Fundamental CAD Algorithms

Melvin A. Breuer, *Fellow, IEEE*, Majid Sarrafzadeh, *Fellow, IEEE*, and Fabio Somenzi

***Abstract*—Computer-aided design (CAD) tools are now making it possible to automate many aspects of the design process. This has mainly been made possible by the use of effective and efficient algorithms and corresponding software structures. The very large scale integration (VLSI) design process is extremely complex, and even after breaking the entire process into several conceptually easier steps, it has been shown that each step is still computationally hard. To researchers, the goal of understanding the fundamental structure of the problem is often as important as producing a solution of immediate applicability. Despite this emphasis, it turns out that results that might first appear to be only of theoretical value are sometimes of profound relevance to practical problems.**

VLSI CAD is a dynamic area where problem definitions are continually changing due to complexity, technology and design methodology. In this paper, we focus on several of the fundamental CAD abstractions, models, concepts and algorithms that have had a significant impact on this field. This material should be of great value to researchers interested in entering these areas of research, since it will allow them to quickly focus on much of the key material in our field. We emphasize algorithms in the area of test, physical design, logic synthesis, and formal verification. These algorithms are responsible for the effectiveness and efficiency of a variety of CAD tools. Furthermore, a number of these algorithms have found applications in many other domains.

***Index Terms*—Algorithms, computer-aided design, computational complexity, formal verification, logic synthesis, physical design, test.**

I. Introduction

THE TECHNOLOGICAL revolution represented by very large scale integration (VLSI) has opened new horizons in digital design methodology. The size and complexity of VLSI systems demands the elimination of repetitive manual operations and computations. This motivates the development of automatic design systems. To accomplish this task, fundamental understanding of the design problem and full knowledge of the design process are essential. Only then could one hope to efficiently and automatically fill the gap between system specification and manufacturing. Automation of a given design process requires its algorithmic analysis. The availability of fast and easily implementable algorithms is essential to the discipline.

Because of the inherent complexity of the VLSI design problem, it is partitioned into simpler subproblems, the analysis of each of which provides new insights into the original problem as a whole. In this framework, the objective is to view the VLSI design problem as a collection of subproblems; each subproblem should be efficiently solved and the solutions must be effectively combined.

Given a problem, we are to find efficient solution methods. A data structure is a way of organizing information; sometimes the design of an appropriate data structure can be the foundation for an efficient algorithm. In addition to the design of new data structures, we are interested in the design of efficient solutions for complex problems. Often such problems can be represented in terms of trees, graphs, or strings.

Once a solution method has been proposed, we seek to find a rigorous statement about its efficiency; analysis of algorithms can go hand-in-hand with their design, or can be applied to known algorithms. Some of this work is motivated in part by the theory of NP-completeness, which strongly suggests that certain problems are just too hard to always solve exactly and efficiently. Also, it may be that the difficult cases are relatively rare, so we attempt to investigate the behavior of problems and algorithms under assumptions about the distribution of inputs. Probability can provide a powerful tool even when we do not assume a probability distribution of inputs. In an approach called randomization, one can introduce randomness into an algorithm so that even on a worst case input it works well with high probability.

Most problems that arise in VLSI CAD are NP-complete or harder; they require fast heuristic algorithms, and benefit from error bounds. Robustness is very important—the program must work well even for degenerate or somewhat malformed input. Worst case time complexity is important, but the program should also be asymptotically good in the average case, since invariably people will run programs on much larger inputs than the developers were anticipating. It is also important that the program run well on small inputs. Any algorithms used must be simple enough so that they can be implemented quickly and changed later if necessary.

In this paper, we describe fundamental algorithms that have been proposed in the area of test, physical design, logic synthesis, and formal verification. This paper is organized as follows. In Section II, we review fundamental algorithms in the area of test. Then, in Section III, we address physical design problems and review various techniques. In Section IV, we study logic synthesis and formal verification. Finally, we conclude in Section V.

II. Fundamental Algorithms in Test

A. Introduction

In this section, we focus on several issues related to post-manufacturing testing of digital chips. One comprehensive test occurs after packaging, and often involves the use of automatic

Manuscript received January 31, 2000. This paper was recommended by Associate Editor M. Pedram.

M. A. Breuer is with the Department of Electrical Engineering - Systems, University of Southern California, Los Angeles, CA 90089 USA (e-mail: mb@poisson.usc.edu).

M. Sarrafzadeh is with the Computer Science Department, the University of California, Los Angeles, CA 90095 USA (e-mail: majid@cs.ucla.edu).

F. Somenzi is with the Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80302 USA (e-mail: Fabio@Colorado.edu).

Publisher Item Identifier S 0278-0070(00)10450-6.

test equipment (ATE). Another involves testing of chips in the field. A major part of these tests are carried out using nonfunctional data, at less than functional clock rates, and where only static (logic level) voltages are measured. This aspect of the test problem has been highly automated and is the focus of our attention. The major subareas of test that lie within the scope of CAD include test generation for single stuck-at-faults (SSFs), diagnosis, fault simulation, design-for-test (DFT) and built-in self-test (BIST). The latter two topics deal primarily with test synthesis and will not be dealt with in this paper. For a general overview of this topic, the reader is referred to [1].

Initially, we restrict our attention to combinational logic. A *test* for a stuck-at fault in a combinational logic circuit C consists of an input test pattern that 1) produces an error at the site of the fault and 2) propagates the error to a primary output. *Automatic test pattern generation* (ATPG) deals with developing an algorithm for constructing a test pattern t that detects a fault f. *Diagnosis* deals, in part, with 1) generating tests that differentiate between a subset of faults and 2) given the results of applying a test and observing its response, determining what faults can or cannot exists in C. *Fault simulation* deals with determining which faults out of a class of faults are detected by a test sequence. In addition, the actual output sequence of a faulty circuit can be determined.

To automatically generate fault detection and diagnostic tests for a sequential circuit is quite complex, and for most large circuits is computationally infeasible. Thus, designers have developed *design-for-test* techniques, such as scan design, to simplify test generation. Going a step further, test engineers have developed structures that can be embedded in a circuit that either totally or to a large extent eliminate the need for ATPG. These structures *generate* tests in real or near real time within the circuit itself, and *compact* (compress) responses into a final *signature*. Based upon the signature one can determine whether or not the circuit is faulty, and in some cases can actually diagnose the fault. This area is referred to as *built-in self-test*.

Two key concepts associated with test generation are *controllability* and *observability*. For example, to generate a test for a line A that is stuck-at-1, it is necessary that the circuit be set into a state, or controlled, so that in the fault free circuit $A = 0$. This creates an error on line A. Next it is necessary that this error be propagated to an observable signal line such as an output. Scan design makes test generation much easier since flip-flops can be easily made to be pseudoobservable and controllable.

There are five key components associated with most test algorithms or related software systems; namely, 1) a fault model, 2) a fault pruning process, 3) a value system and data structure, and 4) the test procedure itself. The test procedure may deal with test generation, fault simulation or diagnosis.

Manufacturing tests deal with the problem of identifying defective parts, e.g., a defective chip. Defects, such as extra metal or thin gate oxide are often *modeled* using functional concepts, such as a line stuck-at one or zero, two lines shorted together, or a gate or path whose delay is unacceptably high. In general, the number of faults associated with many models is extremely high. For example, if there are n signal lines in a circuit, the number of multiple stuck-at faults is bounded by 3^n. In many cases, researchers have developed quite sophisticated techniques for reducing the number of faults that need be explicitly considered by a test system. These techniques fall into the general category of *fault pruning*, and include the concepts of fault dominance and equivalence.

Very often the efficiency of a test system is highly dependent on the data structures and value system employed. In test generation, a *composite* logic system is often used so one can keep track of the logic value of a line in both a fault-free and faulty circuit.

Finally, the test algorithm itself must deal with complex issues and tradeoffs related to time complexity, accuracy and fault coverage.

To give the reader a flavor of some of the key results in this field, we focus on the following contributions: 1) the D-algorithm test generation methodology for SSFs, 2) concurrent fault simulation, and 3) effect-cause fault diagnosis.

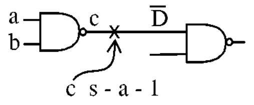
B. Test Generation for Single Stuck-At Faults in Combinational Logic

The D-Algorithm: The problem of generating a test pattern t for a SSF f in a combinational logic circuit C is an NP-hard problem, and is probably the most famous problem in testing. In 1960, J. Paul Roth published his now famous D-algorithm [2], which has remained one of the center pieces of our field. This work employed many important contributions including the use of the cubical complex notation, backtrack programming to efficiently handle implicit enumeration, and the unique concepts of D-cubes, error propagation (D-drive) and line justification.

The D-algorithm employs a five-valued composite logic system where $\mathbf{X} = x/x$, $\mathbf{1} = 1/1$, $\mathbf{0} = 0/0$, $\mathbf{D} = 1/0$ and $\overline{\mathbf{D}} = 0/1$. Here, a/b implies that a is the value of a line in the fault free circuit, and b is its value in the faulty circuit. $\mathbf{X}$ represents an unspecified logic value. Initially all lines in a circuit are set to $\mathbf{X}$. A $\mathbf{D}(\overline{\mathbf{D}})$ represents an error in a circuit. To create an initial error one sets a line that is s–a–0 to a 1, represented by a $\mathbf{D}$, or if s–a–1 to a 0, represented by a $\overline{\mathbf{D}}$.

A form of forward and backward simulation is carried out by a process known as **implication**, where a line at value $\mathbf{X}$ is changed to one of the other line values. Fig. 1(b) shows the truth table for a NAND gate. It is easy to extend this table to include composite line values. So if $a = \mathbf{0}$ and $b = \mathbf{X}$, then forward implication would set $c = \mathbf{1}$. Fig. 1(c) illustrates some examples of backward implication. $\mathbf{D}s$ and $\overline{\mathbf{D}}s$ are implied forward and backward in the same manner based on the truth table shown in Fig. 1(d). Note that for any cube, such as $(abc) = (\mathbf{1D}\overline{\mathbf{D}})$ all "$\mathbf{D}$" entries can be complement to form another logically correct cube, such as $(\mathbf{1}\overline{\mathbf{D}}\mathbf{D})$.

The D-algorithm employs the concept of *J-frontier* and *D-frontier* to keep track of computations to be done as well as lead to an orderly backtrack mechanism. The *J-frontier* is a list that contains the name of each gate whose output is assigned a logic value that is not implied by the input values to the gate, and for which no unique backward implications exists. For example if $a = b = \mathbf{X}$ and $c = \mathbf{1}$, then there are two ways (choices) for satisfying this assignment, namely $a = \mathbf{0}$ or $b = \mathbf{0}$. These gates are candidates for *line justification*.



(a)

a	b	c	
0	x	1	ON-ARRAY
x	0	1	
1	1	0	OFF-ARRAY

(b)

Before a	b	c	After a	b	c
x	x	0	1	1	0
x	1	1	0	1	1

(c)

a	b	c
1	D	$\overline{\mathrm{D}}$
D	1	$\overline{\mathrm{D}}$
D	D	$\overline{\mathrm{D}}$
D	$\overline{\mathrm{D}}$	0

(d)

Fig. 1. Logic system.

The *D-frontier* is a list of all gates whose outputs are $\mathbf{X}$ and which have one or more $\mathbf{D}$ or $\overline{\mathbf{D}}$ input values. These gates are candidates for *D-drive*.

A procedure *imply-and-check* is executed whenever a line is set to a new value. This procedure carries out all forward and backward (unique) implications based on the topology and gates in a circuit. The version of *imply-and-check* presented here is an extention of the original concepts and makes the procedure somewhat more efficient.

Example 1: As an example, consider the circuit shown in Fig. 2(a). Since all gates have a single output we use the same symbol to identify both a gate and its output signal. Also, we denote line X *s–a–*1(0) by $X_1(X_0)$.

Consider the fault F_0. We can consider a pseudoelement F^* placed on this line whose input is $\mathbf{1}$ and output is $\mathbf{D}$. We use the notation $\mathbf{V}(i)$ to denote a line value $\mathbf{V}$ assigned in step i of the algorithm. We also use the symbols $\Rightarrow$, $\Leftarrow$ to denote forward and backward implication, respectively, and J to denote justification. Since $F = \mathbf{1}(0)$, then $B = C = \mathbf{1}(0)$. The *D-frontier* $= \{K, L\}$, and *J-frontier* $= \{\phi\}$.

To drive a $\mathbf{D}$ or $\overline{\mathbf{D}}$ to a primary output, we can select an element from the *D-frontier* and carryout a process known as *D-drive*. If we select gate K, then by assigning $A = \mathbf{1}(1)$ we get $K = \overline{\mathbf{D}}(1)$, and implications results in $G = \mathbf{0}(1)$ and $I = \mathbf{1}(1)$. Now *D-frontier* $= \{N, L\}$ and the *J-frontier* is still empty. If we next drive the error ($\overline{\mathbf{D}}$) through gate N to the primary output, we require $L = M = \mathbf{1}(2)$. Again, carrying out *imply-and-check* we get $H = \mathbf{0}(2)$ and $I = \mathbf{1}(2)$ which in turn implies $L = \overline{\mathbf{D}}(2)$. But since L has already been assigned the value $\mathbf{1}(2)$ a conflict exists. Conflicts are dealt with by backtracking to the last step where a choice exists and selecting a different choice. Naturally, this must be done in an orderly way so that all possible assignments are implicitly covered. In our case, we undo all assignments associated with step 2 and select L rather than N from the *D-frontier*. This results in $L = \overline{\mathbf{D}}(2)$ and eventually with $N = \mathbf{D}(3)$ and a test $A = B = C = D = \mathbf{1}$.□

This example illustrates the need for *multiple-path sensitization*. Note that the test also detects additional faults such as B_0, C_0 and N_0. Later, we see that fault simulation is an efficient way of determining all the SSFs detected by a test pattern.

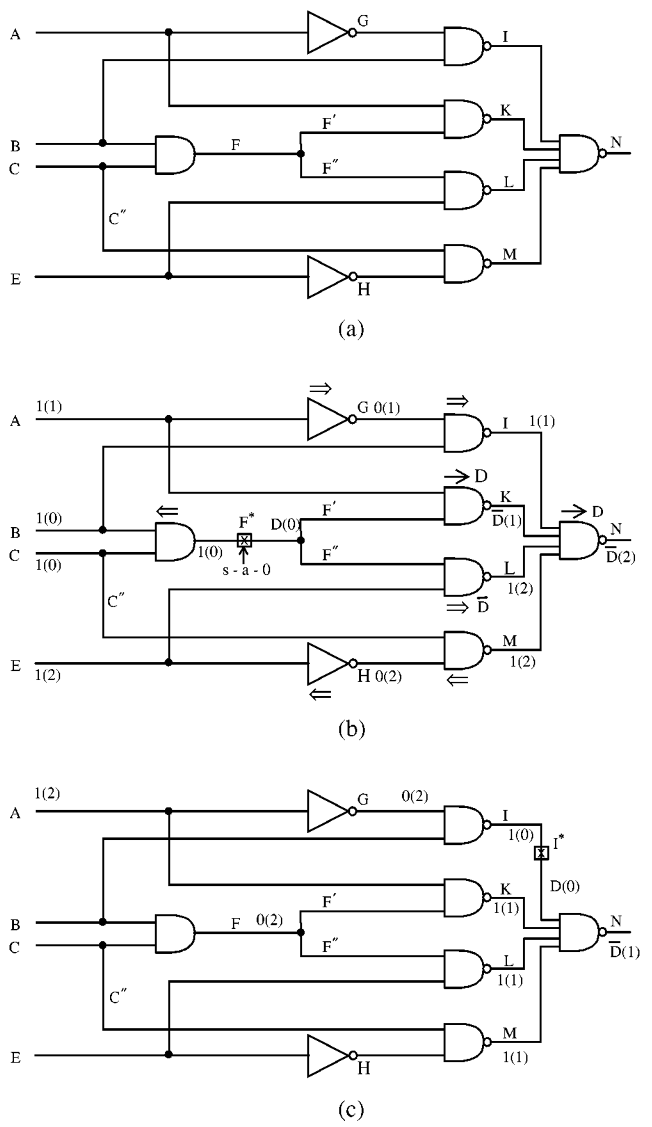


Fig. 2. Circuit for Example 1.

Example 2: To illustrate the concept of line justification consider the fault I_0. Here, we have $I^* = \mathbf{D}(0)$ and $I = \mathbf{1}(0)$ as shown in Fig. 2(c). After we *D-drive* through gate N we have $N = \overline{\mathbf{D}}(1)$ and *J-frontier* $= \{I, K, L, M\}$. Assume we chose to remove I from the *J-frontier* to be processed first. Then one way to justify $I = \mathbf{1}$ is to set $G = \mathbf{0}(2)$, which results in several implications. We continue solving justification problems until *J-frontier* $= \{\phi\}$. If a conflict occurs, backtracking is carried out. □

Fig. 3 shows the pseudocode for the D-algorithm. If no test exist for a fault, the fault is said to be *redundant* and the algorithm terminates in FAILURE. Backtracking is automatically taken care of via the recursive nature of the procedure calls. The controlling value for AND and NAND gates is zero, and for OR and NOR it is one. In addition, AND and OR have an inversion parity of zero, while NOT, NOR, and NAND have an inversion parity of one.

There are numerous other test generation algorithms for SSFs many of which deal with further refinements of the D-algorithm,

```
D-alg()
begin
    if Implyandcheck() = FAILURE then return FAILURE
    if(error not at PO) then
      begin
          if D-frontier = $\phi$ then return FAILURE
          repeat
              begin
                  select an untried gate (G) from D-frontier
                  c= controlling value of G
                  assign $\bar{c}$ to every input of G with value $x$
                  if D-alg() = SUCCESS then return SUCCESS
              end
          until all gates from D-frontier have been tried
          return FAILURE
      end
    /* error propagated to a PO */
    if J-frontier = $\phi$ then return SUCCESS
    select a gate (G) from the J-frontier
    c = controlling value of G
    repeat
      begin
          select an input (j) of G with value $x$
          assign c to j
          if D-alg() = SUCCESS then return SUCCESS
          assign $\bar{c}$ to j /* reverse decision */
      end
    until all inputs of G are specified
    return FAILURE
end
```

Fig. 3. The D-algorithm.

such as PODEM [3] and FAN [4]. ATPG algorithms have also been developed for other fault models such as multiple stuck-at faults, bridging faults (shorts) and delay faults.

C. Fault Simulation

Fault simulation is the process of simulating a circuit S with respect to an input sequence T and set of faults F. This process is normally done for a variety of reasons, such as 1) to determine which faults in F are detected by T, i.e., produce an erroneous output, or 2) determine the response of S to T for each fault $f \in F$. In more general terms, fault simulation is done to determine fault coverage, i.e., the percent of faults detected by T with respect to S and a class of faults. We focus on the class of single stuck-at faults. Techniques for determining the fault coverage with respect to certain class of delay faults require substantially different techniques. Other applications for fault simulation are to guide ATPG in selecting a fault to process, and to provide data for fault diagnostic, such as fault dictionaries. Most fault simulators employ a zero or unit delay gate level model. We use the term good circuit and fault-free circuit interchangeably.

