the other.

The choice of whether or not to apply the associativity transformation depends on the nature of the formulas generated and the appropriate complexity measure for the DAG. Our experiments with a symbolic MOS circuit analyzer clearly indicate that, under the binary cost model, the associativity transformation increases the DAG complexity by a factor of at least 2 for almost all circuits. Furthermore, depending on the particular circuit, it can greatly increase the amount of CPU time spent searching for redundancies. However, omitting the transformation yields formulas with a noticeable number of redundant terms. Hence the desirability of the transformation depends on the intended application of the symbolic analyzer output.

6.5 Redundancy Testing

The redundancy test mentioned in step 7 has proved important when simplifying the formulas arising during MOS circuit analysis. Due to the way the program analyzes a MOS network by solving a series of switch networks, it would otherwise construct complex formulas containing many redundancies. Methods for discovering redundancy range widely in their completeness and efficiency. On one extreme, a method that reliably detects any case where 2 formulas are ordered $x \leq y$ can solve the NP-hard equivalence problem. That is, two formulas are equivalent if and only if both $x \leq y$ and $y \leq x$. On the other extreme, simple graph transformations can apply the simple absorption rule $a \vee (a \wedge b) = a$. Simple approaches, however, miss many opportunities for simplification.

The method discussed below strikes a compromise between efficiency and completeness. It applies a search technique that attempts to prove that an argument is redundant but applies tight controls to avoid combinatorial complexity.

Proposition 4 states that two Boolean formulas are ordered $x \leq y$ if and only if no assignment of 1's and 0's to the variables can cause x to evaluate to 1, while y evaluates to 0. The redundancy test attempts to prove this property by contradiction, in a manner reminiscent of an automatic theorem prover based on the Resolution Principle [33]. That is, it assigns value 1 to x, 0 to y, and determines the logical consequences of these assignments. If it reaches a contradiction, then the formulas are ordered, otherwise they are assumed unordered. The manipulator applies this test to each argument x in the list A in an attempt to drop the argument from the list. That is, for an $\vee$ (respectively, $\wedge$) operation, the

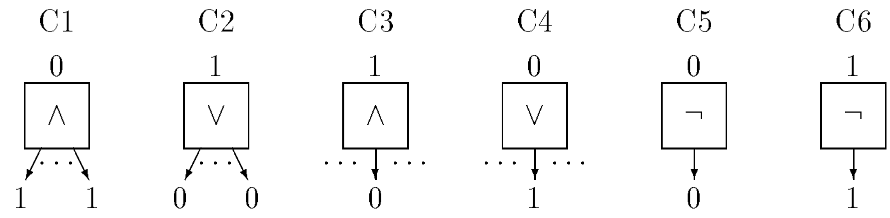

Figure 11: **Contradiction Rules for Redundancy Test.** The search terminates successfully when one of these labelings arises.

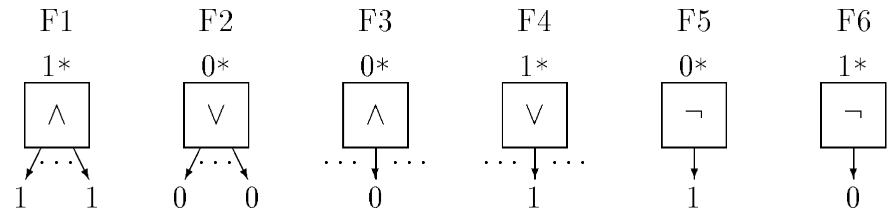

Figure 12: **Forward Inference Rules for Redundancy Test.** These rules change the value of a node based on those of its children. Asterisks mark the values changed from X to 0 or 1.

test searches for a contradiction with x assigned value 1 (resp., 0) and all other arguments assigned 0 (resp., 1).

The redundancy test requires augmenting the DAG data structure with an additional *value* field for each node, set to either 0, 1, or X (indicating an unknown value). Initially, the nodes in argument list A are set to 0 or 1 specifying the proof goal, while the other nodes are set to X. Each node also has a list of pointers to its parents in the DAG. The program searches for a contradiction in a manner similar to the implication step of the D-algorithm

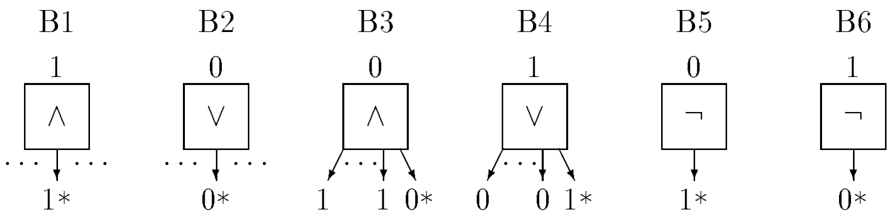

Figure 13: **Backward Inference Rules for Redundancy Test.** These rules change the value of a child based on those of its parent and siblings. Asterisks mark the values changed from X to 0 or 1.

used in test generation [34]. A queue, initialized with the nodes in argument list A plus their parents, holds nodes that are candidates for further inferences. Boolean values are propagated through the DAG by repeatedly removing a vertex from the queue and applying inference rules that may change the node value or that of one or more child. If a node value changes the program adds either its children or parents to the queue as candidates for further inferences. This process continues until either a contradictory labeling is encountered (success), or the queue becomes empty (failure).

