DESIGN VERIFICATION AND FAULT DIAGNOSIS IN MANUFACTURING

When designing and building digital systems, we must ensure that the manufactured final product is exactly what was intended. As shown in Fig. 1, there are two processes in creating digital systems: design process and manufacturing process. Corresponding to these two processes, there are two key issues for ensuring digital systems behave as originally intended. The first is to make sure that what we are designing is correct, that is, the design is exactly the same as what we intend. The second is to make sure that what we are manufacturing is correct, that is, the product is exactly the same as what we have designed. The former process is called *design verification* and the latter is called *manufacturing test and diagnosis*. In this article we will give an overview of design verification and manufacturing fault diagnosis technology.

DESIGN VERIFICATION

As mentioned before, design verification is the process to ensure that what we are designing is exactly what is intended.

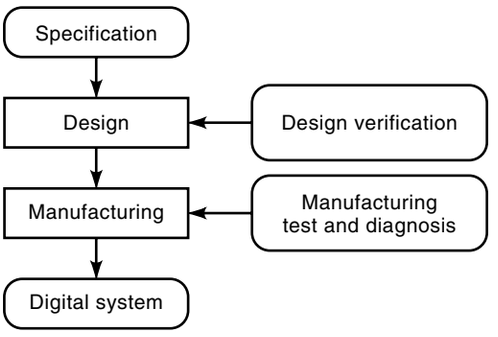

Figure 1. Creating digital systems.

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*. Copyright © 1999 John Wiley & Sons, Inc.

This is one of the most important and sometimes the most time-consuming process in designing complicated systems. The specifications describe what we want, and verification is the process for checking whether the designs satisfy their specifications. The first step for verification is to describe both specification and design in mathematical ways so that we can formally apply logic to them. In the case of digital systems, Boolean functions and mathematical logics such as first-order predicate calculus are typically used, since behaviors of digital systems can be directly described by these types of logic. Once we have mathematical descriptions for specifications and designs, the next step is to verify that designs satisfy their specification via reasoning.

Since verification ensures the correctness of designs with respect to specification, it is done by simulating designs and checking the appropriateness of outputs from simulations. However, this approach cannot be complete until we simulate all possible cases, which is impossible for large circuits with many input signals (i.e., all possible values of n inputs or 2^n combinations). Formal verification is a process that tries to prove the correctness of designs mathematically. It implicitly checks all possible cases and guarantees the correctness of designs for all possible input combinations. Let us clarify the difference between formal verification and simulation with an example.

Figure 2 is an example combinational circuit. It uses a gate called the NAND gate. NAND(x, y) gives a complement of the conjunction of x and y. It generates a 0 at its output only when both inputs are 1. Otherwise, it generates 1. The specification for the circuit is the EXCLUSIVE-OR function of x and y, which must be realized at the output of the circuit. Here EXCLUSIVE-OR is a logic function that gives the value 1 if and only if the two input values are different; otherwise it gives 0. The EXCLUSIVE-OR function of x and y is defined as $x \cdot \bar{y} + \bar{x} \cdot y$. Formal verification is done to make sure that the circuit in Fig. 2 realizes the EXCLUSIVE-OR function at the output.

We can simulate the circuit and test for its correctness. Verification by simulation is sometimes called *validation,* since it does not guarantee the correctness of the design completely unless we can simulate all possible cases, which is mostly impossible for large circuits. What we can do is to test some but not all, cases. Since digital systems are described in mathematical logic or its extensions, their behaviors can be simulated by repeatedly computing logic functions. By simulating the functions of NAND in the circuit, we can obtain the values for the output of the circuit. We need to check all four cases of possible input combinations for two variables.

On the other hand, formal verification of the circuit in Fig. 2 is to prove that its output is mathematically equivalent to the logic function EXCLUSIVE-OR of x and y. This can be checked by manipulating the Boolean formulas generated from the circuit in the following way:

$z = \overline{b \cdot c}$, $b = \overline{x \cdot a}$, $c = \overline{a \cdot y}$, $a = \overline{x \cdot y}$	Definitions from the circuit
$z = \overline{x \cdot \overline{x \cdot y} \cdot \overline{x \cdot y} \cdot y}$	Substitution
$z = x \cdot \overline{x \cdot y} + \overline{x \cdot y} \cdot y$	DeMorgan's law
$z = x \cdot (\bar{x} + \bar{y}) + (\bar{x} + \bar{y}) \cdot y$	DeMorgan's law
$z = x \cdot \bar{y} + \bar{x} \cdot y$	Simplification

The last formula is the definition of the EXCLUSIVE-OR function of x and y. Since manipulation of all preceding formulas is independent of the values of x and y, the circuit is formally verified to be equivalent to the EXCLUSIVE-OR function of x and y.

Formal verification of logic circuits using transformations of logic formulas like those just given is sometimes called *theorem-proving-based verification,* since it is trying to prove mathematically the correctness of designs by manipulating logic formulas. As can be seen from the previous example, an appropriate ordering of the application of various transformations, such as substitution, De Morgan's law, and simplification, must be identified in order to obtain the goal formulas (i.e., formulas in the specification). Moreover, if the designs are not correct, transformations do not work and the verification process may not terminate. Therefore appropriate user guidance is essential, and so the verification process is interactive, that is, each transformation of the formulas is guided by users who are verifying the designs. There is a significant amount of research on the use of theorem-proving methods for formal verification (1). Although there has been much success, this method is not yet widely used because it is not completely automatic and needs human interaction.

Automatic verification techniques perform an exhaustive case analysis for all combinations of values of variables, similar to the simulation of all possible cases. Typically, the techniques are based on case analysis. They first analyze the case for which the first chosen variable in the formula is 0 and then check the case for which that variable is 1, and so on. This case pattern may have to continue for all variables in the formula. Fortunately, in most cases, we can reach special cases where we can decide the value of the formula immediately. For example, suppose we analyze the formula, $x_1 \cdot x_2 \cdot x_3$. When the variable x_1 is set to 0, the entire formula immediately becomes 0 regardless of the values of the other variables. Further analysis is unnecessary for this case. Although this case analysis technique performs much better than exhaustive simulation, it is still very time consuming as its execution time grows exponentially in principle. Because of this, the case analysis technique cannot be applied to large circuits. Situations have, however, changed completely since a new data representation method for logic functions in computers, called binary decision diagrams (BDDs) (2–4), and its efficient manipulation algorithms were proposed in the 1980s. By using BDDs, significantly larger circuits can be verified in much less time.

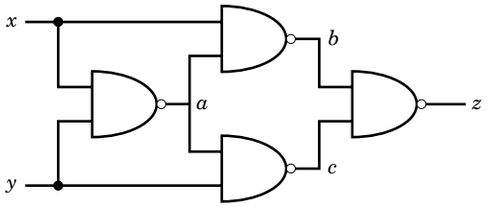

Figure 2. An example circuit that realizes an EXCLUSIVE-OR function.

Binary Decision Diagram

The binary decision diagram was proposed in the late 1980s and since then it has been widely used for various problems in

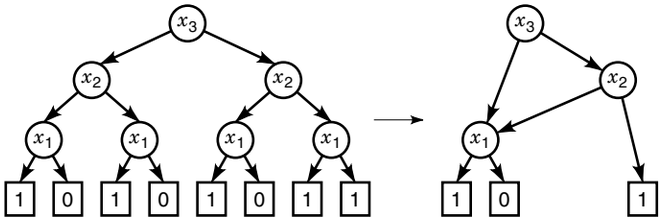

Figure 3. Decision tree and its corresponding binary decision diagram.

computer science, especially in computer-aided-design areas. Here we briefly introduce BDD.

BDDs are derived from binary decision trees. An example of a binary decision tree is shown in the left side of Fig. 3. It is basically an all-case analysis of the given logic function based on the values of variables. x_1, x_2, x_3 are variables and 0 and 1 are constants. Each left edge indicates that the value of that variable is 0, whereas each right edge indicates that the value is 1 (unless constant values are added to edges as attributes). We first fix the ordering of variables. In this case the ordering is x_3, x_2, x_1. On all paths from the root node to the leaves, all variables must appear only in this order. By traversing the edges from the root node, we can determine the value of the function. For example, the value of the function for $x_1 = x_2 = x_3 = 0$ is 1 whereas the value for $x_1 = x_2 = 0$, $x_3 = 1$ is 0. Please note that the sizes of binary decision trees are exponential with respect to the numbers of variables. BDD is derived from this tree by removing redundant nodes, as can be seen from the right side of the figure.

