Multilevel Logic Synthesis

R. K. BRAYTON, FELLOW, IEEE, G. D. HACHTEL, FELLOW, IEEE, AND
A. L. SANGIOVANNI-VINCENTELLI, FELLOW, IEEE

A survey of logic synthesis techniques for multilevel combinational logic is presented. The goal is to provide more in-depth background and perspective for people interested in pursuing or assessing some of the topics in this emerging field. Introductions, capsule summaries, and, in some cases, detailed analysis, of the synthesis methods which have become established as practically significant are provided. Also included are some methods which have theoretical interest and potential for future impact.

I. Introduction

A long-term goal for computer-aided design (CAD) systems is the automatic synthesis from a behavioral description to silicon, producing near-optimal results that meet the specifications set by the designer and that are competitive with or better than manually aided designs. This capability will become increasingly important as the application-specific integrated circuit (ASIC) market continues to meet its rapid growth projections. The quality of such systems and the ability to quickly produce correct designs will be crucial for competitiveness in this market.

As various CAD areas have matured, they have provided algorithms and programs which then are improved, documented, supported, and made commercially available. Historically, this has happened with simulation and physical design. In physical design, automatic layout tools, placement and routing, cell editors, design rule checkers, extractors, etc. are widely available and widely used. Logic synthesis is the next higher level of abstraction. This area is at the knee of the commercial development curve; initial software offerings are available, and it is already evident that these are successful.

Logic synthesis fits between the register transfer level (RTL) specification of a digital design and the netlist of gates specification. It provides the automatic synthesis of near-optimal logic netlists, whether the goal is minimum delay, minimum area, or some combination. Logic synthesis is usually considered as dealing with all facets of pure combinational logic, including its optimization, design for testability, and verification. Logic synthesis is applied to the logic extracted from an RTL language. If the language includes storage constructs, these are usually set aside with their inputs and outputs being outputs and inputs, respectively, to combinational logic blocks. The resulting combinational logic blocks are operated on by the logic synthesis algorithms separately. Finally, the results are reconnected to provide a single overall design. During this process, information may be extracted about the environment in which a logic block is to operate. This may include signal arrival and required times, parasitics, and don't-care conditions. Logic synthesis is the problem of using this information to produce a correct implementation which meets timing and testability constraints and minimizes area.

After logic synthesis, the next level of abstraction is logic that includes memory devices, referred to as sequential logic. Although some systems leave elements in the memory when manipulating the combinational logic, little has been done, besides the application of a few rules, to treat memory on an equal basis with logic gates and to develop algorithms and theory for these types of networks. However, this is becoming an extensive research area and, in the next few years, we expect to see sophisticated commercial offerings for simultaneous synthesis of logic and memory.

The logic synthesis area is usually divided into two-level synthesis (PLA) and multilevel synthesis. Two-level logic minimization has been used to synthesize PLA's for control logic. Because of the architecture inherent to PLA's, optimization methods focus almost exclusively on minimizing the number of PLA product terms, which in turn minimizes the PLA area. The area of two-level combinational logic minimization has already matured. One can routinely find a minimum or near-minimum sum-of-products form for a logic function. These functions can be multiple output, incompletely specified, and functions with multiple-valued input variables. Functions with hundreds of inputs and outputs are within the realm of the algorithms. The optimization can also be done in a reasonable amount of computing time [22].

The other method for implementing logic, which is useful for both control and data-flow logic, is multilevel logic, sometimes called random logic. The design of random logic has as objectives:

Manuscript received June 1, 1989; revised November 6, 1989. This work was supported in part by NSF/DARPA grant MIP-8719546 and in part by DARPA under contract N00039-87C-0182.

R. K. Brayton and A. L. Sangiovanni-Vincentelli are with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

G. D. Hachtel is with the Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309.

IEEE Log Number 9035196.

- minimize overall layout area of the fabricated chip;
- minimize critical path delay time;
- maximize the testability of the synthesized logic, and provide a complete set of test vectors as a byproduct of the optimization.

Because of the increased potential for reusing sublogic, there are more degrees of freedom in the solution space than in the PLA case. Consequently, it has been much more difficult to synthesize this type of logic at a level competitive with manual synthesis.

However, in the past five years, the area of multilevel logic synthesis has blossomed. Not only is it a very active area of continuing research, but also the methods and algorithms developed thus far have been successfully adopted in commercially available products and in software available internally in the larger companies. CAD enterprises, such as Synopsys, Silc, Trimeter (now part of Mentor), VLSI Technology, and Silicon Compilers Systems, offer sophisticated multilevel logic-synthesis capabilities. Large companies, such as IBM, AT&T, NEC, and NTT, have a production code that has been used routinely for several years in chip synthesis.

A capsule history of the more recent developments in multilevel logic synthesis provides a contrast between two basic approaches adopted. It starts in the late 1970's with the development at IBM of the LSS system [36] using rule-based local transformations. The current LSS system, which has continued to evolve, is used in IBM production for the synthesis of many chips used in their medium and large computers. The local-transformation/rule-based methods use a set of *ad hoc* rules which are fired when certain patterns are found in the network of logic gates. A rule transforms a pattern for a local set of gates and interconnections into another equivalent one. Since rules need to be described, and hence must know about each gate type, the rule-based approach usually requires that the description of the logic be confined to a limited number of gate types, such as AND, OR, and NAND, or to those gates in a technology library for which the rules have been derived. In addition, the transformations have limited optimization capability since they are local in nature and do not have a global perspective on the design. Other examples of rule-based systems are those in use at NEC and Trimeter.

Beginning in about 1981, in parallel with and much influenced by activity in two-level logic synthesis, an approach evolved based on algorithmic transformations. The algorithmic point of view uses two phases: a technology-independent step based on algorithms for manipulating general Boolean functions [23] and a technology-mapping step where the design described in terms of generic Boolean functions is mapped into a set of gates that can be implemented in the design method of choice (gate-arrays, standard-cells, macro-cells). Both rule-based approaches and algorithmic approaches have been successful. Algorithmic systems are MIS [16], BOLD [5], [6], [10], [50], [22], [24], [93]–[95], and those used at Synopsys, Silc, AT&T, Eindhoven, and the University of California–Santa Cruz. As shown in this survey, a distinguishing feature for most of these systems is the extent to which they are able to exploit the degrees of freedom of the design problem in the optimization process.

Most logic synthesis systems divide the technology-independent phase of the design problem into two major subproblems:

1) create or modify the overall "architecture" of the given logic to produce a near-optimal "structure" where common sublogic is identified;
2) "optimize" the logic with respect to the structure obtained in Step 1—for example, make logic components optimal with respect to two-level minimization.

In the algorithmic approach, Step 1 is divided into algebraic and Boolean approaches. In Step 2, a major confluence occurs between optimal synthesis and testing.

Recently, there has been a trend toward combining the technology-independent activity and technology mapping, using the algorithmic methods in the initial stages of the synthesis, and the rule-based approach in the final stage when technology considerations are important. Examples of this combined approach are SOCRATES [5] and the more recent versions of LSS [8].

In this paper, we survey the algorithms and alternative approaches used, the representation of the logic, the quality of results obtained, the relation to other areas such as testing, and some of the frontiers of research currently being pursued. The goal is to provide background and perspective for people interested in pursuing or assessing some of the topics in more depth. We provide summaries of synthesis methods which have been established as being practically significant, as well as those which have theoretical interest and/or potential for future impact. Even though we tried to be complete, the description of the techniques of logic synthesis may be considered uneven at times because of the importance given to some approaches such as algebraic methods versus others such as rule-based methods. This bias is mostly due to our own experience in using the methods, reported in more detail here, for building the logic-synthesis systems MIS and BOLD.

The paper is organized as follows: in sections II and III we define basic notation and discuss the representation of combinational logic by an abstraction known as a Boolean network. Sections IV and V are treated at a technology-independent level of abstraction. Section IV treats the "creative" part of the logic synthesis, that of creating the basic, overall "architecture" of the multilevel logic. Section V treats the part most like two-level minimization, the task of optimizing the logic with respect to the given basic structure. In section VI we discuss means for defining and determining equivalence between Boolean networks and the relation with testing and redundancy removal. Section VII focuses on mapping the optimized technology-independent representation into a specified target technology. Section VIII gives an overview of the related rule-based methods.

II. Notation and Definitions

Logic, or Boolean, variables are denoted by lower case letters, e.g., $x_1, x_2, \cdots$ or $a, b, c \cdots$. A Boolean variable can take on just two values, 0 or 1. This is denoted by $B = \{0, 1\}$. It is common to refer to the statement "x has the value 1" simply as x and "x has the value 0" as $\bar{x}$. Then x and $\bar{x}$ are referred to as "literals." A logic function f is a function

of logic variables and has value in $\{0, 1\}$; written $f: B^n \rightarrow B$, where n is the number of logic variables.

One way of representing a logic function is as a "sum-of-products." A product, or "cube," is the product of literals, e.g., $a\bar{c}d$. Equivalently, we can think of a cube as a "set" of literals, e.g., $\{a, \bar{c}, d\}$. We often use the notation $l \in c$ to mean that the literal l is in the set (cube) of literals c, i.e., that l is one of the literals making up the product term c. Equivalently, a cube, e.g., $a\bar{c}d$, is the set of all points (sometimes called minterms or vertices) in the input space B^n that satisfy "$a = 1$ and $c = 0$ and $d = 1$." This set of points is called a cube because of its geometrical interpretation in the Boolean n-cube, B^n. Note that if the size of the input space is n variables, and a cube has k literals in it, then the number of vertices in the cube is 2^{n-k}. A "sum-of-products" is a set of cubes where it is understood that the function f it represents is obtained by summing (performing the logical OR) of all the points in all the cubes in the set. Such a function f is a "completely specified" logic function; it evaluates to 1 if the input vertex is in the set, to 0 otherwise.

Generally, a logic function f can be thought of as the set of all input points (minterms or vertices of B^n), which satisfy $f(v) = 1$; this set is referred to as the "on-set" of f. Similarly, the complement of a function, denoted by $\bar{f}$, is the set of vertices which satisfy $f(v) = 0$; this set is referred to as the "off-set" of f. In a more general situation, a logic function may be "incompletely specified," in that there is a set of vertices for which we do not care if the function has a value of 1 or 0. These "don't-care points" can be used to represent the function in a more compact form. An incompletely specified function is denoted by the triplet (f, d, r) of completely specified functions, a partition of B^n, where f is the on-set, d is the don't-care set, and r is the off-set. A "cover F" of an incompletely specified function is a completely specified function (typically in sum-of-products form) such that $f \subseteq F \subseteq f + d$. Said in another way, $f \subseteq F$ and $F \cap r = \phi$.

Any completely specified logic function can always be represented as a sum of products. A sum-of-products expression for a function is not unique. For example, the following function whose on-set is the set of vertices

$$\{\bar{a}\bar{b}\bar{c}, a\bar{b}\bar{c}, a\bar{b}c, abc, \bar{a}bc, \bar{a}b\bar{c}\}$$

can be represented in sum-of-products form as

$$a\bar{b} + ac + \bar{a}b + \bar{a}\bar{c}$$

or as

$$a\bar{b} + bc + \bar{a}\bar{c}.$$

The task of two-level logic minimization is to find a sum-of-products expression which is a cover for a given incompletely specified logic function and which has the least number of product terms.

We use the notation f_x to denote the logic function obtained from f by replacing x by 1; said differently, f_x is f evaluated at $x = 1$. This new logic function is called the "cofactor of f with respect to x." Similarly, $f_{\bar{x}}$ is obtained by replacing x by 0 and is called the "cofactor of f with respect to $\bar{x}$." For example, if

$$f = abx + \bar{a}c\bar{x} + \bar{c}d + ae$$

then

$$f_x = ab + \bar{c}d + ae$$

$$f_{\bar{x}} = \bar{a}c + \bar{c}d + ae.$$

Note that f_x and $f_{\bar{x}}$ are functions independent of the variable x. In general, a function f is independent of x if and only if $f_x \equiv f_{\bar{x}}$.

An "implicant" of a function is a product term (cube) q that is contained in $f + d$ and such that $q \cap r = \varnothing$. A "prime" (also called a prime implicant) p of a function is an implicant such that all the cubes that contain p have nonzero intersection with the off-set of the function, i.e., p cannot be enlarged as a product term (removing some literals) without including some of the off-set. Thus the cube abc is enlarged to the larger cube bc by dropping the literal a. This increases the number of minterms (vertices) in the space that are included in the cube. If all such new vertices are still outside the off-set, then the enlarged cube is still an implicant of the function. Thus a prime is a cube that is not contained in any other implicant of the function. In the preceding example, the product term abc is not a prime because it can be enlarged by expanding it to be bc without including any vertex in the off-set; the extra vertex included in the expanded cube is $\bar{a}bc$, which is also in the on-set of the function.

We briefly review some of the heuristics used in a two-level minimization program such as ESPRESSO [22]. There are three basic operations repeated in a loop: EXPAND, IRREDUNDANT-COVER, and REDUCE. EXPAND locates, with a heuristic process, the largest prime containing a given implicant of the Boolean function. The heuristic process maximizes the probability that other implicants will be completely covered by the selected prime. IRREDUNDANT-COVER removes a maximal set of nonessential primes. Both EXPAND and IRREDUNDANT-COVER remove literals or cubes from the logic function. After these two operations, the Boolean function is prime and irredundant, a local minimum in the synthesis process. The REDUCE operation is an "uphill" move which enables the optimization process to climb out of a local minimum and move closer to the global minimum during the next EXPAND and IRREDUNDANT-COVER cycle. REDUCE does this by replacing each prime implicant by a smallest implicant that covers all the essential vertices of the prime implicant. Since this adds literals, the associated logic cost of the implicant increases, but after REDUCE, EXPAND can be called to expand in different directions to possibly decrease the number of cubes.

A "multilevel implementation" of a function or a set of functions is one where an unlimited number of intermediate signals is allowed. In a two-level implementation, the only intermediate signals are product terms formed from the inputs. In multilevel, an intermediate signal may be the output of a two-level function whose inputs may also be outputs of other two-level functions. Generally, we can think of a multilevel implementation as an arbitrary interconnection of two-level functions, with the provision that the structure has no cycles in its dependency graph.

III. Representation of the Network and Nodes

A. Network Representation

A "Boolean network" is a directed acyclic graph. Associated with each node of the graph is a variable, y_i, and a representation of a logic function, f_i. A directed arc from node i to node j is in the graph if node j uses the variable y_i explicitly in the representation f_j. The set of variables that

f_j explicitly depends on is called the "support of f_j," denoted S_{f_j}. A node j is a "fan-in" of i if there is an arc from node j to node i. A node j is a "transitive fan-in" of i if there is a directed path in the Boolean network connecting j to i. A node j is a "fan-out" of i if there is an arc from i to j. A node j is a "transitive fan-out" of i if there is a directed path in the Boolean network connecting i to j.

Some of the nodes in the graph are designated as outputs to the network, called the "primary outputs." Any node that has an arc directed from it to another node is an intermediate node. A node can be both an output and an intermediate node.

A Boolean network is an implementation or representation of a set of incompletely specified Boolean functions. It is a representation in the same way that a PLA or sum-of-products form is a representation of a set of logic functions. The representation is not unique. For multilevel minimization, we seek a representation with several objectives. One is to minimize area. A good measure that seems to be well correlated with this is the total number of literals in all the function representations f_j at the nodes. Another objective is the delay through the network. In general, one is interested in implementing a set of functions which meet given delay constraints while minimizing area. The number of cubes in the representation, the primary objective for two-level minimization, is of interest for multilevel only as it correlates with the total number of literals.

Network Don't Cares: Don't cares are extremely important in minimizing logic. In minimizing multilevel logic, we assume (as with PLAs) that we are given an initial representation and a set of don't cares for each output ("external don't cares"). Generally, the don't cares common to all outputs are input patterns which will never occur. These may arise through the digital system specification, e.g., in a microprocessor design, certain instruction codes may not be used and therefore will never occur as a valid input. Another example occurs when one block of combinational logic is the input to another. The first block may have output bit patterns which will never occur because of the type of logic function being implemented. Since these outputs are inputs to the next block, the bit patterns which do not occur are don't cares for the second block of logic. In both cases, we can interpret the patterns, which never occur, as "states" that are not controllable. Using testing nomenclature, one says that the state is not "justifiable." In general, these don't cares occur because of the structure which appears before the input to a block of logic.

Those don't cares that are specific to the separate output functions usually arise from the way each output is used. If, because of the circuitry that fans out from a set of signals, the value of this set cannot be observed at prespecified observation points (true outputs), then the conditions under which the signals cannot be observed are don't cares for the signals. In the example of one combinational logic block feeding another, the second block serves as a filter for the first and can cause nonobservability of some of the outputs under certain input conditions. For example, suppose we have two blocks of logic, the first computing an arithmetic function and the second implementing an enable signal which controls whether or not the arithmetic result is latched at the outputs. Clearly the output of the arithmetic function is not observable under the conditions which disable the latch. Thus these are observability don't care conditions for the arithmetic logic block. In the parlance of the testing literature, one says that under these conditions the arithmetic logic is not able to "propagate."

We will see (cf., section V-F1) that a don't care set representing output usage is, in general, insufficient to capture this information completely. Equivalence relations have been proposed as a more general notion [20], [21]. This leads to the concept of "Boolean relations" discussed in section V-F1. However, since the use of don't cares is a much more developed area, in this paper we will continue the tradition, used in PLA synthesis, of using external don't cares to capture some of this information.

Unfortunately, the full set of don't cares is often not given. This is especially true if the logic has been designed manually, but it has also been true for logic specified in a high-level language. Recently, more effort has been directed toward identifying, extracting, and using don't cares in an environment where the logic is specified in a high-level language and synthesized using multilevel logic. We view this as a key development for the future.

Extracting don't cares: There are cases where the don't cares can be extracted automatically from the structure of the circuit being optimized. For example, if the design is fully specified at the logic level and consists of interconnected parts of logic which can be optimized separately, then the set of don't cares arising from the interconnection, as described in the preceding, can be assembled automatically. However, if the full structure of the design is not known, don't cares still can be extracted automatically from a hardware description language representation.

Often, hardware description languages (HDL's) provide the behavioral descriptions of combinational logic [88]. According to the principles of extracting Boolean networks equivalent to these HDL specifications, any primary input minterm should be regarded as a don't care condition if the primary output variable has not been assigned an expression during "execution" of the HDL model. This permits the modeler to save time by not having to specify logic for cases that will not occur, or will occur but will not be used. This idea permits the derivation of implied don't care functions associated with all variables in the HDL description. This don't-care set can be conceptually written:

$$d_k = \prod_j \bar{c}_j,$$

where $\bar{c}_j$, is the complement of the condition under which expression j is activated during "execution" of the HDL model. Thus the conditions under which j is not activated are implied to be don't cares.

Since this mechanism assumes that the HDL description is correct, it is important that the language processor issue a warning and produce information about the implied don't cares.

Boolean network equivalence, prime, and irredundant networks: Let a Boolean network with primary inputs *PI* and primary outputs *PO* be defined as $\eta(PI, PO)$. Two Boolean networks $\eta_1(PI, PO)$ and $\eta_2(PI, PO)$ are "equivalent" if for all values of corresponding primary inputs not in the don't care sets, the corresponding primary outputs are equal. A cube of an internal node of a Boolean network is "prime" if removal of any of its literals makes the Boolean network so obtained not equivalent to the original one. A cube of a cover of an internal node of a Boolean network is "redun-

dant" if the Boolean network obtained by removing the cube is equivalent to the original one. A Boolean network is said to be prime if all its cubes are prime, and irredundant if all its cubes are irredundant. In the case of a network which consists only of NAND's, only of NOR's, or of alternating AND/OR gates, then it is prime and irredundant if and only if it is 100-percent testable for all single stuck-faults.

B. Node Representation

Each node of a Boolean network has associated with it a representation of a logic function. The question of how this function is represented is important. Although any valid representation is allowed, some representations may be preferred because they are

- more efficient in memory
- more indicative of the complexity of the final implementation
- more efficient to manipulate.

In this section we survey some of the choices available.

In two-level theory, these issues don't arise since the representation and the final implementation are the same, namely the sum-of-products form. However, for multilevel, there are a number of choices, and which of these is best is still debatable.

Merged view—The network is represented so that each node is a valid "gate" chosen from a library of gates to be used in the final implementation. Thus representation and implementation are one. The advantage of this is that as each change is made to the network, one can accurately evaluate its effect on the implementation in terms of area and delay.

Separated view—Two representations are allowed. One is technology independent, i.e., it does not have any connection with the final building blocks to be used in the implementation. The other is the technology-dependent view which uses only "valid" gates, i.e., those in a cell library or meeting some criterion.

In the technology-independent view, there are also several choices.

General node—Each node can be a representation of an arbitrary logic function. A possible advantage of this is that a theory can be developed more easily.

Generic node—Every node in the network is the same function, e.g., a two-input NAND gate. The advantage is that each node is very simple. There is no need to store a general logic function at a node since each node is the same function and only the inputs are different. Although there can be many more nodes than required for the general node description, some manipulations are much faster using this structure. The disadvantage is that the network is finely decomposed in a particular way, and this may obscure some natural structures in the network.

Discrete node—A node can be one of a small set of logic functions, such as AND, OR, NOT, DECODE, ADD. Multiple output nodes are also allowed. Generally, this type of representation is used only in rule-based systems. One advantage is that complex blocks of logic, like a DECODE function, can be kept grouped together and manipulated as a single unit. However, a general theoretical basis for such networks seems much more difficult.

For the majority of this paper, we use the general node representation since (except for multiple-output nodes) it includes all others as special cases. A more complete theory and body of algorithms has been developed for this point of view.

1) Sum-of-Products: The most obvious representation for the general node is the sum-of-products form. This is the one most used as the nominal representation in Boolean networks, possibly because of the influence from PLA optimization problems. This is a natural choice mainly because there are highly developed techniques for manipulating logic in this form, e.g., two-level minimization, factoring, decomposition, tautology, and combining logic functions using logic operations like AND, OR, etc. Even though we may prefer to have logic represented some other way, present techniques generally require conversion to sum-of-products, manipulation with the developed algorithm, and conversion back. Thus one can argue that this should be the nominal representation.

2) Factored Forms: Factored forms are probably a more natural representation for multilevel synthesis. Roughly, a factored form is a parenthesized expression, e.g.,

$$(ab + \bar{b}c)(c + \bar{d}(e + a\bar{c})) + (d + e)(fg).$$

An argument for factored forms is that they are a natural multilevel representation. A factored form is isomorphic to a tree structure, where each internal node is an AND or OR operator and each leaf is a literal. This leads to a simple and relatively efficient multilevel implementation of the function of the node. A representation which accurately measures complexity is important in guiding the synthesis process, since synthesis can be seen as a sequence of transformations which may or may not be accepted, depending on the quantity of complexity decrease obtained. Factored forms have this property while still providing a technology-independent representation.

The count of the number of literals in a factored form is well correlated with the complexity of the function and can be translated directly into the number of transistors required for an implementation. Of course, this only indirectly measures area since wiring is also an important contribution to the total area. It has been suggested that a better area estimator would be the number of literals in the factored form plus a term proportional to the number of gates or nodes, or the number of terminals in the network. However, experiments show that literal count still has a remarkably good correlation with the total layout area.

Another argument favoring factored forms over the sum-of-products is that the factored form implicitly represents both the function and its complement. A complement factored form can be obtained directly by applying DeMorgan's law to the factored form. Thus AND's are converted to OR's, and vice versa, and literals are negated. This produces a factored form for the complement which has the same literal count. This result coincides with the notion that in a multilevel implementation, a function and its complement are almost equally complex, separated only by the cost of an inverter. This is in contrast with the sum-of-products form, where the number of cubes in a function can be exponentially larger than in its complement. In this regard

we may think of the factored form as representing both the function and its complement, similar to the binary decision diagram discussed in the following.

It was noted that a factored form is an AND/OR tree. Thus, if each general node is decomposed into an AND/OR tree, then we have a network in the discrete node representation. The only distinction is that the general node serves as a cluster for a subset of discrete nodes (the tree).

The difficulty with factored forms is that methods for manipulating them have not been highly developed. However, this has stimulated the development of factored-form manipulation methods similar to recent extensions in methods for sum-of-products manipulation. Three such efforts have been reported recently. The first [11] is motivated by the generic representation point of view, which leads to more efficient storage as well as possibly faster methods for manipulation. This representation is basically a type of factored form. In [11], methods for finding common factors and methods for logic minimization were proposed. In a second effort [97], the standard factored form representation is used, where a node is either an AND or an OR, and most of the Boolean function manipulation methods are extended. However, a method for logic minimization is missing. A third development [67] starts from a sum-of-products form and asks for a minimization procedure which has as its goal a minimal factored form. Here it is recognized that the minimal number of cubes, the normal goal of two-level minimizers (such as ESPRESSO), is inappropriate. A minimizer based on a minimal factored form has been developed.