Figure 11 illustrates the set of contradictory labelings that cause the search to terminate successfully. In this figure, the label above the node indicates the value associated with the node, while the labels below indicate the values associated with the children. Note that rules C1 and C2 require all children to have a particular value, while rules C3 and C4 require only a single child with the specified value.

Figures 12 and 13 present the set of inference rules by which Boolean values are propagated through the DAG. For each rule, an asterisk indicates the value changed by the rule from X to 0 or 1. Figure 12 illustrates the set of *forward* inference rules, i.e., those that cause the value of a node to change based on the values of its children. For example, rule F1 indicates that if all children of an $\wedge$ node have value 1, then it too must have value 1. Rule F3 indicates that if an $\wedge$ node has any child with value 0, then it must have value 0. Successful application of a forward inference rule to a node causes queueing of any parent not already in the queue.

Figure 13 illustrates the set of *backward* inference rules, i.e., those that cause the value of one or more child to change based on the value of the node and possibly the values of the other children. For example, rule B1 indicates that if an $\wedge$ node has value 1, then every child must have value 1. Rule B3 indicates that if an $\wedge$ node has value 0 and all but one child have value 1, the remaining child must have value 0. Successful application of a backward inference rule to a node causes queuing of any child whose value changes and is not already on the queue. Any other parent of a child whose value changes is also queued, unless it is already in the queue.

Figure 14 shows an example of how the redundancy test can prove that two formulas x and y are ordered $x \leq y$. This example was adapted from one that can actually occur during the symbolic analysis of a MOS circuit, demonstrating the need for a sophisticated redundancy test. The figure does not show the descendants of the nodes labeled with operation *op*, as they are not required to prove redundancy. This example arises when a back substitution step of Gaussian elimination requires an evaluation of the form $y \leftarrow x \vee y$. As a result of the successful redundancy test, formula y remains unchanged, yielding a significant simplification. For purpose of discussion, each node is labeled with an identifying letter to its left. The initial values assigned to the nodes are shown to their right. The queue initially contains nodes `a`, `d`, and `c`. The search proceeds by the series of steps shown in Table 1. It terminates once a conflicting labeling is found at node `e`.

The following analysis of the search algorithm shows that it has linear complexity, as measured in the total number of branches in the DAG. The search only queues a node when the value on the node, one of its parents, or one of its children changes from X to 0 or 1. Each branch in the DAG can cause the nodes on either end to be queued at most once. Therefore, the total number of queuing operations cannot exceed twice the number of branches in the

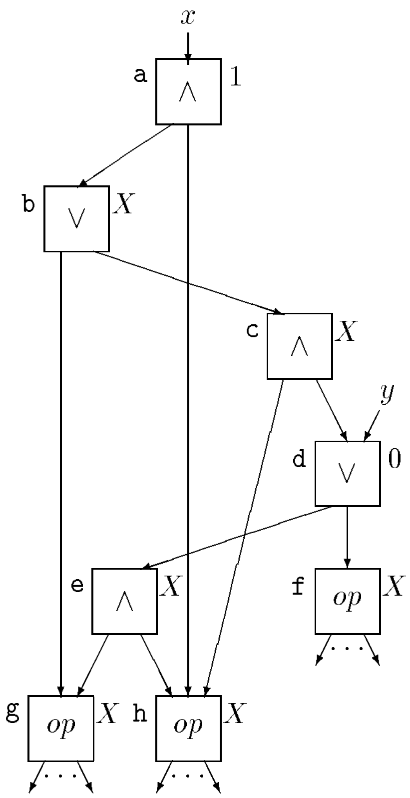

Figure 14: **Redundancy Search Example.** The search proves that formula x is redundant with respect to formula y. Nodes are labeled by their values at the start of search and by letters for discussion in the text.

Step	Node	Rule	Changed Nodes	New Value	Queued Nodes
1.	`a`	B1	`b`, `h`	1	`b`, `h`
2.	`d`	B2	`e`, `f`	0	`e`, `f`
3.	`c`	F3	`c`	0	
4.	`b`	B4	`g`	1	`g`
5.	`h`	none			
6.	`e`	C1			

Table 1: **Inference Sequence for Redundancy Search Example**. The search finds a contradiction at node `e`.

DAG. The set of inference rules can be applied to a node in constant time, if counts are maintained for each node specifying the number of children with value 0, 1, and X. Hence the algorithm has time complexity linear in the DAG size. Furthermore, the constant of proportionality is quite small.