Figure 4 shows ways to generate BDD from the binary decision tree. First, isomorphic subgraphs are merged as can be seen from the first transformation in the figure. For example, the left three nodes for x_3 are isomorphic and are merged. Then any nodes with two edges going to the same nodes are deleted, as can be seen from the second transformation of the figure. If the two edges go to the same nodes, the function does not depend on the value of that variable for that particular case, and hence those nodes can be deleted. After these steps, binary decision trees become binary decision graphs, since there is sharing of subgraphs. As can be seen from Fig. 4, BDD is a lot smaller than the binary decision tree in general. An important fact is that sizes of BDDs can be polynomial for many useful logic functions, such as adders, parity functions, and most control circuits. Another key issue is that BDD is a canonical representation for logic functions with respect to the predetermined orderings of variables. That is, if the two logic functions are equivalent, their corresponding BDDs will be isomorphic as long as they are using the same ordering of variables. This is an important fact when we apply

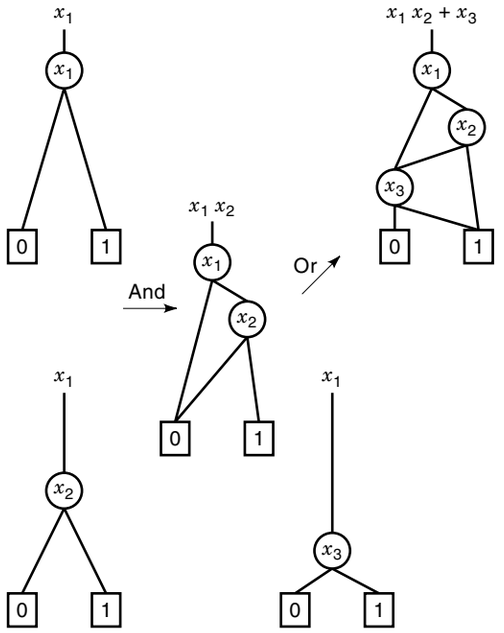

Figure 5. Using "apply" to manipulate logic operatons on BDDs.

BDD to verification problems. Because of these advantages BDD is now widely used.

Although BDD can be obtained from binary decision trees as shown in Fig. 4, this is not an efficient way to generate BDDs, since sizes of binary decision trees are exponential with respect to the numbers of variables. So we need more efficient ways to generate BDD directly from logic circuit representation. This can be done by the procedure "apply" that computes logic operations directly on BDDs. Examples of apply processes are shown in Fig. 5. The apply procedure basically traverses the two given BDDs from the roots to the leaves in a depth-first order. For each step in the depth-first traversal of the two BDDs, it applies logic operations, such as AND and OR, on the two current nodes and generates a new node that corresponds to the results of logic operations. The amount of time for completion of this procedure is proportional to the product of the sizes of the BDDs that it traverses, and hence it is very efficient as long as the BDD sizes can be kept small. By using the apply procedure, we can generate BDD directly from logic circuits and do not have to generate binary decision trees.

Although BDD is a very efficient and also effective way to manipulate logic functions, it surely has several drawbacks. One of the most important is the fact that sizes of BDDs are very sensitive to ordering of variables. Figure 6 shows an ex-

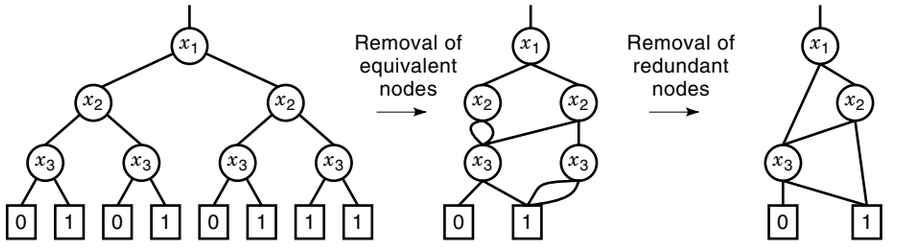

Figure 4. BDD is a canonical representation for logic functions.

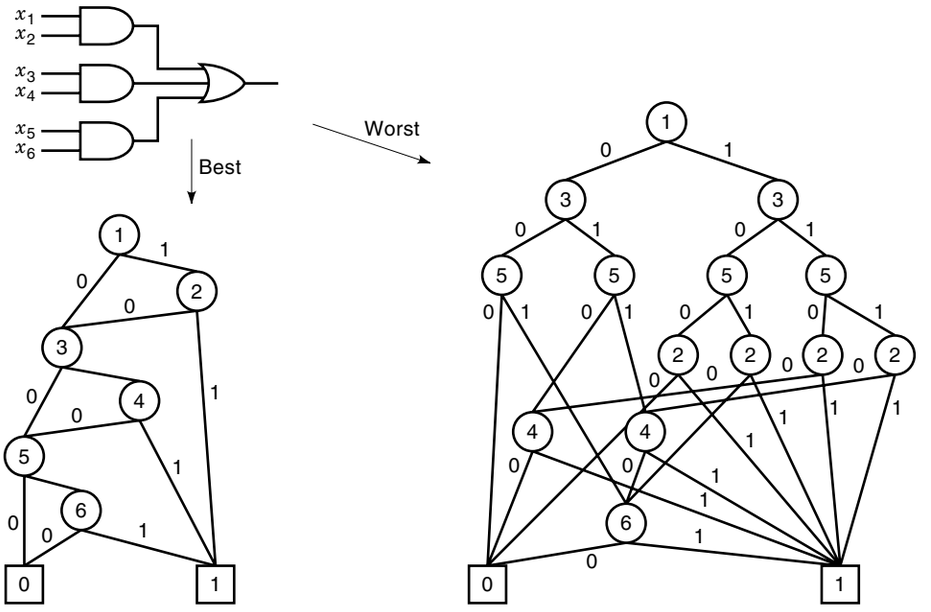

Figure 6. Ordering of variables is important for BDD.

treme case. The two BDDs represent the same logic function that corresponds to the output of the circuit diagram in the figure. The left BDD uses the best ordering, x_1, x_2, x_3, x_4, x_5, x_6, whereas the right BDD uses the worst ordering, x_1, x_3, x_5, x_2, x_4, x_6. So, if we use bad ordering of variables, the resulting BDDs can be too large to be manipulated. Variable ordering for BDD is one of the most important problems in BDD-related research. It is known that to find the best ordering is NP-complete; so we have to use heuristic approaches for large logic functions (5). There are several good heuristics for giving good ordering (6–11). These heuristics are generally good for practical use, but sometimes BDDs cannot be built simply because of poor ordering. In that sense, the variable ordering problem for BDD is still a good research topic.

Because BDDs are so widely used, several BDD packages are available in the public domain (12). They include a complete set of useful routines for BDDs, and users can manipulate logic functions in BDDs by just using those routines appropriately.

Practical Verification Technique For Combinational Circuits

In order to compare the equivalence among combinational circuits, it is sufficient to generate BDDs from the circuits and to check if they are isomorphic, since BDD is a canonical representation for logic functions once ordering of variables is fixed.

So, given a circuit, first of all, ordering of variables is determined by using appropriate heuristics. Then we generate BDDs for each gate in the circuit individually using the apply procedure as shown in Fig. 7. After this process, we get the BDD for the output of the circuit.

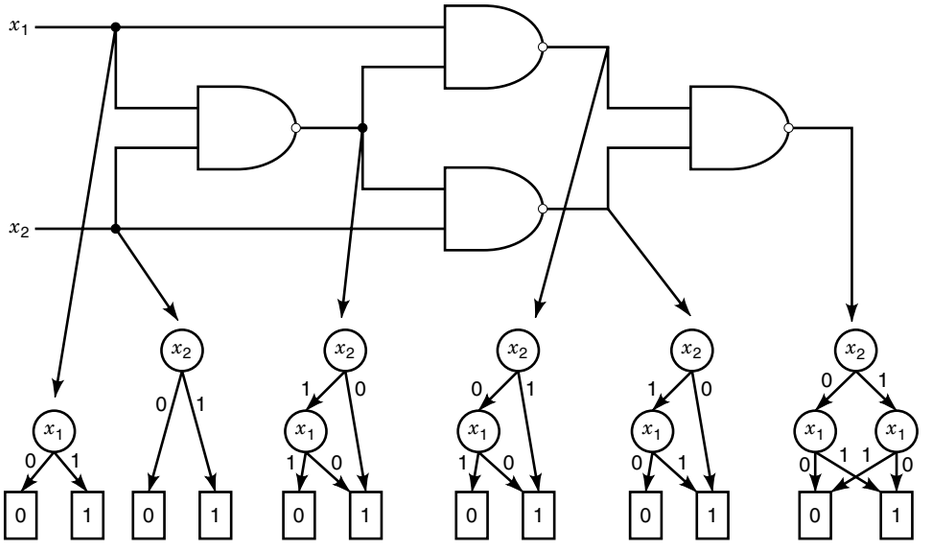

Figure 7. Creating BDDs from circuits.