Another lack in this area is some notion of optimality. Is a given factored form optimum? In the case of sum-of-products there is an effective answer via some form of Quine–McCluskey exact minimization [72]. However, for factored forms the only known optimality result [65] is not practical for functions which depend on more than about six to eight variables.

3) BDD's: Binary decision diagrams (BDD's) are a relatively new and extremely important contribution to logic synthesis [29]. BDD's have increased in importance recently as more applications have been discovered. Generally, one should think of using a BDD whenever an algorithm is described in terms of a truth table. Like a truth table, the BDD is a canonical representation of a completely specified logic function. Recently, these notions have been extended to include incompletely specified logic functions [71].

A BDD is a directed acyclic graph (DAG) representation of a logic function. To help explain the BDD, an example is shown in Fig. 1 of a BDD representing the function $ab + c$. There is one root node (labeled a in the figure) and two leaf nodes, 0 and 1. The root node represents the entire function and the two leaf nodes represent the functions 0 and 1. Each nonleaf node has a variable associated with it (shown inside each node). Each nonleaf has two successors. The first successor points to a node representing the function cofactored with respect to the negative phase of the node variable, the second successor points to the function cofactored with respect to the positive phase. In the figure, the branch labeled $a = 0$ points to a sub-BDD which represents $f_a = c$ and the branch labeled $a = 1$ represents the function $f_a = b + c$. All the variables are ordered (thus the notion of an "ordered" BDD). The ordering imposes the constraint that each successor node must have a variable associated with it that is greater than any of its predecessor variables. In the example, the order is a, b, c. Note that any path from the root to either leaf visits the nodes whose variables are in the proper order (although a variable may be skipped).

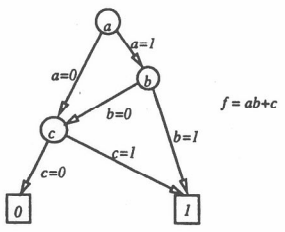


Fig. 1. BDD representing function $ab + b$ using variable ordering a, b, c.

Thus each nonleaf node implicitly represents some Boolean function of those variables whose order is greater than or equal to the order of the variable at the node. The BDD is forced to be "reduced" in the sense that if two internal nodes represent the same function, then they must be the same node. Of course the number of nodes can be exponential, but it has been observed that if the ordering is chosen correctly, this exponential explosion rarely occurs in practical functions. Finding an optimum ordering is extremely difficult; however, good heuristic orderings have been given [70]. It has been demonstrated, since BDD's are canonical given an ordering, that BDD's with good orderings provide a very effective way of verifying that two Boolean networks are equivalent.

Bryant [29] has shown how most logic operations on BDD's can be done in linear or log linear time measured in terms of the number of nodes in the BDD.

An improvement of the standard BDD is to use a negative pointer. A regular (positive) pointer indicates the successor node function, whereas a negative pointer implies the complement of the indicated successor function. This allows the combining of a function and its complement into the same DAG [59]. For example, if one node has a successor g and another has a successor which is the function $\bar{g}$, then instead of using two different nodes to represent these different functions, only one node is necessary if one of the predecessors uses a negative pointer. It has been demonstrated that this idea saves substantial storage without any noticeable penalty in run-time.

BDD's are currently used in verifying if two multilevel networks are equivalent. The technique is simple since a BDD is canonical. Each output of a network is reduced to a BDD over the input variables. Two output functions are equivalent if and only if their BDD's are isomorphic. Checking isomorphism of BDD's is extremely fast. BDD's have also been used to provide an initial multilevel decomposition of a network using the one-to-one mapping from a BDD to a multiplexor decomposition. Each node in a BDD maps into a multiplexor controlled by the node variable. The other two inputs are the outputs of the successor nodes. For example, if y_x and $y_{\bar{x}}$ represent the outputs of the successor nodes, then the node function is the multiplexor function

$$xy_x + \bar{x}y_{\bar{x}}.$$

In recent work by Muroga [78], an initial decomposition is obtained by a procedure similar to this.

4) Multivalued Decision Diagrams and Incompletely Specified Functions: More recently, BDD's have been used, in conjunction with Muroga's method of transduction (cf. section V-C), in a logic synthesis system at Fujitsu [71]. Here it is necessary to extend the BDD so that it has three leaf nodes, 0, 1, and don't care. The transduction method computes a compatible set of permissible function (CSPF) at each node, which is an incompletely specified function. These are used to identify redundancies and to substitute one function into another.

BDD's have been extended also to include multivalued variables and multivalued outputs (MDD's) [91]. Instead of each node having two successors, a node associated with a multivalued variable has up to p successors, where p is the number of possible values of the multivalued variable associated with the node. As before, each node in the MDD uniquely represents a multivalued logic function. A graph is reduced if no two nodes represent the same function. It has been shown that the MDD is a canonical representation, and that most of Bryant's results for BDD's extend quite easily and naturally. The expectation is that this extension of the BDD's will gain importance as multivalued multilevel functions become more important in future developments [69].

5) If-Then-Else DAG's: Another generalization of BDD's proposed by Karplus [59] is called *if-then-else* DAG's. In contrast to the BDD, each internal node has three outgoing pointers. The first, the *if* part, is another *if-then-else* DAG and hence represents an arbitrary Boolean expression. The second, the "true" part, is the one taken whenever the *if* expression evaluates to true. The third, the *else* part, is the one taken whenever the *if* expression evaluates to false. Thus the node test in the BDD is a single variable whereas in the *if-then-else* DAG it is an arbitrary Boolean expression. Karplus gives seven rules for constructing the *if-then-else* DAG which will insure that two DAG's are equal if and only if their Boolean expressions are also equal (i.e., it is canonical). This structure is an interesting generalization of BDD's and it remains to be seen how effective it is in various applications, although the concept has already led to more efficient methods for constructing the regular BDD's.

IV. Logic Decomposition/Restructuring

The objective of multilevel logic synthesis is to find the best multilevel structure. Often the logic to be implemented is extracted directly from a register transfer language (RTL) and thus has a natural multilevel form. This may or may not be the best structure, but it is important not to destroy it (e.g., arbitrarily flatten it to two-level) until it is assessed. On the other hand, some logic, particularly control and finite-state-machine logic, is more naturally described in two-level form and no initial structure is given. In either case, we have the problem of finding the best multilevel structure, but in the first case we may have an advantage in that the user may have given a good multilevel structure by virtue of how it was structured in the RTL input. This section is concerned with various techniques which allow us to restructure the initial logic description. The methods are divided into algebraic methods, which are fast, and Boolean methods, which are slower (at times, much slower) but have the power to explore the entire restructuring space in a more general way.

A. Basic Operations

The goal of multilevel logic optimization is to obtain an equivalent representation of a given logic function that is optimal with respect to a cost function involving area and delay. In manipulating the initial representation of the logic function, the following five operations are key.

The "decomposition" operation on a Boolean function is the process of re-expressing a single function as a collection of new functions. For example, the process of translating

$$F = abc + abd + \overline{ac}\,\overline{d} + \overline{b}\,\overline{c}\,\overline{d}$$

to

$$F = XY + \overline{X}\,\overline{Y}$$

$$X = ab$$

$$Y = c + d$$

is decomposition. Note that the fan-in F (the variables on which F depends explicitly) was altered by this operation.

A related operation, but applied to many functions, is the "extraction" operation. It is the process of identifying and creating some intermediate functions and variables, and re-expressing the original functions in terms of the original as well as the intermediate variables. There is significant practical difference between this and the decomposition operation. For example, extraction applied to the following three functions

$$F = (a + b)cd + e$$

$$G = (a + b)\overline{e}$$

$$H = cde$$

yields

$$F = XY + e$$

$$G = X\overline{e}$$

$$H = Ye$$

$$X = a + b$$

$$Y = cd$$

where multiple-fan-out nodes X and Y have been created.

This operation identifies common subexpressions among different logic functions forming a network. New nodes are created, but each of the logic functions in the original network is simplified as a result of the introduction of the new nodes. The optimization problem associated with the extraction operation is to find a set of intermediate functions such that the resulting network is optimal in an appropriate sense.

"Factoring" is the process of deriving a factored form from a sum-of-products form of a function. For example,

$$F = ac + ad + bc + bd + e$$

can be factored to

$$F = (a + b)(c + d) + e.$$

The associated optimization problem is to find a factored form with the minimum number of literals. In this case, we simply change the representation of the function.

"Substitution" (also called "resubstitution") of a function G into F is the process of expressing F as a function of

its original inputs and G. For example, substituting

$$G = a + b$$

into

$$F = a + bc$$

produces

$$F = G(a + c).$$

This operation creates an arc in the Boolean network connecting the node of the function being substituted (G) to the node of the function being substituted into (F).

"Collapsing" (also called "elimination" or "flattening") is the inverse operation of substitution. If G is a fan-in of F, collapsing G into F re-expresses F without G (undoes the operation of substituting G into F). For example, if

$$F = Ga + \bar{G}b$$

$$G = c + d$$

then, collapsing G into F results in

$$F = ac + ad + b\bar{c}\bar{d}$$

$$G = c + d.$$

If the node G is not an output, it may be eliminated, resulting in a Boolean network with one less node.

All of the operations use techniques that are analogous to multiplication and division. In fact, "division" plays a key role in multilevel logic optimization. In this section, the concept of division as well as effective algorithms for division are reviewed. Algorithms for factorization, decomposition, extraction, substitution, and collapsing, based on these results, are presented.

B. Division and Common Divisors

Since Boolean algebra does not have additive or multiplicative inverses, in mathematical terms there can be no division operation. However, in optimizing logic functions, it is important to define operations which, when given functions f and p, find functions q and r such that $f = pq + r$. Every such operation is similar to the division operation in other algebras and is therefore called, with a little abuse of mathematical terms, "division" of f by p generating "quotient q" and "remainder r." The function p is called a "Boolean divisor" of f if r is not null and a "Boolean factor" if r is null. Such a division operation is not unique. Even for a given division operation, the resulting q and r may be dependent upon the particular representation of f and p.

The number of Boolean factors and divisors of a given logic function can be very large, as made evident by the following Propositions:

Proposition 1: A logic function g is a Boolean factor of a logic function f if and only if $f\bar{g} = \emptyset$, i.e., $f \subseteq g$.

Proposition 2: If $fg \neq \emptyset$, then g is a Boolean divisor of f.

The two propositions show that for any logic function f there are many Boolean divisors and factors; in fact, any function containing f is a Boolean factor of f and any function with at least one miniterm common with f is a Boolean divisor of f. This poses a problem in choosing a best factor since there are so many factors. If the domain is restricted to a particular subset of expressions, then the division operation is unique and much easier and faster to carry out. This restricted version of division is called "algebraic division." The following definitions make this notion precise.

The "product" of two cubes c and d is a cube defined by (recall that a cube can be viewed as a set of literals)

$$cd = \begin{cases} 0 & \text{if } \exists x (x \in c \cup d \text{ and } \bar{x} \in c \cup d) \\ c \cup d & \text{otherwise} \end{cases}$$

The "product" of two expressions F and G is a set defined by

$$FG = \{cd \mid c \in F \text{ and } d \in G \text{ and } cd \neq \emptyset\}.$$

Notice that $cd = 0$ if and only if $c \cup d$ contains both a literal and its complement.

We say that F is an "algebraic expression" if F is a set of cubes such that no one cube contains another: e.g., $a + ab$ is not an algebraic expression since cube a contains cube ab.[1] FG is an "algebraic product" if F and G are algebraic expressions and have disjoint support (that is, they have no input variables in common). Otherwise, FG is a "Boolean product." For example, $(a + b)(c + d) = ac + ad + bc + bd$ is an algebraic product and both $(a + b)(a + c) = a + ab + ac + bc$ and $(a + b)(\bar{b} + c) = a\bar{b} + ac + bc$ are Boolean products.

An operation (OP) is called "division" if, given two functions f and p, it generates q and r (OP(f, p) = (q, r)) such that $f = pq + r$. If pq is an algebraic product, OP is called an "algebraic division;" otherwise pq is a Boolean product and OP is therefore called a "Boolean division." Note that an algebraic divisor (factor) is also a Boolean divisor (factor).

C. Algebraic Methods

Decomposition based on Boolean manipulations can be quite expensive computationally, but in principle can achieve optimum results. On the other hand, the algebraic manipulations can be made much faster and, especially when iterated with selective collapsing operations, can give very good results. One task of logic synthesis is to decide when to use each kind of manipulation in order to obtain a good combination of run-time efficiency and quality of results.

This leads to the most often used paradigm for multilevel logic synthesis [4], [14], [16], [46]:

- minimize each logic function to obtain an algebraic expression,
- perform algebraic operations, including decomposition, extraction, factorization, resubstitution, and elimination, on these expressions,
- optionally iterate steps 1 and 2.

These operations may be enriched with a few Boolean operations that improve the overall result without penalizing the running-time efficiency of logic optimization.

The next three sections review the basic algorithms used to perform algebraic operations. Section IV-G covers Boolean operations.

1) Algebraic Division and Its Complexity: In general, we face two tasks in using either notion of division. The first

[1]The containment of a cube c_1 by another cube c_2 is confusing if we view each cube as a set of literals. We shall always refer to one cube containing another if the set of "minterms" in one contains the set of minterms in the other.

is to find a good candidate divisor, and the second is to carry out the division, i.e., to determine, given p and f, the quotient q and remainder r so that $f = pq + r$.

Care should be taken to make this algorithm as fast as possible since it will be used many times in the inner loops in a logic synthesis system and is a key subroutine of many of the other algorithms.

"Weak division" is a specific example of algebraic division. As far as we know, it is the only form of algebraic division used. It has the virtue of making the result (quotient and remainder) unique. The name "weak" refers to its power in relation to Boolean division (also called strong division). Given two algebraic expressions f and p, a division is called "weak division" if

1) it generates q and r such that pq is an algebraic product,
2) r has as few cubes as possible, and
3) $pq + r$ and f are the same expression (having the same set of cubes).

Given the expressions f and p, it can be shown that q and r generated by weak division are unique. WEAK_DIV denotes the operation of weak division. Often, f/p is used to denote the quotient of "weak-dividing" f by p. In Fig. 2

```
WEAK_DIV(f,p):
U =Set { u_j } of cubes in f with literals not in p deleted.
V =Set { v_j } of cubes in f with literals in p deleted.
/* note that u_j v_j is one the j-th term of f */
V^i = { v_j ∈ V : u_j = p_i }.
q = ∩ V^i.
r = f − pq
```

Fig. 2. Algorithm WEAK_DIV.

is a sketch of an $O(n \log n)$ (n is the number of product terms in f and p) algorithm proposed for weak division [23]. This algorithm achieves its $n \log n$ performance by encoding and ordering the terms in U and V. McGeer found a linear algorithm for weak division given that expressions f and p have their cubes already encoded and sorted [74]. It was shown that algorithms could be found which are linear and produce their results as a set of cubes in sorted order. Thus an initial sorting of the cubes of all functions at the beginning of the network manipulation would suffice. Thereafter, linear algorithms could be employed.

D. Kernels and Kernel Intersections

1) Basic Definitions: The notion of a kernel of an algebraic expression was introduced in [23] to provide means for finding subexpressions common to two or more expressions, i.e., to find good candidate divisors. All operations used to find kernels are algebraic (i.e., algebraic product, algebraic division, etc.), but the word "algebraic" is omitted for brevity. In particular, algebraic division is done by WEAK_DIV.

An expression is "cube-free" if no cube divides the expression evenly (e.g., $ab + c$ is cube-free; $ab + ac$ and abc are not cube-free). Notice that a cube-free expression must have more than one cube.

The "primary divisors" of an expression f are the set of expressions

$$\mathcal{D}(f) = \{ f/c \mid c \text{ is a cube} \}.$$

The kernels of an expression f are the set of expressions

$$\mathcal{K}(f) = \{ g \mid g \in \mathcal{D}(f) \text{ and } g \text{ is cube-free} \}.$$

In other words, the kernels of an expression f are the cube-free primary divisors of f.

A cube c used to obtain the kernel $k = f/c$ is called a "co-kernel" of k, and $\mathcal{C}(f)$ is used to denote the set of co-kernels of f. For example, the kernels and their corresponding co-kernels of the function

$$x = adf + aef + bdf + bef + cdf + cef + g$$
$$= (a + b + c)(d + e)f + g$$

are listed in Table 1, where for convenience we have shown the kernels in factored form. Notice that a kernel may have

Table 1 Kernels and Co-Kernels of $(a + b + c)(d + e)f + g$

Kernel	Co-Kernel	Level
$a + b + c$	df, ef	0
$d + e$	af, bf, cf	0
$(a + b + c)(d + e)$	f	1
$(a + b + c)(d + e)f + g$	1	2

more than one co-kernel even though the kernel of a co-kernel is unique. A co-kernel can be the trivial cube 1 if the original expression is cube-free.

For certain operations described in the following sections, it is nearly as effective and frequently more efficient to compute a certain subset of $\mathcal{K}(f)$ rather than the full set. This leads to the following recursive definition. Let

$$K^n f = \begin{cases} \{k \in \mathcal{K}(f) \mid \mathcal{K}(k) = \{k\}\} & n = 0 \\ \{k \in \mathcal{K}(f) \mid \forall k_1 \in \mathcal{K}(k), \text{ such that } k_1 \neq k \text{ and} \\ \quad k_1 \in \mathcal{K}^{n-1}(f)\} & n > 0 \end{cases}$$

Using these sets, we define the "level" of a kernel as follows. If $k \in \mathcal{K}^0(f)$, then k is a level-0 kernel of f. If $k \in \mathcal{K}^n(f)$ and $k \notin \mathcal{K}^{n-1}(f)$, then k is a level-n kernel of f. According to the definition, a kernel is said to be of level-0 if it has no kernels except itself. Similarly, a kernel is of level-n if it has at least one level n-1 kernel but no kernels (except itself) of level-n or greater. This gives us a natural partition of the kernels since

$$\mathcal{K}^0(f) \subset \mathcal{K}^1(f) \subset \mathcal{K}^2(f) \subset \cdots \subset \mathcal{K}^n(f) \subset \mathcal{K}(f).$$

2) Computing the Kernels: All the kernels of a given function f can be found by applying the definition in a straightforward way. The kernel-generation algorithm proposed by Brayton and McMullen [23] makes f cube-free first by finding its largest cube-factor. It then selects the literals of f in lexicographical order and divides them into f; the resulting quotient is a kernel if it is cube-free. If it is not, then it is made cube-free by selecting its largest cube-factor. The procedure is repeated on the resulting functions until functions with no kernels (kernel of level-0 of f) are found. A major efficiency can be obtained by noting that if the largest cube factor extracted contains an already selected literal, then the current branch can be terminated, since all kernels

that can be found by continuing have already been generated. This leads to an algorithm in which no co-kernel is duplicated and which is quite simple [24].

3) Fundamental Theorem: The following theorem is key.

Theorem 4.3: If two expressions f and g have the property that any $k_f \in \mathcal{K}(f)$ and any $k_g \in \mathcal{K}(g)$, implies that $|k_g \cap k_f| \leq 1$ (k_g and k_f have at most one term in common), then f and g have no common nontrivial algebraic divisors. (A nontrivial divisor has at least two terms).

This theorem is used for detecting if two or more expressions have any common algebraic divisors other than single cubes. This can be done by computing the set of kernels for each logic expression, and forming nontrivial (more than one term) intersections among kernels from different functions. If this intersection set is empty, then we need only look for divisors consisting of single cubes (which is an easier task). In other words, we need not compute the set of all algebraic divisors for each expression to determine if there are common nontrivial algebraic divisors. This leads to great run-time efficiency since the set of kernels is much smaller than the set of all algebraic divisors, and secondly, in the algorithm for computing kernels, the cube-free property of kernels leads to a very effective method for trimming the search tree for the kernels. On the other hand, if we find a nontrivial intersection, then this is a candidate algebraic divisor common to two or more functions. Knowledge of which functions these kernels came from and which co-kernels were used allows us to assess the value of this potential divisor.

E. Algebraic Methods for Logic Operations

The operations of extraction, decomposition, factoring, and substitution can be carried out quite effectively in the algebraic domain using weak-division and kernels. In this subsection, proceeding in increasing complexity, we present algorithms for substitution, then factoring, decomposition, and finally, extraction.

1) Substitution: "Algebraic substitution" consists of the process of dividing the function f_i at node i in the network by the function f_j (or by $\bar{f}_j$) at node j. During substitution, if f_j is an algebraic divisor of f_i, then f_i is transformed into

$$f_i = hf_j + r;$$

similarly for $\bar{f}_j$. In practice, we attempt this for each pair f_i, f_j in the Boolean network, implying as many as $2n^2$ algebraic divisions, if there are n nodes in the network.

The following observations are trivial but important in circumventing most of these divisions. The function f_j is not an algebraic divisor of f_i if

1) f_j contains a literal not in f_i,
2) f_j has more terms than f_i,
3) for any literal, the number of times it occurs in f_j exceeds that in f_i,
4) f_i is in the transitive fan-in of f_j.

In some cases, we are not interested in the result of division if the quotient f_i / f_j is only a single cube. This can be detected by another useful filter: if for any literal the count for f_j equals the count for f_i, then (f_i / f_j) is, at most, a single cube.

2) Factorization and Decomposition: The definitions of factoring and decomposition, as given in section IV-A, show that the basic operations involved are the identification of a divisor and division of a function by that divisor. Decomposition is basically identical to factoring except that divisors yield new nodes in the Boolean network, and as such can fan-out and be used in their negative phase.

The problem of "optimum" factoring and decomposition has been the object of intense study in the past, but the number of proposed techniques which are practical for large networks (e.g., more than 1000 gates) is limited. The techniques reviewed here are the optimum NAND-gate synthesis of Dietmeyer and Su [40], and the algebraic approach of [23] and [15].

One of the first techniques practical for large circuits is the factoring technique of Dietmeyer and Su [40]. Their technique starts with a minimized sum-of-products representation of a single-output function. The factors considered are "single-cube factors." The single-cube factor is identified from the representation of the logic function by choosing a "common factor subarray" and "common factor" which maximizes the "figure-of-merit." The figure-of-merit is the width of the cube factor (number of literals) times the height of the common factor subarray (number of cubes having these literals as a subset). Three techniques are given for implementing the common factor and the common factor subarray using NAND-gates. All three are evaluated for the common factor which maximizes the figure-of-merit and the one requiring the fewest gates is chosen. The evaluation function for each implementation counts the number of inverters and bounded fan-in gates needed to realize the circuit, assuming this common factor is chosen. Dietmeyer and Su proposed two algorithms for finding the common factor and common factor subarray: one which finds the common factor with a maximum figure-of-merit, and a heuristic algorithm which rapidly finds a factor with a good figure-of-merit.

The primary limitation of Dietmeyer–Su factoring is that common factors which consist of more than one cube are not considered. While it is possible to find multiple-cube factors during common-cube extraction, nothing in the heuristic cost function for a common factor guides the selection toward these factors.

The technique proposed in [23] and [15] is based on kernels and finds multiple-cube factors. There are several incarnations of this idea; however, they can all be represented by the generic algorithm shown in Fig. 3. Given a

```
GFACTOR(F) {

If F has no factor, return
D =DIVISOR(F)
(Q,R) =DIVIDE(F,D)
return GFACTOR(Q)GFACTOR(D)+ GFACTOR(R)
}
```

Fig. 3. Basic factorization algorithm.

function F, procedure DIVISOR(F) finds a candidate divisor, D, which, when substituted into F, simplifies the expression. The quotient Q is found by dividing D into F according to the division procedure DIVIDE(F, D). Various options for the procedures DIVISOR and DIVIDE are discussed in the

following. The function F can thus be represented as a partially factored form

$$F = QD + R$$

where R is the remainder. The basic algorithm then proceeds to recursively factor Q and D using the same method. (This basic procedure can be made "optimal" in the sense that it will produce "maximally" factored forms if minor modifications are applied [16], [18], [97]).

The various forms of the algorithm are obtained by choosing different subalgorithms to implement the routines DIVISOR and DIVIDE. For DIVIDE, algebraic division is often used even though it may not be as effective as Boolean division. Algebraic division is, of course, much faster than its Boolean counterpart.