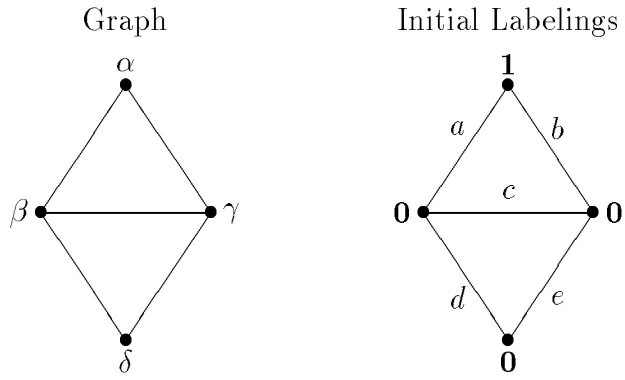

Figure 15: **Example System of Equations.**

However, since the search must be initiated once for each argument every time a Boolean operation is performed, the total time spent searching for redundancies can become quite large. Our implementation controls the time spent searching in 2 ways. First, the search need only consider nodes that are descendants of the nodes in the argument list A. The program avoids visiting extraneous nodes by keeping the value fields of the nodes initialized to a special value indicating "unreachable." Before a search begins, the program traces all descendants of the nodes in A and sets their values to X. The search then only visits nodes with values X, 0, or 1, and upon termination resets all nodes to "unreachable". This constraint will not reduce the success rate of the search. Second, the search proceeds in breadth-first order (by using a first-in, first-out queueing discipline), and can be constrained to give up once it reaches a specified depth. This constraint reduces the success rate of the search, but eliminates cases requiring extensive search having little chance of success. With appropriate constraint, experience indicates that this approach to redundancy testing yields significant simplifications for a reasonable computational effort.

It must be emphasized, however, that the redundancy test is not complete. For example, it will recognize that $a \wedge (b \vee d)$ is redundant with respect to $(a \wedge b) \vee (a \wedge d)$, but not *vice-versa*, even though the two expressions are equivalent. The algorithm could be extended to one that detects all redundant cases by adding combinatorial search and backtracking. However, this could greatly increase the computational effort, especially considering that in most cases the redundancy test will fail.

7 Symbolic Analysis Example

As with many algorithms designed for computer implementation, this analysis method is very tedious to execute by hand. For systems of significant size, the DAG becomes far to large to draw. Small systems, on the other hand, lend well to exhaustive path analysis. Hence it is hard to demonstrate the advantages of our method with an example. With these limitations in mind, several useful insights can be gained from a simple example.

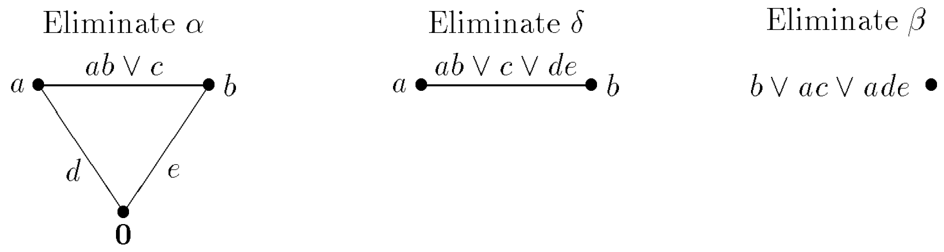

Figure 16: **Elimination Steps for Example System.**

Figure 15 shows the graph corresponding to a bridge network with source terminal α. In this example, the edge labeling is symmetric: $A(u,v) = A(v,u)$, and hence the system structure can be represented by an undirected graph. The steps of Gaussian elimination preserve this symmetry, and a straightforward modification of the elimination code reduces the number of operations by almost a factor of 2. Figure 16 shows the sequence of labelings produced by the successive elimination steps. For readability, the Boolean formulas were simplified by hand and are shown without $\wedge$ symbols. The back substitution steps yield:

$$\begin{aligned} x(\gamma) &= b \vee ac \vee ade \\ x(\beta) &= a \vee bc \vee bde \\ x(\delta) &= ad \vee bcd \vee be \vee ace \\ x(\alpha) &= \mathbf{1} \end{aligned}$$

Figure 17 shows the complete DAG produced in analyzing this system. For readability, the branches to nodes representing variables are indicated by the variable names. This DAG looks very complex, and it is difficult to verify that it correctly characterizes the network. Observe, however, that this representation of the formulas involves only 10 Boolean operations. The formulas derived by hand simplification appear much more readable, but they require a total of 11 Boolean operations. Furthermore, under the binary cost model, (a better measure of the evaluation cost for a set of formulas), the DAG has cost 11, whereas a straightforward implementation of the hand-derived formulas has cost 19. Only with considerable effort can the hand-derived formulas be transformed into ones involving a total of 11 binary operations. This example shows that our method produces results that are very compact although difficult for humans to read. Compactness counts more for results that are used by other computer programs.

8 Conclusion

This paper has shown that a careful choice of algorithm and data structures yields a far more efficient solution than does a naive approach. Furthermore, by casting the problem in terms of systems of equations, the wealth of knowledge that has accumulated about solving linear systems could be applied.