Gate level good circuit logic simulation technology is quite mature, and includes mechanisms such as compiled-driven, table-driven and event-driven simulation. In addition, logic simulators can handle 1) complex timing models, 2) bidirectional devices, and 3) multivalued logic including unknown, transitions and high-impedance states. Many fault simulators are built as extensions to good circuit logic simulators.

Since a fault in a circuit produces a different circuit, it too can be simulated using a conventional logic simulator. If its response to a test T is different from that of the fault-free circuit, we say that T detects the fault. Tests having SSF coverage above 98% are often sought. Since a fault-free and faulty circuit differ in a very minor way, e.g., one line may be stuck-at 0 or 1, the same circuit description is used for both. In some cases, a “patch” in the description or code is used so that when a signal line is being processed, a check is made as to whether or not it is associated with a fault. If so an appropriate action is taken.

For a circuit with n signal lines, the number of SSFs is $O(n)$, and hence simulating these faults one at a time can take $O(n)$ times longer than a good circuit simulation. So the primary emphasis in fault simulation is on efficiency.

There are two major classes of circuits addressed by fault simulators, namely sequential circuits (including asynchronous circuits), and those employing full scan and/or logic BIST. For the latter category the circuit is considered to be combinational and delay is generally ignored. In all cases, numerous faults need to be processed. Because the order in which faults are processed appears to be a third order effect on efficiency, we ignore it in this presentation. For sequential circuits, test patterns must be processed in their natural order, i.e., in the order they occur in a test sequence. For combinational circuits, however, test patterns can be processed in an arbitrary order if we ignore the concept of fault dropping.[1]

To appreciate and help identify the key concepts in fault simulation one must understand the attributes that researchers have combined in the evolution of these systems. Some of these concepts are listed in Table I.

In the area of accelerating simulation time, developers have noted that computers are word oriented, and when employing logic operators, all bits of a word are processed simultaneously and independently. So, for example in *parallel fault simulation*, a unique fault can be processed simultaneously (concurrently) in each bit position of a word. Seshu [5] exploited this technique when simulating sequential circuits. On the other hand Waicukauski *et al.* [6] focused on simulating combinational circuits and hence simultaneously processed a unique test pattern in each bit position of a word.

Another important observation related to fault simulation is that the logic state of most faulty circuits very closely tracks the state of the fault-free circuit, pattern by pattern. That is, the binary value of a flip-flop x at simulation clock cycle t is usually the same in the fault-free circuit as in some arbitrary faulty circuit. Hence one could consider simulating the “difference” between each fault circuit and the fault-free circuit. We refer to this as *strong temporal correlations*. This observation is the kernel behind Deductive [7] and Concurrent [8] fault simulation.

In the next few paragraphs, we describe the concept of concurrent fault simulation.

Concurrent Fault Simulation: Concurrent fault simulation explicitly simulates a fault-free circuit using classical table-driven event-direct techniques. In addition, since most faulty circuits are strongly temporarily correlated to the fault-free

[1] *Fault dropping* refers to the technique of not simulating a fault once it is detected by a test pattern.

TABLE I
A DESIGN SPACE FOR FAULT SIMULATORS

Acceleration Techniques:	
	Word parallel
	Strong temporal correlations
	Implicit fault relations
Problem Characteristics:	
	Sequential vs combinational logic
	Multiple patterns
	Multiple faults
Versatility:	
	Logic values
	Delay models

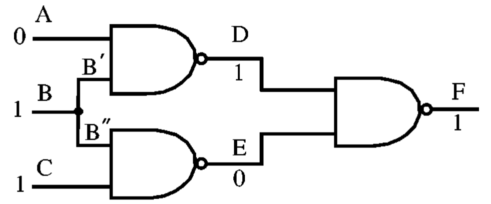


Fig. 4. Example circuit.

circuit, they are simulated *implicitly*, i.e., events that occur in the fault-free circuit also occur in most faulty circuits. Those cases where this is not the case are simulated *explicitly*.

Let S be a logic circuit and S_f the same circuit except it contains a fault f in some signal line. We associate with each element x in S, such as a gate or flip-flop, a *concurrent* fault list, denoted by CL_x. CL_x contains entries of the form $(f,\, V_{x_f})$, where f is a fault and V_{x_f} is a set of signal values. Let x_f denote the replica or image of x in S_f. Note that in general f is not related to x. For example, referring to Fig. 4, fault D_1 defines a circuit S_{D_1}, and the image of gate F in this circuit is F_{D_1}.

Let $V_x(V_{x_f})$ denote the ensemble of input, output, and possibly internal state values of $x(x_f)$. Note that if x were a flip-flop or register it would have a state. A fault g is said to be a *local fault* of x if it is associated with either an input, output, or state of x. Referring to Fig. 4, the local faults of F are F_1, F_0, D_1, D_0, E_1, and E_0. At a specific point in simulation time, assume the signal values are as shown in Fig. 4. In the fault free circuit, gate F would be associated with the pair $(0,\, V_0)$, where $V_0 = (D,\, E,\, F) = (0,\, 1,\, 1)$, were the fault-free circuit is denoted by the index zero. The element F would be associated with several entities, including the local fault entries $(D_0,\, (0,\, 0,\, 1))$, and $(C_0, (1,\, 1,\, 0))$, where the fault C_0 forces line E to a one. The path C–E–F is a *sensitized path*, and there exist one stuck-at fault on each segment of this path that is detected by the input pattern.

During simulation CL_x contains the set of all elements x_f that differ from x at the current simulated time. If $V_{x_f} \neq V_x$, then $(f,\, V_{x_f}) \in CL_x$. Also, if f is a local fault of x, then $(f,\, V_{x_f}) \in CL_x$ even if $V_{x_f} = V_x$.

A fault is said to be *visible* on a line i when the value i in S and S_f differ. Concurrent simulation employs the concept of events and simulation is primary involved in updating the dynamic data structures and processing the concurrent fault lists.

The initial content of each CL_x consists of entries corresponding to the local faults of x. Concurrent simulation normally employs fault dropping and, thus, local faults of x remain in CL_x until they are detected, and subsequently are dropped. During simulation new entries in CL_x represent elements x_f whose values become different from the values of x. These are said to *diverge* from x. Conversely, entries removed from CL_x represent elements x_f whose values become identical to those of x. These are said to *converge* to x. Efficient dynamic memory management is needed in order to process lists quickly and not waste storage space.

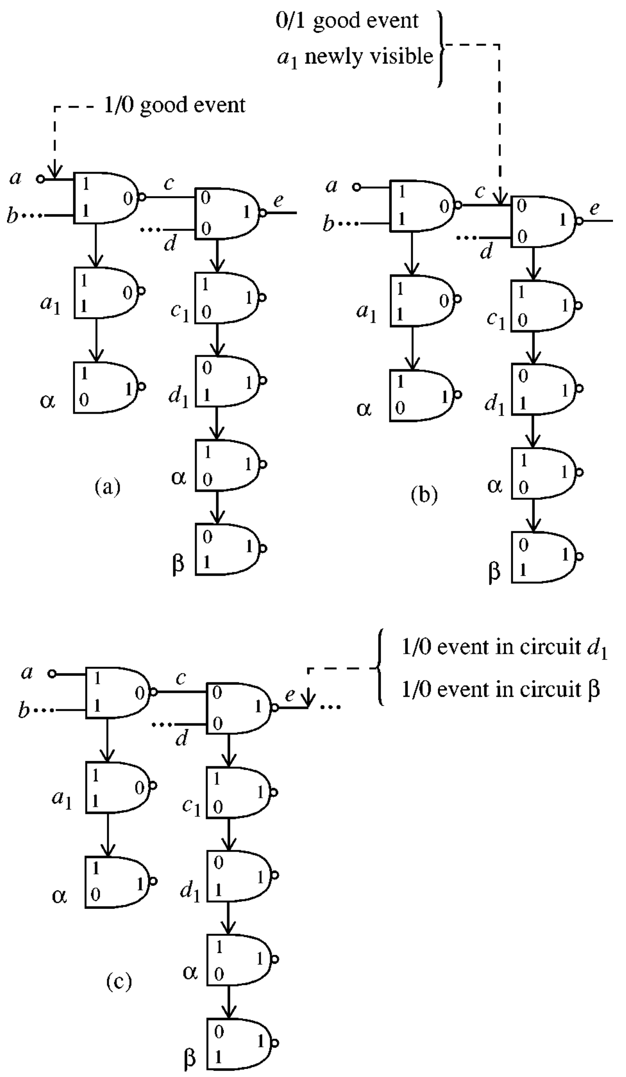


Fig. 5. Changes in fault lists and creation of events during simulation.

Before presenting the algorithm for concurrent simulation, we illustrate the major concepts with an example.

Example 3: Consider the circuit shown in Fig. 5(a). Here, only two elements (gates) in S are depicted, namely c and e. The fault list are shown symbolically as gates linked together. Hence, the first element in CL_c is $(a_1,\, V_{a_1})$, where $V_{a_1} = (1,\, 1,\, 0))$. Note that a_1 is a local fault, while α is not. The entry $(\alpha,\, (1,\, 0,\, 1)) \in CL_c$ since $(1,\, 0,\, 1) \neq (1,\, 1,\, 0)$. To process the current event "line a changes from a one to a zero in the fault free circuit," we turn to Fig. 5(b). Here, we see that because line a has a new value, when gate c is evaluated we must schedule the future event "line c in fault free circuit changes from zero to one at present time plus delay of gate c." Note that the event on line a occurs not only in the fault free circuit S but implicitly in

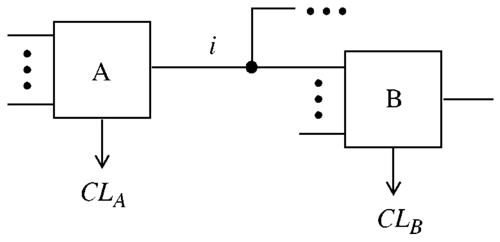


Fig. 6. Elements A and B.

all fault circuits that do not have entries in CL_c. The entries in CL_c are processed explicitly. Note that the event on line a does not effect the first entry in CL_c, since it corresponds to line a s–a–1. Evaluating the entry corresponding to gate c_α does not result in a new event since the output of gate c and c_α will be the same once c is updated.

When the value of c is eventually updated, fault a_1 is identified as being newly visible on line c.

The good event on line c does not produce an event on line e. As seen in Fig. 5(c), the newly visible fault a_1 produces an entry in CL_e. The new value of c also produces the following changes to CL_e: for entry $d_1(\beta)$, the output changes from one to zero that in turn produces an event tied to $d_1(\beta)$ only. It is, thus, seen that there are two types of events, namely good circuit events that apply to not only the fault-free circuit but also all faulty circuits, except for those that attempt to set a line that is s–a–δ to a value other than δ, and events that are specific to a faulty circuit. □

We now describe the concurrent simulation procedure, using the elements A and B shown in Fig. 6.

At a specific scheduled time, we can have an event on line i in the fault-free circuit and/or in several faulty circuits. The set of scheduled simultaneous events is called a *composed event* and is denoted by (i, L), where L is a list of pairs of the (f, v'_f), f is the name (index) of a fault and v'_f is the scheduled value.

Fig. 7 describes how the composed event at A is processed. Here, $v(v_f)$ is the current value of line i in the circuit $S_0(S_f)$. If an event in the fault-free circuit occurs of the form line i changes from v to v', then all entries in CL_A are processed; otherwise only the pertinent entries in CL_A are analyzed.

At the termination of this procedure NV contains the list of all newly visible faults. Next it is necessary to schedule future events created by the processing of (i, L) with respect to CL_A. This is done by processing every element on the fan-out list of i, such as B.

If an event occurs in the fault-free circuit, then B is processed in a normal way. The processing of an element $B_f \in CL_B$ depends on which lists CL_B, L and/or NV contain f. For the sake of completeness the appropriate actions are summarized in Table II.

In some of these cases, we refer to the concept of "activate B_f." *Activated elements* are individually evaluated and any events produced are merged into composed events. If an event is produced in the fault-free circuit, then the process of deleting entries from the fault lists will be done when the composed event is retrieved from the events list and processed, as seen in Fig. 7.

```
NV = 0
if $i$ changes in the good circuit then
  begin
    set $i$ to $v'$ in the good circuit
    for every $f \epsilon CL_A$
      begin
        if $f \epsilon L$ then
          begin
            set i to $v'_f$ in circuit $f$
            if $V_{A_f} = V_A$ then delete $f$ from $CL_A$
          end
        else /* no event in circuit $f$ */
          if $v_f = v$ then add newly visible fault $f$ to $NV$
          else if $V_{Af} = V_A$ then delete $f$ from $CL_A$
      end
  end
else /* no good event for $i$ */
  for every $f \epsilon L$
    begin
      set $i$ to $v'_f$ in circuit $f$
      if $V_{A_f} = V_A$ then delete $f$ from $CL_A$
    end
```

Fig. 7. Processing of a composed event (i, L) at the source element A.

If no event in the fault-free circuit is generated, any activated element B_f that does not generate an event should be compared to the fault-free element, and if their values agree, f should be deleted from CL_B.

In summary, concurrent simulation has many important attributes, such as the ability to 1) accommodate complex delay models for the elements being simulated, 2) employ multivalued logic; 3) except for local faults it explicitly simulate only elements that differ (in logic values) from the fault-free circuit, and implicitly simulate all other elements; 4) accommodate a wide range of circuit structures including combinational, synchronous and asynchronous logic, and 5) employ a wide range of primitive elements such as gates, ALUs and memories. Its primary disadvantages are 1) implementations complexity, 2) large storage requirements, and 3) need for efficient list processing and memory management.

Critical path tracing [9] is a technique that is radically different from concurrent fault simulation. It is intended for combinational circuits, borrows on techniques from test generation, and employs special procedures to efficiently handle fan-out free regions of a circuit as well as cones (single output) of logic.

D. Logic-Level Fault Diagnosis

Fault diagnosis deals with the process of identifying what is wrong with a circuit given the fact that the circuit produced the wrong response to a test. It is often assumed that the fault is an element of a well-defined class, such as the SSFs. Faulty circuits contain defects and often these defects do not correspond to the simple fault model assumed, in which case fault diagnosis can lead to ambiguity or misdiagnosis. Also, since some faults are equivalent to others, without probing inside a circuit, fault resolution is sometimes not precise.

One of the most common forms of diagnosis is carried out using a fault dictionary [5]. Here, using a fault simulator one

TABLE II
PROCESSING ELEMENT $B_f \in CL_B$

Case 1: (B_f exists in CL_B and no independent event on i occurs in N_f). If a good event exists and it can propagate in N_f, then activate B_f. The good event on line i can propagate in the circuit f if $v_f = v$ and f is not the local s-a-v fault on the input i of B_f.
Case 2: (B_f exist in CL_B and an independent event on i occurs in N_f). Activate B_f.
Case 3: (An independent event on i occurs in N_f, but f does not appear in CL_B). Add an entry f to CL_B and activate B_f.
Case 4: (f is newly visible on line i and does not appear in CL_B). Add an entry for f to CL_B.
Case 5: (f is a newly visible fault on line i, but an entry for f is already present in CL_B). No action.

can build an array indicating test pattern number, fault number (index) and response. Given the response from a circuit under test (CUT) one can search this dictionary for a match which then indicates the fault. This diagnostic methodology leads to very large dictionaries and is not applicable to multiple faults. This form of diagnosis is referred to as **cause–effect analysis**, where the possible causes (faults) lead to corresponding effects (responses). The faults are explicitly enumerated prior to constructing the fault dictionary.

Effect–Cause Analysis: In this section, we briefly describe the **effect–cause analysis** methodology [10], [11]. This diagnosis technique has the following attributes: 1) it implicitly employs a multiple stuck-at fault model and, thus, does not enumerate faults; 2) it identifies faults to within equivalence classes; and 3) it does not require fault simulation or even the response from the fault-free circuit.

Let C be a model of a fault-free circuit, C^* an instance of C being tested, T the test sequence, and $R(R^*)$ the response of $C(C^*)$ to T. In effect-cause analysis, we process the actual response R^* (the effect) to determine the faults in C^* (the cause). The response R is not used. Effect-cause analysis consists of two phases. In the first phase, one executes the *Deduction Algorithm,* where the internal signal values in C^* are deduced. In the second phase, one identifies the status of lines in C^*, i.e., which are fault-free or normal (n), which only take on the values 0 or 1, and which cannot be s–a–0 or s–a–1, denoted by $\overline{0}$ and $\overline{1}$, respectively.

Note that by carrying out a test where only the primary output lines are observable, it is not feasible to always identify a fault to a specific line. This occurs for reasons such as fault equivalence and fault masking. A few examples will help clarify the complexity of this problem.

Example 4: Consider a single output cone of logic C^*.

a) If the output line is s–a–δ, where $\delta \in \{0, 1\}$, then all other faults in C^* are masked, i.e., cannot be identified by an input/output (I/O) experiment and have no impact on the output response.
b) Assume the output is driven by an AND gate. Then any input line to this gate that is s–a–0 is equivalent (indistinguishable) from the output s–a–0 as well as any other input s–a–0.

□

Effect–cause analysis relies on many theoretical properties of logic circuits, several of which were first identified during the evolution of this work. We next list those results that are most important for understanding the development and correctness of the Deduction Algorithm.

We assume a line in C^* is either normal, s–a–0 or s–a–1. A *normal path* is a sequence of normal lines separated by fault-free gates.

- (Normal Path): The logic values of an internal line ℓ can be deduced from an I/O experiment only if there exists at least one normal path connecting ℓ with some PO.

We assume a natural lexicographic ordering of the lines in C and of test patterns in T. Therefore, we do not distinguish between a signal line ℓ and the ℓth line in a circuit, nor the test pattern t and the tth test pattern.

Let $v^*_{t\ell}$ be the value of signal line ℓ in C^* when test pattern t is applied.

Let matrix $V^* = [v^*_{t\ell}]$. In C, the signal values are denoted by $V^0 = [v^0_{t\ell}]$.

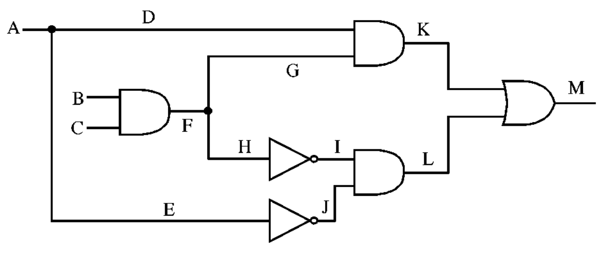
- **(PO):** If line ℓ is s–a–δ, $\delta \in \{0, 1\}$, then $v^*_{t\ell} = \delta \; \forall t \in T$.
- **(P1):** Let ℓ be the output of gate g. If ℓ is normal then for all $t \in T$, $v^*_{t\ell}$ and the values of the inputs to g must cover a primitive cube of g.
- **(P2):** If k is a fan-out branch (FOB) of j, then lines k and j have the same values in every test.
- **(P3):** If line i is a normal PI, then $v^*_{ti} = v^o_{ti} \; \forall t \in T$.
- **(P4):** Consider the basic primitive gates AND, OR, NAND, NOR, and INVERTER. Then for every pair of primitive cubes in which the output of a gate has complementary values, there exists an input with complementary values.
- **(Complete Normal Path):** If a P0 line w has complementary values in t and t', then there exists at least one complete normal path in C^* between some PI and w, and every line on this path has complementary values in t and t'.