The DIVISOR routine selects kernels or other divisors of F. Its versions differ in the care with which the divisor is chosen. The simplest algorithm, QUICK_DIVISOR, or QD for short, quickly selects just one level-0 kernel. When QD is substituted for DIVISOR in GFACTOR, the resulting procedure is called QUICK_FACTOR, or QF. Since QD finds an arbitrary level-0 kernel, the quality of the final result produced by QF is suspect. However, this can be improved by performing a second division by the quotient made cube-free to obtain the candidate factor [12].

A more careful choice of divisor leads to the algorithm BEST_KERNEL, which greedily selects the kernel (k) which, when substituted into F, maximally reduces the total number of sum-of-product literals of F and k. The procedure obtained by substituting BEST_KERNEL for DIVISOR in GFACTOR is called GOOD_FACTOR, or GF. Since BEST_KERNEL finds all the kernels, GF represents a trade-off of speed for obtaining better quality results.

These factoring algorithms are obviously heuristic, since the procedure cannot be guaranteed to generate optimum results with respect to the cost function selected. The quality of the factoring is monitored by computing the literals in the sum-of-products form of each factor. Since the choice of common divisor is restricted to kernels only, the results of factoring depend largely on the initial sum-of-product forms.

In multilevel minimization as performed in the system MIS [15], factoring algorithms are used repeatedly to estimate the cost of a Boolean network, since the cost of the nodes is estimated to be the number of literals in the factored form of the node functions. Here the speed of the algorithm is essential and QF is favored. However, toward the end of the minimization process, when it is important to have an accurate evaluation of the cost of the network, GF is used.

Note that Dietmeyer–Su's procedure is a special case of GFACTOR where the DIVISOR routine searches for single-cube divisors only.

Other versions of GFACTOR would involve Boolean operations, which are discussed in section IV-G.

3) Extraction: The extraction operation identifies common subexpressions and manipulates the Boolean network accordingly. Algebraic decomposition and substitution can be combined to provide an effective extraction algorithm.

In particular, procedure QUICK_DECOMPOSITION applies QF to a given node and creates a new node in the network, for each new factor of this node provides a very fast method for breaking down a Boolean network quickly.

```
QUICK_EXTRACTION(F) {

Apply QUICK_DECOMPOSITION to each node of the network

Perform all possible pairwise algebraic substitutions

Eliminate all single literal functions

Eliminate all functions with small value

}
```

Fig. 4. Quick extraction algorithm.

It may be combined with algebraic substitution to form a fast extraction procedure, as shown in Fig. 4.

At the end of the QUICK_DECOMPOSITION step, each node of the network cannot be factored, so each literal appears only once. Substitution identifies identical nodes, and one is substituted into the other, leaving a node whose logic function is a cube with a single literal. These are eliminated along with the nodes that have small value, typically those which do not fan-out.

The motivation behind this is that QF is very fast but still identifies good kernels for factoring each single function well. The kernels become nodes of the Boolean network and substitution identifies common nodes. Thus common divisors identified in this way are also near best for factoring. Of course, this is not always the best choice and not all common divisors are found, but the method is very fast and the results are quite good.

F. Rectangle Covering

The key problem in the algebraic operations presented in the preceding is the identification of a divisor. We have seen that kernels offer a good set of divisors, both for factoring (or decomposition) and extraction. It is surprising that the problem of finding a kernel and, generally, finding a common single- and multiple-cube divisor, can be reduced to the same mathematical problem [16]–[18], [82]. In addition to being elegant, this formulation favors the development of fast and effective algorithms.

In this subsection, the concepts of rectangles and rectangle covering are introduced. Then the formulation of kernel determination and the common subexpression identification in terms of rectangles are given. Finally, some algorithms for rectangle covering are given. In this subsection, we closely follow [82].

1) Basic Definitions: A "rectangle" (R, C) of a matrix B, $B_{ij} \in \{0, 1, *\}$[2] is a subset of rows R and a subset of columns C such that $B_{ij} \in \{1, *\}$ for all $i \in R$, $j \in C$.

A rectangle (R_1, C_1) is said to "strictly contain" rectangle (R_2, C_2) if $R_2 \subseteq R_1$ and $C_2 \subset C_1$ or $R_2 \subset R_1$ and $C_2 \subseteq C_1$.

A "prime rectangle" (R, C) of B is a rectangle which is not strictly contained in any other rectangle of B.

The "co-rectangle" of a rectangle (R, C) is the pair (R, C'), where C' is the set of columns not in C.

A set of rectangles $\{(R^k, C^k)\}$ form a "rectangle cover" of a matrix B if $B_{ij} = 1$ implies $i \in R^k$, $j \in C^k$ for some k. Thus each 1 in B must be covered by at least one rectangle from the cover. A covering need not be disjoint, so that a 1 in B

[2]The $*$'s in B represent don't cares and are introduced by some algorithms in solving the "covering" problem for B.

may be covered by more than one rectangle. The points of B which are labeled * are not required to be covered by any rectangle in the cover. These points represent "don't-care" points in the matrix.

Each rectangle (R^k, C^k) has an associated weight (or cost) defined by a "weight function $w(R^k, C^k)$." The weight of a rectangle cover $\{(R^k, C^k)\}$ is defined as the sum

$$\sum_k w(R^k, C^k).$$

The "minimum-weighted rectangle-covering problem" is that of finding a rectangle cover of a matrix with minimum total weight.

2) Rectangles and Kernels: Rectangles in a matrix provide an alternate way of representing and interpreting the kernels of a logic function. Consider the expression $x = ab\bar{d} + acd + bcd$ but represented as a Boolean matrix B (called the "cube-literal matrix"), where there is one row for each term in the disjunctive form and one column for each different literal. For example, the expression x is represented as follows.

	a	b	c	d	$\bar{d}$
$ab\bar{d}$	1	1	0	0	1
acd	1	0	1	1	0
bcd	0	1	1	1	0

The correspondence between rectangles of the Boolean matrix for f and kernels for f is given by the following discussion and was suggested by an observation of A. Wang: that intersections of kernels can be obtained by the kerneling algorithm.

The expression corresponding to a co-rectangle of the expression is determined by the entries in the Boolean matrix restricted to the rows and columns of this co-rectangle. For the rectangle $\{R, C\} = \{(2, 3), (3, 4)\}$ the co-rectangle is $\{R, C\} = \{(2, 3), (1, 2, 5)\}$ in the preceding example, and the corresponding expression is $a + b$. Thus the co-rectangle corresponds to a kernel. The rectangle itself corresponds to the co-kernel, i.e., the cube divisor used to obtain the kernel. The cube divisor is the set of literals corresponding to the columns C; in the preceding example, this is the cube cd.

The following proposition states more precisely the relationship between kernels and co-rectangles, and co-kernels and rectangles.

Proposition 4.4: c is a co-kernel of f if and only if it is the cube corresponding to a prime rectangle of the cube-literal matrix of f with at least two rows. A kernel is the expression associated with the co-rectangle of a prime rectangle.

From the rectangle interpretation of kernels, it is also possible to understand more clearly the notion of the level of a kernel. A level-0 kernel is the co-rectangle of a prime rectangle which has no other rectangle containing its column set. In other words, it corresponds to a prime rectangle of maximal width. A prime rectangle of maximal height corresponds to a kernel of maximal level, i.e., one whose row set is not contained in any other rectangle.

3) Common-Cube Extraction: Common-cube extraction is the process of finding cubes common to two or more expressions and extracting the common cube to simplify each of the expressions [82]. The optimization problem is to find the particular cubes to introduce into the network to provide an optimum decomposition.

Common cubes can be identified easily using the cube-literal matrix. First, the cube-literal matrix for the Boolean network is created. A rectangle in the cube-literal matrix identifies a cube which can be extracted from the network. The columns of the rectangle identify the literals in the common-cube, and the rows identify the cubes (and expressions) where the common cube appears.

The weight function for a rectangle measures the optimization goal for cube extraction. To minimize the total number of literals in the network, the weight of a rectangle is chosen so that the weight of a rectangle-cover of the cube-literal matrix equals the total number of literals in the network after the new single-cube functions are added to the network. Hence the minimum-weighted cover corresponds to the optimum "simultaneous" extraction of a collection of cubes.

For cube extraction, the weight of a rectangle is defined as

$$w(R, C) = \begin{cases} |C| & \text{if } |R| = 1 \\ |R| + |C| & \text{if } |R| > 1 \end{cases}$$

If a rectangle (R, C) has only a single row, this corresponds to leaving the cube unchanged in the network (no extraction); hence the weight of this rectangle counts the number of literals in the cube. If the rectangle has more than one row, this corresponds to creating a new single-cube function (with $|C|$ literals), and substituting this new function into $|R|$ other cubes at a cost of $|R|$ additional literals; hence the weight of a multiple-row rectangle is $|R| + |C|$.

When searching for a rectangle to extract, it is useful to define the "value" of a rectangle. For cube extraction, the value of a rectangle is defined as

$$v(R, C) = |\{(i, j) | B_{ij} = 1, i \in R, j \in C\}| - w(R, C).$$

The value reflects the desirability of choosing the rectangle and is equal to the number of literals which would be saved in the network if this rectangle is extracted. This is simply the number of 1 points covered by the rectangle minus the weight of the rectangle. No additional literals are saved for covering a * in the matrix; hence these are not counted. If a rectangle contains only points which are 1, then the value of a rectangle for cube extraction is the area minus the perimeter.

4) Kernel-Intersection Extraction: As discussed in section IV-D, kernels can be used effectively in obtaining common subexpressions. The choice of an "optimal" kernel intersection, i.e., a kernel intersection that will most reduce the number of literals in a Boolean network once substituted into the nodes of the network, is a complex optimization problem. However, it can also be expressed as a rectangle covering problem [82].

The Boolean matrix associated with the optimal kernel-intersection problem is called the "co-kernel-cube matrix." A row in this matrix corresponds to a co-kernel (and its associated kernel), and each column corresponds to a cube present in some kernel. The entry B_{ij} is set to 1 if the kernel associated with row i contains the cube associated with column j.

For example, given the equations

$$F = af + bf + ag + cg + ade + bde + cde$$
$$G = af + bf + ace + bce$$
$$H = ade + cde$$

the kernels (associated co-kernels are shown in parentheses) of F are $\{de + f + g(a), de + f(b), a + b + c(de), a + b(f), de + g(c), a + c(g)\}$. The kernels (and co-kernels) of G are $\{ce + f(a, b), a + b(f, ce)\}$, and the only kernel of H is $\{a + c(de)\}$. For ease of presentation, the functions F and G, which themselves are kernels, are not listed in the set of kernels. The co-kernel cube matrix is easily constructed from this data. The unique cubes from all of the kernels are a, b, c, ce, de, f, and g; these cubes are used to label the columns of the matrix. There are thirteen kernels, and the corresponding co-kernels are used to label the rows of the matrix.

The product of a co-kernel for a row and the kernel-cube for a column yields a cube of the expression of one or more of the original functions. For reference, the cubes of the original expressions are numbered from 1 to 13, e.g., af is labeled 1, bf 2, and so on. The number of the cube resulting from the product of the co-kernel for row i and the kernel-cube for column j is placed at position B_{ij} in the co-kernel cube matrix. For example, the co-kernel a when multiplied by the kernel $de + f + g$ yields the cubes numbered 5, 1, and 3, which are ade, af, and ag. Note that there is often more than one way to form each cube in an expression. For example, cube 1 (af) is created by the co-kernel a multiplying the kernel $de + f + g$, and by the co-kernel f multiplying the kernel $a + b$.

The co-kernel cube matrix for the previous example is as follows.

			a	b	c	ce	de	f	g
			1	2	3	4	5	6	7
F	a	1	·	·	·	·	5	1	3
F	b	2	·	·	·	·	6	2	·
F	de	3	5	6	7	·	·	·	·
F	f	4	1	2	·	·	·	·	·
F	c	5	·	·	·	·	7	·	4
F	g	6	3	·	4	·	·	·	·
G	a	7	·	·	·	10	·	8	·
G	b	8	·	·	·	11	·	9	·
G	ce	9	10	11	·	·	·	·	·
G	f	10	8	9	·	·	·	·	·
H	de	11	12	·	13	·	·	·	·

A rectangle of the co-kernel cube matrix identifies an intersection of kernels; this kernel intersection is a common subexpression in the network. The columns of the rectangle identify the cubes in the subexpression, and the rows of the rectangle identify the particular functions that the subexpression divides. The entries covered by the matrix correspond to cubes from the original network.

From the previous example, the prime rectangle ($\{3, 4, 9, 10\}$, $\{1, 2\}$) identifies the subexpression $a + b$ which divides the functions F and G. Cubes numbered 1, 2, 5, 6, 8, 9, 10, and 11 from the original set of functions are covered by this rectangle. This corresponds to the factorization of the equations into the form

$$F = deX + fX + ag + cg + cde$$

$$G = ceX + fX$$

$$H = ade + cde$$

$$X = a + b.$$

The weight of a rectangle of the co-kernel cube matrix is chosen to reflect the number of literals in the network if the corresponding common subexpression is inserted into the network. A minimum-weighted rectangle-cover of the co-kernel cube matrix then corresponds to a "simultaneous" selection of a set of subexpressions to add to the network in order to minimize the total number of literals in the network.

As before, weights w_i^r and w_j^c are defined for the rows and columns of the matrix, and the weight of a rectangle is defined in terms of these weights. Also, values V_{ij} are defined for the elements of the matrix, and the value of a rectangle is defined in terms of the values of the elements covered by the rectangle, and the weight of the rectangle.

The value of a rectangle (R, C) of the co-kernel cube matrix is thus defined as

$$v(R, C) = \sum_{i \in R, j \in C} V_{ij} - w(R, C).$$

Note that the co-kernel cube matrix may be quite large. Thus finding the best kernel intersection may be expensive. It is sometimes more effective to reduce the number of kernels to be examined; for example, restricting to the set of all level-0 kernels.

5) Minimum-Weighted Rectangle Covering Algorithms: We have seen that an optimum solution to the minimum-weighted rectangle covering problem yields optimum algebraic extraction, including common-cube and kernel-intersection extraction, offering a unified approach to the extraction, factorization, and decomposition problems. However, the minimum-weighted rectangle covering problem is NP-complete [82] and an exact solution is possible only in limited cases.

In general, a heuristic procedure is preferred to apply rectangle covering to algebraic extraction. Heuristic procedures can be divided into two categories:

1) extract the "best" rectangle, substitute it into the expressions, and then reapply the procedure to a new modified matrix to take into account the operations performed;
2) find a "good" rectangle cover by generating a cover and then refining it; substitute the corresponding expressions into the Boolean network and repeat on a new modified matrix to take into account the operations performed.

The advantage of the first technique is that it takes into account immediately common factors between the newly extracted function and the rest of the logic network. The disadvantage of this approach is that it selects greedily only one rectangle at a time and does not account easily for the simultaneous extraction of multiple rectangles.

Both these approaches have been implemented in MIS [82] by R. Rudell based on a very efficient sparse-matrix representation.

6) Simultaneous Selection of Rectangles: An alternate approach to the greedy nature of the previous approach is to find a minimum-weight rectangle cover and then extract simultaneously all of the rectangles from the matrix. This algorithm is shown in Fig. 5 and is analogous to a single pass of the EXPAND, IRREDUNDANT, REDUCE sequence of ESPRESSO [22]. This operation can be iterated, as done in ESPRESSO, by defining an expand procedure to expand

```
COVERING_EXTRACT(B) {
P =RECT_PRIME_COVER(B)
P =RECT_IRREDUNDANT(P, B)
P =RECT_REDUCE(P, B)
extract the rectangles of P
}
```

Fig. 5. Algorithm COVERING_EXTRACT.

each rectangle from an initial covering into a prime rectangle. This is then made irredundant and reduced, with the reduced rectangles becoming the input to the first part for reexpansion. Iteration would continue until no decrease in weight is obtained. As in ESPRESSO, this style of heuristic algorithm depends on finding good heuristics for choosing the direction for expansion, and the sequence in which the rectangles are reduced.

G. Boolean Methods

The algebraic methods are fast because the logic function is treated as a polynomial, and hence fast methods of manipulation are available. Although some optimality is sacrificed, this is acceptable in multilevel logic synthesis during the initial phases of the synthesis process. However, when these methods fail to produce improved results, stronger methods can be used if further optimization is desired. These stronger methods, called Boolean methods, treat the logic expression as a true logic function using all the Boolean identities as well as don't cares to achieve a better answer. While these are slower, they can be very effective in overcoming a local "algebraic" minimum. Often, after a Boolean method is used, the algebraic methods can be repeated, usually with further improvement. In many cases, this iteration can be repeated with continued improvement.

1) Boolean Division: Some of the Boolean methods are based on replacing algebraic division, in the operations discussed, by Boolean division. The process of Boolean division is the following. Given a function f and a divisor p, find a quotient q and remainder r such that $f = gq + r$, and such that q and r are as "simple" as possible. To meet this objective, we use any of the Boolean identities and don't cares available. This division is implemented by introducing an artificial variable for p, say $z = p$. Then a "local" don't-care set is generated, $z\bar{p} + \bar{z}p$, i.e., all the values of the variables in the net that make z different from p can never occur and hence are don't cares. Then f is minimized using this, along with other don't cares that may be available. In addition, there are several options to this basic procedure depending on how the result of division is to be used.

1) we would like z to remain in the answer.
2) we would like $\bar{z}$ not to be in the answer.
3) we would like to control what is "simple."

The reason for z to be in the final result $f_{\min}$ is that the quotient q obtained by the division is defined as $f_{\min}/z$. Thus z must be forced into the final answer. This can be enforced by modifying the EXPAND procedure of the minimizer; whenever a cube is to be expanded, z is not expanded until after expanding all other variables. If we want to allow the answer to be expressed in terms of $\bar{z}$, then we can force $\bar{z}$ and define its corresponding quotient as $\bar{q} = f_{\min}/\bar{z}$. Thus $f_{\min} = zq + \bar{z}\bar{q} + r$. Similarly $\bar{z}$ can be removed from the result by expanding it first (we know that it will expand, and hence $\bar{z}$ will not appear). The requirement that the result be simple is related to the discussion in section V-C-2 of having a minimizer which returns the simplest factored form. Since such a minimizer is not yet available, several heuristics have been employed. One is to ask for a result which has the minimum literal or (minimum variable) support.

Minimum literals heuristic: This is a heuristic that can be added to a two-level minimizer just before cubes in the cover are expanded to primes. The objective is, given a set of cubes, to find a prime cover which has the minimum number of distinct literals in the expanded cover (minimum literal support). (A similar objective asks for the minimum number of variables in the support. However, experiments have shown that the minimum literal heuristic leads to better results.) Note that the literal support of a function is exactly the set of nets that must be routed to this function, so a side benefit may be a netlist that is easier to route.

Finding the minimum set of literals is relatively simple; only expand a literal in any cube if it can be expanded from all cubes in which it occurs. To obtain the minimum answer, the order of expansion is important. The solution to this is obtained by using a blocking matrix as defined in [22]: for each cube in the cover, the "literal" blocking matrix is a Boolean matrix with a 1 in position i, j if literal l_j appears in the cube to be expanded, and $\bar{l}_j$ appears in the ith cube of the off-set. A "super blocking matrix" is formed by concatenating the literal-blocking matrices for all the cubes in the cover. The set of columns of the minimum column cover represents the literals that can be simultaneously removed from all the cubes of the cover. The minimum column cover for this matrix is found by an efficient algorithm [22]. After removing these literals and arriving at a cover with minimum literal support, further expansion of the resulting cubes is done in the usual way. The result is a prime cover with the guaranteed minimum number of literals in its support (provided the minimum column cover problem was solved exactly).

Boolean resubstitution: In BOLD, an operation called Boolean resubstitution [49] is used. It can be seen as a generalization of the ESPRESSO REDUCE operation to the multilevel context, and adds variables to the support of the function being minimized. In BOLD, both Boolean resubstitution and algebraic decomposition generate the overall structure of the Boolean network. (A mechanism similar to Boolean resubstitution is the subset-support filter described in section V-C-2).

The basic idea for each node function of the Boolean network and for each intermediate variable is

1) reduce each node function with respect to the variable (i.e., REDUCE the sum-of-products representation of the node function);
2) expand the node function; literals corresponding to the variable are not expanded;
3) expand with respect to the variable.

This process can be quite effective in some problems by exploiting already existing logic, although the critical path length may increase if not controlled during the resubsti-

tution process. Also in its raw form, this is an expensive procedure—in some problems, 90–95 percent (or more) of the execution time is spent in Boolean resubstitution. Currently [77] these drawbacks have been almost completely eliminated by employing filters on the candidate variable set [84], and by using implication (cf. section V-D-1) information to decrease the work in reducing and expanding [85] the candidate substitution variables.

2) Spectral Methods: Spectral methods focus on transforming the input space B^n into one represented in a different basis so that the functions to be implemented, as functions of the new basis, have more obvious and simpler implementations. For example, if the transformed function becomes a single AND or XOR, then the logic that must be implemented requires only one gate plus the logic to perform the input transformation. An interesting way to look at this topic is to envision the Boolean n-space as a Boolean cube, and a Boolean function f on this space as a set of vertices on this cube. All vertices where $f = 1$ are given a black dot. The objective of the input transformation is to rotate sequentially and transform (like a Rubik's cube) the faces of this cube so that most of the black dots are moved to or near the same face. The transformations of the faces represent intermediate logic functions which create an initial decomposition. After this, the function, as a function of these intermediate variables, is a simpler function. For example, if all the black dots occupy, after the transformation, an entire face or cube of the space, then the function can be implemented as a single AND term. This point of view has been proposed by [41]. The main idea is to be able to transform the input space so that the function becomes much simpler in the new variables.

The transformations considered by the spectral methods have some similarity with Fourier transforms and can be computed in $O(N \log N)$, but here N is the number of minterms in the space. Thus the direct approach to implementing the spectral methods has until recently found little practical application. We discuss some of the classic approaches to this and point to some recent literature which attempts to lower the computational complexity.

XOR decomposition: XOR's functions are examples of Boolean factors that are difficult to identify. Algebraic methods do not perform well on functions that have good XOR decompositions.

The idea of XOR decomposition is to implement a Boolean function as two logic blocks. The first block consists completely of XOR gates and the second block consists of unrestricted combinational logic. The problem is defined as follows. Given a function f with n inputs and k outputs, find a decomposition of f into two blocks of logic, σ and f_σ, such that σ consists completely of XOR gates and has n inputs and n outputs, f_σ consists of unrestricted combinational logic and has n inputs and k outputs, and the "simplicity" of f_σ is maximized (simplicity is defined below). σ is called the linear block, and f_σ the nonlinear block. The block σ can be viewed as a linear transformation of the input space which effects a change of basis.

To find the XOR decomposition of a k-output function $f = \{f^0, f^1, \cdots f^{k-1}\}$, an autocorrelation function B is used:

$$B_f(\tau) = \sum_{i=0}^{k-1} B_{f^{(i)}}(\tau) = \sum_{i=0}^{k-1} \sum_{x \in \{0,1\}^n} f^{(i)}(x) f^{(i)}(x \oplus \tau)$$

$$\forall \tau \in \{0, 1\}^n.$$

In general, if we translate a function f by $\tau(f(x) \Rightarrow f(x \oplus \tau))$, then the autocorrelation function, $B_f(\tau)$, is a measure of how well the function f correlates with its translated image. For this particular problem, we may visualize the autocorrelation function as follows. Picture the Boolean n-cube of the input space B^n. Given a single output Boolean function $f^{(i)}$ defined on this space, put markers over every point in the n-cube in the on-set of $f^{(i)}$. Now pick some point $\tau \in B^n$. Without moving the markers, rotate the Boolean n-cube in place so that τ and the point $(0 \cdots 0) \in B^n$ switch places. The autocorrelation $B_{f^{(i)}}(\tau)$ is a measure of the number of points in B^n that are covered by markers both before and after the rotation.

In [96] it is proposed to measure the simplicity of a function as the sum over all the function's outputs of the number of miniterms in the on-set that are distance 1 from each other. This can be defined in terms of the autocorrelation function for special τ's, i.e., any of the n miniterms that are distance 1 from the origin. (Note that when an input x is XORed with a constant τ, the effect is to translate x by $\tau(x \oplus \tau)$.) Now the problem is to compute the autocorrelation function.