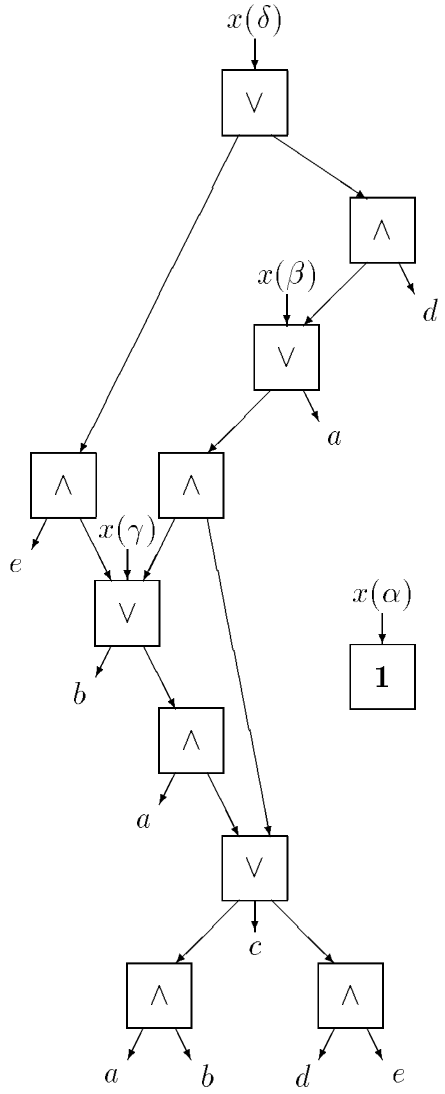

Figure 17: **DAG Produced in Analyzing Example Network.** Although appearing more complex than the formulas derived by hand, the DAG representation is actually more compact.

The symbolic analysis technique described in this paper has a wide range of applications beyond switch networks. Direct methods such as Gaussian elimination are examples of *oblivious* algorithms. That is, the control sequence depends only on the graph structure of the system to be solved and not on the data values.[3] In contrast, iterative methods perform data-dependent branches when testing for convergence or when deciding which vertex to update next. Any oblivious algorithm can be executed symbolically to yield some explicit

[3]Although data-dependent pivoting may be required when solving linear systems for numerical reasons.

representation (e.g., a DAG) of the output generated by the program for all possible input data. Such a preprocessing step can yield a significant performance advantage in applications that must evaluate many systems sharing a common structure but differing in data values. Furthermore, the representation generated by symbolic analysis can be executed by hardware that supports only the operations of the underlying algebra, rather than general purpose computation. Hardware that supports only restricted domains such as Boolean operations can achieve very high performance at a reasonable cost. Problems that can be solved by Gaussian elimination and hence are amenable to symbolic analysis include: linear systems, shortest path calculations, bottleneck flow path calculations, and conversion of finite automata to regular expressions [16].

Solving path problems by Gaussian elimination becomes especially attractive as computers with parallel processing capabilities become available. The "greedy" pivot selection rule presented here gives very good results in terms of the total size of the formulas. If the results are to be executed on machines that support high degrees of parallelism, however, the potential concurrency allowed by the formulas should be maximized as well. As the DAG of Figure 17 illustrates, greedy pivot selection tends to yield "long, skinny" formulas without much potential for concurrent evaluation. On the other hand, pivot selection based on nested dissection [23, 35] yields "short, fat" formulas, many terms of which could be evaluated simultaneously. In particular, the family of GSP graphs satisfies a *2-separator* theorem, meaning that it is always possible to find 2 vertices whose removal would split the graph into two GSP graphs of roughly equal size. For such graphs, nested dissection yields formulas with $O(n)$ operations, although the constant of proportionality would be somewhat higher. However, the formulas also have maximum depth $O(\log\ n)$, giving sublinear evaluation times if sufficient resources are available. Various other graph classes lead to formulas with sublinear maximum depth. In contrast, no iterative method can give sublinear performance for the graph structures of interest regardless of the processing hardware.

Symbolic analysis, as presented here, simply transforms one description of a Boolean computation into another, that is from a switch network to a set of formulas. Some applications, such as proving two networks equivalent or that a network implements a given function, require stronger results. These problems are NP-hard [18], and many believe efficient algorithms for such tasks do not exist. However, several approaches yield practical results in many instances. One approach uses a different representation of Boolean functions that makes equivalence testing more straightforward. The author [36] has devised a representation based on a different type of directed acyclic graph that is canonical, i.e., a given function has a unique representation. Equivalence testing then becomes a simple matter of testing whether two graphs match exactly. Furthermore, many of the functions encountered in logic design applications are represented by reasonably small graphs. Symbolic analysis could also be performed using these graphs as the underlying data structure for representing Boolean functions, yielding a canonical description of the network function.

A Proofs of Theorems

A.1 Systems of Boolean Equations

Theorem 1 *Any Boolean system $[A, b]$ has a unique solution x given by the limit to the sequence x^i, where $x^0(v) = \mathbf{0}$ for all v, and $x^i(v)$ is defined for all $i > 0$ and all v as:*

$$x^i(v) \;=\; b(v) \vee \bigvee_{(u,v)\in E} [x^{i-1}(u) \wedge A(u,v)]. \tag{4}$$

Proof: First, we will show by induction on i that $x^i \leq x^{i+1}$. The basis clearly holds, because $x^0(v) = \mathbf{0} \leq x^1(v)$. Assuming by induction that $x^i(u) = x^i(u) \vee x^{i-1}(u)$ for any vertex u, expanding Equation 4 for x^{i+1} and separating terms gives

$$\begin{aligned} x^{i+1}(v) &= b(v) \vee \bigvee_{(u,v)\in E} \Big([x^i(u) \vee x^{i-1}(u)] \wedge A(u,v)\Big) \\ &= \left[b(v) \vee \bigvee_{(u,v)\in E} [x^i(u) \wedge A(u,v)]\right] \;\vee\; \left[b(v) \vee \bigvee_{(u,v)\in E} [x^{i-1}(u) \wedge A(u,v)]\right] \\ &= x^{i+1}(v) \vee x^i(v). \end{aligned}$$