The process of analyzing C and t results in conclusions that are referred to as *forced-values* (FVs). Determining forced values is similar to carrying out an implication process based on T. Forced values are determined as a preprocessing step to the Deduction Algorithm.

Definition 1: Line ℓ has *property* FV^c in test t, where $c \in \{0, 1\}$, iff either $v^*_{t\ell} = c$ or else $v^*_{t'\ell} = \overline{c} \; \forall t' \in T$. A shorthand notation for this concept is to write $FV_{t\ell} = c$. □

- **(P5):** For every PI i, $FV_{ti} = v^0_{ti} \; \forall t \in T$, i.e., the expected values of a PI are its FVs.
- **(P6):** If k is a FOB of j, then $FV_{tk} = FV_{tj} \; \forall t \in T$, i.e., a fan-out branch has the FVs of its stem.
- **(P7):** Let ℓ be the output of a noninverting (inverting) gate having inputs $x_1, x_2, \ldots, x_p$. Then $FV_{t\ell} = c(\overline{c})$ iff $FV_{tx_i} = c$ for $i = 1, 2, \ldots, p$.

Here, we see that we can deduce information about the output of a gate if all the inputs of the gate have FV^c for test t.



(a)

	A	B	C	D	E	F	G	H	I	J	K	L	M
t_1	0	1	1	0	0	1	1	1	0	1	0	0	0
t_2	1	1	0	1	1	0	0	0	1	0	0	0	0
t_3	1	0	1	1	1	0	0	0	1	0	0	0	0
t_4	1	1	1	1	1	1	1	1	0	0	1	0	1
t_5	0	0	1	0	0	0	0	0	1	1	0	1	1

(b)

	A	B	C	D	E	F	G	H	I	J	K	L	M
t_1	0	1	1	0	0	1	1	1	0	1			
t_2	1	1	0	1	1					0			
t_3	1	0	1	1	1					0			
t_4	1	1	1	1	1	1	1	1	0	0	1	0	
t_5	0	0	1	0	0					1			

(c)

Fig. 8. (a) Circuit to be analyzed. (b) Expected values. (c) Forced values.

Notation: Let $\mathcal{F}_{\ell}^{c}$ be the set of tests in which ℓ has $FV\ c$, i.e., $\mathcal{F}_{\ell}^{c} = \{t\colon FV_{t\ell} = c\}$.

- **(P8):** If for some $t' \in T$, $v_{t'\ell}^{*} = c$, then $v_{t\ell}^{*} = c\,\forall\, t \in \mathcal{F}_{\ell}^{c}$. This result represent vertical (between tests) implication and will be illustrated later.

Properties P5–P7 allow one to determine FVs at the PIs and move FVs forward through a circuit.

As stated previously, if $FV_{t\ell} = c$, then ℓ has property FV^{c} in test t independent of the fault situation in C^{*} and, thus, independent of the status of other lines in C^{*}. However, there are situations when the status of a line may depend on the value of other lines. This leads to the concept of conditional forced values (CFVs) [10].

Example 5: Consider the circuit shown in Fig. 8(a). In Fig. 8(b), we show the expected (fault-free) values in response to the test $t_1, t_2, \ldots, t_5$. While only the values applied at A, B, and C are used in the deduction process, the other values are of interest for comparison. Fig. 8(c) shows the forced values resulting from using properties P5–P7. Note that the forced value are a subset of the values obtained if C were simulated for each test pattern t_i. These values are determined by a preprocessing step. The steps in the Deduction Algorithm are guided by the use of a decision tree (see Fig. 9).

The contents of a decision node, represented by a circle, has the form $(\ell;\ t)$ where ℓ is a normal line whose value in test t requires justification. Branching can occur at decision nodes when choices of an assignment, represented by a square node, exist. A terminal node is represented by a square and contains an integer which is an index for a solution, or an X that denotes an inconsistency and results in backtracking.

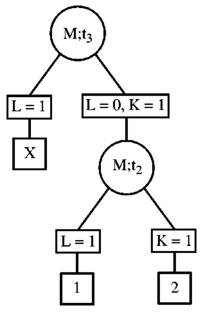


Fig. 9. Decision tree for Example 5.

	A	B	C	D	E	F	G	H	I	J	K	L	M
t_1	0			0			1				0	0	0
t_2	1			1									1
t_3	1			1									1
t_4	1			1			1				1	0	1
t_5	0			0							0	0	0

(a)

	A	B	C	D	E	F	G	H	I	J	K	L	M
t_1	0	1	1	0		1	1	1	0	1	0	0	0
t_2	1	1	0	1		0		0	1	1		1	1
t_3	1	1	1	1		1	1	1	0	1	1	0	1
t_4	1	1	1	1		1	1	1	0	1	1	0	1
t_5	0	1	1	0		1	1	1	0	1	0	0	0
Y =	*n*	1	*n*	*n*		*n*	1	*n*	*n*	1	*n*	*n*	*n*

(b)

	A	B	C	D	E	F	G	H	I	J	K	L	M
t_1	0			0			1				0	0	0
t_2	1			1			1				1		1
t_3	1			1			1				1	0	1
t_4	1			1			1				1	0	1
t_5	0			0							0	0	0
Y =	*n*			*n*			1				*n*	0	*n*

(c)

Fig. 10. Computations associated with Example 5, (a) values deduced prior to first decision point, (b) first solution, (c) second solution.

Assume the response of C^{*} to T is $R^{*} = 01\,110$, as shown in Fig. 10(a). Since line M has both zero and one values it is normal and hence there exist one or more normal paths from PIs to M (Complete Normal Path). For an OR gate, an output of 0 implies all inputs are zero. Thus, for t_1 and t_5, $M = 0$

implies $L = K = 0$. Since L has FV^0 in t_4 and a 0 value has been deduced for L (in t_1 and t_5), then $L = 0$ in t_4. This is an example of a **vertical implication**, i.e., values in one test implying line values in another test. This concept is unique to the deduction algorithm. Knowing $L = 0$ and $M = 1$ in t_4 implies $K = 1$ since M is an OR gate. This is an example of **horizontal implication**. Thus, K is normal! Continuing, $K = 1$ in t_4 implies $D = 1$ and $G = 1$ in t_4, which generate the vertical implications $D = 1$ in t_2 and t_3, and $G = 1$ in t_1. Now $G = 1$ in t_1 implies $D = 0$ in t_1, which implies $D = 0$ in t_5. At this point all the values of D can be assigned to its stem A (P2). No more implications exists.

We next attempt to justify the value of M in t_2 and t_3. We first select t_3 as shown in the decision tree in Fig. 9. We initially try the assignment $L = 1$. Carrying out the resulting implications results in a conflict, i.e., line A is assigned both a zero and a one. Since $K = X$ in this analysis, we can reverse our decision and set $L = 0$ and $K = 1$ as our next decision (see decision tree). Going from the terminal node labeled X and the new decision node is done via backtracking. Now $K = 1$ in t_3 implies $G = 1$ in t_3. Again there are no more implications possible, so a new decision node is created, labeled $(M;\, t_2)$. The two assignments $L = K = 1$ lead to two solutions shown in Fig. 10(b) and (c), respectively.

Note that lines C, F, H, I and L are identified as being normal with respect to the first solution only. The lines A, D, K, and M are normal in both solutions and, therefore, are actually normal (fault-free) lines in C^*. Note also that A, D, K, M designate a path between a PI and a PO. □

The reader is referred to [10] and [12] for details of the Deduction Algorithm.

The second phase of effect–cause analysis deals with determining the states of the lines in C^*. A complete analysis of the mapping of the results generated by the Deduction Algorithm to potential failures in C is again beyond the scope of this paper. A brief overview, however, will be presented.

We represent the status s_ℓ of a line ℓ by zero, one, or n, where zero(one) represents s–a–0(1), and n represents normal. Then a *fault situation* is defined by the vector $F = [s_\ell]$. Recall that C is the fault free circuit. Let C^F denote the circuit C in the presence of fault situation F. Clearly, if $F = [n, n, \ldots, n]$ then $C^F = C$. C^F realizes the Boolean switching function Z^F. If $Z^{F_1} = Z^{F_2}$ then we say that F_1 and F_2 are *indistinguishable*. Let $\mathcal{F}(F_j) = \{F_i | Z^{F_i} = Z^{F_j}\}$. Thus, the fault situations can be partitioned into equivalence classes. The faults in $\mathcal{F}(F = [n, n, \ldots, n])$ are undetectable and, therefore, redundant. Let $Z^F(T)$ be the response of C^F to T. Then we say that F_1 and F_2 are *equivalent under* T iff $Z^{F_1}(T) = Z^{F_2}(T)$. Let $\mathcal{F}^* = \{F | Z^F(T) = R^*\}$. Our goal in fault analysis (phase 2) is to identify one or more members of the set $\mathcal{F}^*$. For each solution generated by the deduction algorithm we derive a *kernel* fault that corresponds (covers) a family of faults that may actually exist in C^*. Let $Y = [y_\ell]$ be a kernel fault, where $y_\ell \in \{0, 1, n, u\}$, and $y_\ell = u$ means that the state of ℓ is unknown. Note that $y_\ell = 0, 1$ and n have the same meaning as $s_\ell = 0, 1$ and n. Y can be constructed as follows: 1) $y_\ell = n$ iff both zero and one values have been deduced for ℓ; 2) $y_\ell = 0(1)$ iff only zero(one) values have been deduced for ℓ, and 3) $y_\ell = u$ iff no values have been deduced for ℓ. Any fault situation F obtained from a Y by replacing us with zero, one, or n satisfy $Z^F(T) = R^*$.

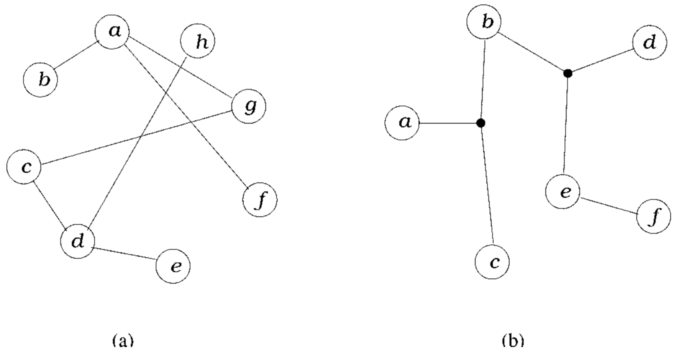


Fig. 11. (a) Graph. (b) Hypergraph.

Referring to our previous example we obtain two solutions, shown in the lower part of Fig. 10(b) and (c). To obtain finer resolution on the fault sites one can apply additional test patterns. Once the subset of lines are identified where faults may exists, specific tests that activate these faults can be constructed.

Finally, probing can be used to access internal lines and hence increase observability [12].

III. Fundamental Algorithms in Physical Design

A. Partitioning

A chip may contain tens of millions of transistors. Layout of the entire circuit cannot be handled in a flat mode due to the limitation of memory space as well as computation power available. Even though fabrication technologies have made great improvements in packing more logic in a smaller area, the complexity of circuits has also been increasing correspondingly. This necessitates partitioning a circuit and distributing it across several regions in a chip or across several chips. Thus, the first step in the physical design phase is partitioning which can significantly influence the circuit performance and layout costs.

Partitioning is a complex problem which is NP-complete. The nature of the partitioning problem along with the size of the circuit makes it difficult to perform an exhaustive search required to find an optimal solution.

To study the partitioning problem clearly, graph notations are commonly used. A graph $G = (V, E)$ consists of a set V of vertices, and a set E of edges. Each edge corresponds to a pair of distinct vertices [see Fig. 11(a)]. A hypergraph $H = (N, L)$ consists of a set N of vertices and a set L of hyperedges, where each hyperedge corresponds to a subset N_i of distinct vertices with $|N_i| \geq 2$ [see Fig. 11(b); e.g., the connection interconnecting vertices a, b, and c is a hyperedge]. We also associate a vertex weight function ω: $V \rightarrow IN$ with every vertex, where IN is the set of integers. Thus, a circuit can be represented by a graph or a hypergraph, where the vertices are circuit elements and the (hyper)edges are wires. The vertex weight may indicate the size of the corresponding circuit element.

A *multiway partition* of a (hyper)graph H is a set of nonempty, disjoint subsets of $N = \{N_1, \ldots, N_r\}$, such that $\bigcup_{i=1}^{r} N_i = N$ and $N_i \cap N_j = \emptyset$ for $i \neq j$. A partition is *acceptable* if $b(i) \leq \omega(N_i) \leq B(i)$, where $\omega(N_i)$ is the sum of the weight of vertices in N_i, $B(i)$ is the maximum size of part i and $b(i)$ is the minimum size of part i, for $i = 1, \ldots, r$; $B(i)$s and $b(i)$s are input parameters. A special case of multiway partitioning problem in which $r = 2$ is called the *bipartition* problem. In the bipartition problem, $B(i)$ is at least α times the sum of the weight of all vertices, for some α, $(1/2) \leq \alpha < 1$. Typically, α is close to 1/2. The number α is called the *balance factor*. The bipartition problem can also be used as a basis for heuristics in multiway partitioning. Normally, the objective is to minimize the number of *cut* edge, that is, the number of hyperedges with at least one vertex in each partition.

Classic iterative approaches known Kernighan–Lin (KL) and Fiduccia–Mattheyses (FM) begin with some initial solution and try to improve it by making small changes, such as swapping modules between clusters. Iterative improvement has become the industry standard for partitioning due to its simplicity and flexibility. Recently, there have been many significant improvements to the basic FM algorithm. Multilevel approaches are very popular and produce superior partitioning results for very large sized circuits. In this section, we discuss the basic FM algorithm and hMetis, a multilevel partitioning algorithm, which is one of the best partitioners.

The KL and FM Algorithms: To date, iterative improvement techniques that make local changes to an initial partition are still the most successful partitioning algorithms in practice. One such algorithm is an iterative bipartitioning algorithm proposed by Kernighan and Lin [13].

Given an unweighted graph $G = (V, E)$, this method starts with an arbitrary partition of G into two groups V_1 and V_2 such that $|V_1| < \alpha \cdot |V|$ and $|V_2| < \alpha \cdot |V|$, where α is the balance factor as defined in the previous subsection and $|V|$ denotes the number of vertices in set V. A *pass* of the algorithm starts as follows. The algorithm determines the vertex pair (v_a, v_b), $v_a \in V_1$ and $v_b \in V_2$, whose exchange results in the largest decrease of the cut-cost or in the smallest increase if no decrease is possible. A cost increase is allowed now in the hope that there will be a cost decrease in subsequent steps. Then the vertices v_a and v_b are locked. This locking prohibits them from taking part in any further exchanges. This process continues, keeping a list of all tentatively exchanged pairs and the decreasing gain (or cut-cost), until all the vertices are locked.

A value k is selected to maximize the partial sum $\sum_{i=1}^{k} g_i = \text{Gain}_k$, where g_i is the gain of the ith exchanged pair. If $\text{Gain}_k > 0$, a reduction in cut-cost can be achieved by moving $\{v_{a1}, \ldots, v_{ak}\}$ to V_2 and $\{v_{b1}, \ldots, v_{bk}\}$ to V_1. This marks the end of one pass. The resulting partition is treated as the initial partition, and the procedure is repeated for the next pass. If there is no k such that $\text{Gain}_k > 0$ the procedure halts. A formal description of KL algorithm is as follows.

```
Procedure: KL heuristic($G$);
  begin-1
    bipartition $G$ into two groups $V_1$ and
    $V_2$, with $|V_1| = |V_2| \pm 1$;
    repeat-2
      for $i = 1$ to $n/2$ do
        begin-3
          find a pair of unlocked vertices
          $v_{a_i} \in V_1$ and $v_{b_i} \in V_2$ whose exchange
            makes the largest decrease or
            smallest increase in cut-cost;
          mark $v_{a_i}$, $v_{b_i}$ as locked and store
            the gain $g_i$;
            end-3
        find $k$, such that $\sum_{i=1}^{k} g_i = Gain_k$
          is maximized;
        if $Gain_k > 0$ then
        move $v_{a_1}, \ldots, v_{a_k}$ from $V_1$ to $V_2$ and
          $v_{b_1}, \ldots, v_{b_k}$ from $V_2$ to $V_1$;
    until-2 $Gain_k \leq 0$;
  end-1.
```

The *for-loop* is executed $O(n)$ times. The body of the loop requires $O(n^2)$ time. Thus, the total running time of the algorithm is $O(n^3)$ for each pass of the repeat loop. The repeat loop usually terminates after several passes, independent of n. Thus, the total running time is $O(cn^3)$, where c is the number of times the repeat loop is executed.

Fiduccia and Mattheyses [14] improved the Kernighan-Lin algorithm by reducing the time complexity per pass to $O(t)$, where t is the number of hyper-edge ends (or, *terminals*) in G. FM added the following new elements to the KL algorithm:

1) only a single vertex is moved across the cut in a single move;
2) adding weights to vertices;
3) a special data structure for selecting vertices to be moved across the cut to improve running time (this is the main feature of the algorithm).

We shall first discuss the data structure used for choosing the (next) vertex to be moved. Let the two partitions be A and B. The data structure consists of two pointer arrays, $list\,A$ and $list\,B$ indexed by the set $[-d_{\max} \cdot w_{\max}, d_{\max} \cdot w_{\max}]$ [see Fig. 12]. Here, $d_{\max}$ is the maximum vertex degree in the hypergraph, and $w_{\max}$ is the maximum cost of a hyperedge. Moving one vertex from one set to the other will change the cost by at most $d_{\max} \cdot w_{\max}$. Indexes of the list correspond to possible (positive or negative) gains. All vertices resulting in gain g are stored in the entry g of the list. Each pointer in the array $list\,A$ points to a linear list of unlocked vertices inside A with the corresponding gain. An analogous statement holds for $list\,B$.

Since each vertex is weighted, we have to define a maximum vertex weight W such that we can maintain the *balanced partition* during the process. W must satisfy $W \geq w(V)/2 + \max_{v \in V}\{w(v)\}$, where $w(v)$ is the weight of vertex v. A *balanced partition* is one with either side of the partition having a total vertex weight of at most W, that is, $w(A), w(B) \leq W$. A balanced partition can be obtained by sorting the vertex weights in decreasing order, and placing them in A and B alternately.