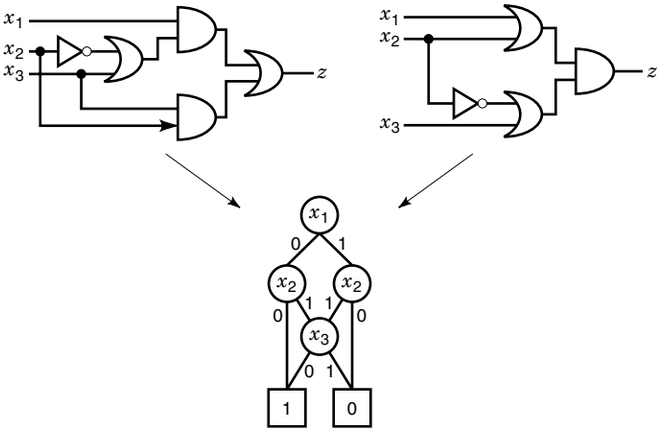

Figure 8. Verification based on BDD.

We repeat this process on the other circuit to be compared and then check if the two BDDs obtained are isomorphic (13). An example verification based on this approach is shown in Fig. 8. In this case, both circuits give the same isomorphic BDD and so they are logically equivalent. In this approach the most important part is how to obtain ordering of the variables of BDDs, since it will determine whether we can verify circuits. If we can have a good ordering, the BDD size can be relatively small and we may be able to finish BDD construction. But if we use a bad ordering, the BDD construction process may not finish because of the prohibitively large size. By using a good heuristic for variable ordering, the state-of-the-art verifier based on this approach can verify circuits having up to a couple of thousands of gates.

How can we proceed if the circuits to be verified are much larger than a couple of thousands of gates? One way is to construct a "miter" as shown in Fig. 9 (14,15). The two circuits to be compared are connected by an EXCLUSIVE-OR gate. Then if the two circuits are equivalent, the output of the EXCLUSIVE-OR gate is always 0. So, we have only to build BDD for the output of the EXCLUSIVE-OR gate and check if it is a constant 0 or not. In so doing, we do not necessarily build a BDD for each circuit. Instead, we can construct a BDD for the output of the EXCLUSIVE-OR gate by traversing the circuit from output to input. Hence, even if the BDDs for the original two circuits are large, the BDD that we construct may not become large. Although this is a better approach, it may still not be sufficient to solve verification problems for large circuits, because the sizes of intermediate BDDs during construction of the BDD for the output may become too large.

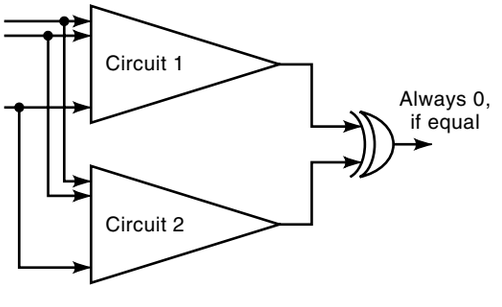

Figure 9. Creating a miter to check the equivalence of two circuits.

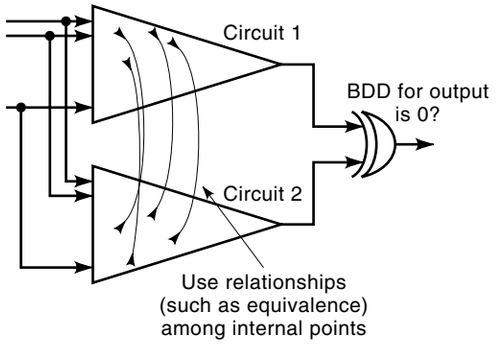

Figure 10. Use relationship among internal signals to reduce the size of BDD for output.

The approach just mentioned can, however, be significantly improved by using information on the relationship among values of internal signals in the two circuits. For the equivalence check of two combinational circuits, there are cases in which we can verify many larger circuits, for example, circuits having 100,000 gates or larger. One such case involves two similar circuits, for example, one circuit is a slight modification of the other. This occurs frequently in real designs, as designers try to improve the performance of circuits by modifying circuits partially or incrementally. If the two circuits are similar we can expect much signal value dependency among internal signals in the two circuits. For example, if the circuit optimization performed by designers consists of just inserting buffers to speed up a circuit, we will see much internal equivalence between the two circuits. By using internal equivalence we can partition circuits into smaller ones and will only need to check the equivalence among those partitioned circuits instead of the original large circuits.

Also, we can use relationships among internal signals in order to reduce sizes of intermediate BDDs when constructing BDDs for the output of the EXCLUSIVE-OR gate from output to inputs (see Fig. 10). By appropriately using those relationships and reducing BDD sizes, we can verify circuits having more than 100,000 gates rather easily if the two circuits to be compared are similar. Since this approach can treat circuits of real-life sizes, it is becoming widely used (16,17).

Formal Verification of Sequential Circuits

So far we have discussed only combinational circuits. Now we describe techniques on how to verify sequential circuits formally. First we discuss comparison between two sequential circuits. Since sequential circuits generate output sequences of varying time units, we have to make sure that the outputs have the same values at all times. That is, as shown in Fig. 11, two sequential circuits are connected and we check to see if the values of the outputs are always same (18). Since there are only finite number of flip-flops, the number of possible states in the sequential circuits is finite. Therefore, when we have checked the values of the outputs for all possible states in the two circuits, we can finish verification. For each state, essentially the same procedure as for combinational verification is followed, using the method shown in the previous sections.

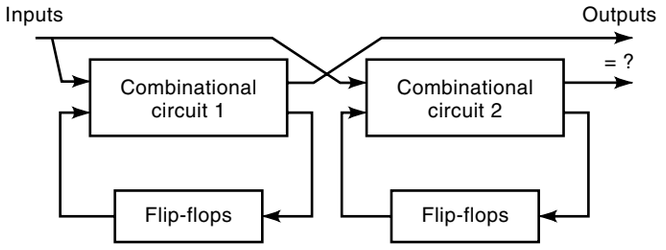

Figure 11. Verification of sequential circuits.

A state-transition graph can be extracted from the sequential circuit. An example state transition graph is shown in the left side of Fig. 12. s_0 is the initial state, which corresponds to the reset state of the original circuit. In this case, there are three additional states and state transitions that interconnect them. All possible behaviors are represented as are all possible state transitions starting from the initial state s_0, as shown on the right side of the figure. This is also called a computation tree, because it represents all possible computations that can be done by the state-transition graph on the left side.

Thus the goal of sequential verification is to ensure that the values of the outputs are equal to the specified values at each node of the computation tree. This can be checked by traversing the state-transition graph features one by one until a state that has been already traversed is reached. This is basically a depth-first search on computation trees. The time to complete this process, however, is exponential in the number of flip-flops, since there are 2^n states in n flip-flop circuits. Hence this approach does not work for large circuits (19,20).

Another method for traversing state-transition graphs is based on a width-first traversal on computation trees, as shown in Fig. 13. It maintains a set of states that have already been checked. First the set has just the initial state s_0 in the case of Fig. 13. In the next step, it will have s_1, s_2, and s_3 as well. Those are the states that can be reached directly by a single state transition from the state s_0. Then, in the next step, we see that no more states can be added to the set, and therefore the search terminates and we have traversed everything. The key idea here is to process sets of states instead of each state individually.

The next question is how to represent sets of states efficiently. One commonly used approach is to represent sets with their characteristic logic functions. We introduce new variables to encode each state in the state-transition graph.

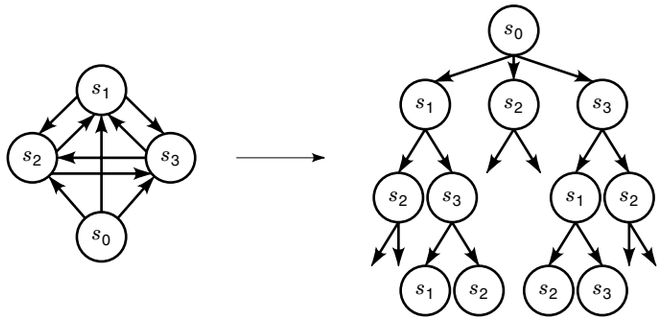

Figure 12. State-transition graph and its trace of transitions.

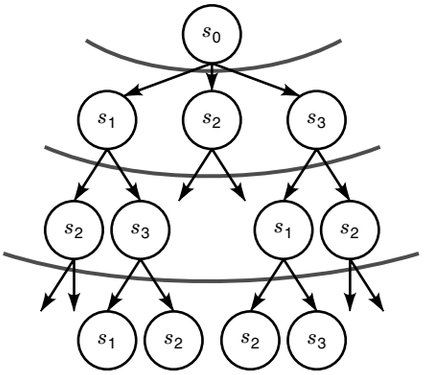

Figure 13. Breadth-first search of state transitions.

Basically we need $\log_2$(numbers of states) new variables. Then we assign values of those variables so that each state has different values. This is a type of state assignment for the given state-transition graph. Then a set of states can be represented as a disjunction of values of the variables for those states.