Special methods are applicable to this because they provide an elegant method of computing $B_f(\tau)$ using the Weiner–Khinchin theorem [76]. One such transform is the Reed–Muller transformation:

$$R^n F = S$$

where F and S are the truth tables of the function before and after the transformation, respectively, and

$$R^n = \begin{bmatrix} R^{n-1} & 0 \\ R^{n-1} & R^{n-1} \end{bmatrix}$$

where

$$R^1 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

Each row k of R^n is the truth table for the so-called kth Reed–Muller function $r^{(k)}$. This set of functions forms the new basis for the transformed space. The elements S_k of S are the spectral coefficients. (Note that the functions f can now be expressed as $f = \oplus_k S_k r^{(k)}$ which represents f as an XOR sum.) S_k can be interpreted as the correlation between the given function vector f and $r^{(k)}$ [54]. A drawback of this approach is that a truth table representation is needed to compute the spectral coefficients, and thus the overall complexity is exponential.

In place of using a single linear decomposition, a decomposition into a linear block σ followed by a nonlinear block $f_\sigma(f(x) = f_\sigma(\sigma x))$ can be accomplished by constructing a matrix σ as follows. Let T be an $n \times n$ matrix with columns τ_i, $T = [\tau_0, \tau_1, \cdots, \tau_{n-1}]$, where

1) $\tau_0 = \text{argmax}_{\tau \neq 0} B_f(\tau)$
2) L_i is the linear space spanned by $\{0, \tau_0, \cdots, \tau_i\}$
3) $\tau_i = \text{argmax}_{\tau \in L_{i-1}} B_f(\tau)$
4) $\sigma = T^{-1}$.

Then a direct implementation of the linear part σ is

$$z_i = \oplus_{0 \leq j \leq m-1} \sigma_{ij} x_j.$$

The nonlinear part is implemented in the usual way as a function of z, i.e., each miniterm in the on-set of f is trans-

lated into another miniterm by σ. The set of all these miniterms represents a new function $f(z)$, which is then implemented by some multilevel synthesis process (MIS, ESPRESSO, etc.).

Of course, if one implements this procedure directly, the complexity is still exponential since the basis of the transformations involves truth-table-like manipulations. Varma and Trachtenberg [96], in an attempt at making spectral methods more practical, suggest a method of using a sum-of-products cover of a function to calculate autocorrelation coefficients. They introduce the concept of an "arithmetic" cover, which is a redundant cover produced by a sequence of pair-wise intersections of cubes in the cover. Each iteration in the sequence has a different sign associated with its resultant cubes. The idea is that alternating signs of each iteration in a prescribed manner will properly count the cube intersections and thus properly count the total number of minterms covered by the function. They give heuristics to generate a reduced set of autocorrelation coefficients from the arithmetic cover and propose to calculate the autocorrelation coefficients approximately by arbitrarily truncating the sequence of pair-wise intersections that produces the arithmetic cover. This corresponds to approximating a Fourier series by its lower order components.

Mixed generalized Reed-Muller form: A Reed-Muller form is simply an XOR sum-of-products where every variable appears in its positive phase only, and where the OR operator is replaced by the XOR operator. It can be easily derived from the regular sum-of-products by replacing any complemented variable $\bar{x}$ by $x \oplus 1$ and multiplying out the resulting expression using the distributive law. For example,

$$a\bar{b}c\bar{d} = a(b \oplus 1)c(d \oplus 1) = abcd \oplus abc \oplus acd \oplus ac.$$

The mixed generalized Reed-Muller form allows both polarities of a variable to occur. In [53], a program (EXORCISM) is described in which heuristic minimization of logic functions in this form are performed. It is claimed that this is useful for circuits such as arithmetic and communication circuits, encrypting schemes, coding for error control, etc. It is also claimed that Reed-Muller forms are candidates for easily testable circuits with "function independent" testing [79].

3) Methods of Ashenhurst and Curtis: These methods aim at re-expressing a logic function as a function of other functions, i.e., as a multilevel network. Ashenhurst [3], in a fundamental paper, stated the simple disjunctive decomposition theorem: a function $f(A, B)$ is decomposable with "bound set" A and "free set" B (i.e., $f(A, B) = F(\phi(A), B)$) if and only if its $2^{|B|} \times 2^{|A|}$ Karnaugh map, with the variables B defining the rows and A defining the columns, has at most four distinct kinds of columns:

1) all 0's
2) all 1's
3) a fixed pattern of 0's and 1's
4) the complement of (3).

Curtis [34] extended Ashenhurst's results to include a multiple decomposition as follows: a function $f(A, B)$ is expressible as a composite function $F(\phi_1(A), \cdots, \phi_k(A), B)$ if and only if its $2^{|B|} \times 2^{|A|}$ Karnaugh map has, besides 0 and 1, at most 2^k distinct column vectors. For $k = 1$, this theorem reduces to Ashenhurst's theorem. Curtis also stated that a switching function $f(A, B)$ is expressible as a composite function

$$F(\phi_1(A), \cdots, \phi_p(A), \eta_1(B), \cdots, \eta_q(B))$$

if and only if the $2^{|B|} \times 2^{|A|}$ and $2^{|A|} \times 2^{|B|}$ Karnaugh maps have at most 2^p and 2^q distinct columns, respectively.

To see how this works, imagine the truth table for f reshaped into a matrix where each row is indexed by a minterm in the B space. Let m be a minterm of the B variables. Then f_m, the cofactor of f with respect to m, is the mth row of this matrix. Now, by the generalized Shannon expansion,

$$f = \sum m f_m(A).$$

If besides 0 and 1 there are at most k distinct functions among the $f_m(A)$, then f can be written

$$f = \sum_{i=1}^{k} g_i(B)\phi_i(A) + g_0(B)$$

where $g_i(B)$ is a logic function consisting of all miniterms in the B space with the common row $\phi_i(A)$. In addition, instead of implementing each of the k functions $g_i(B)$ (which are mutually exclusive), one can encode them with $q = \log_2 k$ functions η_j for a simpler implementation. Then f can be written as

$$f = \sum_{i=1}^{k} ((\eta_1(B), \cdots, \eta_q(B)) = e_i)\phi_i(A)$$

where e_i is the encoding for $g_i(B)$. Of course, the choice of this encoding is important to obtain functions η_j that are as simple as possible.

Roth and Karp [80] presented a procedure for Boolean function decomposition which operates on the on-set and off-set covers, rather than on the truth table, for the function. However, their algorithm still requires exponential time to find the minimum cost decomposition.

Periodically, these methods have been rediscovered and some implementations have been attempted but with notable lack of success on practical problems. Some other interesting methods of decomposition to be mentioned are Muroga's method for decomposition into a minimum number of negative gates [55] and Davidson's NAND decomposition [37].

H. Output Phase Assignment

In minimizing logic for PLA's, it is well known that the phase assigned to each output can make a substantial difference in the final area. However, for multilevel logic, this should not be a factor, since one can obtain either phase of an output by simply inserting an inverter. Thus it is surprising that methods for finding an output phase assignment for multilevel logic have been developed, and substantial improvement in the area has been reported in some examples [98]. A possible explanation is that the algorithms currently used in multilevel synthesis are relatively weak, especially if only algebraic operations are used, and cannot examine all possible optimizations. For example, the algebraic algorithms perform cube and kernel extraction only on the positive phases of the functions at the nodes. In addition, the cube or kernel is selected on the basis of its value. Current implementations make this value judgement only on the basis of the positive phase of all the node functions

present, and miss the good algebraic kernels of the complement functions.

Thus there are possibly two choices for obtaining better results. One is to start with a two-level version of the logic and apply phase assignment techniques similar to that done for PLA implementations [98]. After selecting a phase assignment, regular multilevel synthesis is invoked. The second choice is to ignore phase assignment and to extend the cube- and kernel-finding algorithms to ones which examine both phases in order to extract and evaluate cubes and kernels. To date, a comparison between these two approaches has not been done. It would be of interest if the results obtained are still substantially different, thus suggesting other mechanisms at work.

I. Restructuring for Timing

Being able to meet performance requirements is absolutely essential in synthesizing logic circuits. As circuit complexity increases, manual methods for performance improvement become impractical and must be replaced with automatic performance-optimization systems. These must work with different levels of circuit hierarchy and at various steps of the design process (e.g., retiming, reducing delay in combinational logic, delay-driven layout, etc.).

Timing optimization of combinational circuits can be viewed as a three-phase process. In the first phase, circuits are globally restructured to have better "timing properties." As a simple example, Fig. 6 shows two equivalent circuits. If the arrival times of all the inputs are the same, circuit (b) is preferred over circuit (a), for it reduces the output arrival time. On the other hand, if input u is the critical signal, circuit (a) becomes superior. Thus, even though the two circuits have the same area, one is better than the other when speed is important. Here the quality of the circuits is judged not by the detailed timing diagrams, but rather by the circuit structure. A more sophisticated example of global restructuring is the conversion from a carry-ripple adder to a carry-look-ahead adder. The restructuring-for-timing phase is characterized by its independence from the target technology. The objective here is to look for global structural changes of circuits to achieve delay reductions that cannot be obtained by lower-level techniques such as transistor sizing or buffering.

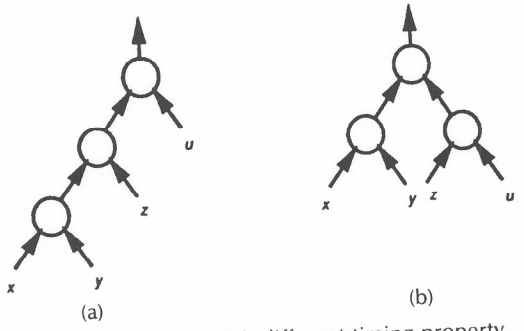


Fig. 6. Equivalent circuits with different timing property.

A second phase of timing optimization may be performed during the physical-design process. Here the target technology is known and more accurate timing information is available. Optimization involves transistor sizing, buffering, and delay-driven placement. This phase is characterized by its dependence on a particular target technology and on the existence of fast and relatively accurate timing simulators.

A third and last phase of timing optimization may be performed when actual designs are available. There, much more accurate timing analyzers can be used to fine-tune the circuit parameters. This phase serves both the optimization and verification purposes.

There have been several previous attempts to solve the timing restructuring problem. SOCRATES [5] uses a rule-based approach and tries to achieve global restructuring through a sequence of local transformations. More recently, an algorithmic-based restructuring technique was developed in the Yorktown Silicon Compiler [13] and in [89]. In this section we review the algorithmic-based techniques for restructuring for timing, while the rule-based techniques are reviewed in section VIII.

1) Basic Definitions: The "arrival time" of a signal is the time at which the signal settles to its steady-state value. A_s is used to denote the arrival time of signal s. (All times are relative to an arbitrary, but common, reference point). The "required time" of a signal is the time at which the signal is required to be stable. R_s is used to denote the required time of signal s.

The "slack" of a signal is the difference between its required time and arrival time. S_s is used to denote the slack of signal s and is defined as

$$S_s = R_s - A_s.$$

It is clear that the slack value of a signal measures its criticality, i.e., signals with negative slacks are considered to be critical. Unlike arrival times and required times, slacks have no reference point; hence slacks are sometimes more convenient to use.

One method of timing optimization uses the approach of mapping the network into two-input NAND gates and inverters and then uses a unit delay model. It is specifically designed to be used in conjunction with technology mapping in MIS [16]. The general approach is first to minimize the area of a network without concern for the delay (e.g., all the global common factors have been extracted out). Next, the network is decomposed into two-input NAND gates and inverters, which is the input format for the technology mapping algorithms. At this point, timing optimization is invoked to restructure the circuit into an alternative two-input NAND-gate and inverter form in which critical paths are reduced at the possible expense of area. The output of timing optimization is then fed directly to the technology-mapping stage.

Timing constraints are specified as input-arrival times of primary inputs and output-required times of primary outputs. The goal of timing optimization is to meet the timing constraints while keeping the area increase to its minimum.

2) Restructuring Algorithm: The critical section of a Boolean network is composed of critical paths from primary inputs to primary outputs. Given a critical path, the total delay on the path can be reduced if any section of the path is sped up. For example, Fig. 7(a) shows a critical path, $a - x - y$. The critical path can be reduced by first collapsing x and y and then redecomposing y in a different way to minimize the critical path, as shown in Fig. 7(b). This method (first collapsing along a critical path and then redecom-

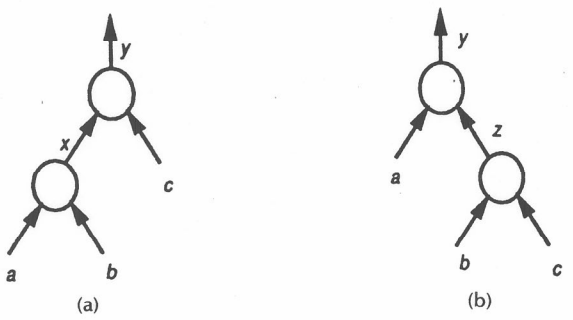


Fig. 7. Reducing delay by collapsing and redecomposition.

posing to shorten the critical path) is the basic step taken in restructuring. The nodes along the critical paths chosen to be collapsed and resynthesized form the "resynthesis region."

Since a critical section usually consists of several overlapping critical paths, the algorithm selects a minimum set of subsections, "resynthesis points," which when sped up will reduce the delays on all the critical paths. A weight is assigned to each candidate resynthesis point to account for possible area increase and for the total number of resynthesis points required. The goal is to select a set of points which cuts all the critical paths and has minimum total weight; this set is called "minimum-weighted node cut-set."

Once the resynthesis points are chosen, they are sped up by the collapsing–decomposing procedure. The simplified delay model is then used to find the new critical section of the network. The algorithm proceeds iteratively until the timing requirement is satisfied or no improvement can be made. The following is an outline of the algorithm.

SPEED_UP(η)

1) Compute the arrival and required times for all the nodes in η, using the supplied arrival times at the primary inputs and the required times at the primary outputs.
2) Find all the critical nodes in η.
3) Compute a weight for each critical node.
4) Find the minimum weighted cut-set of all the critical paths.
5) Partially collapse along the critical path at each node on the cut-set. The length of section along each critical path to be collapsed is controlled by a parameter d.
6) Redecompose each collapsed node into two-input NAND gates and inverters.
7) If the timing requirement is satisfied, done.
8) If the circuit improved from the previous iteration, go to step 1.

Computing weights of critical nodes: The weight assigned to each node in the critical section must reflect 1) its potential for speed-up and 2) an area penalty incurred if the node is chosen. Some critical nodes are easier to speed up than others. For example, in Fig. 8(a), all the nodes are critical. If node y is selected, collapsing its critical fan-in into y will result in a node with one critical input x and two noncritical inputs. So, it is easy to decompose it such that the critical path is reduced, as indicated in Fig. 8(b). If, on the other hand, x is chosen, collapsing its critical fan-ins into x will result in a node with all of its fan-ins being critical.

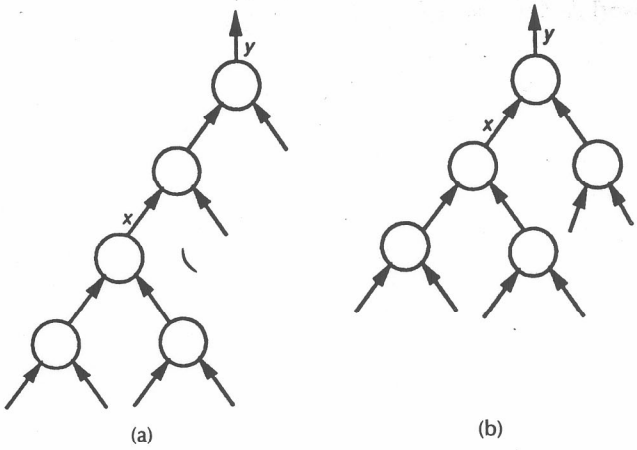


Fig. 8. Node y is easier to speed up than x.

So, there is no decomposition that can reduce the critical paths in this case. The weight of a critical node should reflect how easy it is to resynthesize at the node.

It is also possible that, to reduce a critical path, certain nodes have to be duplicated. For example, Fig. 9(a) is part

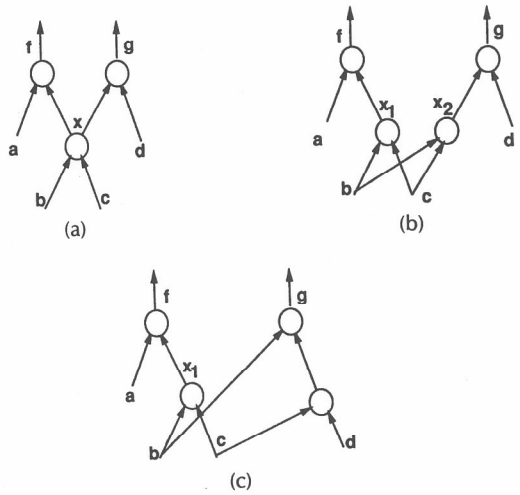


Fig. 9. Area increase during resynthesis.

of a network with critical signals $b - x - g$. If g is chosen as a resynthesis point, x needs to be collapsed into g and redecomposed in a different way. Since f also depends on x, x needs to be duplicated before the collapsing, as indicated in Fig. 9(b). Now the critical path becomes $b - x_2 - g$ and can then be reduced as shown in Fig. 9(c). This increase in area should be reflected in the weight of g. Also, it may be that x existed initially because it had good area value. Now, however, its fan-out has been reduced, so it should be examined again for its area value and eliminated if profitable.

Both the potential for speed-up and the area increase of a critical node depend on the size of the "resynthesis region" at the node. The resynthesis region of a node consists of a set of critical nodes within a distance d from the node. d is a parameter for the global restructuring algorithm and can be used to control the amount of speed-up to be made in each iteration.

To find the minimum weighted cut-set of a Boolean net-

work, a flow network [64] is constructed and the max-flow min-cut algorithm is used.

3) Resynthesis: Once the minimum weighted node cut-set is found, each node on the cut-set is then resynthesized. The resynthesis of a node x involves collapsing all the critical fan-ins of x, within a distance d from x, into x, and then decomposing x back to two-input NAND gates and inverters such that the critical path is minimized. The objective of the redecomposition is to minimize A_x, the arrival time of x. For this, timing-driven decomposition algorithms are used.

"Timing-driven decomposition" refers to decomposing a single-output logic function, given the arrival times of all the inputs and a delay model, into a tree of two-input NAND gates and inverters such that the output arrival time is minimized. Existing methods include rule-based approaches [35], [5] and tree-balancing techniques [62]. These techniques work on an existing decomposition and incrementally modify the decomposition to reduce the output arrival time. A direct constructive algorithm is described in [89]. Given a function and arrival times of all the inputs, the algorithm decomposes "optimally" the function into two-input NAND gates and inverters with minimum output arrival time. Exact conditions are given, under which the algorithm produces optimum results.

4) Incremental Delay Trace: Global timing optimization depends heavily on the delay information, such as slacks or arrival times, derived using a chosen delay model. Most procedures perform a complete delay trace over the entire network each time the network is restructured. However, restructuring algorithms modify at each iteration only a small section of the network. It is unnecessary and quite wasteful to recompute all the timing information. A simple example is in Fig. 10. The numbers at the nodes are the current arrival times. The underlying delay model is "unit-delay." Suppose that during the resynthesis, the arrival time of x changes from 1 to 2. The only node whose arrival time is affected is y. Thus only a few nodes were affected.

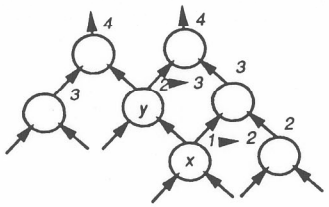


Fig. 10. When A_x changes from 1 to 2, only A_y is affected.

In [97], Boolean networks are abstracted as directed acyclic graphs (DAG's) and several graph theoretical results are developed concerning the properties of DAG's and the ordering of the nodes in DAG's. Using the notion of topological ordering, efficient incremental delay trace algorithms are given which provide greatly improved efficiency during the restructuring for timing process.

V. Logic Optimization/Minimization

This section is concerned with techniques which, given a restructured multilevel network, try to optimize the node functions and, to some extent, improve incrementally the structure. The result may lead to additional possibilities for restructuring which in turn may allow further optimization. The key to optimization is the use of don't cares. These may come from many sources and an understanding of the role they play is fundamental. As we will see, basically all the techniques presented in this section make use of don't cares, some directly and some implicitly. The topic of section V-E on node invariance shows that by understanding which don't cares are used by the various network transformations, one can make definitive statements about the preservation of testability and the test sets for a network.

A. Internal Don't Cares

In section III-A-1, we discussed external don't cares and their several sources. Internal don't cares arise in multilevel logic because of the structure of the Boolean network. The internal don't cares must be deduced from the given network structure. They are divided into satisfiability and observability don't cares.

Satisfiability don't cares: These don't cares are a result of the existence of the additional intermediate variables y_j introduced at the intermediate nodes of a Boolean network. As an example, consider the network

$$x = \bar{a}\bar{b}$$
$$y = \bar{c}\bar{d}$$
$$f = \overline{xy}$$

which implements $f = (a + b)(c + d)$. For any node that uses the intermediate variables x and y, we have the option of eliminating x and y or expanding the Boolean space to include these variables. If we do the latter, there are combinations of variables which will never occur. For example, the combination $x = 0, a = 0, b = 0$ will never occur, and in general, since $x = \bar{a}\bar{b}$, then $x \neq \bar{a}\bar{b}$ will never occur. This is expressed by the logic function

$$x(a + b) + \overline{xab}.$$

The intermediate nodes of a Boolean network impose the relation

$$y_j = f_j(x, y)$$

where x is the set of primary inputs and y the set of intermediate variables. Of course, since the Boolean network is acyclic, f_j depends on only a subset of the y variables. In the space $B^{|x|+|y|}$, the "satisfiability don't-care (SDC) set" is given by

$$\text{SDC} = \sum (y_j \bar{f}_j + \bar{y}_j f_j).$$

Sometimes it is appropriate to leave out the don't care contribution from node i. This is denoted as

$$\text{SDC}_i = \sum_{j \neq i} (y_j \bar{f}_j + \bar{y}_j f_j).$$

The SDC gives all the internal patterns of signals that will never occur, due to the network structure, and is called the satisfiability don't-care set because each of the relations

$$y_j = f_j(x, y)$$

must be satisfied during the correct operation of the network. The part in the SDC contributed by the function f_j, $(y_j \bar{f}_j + \bar{y}_j f_j)$ is called the "local satisfiability don't-care set."

An alternative view of SDC can also be given as follows. In general, a Boolean function $f_j(v)$ is defined on the space of primary inputs B^n. However, multilevel functions are

represented using intermediate variables, and hence lead to the definition of the function on a space of combined intermediate variables and primary inputs B^{n+m}. However, this latter description is incomplete if the topology of the Boolean network is not considered, since mutual dependencies among the variables guarantee that a large set of points on this extended space can never occur (the SDC set). For example, the cube corresponding to $f = xy$ is shown in Fig. 11. The black dots correspond to the elements of the SDC set.

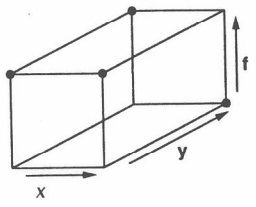


Fig. 11. Cube representation of $f = xy$.

In some sense, the SDC really is not a don't care set at all, since it is trivial (the empty set) when viewed as a set in the primary input space. Also, the size of the SDC set is uniquely determined given only the number of primary inputs (n) and the number of intermediate and output variables (m) in the network ($|\text{SDC}| = 2^{n+m} - 2^n$). This is based on the observation that the number of care points (2^n) in the space remains constant independent of m.

Observability don't cares: These don't cares occur because at each node there is a network structure that limits the observability of the value of the node as seen at a primary output. In discussing this, we need to extend the notion of a cofactor of a function to that of a Boolean network η with respect to a literal x. This results in the cofactored network denoted by η_x. This is obtained simply by cofactoring each node function f_j of η with respect to the literal x. However, there is a subtle distinction to be made here. Note that each fan-out of x, f_j, becomes f_{jx}, a function independent of x. Thus the network η_x can be seen as η with a signal x stuck at 1. A node in the network is defined not only by a logic function f_j, but also by its interrelation with the other nodes in the Boolean network. Thus a node j whose node function f_j does not depend explicitly on x is unaltered in the new network η_x, i.e., $f_j = f_{jx}$, but if it depends implicitly on x in η, then f_{jx}, as a function of the primary inputs, is changed.

The observability of a node j at an output k of a network is the notion that f_k is implicitly dependent on y_j. This is given by the function

$$\frac{\partial f_k}{\partial y_j} \equiv f_{ky_j} \oplus f_{k\bar{y}_j}$$

where $\oplus$ is the XOR operator. The meaning of this is that $\partial f_k / \partial y_j$ gives the input conditions under which the output f_k differs in the two networks η_{y_j} and $\eta_{\bar{y}_j}$, i.e., the conditions under which the value of y_j can be observed at output f_k. The preceding expression is called the Boolean difference of f_k with respect to y_j.