This algorithm starts with a balanced partition A, B of G. Note that a move of a vertex across the cut is allowable if such a move does not violate the balance condition. To choose the next vertex to be moved, we consider the maximum gain vertex $a_{\max}$ in $list\,A$ or the maximum gain vertex $b_{\max}$ in $list\,B$, and move

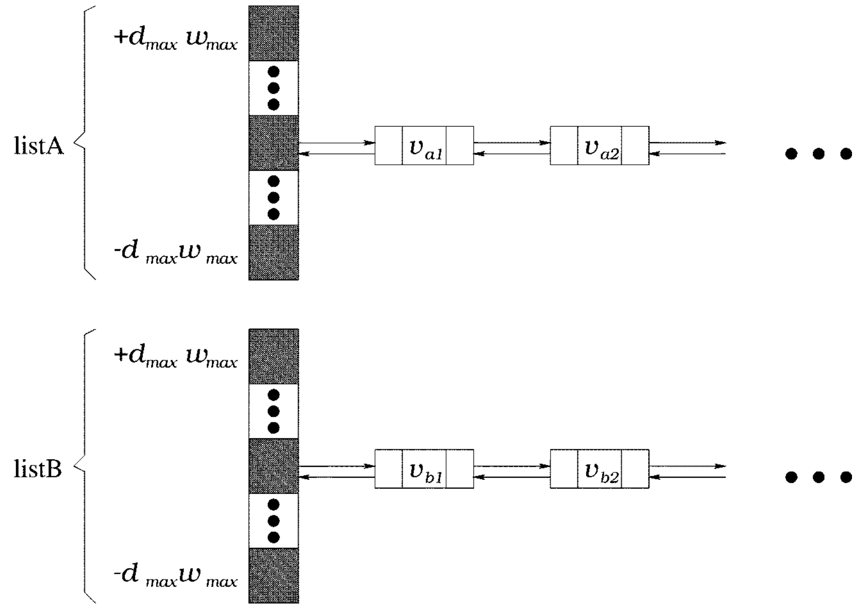


Fig. 12. The data structure for choosing vertices in FM algorithm.

them across the cut if the balance condition is not violated. As in the KL algorithm, the moves are tentative and are followed by locking the moved vertex. A move may increase the cut-cost. When no moves are possible or if there are no more unlocked vertices, choose the sequence of moves such that the cut-cost is minimized. Otherwise the pass is ended.

Further improvement was proposed by Krishnamurthy [15]. He introduced a *look-ahead* ability to the algorithm. Thus, the best candidate among such vertices can be selected with respect to the gains they make possible in later moves.

In general, the obtained bipartition from KL or FM algorithm is a local optimum rather than a global optimum. The performance of KL–FM algorithm degrades severely as the size of circuits grows. However, better partitioning results can be obtained by using clustering techniques and/or better initial partitions together with KL-FM algorithm. The KL–FM algorithm (and its variations) are still the industry standard partitioning algorithm due to its flexibility and the ability of handling very large circuits.

hMetis—A Multilevel Partitioning Algorithm: Two-level partitioning approaches consist of two phases. In the first phase, the hypergraph is coarsened to form a small hypergraph, and then the FM algorithm is used to bisect the small hypergraph. In the second phase, they use the bisection of this contracted hypergraph to obtain a bisection of the original hypergraph. Since FM refinement is done only on the small coarse hypergraph, this step is usually fast. However, the overall performance of such a scheme depends on the quality of the coarsening method. In many schemes, the projected partition is further improved using the FM refinement scheme.

Multilevel partitioning approaches are developed to cope with large sized circuits [16]–[18]. In these approaches, a sequence of successively smaller (coarser) graph is constructed. A bisection of the smallest graph is computed. This bisection is now successively projected to the next level finer graph, and at each level an iterative refinement algorithm such as KL–FM is used to further improve the bisection. The various phases of multilevel bisection are illustrated in Fig. 13. During the coarsening phase, the size of the graph is successively decreased; during the initial partitioning phase, a bisection of the smaller graph is computed; and during the uncoarsening and refinement phase, the bisection is successively refined as it is projected to the larger graphs. During the uncoarsening and refinement phase the dashed lines indicate projected partitionings, and dark solid indicate partitionings that were produced after refinement. G_0 is the given graph, which is the finest graph. G_{i+1} is next level coarser graph of G_i, vice versa, G_i is next level finer graph of G_{i+1}. G_4 is the coarsest graph.

The KL–FM algorithm becomes a quite powerful iterative refinement scheme in this multilevel context for the following reason. First, movement of a single vertex across partition boundary in a coarse graph can lead to movement of a large number of related vertices in the original graph. Second, the refined partitioning projected to the next level serves as an excellent initial partitioning for the KL–FM refinement algorithms. Karypis and Kumar extensively studied this paradigm in [19] and [20]. They presented new graph coarsening schemes for which even a good bisection of the coarsest graph is a pretty good bisection of the original graph. This makes the overall multilevel paradigm even more robust. Furthermore, it allows the use of simplified variants of KL–FM refinement schemes during the uncoarsening phase, which significantly speeds up the refinement without compromising the overall quality.

Metis [20], a multilevel graph partitioning algorithm based on this work, routinely finds substantially better bisections and is very fast. In [21] and [22], Karypis and Kumar extended their graph partitioning algorithm to hypergraph and developed hMetis. When comparing different partitioning tools on large-sized circuits, Alpert [23] found that hMetis performs the best.

The notion of *ratio cut* was presented to solve the partitioning problem more naturally. The ratio cut approach can be described as follows: Given a graph $G = (V, E)$, (V_1, V_2) denotes a cut

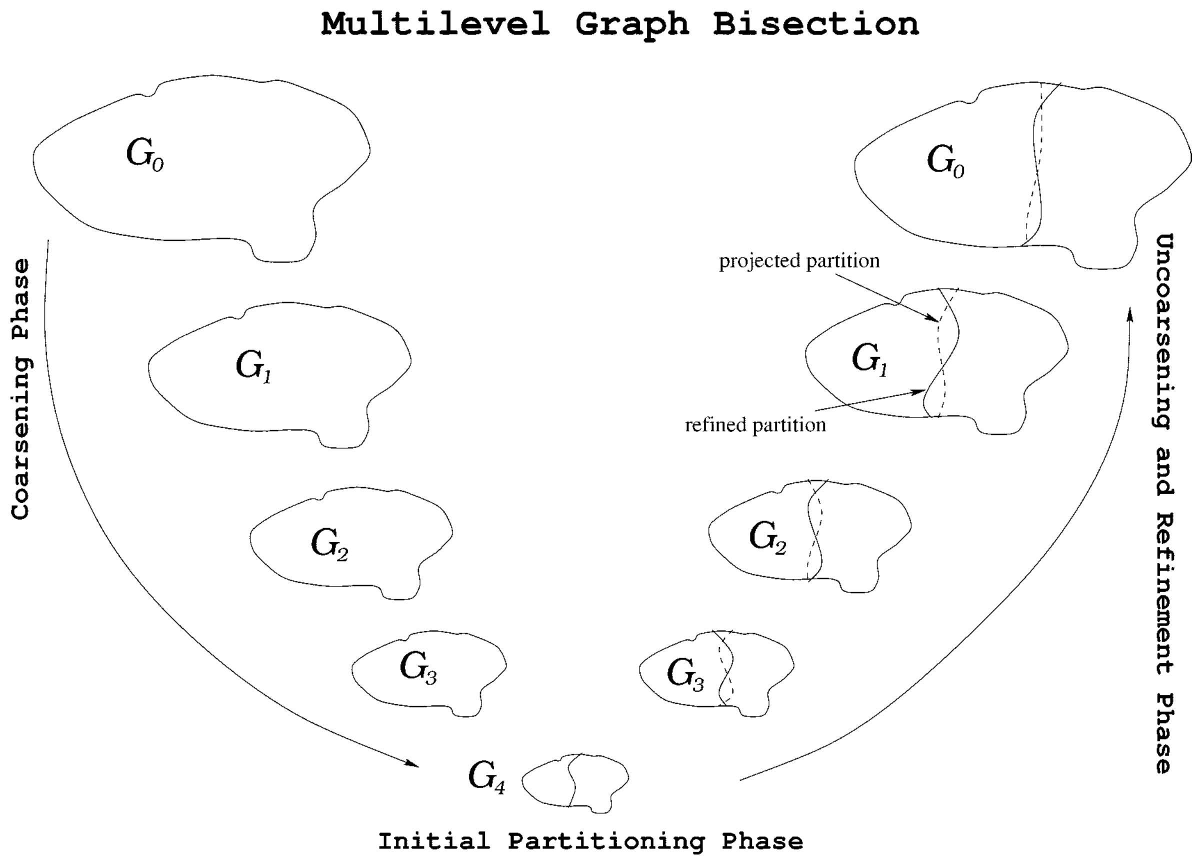

Multilevel Graph Bisection
G_0
G_1
G_2
G_3
G_4
Coarsening Phase
Initial Partitioning Phase
Uncoarsening and Refinement Phase
projected partition
refined partition

Fig. 13. The various phases of the multilevel graph bisection.

that separates a set of nodes V_1 from its complement $V_2 = V - V_1$. Let c_{ij} be the cost of an edge connecting node i and node j. The *cut-cost* is equal to $C_{V_1V_2} = \sum_{i \in V_1} \sum_{j \in V_2} c_{ij}$. The *ratio* of this cut is defined as $R_{V_1V_2} = C_{V_1V_2}/(|V_1| * |V_2|)$, where $|V_1|$ and $|V_2|$ denote the size of subsets V_1 and V_2, respectively. The objective is to find a cut that generates the minimum ratio among all cuts in the graph. By using the ratio cut as the cost function, most iterative partitioning heuristics including KL–FM can be modified to handle the ratio-cut problem (e.g., see [24]).

The following extensions and variations of the partitioning problem are of particular interest.

- By adding different constraints, the partitioning problem is useful in various areas. However, how to effectively handle more constrainted partitioning problems is still an open issue.
- Bisection has been studied extensively for over three decades. How to efficiently solve a multiway partitioning problem is yet an open problem.
- How to handle cost functions other than cut, e.g., wirelength and congestion.

B. Floorplanning

In the floorplanning phase, the macro cells have to be positioned on the layout surface in such a manner that no blocks overlap and that there is enough space left to complete the interconnections. The input for the floorplanning is a set of modules, a list of terminals (pins for interconnections) for each module and a netlist, that describes the terminals which have to be connected. At this stage, good estimates for the area of the single macro cells are available, but their exact dimensions can still vary within a wide range. Consider for example a register file module consisting of 64 registers. These alternatives are described by shape-functions. A shape-function is a list of feasible height/width-combinations for the layout of a single macro cell. The result of the floorplanning phase is the sized floorplan, which describes the position of the cells in the layout and the chosen implementations for the flexible cells.

In this stage, the relative positions of the modules to be laid out are determined. Timing, power, and area estimations are the factors guiding the relative placement. Floorplanning can be used to verify the feasibility of integrating a design onto a chip without performing the detailed layout and design of all the blocks and functions. If control logic is implemented with standard cells, then the number of rows used for the modules is not necessarily fixed. Many rows will produce a block that is long and skinny; few rows will produce a block that is short and wide. As other examples, folding and partitioning of a PLA can be used to modify the aspect ratio of the module, or the number of bits used for row and column decoding in a RAM or ROM module can also modify their aspect ratio.

Automatic floorplanning becomes more important as automatic module generators become available which can accept as constraints or parts of the cost functions, pin positions, and aspect ratios of the blocks. Typically, floorplanning consists of the following two steps. First, the topology, i.e., the relative positions of the modules, is determined. At this point, the chip is viewed as a rectangle and the modules are the (basic) rectangles whose relative positions are fixed. Next, we consider the *area optimization problem*, i.e., we determine a set of implementations (one for each module) such that the total area of the chip is minimized. The topology of a floorplan is obtained by recursively using circuit partitioning techniques. A *partition* divides a given circuit into k parts such that: 1) the sizes of the k parts are as close as possible and 2) the number of nets connecting the k parts is minimized. If $k = 2$, a recursive bipartition generates a *slicing floorplan*. A floorplan is slicing if it is either a basic rectangle or there is a line segment (called slice) that partitions the enclosing rectangle into two slicing floorplans. A slicing floorplan can be represented by a *slicing tree*. Each leaf node of the slicing tree corresponds to a basic rectangle and each nonleaf node of the slicing tree corresponds to a slice.

There exist many different approaches to the floorplanning problem. Wimer *et al.* [25] described a branch-and-bound approach for the floorplan sizing problem, i.e., finding an optimal combination of all possible layout-alternatives for all modules after placement. While their algorithm is able to find the best solution for this problem, it is very time consuming, especially for real problem instances. Cohoon *et al.* [26] implemented a genetic algorithm for the whole floorplanning problem. Their algorithm makes use of estimates for the required routing space to ensure completion of the interconnections. Another widely used heuristic solution method is simulated annealing [27], [28].

When the area of the floorplan is considered, the problem of choosing for each module the implementation which optimizes a given evaluation function is referred to as the *floorplan area optimization problem* [29].

A floorplan consists of an enveloping rectangle partitioned into nonoverlapping basic rectangles (or modules). For every basic rectangle a set of implementations is given, which have a rectangular shape characterized by a width w and a height h. The relative positions of the basic rectangles are specified by the *floorplan tree*: the leaves are the basic rectangles,the root is the enveloping rectangle, and the internal nodes are the *composite rectangles*. Each of the composite rectangles is divided into k parts in a *hierarchical floorplan* of order k: if $k = 2$(slicing floorplan), a vertical or horizontal line is used to partition the rectangle; if $k = 5$, a right or left *wheel* is obtained. The general case of composite blocks which cannot be partitioned in two or five rectangles can be dealt with by allowing them to be composed of L-shaped blocks. Once the *implementation* for each block has been chosen, the size of the composite rectangles can be determined by traversing through upwards the floorplan tree; when the root is reached, the area of the enveloping rectangle can be computed. The goal of the floorplan area optimization problem is to find the implementation for each basic rectangle such that the minimum area enveloping rectangle is obtained. The problem has been proven to be NP-complete in the general case, although it can be reduced to a problem solvable in polynomial time in the case of slicing floorplans.

Since floorplanning is done very early in the design process, only estimates of the area requirements for each module are given. Recently, the introduction of simulated annealing algorithms has made it possible to develop algorithms where the optimization can be carried out with all the degrees of freedom mentioned above. A system developed at the IBM T.J. Watson Research Center and use the simulated annealing algorithm to produce a floorplan that not only gives the relative positions of the modules, but also aspect ratios and pin positions.

Simulated Annealing: Simulated annealing is a technique to solve general optimization problems, floorplanning problems being among them. This technique is especially useful when the solution space of the problem is not well understood. The idea originated from observing crystal formation of materials. As a material is heated, the molecules move around in a random motion. When the temperature slowly decreases, the molecules move less and eventually form crystalline structures. When cooling is done in a slower manner, more crystal is at a minimum energy state, and the material forms into a large crystal lattice. If the crystal structure obtained is not acceptable, it may be necessary to reheat the material and cool it at a slower rate.

Simulated annealing examines the *configurations* of the problem in sequence. Each configuration is actually a feasible solution of the optimization problem. The algorithm moves from one solution to another, and a global cost function is used to evaluate the desirability of a solution. Conceptually, we can define a *configuration graph* where each vertex corresponds to a feasible solution, and a directed edge (v_i, v_j) represents a possible movement from solution v_i to v_j.

The annealing process moves from one vertex (feasible solution) to another vertex following the directed edges of the configuration graph. The random motion of the molecules at high temperature is simulated by randomly accepting moves during the initial phases of the algorithm. As the algorithm proceeds, temperature decreases and it accepts less random movements. Regardless of the temperature, the algorithm will accept a move (v_i, v_j) if $cost(v_j) \leq cost(v_i)$. When a local minimum is reached, all "small" moves lead to a higher cost solution. To avoid being trapped in a local minimum, simulated annealing accepts a movement to higher cost when the temperature is high. As the algorithm cools down, such movement is less likely to be accepted. The best cost among all solutions visited by the process is recorded. When the algorithm terminates, hopefully it has examined enough solutions to achieve a low cost solution. Typically, the number of feasible solutions is an exponential function of the problem size. Thus, the movement from one solution to another is restricted to a very small fraction of the total configurations.

A pseudocode for simulated annealing is as follows:

```
Algorithm Simulated annealing
Input: An optimization problem.
Output: A solution $s$ with low cost.
begin-1
 $s :=$ random initialization.
 $T := T_0$. /* initial temperature */
 while $not\, frozen(T)$ do
  begin-2
  $count := 0$.
  while $not\, equilibrium(count, s, T)$ do
   begin-3
   $count := count + 1$.
```

```
            $nexts := generate(s)$.
            if ( $cost(nexts) < cost(s)$) or
              ( $f(cost(s), cost(nexts), T) > random(0, 1)$)
              then $s := nexts$.
          end-3.
          update($T$).
      end-2.
  end-1.
```

$generate()$ is a function that selects the next solution from the current solution s following an edge of the configuration graph. $cost()$ is a function that evaluates the global cost of a solution. $f()$ is a function that returns a value between zero and one to indicate the desirability to accept the next solution, and $random()$ returns a random number between zero and one. A possible candidate function f is the well-known Boltzmann probability function $e^{\Delta C/(k_B T)}$, where ΔC is the cost change (i.e., $\Delta C = cost(s) - cost(nexts)$) and k_B is the Boltzmann constant. The combined effect of $f()$ and $random()$ is to have high probability of accepting a high-cost movement at high temperature. $equilibrium()$ is used to decide the termination condition of the random movement, and $update()$ reduces the temperature to cool down the algorithm. $frozen()$ determines the termination condition of the algorithm. The algorithm is usually frozen after an allotted amount of computation time has been consumed; a sufficiently good solution has been reached, or the solutions show no improvement over many iterations.

A solution of the floorplanning problem can be represented by a floorplan tree. The cost of the solution can be computed via this tree representation. We can use the simulated annealing technique to find a good floorplanning solution which corresponds to a low cost.

It is quite often that certain macro cells need to be pre-placed. Areas occupied by these cells become blockages for floorplanning. This adds complexities to the original floorplanning algorithm. Simulated annealing based approaches can handle this problem with modifications.

Simulated annealing has been very successful in floorplanning. As the design and module library grow in size, the performance of simulated annealing degrades drastically. The open question is: can we find a more effective heuristic than simulated annealing to solve the floorplanning problem?

C. Placement

The placement problem can be defined as follows. Given an electrical circuit consisting of modules with predefined input and output terminals and interconnected in a predefined way, construct a layout indicating the positions of the modules cells such that some performance measures such as estimated wire length and/or layout area are minimized. The inputs to the problem are the module description, consisting of the shapes, sizes, and terminal locations, and the netlist, describing the interconnections between the terminals of the modules. The output is a list of x- and y-coordinates for all modules. We need to optimize chip area usage in order to fit more functionality into a given chip. We need to minimize wirelength to reduce the capacitive delays associated with longer nets, speed up the operation of the chip, and reduce area. These goals are closely related to each other for standard cell and gate array design styles, since the total chip area is approximately equal to the area of the modules plus the area occupied by the interconnect. Hence, minimizing the wire length is approximately equivalent to minimizing the chip area. In the macro design style, the irregularly sized macros do not always fit together, and some space is wasted. This plays a major role in determining the total chip area, and we have a tradeoff between minimizing area and minimizing the wire length. In some cases, secondary performance measures may also be needed, such as the preferential minimization of wire length of *critical* nets, at the cost of an increase in total wire length. Module placement is an NP-complete problem and, therefore, cannot be solved exactly in polynomial time [Donath 1980]. Trying to get an exact solution by evaluating every possible placement to determine the best one would take time proportional to the factorial of the number of modules. This method is, therefore, impractical for circuits with any reasonable number of modules. To efficiently search through a large number of candidate placement configurations, a heuristic algorithm must be used. The quality of the placement obtained depends on the heuristic used. At best, we can hope to find a good placement with wire length quite close to the minimum, with no guarantee of achieving the absolute minimum.