Thus, the sequence is nondecreasing. Since the domain $\mathcal{B}$ is finite, there must be some value k such that $x^k = x^{k+1}$.[4] From Equation 4 and by induction on i, it is easy to see that $x^k = x^i$ for all $i \geq k$, and consequently the sequence converges to a unique value. Furthermore, this vertex labeling clearly satisfies the system $[A, b]$.

Now suppose that some other labeling y satisfies the system $[A, b]$. We will show by induction on i that $y \geq x^i$ for all i. Clearly $y(v) \geq \mathbf{0} = x^0(v)$, and hence the basis condition holds. Now suppose that $x^{i-1}(u) \vee y(u) = y(u)$ for every vertex u. Then

$$\begin{aligned} y(v) &= b(v) \vee \bigvee_{(u,v)\in E} \Big([x^{i-1}(u) \vee y(u)] \wedge A(u,v)\Big) \\ &= \left[b(v) \vee \bigvee_{(u,v)\in E} [x^{i-1}(u) \wedge A(u,v)]\right] \vee \left[b(v) \vee \bigvee_{(u,v)\in E} [y(u) \wedge A(u,v)]\right] \\ &= x^i(v) \vee y(v) \end{aligned}$$

for every vertex v, indicating that $y \geq x^i$. Hence any labeling that satisfies $[A, b]$ must be greater than or equal to the limit of the sequence.

□

Theorem 2 *If $x(v)$ is defined for all $v \in V$ as*

$$x(v) \;=\; \bigvee_{u\in V} \; \bigvee_{p\in P_{u,v}} \left[b(u) \wedge \bigwedge_{(s,t)\in p} A(s,t)\right] \tag{5}$$

[4]In fact, k must be less than n.

then x is the solution of the system $[A, b]$.
Proof: First, we will show that x satisfies the system. Any path $p \in P_{u,v}$ must consist of either a single vertex (only in the case where $u = v$), or a path $p' \in P_{u,w}$ followed by the edge (w, v). Hence, Equation 5 can be expanded as

$$\begin{aligned} x(v) &= b(v) \vee \bigvee_{(w,v)\in E} \left(\bigvee_{u\in V} \bigvee_{p'\in P_{u,w}} \left[b(u) \wedge \bigwedge_{(s,t)\in p'} A(s,t) \right] \wedge A(w,v) \right) \\ &= b(v) \vee \bigvee_{(w,v)\in E} [x(w) \wedge A(w,v)]. \end{aligned}$$

Replacing w by u in the final summation gives

$$x(v) \quad = \quad b(v) \vee \bigvee_{(u,v)\in E} [x(u) \wedge A(u,v)]$$

showing that x satisfies $[A, b]$.

Second, for any path p to vertex v containing i edges, and any vertex labeling y satisfying $[A, b]$, we will show by induction on i that the effect of this path is less than or equal to $y(v)$. That is, if $p \in P_{u,v}$, then

$$b(u) \wedge \bigwedge_{(s,t)\in p} A(s,t) \quad \leq \quad y(v).$$

The basis case clearly holds, because a path to v containing 0 edges must originate at v, and $b(v) \leq y(v)$ by Proposition 5. Now assume for $i > 0$ that the induction assumption holds for any path with $i - 1$ edges. A path from u to v containing i edges must consist of a path p' from u to some vertex w containing $i - 1$ edges followed by the edge (w, v). Applying the induction assumption to p', rearranging terms, and applying Proposition 1, gives

$$\begin{aligned} b(u) \wedge \bigwedge_{(s,t)\in p} A(s,t) &= \left[b(u) \wedge \bigwedge_{(s,t)\in p'} A(s,t) \right] \wedge A(w,v) \\ &\leq y(w) \wedge A(w,v) \\ &\leq b(v) \vee \bigvee_{(u,v)\in E} [y(u) \wedge A(u,v)] \\ &= y(v). \end{aligned}$$

Since this result holds for all paths to v, Proposition 2 shows that it must hold for their sum. Hence x is the minimum vertex labeling satisfying $[A, b]$.

□

Theorem 3 *x is the solution of the system $[A, b]$ if and only if $\overline{x}$ is the solution of the dual system $[\overline{A}, \overline{b}]^D$.*
Proof: DeMorgan's Laws can be generalized to the following rule for complementing a Boolean formula:

Complement every variable, replace every $\wedge$ by $\vee$, every $\vee$ by $\wedge$, every $\mathbf{0}$ by $\mathbf{1}$, and every $\mathbf{1}$ by $\mathbf{0}$.