There may be conditions under which the value of y_j cannot be observed at any of the outputs. Assume for the moment that the external don't-care sets DX_k are empty. (This restriction will be removed in the following.) Then these conditions are given precisely by

$$\text{ODC}_j \equiv \prod_{k \text{ an output}} (f_{ky_j} = f_{k\bar{y}_j}) = \prod_{k \text{ an output}} \overline{\frac{\partial f_k}{\partial y_j}}.$$

This is called the "observability don't-care (ODC) set" for the signal y_j. Note that, unlike SDC, which is common for the entire network, the ODC_j is specific to the node y_j.

We can now make a precise connection with testing[3]. A signal y_j can be tested for, say, "stuck-at-1," by finding an input test vector v such that

a) $(\overline{\text{SDC}_{\bar{y}_j}})_v \neq \phi$, and
b) $(\overline{\text{ODC}_j})_v \neq \phi$.

The first condition says that, when we evaluate the network at v, the value of y_j is 0 and the other values of the intermediate variables y_i satisfy the compatibility relations $y_i = f_i(v, y)$. The second condition says that the value of v is such that it allows the value of y_j to be observed, at least at one output;

$$(\overline{\text{ODC}_j})_v = \sum_{k \text{ an output}} \left(\frac{\partial f_k}{\partial y_j}\right)_v \neq \phi.$$

Thus $\overline{\text{SDC}}$ and $\overline{\text{ODC}_j}$ contain the precise conditions under which y_j can be combinationally tested for stuck-at-1 or stuck-at-0. These conditions are associated respectively with the ability to justify the fault and to propagate it to an output.

Generally, $\partial f / \partial y_j$ is not easy to compute. There is a chain rule that can be used; however, this becomes complex quickly. For example, assume that f depends explicitly on $g_1, g_2, \cdots, g_n$, which in turn depend implicitly on y_j. Then, according to the chain rule,

$$\begin{aligned}\frac{\partial f}{\partial y_j} = {} & \frac{\partial f}{\partial g_1}\frac{\partial g_1}{\partial y_j} \oplus \frac{\partial f}{\partial g_2}\frac{\partial g_2}{\partial y_j} \oplus \cdots \oplus \frac{\partial f}{\partial g_n}\frac{\partial g_n}{\partial y_j} \\ & \oplus \frac{\partial^2 f}{\partial g_1 g_2}\frac{\partial g_1}{\partial y_j}\frac{\partial g_2}{\partial y_j} \oplus \frac{\partial^2 f}{\partial g_1 \partial g_3}\frac{\partial g_1}{\partial y_j}\frac{\partial g_3}{\partial y_j} \oplus \cdots \\ & \oplus \frac{\partial^2 f}{\partial g_1 \partial g_n}\frac{\partial g_1}{\partial y_j}\frac{\partial g_n}{\partial y_j} \oplus \cdots \oplus \frac{\partial^2 f}{\partial g_{n-1}\partial g_n}\frac{\partial g_{n-1}}{\partial_{y_i}}\frac{\partial g_n}{\partial y_j} \\ & \oplus \frac{\partial^3 f}{\partial g_1 \partial g_2 \partial g_3}\frac{\partial g_1}{\partial y_j}\frac{\partial g_2}{\partial y_j}\frac{\partial g_3}{\partial y_j} \oplus \cdots \\ & \oplus \frac{\partial^n f}{\partial g_1 \partial g_2 \cdots \partial g_n}\frac{\partial g_1}{\partial y_j} \cdots \frac{\partial g_n}{\partial y_j}.\end{aligned}$$

The terms $\partial g_i / \partial y_j$ can be obtained by recursive application of the chain rule. The high-order terms in the preceding expression can be associated with self-sensitizing reconvergent paths from y_j to an output [73].

The relation between logic minimization and testing is straightforward. If a signal y_j is not combinationally testable for, say, stuck-at-1, then the Boolean network will be unaffected[4] if y_j is replaced by 1, thereby simplifying the

[3]In this discussion, when we speak of testing we mean testing the block of combinational logic assuming that all inputs are controllable and all outputs are observable. This "combinational" testing does not take into account the actual testing environment which may exist. For example, the outputs may not be scan latches, so observability of the outputs may be reduced by how the outputs are used. This, of course, is related to the external don't cares.

[4]It will be unaffected "statically," but it may have an effect on its timing behavior [73].

network. Indeed, not only can the node j be eliminated, but those nodes which are the immediate fan-outs of j can be replaced by f_{iy_j}, which is a simpler function. The signal y_j is said to be redundant, and the methods of optimization which exploit this connection are referred to as "redundancy removal" techniques [9].

One of the transformations made on a network during logic synthesis is to replace the node function f_j by a simpler equivalent one. An important consequence that can be derived using the preceding don't care sets is as follows.

Theorem V-1: Any function $\tilde{f}_j$ that can replace f_j in η, resulting in an equivalent combinational network, is a cover of the incompletely specified function derived from f_j and the don't-care sets SDC_j, ODC_j.

This is important because all possible simplifications of f_j can be obtained by "two-level minimization" using a uniquely derived don't-care set of node j. In practice, the don't-care set used is usually a small subset of this, but the theory displays what is possible and makes the precise relation with testing clear. Thus redundancy removal as just discussed can be seen as a special case of the more powerful node minimization based on don't cares.

The observability don't-care set ODC_j may be obtained by exhaustive search, by computing, for each primary output i in the transitive fan-out of j, the sets

$$\{x \in B^n \mid (y_i)_{y_j}(x) \equiv (y_i)_{\overline{y}_j}(x)\}.$$

This requires 2^n "fault simulations" if applied in a straightforward way ($x \in \text{ODC}_j$ if no difference is observed at any primary output node). An efficient approximation [48] may be obtained by the following.

1) For i in the "immediate" fan-out of j, let

$$E_{i,j} = \left\{\overline{\frac{\partial f_i}{\partial y_j}}(x)\right\}.$$

2) If G is a sum-of-products cover and j an intermediate node, RESTRICT(j, G) is a sum-of-products cover G', constructed from G by deleting all cubes containing variables in the transitive fan-out of node j.

A subset of ODC_j can be defined recursively as follows. If j is a primary output, then $DO_j = \varnothing$ else

$$DO_j = \prod_{i \in FO(j)} \text{RESTRICT}\,(j, E_{i,j} + DO_i).$$

The recursion proceeds backwards from the outputs, depth first. (A similar recursive approximation is discussed in [19], but the RESTRICT operation was missing and the approximation was not always a valid subset.) This approximation is related to the "observability cover" used by Brglez *et al.* in [26], [30], and [31]. The following result states the relation between DO_j and ODC_j.

Theorem V-2 (Approximation of ODC_j): For all intermediate nodes j

$$DO_j\overline{\text{SDC}}_j \subseteq \text{ODC}_j\overline{\text{SDC}}_j.$$

If no reconvergent paths exist from a node j to the primary outputs of η, then

$$DO_j\overline{\text{SDC}}_j \equiv \text{ODC}_j\overline{\text{SDC}}_j.$$

More recently, other valid approximations to ODC_j have been proposed (cf. section V-C).

1) Effect of External Don't Cares: The conditions under which a node can be simplified should be augmented by the use of the external don't-care sets DX_k. This can be done by adding these to the definitions of ODC_j and DO_j as follows:

$$\tilde{\text{ODC}}_j = \prod_{k \text{ an output}} \left(\overline{\frac{\partial f_k}{\partial y_j}} + DX_k\right).$$

$$\tilde{DO}_j = \prod_{i \in FO(j)} \text{RESTRICT}\,(j, E_{i,j} + \tilde{DO}_i)$$

where $\tilde{DO}_k = DX_k$ if k is a primary output. The extensions of Theorems V-1 and V-2 to include these external don't cares is straightforward.

B. Node Minimization

One of the most powerful techniques used in multilevel logic synthesis is node minimization. Node minimization is particularly effective when a Boolean network is partially collapsed around a given node. Usually this is done by eliminating nodes which have little fan-out. In this collapsing process, nodes with large logic functions are created. The hope is these can be effectively simplified. This is done using a two-level minimizer (such as ESPRESSO). The objective is to utilize the implicit don't care conditions, which exist at each node of a Boolean network, to perform two-level logic minimization on the Boolean function associated with the node. Unfortunately, since the entire don't-care set is extremely large, this ideal is prohibitively expensive for most practical circuits. Consequently, approximation of this subproblem becomes necessary, and two alternative heuristic approaches have been studied [4], [49], [77], [16], [68], [85].

The first heuristic, the tautology-based approach, uses multilevel equivalence and tautology-checking algorithms directly, adapting ideas from the testing literature. The advantage of this approach is that all don't-care conditions are automatically accounted for, so the optimizer has the opportunity to account for all optimization degrees of freedom. Implementations [4], [49], [50], [77] to date include most, but not all, ESPRESSO heuristics [22]; hence the power of the two-level logic minimization is somewhat diluted. In particular, the option of exact minimization [83] is lost. Although area-minimality may not be achieved, 100-percent testability of single stuck faults is guaranteed.

The second approach, the don't-care approach [16], [68], employs the complete power of the ESPRESSO minimization heuristics. However, the don't-care sets are approximated, and hence some of the optimization degrees of freedom are sacrificed, so again a suboptimal solution is obtained. Both of these approaches have their virtues. Comparisons to date favor the first in optimization quality and testability, and the second in execution time; however, continued improvements in both approaches [33], [85] may alter these observations.

1) The Tautology-Based Approach: In [4], an algorithm for node minimization is described, based on the EXPAND, REDUCE, and IRREDUNDANT_COVER operations of ESPRESSO [22]. However, these operations do not use the off-set and two-level tautology as in ESPRESSO, but are based on tautology checking [51]. Tautology checking is a method to verify that two logic functions are equivalent.

This method can be applied to check whether a literal or a cube is redundant by removing the literal or cube and checking whether the remaining function is equivalent to the original one.

When applied to multilevel logic synthesis, the EXPAND, REDUCE, and IRREDUNDANT_COVER operations can be based on multilevel tautology to check whether a given literal or cube is redundant. In concert, these procedures are referred to as the "ESPRESSO loop." The efficiency of this approach is determined by the quality of the implementation of the algorithms which form the basis for multilevel tautology, e.g., test pattern generation and logic implication (see section VI).

In [4], a procedure called ESPRESSO_MLT is described which generalizes this to multilevel logic. It takes a Boolean network, and proceeds with a number of optimization passes over all node functions. In each pass, every node is visited and the following steps are applied:

- constant-function check
- ESPRESSO loop
- Boolean resubstitution
- special-case flattening.

When an entire pass is completed without change, the Boolean network can be guaranteed to be prime, irredundant, and 100-percent testable for single-input stuck faults. The tests for all the input stuck faults are automatically supplied as a byproduct of the minimization.

The idea of constant-function check is that if the function is a constant, then at most two equivalence checks (check for constant zero and check for constant one) are required.

Boolean resubstitution, as implemented in BOLD, is a generalization of the REDUCE operation to the multilevel context. Not only does it add literals, but also variables to the support of the function being minimized. Since Boolean resubstitution can discover Boolean factors, it can modify the overall structure of the Boolean network.

The idea of special-case flattening is to save computation time and improve solution quality by eliminating certain trivial functions. Typical moves are to

1) eliminate buffers (trivial) and inverters by referring to the negative phase of the fan-in of the inverter,
2) substitute multilevel "AND" functions into a single-cube "AND" function,
3) substitute multilevel "OR" functions into single-cube functions which reference the negative phase variable, and
4) collapse any intermediate function into any fan-out that has only one literal.

These special cases are chosen because the improvement is known *a priori* and no additional processing is required to realize improvement.

To motivate this approach to multilevel minimization, consider

$$f_1 = \bar{x}_1\,\bar{x}_2 + y_3$$

$$y_2 = f_2 = x_1\bar{x}_2 + \bar{x}_1 x_2$$

$$y_3 = f_3 = x_1 x_2 \bar{y}_2 + \bar{x}_1\bar{x}_2.$$

Here x_1 and x_2 are primary inputs and f_1 and f_2 are primary outputs. This representation has three functions (functions correspond roughly to gates in a standard cell netlist), twelve literals (corresponding roughly to transistors), and seven inputs (corresponding to the amount of interconnect). From a delay viewpoint, the circuit has three levels of logic, and from a testability viewpoint, there are three nontestable input stuck faults, namely y_2 stuck-at-0 in f_3, and x_1 and x_2 stuck-at-1 in f_3.

Given this initial representation, applications of the subprocedures EXPAND, IRREDUNDANT_COVER, and BOOLEAN_RESUB lend to the following changes.

- EXPAND: $x_1 x_2 \bar{y}_2$ is in SDC_3, which implies that $\bar{y}_2$ can be dropped from the first cube of f_3.
- IRREDUNDANT_COVER: $\bar{x}_1\bar{x}_2$ is in SDC_3, which implies that the second cube of f_3 is redundant.
- BOOLEAN_RESUB: $\bar{x}_1\bar{x}_2\bar{y}_2$ is in SDC_1, which implies literal $\bar{y}_2$ can be added to the first cube of f_1. The original literals of this cube are removed by the next EXPAND step, and the second cube of f_1 is removed by the next IRREDUNDANT_COVER step. Note that f_3 itself becomes redundant, leading to the final (optimal) representation.

In this example, the ESPRESSO_MLT procedure derives the optimum multilevel representation

$$f_1 = \bar{y}_2$$

$$f_2 = x_1\bar{x}_2 + \bar{x}_1 x_2$$

which has two functions, five literals, and three inputs. It has only two levels of logic and is 100-percent testable, with a complete test set 00,01,10 provided.

2) The Don't-Care-Based Approach: A second approach to node minimization is based on a more direct application of two-level minimization. Partial collapsing of the network usually precedes this in order to obtain larger functions at each node with more hope of significant minimization. As a further aid in the minimization process, don't cares in the form of a subset of the SDC and ODC are gathered. After the two-level form is minimized, it is factored and decomposed.

There are several problems with this approach.

1) The two-level minimizer has as its prime objective the number of cubes in the sum-of-products form.
2) The don't-care set is large, so when a minimizer like ESPRESSO, which computes the off-set, is used, the off-set produced may be huge.

Recently, several attempts have been made to overcome both these difficulties.

New methods for two-level minimization in the multilevel environment: The first idea is that the off-set should not be computed. One way of avoiding the off-set is to use tautology-based algorithms. Although these methods may lose some quality in the final cover, tautology-based methods can be used and can be quite effective when additional information about the typical case encountered in the multilevel application [85] is available. Pertinent information is that

- the don't-care set is usually represented mostly by primes,
- the initial cover of a node function is usually small, and
- there are many variables, due to the presence of many intermediate nodes.

The existence of many variables means that during the reduce operation, one creates cubes with many literals. Thus many trials must be attempted during tautological expansion. Since the final expanded cubes usually have a small number of literals, most of these expansions are successful. This leads to the idea of attempting to expand several variables at once. Generally, this speeds up the expansion in two ways. First, on the average, less tautology calls are tried, and second, more tautologies are unsuccessful (negative tautologies (NP complete) are faster than positive tautologies (co-NP complete)). Finally, in addressing the first problem of the preceding, one can choose heuristics which are directed toward few literals rather than few primes [85].

A second method of avoiding the off-set computation is to use a new concept called the reduced off-set [68]. This idea is the recognition that when expanding any one cube to a prime, the entire off-set is not required or useful. An example is the problem:

$$\text{on-set: } \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + abc$$

$$\text{off-set: } a\bar{b} + b\bar{c} + \bar{a}b.$$

Note that in expanding the term $\bar{a}\bar{b}\bar{c}$, the point abc in the on-set is of no use and may as well be included in the off-set during the expansion of $\bar{a}\bar{b}\bar{c}$. This leads to the reduced off-set $a + b$ for this expansion. The interesting points here are that the reduced off-set can be obtained without ever computing the off-set, and even though the reduced off-set is special for each cube to be expanded, only a few reduced off-sets will suffice for expanding the entire on-set. The reduced off-set concept can be combined with the normal ESPRESSO algorithms to produce a version based on reduced off-set [68] or can be used in a special reduction/expansion process aimed at multilevel applications [85].

Filtering the don't cares: Similar to the idea behind the reduced off-set, there are parts of the don't-care set which will not be useful in obtaining a minimum representation. In addition, there are parts which most probably will never be useful. Don't-care filters [84] are based on

- the form of the matrix representation of don't-care cubes, and
- the topology of the multilevel network.

In most applications so far, the don't-care set is derived entirely from the SDC. Suppose the SDC or a subset is represented in "cube-matrix form." The cube-matrix form is a matrix with rows corresponding to cubes, and columns corresponding to variables. An entry is 0 if the variable occurs in negative phase in that cube, 1 if in positive phase, and 2 if the variable does not occur in the cube. One can show that if the sum-of-products form of the current representation of the function plus the don't-care matrix has the block form

$$M = \begin{bmatrix} A & e & B \\ 2 & f & C \end{bmatrix}$$

where (e, f) is a single column and 2 is a submatrix of all 2's; and if the rows associated with f, C, are all don't care, then at least one minimum solution remains when the second block of rows $(2, f, C)$ are eliminated from the don't-care set. This is called an "exact filter" since no optimality is lost by its use. Since it is easy to detect the specified form of M given in the preceding, the exact filter should always be used.

Many "inexact filters" have been tried, and several have been found to be quite effective. One is the "subset-support filter" based on topological ideas [84]. This filter is applied when a node in the network is to be minimized. A don't-care set is generated based on the local SDC of only the nodes which have their support contained in the support of the node to be minimized. This greatly reduces the don't-care set and the run-time of the node-minimization process without much loss of quality compared to using the entire SDC. Using this filter, a more robust logic-minimization process can be built, since large off-sets are then rarely encountered. These filters can also be combined with the reduced off-set for further advantage. Currently, the subset-support method is the default process used in the MIS system when a node is simplified.

C. Transduction

The "transduction" (*trans*formation and re*duction*) methods originated with Muroga and students [55] in the middle 1970's. The ideas are intimately related to observability don't-care sets. Although transduction was given originally for NOR networks, it has been generalized recently to networks with different operators at the nodes (AND's OR's NAND's NOR's) [78]. The key idea behind transduction is that each node in the network is an incompletely specified function of the primary inputs. This is the same incompletely specified function discussed in section V-A, except that in the latter the functions are represented as functions of the intermediate variables also. The satisfiability don't-care set effectively relates these intermediate variables to the primary inputs. In transduction, these incompletely specified functions are called the "maximum set of permissible functions" (MSPF). The interpretation is that each function represents all possible permissible implementations at that node. As with other versions of this same concept, these functions are too expensive to compute in practice, so a subset is obtained (cf. section V-A). However, in transduction the subset obtained (called the CSPF) has some additional interesting properties.

1) Compatible Set of Permissible Functions: The subset of most interest is called the "compatible set of permissible functions" (CSPF). The word "compatible" is the important operative here. A CSPF at a node is a cleverly chosen subset of the MSPF. It is constructed so that a choice of representation of the CSPF at a node allows the already computed CSPF's at the other nodes to remain valid, in the sense that they are still a subset of the MSPF (which may be changed). The advantage of the CSPF is that a node or signal identified to be redundant (essentially a CSPF representing a function of only 0's or don't care) can be set to 0, i.e., removed from the network. Hence redundancy removal can be performed simultaneously by computing all the CSPF's and removing all signals identified as redundant. This is different from other methods of redundancy removal, which require that once one signal is removed, all other indicated redundancies must be verified to be redundant before they can be removed.

The computation of the CSPF's was given originally for NOR circuits only; it is based on ordering the inputs to a NOR gate. The highest ordered input is chosen to be the dom-

inating or controlling input, so that when it is 1, the other inputs can be don't care. Hence, for each input minterm, searching from highest to lowest order among the inputs of the NOR, the first input with a 1 is found. For that input signal, it is required that it compute a 1, but for all other inputs, for that miniterm, a 1 or 0 is permissible. Starting at the outputs, the external don't cares DX_k for the output are put in the permissible function. The computation proceeds from outputs to inputs. A CSPF is computed for each fan-out and each node. If a node has a don't care at a minterm, then each of its NOR inputs is also don't care. A node's output CSPF is computed as the AND of all its fan-out CSPF's. Note that even though these computations are stated in terms of minterms, which implies a truth-table-like inspection, they can be formulated using BDD's, as done recently at Fujitsu [71].

Like BDD's, the order imposed for the CSPF computation is important. The observation that the signal with the least order will inherit the most don't cares motivates an ordering heuristic which ranks the signals in increasing order of the probability of being redundant. These probabilities can be estimated by random simulation. Another heuristic assigns highest order to signals with the greatest number of miniterms in the on-set of the function. Since 1's are the controlling value for NOR gates, this heuristic attempts to introduce more don't cares in connections with lower order signals and hence increases the likelihood of these signals being removed.

2) Transformations and Reductions: The transduction method is based on the following CSPF-based transformations and reductions:

1) redundant circuits pruning,
2) connection addition and deletion,
3) connection substitution,
4) gate substitution,
5) gate merging.

We illustrate the general ideas by discussing two of these.

Gate Merging: Gate merging is illustrated with NOR gates. Two gates g_1 and g_2 can be replaced with one gate g by the following computation.

1) Find two gates such that $\text{CSPF}(g_1) \cap \text{CSPF}(g_2) \neq \phi$.
2) Form a new gate g which is the NOR of all the inputs of g_1 and g_2.
3) Remove g_1 and g_2 from the Boolean network.
4) Make g fan out to all fan-out points of both g_1 and g_2.

Connection/disconnection: This transformation is similar to the REDUCE process in two-level minimization (*cf.* section V-B-1). "Connection" is simply concerned with adding an extra input to a NOR gate. This can be done if the new function of the NOR gate, with the extra input, remains with the CSPF after connection. It is relatively easy to make this calculation using BDD's. Disconnection is simply redundancy removal.

The objective of the connection/disconnection transformation is to make a particular gate g redundant. This is done by making all possible direct connections from the transitive fan-in of g to gates in the transitive fan-out of g. The hope is that these connections will effectively bypass g and thus make g redundant.

D. Global Flow

"Global flow" [95], [9], [10], [93] is a technique that has been employed extensively in the LSS system [35] and is based on compiler optimization methods. In contrast to transduction, global flow tries to minimize the fan-out of a chosen gate, using a more global view via cut-sets of a derived graph. Also, the connections made by global flow are derived from implications instead of CSPF's.

1) Implications: Global flow analysis collects information (implications) of the form $y_i = b_i \Rightarrow y_i = b_j$, where y_i and y_j are inputs or internal signals in a Boolean network, and b_i, $b_j \in \{0, 1\}$. This information is collected in sets, called forcing sets, denoted by $\mathcal{F}_{ij}(x)$, where $i, j \in \{0, 1\}$ and x is a signal in the network. For example, if $y \in \mathcal{F}_{10}(x)$, then $x = 1 \Rightarrow y = 0$.

2) Relation to Don't Cares: The sets $\mathcal{F}_{ij}(x)$ are directly related to two-literal cubes of the SDC set.

Theorem V-3: In a Boolean network, $y_i = b_1 \Rightarrow y_j = b_2$, $b_k \in \{0, 1\}$ if and only if $(y_i = b_1)(y_j = \bar{b}_2)$ is an implicant of the SDC of the network.

Thus, $y_i = 1 \Rightarrow y_j = 1$ can be expressed equivalently as: $y_i\bar{y}_j$ is a don't care. The cube $y_i\bar{y}_j$ is part of the SDC of the network. In some cases, $y_i\bar{y}_j$ may not appear explicitly as a term in the summation of the individual local SDC's; SDC $= \Sigma(y_j \bar{f}_j + \bar{y}_j f_j)$. However, this implicant will result from the (possibly iterated) consensus of cubes belonging to different terms of the SDC. Since iterated consensus produces all primes, either $y_i \bar{y}_j$ results, or one of the literals y_i or $\bar{y}_j$ is a prime of the SDC. In the latter case, the associated gate is completely redundant, since its output value is a don't care.

3) Computing Approximations to the Forcing Sets: Since computing the complete forcing sets $\mathcal{F}_{ij}(x)$ is too expensive in general the subsets $\mathcal{C}_{ij}(x) \subseteq \mathcal{F}_{ij}(x)$ are used.

These subsets can be computed using recurrence relations and are defined to be the least fixed point of these relations. An important feature is that these subsets can be computed simultaneously for all x.