Let us see an example, shown in Fig. 14. Since there are four states, we need two variables for encoding of states. Suppose they are x and y, and we use the following state encoding:

$$\begin{array}{ll} A & (x, y) = (0, 0) \\ B & (x, y) = (0, 1) \\ C & (x, y) = (1, 0) \\ D & (x, y) = (1, 1) \end{array}$$

From this, we can get the corresponding state-transition table as shown in Fig. 15. In the table, x and y are encoding variables corresponding to the present states and x' and y' are those corresponding to the next states. From this table, we can compute transition relations for the state transition graph as follows:

$$\mathrm{TR}(x, y, x', y') = \overline{x} \cdot \overline{y} \cdot \overline{x'} \cdot y' + \overline{x} \cdot y \cdot x' \cdot \overline{y'} + \overline{x} \cdot y \cdot x' \cdot y' + x \cdot \overline{y} \cdot \overline{x'} \cdot y' + x \cdot \overline{y} \cdot x' \cdot y'$$

$\mathrm{TR}(x, y, x', y')$ is 1 if and only if there is a state transition from the state (x, y) to the state (x', y').

Now we can traverse the state transition graph in Fig. 14 in a breadth-first order. Let us assume the initial state to be $\{A\}$. In the next step we get the set of states $\{A, B\}$. Then we get $\{A, B, C, D\}$ in the following step. This can be computed

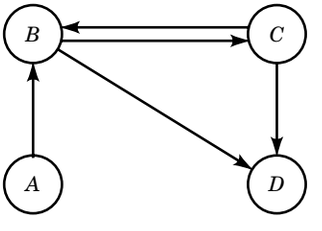

Figure 14. Symbolic manipulation of breadth-first traversal of the state-transition graph.

Present x y	Next x' y'
A 0 0	B 0 0
B 0 1	C 0 1
B 0 1	D 0 1
C 1 0	B 1 0
C 1 0	D 1 0

Figure 15. State-transition table corresponding to the state transition graph.

using the transition relation and the state-encoding variables x, y. For example, in order to get the set of states $\{A, B, C, D\}$ from the set of states $\{A, B\}$, we compute as follows. $\{A, B\}$ can be represented as $\bar{x} \cdot \bar{y} + \bar{x} \cdot y = \bar{x}$ and so we compute

$$\bar{x} \cdot \mathrm{TR}(x, y, x', y') = \bar{x} \cdot \bar{x} \cdot \bar{y} \cdot \overline{x'} \cdot y' + \bar{x} \cdot y \cdot x' \cdot \overline{y'} + \bar{x} \cdot y \cdot x' \cdot y' + x \cdot \bar{y} \cdot \overline{x'} \cdot y' + x \cdot \bar{y} \cdot x' \cdot y' = x + y$$

$x + y = \bar{x} \cdot y + x \cdot \bar{y} + x \cdot y$ corresponds to the set $\{B, C, D\}$. By adding the original set $\{A, B\}$, the result is $\{A, B, C, D\}$. Since computing state transitions is now formalized as the manipulation of logic functions, this process can be efficiently automated by using BDDs. It is called the *symbolic traversal* of state-transition graphs and is now widely used. State-of-the-art implementation of this approach can verify circuits having up to around 200 flip-flops, which may have 2^{200} states (21–24).

MANUFACTURING FAULT DIAGNOSIS

Fault location for digital logic circuits is studied here. After testing is performed to determine whether a circuit is faulty, fault location or *diagnosis* is performed to locate the failure. Diagnosis may be performed with a view to improving the manufacturing process or may be intended for the identification and replacement of a faulty subcircuit. Efficient diagnosis has been known to yield rapid improvement. Given a defective chip and good design criteria, the aim of the diagnostic process is to identify a subset of faults that can explain all the errors observed while testing the chip. Techniques described in this article are typically used to reduce the time required for expensive failure analysis procedures that aim at the physical confirmation of the defect (e.g., under an electron microscope). The time reduction is achieved by reducing the number of candidates to examine by analysis at the logic level.

We shall first review diagnosis techniques based on their classification of usage of precomputed information (as opposed to run-time analysis) in the diagnosis process. The techniques are broadly grouped under static (cause–effect), dynamic (effect–cause), and integrated techniques. Then, we briefly review work on important tools required for diagnosis, diagnostic fault simulation, and diagnostic test generation. After this, we review diagnosis techniques specifically designed to handle unmodeled faults. Specific techniques that are representative of their genre are explained in greater detail whenever possible.

Diagnosis Strategies

Diagnostic techniques can be broadly classified into three groups. The first group, called *static* (*cause–effect*) fault diagnosis, uses precomputed information in the form of fault dictionaries for matching with the faulty responses produced by defective circuits (25–33). Fault dictionaries store output information, produced by the circuit under consideration, on application of the given set of test vectors and under the influence of the set of modeled faults. In contrast, *dynamic* (*effect–cause*) diagnosis techniques detect the faulty behavior of the circuit while the test set is applied (34–40). Recent trends show the increasing popularity of *integrated* diagnosis techniques in which the focus is on using small amounts of precomputed information and coupling this with efficient dynamic algorithms to perform fault location (31,41).

The main advantage of static fault diagnosis techniques occurs when multiple copies of the same design are being diagnosed (as in an integrated-circuit manufacturing process). Another significant advantage of the fault dictionary approach is that it is relatively simple to use. However, a common problem associated with these techniques is that it is typically infeasible to store all the precomputed information. (Typical full fault dictionaries can require several gigabytes of storage for even moderately large circuits containing 20,000 gates.) Hence, research in this direction has concentrated on providing compact fault dictionaries. The main motivation for dynamic diagnosis algorithms is that they do not require any precomputed information. This eliminates the storage problem with fault dictionaries and also relies to a lesser extent on the type of defects being diagnosed. However, this results in the fact that the time spent for diagnosing each single faulty unit is typically much larger than that required by static techniques. Hence, research in this area has concentrated on reducing the run times. Integrated techniques have been proposed to incorporate the advantages of both the static and dynamic techniques. The main advantage of integrated fault diagnosis is the flexibility provided in choosing the kind and amount of precomputed information. This, in turn, has an effect on the time required for performing diagnosis at run time.

Static Fault Diagnosis. An example of a fault dictionary is shown in Fig. 16(a) for a circuit with six modeled faults, two vectors, and two primary outputs. A typical use of the information in this dictionary could be in the following manner: If the faulty response produced by a defective chip on the application of vectors v_1 and v_2 was 10 and 11, then the dictionary could be used to indicate fault 5's presence in the defective chip. Techniques for handling situations when the faulty responses do not match with any of the stored responses (*exactly*) are discussed later in this article under the section Unmodeled Fault Diagnosis. Since fault dictionaries are typically prohibitively large to store, fault-dictionary compaction has been an important focus of research. Past work addressing the size problem has yielded solutions in two distinct directions. The first set of contributions provide fault-dictionary compaction targeting high modeled fault resolution (25,29–32), while the second set offers alternative representations for storing the full fault dictionary (30,31,33).

Fault-Dictionary Compaction Research

Pass/Fail Dictionary (29). This type of fault dictionary records the faults detected, potentially detected, and not de-

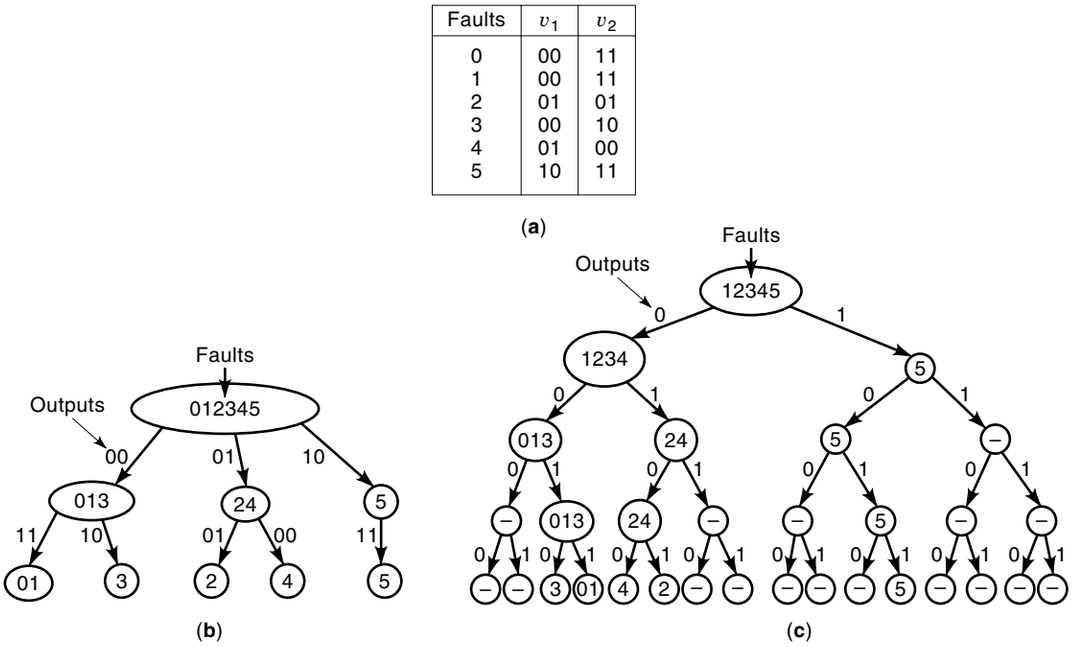

Figure 16. (a) Matrix dictionary; (b) vector-based tree; (c) output-based tree.

tected for each vector. It does not record detections separately by output. It is created by a single full-fault simulation and is much smaller than a full-fault dictionary. But, as might be expected, this dictionary loses some diagnostic capability when compared with the full-fault dictionary.