From this rule, we can see that the conditions for a labeling y to satisfy system $[A, b]$ are identical to those for $\overline{y}$ to satisfy the the dual system $[\overline{A}, \overline{b}]^D$. Furthermore, if $x \leq y$ for all y in some set Y, then $\overline{x} \geq \overline{y}$ for all $y \in Y$, and *vice-versa*. Therefore, the minimum labeling satisfying $[A, b]$ must equal the complement of the maximum labeling satisfying $[\overline{A}, \overline{b}]^D$, and *vice-versa*.

□

A.2 Gaussian Elimination

Theorem 4 *The Gaussian elimination algorithm of Figure 1 solves the system* $[A, b]$.
Proof: The key idea of the proof is to show that each time a vertex v_i is eliminated, a modified system $[A_i, b_i]$ is created over (V_i, E_i) such that the solution of this system equals the solution of the original for all $v \in V_i$. Assume for simplicity that there are no edges of the form (v, v) in E. It can easily be shown that removing such edges will not alter the system solution. Furthermore, the elimination code does not add any such edges as fill-in. We also adopt the convention that $A(u, v) = \mathbf{0}$ whenever $(u, v) \notin E$, and similarly that $A_i(u, v) = \mathbf{0}$ whenever $(u, v) \notin E_i$. Under these two conventions, Equation 1 can be written in two ways:

$$\begin{aligned} x(v) &= b(v) \vee \bigvee_{u \in V} [x(u) \wedge A(u, v)] \\ &= b(v) \vee \bigvee_{u \in V - \{v\}} [x(u) \wedge A(u, v)]. \end{aligned} \tag{6}$$

Define the system $[A_0, b_0]$ over the graph (V_0, E_0) as $A_0 = A$ and $b_0 = b$. For $n \geq i \geq 1$ define the system $[A_i, b_i]$ over the graph (V_i, E_i) as

$$A_i(u, v) = A_{i-1}(u, v) \vee [A_{i-1}(u, v_i) \wedge A_{i-1}(v_i, v)] \tag{7}$$

and

$$b_i(v) = b_{i-1}(v) \vee [b_{i-1}(v_i) \wedge A_{i-1}(v_i, v)]. \tag{8}$$

Observe that the definition of A_i preserves the property that $A_i(u, v) = \mathbf{0}$ when $(u, v) \notin E_i$, because E_i is guaranteed to have an edge (u, v) if both (u, v_i) and (v_i, v) are in E_{i-1}.

For $1 \leq i \leq n$, define the labeling x_{i-1} over the vertices of V_{i-1} as:

$$x_{i-1}(v) = \begin{cases} b_{i-1}(v_i) \vee \bigvee_{u \in V_i} [x_i(u) \wedge A_{i-1}(u, v_i)], & v = v_i \\ x_i(v), & v \in V_i \end{cases} \tag{9}$$

Note that $V_n = \emptyset$, and hence x_{n-1} is well defined. It can also be seen that the labeling x produced by the code of Figure 1 equals the labeling x_0 defined by Equation 9 for $i = 1$.

We will show by induction on i (starting from $n-1$ and working downward) that x_i is the solution of the system $[A_i, b_i]$. Clearly x_{n-1} is the solution of $[A_{n-1}, b_{n-1}]$, because Equations

6 and 9 both reduce to $x_{n-1}(v_n) = b_{n-1}(v_n)$. Assume that x_i is the solution of the system $[A_i, b_i]$. We must show that x_{i-1} satisfies the system $[A_{i-1}, b_{i-1}]$, and that $x_{i-1} \leq y$ for any other labeling y satisfying this system.

For $v \in V_i$, given that x_i satisfies $[A_i, b_i]$, $x_i(v)$ can be expanded using Equation 6 as

$$x_i(v) \quad = \quad b_i(v) \vee \bigvee_{u \in V_i} [x_i(u) \wedge A_i(u, v)].$$

The definitions for $b_i(v)$ and $A_i(u, v)$ can then be substituted to give

$$\begin{aligned} x_i(v) \quad = \quad & b_{i-1}(v) \vee [b_{i-1}(v_i) \wedge A_{i-1}(v_i, v)] \ \vee \\ & \bigvee_{u \in V_i} \Big[x_i(u) \wedge \Big(A_{i-1}(u, v) \vee [A_{i-1}(u, v_i) \wedge A_{i-1}(v_i, v)]\Big)\Big]. \end{aligned}$$

Rearranging terms gives

$$\begin{aligned} x_i(v) \quad = \quad & b_{i-1}(v) \vee \bigvee_{u \in V_i} [x_i(u) \wedge A_{i-1}(u, v)] \quad \vee \\ & \left[\left(b_{i-1}(v_i) \vee \bigvee_{u \in V_i} [x_i(u) \wedge A_{i-1}(u, v_i)]\right) \ \wedge \ A_{i-1}(v_i, v)\right] \end{aligned}$$

Substituting the definition for x_{i-1} gives

$$\begin{aligned} x_{i-1}(v) \quad = \quad x_i(v) \quad & = \quad b_{i-1}(v) \vee \bigvee_{u \in V_i} [x_{i-1}(u) \wedge A_{i-1}(u, v)] \quad \vee \quad [x_{i-1}(v_i) \ \wedge \ A_{i-1}(v_i, v)] \\ & = \quad b_{i-1}(v) \ \vee \bigvee_{u \in V_{i-1}} [x_{i-1}(u) \wedge A_{i-1}(u, v)] \end{aligned}$$

For $v = v_i$, we can substitute $x_{i-1}(u)$ for $x_i(u)$ in Equation 9 giving

$$x_{i-1}(v_i) \quad = \quad b_{i-1}(v_i) \vee \bigvee_{u \in V_i} [x_{i-1}(u) \wedge A_{i-1}(u, v_i)].$$

Combining these two cases we see that x_{i-1} satisfies $[A_{i-1}, b_{i-1}]$.