In [93], the functions f_j at the nodes in a network are restricted to NOR's. The recurrence relations defined for subsets $\mathcal{C}_{10}(x)$ and $\mathcal{C}_{11}(x)$ of $\mathcal{F}_{10}(x)$ and $\mathcal{F}_{11}(x)$, respectively, are given for NOR gates only, by the following equations, where the relation (y, s) means that y is an input to the gate with output signal labelled s.

$$\mathcal{C}_{10}(x) = \mathcal{C}_{10}(x) \cup \{s: \exists (y, s)[y \in \mathcal{C}_{11}(x)]\} \cup \{s: \exists (s, y)[y \in \mathcal{C}_{11}(x)]\} \cup \{s: x \in \mathcal{C}_{10}(s)\}$$

$$\mathcal{C}_{11}(x) = \mathcal{C}_{11}(x) \cup \{s: \forall (y, s)[y \in \mathcal{C}_{10}(x)]\} \cup \{s: \exists (s, y), y \in \mathcal{C}_{10}(x), \forall (t, y) \cdot [t \neq s \Rightarrow t \in \mathcal{C}_{10}(x)]\} \cup \{x\}. \quad (2)$$

These relations can be solved for their least (smallest set) fixed point by iteration until no change in any set occurs. Additionally, subsets of these can be computed even more efficiently if some of the clauses in (1) and (2) are omitted. Indeed, part of the computation, (1) and (2), of the sets $\mathcal{C}_{ij}(x)$ for all x amounts to a kind of restricted iterated consensus. For future reference, we remark that the fourth clause in (1) is called the "contrapositive" implication.

4) The Global Flow Method—Reduction and Expansion: The process described in [95] focuses on a signal x and modifies its fan-out in such a way that it preserves the behavior but reduces the network in some way, e.g., wiring, area, or timing. This process is described as a reduction–expansion iteration, in analogy with the way that two-level logic is minimized heuristically [22].

The first process is a reduction which shows how any signal y in $\mathfrak{F}_{1j}(x)$, $j \in \{0, 1\}$ can be altered to include x. More precisely, this is as follows.

Theorem V-4. The following transformations on a Boolean network are valid.

1) If $y_i \in \mathfrak{F}_{11}(x)$, then replace f_i with $x + f_i$.
2) If $y_i \in \mathfrak{F}_{10}(x)$, then replace f_i with $\bar{x} f_i$.

This process is called "reduction," since it adds literals to the network and is analogous to the REDUCE operation in [22].

Definition 1: The 1-frontier of a signal x is defined as the set of signals s such that

1) $s \in \mathcal{C}_{1i}(x)$, $i \in \{0, 1\}$,
2) there is a path $(s, j_1, j_2, \cdots, \text{OUTPUT})$ such that no j_k is in $\mathcal{C}_{1i}$,
3) in the directed graph of the network, s is reachable from x, i.e., s is in the transitive fan-out of x.

Definition 2: Let $G_1(x)$ be the graph consisting of nodes $j \in \mathcal{C}_{1i}(x)$, j in the transitive fan-out of x, and edges (j, k), $j \in S_{fk}$, $k \in \mathcal{C}_{1i}(x)$. In addition, add edges (j, k) if there is a path from j to k, where the only nodes along the path in $\mathcal{C}_{1i}$ are j and k. A 1-cut-set of a signal x is a set of signals in any of the $\mathcal{C}_{1i}(x)$, $i \in \{0, 1\}$, which in $G_1(x)$ separates x from its 1-frontier. For example, the 1-frontier itself is such a cutset.

A similar set of definitions holds for a signal's 0-frontier and corresponding 0-cut-set.

Theorem V-5: If all the gates of a chosen 1-cut-set (0-cut-set) of x are reduced according to Theorem V-4, then any gate $k \in G_1$, $k \notin$ 1-cutset (0-cutset) may be expanded by replacing F_k by $F_{k\bar{x}}$ (F_{kx}).

This process is called "expansion" since it removes literals from the network, analogous to the EXPAND operation for two-level minimization. As seen from Theorem V-3, the $\mathcal{C}_{ij}$ sets are related to the SDC. The 1-cutset and 1-frontier are related to ODC_i.

The global flow algorithm is as follows.

1) Compute the sets $\mathcal{C}_{ij}(x)$ for all x in the network.
2) Choose a signal x and value $i \in \{0, 1\}$.
3) Using the $\mathcal{C}_{ij}(x)$ set, find the i-frontier of x.
4) Find a minimum weighted i-cutset in G_1 of signals separating x and its i-frontier.
5) Reduce the i-cutset according to Theorem V-4 (reduction).
6) Cofactor the remaining fan-out of x in G_1 according to Theorem V-5 (expansion).

The contrast between the reduce/expand process of global flow, and the reduce/expand process of two-level minimization is illuminating. In two-level minimization, we expand or reduce by selecting a cube and then expand or reduce for all literals of that cube. In global flow, we select one literal and expand or reduce many cubes in the Boolean network associated with that literal.

One needs good heuristics to choose which signal x to use, and which criterion to use in finding the minimum cut-set. After a network has been changed, it is also necessary to recompute the sets $\mathcal{C}_{ij}(x)$. It is more efficient to do this incrementally, but no efficient incremental update procedure is known.

5) Free Nodes: A recent improvement on these ideas [8] is the observation that certain nodes, called "free nodes," can be deleted from the graph of implied nodes. This reduces the size of the cutset and helps simplify the network even further. A free node z is one that is implied by a node y in the forcing set of x, but y is not in the transitive fan-out of x. Since the free node z may be in the transitive fan-out of x, deleting it in the graph G_1 makes G_1 smaller. Thus the cutset of G_1 is potentially reduced, which means that the expansion process can be more effective. Note that only the free nodes that are also in the frontier are omitted from the graph. This may reduce the size of the cutset by reducing the size of the "cone" of implications in the transitive fanout of the signal x. Leaving the other free nodes in the graph cannot increase the size of the cutset.

E. Node Invariance

The use of various satisfiability and observability conditions in networks in multilevel synthesis [6] was discussed in section V-A. Use of don't-care sets which arise from the topological structure of the network has been the focus of several algorithms. Further, implications through the use of $\mathcal{C}$-sets and $\mathfrak{F}$-sets have been shown to be equivalent to a subset of the SDC set (cf. section V-D-1). However, the use of these various forms of don't-care sets has been impeded historically by the feeling that transformations on the network invalidated them, and so recomputation was essential after each transformation. Since recomputation is fairly expensive, interest has centered on incremental updates. The focus of invariance research [75] has been to discern which transformations affect these sets and how they are affected.

The general process of logic synthesis can be viewed as a sequence of well-defined "transformations" of the network which yields a network realizing the same function. The transformations of the network may be divided into those that change the dimensionality of the extended space and those that do not. Clearly, transforms which change the space change the SDC set, but they do so in such a way that the set over the expanded (or contracted) space can be easily derived. For example, the addition of a dimension requires the addition to the SDC set of precisely those points contained in the "local SDC" of the function being added; hence we can write

$$\text{SDC}' = \text{SDC} + (y_j \oplus f_j).$$

The effect of deleting a variable is similar and can be expressed as

$$\text{SDC}' = \text{SDC}_{y_j}\, \text{SDC}_{\bar{y}_j}.$$

The question "when does a transformation leave the SDC set fixed?" can be rephrased as "what information permits a transformation to change the 'global function'[5] at a node,

[5]The global function at a node is the function viewed as a function of the primary inputs.

and so to change the SDC set?" The answer is that the *only* information which permits a transformation to change the global function of the node is precisely the don't-care set for the node defined over the primary inputs. In general, besides the SDC, which is the null set as seen from the primary inputs, there are external and observability don't-care sets. However, many transformations in practice do not use them. Formally, a transformation does not use a don't-care set if the transformation remains invariant when the don't-care sets are reduced to ϕ. In such a case, the transformation must leave the SDC invariant.

Since the don't-care sets for a Boolean network η are specifications of a circuit, we can formally define a "circuit" as an ordered pair (η, D), where η is a Boolean network and D is a vector of don't-care sets; D_i is the don't-care set for node f_i on a network. D_i is equal to the union of the external and observability don't-care sets of node f_i.

A "transformation" $\mathfrak{I}$ is a mapping from a circuit (η, D) to a new circuit (η', D'). In general, $\eta \not\equiv \eta'$ and $D \not\equiv D'$. In this discussion, we consider only transformations where the set of nodes is preserved by the transformation.

A transformation $\mathfrak{I}:(\eta, D) \Rightarrow (\eta', D')$ is said to be "invariant" if for each node y_j, the global function $\tilde{f}_j$ of y_j is preserved. If the global function of each node is preserved, then the SDC must be preserved. The key result on the preservation of these sets is as follows.

Theorem V-6: Let $\mathfrak{I}(\eta, D) = (\eta', D')$ be any transformation of a network. $\mathfrak{I}$ is invariant iff for every $\tilde{D}$, $\mathfrak{I}(\eta, \tilde{D}) = (\eta', D'')$ (i.e., η' is obtained independent of D).

Interestingly, this theorem implies that all the transformations within the MIS-II [16] synthesis system, with the sole exception of "node-simplify," do not change the SDC. In particular, the forcing sets, used in global flow and other applications, remain invariant under these transformations. Further, "node-simplify" does not change the SDC unless external or observability don't-cares are used.

F. Hierarchy of Networks

Although don't cares are a powerful source of degrees of freedom for optimizing a network, they do not and cannot capture all the degrees of freedom. This observation [20] has led to research on additional methods for describing and using this flexibility.

1) Insufficiency of Don't Cares and Boolean Relations: In a logic network specified by a hierarchy where one block of logic feeds another, it has been observed that don't cares are not sufficient for representing all the flexibility with which each block can be simplified. An example is shown in Fig. 12, where the first block, an adder, feeds its output to a comparator. We consider the effect on the minimization of the adder due to the filtering effect of the comparator.

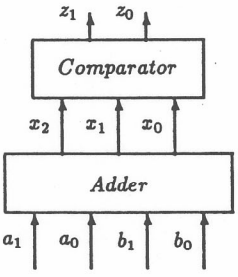


Fig. 12. Hierarchical network.

The function of the comparator is given by

$$z = 01 \Rightarrow a + b < 3$$
$$z = 00 \Rightarrow (a + b = 3) \vee (a + b = 4)$$
$$z = 10 \Rightarrow a + b > 4.$$

Input values 000, 001, and 010 are not distinguished by the comparator; thus {000, 001, 010} forms an equivalence class. The other equivalence classes are {011, 100} and {101, 110, 111}.

This leads to a specification for the adder that takes account of this additional flexibility:

$a_1a_0b_1b_0$	$x_2x_1x_0$
0000	{000, 001, 010}
0001	{000, 001, 010}
0010	{000, 001, 010}
0100	{000, 001, 010}
1000	{000, 001, 010}
0011	{011, 100}
0101	{000, 001, 010}
0110	{011, 100}
1001	{011, 100}
1010	{011, 100}
1100	{011, 100}
0111	{011, 100}
1011	{101, 110, 111}
1101	{011, 100}
1110	{101, 110, 111}
1111	{101, 110, 111}

This table is interpreted as a truth table where the set listed to the right of an input miniterm is a list of acceptable outputs of the implementation. This is an example of a Boolean relation which is a generalization of a Boolean function. In general, a "Boolean relation" is a one-to-many mapping; for each input miniterm there can be more than one acceptable output pattern. A don't care on an output is a special case of this. For example, for the miniterm 0000 in the example, we could express one set of choices for the outputs as 00−, which says that the outputs could be either 000 or 001; we don't care if output x_0 is 0 or 1. However, this don't care does not express that the output could also be 010. In fact, there is no way to express the set {000, 001, 010} with only output don't cares. In the example, if we use only don't cares in the outputs, the best choice of don't cares, constrained to include the normal adder as an acceptable implementation, leads to the minimized two-level function, as follows.

$a_1a_0b_1b_0$	$x_2x_1x_0$
11-0	011
-110	011
10-1	011
-011	011
-111	100
11-1	100
111-	010
1-1-	100

On the other hand, a special minimizer [90], which is based on an extension of the Quine–McCluskey method to Boolean relations, produces a much simpler minimum solution.

$a_1a_0b_1b_0$	$x_2x_1x_0$
0–1–	010
1–0–	010
1–1–	100
––– 1	001
–1––	001

2) Minimizing Boolean Relations: The key observation in minimizing a Boolean relation (denoted $\mathcal{R}$) is the notion of compatible functions. A Boolean function f is *compatible with* $\mathcal{R}$ ($f \prec \mathcal{R}$) if

$$f(x) \in \mathcal{R}(x), \forall x \in B^r$$

where $\mathcal{R}(x)$ denotes the set of all output patterns acceptable for the input miniterm x. The minimum implementation is some function compatible with $\mathcal{R}$. This leads to the notion of a c-prime, which is a prime of any function compatible with $\mathcal{R}$. By a slight modification of the consensus operation, it is possible to generate all c-primes of $\mathcal{R}$ without having to generate the set of all functions f compatible with $\mathcal{R}$, a much larger set. After the set of c-primes is generated, the selection of an optimum set of c-primes is done by solving a binate-covering problem [90]. Although this procedure is expensive, there is hope that a suitable heuristic minimizer, based on the ESPRESSO expand/irredundant/reduce paradigm, can be constructed.

Another context in which Boolean relations arise is a finite-state machine with sets of equivalent states. In any implementation of the FSM, it is not important which state among an equivalent set is used, and in fact, all the equivalent states may be useful in constructing a minimum implementation. In the machine, the condition (input, old-state) ⇒ (newstate, output) may be implemented by using any one of the states equivalent to newstate. This gives rise to "symbolic" Boolean relations [66].

VI. Logic Synthesis and Testing

Today, when a designer implements a combinational logic design, consisting of a collection of primitive (e.g., NAND, NOR) or more complex logic gates, the designer is working toward a multivariate objective—to meet area, speed, reliability, and technology constraints while *also* trying to ensure the design is testable with a small set of patterns. This latter task requires that the designer eliminate redundancy during the test phase and perhaps add additional test points to the circuit. Needless to say, it is almost impossible for the designer to find an optimal trade-off of all of these competing objectives for anything but the most simple circuits, or when one or more of these constraints is very loose (e.g., when circuit delay is not an important factor). For this reason, it has been the case that designers will first attempt to find a feasible solution that meets performance and area goals and later modify the design, if necessary, to meet test requirements. Testing, even for combinational circuits, has been a post-design activity.

Logic synthesis will impact in a significant way on testing and design for testability. In fact, as we have seen previously, the optimization process carried out during logic synthesis yields a circuit that, ideally, has no redundancy, and as such is 100-percent testable. Some techniques, such as multilevel tautology, also yield test vectors as byproducts. Conversely, automatic test pattern generation (ATPG) identifies redundant faults and hence can be used to eliminate redundant logic.

In this section, we take a close look at the relationships between logic optimization and testing, and evaluate the effect that logic synthesis transformations have on the testability of a combinational circuit.

A. Equivalence, Logic Minimization, and Testing

The problem of verifying that two Boolean networks are equivalent, without regard to their internal structure, will be referred to as the "general equivalence" problem (sometimes called "formal verification"). This problem often arises in the context of checking whether the *overall* effect of some logic synthesis tool has altered the behavior of a given network. However, in some cases, like ATPG (automatic test pattern generation) or logic minimization, it is known that the two networks differ only in one literal of one cube of one function. In this case the equivalence question is referred to as the "constrained equivalence problem." This latter problem is much simpler and can be solved efficiently in practical cases. However, both problems are co-NP complete. ATPG, being related to the negation of constrained equivalence, is NP complete.

All formal methods for multilevel equivalence are exhaustive in the sense that if a primary input variable assignment exists which causes a primary output variable to evaluate to a zero, then this assignment will be discovered. Often a form of binary recursion is employed to systematically search all possible primary input variable assignments; and both the general and constrained equivalence algorithms employ a framework based on recursive binary partitioning of the Boolean n-cube B^n. At each step in this partitioning, the representation is duplicated, a new (single) variable assignment is made to effect the partitioning, and the logic of each partition is simplified with an evaluation (simulation) procedure. The heuristic for selecting this variable is very important, since efficiency rests on pruning the binary recursion through identification of special cases.

There are techniques, for example, subtree matching [57], for solving the constrained problem that are definitely not applicable to the general problem. In addition, techniques for constrained equivalence, such as unique sensitization and multiple path backtrace [33], may not be applicable to the general problem. Generally, no one method or program written for general or constrained equivalence is appropriate for all problems. Thus a different set of tools is required for these two classes of problems.

B. The ATPG Process

In the context of this paper, we view ATPG as a specific case of the constrained equivalence process in which a fault is asserted (that is, a change is made to one node of the Boolean network) and a contradiction is sought, proving the

modified network is not equivalent to the original. The satisfiability don't-care set, which has already been shown to play a major role in logic minimization (cf. section V-B-2; also [85]), is also very significant in the ATPG context. In this context, which is dominated by simulation techniques, satisfiability don't cares are identified when implications of logical assertions are contradictory. When this situation occurs, it may be referred to as a "satisfiability contradiction." Classical work on using implications in ATPG [44] exploited the forward implications that could be derived from simulation, but backward, or contrapositive, implications were basically ignored. FAN [43] improved on this by accounting for the contrapositive implications indirectly in its "multiple path backtrace" procedure. However, until recently, ATPG work appears to have lagged behind techniques such as the work on global flow (section V-D), which accounted for all implications, including the contrapositive. Schulz changed that by pioneering the use of contrapositive implications in the ATPG context, and referred to his technique as "learning," since the contrapositive implications could not be deduced during simulation but had to be "remembered" from the results of previous simulations.

Based on improved unique sensitization, and on his learning technique, Schulz discovered an ATPG approach [87], [86] modeled after FAN [43] which obtained remarkable results. Namely, tests for all irredundant faults could be obtained with a backtrack limit of ten and no aborted faults, while identifying all the redundant faults. Empirical evidence (remember that ATPG is NP complete) is that the following can be true even for "industrial strength" circuits such as the ones in the ISCAS benchmark set.

- Constrained equivalence checking for redundant faults can be performed in time linear in the size of the network.
- Constrained equivalence checking for irredundant faults can be performed in time linear in the size of the network.
- Fault simulation can drastically reduce the time required for constrained equivalence checking.

Schulz obtained his results with a three-pass methodology for ATPG. Each succeeding pass would invoke more powerful methods for the ATPG process. Random test pattern generation followed by fault simulation was applied prior to invoking any of the deterministic ATPG algorithms, and a parallel fault simulator was invoked after each testable fault was identified. Similar results were obtained in [33].

Since there were no aborted faults in these experiments using the ISCAS benchmarks, and the backtrack limit was fixed at ten, the binary recursion tree had not more than eleven leaves on any individual constrained equivalence check. Since, on any given leaf of the recursion tree the simulation/implication work is of linear complexity, and since fault simulation itself is of linear complexity, it is clear that, for these examples, this type of equivalence checking is, effectively, of linear complexity. It was observed that in cases where there are no redundant faults, fault simulation dominates the overall CPU time consumption (ISCAS examples c880, c499, c6288). It is interesting that for c6288 (a 16-bit multiplier), all faults but one are detected by fault simulation of the 2000 random patterns, so the constrained equivalence checker is called only once.

C. Redundancy Identification and Removal

In a sense, ATPG can be thought of as "redundancy identification" (RI). ATPG technology can also be easily harnessed to the task of "redundancy removal." In this methodology, ATPG tools identify redundancies one at a time and put them into a list. Note that a redundant fault implies that part of the logic is redundant and can be removed. However, only the logic corresponding to the first fault in the list can be removed safely. In fact, other faults may not be redundant any more because of the change in the logic. All the other faults have to be retested after the first fault has been removed, although in practice we observe much independence between the redundant faults. Once the entire list has been rechecked, the entire ATPG/RI process can be repeated. These two steps are iterated until all redundancies are removed. In this way, prime and irredundant networks could be obtained in times ranging from seconds to tens of minutes on a Sun4/280. Possible heuristic improvements on this basic approach are numerous, including those suggested in [28].

An alternative redundancy removal approach was proposed by Brand [9]. In this approach, local topological searches were used to identify redundant nodes in the logic. However, this method was not exact in the sense that it did not identify all redundant nodes.

It is of special interest to speculate on the "optimality" of a circuit that has been designed by the following two-step process.

1) Determine an initial overall structure, perhaps using the algebraic decomposition methods of section IV.
2) Use ATPG techniques like those discussed in the preceding to identify and remove all discovered redundancies [28], [33].

Some initial results on this subject are presented in [33]: *all* of the ISCAS benchmarks can be reduced to prime and irredundant form in less than 1 hour, overall, on a Sun 4/280. Compared to MIS optimization results reported on these ISCAS benchmarks [84], the redundancy removal program obtained roughly similar optimization quality overall. A speed advantage of more than 20 was observed favoring simple redundancy removal. On the other hand, the full MIS script includes restructuring as well, and it has been observed that restructuring has very little effect on the ISCAS circuits. Thus a second experiment was done recently where the MIS operations were restricted to those analogous to the ones performed in [33], essentially node minimization and cube extraction. Similar results, in terms of quality, were obtained. The time comparisons varied from a factor of 50 favoring redundancy removal, to a factor of 4 favoring node minimization. Quality varied from 121 fewer literals favoring redundancy removal to 807 fewer literals favoring node minimization. The speed comparisons show that redundancy removal is surprisingly fast in some cases, making this an effective technique for logic minimization. The literal comparison also shows that redundancy removal is quite effective; however, it should be cautioned that the ISCAS examples are known to be special and to represent only a subclass of circuits. At the time of this writing, more examples need to be done to fully position redundancy removal, using the efficient techniques presented in [33], within the other operations available for multilevel logic synthesis.

D. Synthesis for Testability

A complete system for "easily" testable logic was reported in [27]. With regard to testability which is guaranteed complete in one sense or another, several papers have appeared. One approach [4] for area optimization is guaranteed to produce logic that is prime, irredundant, and 100-percent testable for single stuck faults. Further, in this approach, the tests are provided as a byproduct of the optimization. Obviously, the procedures for performing such optimization efficiently have a strong correlation with ATPG algorithms [42], [87], [86].

More recently [47], [39], [63], [81] logic synthesis has begun to focus on testability itself as an optimization criterion. These papers pursue the ideal of optimizing for complete testability while still striving to reduce network area through algebraic transformations. The methods discussed in [4], [47], [39], and [63] all produce logic which is 100-percent testable for single stuck faults. The method of [39] does this for finite-state machine logic. References [47] and [63] produce logic which is 100-percent testable for multiple stuck faults as well, but [63] is restricted to a class of three-level CMOS circuits and [47] is restricted to a general class of circuits that can be synthesized by the class of algebraic transformations discussed in section IV-C. The methods in [63] and [81] produce logic which is 100-percent testable for path delay faults as well. In the finite-state machine context, [32] gives a method for state assignment which produces an initializable reset state and overcomes many of the problems associated with sequential testing.

1) 100-Percent Multifault Testable Networks: In [52], [47], a theory was outlined for relating 100-percent testability for two-level prime and irredundant Boolean networks to 100-percent multifault testability for these same networks. Much is known about this subject from early research (see [25, p. 65] and its references), but since then new possibilities arose with the advent of algebraic decomposition as a primary tool for area optimized synthesis of multilevel logic. In [47], it was shown that for networks derived by a suitable class of algebraic decomposition operations, all tests needed for 100-percent multifault testability can be derived from the underlying equivalent two-level structure. A procedure was given that can produce area-optimized (but not necessarily optimum) multilevel networks which are 100-percent multifault testable when this two-level structure either is given or can be computed. Also, identification was made of classes of algebraic and topological optimization operations under which the 100-percent multifault testability property is invariant.

A testability property called SPI (simultaneously prime and irredundant) was defined in [52], and it was shown that a *set* of prime and irredundant two-level *single-output* functions was SPI. Briefly, if a network is SPI, removal of *any* set of nodes, literals, or cubes produces a nonequivalent network. Obviously, if a network is SPI, it is multifault testable. For NAND networks, SPI and multifault testability are equivalent concepts. Thus SPI can be thought of as the appropriate generalization of the classical multifault testability property to the case where an individual node in the Boolean network can have an arbitrarily complex Boolean function associated with it. Thus faults *internal* to these functions need to be accounted for as well.

In [47], it was shown that the SPI property was invariant under a large class of algebraic and technology mapping transformations; hence a path was established for producing multilevel, technology mapped, 100-percent multifault testable networks which have had the benefit of algebraic minimization operations.

2) Testability Invariance: In [47], it is demonstrated that algebraic methods preserve testability. This result is then applied to networks that start as single-output optimized two-level functions. Under these conditions, the initial network is completely multifault testable. Hence if only algebraic procedures are used, the complete multifault testability will be preserved in the derived multilevel network. However, it makes sense to ask under which general conditions transformations used in logic synthesis preserve testability.