Compact Dictionary (29). One method of enhancing the diagnostic capability of the pass/fail dictionary is to add output information. Such an approach is used in the creation of the compact fault dictionary. The compact algorithm is computationally intensive, requiring multiple simulations of all vectors against some faults, plus a full-fault simulation to produce the vector dictionary and another to produce the final dictionary after extra columns are added. The dictionary produced is known to be considerably compressed, with no loss of modeled fault resolution (30).

Sequential Dictionary (30). In this technique, a pass/fail dictionary is enhanced by a single full-fault simulation. An entry is added to the dictionary for any vector and output that distinguishes between any pair of faults not previously distinguished. This is computationally cheaper that the compact dictionary generation algorithm. There is no loss of modeled fault resolution.

List Splitting Dictionary (30). This dictionary is created by using efficient list splitting. The lists correspond to faults that are not distinguished at each vector–output combination in the diagnosis process. However, it is not accurate for sequential circuits; hence the diagnostic resolution suffers.

Drop on K *Dictionary (30).* While creating this dictionary, the fault simulator drops each fault after its Kth detection and creates an otherwise standard dictionary, including possible detections until each fault's Kth definite detection. This technique assumes that K detections distinguish between most fault pairs and that some faults cause errors for many vectors, filling dictionaries with unneeded data. Simulation costs here are less than those for a full-fault dictionary.

First Failing Pattern Dictionary (30). This is a special case of the drop on K dictionary for $K = 1$.

Detection Frequency Dictionary (30). A full-fault simulation is performed, and for each fault f, the number of vectors definitely (d_f) and potentially producing errors (p_f) are counted. Each fault can cause errors numbering between d_f and $d_f + p_f$. The list of faults that causes each possible number of errors forms an indistinguishability class for this dictionary. The resolution of this dictionary is poor in comparison with other schemes.

Tree-Based Compaction Dictionary (32). Diagnostic experiment trees (as shown in Fig. 16) have also been used to identify information that is not diagnostically useful (for modeled faults) to provide compact dictionaries. An example of information that is eliminated corresponded to output information for faults after they were completely distinguished from other faults.

Full-Fault-Dictionary Representation Research. A key problem with the compaction techniques that have been previously described lies in the fact that the information that they identify as *diagnostically useful* is useful only with respect to modeled faults. Hence, the diagnostic accuracy of such dictionaries in the presence of unmodeled faults may degrade. Thus there is a necessity for developing storage structures that enable efficient representation of the information in the full-fault dictionary. This approach is orthogonal to compaction, which has achieved storage savings by removing output information.

Matrix Dictionary (42). Full-fault dictionaries need to store output information corresponding to each vector and fault pair. Conventionally, they have been stored using a matrix representation. For a circuit with v vectors, o outputs, and f faults, the size of the matrix dictionary is vof bits for combinational circuits and $2vof$ bits for sequential circuits.

List Dictionary (31). List-based dictionaries have been proposed as an alternative to the matrix representation (31). The list dictionary records only information corresponding to detections.

Tree-Based Fault-Dictionary Compaction and Representation. Diagnostic experiment trees (32,33,43) are powerful tools for modeling the information corresponding to a diagnostic experiment. Diagnostic experiment trees are *labeled trees;* hence the dictionary storage problem can be reduced to a labeled-tree encoding problem. Two labeled trees that were used to represent the diagnostic experiment are shown in Figs. 16(b) and 16(c).

Definition 1 [Vector-Based Diagnostic Experiment Tree $T_V(V, E)$]. A diagnostic experiment tree in which each level represents the application of a test vector and in which each edge $e \in E(T_V)$ is associated with a list of outputs $O(e)$ that is the set of *all* the primary outputs of the circuit is called a vector-based diagnostic experiment tree.

Definition 2 [Output-Based Diagnostic Experiment Tree $T_O(V, E)$]. A diagnostic experiment tree in which each level represents a (test vector, output) pair rather than a test vector, and in which each edge $e \in E(T_O)$ is associated with a single primary output of the circuit is called an output-based diagnostic experiment tree.

Example. Figures 16(b) and 16(c) show the vector-based and output-based diagnostic experiment trees corresponding to the full-fault dictionary shown in the matrix format in Fig. 16(a).

The information embedded in the vector-based diagnostic experiment tree is fully exploited to identify output sequences that may be eliminated to produce highly compact dictionaries even while they retain high diagnostic resolution with respect to modeled faults. The compact storage structures developed for storing the information identified to be useful provide compaction of up to 2 orders of magnitude (32). For full-fault-dictionary representation, it is shown that both of the labeled trees can be efficiently represented by disjointly storing the label information and the underlying unlabeled tree. The vector-based tree is encoded by the use of a compact binary code, while the regular structure of the output-based tree is exploited to provide a spectrum of *eight* alternative representations for the full-fault dictionary. It is worth noting that the currently known *list* and the *matrix* formats arise as special cases in this framework. The results give some of the best currently known storage requirements for full-fault-dictionary representation (33).

Dynamic Diagnosis. Dynamic diagnosis techniques analyze the output responses produced by the failed chip at diagnosis time with the possible use of diagnostic fault simulation to derive a set of failures that best explain the set of observed responses. The approach does not require the storage of any precomputed information. We present a brief overview of dynamic diagnosis research with emphasis on work targeting large, practical circuits.

The Deduction Algorithm (42). This analysis processes the response obtained from the faulty unit to determine the possible stuck-at faults that can generate that response, based on deducing internal values in the unit under test (UUT). Any line for which both 0 and 1 values are deduced can be neither s-a-0 (stuck-at-0) nor s-a-1 (stuck-at-1) and is identified as fault-free. Faults are located on some of the lines that cannot be proved normal. Internal values are computed by the *deduction algorithm,* which implements a line-justification process the primary goal of which is to justify all the values obtained at the POs (primary outputs), given the tests applied at the PIs (primary inputs). Backtracking is used either to recover from incorrect decisions or to generate all possible solutions. However, no results are available from this work for circuits of practical size.

The Pair-Analysis Approach (34). In contrast to other techniques, this work considers pairs of vectors rather than single vectors. This gives the method an additional capability to encode polarity of different paths in the circuit by applying transitions on a limited number of inputs. The primary claim in this paper is that by the use of this technique, all faults can be diagnosed to their equivalence classes. This work is applicable only to combinational circuits.

Sensitizing Input Pairs (45). A technique that has some similarity to the pair-analysis approach has been recently proposed. This is the first work that successfully provided analysis-based solutions to nontrivial sequential circuits. However, like other analysis techniques, it is still not possible to apply this technique to large circuits.

Full-Scan Diagnosis Algorithms (35,36). This work targets full-scan designs. The heart of this work lies in an efficient vector parallel fault simulator that rapidly reduces the number of candidate faults based on the faulty responses and the expected failures due to the fault.

Modeled Fault Simulation (38,46,47). A common dynamic diagnosis strategy that has been used to diagnose large circuit defects is to obtain expected output responses by the use of modeled fault simulation. However, due to the excessive fault-simulation costs, the time taken to perform the diagnosis may be large for repeated diagnosis of large circuits.

Path Tracing (PT) (40). A strategy for dynamic diagnosis with reduced diagnostic fault simulation time performs fault dropping during diagnosis time with the help of critical path tracing. Faults are dropped when it is decided that they are on lines that do not influence any faulty output lines.

Example Dynamic Diagnosis. An example of a diagnosis decision arrived based on path tracing is shown in Fig. 17. The output of gate *e* fails. The path trace starts from this output and proceeds to the inputs. Because gate *e* has two controlling inputs, the trace continues from one of them. Node *B*, which is part of the bridging fault *A@B* (node A shorted with node B), is included (along with other candidates on the paths traced) in the candidate set of faulty nodes by the path–trace procedure.