Placement algorithms are typically divided into two major classes: constructive placement and iterative improvement. In constructive placement, a method is used to build up a placement from scratch; in iterative improvement, algorithms start with an initial placement and repeatedly modify it in search of a cost reduction. If a modification results in a reduction in cost, the modification is accepted; otherwise it is rejected. Constructive placement algorithms are generally very fast, but typically result in poor layouts. Since they take a negligible amount of computation time compared to iterative improvement algorithms, they are usually used to generate an initial placement for iterative improvement algorithms. More recent constructive placement algorithms, such as numerical optimization techniques [30], [31], integer programming formulation [32], and simulated annealing-based methods [33] yield better layouts but require significantly more CPU time. One of the biggest challenge for placement tools is the rapid growth in circuit size. A good placement algorithm has to be more effective than ever in finding a good layout as quickly as possible.

Quadratic Algorithm: One of the objectives of the placement problem is to reduce the total wirelength. Assume cells i and j are connected by a net e. The physical location of cell i and j is at (x_i, y_i) and (x_j, y_j), respectively. The linear wirelength of e is $L_e = |x_i - x_j| + |y_i - y_j|$, and the quadratic wirelength of e is $Q_e = (x_i - x_j)^2 + (y_i - y_j)^2$. The total linear wirelength of a layout is $WL = \sum_e L_e$ and the total quadratic wirelength is $WL^2 = \sum_e Q_e$. When a net is connecting more than two cells (a multipin net), we can replace this net with a number of two-pin nets. A typical method is to use a clique of two-pin nets to replace the original multipin net. Each two-pin net in the clique get a weight of $2/p$ if the original multipin net has p cells incident to it.

Linear wirelength is widely used because it correlates well with the final layout area after routing. Quadratic wirelength is an alternative to use in placement. Experimental results show that the quadratic wirelength objective over-penalizes long wires and has a worse correlation with the final chip area [34]. However, since the quadratic wirelength objective is analytical, numerical methods can be used to solve for an optimal solution. The quadratic algorithm uses quadratic wirelength objective and analytical methods to generate a layout. It is very fast and leads to relatively good results.

Matrix representations and linear algebra are often used in quadratic placement algorithms. Assume the given circuit is mapped into a graph $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots, e_m\}$. A nonnegative weight $w(e)$ is assigned to each edge $e \in E$. All the nets can be represented using an *adjacency matrix* $A = (a_{ij})$ which has an entry $a_{ij} = w(v_i, v_j)$ if $(v_i, v_j) \in E$ and $a_{ij} = 0$, otherwise. The physical locations of all the vertices can be represented by n-dimensional vectors $X = (x_i)$ and $Y = (y_i)$, where (x_i, y_i) is the coordinates of vertex v_i.

The x and y directions are independent in quadratic wirelength objective. We can optimize the objective separately in each direction. The following discussions will be focused on optimizing the quadratic objective in x direction. The same method can be used symmetrically in the y direction.

The total quadratic wirelength in x direction of a given layout can be written as

$$\Phi_Q(X) = \tfrac{1}{2} X^T Q X + d^T X = \sum_{i,j=1}^{n} a_{ij} (x_i - x_j)^2 + d^T X. \tag{1}$$

$Q = (q_{ij})$ is a $n \times n$ *Laplacian matrix* of A. Q has entry q_{ij} equal to $-a_{ij}$ if $i \neq j$, and q_{ij} equal to $\sum_{j=1}^{n} a_{ij}$ otherwise, i.e., q_{ij} is the degree of vertex v_i. The optional linear term $d^T X$ represents connections of cells to fixed I/O pads. The vector d can also capture pin offsets. The objective function (1) is minimized by solving the linear system

$$QX + d = 0. \tag{2}$$

The solution of this system of linear equations usually is not a desired layout because vertices are not evenly distributed. This results in a lot of cell overlaps which are not allowed in placement. A balanced vertex distribution in the placement area needs to be enforced. This can be achieved by either re-assigning vertex locations after solving (2) or adding more constraints to (1).

Existence of fixed vertices is essential for quadratic algorithms. Initially, I/O pads are fixed vertices. If no I/O pads are present, a trivial solution of (1) will be having all the vertices located at the same place. Existence of I/O pads forces vertices to separate to some extent. We can further spread the vertices to achieve a balanced distribution based on the solution of (2). This spreading procedure is based on heuristics and is not optimal. Iterative approaches can be used to further improve the layout. In the next iteration, a number of vertices can be fixed. A new layout can be obtained by solving a new equation similar to (2). We can increase the number of fixed vertices gradually or wait until this approach converges.

Constraints can also be added to (1) to help balance the vertex distribution. Researchers have added spatial constraints so that the average location of different groups of vertices will be evenly distributed in the placement area [30], [34], [35]. They recursively reduce the number of vertices in vertex groups by dividing them into smaller groups. Eisenmann *et al.* [36] added virtual spatial nets to (2). A virtual spatial net will have a negative weight if two cells incident to it are close to each other. As two cells get closer, the absolute value of their spatial net weight gets larger. The virtual spatial nets between cells tends to push overlapped cells further away from each other due to the negative weights. The weights on virtual spatial nets are updated each iteration until the solution converges.

The layouts produced by quadratic algorithm are not the best compared to layouts produced by other placement algorithms. The reason why quadratic algorithm is still attractive and widely used in industry is because of its fast speed. It can represent interactions between locations of cells and connections between cells using one simple linear equation (1). However, it needs heuristics to balance the cell distribution in the placement area. The effectiveness of these heuristics will highly affect the quality of the final layout.

Routing/congestion-driven placement aims to reduce the wiring congestion in the layout to ensure that the placement can be routed using the given routing resources. Congestion can be viewed using a supply-demand model [37]. Congested regions are where the routing demand exceeds the routing resource supply. Wang and Sarrafzadeh [38] pointed out that the congestion is globally consistent with the wirelength. Thus, the traditional wirelength placement can still be used to effectively reduce the congestion globally. However, in order to eliminate local congested spots in the layout, congestion-driven approaches are needed.

D. Routing

Routing is where interconnection paths are identified. Due to the complexity, this step is broken into two stages: global and detailed routing. In global routing, the “loose” routes for the nets are determined. For the computation of the global routing, the routing space is represented as a graph, the edges of this graph represent the routing regions and are weighted with the corresponding capacities. Global routing is described by a list of routing regions for each net of the circuit, with none of the capacities of any routing region being exceeded.

After global routing is done, for each routing region the number of nets routed is known. In the detailed routing phase, the exact physical routes for the wires inside routing regions have to be determined. This is done incrementally, i.e., one channel is routed at a time in a predefined order.

The problem of global routing is very much like a traffic problem. The pins are the origins and destinations of traffic. The wires connecting the pins are the traffic, and the channels are the streets. If there are more wires than the number of tracks in a given channel, some of the wires have to be rerouted just like the rerouting of traffic. For the real traffic problem, every driver wants to go to his destination in the quickest way, and he may try

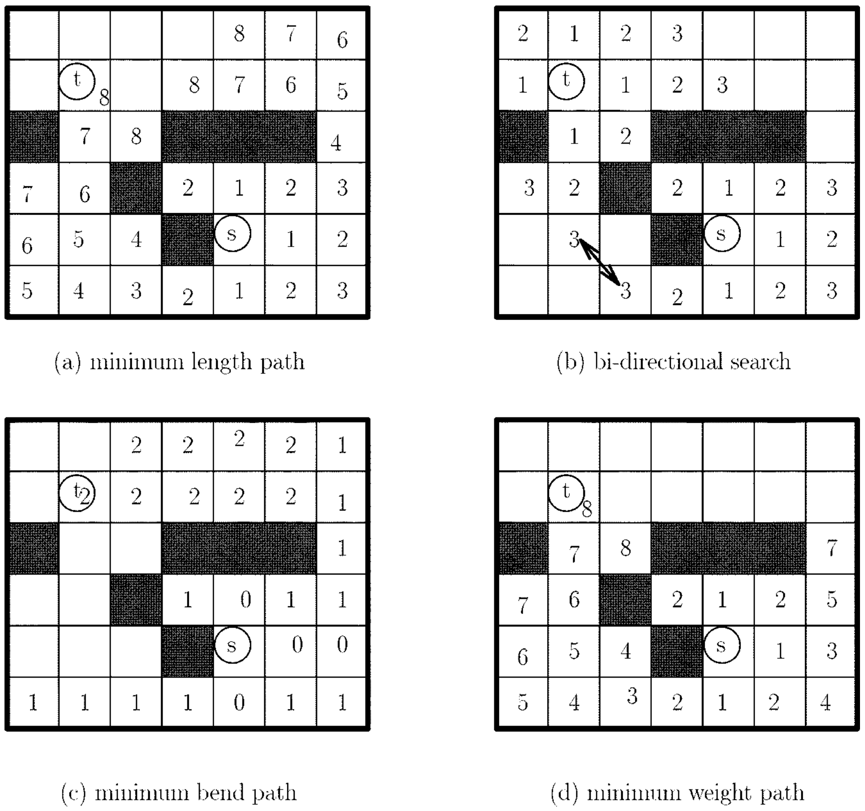


Fig. 14. An example demonstrating Lee's algorithm from source s to sink t. (a) Minimum length path. (b) Bidirectional search. (c) Minimum bend path. (d) Minimum weight path.

a different route every day. Finally every driver selects the best route possible for him and the traffic pattern is stabilized. Intuitively, we can do the same for the routing problem. In global routing, the usual approach was to route one net at a time sequentially until all nets are connected. To connect one net, we could use the maze running algorithm, simulated annealing or other algorithms. Maze running is the standard in industry.

Maze Running: Maze running was studied in the 1960s in connection with the problem of finding a shortest path in a geometric domain. A classical algorithm for finding a shortest path between two terminals (or, points) was proposed in [39]. The idea is to start from one of the terminals, called the source terminal. Then, *label* all grid points adjacent to the source as 1. The label of a point indicates its distance to the source. Any unlabeled grid point p that is adjacent to a grid point with label i is assigned label $i+1$. We assign all label is before assigning any label $i+1$. Note that two points are called adjacent only if they are either horizontally or vertically adjacent; diagonally adjacent points are not considered adjacent, for in routing problems we deal with rectilinear distances. The task is repeated until the other terminal of the net is reached. See Fig. 14(a). We need to backtrack from the target to the source to find the underlying path. This procedure is called Maze-Running Algorithm.

Several extensions of maze running have been studied as discussed below. The reason for extending the maze running techniques, as opposed to inventing new tools, is that the approach is simple and easy to understand and predict its performance.

Optimization of Memory Usage: A major drawback of maze running approaches is the huge amount of memory used to label the grid points in the process. Attempts have been made to circumvent this difficulty. One solution is to use an encoding scheme where a grid just points to neighbors instead of storing the actual distance. Therefore, at each grid point we need to store $O(1)$ bits instead of $O(\log n)$ bits, where n is the number of grid points. There are other effective memory optimization schemes that also speed up the process; indeed, they are primarily for speeding up the process. One of the most effective approaches is based on bidirectional search as discussed below.

Minimum-Cost Paths: In our previous discussion, we have tried to minimize the length of the path between the source and the sink, that is, minimization of the distance in L_1 metric. However, we might also be interested in other objectives.

If we want to minimize the number of bends in the path we proceed as follows. All grid points that are reachable with zero bends from the source are labeled with zero. Note that these grid points are either adjacent to the source or are adjacent to a grid point with label zero. All grid points that are reachable from a grid point with label zero with one bend are labeled with one. In general, in stage i, all grid points that are reachable from a grid point with label $i-1$ with one bend are labeled with i. Note that for each grid point with label i, we also need to store the direction of the path (if there are more than one path, all directions—at most four—need to be stored) that connects the source to that grid point with i bends. An example is shown in Fig. 14(c).

We may also be interested in obtaining a minimum cost path, where the cost of a path is defined by the user. For example, with reference to Fig. 14(d), we may be interested in finding a path that minimizes the use of the right boundary. To accomplish this task, every connection that uses the right boundary is assigned a cost of two. Then, we proceed with the traditional maze running algorithm. Except for grid points that are on the right boundary, we skip assigning labels every other time. That is, if the grid point is adjacent to a grid point with label i, we do not assign a label to it until stage $i+2$.

Multilayer Routing: Multilayer routing can be achieved with the classical maze running algorithm. The main difference is that the maze is now a three-dimensional maze (or a three-dimensional grid). The labeling proceeds as before. If we want to minimize the number of layer changes, we can assign a higher cost to traversing the grid in the third dimension.

Multiterminal Routing: This procedure involves first interconnecting two terminals, as before. Then, start from a third terminal and label the points until the path between the first two terminals is reached. This task is repeated for all unconnected terminals.

Routing is the last stage in physical design. All the parameters of the design (e.g., layout area, power consumption, timing delay, etc.) can be accurately measured after routing. While the general algorithm for routing remains simple and straight forward, a large number of detailed issues need to be considered.

E. Clock Tree Synthesis

For a circuit to function correctly, clock pulses must arrive nearly simultaneously at the clock pins of all clocked components. Performance of a digital system is measured by its cycle time. Shorter cycle time means higher performance. At the layout model, performance of a system is affected by two factors, namely signal propagation time and clock skew. Clock tree synthesis is of fundamental importance. A number of algorithms have been proposed. In particular, the hierarchical recursive matching tree of Kahng, Cong, and Robins [40] and the deferred-merge embedding approach (DME) of [41] are commonly used. However, due to lack of space, we shall omit description of these algorithms.

IV. Fundamental Algorithms in Logic Synthesis and Formal Verification

The last two decades have seen the transition of logic synthesis from a mainly academic pursuit to an accepted design method. On the one hand, research has closed the gap between the quality of circuits designed by CAD tools, and by experienced human designers. On the other hand, the large improvement in productivity afforded by logic synthesis has made the recourse to it almost unavoidable. Application specific intgrated circuit (IC) [ASIC] design, by its nature, has been the first to be deeply influenced by logic synthesis. Improvements in the handling of timing [42] and power consumption [43] have led to wider acceptance. In this paper, we concentrate on the algorithms that have defined logic synthesis in the early eighties, and that still inform, albeit through many extensions and transformations, today's synthesis tools.

Formal methods try to address the limitations of traditional simulation-based verification techniques. Only in recent times has formal verification begun gaining acceptance, in the form of combinational equivalence checking [44], [45], and model checking [46]. The most widely used specification mechanisms are temporal logics and automata. By presenting model checking for CTL* [47], we essentially cover both these approaches. The recent success of model checking owes a great deal to binary decision diagrams (BDDs) [48]. These graphical representations of logic functions have had a profound impact not only on formal verification, but also on synthesis and test generation. Hence, we present a brief overview of their properties and discuss in particular symbolic model checking [49].

A. Two-Level Minimization

A logic expression is formed from a set of variables ranging over $\{0, 1\}$ by applying negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$). A literal is either a variable or its negation. A *term* is a conjunction of literals from distinct variables. An expression in *disjunctive normal form* (DNF) (also called a *sum of products*) is the disjunction of a set of terms. A two-level expression for a logic function is either a DNF expression or a *conjunctive normal form* (CNF) expression. The definition *conjunctive normal form* is dual to the definition of disjunctive normal form. We can, therefore, concentrate on the minimization of DNF expressions without loss of generality. A term such that the function is true whenever the term is true is an *implicant* of the function. An implicant that contains one literal for each variable is a *minterm* of the function. An implicant of a function that does not imply any other implicant is a *prime implicant*.

Given a logic function, we are interested in finding a DNF expression that represents it and is of minimum cost. The cost of the DNF is a function of the number of terms, and the total number of literals. To simplify the discussion, we shall assume that our first priority is to minimize the number of terms. Therefore, Quine's theorem [50] guarantees that the exact solution to the minimization problem is obtained by computing the set of the *prime implicants* of the function and then selecting a subset of minimum cost that covers the function [51].

It is often the case that the function for which a minimum cost DNF is sought is not completely specified. In its simplest form, incomplete specification is represented by a set of minterms for which the value of the function does not matter. The minimization algorithm may choose whatever values help reduce the cost of the solution. Accounting for this form of incomplete specification does not make the minimization problem more difficult. It is sufficient to consider the don't care minterms as part of the function when generating the prime implicants, and ignore them when setting up the covering problem.

Two-level minimization is an NP-hard problem [52], and amounts to solving a covering problem. In spite of recent advances in the generation of the prime implicants [53], and in the branch and bound algorithms [54], computing a DNF of minimum cost remains prohibitive for functions with many minterms and implicants. Heuristic algorithms are, therefore, in common use. A heuristic minimizer like *Espresso* [55], [56], is given a DNF expression for the function of interest; it tries to iteratively improve the DNF until some criterion is met. This may be the exhaustion of allotted computing resources, or in the case we shall consider, the achievement of a solution that is difficult to improve. We shall assume that the solution is a *prime and irredundant* expression. That is, each term is a prime implicant, and no term can be dropped from the DNF without altering the represented function.

The iterative improvement of the solution consists of elementary moves that change the DNF without changing the function. These elementary moves *expand*, *reduce*, or *discard* terms. Expansion of a term removes one or more literals from it, so that it expands to cover more minterms. For instance, the second term of the left-hand side of the following equality can be expanded to yield the right-hand side:

$$x_1 \vee (\neg x_1 \wedge x_2) = x_1 \vee x_2.$$

Expansion is possible in this case because the minterm $x_1 \wedge x_2$ that is added to $\neg x_1 \wedge x_2$ is already covered by the term x_1. Reduction is the inverse of expansion. In our example, reduction would produce the left-hand side from the right-hand side. Finally, a term can be dropped if all minterms it covers are covered by at least another term of the DNF, as in the following example:

$$(x_1 \wedge x_2) \vee (\neg x_2 \wedge x_3) \vee (x_1 \wedge x_3) = (x_1 \wedge x_2) \vee (\neg x_2 \wedge x_3).$$

Of the three types of move, expansion and discard of terms decrease the cost of the DNF. Reduction, on the other hand, increases the cost by adding literals. The first two types of moves are sufficient to produce a prime and irredundant DNF, but reduction is important to allow the minimizer to escape local minima. *Espresso*, therefore, organizes the three operations in an optimization loop that visits one new local minimum solution at each iteration and stops when no improvement is achieved in the course of one iteration. At the beginning of each iteration, all terms are maximally reduced. Then all terms are maximally expanded. In the end, as many redundant terms as possible are discarded.

The maximal reduction of a term t of a DNF f can be computed recursively. The algorithm is based on Boole's expansion theorem

$$f = (x \wedge f_x) \vee (\neg x \wedge f_{\neg x}) \tag{3}$$

where f_x, the *positive cofactor* of f with respect to x is obtained by assigning 1 to x in f. The *negative cofactor* of f, $f_{\neg x}$, is similarly defined. Let f^- be the DNF obtained by removing t from f. The maximal reduction of t is the smallest (least number of minterms) term u that covers all the minterms in t not in f^-. This is computed as $u = t \wedge \rho(f_t^-)$, where $\rho(f_t^-)$ is the smallest term implied by the complement of f_t^-. (Cofactoring with respect to a term t means cofactoring with respect to all literals in t.) If we let $\sigma(h)$ be the smallest term implied by a DNF expression h, we can write

$$\rho(g) = \sigma(x \wedge \rho(g_x) \vee \neg x \wedge \rho(g_{\neg x})). \tag{4}$$

The computation of $\sigma(h)$ is simple: A literal appears in $\sigma(h)$ iff (if and only if) it appears in all terms of h. Equation (4) is applied until the DNF expression is simple enough that the result can be computed directly. (E.g., if g consists of a single term.)