We discuss first the condition that the output of a node is testable for stuck-at-faults. (This can be made equivalent to edge testability by artificially inserting a buffer on each edge; the testability of the output of the buffer becomes equivalent to the testability of the edge.)

As discussed in section V-A, the observability function of a node x at a node with function f is defined as

$$\frac{\partial f}{\partial x} = f_x \oplus f_{\bar{x}}$$

(assuming the external don't-care sets are empty); and the observability function of a node x over a network is the sum of its observability at each primary output:

$$\overline{\mathrm{ODC}}_x = \sum_{\text{outputs } f_i} \frac{\partial f_i}{\partial x}.$$

The set of tests for stuck-at-1 and stuck-at-0 are therefore, respectively, $\bar{x}\overline{\mathrm{ODC}}_x$ and $x\overline{\mathrm{ODC}}_x$. Thus, to determine when the observability of node x is unaffected, it is sufficient to determine when the cofactors of f_j with respect to x are unaffected. For this, it is useful to have a geometric picture of the cofactor of f with respect to a node y_j in a network.

The cofactor of f with respect to y_j is the set of possible points—that is, points outside the SDC—where $f = 1$ and $y_j = 1$; in other words, the set $\overline{\mathrm{SDC}}_{fy_j}$. However, it is not quite correct. A point on the quarter-space fy_j may be impossible because $y_j = 1$ may be impossible for that vertex. In general, the "semantics" of cofactor is that y_j is stuck to 1 without regard to the value of f_j. Therefore we must exclude from SDC the set of points $y_j\bar{f}_j$ when considering the cofactors with respect to y_j, and the set $f_j\bar{y}_j$ when considering the cofactors with respect to $\bar{y}_j$. This can be done by considering the set defined in section V-A:

$$\mathrm{SDC}_j = \sum_{k \neq j} y_k \oplus f_k.$$

Theorem VI-1: The cofactor of f with respect to y_j is the set of vertices on the extended space in the set

$$(\overline{\mathrm{SDC}}_j)_{y_j f}.$$

It would seem that every transformation of any strength at all would affect the test set of at least some node, so the problem is not to characterize the transforms which leave the entire set of tests invariant but, rather, to characterize for a given transform which don't-care sets are left invariant. Fortunately, these are easily characterized.

Theorem VI-2: Let $\mathfrak{I}(\eta, D)$ be an invariant transform. If SDC_j is left unchanged by $\mathfrak{I}$, then the observability function of y_j is unchanged.

To determine which transforms leave SDC_j unchanged, we observed that a) an invariant transform preserves the cofactors with respect to every primary input; and b) if the function attached to node y_j is ignored by the transform, then y_j is indistinguishable from a primary input and so its cofactors, and hence its observability function, are preserved by the transform.

As an example, consider the transformation given by the algebraic division procedure [74], [16]. Algebraic division examines only two functions, those of the divisor and the dividend. Hence it follows that at most the testability of the divisor and the dividend are affected. This observation also holds for the Boolean division algorithm in Fig. 13 [16], since

```
bool-divide(f, g)

    q, r ⇐ alg-divide(f, g)

    minimize r with respect to the DC set (x ⊕ g) + qx

    (quotient, remainder) ⇐ alg-divide(r, x)

    return (quotient + q, remainder)
```

Fig. 13. Boolean division.

the only functions referred to in this procedure are those of f and g. Hence this Boolean division procedure has no greater or lesser effect on testability than does the algebraic procedure. Although this result seems counterintuitive, the fact that this Boolean division procedure uses only the local SDC associated with the divisor means that in some sense it has no more information than the algebraic procedure.

More powerful Boolean division procedures (cf. section IV-G-1) expand the don't-care set somewhat to gain more information. Hence these revised procedures can affect testability and redundancy more globally. However, a transformation can only affect the observability of those nodes y_j such that the effect of the transformation is dependent upon the function f_j.

The effect of various transformations on the observability of various nodes are stated in the following. The informal proof of each result is that the named transformation is dependent only upon the functions of the named nodes.

Corollary VI-3: Algebraic division and the Boolean division of Fig. 13 affect the observability function of (at most) the divisor and dividend.

Corollary VI-4: Collapsing node f into node g affects the observability function of (at most) f and g.

Corollary VI-5: Adding node f to or deleting node f from the network does not affect the observability function of any node $g \neq f$.

Further, if one attempts to simplify a node using the subset-support filter [84] (cf. section V-B-2) on the SDC and using no external or observability don't-cares, one can use these results to make strong statements concerning the preservation of testability of most of the network. Finally, using the duality of node and edge testability, if the observability function for each fan-out edge of a node is preserved, then the observability for the node is obviously preserved. Thus the following is known.

Theorem VI-6: Algebraic and the Boolean division of Fig. 13 do not affect the observability function of the dividend, f. Similarly, collapsing node g into node f does not affect the observability function of f.

This theorem has an interesting and counterintuitive consequence. Only the observability of the node g (respectively, the divisor or the node being collapsed) is affected by the division or collapsing transformation. This runs deeply counter to one's intuition, which suggests that the inputs to g are similarly affected, since these inputs have paths through g to the output. However, if g in fact divides into f in a nontrivial way, then every input of g is already an input of f. Hence the paths from the inputs of g to primary outputs are neither created nor destroyed.

VII. Technology Mapping

After a technology-independent optimization of a set of logic equations, the result must be mapped into a feasible circuit which is optimum with respect to area and satisfies a maximum critical-path delay. The role of technology mapping is to finish the synthesis of the circuit by performing the final gate selection from a particular library. The algorithms chosen for technology mapping are made less complex because they can be constrained by the structure produced by the technology-independent optimizations. It is not the role of technology mapping to change the structure of the circuit radically; for example, by finding common subexpressions between two or more parts of the circuit. Likewise, it is not the role of technology mapping to reduce the number of levels of logic along the critical path. The role of technology mapping is the actual gate choice to implement the equations—for example, choosing the fastest gates along the critical path and using the most area-efficient combination of gates off the critical path.

There are several characteristics which are desirable for a technology mapping algorithm: it should

1) adapt easily to different libraries,
2) support irregular collections of logic functions,
3) handle detailed technology-dependent cost functions,
4) be efficient in execution time.

First, it is desirable that the technology mapping algorithm be able to adapt to a variety of different libraries with minimal effort. This is difficult because many libraries have an irregular collection of logic functions available as primitives. An algorithm which depends on characteristics of a particular library (for example, availability of a complete set of CMOS and-or invert gates) is of limited use. Also, an algorithm which is geared to a subset of the gates in a library is limited in its optimization potential. To achieve the goal of library adaptability, technology mapping should be "user-programmable," i.e., the user should be able to provide new gates to the technology mapper without understanding its detailed operation, and these gates should be used effectively.

During technology mapping, simple cost functions such as transistor count or levels of logic will not provide high-quality circuits. Instead, it is necessary to consider more detailed models for the cost of a gate in the actual target technology. This detailed level of modeling, coupled with

gates which have irregular area and delay cost functions, greatly complicates the technology mapping process.

Therefore, to provide high-quality results for different libraries and circuits, a technology mapping algorithm must make few assumptions about the relative cost and performance of the gates in a library, and must be prepared to model accurately the cost functions which are to be optimized.

While it is always desirable to have an efficient algorithm, generally the execution performance of the technology mapping algorithm is less important than the quality of the final result. This is true for the last optimization of a circuit before fabrication. However, the steps of technology-independent optimization and technology mapping are often iterated by a logic synthesis system if the performance goals are not initially met. Technology mapping in this case operates as an accurate predictor of the quality of a technology-independent representation. These results are fed back to the technology-independent optimization to improve the final implementation. Therefore it is desirable that a technology mapping algorithm support a fast execution mode as well as a higher-quality optimization.

The two basic approaches used for technology mapping are

1) rule-based techniques [35], [46], [56];
2) graph covering techniques [60], [16].

Rule-based techniques have the same structure as rule-based techniques for technology-independent optimization [35]. These are discussed in section VIII. It is important to mention that a rule-based system can combine the technology-independent and technology mapping stages, providing, in principle, a more global view of logic optimization. However, the nature of rule-based systems is to perform local optimization. Thus the rule-based method provides an interesting contrast with the two-phase approach which separates the technology-independent and dependent phases but which offers a more global view within each of the phases.

The local-transformation/rule-based techniques have suffered historically from inflexibility and large execution times; however, they have demonstrated the ability to produce high-quality results.

In this section, we focus on techniques based on graph covering. These techniques match quite well the requirements discussed above.

A. Graph Covering and Technology Mapping

The approach of using directed-acyclic-graph (DAG) covering for technology mapping in logic synthesis was first proposed by K. Keutzer of AT&T Bell Laboratories in DAGON [60]. His thesis was that technology mapping for logic synthesis is closely related to the problem of code generation for software compilers, and hence the advanced techniques that have been developed for code generation should be applicable to technology mapping.

The problem of code generation in a compiler is to map a set of expressions onto a set of machine instructions for the target machine. Extensive research into compilers has led to efficient ways of formulating and solving this problem [2]. Each machine instruction is decomposed into a directed acyclic graph (DAG) of atomic operations, called a pattern. Each instruction has a cost associated with it which represents the relative cost, in execution time, of choosing that instruction. The sequence of high-level expressions is also represented by a DAG of atomic operations. The optimum code generation problem is equivalent to finding an optimum cost cover of the subject DAG by the pattern DAG's.

A similar approach is taken for the technology mapping problem. A set of base functions is chosen such as a two-input NAND-gate and an inverter. The logic equations are optimized in a technology-independent manner and then converted into a graph where each node is restricted to one of the base functions. This graph is called the "subject graph." The logic function for each library gate is also represented by a graph where each node is restricted to one of the base functions. Each graph for a library gate is called a "pattern graph." For any given logic function there are many different representations of the function using the base function set. Therefore each library gate is represented by many different pattern graphs.

The technology mapping problem is viewed as the optimization problem of finding a minimum cost covering of the subject graph by choosing from the collection of pattern graphs for all gates in the library. A "cover" is a collection of pattern graphs such that every node of the subject graph is contained in one (or more) of the pattern graphs. The cover is further constrained so that each input required by a pattern graph is actually an output of some other pattern graph. For area optimization, the cost of a cover is defined as the sum of the areas of the individual gates. For minimum delay optimization, the cost of a cover is defined as the critical path delay of the resulting circuit using an appropriate delay model. For the more typical problem of optimizing for minimum area under a given timing constraint, any cover which results in a circuit with critical path delay greater than that allowed for any output is considered an illegal cover; thus the minimum-area legal cover is the optimization goal. If there are no legal covers, the cover of minimum delay is considered the desired solution.

The critical parts of the procedure are the selection of the set of base functions and the optimization technique used to solve the covering problem.

B. Choice of Base Functions

The choice of a set of base functions is arbitrary as long as the base function set is functionally complete. However, this decision does influence the number of patterns needed to represent the gates in a library and the quality of the solution provided by DAG-covering. The goal is to find the base-function set which provides the highest level of optimization and produces a small set of patterns. In MIS [16], [38], a base-function set of a two-input NAND-gate and an inverter is used. This set can be proved [82] to be as good in terms of optimization potential as any other set containing two-input NOR-, AND-, OR-gates and inverters.

When both a NAND-gate and NOR-gate are used in the base-function set, the number of patterns required to represent some functions increases. For example, using both a two-input NAND-gate and a two-input NOR-gate, a large number of pattern graphs are required for all representations of the gate $f = \overline{ab + cd}$. Variations such as three NAND-gates (with inverters), three NOR-gates (with inverters), and other rep-

resentations using both NAND-gates and NOR-gates are possible patterns. Using only the two-input NAND-gate reduces the number of patterns to one.

The covering paradigm implies that each node of the subject graph is covered by a pattern but cannot be split and partially covered by two patterns. Therefore the granularity of the base function set affects the optimization potential. Thus a fine resolution base-function set allows for more covers, and hence better quality solutions. However, this has a price—more patterns are required to represent the logic function for some gates. In DAGON, two-input, three-input and four-input NAND-gates are used as the base-function set. With this approach, the logic function

$$f = \overline{abcd + efgh + ijkl + mnop}$$

requires only one pattern—a tree of five four-input NAND-gates. Representing all patterns for this same function using two-input NAND-gates and inverters requires 18 patterns. However, given the possibility for improved optimization, the finer resolution base function appears to be the better approach.

C. Creating the Subject Graph

A logic network has many representations as graphs of components from the base-function set, and each representation is a potential subject graph for DAG-covering. Each starting point leads to a graph cover of different cost. Even if the covering problem is solved exactly, every one of these starting points should be considered for an optimum solution.

Therefore heuristics are used to find a near-optimal form for the subject graph. As mentioned in the introduction, these optimizations include algebraic decomposition and Boolean simplification techniques using technology-independent cost functions. The number of nodes in the subject graph is used as a technology-independent estimate of the area of the circuit. The total number of literals in the sum-of-products form is effectively the same area estimator. The longest path from an input to an output in the subject graph is used to estimate the delay of the circuit.

The goal of technology-independent optimization should be to find a circuit representation which provides a good starting point for DAG-covering. The optimized equations are then transformed into two-input NAND-gate and inverter form in a straightforward manner.

This transformation is the same used for restructuring for timing [89]. In fact, the starting point provided by the timing restructuring algorithm is a good one for technology mapping [82]. However, it still remains an open problem to determine which of the possible subject graphs of two-input NAND's and inverters yields the optimum solution when an optimum covering algorithm is applied.

1) The DAG-Covering Problem: DAG-covering-by-DAG's is NP-hard even with only three pattern graphs (inverter, two-input NAND, two-input NOR) and if each subject graph node has no more than two incoming and outgoing edges [61].

An exact covering algorithm has been proposed in [82], based on a branch-and-bound procedure. However, the complexity of the algorithm is so large that only trivial problems could be solved. On the other hand, it is debatable whether this problem needs to be solved exactly since the subject graph is already the result of a heuristic mapping and hence does not reflect the most general optimization problem that needs to be solved. A more effective approach would be to develop a heuristic DAG-covering algorithm. However, this is still an open problem. (L. Lavagno at Berkeley has experimented with a number of heuristic with some degree of success, where XOR's and multiplexors are allowed in the gate library. This has been implemented in MIS2.1.)

An alternative approach to the DAG-covering problem is to simplify it so that the simplified problem could be solved effectively (for example, in linear time). Of course, the quality of the final solution will depend on the reduction of the search space.

Keutzer in DAGON [60] has proposed reducing the DAG-covering problem to a set of tree-covering-by-trees problems. His procedure is based on the following steps:

1) partition the subject graph into trees;
2) cover each tree optimally;
3) piece the tree-covers into a cover for the subject graph.

This approach has proved quite effective. In particular, it can be shown that if the cost function is additive, such as area, the tree-covering problem can be solved with a linear complexity algorithm based on dynamic programming. DAGON is a technology mapping program written by Keutzer on top of the tree manipulation tool "twig" [92]. Twig was originally developed to provide a flexible framework for building efficient algorithms for tree matching and for solving the tree-covering problem. Twig uses the Aho-Corasick [1] string-matching algorithm for matching and the Aho-Johnson [2] dynamic programming algorithm for optimum tree covering.

Its weak points are in the loss of global view due to the step of partitioning into trees. Covers across partition boundaries are not allowed. It will be interesting to see whether different partitioning algorithms can substantially improve the results obtained with this procedure.

The approach followed in MIS [16], [38] is patterned after DAGON. To improve the quality of the solution, additional covers are exposed by replacing any straight interconnection between gates with a pair of inverters. This augments the search space substantially at little cost.

D. Delay Optimization and Graph Covering

Synthesis for performance is increasingly important due to the competitive pressures for electronic systems with maximum performance. Thus a solution for technology mapping must consider timing in a direct way. If the delay were independent of the gate driven (i.e., a constant load model is used), then a dynamic-programming algorithm of linear complexity could be applied as well. Thus far, even though this model is not accurate, graph-covering-style technology mapping for delay has been carried out under this assumption. The results obtained are reasonable but by no means optimum. In fact, for a general delay cost function, the optimum cover depends on the forward part of the tree, and hence the dynamic-programming algorithm may not find the optimum result.

Keutzer and Vancura [62] use tree-height reduction to reduce the number of gates in a critical path and then do

drive buffering by placing a constraint on the amount of drive a gate must have to be used at a fan-out point of the network. Berman and Carter [7] investigate the problem of optimally powering up a node with a powering tree to minimize the required time at the source node. They prove that the problem is NP-complete and propose a heuristic algorithm which gives good results.

Rudell [82] has suggested a method to solve the minimum delay optimization problem for trees and the constrained-by-timing area optimization problem. His idea is based on a binning technique for the pin-loads as follows.

1) The unique set of pin-loads is determined and binning functions are constructed.
2) An array of solutions at each node of the subject tree is obtained, one per bin.
3) The arrival time for each cover for each load value is computed.
4) At each input of the cover, the optimum solution for driving the corresponding pin-load is selected.
5) The final cover is chosen based on the external load at the root of the tree.

The cover obtained by this technique is a minimum-delay cover. Note that this approach subsumes all technology mapping-related problems such as phase assignment and discrete sizing. It can also be generalized to solve the problem of technology mapping for optimum area cover with delay constraints.

The complexity of the algorithm is still linear, but it depends on the number of load-pins and arrival-time bins. For a reasonable library, we can have as many as 100 different pin-loads and 10 000 arrival bins (0.01 ns for 100 ns) yielding 1 000 000 solutions per node! Hence, to make this algorithm practical, Rudell devised an approximate technique that uses only a fixed number of bins. A clustering algorithm provides a good value for the bins so that the approximation due to the insufficient number of bins is minimized. A straightforward implementation of the algorithm runs only four times slower than the standard algorithm.

VIII. Rule-Based Methods

A popular technique for logic optimization uses "local transformations" or "rule-based systems." This includes the logic synthesis systems LSS [35], SOCRATES [46], and LORES [56]. LSS uses local transforms; where both technology-independent transformations and technology mappings are addressed uniformly [58]. Both LSS and SOCRATES use rule-based techniques as a part of a larger optimization system. For example, SOCRATES uses two-level minimization and algebraic decomposition as part of its optimization strategy, and LSS uses cube-factoring, global-flow, and other high-level transformations.

A rule-based system is a collection of rules and techniques for selecting when and where to apply a rule to improve the circuit quality. Each rule is expressed as a pair ("target graph," "replacement graph"). A rule is applied by identifying a portion of the circuit which contains a subgraph isomorphic to the target graph, and replacing the subgraph with the replacement graph. Each rule application preserves the circuit functionality. Technology mapping from Boolean equations starts with a straightforward translation of the equations into gates in the library (for example, using only AND-gates and OR-gates) or into a generic gate (e.g., NAND's). The circuit quality is improved through the iterative application of rules. Example rules from SOCRATES are shown in Fig. 14.

The operation of a rule-based system can be understood by viewing the optimization as a search on a "state-space graph." The state-space graph is a directed graph, where the nodes represent legal circuit configurations with the desired functionality, and a directed edge exists from node v_i to v_j if a rule application can transform the circuit of node v_i into the circuit of node v_j. Each node in this graph has an associated cost based on the area and delay of the corresponding circuit. The optimization starts from an arbitrary node in the state-space graph, and each application of a rule is a move to an adjacent node. The goal is to find the minimum cost node in the state-space graph.

The difficulty in applying rule-based techniques for optimization is in solving the problem of searching the state-

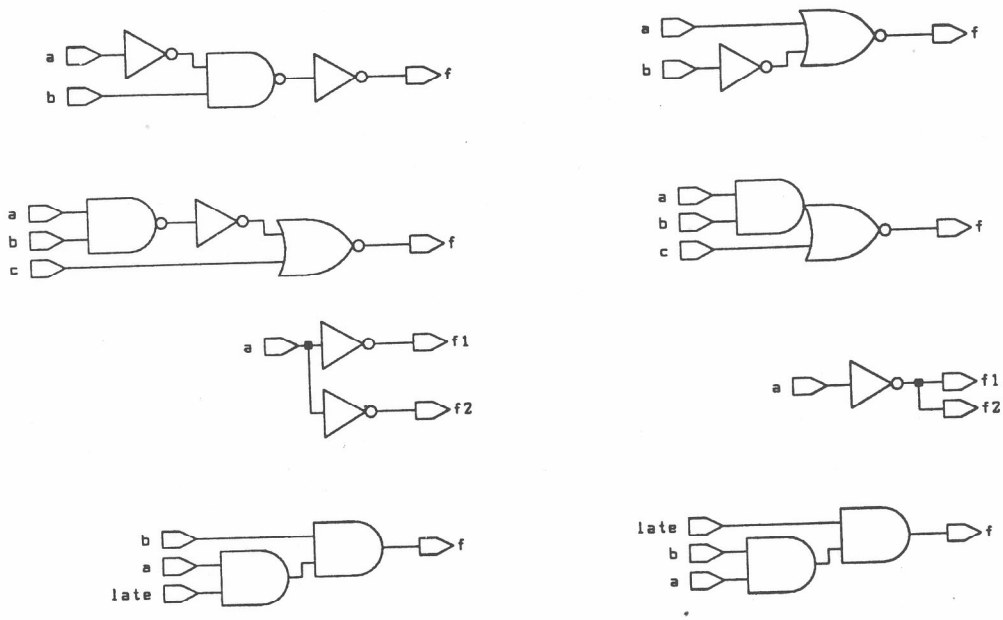


Fig. 14. Example rules from SOCRATES.

space graph for the minimum cost node. The approach reported in LSS and LORES is a greedy strategy. The edges from the current node are examined in a predetermined order, and the first edge which improves the value of the cost function is selected. The new node is taken as the current node, and the process is repeated until a local minimum is reached. A local-minimum node has no outgoing edge that improves the value of the cost function.

In SOCRATES, a search technique replaces the greedy strategy. The search starts from the current node and examines a fixed number of adjacent nodes (the "breadth" of the search). For each of these adjacent nodes, the search is repeated up to a fixed distance from the current node (the "depth" of the search). The best move seen in this set of nodes surrounding the current node defines a sequence of transformations. The next l transformations are taken as the next set of moves, and the search is continued. The parameter l controls the amount of look-ahead actually taken. A sequence of rule applications is accepted if the cost function at the end of the sequence is improved. This allows intermediate rule applications to temporarily increase the cost function. A state-space search is very expensive in terms of execution time unless the search parameters are controlled. In SOCRATES , a "meta-level rule-based expert system" [46] controls the breadth and depth of the search at each step and chooses the rules to apply. Results show a 12-percent improvement in area optimization for the partial state-space search over the greedy approach mentioned earlier. Typical values for the parameters are $l = 1$, depth $= 2$, and breadth $= 4$.

The primary advantages of a rule-based system are a) its flexibility in the types of rules and cost-functions that can be considered during the optimization, and b) the relative ease with which the core transformation system can be developed. The primary drawbacks of this approach are the difficulty in creating, maintaining, and modifying the rule-base, and the difficulty of incorporating new gates into a library. Also, unless artfully controlled, rule-based systems tend to be expensive in terms of execution time and offer an unpredictable quality of the result. These points are elaborated next.

The execution time of a rule-based system is determined by the number of nodes examined in the state-space graph. Examining each node requires a computation of the cost function, including the area and delay of the corresponding circuit. Even using incremental techniques, computation of the delay cost function is expensive (on the order of seconds for a 1-MIP computer on a large circuit [45]). This problem is aggravated by a bounded state-space search, where the number of cost function evaluations grows exponentially in the depth of the search. Therefore the cost of a rule-based approach is especially expensive if searching is used to provide high-quality results.

The quality of the circuits produced by a rule-based system depends on the completeness of the set of rules, and the quality of the heuristics which guide the walk in the search space graph. If the rules are not complete, the state-space graph may not be connected, leading to the impossibility of reaching particular solutions. Using a greedy search and a predetermined order to evaluate adjacent nodes has the problem of becoming stuck in a local optimum which is far from the global optimum. It is not clear if a limited state-space search adequately avoids suboptimal local minima.

Another problem with rule-based systems is that it is difficult to incorporate new gates into a library. A technique which provides technology portability is to define a master gate library, and to write all rules in terms of this library. It is then assumed that all libraries will be a subset of this master library. If a library has a gate which is not in the master library, it cannot be involved in the optimization process unless additional rules are added. This difficulty is compounded further by the interaction between the set of rules and the heuristics which control the rule application. For this reason, rule sets are handcrafted with a particular technology and design style in mind. Adding a new rule is not simply a matter of adding the rule to the set of transformations; it is also necessary to consider how the new rule will interact with the other rules in the system. In the limit, it may be necessary to rewrite the heuristics which control the order in which the rules are applied. For this reason, there is a significant effort to port the system to new libraries and technologies.