Now suppose that a vertex labeling y defined over V_{i-1} satisfies the system $[A_{i-1}, b_{i-1}]$. Define the labeling y' over V_i as $y'(v) = y(v)$ for all $v \in V_i$. We will first show that y' satisfies the system $[A_i, b_i]$, and therefore by the induction assumption that $x_{i-1}(v) = x_i(v) \leq y'(v) = y(v)$ for all $v \in V_i$. Then we will show that $x_{i-1}(v_i) \leq y(v_i)$, thereby completing the proof that $x_{i-1} \leq y$. For $v \neq v_i$, expanding $y(v)$ using Equation 6 and substituting the definition of y' gives

$$y'(v) \quad = \quad y(v) \quad = \quad b_{i-1}(v) \vee [y(v_i) \wedge A_{i-1}(v_i, v)] \ \vee \bigvee_{u \in V_i} [y'(u) \wedge A_{i-1}(u, v)].$$

Expanding $y(v_i)$ using Equation 6 gives

$$\begin{aligned} y'(v) \quad = \quad & b_{i-1}(v) \vee [b_{i-1}(v_i) \wedge A_{i-1}(v_i, v)] \vee \\ & \bigvee_{u \in V_i} [y'(u) \wedge A_{i-1}(u, v_i) \wedge A_{i-1}(v_i, v)] \ \ \vee \ \bigvee_{u \in V_i} [y'(u) \wedge A_{i-1}(u, v)]. \end{aligned}$$

Combining terms and substituting the definitions of b_i and A_i gives

$$y'(v) \;=\; b_i(v) \vee \bigvee_{u\in V_i} [y'(u) \wedge A_i(u,v)].$$

Therefore y' satisfies the system $[A_i, b_i]$. For $v = v_i$, we can assume that $x_{i-1}(u) \vee y(u) = y(u)$ whenever $A_{i-1}(u, v_i) \neq \mathbf{0}$. Hence, $y(v_i)$ can be expanded by Equation 6 as

$$\begin{aligned}
y(v_i) \;&=\; b_{i-1}(v_i) \vee \bigvee_{u\in V_{i-1}} \Big([x_{i-1}(u) \vee y(u)] \wedge A_{i-1}(u,v_i)\Big) \\
&=\; \left[b_{i-1}(v_i) \vee \bigvee_{u\in V_{i-1}} [x_{i-1}(u) \wedge A_{i-1}(u,v_i)]\right] \vee \\
&\qquad \left[b_{i-1}(v_i) \vee \bigvee_{u\in V_{i-1}} [y(u) \wedge A_{i-1}(u,v_i)]\right] \\
&=\; x_{i-1}(v_i) \vee y(v_i),
\end{aligned}$$

and hence $x_{i-1}(v_i) \leq y(v_i)$. Thus we have shown that $x_{i-1} \leq y$ for any y satisfying the system $[A_{i-1}, b_{i-1}]$, completing the inductive proof that x_i is the solution of the system $[A_i, b_i]$. We have therefore proved the correctness of the algorithm, because the systems $[A, b]$ and $[A_0, b_0]$ are identical, and the labeling x returned by the algorithm equals x_0.

□

Theorem 5 *A system of equations defined on a graph can be solved by Gaussian elimination such that no vertex has elimination degree greater than 2 if and only if the graph is general series-parallel.*

Proof: Assume the graph (V, E) is constructed by a sequence of productions obeying the rules of Figure 3. Suppose the final step involves adding vertex w and one or more edges to the graph (V', E') and possibly deleting an edge. Then vertex w has degree less than or equal to 2 in (V, E). Furthermore, if w is selected as a pivot in Gaussian elimination, the resulting elimination operations will yield the graph (V', E'). This graph is also GSP, and hence the process can be continued until all vertices are eliminated.

Conversely, suppose Gaussian elimination can be performed for a system defined on the graph (V, E) such that no vertex has elimination degree greater than 2. Then with the graph (V_{n-1}, E_{n-1}) as the basis, where $V_{n-1} = \{v_n\}$ and $E_{n-1} = \emptyset$, we can construct the graph (V, E) by a sequence of production rules, adding vertices in the reverse of their elimination order. If vertex v_i has a single neighbor v in (V_{i-1}, E_{i-1}), then graph (V_{i-1}, E_{i-1}) is constructed from (V_i, E_i) by applying the Acyclic Branch rule with $w = v_i$. If vertex v_i has two neighbors u and v in (V_{i-1}, E_{i-1}), then graph (V_{i-1}, E_{i-1}) is constructed from (V_i, E_i) by applying either the Series or the Parallel-Series rule with $w = v_i$, depending on whether or not $(u, v) \in E_{i-1}$. This process proceeds until it reaches $(V_0, E_0) = (V, E)$.