Expert Systems and Artificial Intelligence Techniques. Diagnosis has been attempted in rule-based expert systems that utilize encoded empirical knowledge obtained from human experts. These systems are not entirely deductive and bear

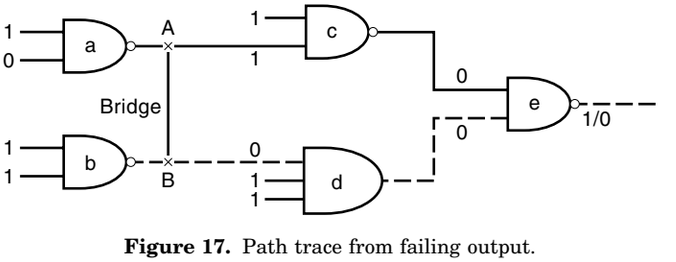

Figure 17. Path trace from failing output.

some resemblance to the fault-dictionary approach. In contrast, some artificial intelligence researchers have proposed techniques that are based on more detailed structural and behavioral models of the system being diagnosed. However, the most important problem with such techniques is that they target only small circuits and do not attempt to tackle the problems that arise with more elaborate designs.

Integrated Diagnosis. The prohibitive size of fault dictionaries and the large run times required for dynamic diagnosis have given rise to integrated fault-diagnosis techniques, in which the focus is on storing a limited amount of essential information and utilizing this information effectively along with analysis or simulation at run time. We now provide an overview of this research.

Dynamic Dictionaries. This approach involves two stages (31,41). The first stage identifies a small group of candidate faults, and then a small part of the full-fault dictionary is generated dynamically in the second stage for just those faults and for only a few of the vectors that detect them. Hence, two-stage fault isolation avoids the static cost normally associated with full dictionaries and most of the computation time that is required in a pure dynamic technique, while still providing most of the resolution. The *limited* dictionary used in the first stage of the two-stage process is a very small dictionary that can be generated by limited fault simulation. The diagnosis algorithm lists all candidate faults that have been observed by comparing observed errors with records in the limited dictionary. Then, in a second stage, a set of vectors is fully simulated against candidate faults, and a matching algorithm ranks all faults. Experimental results were provided for a variety of benchmark circuits and industrial implementations. It was also shown that the loss of resolution incurred was not significant.

State-Information-Based Diagnosis. State-information-based diagnosis solves a crucial problem with traditional diagnostic techniques based on storage (48). Typically, such techniques store only primary output-based information, offering only a black-box view of the circuit and thus little diagnostic flexibility. This technique provides a solution by storing information corresponding to the internal nodes in the circuit, namely the state nodes. The selective storage of state information has been shown to improve the time for diagnostic fault simulation significantly. Experimental results on large circuits were presented.

Level-Information-Based Diagnosis. Precomputed information tracking the diagnostic classes at each level of the diagnostic experiment tree, specifically targeting a reduction in the fault simulation costs to be incurred at diagnosis time, is the key contribution of this work (49). Fault simulation costs are modeled in terms of computations associated with each (fault, vector) pair.

Tools for Diagnosis: Diagnostic Fault Simulation and Test Generation

Diagnostic Fault Simulation. Diagnostic fault simulation is useful for determining the diagnostic capability of a given test set and for generating fault dictionaries and diagnostic information specific to a given test set. Diagnostic capability is reported using various diagnostic measures. Diagnostic test generation involves generating tests to distinguish between fault pairs. Efficient generation of diagnostic test vectors can be assisted by a fast diagnostic fault simulator. Typically, diagnostic fault-simulation techniques have focused on simulation based on stuck-at faults and the developed measures are also for the same models. Rapid techniques are available both for combinational and sequential circuits, and we review the more general case of sequential circuits here.

During fault simulation of a circuit starting from an unknown state, a good or faulty sequential circuit can produce a 0, 1, or X on each primary output for each test vector input, where X is an unknown value whose actual binary value depends on the initial state of the machine. If fault simulation indicates that a fault f_i produces an output of 0 and another fault f_j produces an output of 1 on the same primary output for the same input, then the faults f_i and f_j are said to be distinguished. However if a fault f_i produces an output of 0 or 1 and another fault f_j produces an output of X, then it is possible that the faults f_i and f_j may not be distinguished. Therefore, the pessimistic assumption is made that an output of 1 or 0 is indistinguishable (with respect to this test set) from an output of X.

Diagnostic Measures. Camurati et al. (50) proposed two diagnostic measures. *Diagnostic resolution* (DR) is the fraction of fault pairs distinguished by a test set. *Diagnostic power* (DP) is the fraction of faults that are fully distinguished. A fault is fully distinguished if the test set distinguishes it from every other fault in the fault list. A third measure (51), which gives a more complete picture, is to identify sets of fault-equivalence classes and report the number of these classes by size; this measure is applicable to combinational circuits and sequential circuits that start from a known reset state. This is extended to *indistinguishable fault classes* (38) to account for unknown values occurring at the outputs of sequential circuits during simulation. Another measure, the *diagnostic expectation* (30), is the average of indistinguishability class sizes over all faults. It is assumed that all faults are equally likely to occur.

Distinguishability Matrix Approach. Early methods for performing diagnostic fault simulation for moderately large circuits (38) used a distinguishability matrix. The distinguishability matrix is an $f \times f$ matrix, where f is the number of faults. An entry of 1 indicates that the two faults specified at the intersection of the row and column are distinguished by some sequence of test vectors in the test set. It requires $O(f^2)$ space, and the time complexity is $O(vof^2)$, where v is the number of vectors in the set and o is the number of outputs in the circuit.

List-Based Methods. Ryan, Fuchs, and Pomeranz (30) mention that a more efficient way to represent faults that are indistinguishable by a given test set is by using lists of faults. Jou and Chen (52) and Chen and Jou (53) represent pairs of indistinguishable faults using lists. This representation is a compact implementation of the distinguishability matrix. It is equivalent to storing only those entries of the distinguishability matrix with values of 0. Here, faults may appear in multiple lists.

The indistinguishability relationship between all pairs of faults can be represented as an undirected graph, with the faults as nodes and the indistinguishability relationships between them as edges. Previous approaches essentially represent this graph as an adjacency matrix (38) or as incidence lists (52,53).

Later representations (39) avoided explicit storage of the indistinguishability relationship between all pairs of faults, but represent the indistinguishability relationship between classes of faults. Each fault is present in only one of the classes. This makes the representation more compact than those previously proposed (38,52,53). Although the worst-case space complexity is still $O(f^2)$, experimental results demonstrated that the average memory usage is almost linear for the benchmark circuits. The representation also reduces the number of output response comparisons between faults and hence speeds up the simulation process.

Diagnostic Test-Pattern Generation. Diagnostic automatic test pattern generation (DATPG) is critical to performing efficient fault diagnosis. In diagnostic test generation, the goal is to find a test sequence such that the circuit produces a different response under one fault than it does under another. Such techniques have been primarily targeted towards stuck-at faults and for combinational circuits, although recent work has made progress towards both unmodeled faults and sequential circuits.

The diagnostic test-generation problem for sequential circuits is more acute than its combinational circuit counterpart mainly because of multiple time frames that need to be handled. The problem is compounded by the *unknown* values in state elements; these unknown values may increase the number of fault pairs that need to be explicitly considered by a diagnostic test generator.

Combinational Circuits. Work on DATPG for combinational circuits has been developed based on both functional (e.g., BDD-based) and structural techniques (PODEM-based) (50,54–57). DIATEST (56) is a combinational diagnostic test-generation program that was developed based on the conversion of a conventional test generator into a diagnostic test generator. Complete results (with no aborted fault pairs) were provided on moderate-sized (on the largest standard public benchmark circuits) combinational circuits. Since equivalence identification, much like redundancy identification, is a computationally intensive operation in the DATPG process, techniques to identify combinational equivalences (57–61) have been proposed.

Sequential Circuits. Formal techniques have also been used for sequential circuit diagnostic test generation (62,63); however, the drawbacks of these approaches are the assumption of a fault-free reset state and the inability to handle large circuits due to memory requirement problems. Simulation-based diagnostic test generation algorithms for large sequential circuits have also been presented (64), but there is a lack of indistinguishability identification. Later, a powerful method to modify a conventional sequential test generator into a sequential diagnostic test generator has been proposed (65). The method utilizes circuit netlist modification along with a forced 0/1 or 1/0 (66) value at a primary input in the modified circuit.

Indistinguishability. There is also evidence (62,63,65,67,68) indicating that a main burden of diagnostic test generation is in proving indistinguishability. Another difficulty in solving this problem arises in sequential circuits because the terms *distinguishable, indistinguishable, detectable,* and *undetectable* take on different meanings with different test methodologies [multiple observation time (69,70) or conventional, gate-level test generation with single observation time and three-valued simulation (42)]. Methods to characterize these relations and identify them implicitly (without explicitly making a call to the diagnostic engine for each relation) have simplified the computational task of diagnostic test-pattern generation (67,68).

Unmodeled Fault Diagnosis

The *fault model* used to predict defect behavior plays an important role in diagnosis (47). In order for a fault model to be valid for diagnosis it should accurately model the corresponding defect, and such defects should occur in real circuits (71). It is worth noting that static (cause–effect) techniques are perhaps more dependent on the fault models than dynamic (effect–cause).