One special case in which $\rho(g)$ can be computed directly is when the DNF for g is *unate*. Then, literal $\neg x$ appears in $\rho(g)$ only if x is an implicant of g. This can be checked easily. A DNF expression is unate if at most one literal for each variable appears in it. This condition can be checked inexpensively. A unate cover contains all the prime implicants of the function it represents. Hence, in that case, one can compute $\rho(g)$ by inspection. Unateness is used throughout *Espresso* to speed up various computations.

Consider as an example the maximal reduction of $t = \neg w \wedge \neg y$ in

$$f = (\neg w \wedge \neg y) \vee (\neg w \wedge \neg z) \vee (\neg x \wedge z).$$

From $f_t^- = \neg x \vee \neg z$ it follows that $\rho(f_t^-) = x \wedge z$, and the maximally reduced term is $u = \neg w \wedge x \wedge \neg y \wedge z$.

A term may have more than one maximal expansion. The algorithm tries to select the one that leads to the elimination of the largest number of other terms. The choice among several possible expansions is formulated as a covering problem. *Espresso* computes a DNF for the negation of the given function before entering the optimization loop. An expansion of a term is valid if the expanded term does not intersect any term in the DNF for the complement. Two terms do not intersect iff one term contains one literal and the other contains its negation. We can, therefore, form a *blocking matrix* with one row for each term of the complement, and one column for each literal in the term to be expanded. The (i, j) entry of the matrix is 1 if the negation of the literal of Column j appears in the term of Row i; otherwise it is 0. If the (i, j) entry equals 1, intersection of the expanded term and the term of the complement associated to Row i can be avoided by retaining the literal of Column j. The optimal expansion process is, thus, translated into the problem of finding a set of columns that has ones in all rows. If we want to find the largest expansion, we use the number of columns as cost function. *Espresso*, however, tries to maximize the number of other terms that are covered by the expansion as its primary cost criterion.

The order in which the terms are reduced and expanded affects the result. The order in which the terms are considered has no effect on how much each term can be expanded. However, by considering implicants that cover more minterms first, one maximizes the probability of making a "small" implicant redundant. For a given order, the maximum reduction of a term is unique. However, the terms that are reduced first can be reduced more than those that follow them. The order of reduction is chosen so as to heuristically increase the chance that the successive expansion will make some terms redundant.

Discarding cubes to make a DNF expression irredundant is also formulated as a covering problem. This is similar to selecting a subset of all prime implicants in the exact minimization method. However, only the terms in the current DNF are considered. This usually makes the process much faster.

B. Multilevel Logic Synthesis

Multilevel expressions allow arbitrary nesting of conjunction, disjunctions, and negations. They are sometimes much more efficient than two-level expressions in the representation of logic

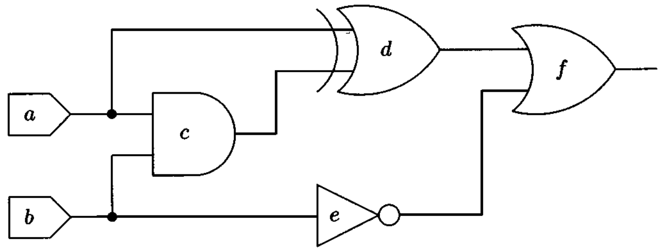


Fig. 15. Local simplification in Boolean networks.

functions: For the parity function of n variables, a multilevel representation grows linearly in size with n, whereas the optimum two-level expression is exponential in n. Since multilevel expressions include two-level expressions as special cases, multilevel expressions can never do worse than two-level expressions.

In multilevel synthesis, we are given a multilevel expression for a set of functions, and we seek another multilevel expression of reduced cost. Though cost depends in general on many parameters, including the area, delay, testability, and power dissipation of the resulting circuit, in our examples we shall concentrate on reducing the number of literals, which is a technology-independent measure of the area required to implement a function. Even with this simplified, and in some respects simplistic cost function, the problem of finding an optimum multilevel expression remains prohibitively difficult, so that only heuristic algorithms are used in practice. These heuristic approaches to multilevel optimization usually combine local optimization with restructuring. The input to the optimization process is a *Boolean network*: an acyclic graph with a logic function associated to each node. Local optimization is concerned with the simplification of the expressions of the node functions. Restructuring is concerned with changes in the structure of the graph.

1) Local Optimization: Algorithms for the optimizations of two-level expressions can be used for the local optimization of Boolean networks. To obtain good results, it is important to extract *don't care* information from the surrounding nodes. Consider the Boolean network of Fig. 15, in which the function of each node is indicated by the node shape. Suppose we are interested in optimizing Node $d = (a \wedge \neg c) \vee (\neg a \wedge c)$. If we consider the node in isolation, we cannot find any improvement. However, we may observe that it is impossible for a to be zero when c is one. In other words, $\neg a \wedge c$ is a don't care condition. Also, when e is 1, f is 1 regardless of d. Therefore, e is also a don't care condition for d. This condition is not in terms of the inputs to d; hence, it is not directly usable. However, it is possible to propagate the information by observing that $a \wedge \neg c$ implies $\neg b$, which in turn implies e. Therefore, $a \wedge \neg c$ is a don't care condition for d. Simplification now produces $d = 0$, because all minterms of the original function are "don't cares." Constant propagation leads then to $f = \neg b$.

Don't care conditions can be collected explicitly and passed to the minimizer, or inferred during the minimization process, which in this case is called *redundancy removal* and is closely reminiscent of ATPG. Returning to Fig. 15, let us check whether the replacement of d by a constant zero would alter the function of the network. A change in d is observable at the output f only if e is 0. This in turns implies that b is 1. However, if b is 1, d is 0 regardless of the value of a. This may be inferred (or *learned*) by assigning both values to a, and observing that $a \rightarrow \neg d$ and $\neg a \rightarrow \neg d$ jointly imply $\neg d$. In conclusion, when d is observable at the output, it must be0. Hence, its replacement by the constant is valid. Referring back to Section II-B, we see that line d *s*–*a*–0 is a redundant fault.

Algorithms that use don't cares can be extended in several ways. The function of a node can be temporarily made more expensive. As in the case of two-level minimization, the moves that increase the cost allow the algorithm to escape local minima. Another extension of the algorithms consists of relaxing the definition of "local" to include a group of related nodes. This may require suitable generalizations of the notion of "don't cares" [57], [58].

2) Restructuring: Restructuring of a Boolean network involves adding and removing nodes, and changing the arcs of the graph, while preserving functional equivalence. We concentrate on the task of factoring a two-level expression, which is at the heart of the restructuring algorithms. Once several complex two-level expressions have been factored, new nodes can be created for the subexpressions, and common factors can be identified.

Factoring of a two-level logic expression f consists of finding other two-level expressions p, q, and r, such that

$$f = (p \wedge q) \vee r \tag{5}$$

and then recursively factoring p, q, and r. Though, strictly speaking, the absence of a multiplicative inverse precludes the existence of division in Boolean algebras, it is customary to call *division* an operation that, given f and p, finds q and r that satisfy (5). The solution is not unique, because adding or removing from q minterms that are not in p or are in r has no effect on $(p \wedge q) \vee r$. Similarly, one can add or subtract from r minterms that are in both p and q. Division can, thus, be formulated as an optimization problem, in which one seeks a simple DNF expression for f in terms of p by specifying don't care conditions that relate p to the other inputs to f. This approach is rather expensive, hence ill-suited for the quick factorization of large Boolean networks. Another approach, called *algebraic division* is, therefore, in common use.

3) Algebraic Techniques: Algebraic division [59] owes its name to its reliance on a restricted set of laws that are shared by Boolean algebras and the ordinary algebra of polynomials over the real field: *associativity*, *commutativity*, and *distributivity* of product (conjunction) over sum (disjunction). Specifically excluded are idempotency ($a \wedge a = a \vee a = a$) and existence of the complement ($a \wedge \neg a = 0$ and $a \vee \neg a = 1$). An example of factorization that cannot be obtained by algebraic division is

$$(a \wedge c) \vee (\neg a \wedge b) \vee (b \wedge c) = (a \vee b) \wedge (\neg a \vee c).$$

The restriction in the scope of the optimization is compensated by the ability to apply algebraic techniques to large circuits. Algebraic techniques also have properties [60] that are not shared by the general Boolean techniques.

To define algebraic division, we first stipulate that the quotient of a term t_1 by another term t_2 is 0, if t_2 contains any literal that is not in t_1; otherwise, the quotient q is the term consisting of all the literals in t_1 and not in t_2. With this definition, either $q = 0$, or $t_1 = t_2 \wedge q$. The quotient of a DNF form f and a term t is the disjunction of all the quotients of the terms in f divided by t. (Here, it is convenient to assume that no term of f implies another term of f.) Finally the quotient of two DNF forms f and p is the disjunction of all terms that appear in all the quotients of f and the terms of p. The remainder is simply the disjunction of all terms that are not in the conjunction of p and the quotient q. It is easily seen that q and r are uniquely determined by this algorithm, and p and q share no variables.

Algebraic factorization proceeds by identifying good factors and then using algebraic division. The identification of good factors is based on the notion of *kernel*. A *primary divisor* of f is the quotient obtained on division of f by a term. A primary divisor is a kernel if no literal appears in all its terms. (Such a primary divisor is said to be *cube-free*.) The importance of kernels stems from two facts: On the one hand, they can be computed efficiently; on the other hand, two expressions have a nontrivial (that is, having at least two terms) common algebraic divisor only if they have kernels with at least two terms in common. Kernels, therefore, allow the algebraic factorization algorithm to greatly restrict the search space without compromising the quality of the results.

Of the approaches to the computation of kernels [61], [62], we briefly review the simple process that yields a *level-0* kernel from a given DNF expression. (A level-0 kernel has no other kernel than itself.) The process is based on iterating the following two steps:

1) the expression is made cube free by dropping all literals that appear in all terms;
2) the expression is divided by one literal that appears in more than one of its terms.

A level-0 kernel is obtained when the second step can no longer be applied. (Each literal appears at most once.)

By dividing a DNF expression f by one of its kernels p one obtains a factorization of f. However, under certain circumstances, the result is not maximally factored. For instance

$$(a \wedge b \wedge c) \vee (a \wedge b \wedge d) \vee (a \wedge e) \vee (a \wedge f) \vee g$$

has $c \vee d$ among its level-0 kernel. Division yields the factored form

$$a \wedge b \wedge (c \vee d) \vee a \wedge (e \vee f) \vee g$$

which can be further factored as

$$a \wedge (b \wedge (c \vee d) \vee (e \vee f)) \vee g.$$

To avoid this and similar inconveniences, one has to examine the quotient produced by division. If the quotient q is a single cube, the original divisor is replaced by a literal of q that appears in the most terms of f. Otherwise, the divisor is replaced by the expression obtained by making q cube-free. These simple modifications guarantee maximum factorization.

4) Technology Mapping: Local optimization and restructuring produce a so-called technology-independent Boolean network that must be adapted to the primitives available for implementation in the chosen technology (e.g., a standard-cell library, a field-programmable gate array architecture, or full-custom logic). We shall consider a popular approach to mapping a Boolean network to a fixed library of logic gates. The nodes of the network and the cells of the library are decomposed in terms of two-input AND gates and inverters; in this way, the problem of technology mapping is reduced to a graph covering problem. This covering problem is NP-hard, but a practically useful polynomial approximation is obtained by partitioning the network graph into a forest of trees [63].

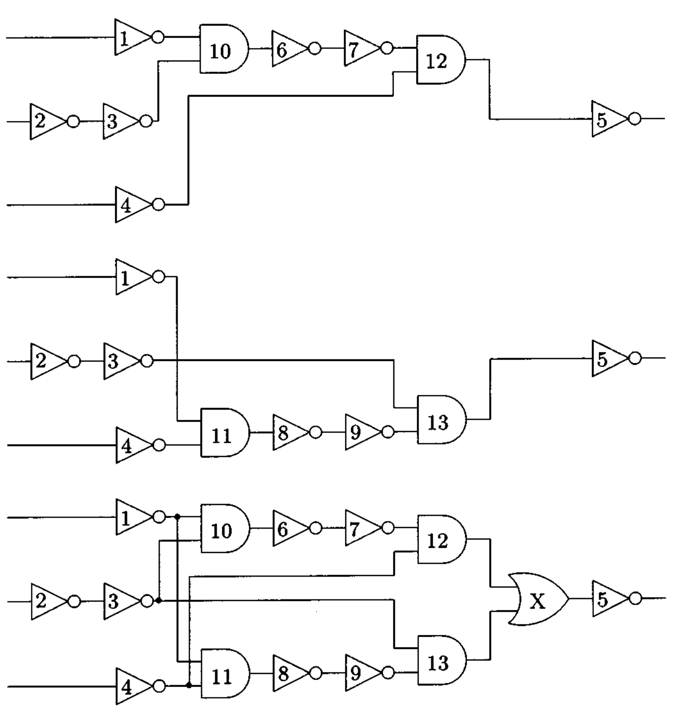


Fig. 16. Two decompositions of a function and their combined representation.

Tree covering can be solved by dynamic programming. Proceeding from the inputs to the outputs, the best cover of each node is found by examining all possible matches of library cells to the node. Suppose the target gate library contains NAND and NOR gates with up to three inputs and inverters, and suppose the cost of each gate is its number of transistors in fully complementary CMOS. With reference to the top circuit of Fig. 16, there is only one way to cover the output nodes of Gates 1, 2, and 4 (namely with an inverter of cost 2). The best cover of the output of Gate 3 is obtained by suppressing the two cascaded inverters.

The only cover of the output of Gate 10 is two-input NOR gate that matches Gates 1, 3, and 10. The cost is computed as the sum of the cost of the NOR gate and the costs of the covers for the inputs to the gate. In this case, the input to Gate 1 has cost zero because it is a primary input. The optimum cost of covering the input to Gate 3, on the other hand, has been determined to be two. Hence, the optimum cost of covering the output of Gate 10 is six. Proceeding toward the output, Gate 6 is considered. There are two possible covers. One is an inverter that matches Gate 6. The cost is in this case eight: two units for the inverter plus the cost of the optimum cover of Gate 10. The second cover is a two-input NAND gate that matches Gates 6 and 10. The total cost is four for the NAND gate plus two for the optimum cover of the output of Gate 1. Since the second cover is cheaper than the first, it is kept as the optimum cover of the output of Gate 6.

This process is continued until the optimum covers and their costs have been determined for all nodes. The circuit is then traversed from the output to the inputs. During this traversal, the optimum cover for the current node is added to the solution; its input nodes are then recursively visited. Continuing our example, the best cover for the primary output is a three-input NAND gate that covers Gates 5, 6, 7, 10, and 12. This gate is added to the solution. Then the output nodes of Gates 1, 3, and 4 are examined. This causes the addition of two inverters to the solution.

This basic scheme can be improved in several ways, among which we recall the Boolean matching approach of [64], and the algorithm of [42], which integrates the decomposition and covering steps. The advantages that stem from simultaneously solving decomposition and covering are illustrated in Fig. 16. The two upper circuits shown there are two decompositions of the same function.

We have seen that when the dynamic programming algorithm is applied to the first of the two decompositions, it finds an optimum solution consisting of a three-input NAND gate and two inverters. However, for the second decomposition the best solution consists of a two-input NOR gate for the subcircuit rooted at Gate 9 and a two-input NAND gate covering the rest of the circuit. This corresponds to a savings of two transistors over the first solution. The dynamic programming approach only guarantees optimality with respect to the chosen decomposition into AND gates and inverters.

The bottom circuit of Fig. 16 illustrates in simplified form the main idea of [42]. The circuit contains both decompositions shown at the top of the figure, and a fictitious *choice gate* (the OR gate marked by an X) connecting them. Dynamic programming can now proceed on this augmented graph from inputs to outputs as before. When the choice gate is reached, the algorithm selects the best cover between its two inputs. A match may span a choice gate, but can use only one of its inputs. The combined representation of multiple decompositions based on choice gates is called *mapping graph*. Its strength lies in the fact that the parts common to two decompositions need not be duplicated. This is illustrated by Gates 1–5 in Fig. 16.

Even though we have shown the construction of the mapping graph by combination of two given decompositions, the algorithm of [42] derives it from a single initial decomposition of the technology-independent Boolean network by application of local transformations. These transformations embed in the mapping graph all the decompositions that can be obtained by application of the associative and distributive properties, and by insertion of pairs of inverters. The graph may contain cycles. On the one hand, these cycles add flexibility to the algorithm, which can, for example, add an arbitrary even number of inverters between two gates. On the other hand, the determination of the optimum covers cannot be accomplished in a single pass from input to outputs, but requires a greatest fixpoint computation.

C. Model Checking

In model checking, we are given a sequential circuit and a property. We are asked to verify whether the model satisfies the property. Specifically, the circuit is supposed to perform a nonterminating computation, and the property specifies which (infinite) runs of the circuit are correct. The circuit is usually modeled as a *Kripke structure*

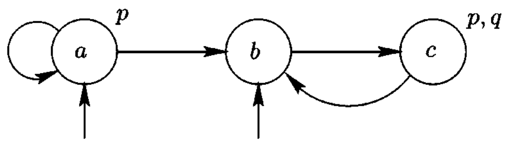


Fig. 17. A simple Kripke structure.

$$K = \langle S, T, S_0, A, L\rangle$$

where S is the finite set of states, and $T \subseteq S \times S$ is the *transition relation*, specifying what pairs of states are connected by transitions. $S_0 \subseteq S$ is the set of *initial states*, A is the set of *atomic propositions*, and L: $S \rightarrow 2^A$ is the *labeling function*, which says what atomic propositions hold at each state.

A Kripke structure is depicted in Fig. 17, for which

$$\begin{aligned} S &= \{a, b, c\} \\ T &= \{(a, a), (a, b), (b, c), (c, b)\} \\ S_0 &= \{a, b\} \\ A &= \{p, q\} \\ L(a) &= \{p\} \\ L(b) &= \{\} \\ L(c) &= \{p, q\}. \end{aligned}$$

The conditions under which transitions are enabled are not captured by T. We are only told that the transition is possible. Alternatively, we can regard the Kripke structure as the model of a *closed* system—that is, a system and its environment. We shall assume a *complete* transition relation; that is, we shall assume that every state has at least one successor.

1) The Logic CTL:* The properties are expressed in various formalisms. We consider the *temporal logic* CTL* [47], whose two subsets *computational tree logic* (CTL) [65] and *linear time logic* (LTL) [66] and [67] are commonly used in practice. CTL* is a *branching time logic* that augments propositional logic with path quantifiers (E and A) and temporal operators (U, R, X, G, and F). From every state, a system may evolve in several possible ways; that is, several computation paths may exist. A branching time logic allows one to express properties that may hold for at least one computation path, or for all computation paths. The temporal operators describe the evolution in time of the propositions. For instance, if φ is a propositional formula (e.g., $p \vee \neg q$), then AGφ means that φ is always true in all computations, and EFφ means that there exists a computation in which φ is either true now or is going to be true in the future. Operators, propositions, and quantifiers can be combined to form more complex formulae like AGFφ, which states that along all computation paths φ is true infinitely often.