□

References

[1] C. E. Shannon, "A Symbolic Analysis of Relay and Switching Circuits", *Trans. of the AIEE*, Vol. 57 (1938), pp. 713–723.

[2] J. A. Brzozowski and M. Yoeli, *Digital Networks*, Prentice-Hall, 1976.

[3] M. Lightner and G. Hachtel, "Implication Algorithms for Switch Level Functional Macromodeling, Implementation, and Testing." *19th Design Automation Conf.*, IEEE (July, 1982), pp. 691–698.

[4] M. Yoeli, and J. A. Brzozowski, "A Mathematical Model of Digital CMOS Networks", *Canadian Conf. on VLSI* (1985).

[5] R. E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Trans. on Computers* Vol. C-33, No. 2 (February, 1984) pp. 160–177.

[6] J. Hayes, "A Unified Switching Theory with Applications to VLSI Design", *Proc. IEEE*, Vol. 70, No. 10 (October, 1982), pp. 1140–1151.

[7] R. E. Bryant, *Boolean Analysis of MOS Circuits*, companion paper (1987).

[8] F. E. Hohn and L. R. Schissler, "Boolean Matrices and Combinational Circuit Design", *Bell Systems Technical Journal*, Vol. 34 (1955), pp. 177–202.

[9] G. Ditlow, W. Donath, and A. Ruehli, "Logic Equations for MOSFET Circuits", *International Symposium on Circuits and Systems*, IEEE (1983), pp. 752–755.

[10] I. N. Hajj, and D. Saab, "Symbolic Logic Simulation of MOS Circuits", *International Symposium on Circuits and Systems*, IEEE (1983).

[11] C. J. Terman, *Simulation Tools for Digital LSI Design*, PhD Thesis, MIT Dept. Elec. Eng. and Comp. Sci. (October, 1983).

[12] E. Cerny, and J. Gecsei, "Simulation of MOS Circuits by Decision Diagrams", *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, Vol. CAD-4, No. 4 (October, 1985), pp. 685–693.

[13] B. A. Carré, "An Algebra for Network Routing Problems", *J. Inst. Maths Applics.*, Vol. 7 (1971), pp. 273–294.

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

[15] D. J. Lehmann, "Algebraic Structures for Transitive Closure", *Theoretical Computer Science*, Vol. 4 (1977), pp. 59–76.

[16] R. E. Tarjan, "A Unified Approach to Path Problems", *J. ACM*, Vol. 23, No. 3 (July, 1981), pp. 577–593.

[17] M. A. Harrison, *Introduction to Switching and Automata Theory*, McGraw-Hill, 1965.

[18] M. R. Garey, and D. S. Johnson, *Computers and Intractability*, Freeman, 1979.

[19] I. Spillinger, and G. M. Silberman, "Improving the Performance of a Switch-Level Simulator Targeted for a Logic Simulation Machine", *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, Vol. CAD-5, No. 3 (July, 1986), pp. 396–404.

[20] R. E. Tarjan, "Fast Algorithms for Solving Path Problems", *J. ACM*, Vol. 23, No. 3 (July, 1981), pp. 594–614.

[21] E. C. Ogbuobiri, W. F. Tinney, and J. W. Walker, "Sparsity-Directed Decomposition for Gaussian Elimination on Matrices", *IEEE Trans. on Power Apparatus and Systems*, Vol. PAS-89, No. 1 (January, 1970), pp. 141–150.

[22] L. W. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, PhD Thesis, Univ. of California, Berkeley, Dept. of Elec. Eng. (1975).

[23] R. J. Lipton, D. J. Rose, and R. E. Tarjan, "Generalized Nested Dissection", *SIAM Journal on Numerical Analysis*, Vol. 16, No. 2 (April, 1979), pp. 346–358.

[24] R. J. Duffin, "Topology of Series-Parallel Networks", *J. Math. Anal. and Applications*, Vol. 10 (1965), pp. 303–318.

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

[26] L. A. Glasser, and D. W. Dobberpuhl, *The Design and Analysis of VLSI Circuits*, Addison-Wesley, 1985.

[27] N. H. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, 1985.

[28] M. Annaratone, *Digital CMOS Circuit Design*, Kluwer Academic Publishers, 1986.

[29] M. Horowitz, private communication (1985).

[30] C. Lutz, S. Rabin, C. Seitz, and D. Speck, "Design of the MOSAIC Element," *Conf. on Advanced Research in VLSI*, MIT (1984), pp. 1–10.

[31] A. V. Aho, and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, Volume II: Compiling*, Prentice-Hall, 1972.

[32] A. V. Aho, and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977.

[33] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle", *J. ACM*, Vol. 12, No. 1 (1965).

[34] M. A. Breuer, and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, 1976, p. 46.

[35] V. Pan, and J. Reif, "Efficient Parallel Solution of Linear Systems", Technical Report TR-02-85, Aiken Computation Laboratory, Harvard University, 1985.

[36] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Computers*, Vol. C-35, No. 8 (August, 1986) pp. 677–691.