Based on Modeled Faults. A typical approach for diagnosing unmodeled faults is to use the information available from the modeled faults in a controlled manner to make conclusions about the presence of unmodeled faults. Issues concerning accuracy and the time required to perform diagnosis govern the kind of matching algorithm being used. These schemes can range from dropping all faults whose response shows a definite mismatch with the observed faulty response (applicable to pure modeled fault diagnosis; fast) (31,35,36) to dropping few or no faults with the use of scoring schemes to obtain a set of candidate faults (applicable to arbitrary unmodeled fault diagnosis; slow) (31,38). Schemes studying the use of various combinations of matching schemes and fault models have also received research attention, and information corresponding to vectors showing failures and vectors showing no failures has been used to obtain separate matching parameters (31,47,72). This approach has been suggested to attain better diagnosis for unmodeled faults. An intuitive explanation for the better accuracies obtained using the separate handling of the failing (failures observed) and passing (good values observed) vectors is given from the fact that obtaining separate parameters makes it possible to explain observed failures as opposed to other matching schemes in which matching of an error is not distinguished from the matching of a good value.

Bridging Fault Diagnosis. A common failure mode in current complementary metal-oxide semiconductor (CMOS) technologies is that of *short circuits*. Thus, many failures can be modeled as *bridging faults* and they have hence received extra attention. Techniques for diagnosing bridging faults have been primarily targeted at combinational circuits because of the large computational overheads associated with the simulation of bridging faults and the lack of a clear understanding of the complete effects of sequential bridging faults. Even for combinational circuits, only a limited set of realistic bridging faults that are extracted from the layout (73) are typically used because of the prohibitively large numbers of all possible bridging faults, even for small circuits. An additional complicating factor for these faults is that a short circuit (that may produce an intermediate voltage value) may be interpreted differently by logic gates downstream from the bridged lines due to variable input logic thresholds. This is known as the *Byzantine generals problem.*

Several techniques have been proposed for bridging-fault diagnosis in combinational circuits. The most popular approaches are ones that use stuck-at dictionaries to diagnose bridging faults. The reason for this is that this avoids compu-

tationally intensive bridging-fault simulation. Millman, McClusky, and Acken (74) presented an approach to diagnose bridging faults using stuck-at dictionaries. Chess et al. (46) and Lavo, Larrabee, and Chess (72) improved on this technique. These techniques enumerate bridging faults and are hence constrained to use a reduced set of bridging faults extracted from the layout. Furthermore, they need to either store a stuck-at fault dictionary or perform stuck-at fault simulation. Chakravarty and Liu (75) proposed a technique based on I_{ddq} (quiescent current) using only good circuit simulation. Chakravarty and Gong (76) described a voltage-based algorithm that used the wired-AND (wired-OR) model. This work (76) implicitly considers all bridging faults. It is worth noting that wired-AND and wired-OR models that are assumed work only for technologies for which one logic value is always more strongly driven than the other. A deductive technique for combinational circuits that does not explicitly simulate faults has been proposed. However, this technique is not complete because it only reduces the candidate set of bridging faults and may end up with a potentially large set of candidates.

BIBLIOGRAPHY

1. A. Gupta, Formal hardware verification methods: A survey, *Formal Methods Syst. Des.,* **1** (2/3): 151–238, October, 1992.
2. S. B. Aker, Binary decision diagrams, *IEEE Trans. Comput.,* **C-27**: 509–516, 1978.
3. R. E. Bryant, Graph-based algorithms for boolean function manipulation, *IEEE Trans. Comput.,* **C-35**: 667–691, 1986.
4. S. Minato, N. Ishiura, and S. Yajima, Shared binary decision diagram with attributed edges for efficient boolean function manipulation, *Proc. 27th ACM / IEEE Des. Autom. Conf.,* 1990, pp. 52–57.
5. S. J. Friedman and K. J. Spowit, Finding the optimal variable ordering for binary decision diagrams, *Proc. 24th ACM / IEEE Des. Autom. Conf.,* 1987, pp. 348–356.
6. M. Fujita, H. Fujisawa, and N. Kawato, Evaluation and implementation of boolean comparison method based on binary decision diagrams, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD '88),* 1988, pp. 6–9.
7. N. Ishiura, H. Sawada, and S. Yajima, Minimization of binary decision diagrams based on exchanges of variables, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD '91),* 1991, pp. 472–745.
8. T. Kakuda, M. Fujita, and Y. Matsunaga, On variable ordering of binary decision diagrams for the application of multi-level logic synthesis, *Proc. Eur. Des. Autom. Conf. (EDAC '91)* , 1991, pp. 50–54.
9. S. Malik et al., Logic verification using binary decision diagrams in a logic synthesis environment, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD '88),* 1988, pp. 6–9.
10. S. Minato, Minimum-width method of variable ordering for binary decision diagrams, *IEICE Jpn. Trans. Fundam.,* **E75-A** (3): March, 1992.
11. R. Rudell, Dynamic variable ordering for ordered binary decision diagrams, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD '93),* 1993.
12. K. S. Brace, R. L. Rudell, and R. E. Bryant, Efficient implementation of a bdd package, *Proc. 27th ACM / IEEE Des. Autom. Conf.,* 1990, pp. 40–45.
13. J. C. Madre and J. P. Billon, Proving circuit correctness using formal comparison between expected and extracted behavior, *Proc. 25th ACM / IEEE Des. Autom. Conf.,* 1988, pp. 205–210.
14. D. Brand, Verification of large synthesized designs, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD '93),* 1993.
15. W. Kunz, Hannibal: An efficient tools for logic verification based on recursive learning, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD '93),* 1993.
16. J. Jain, R. Mukherjee, and M. Fujita, Advanced verification techniques based on learning, *Proc. ACM / IEEE Des. Autom. Conf.,* 1995.
17. A. Kuehlmann and F. Krohm, Equivalence checking using cuts and heaps, *Proc. 34th ACM / IEEE Des. Autom. Conf.,* 1997.
18. O. Coudert and J. C. Madre, A unified framework for the formal verification of sequential circuits, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD '90),* 1990, pp. 126–129.
19. E. M. Clarke and E. A. Emerson, Automatic verification of finite-state concurrent systems using temporal logic specification, *ACM Trans. Programm. Lang. Syst.,* **8** (2): 244–263, 1986.
20. M. Fujita, H. Tanaka, and T. Moto-oka, Logic design assistance with temporal logic, *Proc. IFIP WG10.2 Int. Conf. Hardw. Descript. Lang. Their Appl.,* 1985.
21. J. R. Burch, et al., Sequential circuit verification using symbolic model checking, *Proc. 27th ACM / IEEE Des. Autom. Conf.,* 1990, pp. 46–51.
22. J. R. Burch et al., Symbolic model checking: 10^{20} states and beyond, *Proc. 5th Annu. IEEE Symp. Logic Comput. Sci.,* 1991.
23. R. P. Kurshan, Automata-theoretic verification of coordinating processes, *Lect. Notes Comput. Sci.,* **430**: 414–453, 1990.
24. H. Touati et al., Implicit state enumeration of finite state machines using bdds, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD '90),* 1990, pp. 130–133.
25. R. E. Tulloss, Size optimization of fault dictionaries, *Proc. Int. Test Conf., 1978,* pp. 264–265.
26. R. E. Tulloss, Fault dictionary compression: Recognizing when a fault may be unambiguously represented by a single failure detection, *Proc. Int. Test Conf.,* 1980, pp. 368–370.
27. J. Richman and K. R. Bowden, The modern fault dictionary, *Proc. Int. Test Conf.,* 1985, pp. 696–702.
28. V. Ratford and P. Keating, Integrating guided probe and fault dictionary: An enhanced diagnostic approach, *Proc. Int. Test Conf.,* 1986, pp. 304–311.
29. I. Pomeranz and S. M. Reddy, On the generation of small dictionaries for fault location, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD '92),* 1992, pp. 272–279.
30. P. G. Ryan, W. K. Fuchs, and I. Pomeranz, Fault dictionary compression and equivalence class computation for sequential circuits, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD '93),* 1993, pp. 508–511.
31. P. G. Ryan, *Compressed and Dynamic Fault Dictionaries for Fault Isolation,* Tech. Rep. UILU-ENG-94-2234, Center for Reliable and High-Performance, Urbana-Champaign: Computing, Univ. of Illinois, 1994.
32. V. Boppana and W. K. Fuchs, Fault dictionary compaction by output sequence removal, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD '94),* 1994, pp. 576–579.
33. V. Boppana, I. Hartanto, and W. K. Fuchs, Full fault dictionary storage based on labeled tree encoding, *Proc. VLSI Test Symp.,* 1996, pp. 174–179.
34. H. Cox and J. Rajski, A method of fault analysis for test generation and fault diagnosis, *IEEE Trans. Comput.-Aided Des.,* **7**: 813–833, 1988.
35. J. A. Waicukauski et al., Fault simulation for structured VLSI, *VLSI Syst. Des.,* **6** (12): 20–32, 1985.
36. J. A. Waicukauski and E. Lindbloom, Failure diagnosis of structured VLSI, *IEEE Des. Test Comput.,* **6(4):** 49–60, 1989.
37. M. Abramovici and M. A. Breuer, Fault diagnosis based on effect-cause analysis, *Proc. 24th ACM / IEEE Des. Autom. Conf.,* 1987, pp. 69–76.