The formal definition of CTL* requires the definition of both *state formulae* (asserting something of a state) and *path formulae* (asserting something of a path). Any atomic proposition is a state formula, and any state formula is also a path formula; furthermore

if φ and ψ are state formulae, so are $\neg\varphi$ and $\varphi \wedge \psi$
if φ is a path formula, $\mathtt{E}\varphi$ is a state formula
if φ and ψ are path formulae, so are $\neg\varphi$ and $\varphi \wedge \psi$
if φ and ψ are path formulae, so are $\mathtt{X}\varphi$ and $\varphi\mathtt{U}\psi$.

The logic CTL* is the set of state formulae defined by the above rules. (A path formula can only be a proper subformula of a CTL* formula.) The remaining connectives used in CTL* are defined as abbreviations in terms of $\neg$, $\wedge$, E, X, and U: $\psi\mathtt{R}\varphi$ abbreviates $\neg(\neg\psi\mathtt{U}\neg\varphi)$; $\mathtt{F}\varphi$ abbreviates $\mathbf{true}\,\mathtt{U}\varphi$, $\mathtt{G}\varphi$ abbreviates $\mathbf{false}\,\mathtt{R}\varphi$; and $\mathtt{A}\varphi$ abbreviates $\neg\mathtt{E}\neg\varphi$. Additional Boolean connectives ($\vee$, $\rightarrow$, $\leftrightarrow$, $\ldots$) are defined in terms of $\neg$ and $\wedge$ in the usual way.

For a given structure K, if a state formula φ is true of state s, we write $K, s \models \varphi$. Likewise, if a path formula φ is true of path $\pi = (s_0, s_1, \ldots)$, we write $K, \pi \models \varphi$. The suffix of path π starting at state s_i is denoted by π^i. K is omitted if there is no ambiguity. With these definitions, the truth of a CTL* formula can be defined recursively. If $p \in A$, $s \models p$ iff $p \in L(s)$; if φ is a state formula, $\pi \models \varphi$ iff $s_0 \models \varphi$; otherwise,

$s \models \varphi \wedge \psi$ iff $s \models \varphi$ and $s \models \psi$
$s \models \neg\varphi$ iff $s \not\models \varphi$
$s \models \mathtt{E}\varphi$ iff $\exists\pi$ starting at s such that $\pi \models \varphi$
$\pi \models \varphi \wedge \psi$ iff $\pi \models \varphi$ and $\pi \models \psi$
$\pi \models \neg\varphi$ iff $\pi \not\models \varphi$
$\pi \models \mathtt{X}\varphi$ iff $\pi^1 \models \varphi$
$\pi \models \psi\mathtt{U}\varphi$ iff $\exists i \geq 0,\ \pi^i \models \varphi,$ and $0 \leq j < i \rightarrow \pi^j \models \psi$.

$K \models \varphi$ means that φ holds in all initial states of K. If K is the Kripke structure of Fig. 17, we have $K \not\models \mathtt{AG}p$ and $K \models \mathtt{AGF}p$. We also have $K \not\models \mathtt{EG}p$ and $K \not\models \neg\mathtt{EG}p$. Indeed, $K, b \not\models \mathtt{EG}p$ because there is no path starting at b such that p always holds, and $K, a \not\models \neg\mathtt{EG}p$ because the path that stays in a forever satisfies $\mathtt{G}p$.

With slight abuse of notation, we shall denote the set of states in which φ holds by φ itself. In model checking, the CTL* formula φ, we usually proceed by finding the set of states φ, and then verify that $S_0 \subseteq \varphi$. The recursive definition of the semantics suggests that, to find the states that satisfy a given formula, we recursively analyze its subformulae, and then apply the appropriate case from the above definition. The immediate difficulty that we face in this approach is that in the case of $\varphi = \mathtt{E}\psi$, the subformula ψ is true of a set of paths, not a set of states. We shall address this difficulty gradually, starting from simple cases.

2) Model Checking CTL Formulae: Suppose we want to find the states that satisfy $\mathtt{EF}\varphi$, where φ is a state formula. The key observation is that

$$\mathtt{EF}\varphi = \varphi \cup \mathtt{EXEF}\varphi. \tag{6}$$

Equation (6) says that there is a path from a state s to another state where φ holds if φ holds at s, or if s has a successor with a path to a state where φ holds. Equivalently, $\mathtt{EF}\varphi$ is a fixpoint of the function $\lambda Z.\varphi \cup \mathtt{EX}Z$; it can be proved to be the least fixpoint. In the notation of μ-calculus, this is written

$$\mathtt{EF}\varphi = \mu Z.\varphi \cup \mathtt{EX}Z \tag{7}$$

where μ prescribes a least fixpoint computation and Z is the iteration variable. The set of states satisfying $\mathtt{EX}Z$ is the set of predecessors of states in Z. Tarski's theorem [68] ensures the existence of the fixpoint, which can be computed by repeated application of the function starting with $Z = \emptyset$

$$\mu Z.\tau(Z) = \bigcup_{i \geq 0} \tau^i(0).$$

The finiteness of the state set guarantees convergence. Returning to the Kripke structure of Fig. 17, suppose we want to check whether $K \models \mathtt{EF}p$. We proceed by computing

$$\mathtt{EF}p = \mu Z.\{a, c\} \cup \mathtt{EX}Z$$

since

$$\{a, c\} = \{s \in S : p \in L(s)\}.$$

The iterates of the fixpoint computation are as follows:

$$\begin{aligned} Z_0 &= \{\} \\ Z_1 &= \{a, c\} \\ Z_2 &= \{a, b, c\} \\ Z_3 &= \{a, b, c\}. \end{aligned}$$

(In this case, the last iteration could be avoided by observing that $Z_2 = S$.) As a last step, we verify that $S_0 \subseteq Z_3$. Hence, the formula holds.

With reasoning analogous to that used for $\mathtt{EF}\varphi$, one can show that

$$\mathtt{E}\psi\mathtt{U}\varphi = \mu Z.\varphi \cup (\psi \cap \mathtt{EX}Z) \tag{8}$$

$$\mathtt{EG}\varphi = \nu Z.\varphi \cap \mathtt{EX}Z \tag{9}$$

where ν indicates the greatest fixpoint. In summary, if every temporal operator is immediately preceded by a path quantifier, model checking can be reduced to a series of fixpoint computations for functions that map sets of states to other sets of states. The restriction on the syntax of the formula results in the logic CTL. CTL is not as expressive as CTL*. For instance, the CTL* formula $\mathtt{AFG}p$ has no equivalent in CTL. On the other hand, μ-calculus, in which we have cast the problem of CTL model checking, can express all CTL* (and more). In particular, we can write a μ-calculus formula that computes the set of states along paths that satisfy a condition infinitely often. (Such a condition is called a *fairness constraint*.) Let c designate the set of states that satisfy the desired condition. Then the set of states along paths satisfying $\mathtt{GF}c$ is given by

$$F = \nu Z.\mathtt{EX}(\mathtt{E}Z\mathtt{U}(Z \cap c)). \tag{10}$$

The extension of (10) to more than one fairness condition is straightforward. Suppose we want to compute the states that satisfy $\mathtt{GF}\{b\}$ for the Kripke structure of Fig. 17. The iterates of the fixpoint computation are

$$\begin{aligned} Z_0 &= \{a, b, c\} \\ Z_1 &= \mathtt{EX}(\mathtt{E}\{a, b, c\}\mathtt{U}(\{a, b, c\} \cap \{b\})) \\ &= \mathtt{EX}\{a, b, c\} = \{a, b, c\}. \end{aligned}$$

The computation, therefore, converges to S in one iteration.

The addition of fairness constraints to CTL is useful in itself, because these constraints can help in modeling the environment of the system being verified, or can eliminate spurious behaviors introduced in a model by the abstraction of some details. In addition, a model checking procedure for CTL augmented with fairness constraints is an important building step toward a general solution for CTL* model checking.

3) Model Checking LTL Formulae: Let us consider now another special case: formulae whose only path quantifier is the first operator of the formula. As an example, consider $\mathtt{EGF}p$. If we strip the leading quantifiers from such formulae, we obtain path formulae that contain no quantifiers. Specifically, we obtain the formulae of the logic LTL. We now show how we can model check LTL formulae. That is, we show how to decide if there is a path in the Kripke structure that satisfies the LTL formula. To this effect, we shall convert the formula into an automation that will be composed with the system to be verified. The composition constrains the system to satisfy the LTL property along all its computation paths. If at least one path is still viable in the composed system, the property holds. The conversion from LTL to automata is based on rewriting rules known as the *tableau rules* [cf. (7) and (9)]:

$$\begin{aligned} \mathtt{F}\varphi &= \varphi \vee \mathtt{XF}\varphi \\ \mathtt{G}\varphi &= \varphi \wedge \mathtt{XG}\varphi. \end{aligned}$$

(Similar rules can be written for $\mathtt{U}$ and $\mathtt{R}$.) When applied to $\varphi = \mathtt{GF}p$, these rules produce

$$(p \wedge \mathtt{XGF}p) \vee (\mathtt{XF}p \wedge \mathtt{XGF}p)$$

which can be rewritten as

$$(p \wedge \mathtt{X}\varphi) \vee (\mathtt{XF}p \wedge \mathtt{X}\varphi).$$

This formula says that there are two ways to satisfy φ: by satisfying p in the current state and φ in the next; or by satisfying $\mathtt{F}p$ and φ in the next state (with no obligation in the current state). We can, therefore, create two states in the automation corresponding to these two possibilities. (See Fig. 18. The double circle identifies the accepting state. Both states are initial.) The first state is labeled p. Its successors are the states that satisfy φ: both states. The second state has label $\mathbf{true}$; its successors are the states that satisfy $\psi = \mathtt{F}p \wedge \mathtt{GF}p$. We, therefore, apply the tableau rules to ψ and we find the same formula produced by the expansion of φ. No new states need to be generated, because the successors of the second state are, once again, the two existing states.

To complete the construction, we have to observe that a run of the automation that from some point onward does not leave the second state, does not guarantee the satisfaction of φ because

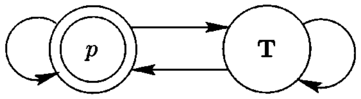


Fig. 18. Automation for the LTL formula $\varphi = \mathtt{GF}p$.

p is not guaranteed to be true infinitely often. This problem is signaled by the presence of $\mathtt{F}p$ in the state. Following the loop simply means satisfying an eventuality by postponement. This postponement cannot be indefinite. To obviate, we add a fairness constraint to the automation, specifying that a valid run must be in a state other than the second state (i.e., in the first state, which is, therefore, an *accepting* state) infinitely often. The resulting automation is called a Büchi automation.

4) Model Checking Full CTL:* Although our simple example does not illustrate all the details of the translation from LTL formula to automation, it does point out the salient facts: That the translation produces a transition structure and one or more fairness constraints. A state satisfies the formula iff, in the composition of the model and the automation for the property, there is a computation path originating at that state that satisfies all fairness constraints. We have seen that this can be determined by evaluating (10). Hence, we have reduced LTL model checking to μ-calculus.

The last step is to show how to model check a generic CTL* formula φ with the techniques developed so far. Notice that the formula, once the abbreviations are expanded, must either be a purely propositional formula (e.g., $p \wedge \neg q$), or contain at least one existential quantifier. The former case is easy. If φ contains a quantifier, then it has a subformula $\mathtt{E}\psi$, where ψ is an LTL formula. We can apply the (existential) LTL model checking algorithm to find the states that satisfy $\mathtt{E}\psi$, and replace $\mathtt{E}\psi$ in φ with a new atomic proposition that holds exactly in those states. The process is repeated until the entire formula φ is replaced by one atomic proposition that is true of all the states where φ is true. If a subformula $\mathtt{E}\psi$ is a CTL formula, the translation to automation is not necessary, and the CTL model checking algorithm can be applied directly to it.

5) Explicit and Symbolic Model Checking: Equations (7)–(10) form the heart of the model checking procedure. They involve the computation of sets of states. In *explicit* model checking algorithms, states are usually represented as bit strings and stored in hash tables. The transition structures are represented in various ways; to fix ideas, we shall assume that they are stored as adjacency lists. Under these assumptions, computation of least fixpoints amounts to reachability analysis of the transition structure, and can be performed in time linear in the number of transitions by depth-first search. The computation of (10) translates into the computation of the strongly connected components of the transition structure. This can also be done in linear time by depth-first search. It should be noted, however, that the translation from LTL to automata may incur an exponential blow-up in size. Therefore, the CTL* model checking problem is overall PSPACE-complete [69]. By contrast, CTL model checking is linear in both the size of the system and the length of the formula.

In symbolic model checking, sets are represented by their characteristic functions. Assuming, without loss of generality, that a state is an element of $\{0, 1\}^n$ for some n, and a transition is, therefore, an element of $\{0, 1\}^n \times \{0, 1\}^n$, then the characteristic function of a set of states W is a function χ_W: $\{0, 1\}^n \rightarrow \{0, 1\}$ that maps the elements of W to 1 and the other states to 0. The most popular representation of characteristic functions is BDDs. (See Section IV-D.) The main advantage of characteristic functions is that very large sets can have very small representations. For instance, the characteristic function $x_1 \vee x_{100}$ represents a subset of $\{0, 1\}^{100}$ containing $3 \cdot 2^{98}$ elements. Intersection, union, and complementation of sets correspond to conjunction, disjunction, and negation of their characteristic functions.

Symbolic computation of least fixpoints also amounts to reachability analysis, and is most naturally performed in breadth-first manner. Let $T(x, y)$ describe the characteristic function of the transition relation of a Kripke structure, and let $P(y)$ describe the characteristic function of a set of states. The variables $x = (x_1, \ldots, x_n)$ range over the origin states of the transitions (the present states), while the variables $y = (y_1, \ldots, y_n)$ range over the destination states of the transitions (the next states). The states that are connected to at least one state in $P(y)$ by a transition in $T(x, y)$ are given by

$$Pre(T, P) = \exists y[T(x, y) \wedge P(y)]. \qquad (11)$$

The conjunction of $T(x, y)$ and $P(y)$ yields the transitions ending in a state in P. The quantification of the y variables discards the destination states, thus producing the desired set of predecessors. This so-called *preimage* computation allows one to compute the states satisfying the EX subformulae in (7)–(10) without explicitly manipulating the individual states or transitions. For the Kripke structure of Fig. 17, suppose that the following encoding of the states is chosen:

$$\{a\} = \neg x_1 \wedge \neg x_0, \quad \{b\} = \neg x_1 \wedge x_0, \quad \{c\} = x_1 \wedge \neg x_0.$$

Then the characteristic function of the transition relation is

$$T(x_1, x_0, y_1, y_0) \\ = \neg x_0 \wedge \neg y_1 \wedge (\neg x_1 \vee y_0) \vee \neg x_1 \wedge x_0 \wedge y_1 \wedge \neg y_0 \ .$$

The characteristic function of the set of states $\{b\}$ in terms of y variables is $P(y_1, y_0) = \neg y_1 \wedge y_0$. The characteristic function of the predecessors of b is, therefore, computed as

$$\exists y_1 y_0 \, [\neg x_0 \wedge \neg y_1 \wedge y_0] = \neg x_0$$

which is seen to correspond to the set $\{a, c\}$.

The ability to deal with symbolic representations of sets of states is highly effective for verification problems in which the quantities of interest have simple characteristic functions. However, it should be noted that this is not always the case. Moreover, computation of strongly connected components by depth-first search is not well suited to symbolic computation. Direct use of (10), on the other hand, leads to an algorithm that requires a quadratic number of preimage computations. In summary, in spite of the great success of symbolic model checking, there remain cases in which the explicit techniques are superior.

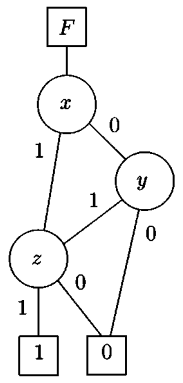


Fig. 19. BDD for $F = x \wedge z \vee y \wedge z$.

D. BDDs

Many algorithms in logic synthesis and formal verification manipulate complex logic functions. An efficient data structure to support this task is, therefore, of great practical usefulness. BDDs [48] have become very popular because of their efficiency and versatility.

A BDD is an acyclic graph with two leaves representing the constant functions 0 and 1. Each internal node n is labeled with a variable v and has two children t (*then*) and e (*else*). Its function is inductively defined as

$$f(n) = (v \wedge f(t)) \vee (\neg v \wedge f(e)). \qquad (12)$$

Thhree restrictions are customarily imposed on BDDs.

1) There may not be isomorphic subgraphs.
2) For all internal nodes, $t \neq e$.
3) The variables are totally ordered: The variables labeling the nonconstant children of a node must follow in the order the variable labeling the node itself.

Under these restrictions, BDDs provide a canonical representation of functions, in the sense that for a fixed variable order there is a bijection between logic functions and BDDs. Unless otherwise stated, BDDs will be assumed to be reduced and ordered. The BDD for $F = (x \wedge z) \vee (y \wedge z)$ with variable order $x < y < z$ is shown in Fig. 19.

Canonicity is important in two main respects: It makes equivalence tests easy, and it increases the efficiency of *memoization* (the recording of results to avoid their recomputation). On the other hand, canonicity makes BDDs less concise than general circuits. The best-known case is that of multipliers, for which circuits are polynomial, and BDDs exponential [70]. Several variants of BDDs have been devised to address this limitation. Some of them have been quite successful for limited classes of problems. (For instance, Binary Moment Diagrams [71] for multipliers.) Other variants of BDDs have been motivated by the desire to represent functions that map $\{0, 1\}^n$ to some arbitrary set (e.g., the real numbers) [72].

For ordered BDDs, $f(t)$ and $f(e)$ do not depend on v; hence, comparison of (3) and (12) shows that $f(t) = f(n)_v$ and $f(e) = f(n)_{\neg v}$. This is the basis of most algorithms that manipulate BDDs, because for a generic Boolean connective $\langle op \rangle$,

$$f \langle op \rangle g = (x \wedge (f_x \langle op \rangle g_x)) \vee (\neg x \wedge (f_{\neg x} \langle op \rangle g_{\neg x})) \, . \qquad (13)$$

Equation (13) is applied with x chosen as the first variable in the order that appears in either f or g. This guarantees that the cofactors can be computed easily: If x does not appear in f, then $f_x = f_{\neg x} = f$; otherwise, f_x is the *then* child of f, and $f_{\neg x}$ is the *else* child. Likewise, for g. The terminal cases of the recursion depend on the specific operator. For instance, when computing the conjunction of two BDDs, the result is immediately known if either operand is constant, or if the two operands are identical or complementary. All these conditions can be checked in constant time if the right provisions are made in the data structures [73].

Two tables are used by most BDD algorithms: The *unique table* allows the algorithm to access all nodes using the triple (v, t, e) as key. The unique table is consulted before creating a new node. If a node with the desired key is already in existence, it is re-utilized. This approach guarantees that equivalent functions will share the same BDD, rather than having isomorphic BDDs; therefore, equivalence checks are performed in constant time.

The *computed table* stores recent operations and their results. Without the computed table, most operations on BDDs would take time exponential in the number of variables. With a lossless computed table (one that records the results of all previous computations) the time for most common operations is polynomial in the size of the operands. The details of the implementation of the unique and computed tables dramatically affect the performance of BDD manipulation algorithms, and have been the subject of careful study [74].