A final problem is the large number of transformations required to provide high-quality optimization. Writing, managing, and verifying all of these rules is a nontrivial process.

Despite these problems, local transformation techniques have demonstrated the ability to produce high-quality results. For example, LSS, SOCRATES, and LORES all report optimization results competitive with human designers [35], [46], [56]. An interesting recent development is the use of global flow in the LSS system. It is reported that this allows the elimination of eight out of nine rewrite rules aimed at reducing connections, and speeds up the program by one-third. Considering these discussions, we may consider this development as a merging of the algorithmic technology-independent methods with the local transformation methods.

IX. Conclusion

Multilevel logic synthesis is a powerful technique for the automatic generation of high-quality combinational circuits. Multilevel logic synthesis consists of a sequence of transformations on a multilevel logic network. These can be applied in an arbitrary sequence; however, the final result may depend heavily on the sequence chosen. The operations can be repeated to improve the results. We have discussed a number of different operations that can be performed on a multilevel logic network. We have attempted to survey as many techniques as possible, although more emphasis and detail have been given to those operations which, in our experience, have proven the most practical. It should be noted that such experience is biased toward the kind of decisions we made at the beginning of the design of the logic synthesis systems MIS and BOLD. Our approach has been heavily tilted toward the use of algorithms with proven properties, and thus we may not have represented fairly the rule-based approach. We will have to leave a thorough treatment of this approach to other authors and refer the reader to the existing literature [36], [35], [58], [5], [46].

Even in discussing various algorithmic approaches, there are many choices, such as spectral methods, transduction, Boolean minimization, and functional decomposition. Again, space considerations made it difficult to treat each of these methods uniformly, so we have chosen to provide more detail for those methods which we believe to be more

relevant for the design community and to relegate some of the other, perhaps promising, methods to a brief summary with references to papers fully describing the approach.

Multilevel logic synthesis, using all the techniques that have been described, can provide logic designs which are competitive or better than most manual designs, even though optimality results are still mostly lacking. We expect, as the research continues in this area, that the algorithms will continue to improve and more theoretical understanding will produce new algorithms, thereby leading to ever-improving capabilities.

Acknowledgment

This paper could not have been written without the help of Reily Jacoby, Rick McGeer, Chris Morrison, Rick Rudell, and Albert Wang, whose Ph.D. theses provided well-written summaries of many of the sections of this survey, as well as extensive bibliographies. We also thank the reviewers whose comments made the paper a bit more balanced in the treatment of some ideas and pointed out areas needing more clarification.

References

[1] A. Aho and M. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, pp. 333–340, June 1975.

[2] A. Aho and S. Johnson, "Optimal code generation for expression trees," *J. ACM*, pp. 488–501, July 1976.

[3] R. L. Ashenhurst, "The decomposition of switching functions," in *Proc. Int. Symp. Theory of Switching Functions*, Apr. 1959.

[4] K. Bartlett, D. Bostick, G. Hachtel, R. Jacoby, M. Lightner, P. Moceyunas, C. Morrison, and D. Ravenscroft, "Bold: A multiple-level logic optimization system," in *Proc. IEEE Int. Conf. on Computer Aided Design*, 1987.

[5] K. Bartlett, W. Cohen, A. de Geus, and G. Hachtel, "Synthesis and optimization of multilevel logic under timing constraints," *IEEE Trans. CAD IC*, vol. CAD-5, Oct. 1986.

[6] K. A. Bartlett, R. Brayton, G. Hachtel, R. Jacoby, C. Morrison, R. Rudell, A Sangiovanni-Vincentelli, and A. Wang, "Multi-level logic minimization using implicit don't cares," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.* vol. CAD-7, no. 6, pp. 723–740, June 1988.

[7] C. L. Berman and J. L. Carter, "The fanout problem: From theory to practice," in *Advanced Research in VLSI: Proceeding of the 1989 Decennial Caltech Conference*, C. L. Seitz, Ed. MIT Press, May 1989, pp. 69–99.

[8] L. Berman and L. Trevillyan, "A global approach to circuit size reduction," in *Advanced Research in VLSI, 5th MIT Conference*. MIT Press, 1988, pp. 203–214.

[9] D. Brand, "Redundancy and don't cares in logic synthesis," *IEEE Trans. Comput.*, vol. C-32, no. 10, pp. 947–952, Oct. 1983.

[10] D. Brand, "Logic synthesis," in *NATO ASI on Logic Synthesis and Silicon Compilation for VLSI*, P. Antognetti, G. DeMicheli, and A. Sangiovanni-Vincentelli, Eds. The Netherlands: Kluwer and Dordrecht, 1987.

[11] D. Brand, "PLA-based synthesis without PLA'S," in *Proc. Int. Workshop on Logic Synthesis*, May 1989.

[12] R. Brayton, "Factoring logic functions," *IBM J. Res. Develop.*, vol. 31, no. 2, pp. 187–198, Mar. 1987.

[13] R. Brayton, N. Brenner, C. Chen, G. Hachtel, C. McMullen, and R. Otten, "The Yorktown silicon compiler," in *Proc. Int. Symp. Circ. Syst. (ISCAS-85)*, pp. 391–394, June 1985.

[14] R. Brayton, R. Camposano, G. De Micheli, R. Otten, and J. van Eijndhoven, "The Yorktown silicon compiler," in *Silicon Complication*, D. Gajski, Ed. Addison-Wesley, 1988.

[15] R. Brayton and C. McMullen, "Synthesis and optimization of multistage logic," in *Proc. Int. Conf. Comp. Des. (ICCD-84)*, pp. 23–28, 1984.

[16] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: Multiple-level interactive logic optimization system," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. CAD-6, no. 6, pp. 1062–1081, Nov. 1987.

[17] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Multi-level logic optimization and the rectangular covering problem," in *Proc. IEEE Int. Conf. on CAD (ICCAD)*, Nov. 1987.

[18] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Multi-level logic synthesis," notes for lectures at Oxford/Berkeley Summer Engineering Programme, July 1989.

[19] R. Brayton, E. Sentovich, and F. Somenzi, "Don't-cares and global flow analysis of Boolean networks," in *Proc. Int. Conf. CAD (ICCAD-88)*, Nov. 1988, pp. 98–101.

[20] R. Brayton and F. Somenzi, "Boolean relations and incomplete specification of logic networks," in *Int. Conference on Very Large Scale Integration*, Aug. 1989.

[21] R. Brayton and F. Somenzi, "Minimization of Boolean relations," in *Proc. Int. Symp. Circ. Syst. (ISCAS-89)*, May 1989.

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

[23] R. K. Brayton and Curt McMullen, "The decomposition and factorization of Boolean expressions," in *Proc. Int. Symp. Circ. Syst. (ISCAS-82)*, Rome, May 1982.

[24] R. K. Brayton, "Algorithms for multilevel logic synthesis and optimization," in *NATO ASI on Logic Synthesis and Silicon Compilation for VLSI*, G. DeMicheli *et al.*, Eds. Dordrecht, The Netherlands: Kluwer, 1987.

[25] M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*. Woodland Hills, CA: Computer Science Press, 1976.

[26] F. Brglez, "Testability characterization of embedded modules via logic minimization," in *Proc. Int. Workshop on Logic Synthesis*, Research Triangle Park, May 1987.

[27] F. Brglez, D. Bryan, J. Calhoun, G. Kedem, and R. Lisanke, "Automated synthesis for testability," *IEEE Trans. Ind. Electron.* vol. IE-56, May 1989.

[28] D. Bryan, F. Brglez, and R. Lisanke, "Redundancy identification and removal," in *Proc. Int. Workshop on Logic Synthesis*, Research Triangle Park, May 1989.

[29] R. E. Bryant, "Symbolic manipulation of boolean functions using a graphical representation," in *Proc. 22nd Design Automation Conf.*, July 1985.

[30] J. Calhoun and F. Brglez, "A framework for hierarchical test generation: Version 1.0," technical report 89-06, Microelectronics Center of North Carolina, P.O. Box 12889, Research Triangle Park, NC 27709, Jan. 1989.

[31] J. Calhoun, D. Bryan, and F. Brglez, "Automatic test pattern generation (ATPG) for scan-based digital logic: Version 1.0," technical report 17, Microelectronics Center of North Carolina, P.O. Box 12889, Research Triangle Park, NC 27709, Dec. 1987.

[32] K-T. Cheng and V. D. Agrawal, "State assignment for initializable synthesis," in *Proc. Int. Conf. Computer Aided Design (ICCAD-89)*, Nov. 1989.

[33] H. Cho, G. Hachtel, R. Jacoby, and P. Moceyunas, "Test pattern generation in logic optimization," in *Proc. Int. Conf. Computer Aided Design (ICCAD-89)*, Nov. 1989.

[34] H. Curtis, "Generalized tree circuit—The basic building block of an extended decomposition theory," *J. ACM*, vol. 8, pp. 562–581, 1961.

[35] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L. Trevillyan, "LSS: A system for production logic synthesis," *IBM J. Res. Develop.*, vol. 28, no. 5, pp. 537–545, Sept. 1984.

[36] J. Darringer, W. Joyner, L. Berman, and L. Trevillyan, "Logic synthesis through local transformations," *IBM J. Res. Develop.*, vol. 25, no. 4, pp. 272–280, July 1981.

[37] Edwards S. Davidson, "An algorithm for NAND decomposition under network constraints," *IEEE Trans. Computers*, vol. C-18, Dec. 1969.

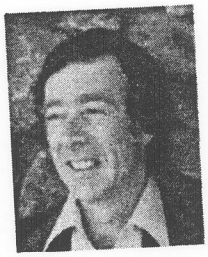
[38] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Technology mapping in MIS," in *Proc. Int. Conf. CAD (ICCAD-87)*, Nov. 1987, pp. 116–119.

[39] S. Devadas and K. Keutzer, "Boolean minimization and algebraic factorization procedures for fully testable sequential machines," in *Proc. Int. Conf. Computer Aided Design (ICCAD-89)*, Nov. 1989.

[40] D. Dietmeyer and Y. Su, "Logic design automation of fan-in

limited nand networks," *IEEE Trans. Comp.*, vol. C-18, no. 1, pp. 11–22, Jan. 1969.
[41] D. M. Du, "Variable transformation—A new approach to synthesizing combinational switching circuits," in *Proc. Workshop on Microelectronic and Information Syst.*, Dec. 1986, pp. 441–461.
[42] H. Fujiwara, *Logic Testing and Design for Testability*, M.I.T. Press Series in Computer Systems. Cambridge, MA: The M.I.T. Press, 1985.
[43] H. Fujiwara and T. Shimono, "On the acceleration of test generation algorithms," *IEEE Trans. Computers*, pp. 1137–1144, Dec. 1983.
[44] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Trans. Computers*, vol. C-30, no. 3, pp. 215–222, 1981.
[45] D. Gregory, personal communication, Apr. 1988.
[46] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel, "Socrates: A system for automatically synthesizing and optimizing combinational logic," in *Proc. 23th Design Automation Conf.*, June 1986, pp. 79–85.
[47] G. Hachtel, R. Jacoby, K. Keutzer, and C. Morrison, "On the relationship between area optimization and multifault testability of multilevel logic," in *Proc. Int. Conf. Computer Aided Design (ICCAD-89)*, Nov. 1989.
[48] G. Hachtel, R. Jacoby, and P. Moceyunas, "On computing and approximating the observability don't care set," in *Proc. Int. Workshop on Logic Synthesis*, Research Triangle Park, May 1989.
[49] G. Hachtel, M. Lightner, R. Jacoby, C. Morrison, and P. Moceyunas, "A tutorial on combinational logic synthesis," in *Proc. Int. Conf. on Computer Aided Design (ICCAD-88)*, 1988.
[50] G. Hachtel, M. Lightner, R. Jacoby, C. Morrison, P. Moceyunas, and D. Bostick, "Bold: The boulder optimal logic design system," in *Hawaii Int. Symp. on Systems Sciences*, 1988.
[51] G. D. Hachtel and R. M. Jacoby, "Algorithms for multi-level tautology and equivalence," in *Proc. IEEE Int. Symp. on Circuits and Systems*, June 1985.
[52] G. D. Hachtel, R. M. Jacoby, and C. R. Morrison, "Techmap: Technology mapping with area and delay optimization," in *Proc. Int. Workshop on Logic and Architecture Synthesis for Silicon Compilers*, Grenoble, France, May 1988.
[53] M. Helliwell and M. Perkowski, "A fast algorithm to minimize multi-output mixed-polarity generalized Reed-Muller forms, in *Design Automation Conf.*, June 1988, pp. 427–432.
[54] S. L. Hurst, D. M. Miller, and J. C. Muzio, *Spectral Techniques in Digital Logic*. New York: Academic Press, 1985.
[55] T. Ibaraki and S. Murago, "Synthesis of networks with a minimum number of negative gates," *IEEE Trans. Computers*, vol. C-20, pp. 49–58, Jan. 1971.
[56] J. Ishikawa, H. Sato, M. Hiramine, K. Ishida, S. Oguri, Y. Kazuma, and S. Murai, "A rule-based reorganization system Lores/EX," in *Proc. Int. Conf. Comp. Des. (ICCD-88)*, Oct. 1988, pp. 262–266.
[57] R. Jacoby, "On the comparison of Boolean functions," Ph.D. thesis, University of Colorado, May 1989.
[58] W. Joyner, L. Trevillyan, D. Brand, T. Nix, and S. Gundersen, "Technology adaptation in logic synthesis," in *Proc. 23th Design Automation Conf.*, June 1986, pp. 94–100.
[59] K. Karplus, "Using if-then-else DAG's for multi-level minimization," in *Decennial Caltech Conference on VLSI*, May 1989.
[60] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," in *Proc. 24th Design Automation Conf.*, June 1987, pp. 341–347.
[61] K. Keutzer, personal communication, Feb. 1989.
[62] K. Keutzer and M. Vancura, "Timing optimization in a logic synthesis system," in *Proc. Int. Workshop on Logic and Architectural Synthesis*, May 1987.
[63] S. Kundu and S. M. Reddy, "On the design of robust multiple fault testable cmos combinational logic circuits," in *Proc. IEEE Int. Conf. Computer Aided Design*, Nov. 1988, pp. 240–243.
[64] Eugene Lawler, *Combinatorial Optimization*. Holt Rinehart Winston, 1976.
[65] Eugene L. Lawler, "An approach to multilevel boolean minimization, *J. Assoc. Comput. Machinery*, vol. 11, July 1964.
[66] B. Lin and A. R. Newton, "Restructuring state machines and state assignment: Relationship to minimizing logic across latch boundaries," in *Int. Workshop on Logic Synthesis*, May 1989.
[67] A. Malik, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic minimization for factored forms," in *Int. Workshop on Logic Synthesis*, May 1989.
[68] Abdul A. Malik, Robert K. Brayton, A. Richard Newton, and Alberto L. Sangiovanni-Vincentelli, "A modified approach to two-level logic minimization," in *Proc. Int. Conf. on CAD (ICCAD-88)*, Nov. 1988.
[69] S. Malik, R. Brayton, and A. Sangiovanni-Vincentelli, "Encoding symbolic inputs for multi-level logic implementation," in *Int. Conf. Very Large Scale Integration*, Aug. 1989.
[70] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic verification using binary decision diagrams in a logic synthesis environment," in *Proc. Int. Conf. Computer Aided Design (ICCAD-88)*, 1988, pp. 6–9.
[71] Y. Matsunaga and M. Fujita, "Multi-level logic optimization using binary decision diagrams," in *IEEE Int. Conf. on Computer-Aided Design*, Nov. 1989.
[72] E. J. McCluskey, "Minimization of boolean functions," *Bell Lab. Tech. J.*, vol. 35, Nov. 1956.
[73] P. McGeer, "On the interaction of functional and timing behavior in combinational circuits," Ph.D. thesis, University of California, Berkeley, 1989.
[74] P. McGeer and R. Brayton, "Efficient, stable algebraic operations on logic expressions," in *Proc. Int. Conf. on Very Large Scale Integration*, Aug. 1987.
[75] P. C. McGeer and R. K. Brayton, "Consistency and observability invariance in multilevel logic synthesis," in *Int. Conf. on Computer-Aided Design (ICCAD-89)*, 1989.
[76] C. Moraga, "Comments on a method of Karpovsky," *Inform. Control*, vol. 35, no. 3, pp. 243–246, 1977.
[77] C. R. Morrison, "Multilevel logic minimization," Ph.D. thesis, University of Colorado, Aug. 1989.
[78] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, "The transduction method—design of logic networks based on permissible functions," *IEEE Trans. Computers*, 1989.
[79] S. M. Reddy, "Easily testable realization for logic functions," *IEEE Trans. Computers*, vol. C-21, pp. 1183–1188, Nov. 1972.
[80] P. J. Roth and R. M. Karp, "Minimization over Boolean graphs," *IBM J. Res. Develop.*, vol. 6, Apr. 1962.
[81] K. Roy, K. De, J. A. Abraham, and S. Lusky, "Synthesis of delay fault testable combinational logic," in *Proc. Int. Conf. Computer Aided Design (ICCAD-89)*, Nov. 1989.
[82] R. Rudell, "Logic synthesis for VLSI design," Ph.D. thesis, University of California, Berkeley, 1989.
[83] R. Rudell and A. Sangiovanni-Vincentelli, "Exact minimization of multiple-valued functions for pla optimization," in *Proc. Int. Conf. on CAD (ICCAD-86)*, pp. 352–355, 1986.
[84] A. Saldanha, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Multilevel logic simplification using don't cares and filters," in *Proc. IEEE Design Automation Conf.*, June 1989.
[85] H. Savoj, A. A. Malik, and R. Brayton, "Fast two-level minimizers for multilevel logic synthesis," in *Proc. Int. Conf. on Computer Aided Design (ICCAD-89)*, Nov. 1989.
[86] M. Schulz and E. Auth, "Advanced automatic test pattern generation and redundancy identification techniques," in *Proc. 18th Int. Symp. on Fault-Tolerant-computing*, 1988.
[87] M. Schulz, E. Trischler, and T. Sarfert, "Socrates: A highly efficient ATPG system," *IEEE Trans. Comput. Aided Design Integrat. Circuits Syst.*, vol. CAD-7, no. 1, pp. 126–137, Jan. 1988.
[88] R. Segal, "BDSYN: Logic description translator; BDSIM: Switch-level simulator," master's thesis, University of California, Berkeley, May 1987; memorandum UCB/ERL M87/33.
[89] K. Singh, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Timing optimization of combinational logic," in *Proc. Int. Conf. CAD (ICCAD-88)*, Nov. 1988, pp. 282–285.
[90] F. Somenzi and R. K. Brayton, "An exact minimizer for boolean relations," in *Int. Conf. Computer-Aided Design (ICCAD-89)*, Nov. 1989.
[91] A. Srinivasan, "Muroga's transduction methods and multivalued decision diagrams," EE290LS Class Project, EECS Dept., University of California at Berkeley, May 1989.

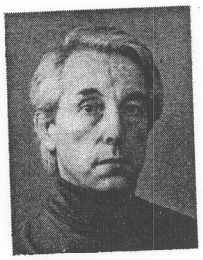
[92] S. Tjiang, "Twig reference manual," technical report, AT&T Bell Laboratories, 1985.
[93] L. Trevillyan and L. Berman, "A global flow approach to circuit optimization," in *Proc. Fifth MIT Conf. on VLSI*, Mar. 1988, pp. 203–214.
[94] L. Trevillyan and L. Berman, "Improved logic optimization using global flow analysis," in *Proc. Int. Conf. Computer Aided Design (ICCAD-88)*, pp. 102–105, Nov. 1988.
[95] L. Trevillyan, W. Joyner, and L. Berman, "Global flow analysis in automated logic design," *IEEE Trans. Computers*, vol. C-35, no. 2, pp. 77–81, Jan. 1986.
[96] D. Varma and E. A. Trachtenberg, "Design automation tools for efficient implementation of logic by decomposition," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, 1989.
[97] A. Wang, "Algorithms for multi-level logic optimization," Ph.D. thesis, University of California, Berkeley, 1989.
[98] C-L. Wey, S-M Chang, and J-Y Jo, "An efficient output phase assignment for multilevel logic minimization," in *Int. Workshop on Logic Synthesis*, May 1989.



Robert K. Brayton (Fellow, IEEE) received the B.S.E.E. degree from Iowa State University in 1956 and the Ph.D. degree in mathematics from MIT in 1961.

From 1961 to 1987 he was a member of the Mathematical Sciences Department of the IBM Thomas J. Watson Research Center at Yorktown Heights, NY. From 1970 to 1972, he was Assistant Director of the department, and in recent years he was a second level Manager for mathematical algorithms. He was a Visiting Professor at MIT in 1965–1966; at the Imperial College, London, in 1975–1976, and at the University of California, Berkeley, in 1985–1986. In 1987 he joined the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley, where he is presently a Professor. His interests and contributions have been in the areas of nonlinear networks, stability theory, numerical methods for differential equations, sparse matrices, simulation of electrical circuits, optimization methods for circuit design, logic synthesis and minimization, silicon compilers, and general CAD/VLSI issues.

Dr. Brayton has authored or coauthored more than 80 technical papers and is the coauthor of two books: *Logic Minimization Algorithms for VLSI Synthesis* (with G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli) and *Computer Aided Design: Sensitivity and Optimization* (with R. Spence). He has served on various IEEE committees, including the Circuits and Systems ADCOM, CANDE, and Large Scale Systems. He is a fellow of the AAAS. He shared (with G. D. Hachtel and F. G. Gustavson) the Best Paper Award of the IEEE Circuits and Systems Society in 1971 for "The Sparse Tableau Approach to Network Analysis and Design." In 1988, he received the Darlington Award of the IEEE Circuits and Systems Society. In addition, he has received two IBM Outstanding Innovation Awards and two IBM patent awards.



Gary D. Hachtel (Fellow, IEEE) received the B.S. degree from the California Institute of Technology in 1959 and the Ph. D. degree from the University of California, Berkeley, in 1965, all in electrical engineering.

He has taught at U.C. Berkeley, at New York University, at U.C.L.A., where he was Regents Lecturer in 1974, and at the University of Denver. From 1965 to 1981 he was with IBM at the Thomas J. Watson Research Center at Yorktown Heights, NY, where he was Manager of Modeling and Systems Design in the Mathematical Sciences Department. Since 1981 he has been Professor of Electrical and Computer Engineering at the University of Colorado, Boulder. His current research is on algorithms for computer-aided design, including logic synthesis, simulation, testing, layout, sparse matrices, and optimization.

Dr. Hachtel was Associate Editor for the International Journal for Numerical Methods in Engineering and for the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS. He is now Associate Editor for the International Journal for Mathematics and Computation in Electronics and Electrical Engineering. He received an IBM Outstanding Contribution Award for integrated circuit modeling in 1968 and an IBM Outstanding Invention Award for the tableau approach to network design. In 1971 he was co-recipient of the best paper award from the Circuits and Systems Society, and in 1979 he received the W.R.G. Baker award for the best IEEE PROCEEDING or TRANSACTIONS article to appear in calendar year 1978. In 1981 he was a distinguished lecturer of the Circuits and Systems Society.

Alberto Sangiovanni-Vincentelli (Fellow, IEEE) received the Dr. Eng. degree (summa cum laude) from the Politecnico di Milano, Italy, in 1971.

From 1971 to 1977, he was with the Istituto di Elettrotecnica ed Elettronica, Politecnico di Milano, Italy, where he held the positions of Research Associate, Assistant, and Associate Professor. In 1976, he joined the Department of Electrical Engineering and Computer Sciences of the University of California at Berkeley, where he is presently Professor. He is a Consultant in the area of computer-aided design to several industries. His research interests are in various aspects of computer-aided design of integrated circuits. He was Associate Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS, and is Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and a member of the Large-Scale Systems Committee of the IEEE Circuits and Systems Society and of the Computer-Aided Network Design (CANDE) Committee. He was Executive Vice-President of the IEEE Circuits and Systems Society in 1983.

In 1981, Dr. Sangiovanni-Vincentelli received the Distinguished Teaching Award of the University of California. At the 1982 IEEE–ACM Design Automation Conference, he was given a Best Paper and a Best Presentation Award. In 1983, he received the Guillemin-Cauer Award for the best paper published in the IEEE TRANSACTIONS ON CAS and CAD in 1981–1982. At the 1983 Design Automation Conference, he received a Best Paper Award. In 1988, he received the Darlington Award of the IEEE Circuits and Systems Society. He is a member of ACM and Eta Kappa Nu.