38. E. M. Rudnick, W. K. Fuchs, and J. H. Patel, Diagnostic fault simulation of sequential circuits, *Proc. Int. Test Conf.,* 1992, pp. 178–186.
39. S. Venkataraman et al., Rapid diagnostic fault simulation at stuck-at faults in sequential circuits using compact lists, *Proc. 32nd ACM / IEEE Des. Autom. Conf.,* 1995, pp. 133–138.
40. S. Venkataraman, I. Hartanto, and W. K. Fuchs, Dynamic diagnosis of sequential circuits based on stuck-at faults, *Proc. VLSI Test Symp.,* 1996, pp. 198–203.
41. P. Ryan, S. Rawat, and W. K. Fuchs, Two-stage fault location, *Proc. Int. Test Conf.,* 1991, pp. 963–968.
42. M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital System Testing and Testable Design,* New York: Computer Science Press, 1990.
43. Z. Kohavi, *Switching and Finite Automata Theory,* New York: McGraw-Hill, 1978.
44. F. C. Hennie, *Finite-State Models for Logical Machines,* New York: Wiley, 1968.
45. N. Yanagida, H. Takahashi, and Y. Takamatsu, Multiple fault diagnosis in sequential circuits using sensitizing sequence pairs, *Proc. Int. Symp. Fault Tolerant Comput.,* 1996, pp. 86–95.
46. B. Chess et al., Diagnosing of realistic bridging faults with stuck-at information, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD '95),* 1995, pp. 185–192.
47. R. C. Aitken and P. C. Maxwell, Better models or better algorithms? Techniques to improve fault diagnosis, *Hewlett-Packard J.,* February, **46** (1): 110–116, 1995.
48. V. Boppana, I. Hartanto, and W. K. Fuchs, Fault diagnosis using state information, *Proc. Int. Symp. Fault Tolerant Comput.,* 1996, pp. 96–103.
49. V. Boppana and W K. Fuchs, Integrated fault diagnosis targeting reduced simulation, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD '96),* 1996, pp. 267–271.
50. P. Camurati et al., A diagnostic test pattern generation algorithm, *Proc. Int. Test Conf.,* 1990, pp. 52–58.
51. K. Kubiak et al., Exact evaluation of diagnostic test resolution, *Proc. 29th ACM / IEEE Des. Autom. Conf.,* 1992, pp. 347–352.
52. J. M. Jou and S.-C. Chen, A fast and memory-efficient diagnostic fault simulation for sequential circuits, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD'94),* 1994, pp. 723–726.
53. S.-C. Chen and J. M. Jou, Diagnostic fault simulation for synchronous sequential circuits, *IEEE Trans. Comput.-Aided Des.,* **16**: 299–308, 1997.
54. P. Camurati et al., Diagnostic oriented test pattern generation, *Proc. Eur. Des. Autom. Conf., (ECAD'90),* 1990, pp. 470–474.
55. J. Savir and J. P. Roth, Testing for, and distinguishing between failures, *Proc. Int. Symp. Fault Tolerant Comput.,* 1982, pp. 165–172.
56. T. Grüning, U. Mahlstedt, and H. Koopmeiners, DIATEST: A fast diagnostic test pattern generator for combinational circuits, *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD'91),* 1991, pp. 194–197.
57. I. Hartanto, V. Boppana, and W. K. Fuchs, Diagnostic fault equivalence identification using redundancy information & structural analysis, *Proc. Int. Test Conf.,* 1996, pp. 294–302.
58. E. J. McCluskey and F. W. Clegg, Fault equivalence in combinational logic networks, *IEEE Trans. Comput.,* **C-20**: 1286–1293, 1971.
59. A. Goundan and J. P. Hayes, Identification of equivalent faults in logic networks, *IEEE Trans. Comput.,* **C-29**: 978–985, 1980.
60. B. K. Roy, Diagnosis and fault equivalences in combinational circuits, *IEEE Trans. Comput.,* **C-23**: 955–963, 1974.
61. A. Lioy, Advanced fault collapsing, *IEEE Des. Test Comput.,* **9** (1): 64–71, 1992.
62. G. Cabodi et al., An approach to sequential circuit diagnosis based on formal verification techniques, *J. Electron. Test.: Theory Appl.,* **4**: 11–17, 1993.
63. K. E. Kubiak, *Symbolic Techniques for VLSI Test and Diagnosis,* Tech. Rep. UILU-ENG-94-2207, Urbana-Champaign: Center for Reliable and High-Performance Computing, University of Illinois, 1994.
64. G. Cabodi et al., GARDA: A diagnostic ATPG for large synchronous sequential circuits, *Proc. Eur. Des. Test Conf.,* 1995, pp. 267–271.
65. I. Hartanto et al., Diagnostic test pattern generation for sequential circuits, *Proc. VLSI Test Symp.,* 1997, pp. 196–202.
66. J. P. Roth, W. G. Bouricius, and P. R. Schneider, Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits, *IEEE Trans. Electron. Comput.,* **EC-16**: 567–579, 1967.
67. V. Boppana, I. Hartanto, and W. K. Fuchs, Characterization and implicit identification of sequential indistinguishability, *Proc. Int. Conf. VLSI Des.,* 1997, pp. 376–380.
68. V. Boppana, *State information-based solutions for sequential circuit diagnosis and testing,* Tech. Rep. CRHC-97-20, Ph.D. thesis, Center Reliable High-Performance Comput., Univ. of Illinois at Urbana-Champaign, 1997.
69. I. Pomeranz and S. M. Reddy, The multiple observation time test strategy, *IEEE Trans. Comput.-Aided Des.,* **40**: 627–637, 1992.
70. I. Pomeranz and S. M. Reddy, Classification of faults in synchronous sequential circuits, *IEEE Trans. Comput.,* **42**: 1066–1077, 1993.
71. R. C. Aitken, Finding defects with fault models, *Proc. Int. Test Conf.,* 1995, pp. 498–505.
72. D. B. Lavo, T. Larrabee, and B. Chess, Beyond the byzantine generals: Unexpected behavior and bridging fault diagnosis, *Proc. Int. Test Conf.,* 1996, pp. 611–619.
73. A. Jee and F. J. Ferguson, Carafe: An inductive fault analysis tool for CMOS VLSI circuits, *Proc. VLSI Test Symp.,* 1993, pp. 92–98.
74. S. D. Millman, E. J. McCluskey, and J. M. Acken, Diagnosing CMOS bridging faults with stuck-at fault dictionaries, *Proc. Int. Test. Conf.,* 1990, pp. 860–870.
75. S. Chakravarty and M. Liu, Algorithms for current monitoring based diagnosis of bridging and leakage faults, *Proc. 29th ACM / IEEE Des. Autom. Conf.,* 1992, pp. 353–356.
76. S. Chakravarty and Y. Gong, An algorithm for diagnosing two-line bridging faults in CMOS combinational circuits, *Proc. 30th ACM / IEEE Des. Autom. Conf.,* 1993, pp. 520–524.

MASAHIRO FUJITA
VAMSI BOPPANA
Fujitsu Labs of America

DETECTION ALGORITHM, RADAR. See RADAR TARGET RECOGNITION.

DETECTION THEORY. See CORRELATION THEORY.

DETECTORS. See DEMODULATION PHOTODETECTORS QUANTUM WELL.

DETECTORS, IONIZATION. See IONIZATION CHAMBERS.

DETECTORS, MICROWAVE. See MICROWAVE DETECTORS.

DETECTORS, SUBATOMIC-PARTICLE. See PARTICLE SPECTROMETERS.

DETECTORS, THERMOPILE. See THERMOPILES.

DETECTORS, ULTRAVIOLET. See ULTRAVIOLET DETECTORS.

DEVICE AND PROCESS MODELING. See MONTE CARLO ANALYSIS.

DEVICE MODELS. See NONLINEAR NETWORK ELEMENTS.

DEVICES, DIAMOND. See DIAMOND BASED SEMICONDUCTING DEVICES.

DEVICES, FIBER-OPTIC. See FIBEROPTIC SENSORS.

DEVICES, ORGANIC. See ORGANIC SEMICONDUCTOR DEVICES.

DEVICES SUPERCONDUCTING. See SUPERCONDUCTING ELECTRONICS.

DEVICES, SURFACE MOUNT. See SURFACE MOUNT TECHNOLOGY.

DIAGNOSIS. See FAULT DIAGNOSIS.

DIAGNOSIS FAULT LOCATION. See DESIGN VERIFICATION AND FAULT DIAGNOSIS IN MANUFACTURING.