The order of variables may have a large impact on the size of the BDD for a given function. For adders, for instance, the optimal orders give BDDs of linear size, while bad orders lead to exponential BDDs. The optimal ordering problem for BDDs is hard [75]. Hence, various methods have been proposed to either derive a variable order from inspection of the circuit for which BDDs must be built, or by dynamically computing a good order while the BDDs are built [76].

An exhaustive list of applications of BDDs to problems in CAD is too long to be attempted here. Besides symbolic model checking, which was examined in Section IV-C-5, BDD-based algorithms have been proposed for most synthesis tasks, including two-level minimization, local optimization, factorization, and technology mapping. In spite of their undeniable success, BDDs are not a panacea; their use is most profitable when the algorithms capitalize on their strengths [77], and avoid their weaknesses by combining BDDs with other representations [44], [45], [78]; for instance with satisfiability solvers for CNF expressions.

V. Conclusion

Even within generous space limits we could only afford to address very few of the algorithms currently in use for test, physical design, synthesis, and verification. The following trends can be identified:

- Although tremendous progress has been made, effective and efficient new algorithms are still needed in all aspects of VLSI CAD.
- Full scan is becoming a de facto standard for testing and diagnosis for sequential circuits.
- BIST is commonly used for array structures and its use will continue to expand at a rapid rate.
- New fault modes will be addressed that deal with cross-coupling electrical phenomena, such as crosstalk, signal integrity, and process variation.
- Manufacturing test techniques will be employed to a greater extent to design validation as the use of functional tests become less useful.
- Congestion minimization and interaction of congestion and timing in placement is a fundamental and under-studied problem.
- Integration of logic and physical design will be a dominant theme in logic synthesis research. The current approaches were not designed to deal with the challenges of deep submicron processes. This often leads to decrease in productivity when designers have to go through several iterations to solve timing, or signal immunity problems. The solutions that will be proposed will include a tighter integration of layout and logic optimization, and the adoption of more controlled design styles (both logic and physical).
- Reconfigurable computers are likely to fill the gap between general-purpose programmable components and ASICs. New logic design problems will emerge that will demand new algorithms for their efficient solutions.
- Formal verification will increasingly rely on abstraction techniques and compositional approaches like assume/guarantee reasoning. Semi-formal approaches, that is, approaches that help increase the confidence in the correctness of a design by augmenting simulation-based verification with formal techniques, will become important.
- From the point of view of the CAD developer, the integration of various engines into powerful tools will become more prevalent.

References

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. Piscataway, NJ: IEEE Press, 1990.

[2] J. P. Roth, "Diagnosis and automatic failures: A calculus and a method," *IBM J. Res. Develop.*, vol. 1, no. 4, pp. 278–291, July 1966.

[3] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Trans. Comput.*, vol. C-30, no. 3, pp. 215–222, Ma. 1981.

[4] H. Fujiwara and T. Shimono, "On the acceleration of test generation algorithms," *IEEE Trans. Compute.*, vol. C-32, no. 12, pp. 1137–1144, Dec. 1983.

[5] S. Seshu, "On an improved diagnosis program," *IEEE Trans. Electron. Comput.*, vol. EC-12, pp. 76–79, Feb. 1965.

[6] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, and J. McCarthy, "Fault simulation for structured VLSI," *VLSI Syst. Design*, vol. 6, no. 12, pp. 20–32, Dec. 1985.

[7] D. B. Armstrong, "A deductive method for simulating faults in logic circuits," *IEEE Trans. Comput.*, vol. C-21, pp. 464–471, May 1972.

[8] E. G. Ulrich and T. G. Baker, "Concurrent simulation of nearly identical digital networks," *IEEE Trans. Comput.*, vol. 7, pp. 39–44, Apr. 1974.

[9] M. Abramovici, P. R. Menon, and D. T. Miller, "Critical path tracing: An alternative to fault simulation," *IEEE Design Test Comput.*, vol. 1, no. 1, pp. 83–93, Feb. 1984.

[10] M. Abramovici and M. A. Breuer, "Multiple fault diagnosis in combinational circuits based on effect-cause analysis," *IEEE Trans. Comput.*, vol. C-29, no. 6, pp. 451–460, June 1980.

[11] ——, "Fault diagnosis in synchronous sequential circuits based on an effect-cause analysis," *IEEE Trans. Comput.*, vol. C-31, no. 12, pp. 1162–1172, Dec. 1982.

[12] M. Abromovici, "Fault diagnosis in digital circuits based an effect-cause analysis, Ph.D. dissertation," Univ. Southern California and USCEE, Tech. Rep. 508, 1980.

[13] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, pp. 291–307, Feb. 1970.

[14] C. M. Fiduccia and R. M. Mattheyes, "A linear time heuristic for improving network partitions," in *Proc. IEEE-ACM Design Automation Conf.*, 1982, pp. 175–181.

[15] B. Krishnamurthy, "An improved min-cut algorithm for partitioning VLSI networks," *IEEE Trans. Comput.*, vol. C-33, pp. 438–446, May 1984.

[16] J. Cong and M. L. Smith, "A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design," in *Proc. IEEE-ACM Design Automation Conf.*, 1993, pp. 755–760.

[17] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," Sandia National Laboratories, Tech. Rep. SAND93-1301, 1993.

[18] S. Hauck and G. Boriello, "An evaluation of bipartitioning techniques," presented at the Chapel Hill Conf. Advanced Research in VLSI, 1995.

[19] G. Karypis and V. Kumar, "A fast and highly quality multilevel scheme for partitioning irregular graphs," *SIAM J. Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1999.

[20] ——, "METIS 3.0: Unstructured graph partitioning and sparse matrix ordering system," Department of Computer Science, University of Minnesota, Tech. Rep. 97-061, 1997.

[21] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: Application in VLSI domain," in *Proc. IEEE-ACM Design Automation Conf.*, 1997, pp. 526–529.

[22] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," in *Proc. IEEE-ACM Design Automation Conf.*, 1999, pp. 343–348.

[23] C. Alpert, "The ISPD98 circuit benchmark suite," in *Proc. Int. Symp. Physical Design*: ACM, Apr. 1998, pp. 18–25.

[24] S. Rao, "Finding near optimal separators in planar graphs," in *Proc. Symp. Foundations of Computer Science*, 1987, pp. 225–237.

[25] S. Wimer, I. Koren, and I. Cederbaum, "Optimal aspect ratios of building blocks in VLSI," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 139–145, Feb. 1989.

[26] J. P. Cohoon, "Distributed genetic algorithms for the floorplan design problem," *IEEE Trans. Computer-Aided Design*, vol. 10, no. 4, pp. 483–492, April 1991.

[27] D. F. Wong, H. W. Leong, and C. L. Liu, *Simulated Annealing for VLSI Design*: Kluwer Academic, 1988.

[28] R. M. Kling and P. Banerjee, "Optimization by simulated evolution with applications to standard cell placement," in *Proc. IEEE-ACM Design Automation Conf.*, 1990, pp. 20–25.

[29] M. Rebaudengo and M. S. Reorda, "GALLO: A genetic algorithm for floorplan area optimization," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 943–951, Aug. 1996.

[30] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich, "GORDIAN: VLSI placement by quadratic programming and slicing optimization," *IEEE Trans. Computer-Aided Design*, vol. 10, p. 365, Mar. 1991.

[31] C. Alpert, T. Chan, D. Huang, and A. B. Kahng, "Faster minimization of linear wirelength for global placement," in *Proc. Int. Symp Physical Design*: ACM, 1997, pp. 4–11.

[32] T. Lengauer and M. Lugering, "Integer programming formulation of global routing and placement problems," in *Algorithm Aspects of VLSI Layout*, M. Sarrafzadeh and D. T. Lee, Eds, Singapore: World Sceintific, 1993.

[33] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[34] G. Sigl, K. Doll, and F. M. Johannes, "Analytical placement: A linear or a quadratic objective function," in *Proc. IEEE-ACM Design Automation Conference*, 1991, pp. 427–431.

[35] D. Huang and A. B. Kahng, "Partitioning-based standard-cell global placement with an exact objective," in *Proc. Int. Symp. Physical Design*, Apr. 1997, pp. 18–25.

[36] H. Eisenmann and F. M. Johannes, "Generic global placement and floorplanning," in *Proc. IEEE-ACM Design Automation Conf.*, 1998, pp. 269–274.

[37] R. Nair, "A simple yet effective technique for global wiring," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 165–172, Mar. 1987.

[38] M. Wang and M. Sarrafzadeh, "Behavior of congestion minimization during placement," in *Proc. Int. Symp. Physical Design*, April 1999, pp. 145–150.

[39] C. Y. Lee, "An algorithm for path connection and its application," *IRE Trans. Electron. Comput.*, vol. EC-10, pp. 346–365, 1961.

[40] A. Kahng, J. Cong, and G. Robins, "High-performance clock routing based on recursive geometric matching," in *Proc. IEEE-ACM Design Automation Conf.*: IEEE/ACM, 1991, pp. 322–327.

[41] J. Cong and C. K. Koh, "Minimum-cost bounded-skew clock routing," in *Proc. IEEE Int. Symp. Circuits and Systems*, 1995, pp. 215–218.

[42] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 813–834, Aug. 1995.

[43] M. Pedram, "Power minimization in IC design: Principles and applications," *ACM Trans. Design Automat. Electron. Syst.*, vol. 1, no. 1, pp. 3–56, 1996.

[44] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proc. IEEE-ACM Design Automation Conf.*, Anaheim, CA, June 1997, pp. 263–268.

[45] J. R. Burch and V. Singhal, "Tight integration of combinational verification methods," in *Proc. IEEE International Conf. Computer-Aided Design*, San Jose, CA, Nov. 1998, pp. 570–576.

[46] R. P. Kurshan, "Formal verification in a commercial setting," in *Proc. IEEE-ACM Design Automation Conf.*, Anaheim, CA, June 1997, pp. 258–262.

[47] E. A. Emerson and J. Y. Halpern, "'Sometimes' and 'not never' revisited: On branching time versus linear time temporal logic," *J. Assoc. Computing Machinery*, vol. 33, no. 1, pp. 151–178, 1986.

[48] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 677–691, Aug. 1986.

[49] K. L. McMillan, *Symbolic Model Checking*. Boston, MA: Kluwer Academic, 1994.

[50] W. V. Quine, "Two theorems about truth functions," *Boletin de la Sociedad Matematica Mexicana*, vol. 10, pp. 64–70, 1953.

[51] E. J. McCluskey Jr., "Minimization of Boolean functions," *Bell Syst. Tech. J.*, vol. 35, pp. 1417–1444, Nov. 1956.

[52] J. F. Gimpel, "A method of producing a Boolean function having an arbitrarily prescribed prime implicant table," *IEEE Trans. Electron. Comput.*, vol. EC-14, pp. 485–488, June 1965.

[53] O. Coudert and J. C. Madre, "Implicit and incremental computation of primes and essential primes of Boolean functions," in *Proc. IEEE-ACM Design Automation Conf.*, Anaheim, CA, June 1992, pp. 36–39.

[54] ——, "New ideas for solving covering problems," in *Proc. IEEE-ACM Design Automation Conf.*, San Francisco, CA, June 1995, pp. 641–646.

[55] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Boston, MA: Kluwer Academic, 1984.

[56] R. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 727–750, Sept. 1987.

[57] R. K. Brayton and F. Somenzi, "Boolean relations and the incomplete specification of logic networks," in *VLSI'89*, G. Musgrave and U. Lauther, Eds. Amsterdam, The Netherlands: North-Holland, 1989, pp. 231–240.

[58] S. Yamashita, H. Sawada, and N. Nagoya, "A new method to express functional permissibilities for LUT based FPGA's and its applications," in *Proc. IEEE Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 1996, pp. 254–261.

[59] R. K. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," in *Proc. IEEE Int. Symp. Circuits and Systems*, Rome, Italy, May 1982, pp. 49–54.

[60] G. D. Hachtel, R. M. Jacoby, K. Keutzer, and C. R. Morrison, "On properties of algebraic transformations and the synthesis of multifault-irredundant circuits," *IEEE Trans. Computer-Aided Design*, vol. 11, no. 3, pp. 313–321, Mar. 1992.

[61] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proc. IEEE*, vol. 78, pp. 264–300, Feb. 1990.

[62] J. Rajski and J. Vasudevamurthy, "The testability-preserving concurrent decomposition and factorization of Boolean expressions," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 778–793, June 1993.

[63] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," in *Proc. IEEE-ACM Design Automation Conf.*, June 1987, pp. 341–347.

[64] F. Mailhot and G. De Micheli, "Technology mapping using Boolean matching," in *Proc. Eur. Conf. Design Automation*, Glasgow, U.K., Mar. 1990, pp. 180–185.

[65] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proceedings Workshop on Logics of Programs*, ser. LNCS 131. Berlin, Germany: Springer-Verlag, 1981, pp. 52–71.

[66] P. Wolper, M. Y. Vardi, and A. P. Sistla, "Reasoning about infinite computation paths," in *Proc. 24th IEEE Symp. Foundations of Computer Science*, 1983, pp. 185–194.

[67] O. Lichtenstein and A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," presented at the 12th Annual ACM Symp. Principles of Programming Languages, New Orleans, Jan. 1985.

[68] A. Tarski, "A lattice-theoretic fixpoint theorem and its applications," *Pacific J. Math.*, vol. 5, pp. 285–309, 1955.

[69] E. A. Emerson and C.-L. Lei, "Efficient model checking in fragments of the propositional mu-calculus," in *Proc. 1st Annu. Symp. Logic in Computer Science*, June 1986, pp. 267–278.

[70] R. E. Bryant, "On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication," *IEEE Trans. Comput.*, vol. 40, no. 2, pp. 205–213, Feb. 1991.

[71] R. Bryant and Y.-A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *Proc. IEEE-ACM Design Automation Conf.*, San Francisco, CA, June 1995, pp. 535–541.

[72] Y.-T. Lai and S. Sastry, "Edge-valued binary decision diagrams for multi-level hierarchical verification," in *Proc. IEEE-ACM Design Automation Conf.*, Anaheim, CA, June 1992, pp. 608–613.

[73] F. Somenzi, "Binary decision diagrams," in *Calculational System Design*, M. Broy and R. Steinbrüggen, Eds. Amsterdam, The Netherlands: IOS, 1999, pp. 303–366.

[74] S. Manne, D. C. Grunwald, and F. Somenzi, "Remembrance of things past: Locality and memory in BDDs," in *Proc. IEEE-ACM Design Automation Conf.*, Anaheim, CA, June 1997, pp. 196–201.

[75] D. Sieling, "The nonapproximability of OBDD minimization," Univ. Dortmund, Tech. Rep. 663, 1998.

[76] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Santa Clara, CA, Nov. 1993, pp. 42–47.

[77] K. Ravi and F. Somenzi, "Hints to accelerate symbolic traversal," in *Correct Hardware Design and Verification Methods (CHARME'99)*, ser. LNCS 1703. Berlin, Germany: Springer-Verlag, Sept. 1999, pp. 250–264.

[78] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. TACAS'99*, Mar. 1999, pp. 193–207.

Melvin A. Breuer (S'58–M'65–SM'73–F'85) received the Ph.D. degree in electrical engineering from the University of California, Berkeley; is currently a professor of both Electrical Engineering and Computer Science at the University of Southern California, Los Angeles, California.

He is the Charles Lee Powell Professor of Electrical Engineering and Computer Science. He was chairman of the Department of Electrical Engineering-Systems from 1991-1994, and is currently chair once again. His main interests are in the area of computer-aided design of digital systems, design-for-test and built-in self-test, and VLSI circuits. He was chair of the faculty of the School of Engineering, USC, for the 1997-98 academic year.

Dr. Breuer is the editor and co-author of *Design Automation of Digital Systems: Theory and Techniques* (Englewood Cliffs, NJ; Prentice-Hall); editor of *Digital Systems Design Automation: Languages, Simulation and Data Base* (Los Alamitos, CA: Computer Science Press); co-author of *Diagnosis and Reliable Design of Digital Systems* (Los Alamitos, CA:Computer Science Press); co-editor of *Computer Hardware Description Languages and their Applications* (Amsterdam, The Netherland: North-Holland); co-editor and contributor to *Knowledge Based Systems for Test and Diagnosis* (Amsterdam, The Netherland: North-Holland); and co-author of *Digital System Testing and Testable Design* (Los Alamitos, CA: Computer Science Press, 1990; reprinted Piscataway, NJ: IEEE Press, 1995) . He has published over 200 technical papers. He was Editor-in-Chief of the *Journal of Design Automation and Fault Tolerant Computing*. He served on the editorial board of the *Journal of Electronic Testing* and was the co-editor of the *Journal of Digital Systems*. He was the Program Chairman of the Fifth International IFIP Conference on Computer Hardware Description Languages and Their Applications. He was co-author of a paper that received an honorable mention award at the 1997 International Test Conference. He was a Fullbright-Hays scholar (1972); received the 1991 Associates Award for Creativity in Research and Scholarship from the University of Southern California, the 1991 USC School of Engineering Award for Exceptional Service, and the IEEE Computer Society's 1993 Taylor L. Booth Education Award. He was the keynote speaker at the Fourth Multimedia Technology and Applications Symposium, 1999, and the Ninth Asian Test Symposium, 2000.

Majid Sarrafzadeh (M'87–SM'92–F'96) received the B.S., M.S. and Ph.D. degrees in 1982, 1984, and 1987 respectively from the University of Illinois at Urbana-Champaign in electrical and computer engineering.

He joined Northwestern University as an Assistant Professor in 1987. Since 2000, he has been a Professor of Computer Science, University of California, Los Angeles. His research interests lie in the area of VLSI CAD, design and analysis of algorithms and VLSI architecture. He has collaborated with many industries in the past ten years including IBM and Motorola and many CAD industries.

Dr. Sarrafzadeh is a Fellow of IEEE for his contribution to "Theory and Practice of VLSI Design." He received a National Science Foundation (NSF) Engineering Initiation award, two distinguished paper awards in ICCAD, and the best paper award for physical design in DAC for his work in the area of Physical Design. He has served on the technical program committee of numerous conferences in the area of Physical Design and CAD, including ICCAD, EDAC and ISCAS. He has served as committee chairs of a number of these conferences, including International Conference on CAD and International Symposium on Physical Design. He was the general chair of the 1998 International Symposium on Physical Design. He has published approximately 180 papers, is Co-Editor of *Algorithmic Aspects of VLSI Layout* (Singapore: World Scientific, 1994), Co-Author of *An Introduction to Physical Design* (New York: McGraw Hill, 1996), and the author of an invited chapter in the *Encyclopedia of Electrical and Electronics Engineering* in the area of VLSI Circuit Layout.

Dr. Sarrafzadeh is on the editorial board of the *VLSI Design Journal*, Co-Editor-in-Chief of the *International Journal of High-Speed Electronics*, an Associate Editor of *ACM Transactions on Design Automation (TODAES)*, and an Associate Editor of IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN.

Fabio Somenzi received the Dr. Eng. degree in electronic engineering from Politecnico di Torino, Torino, Italy, in 1980.

He was with SGS-Thomson Microelectronics from 1982 to 1989. Since then, he has been with the Department of Electrical and Computer Engineering of the University of Colorado, Boulder, where he is currently an Associate Professor. His research interests include synthesis, optimization, simulation, verification, and testing of logic circuits.