Test Pattern Generation for Sequential MOS Circuits by Symbolic Fault Simulation*

Kyeongsoon Cho
Randal E. Bryant
Carnegie Mellon University

Abstract

The COSMOS symbolic fault simulator generates test sets for combinational and sequential MOS circuits represented at the switch level. All aspects of switch-level networks including bidirectional transistors, stored charge, different signal strengths, and indeterminate (X) logic values are captured. To generate tests for a circuit, the program derives Boolean functions representing the behavior of the good and faulty circuits over a sequence of symbolic input patterns. It then determines a set of assignments to the input variables that will detect all faults. Symbolic simulation provides a natural framework for the user to supply an overall test strategy, letting the program determine the detailed conditions to detect a set of faults. Symbolic preprocessing of switch-level networks, combined with efficient Boolean manipulation makes this approach feasible.

1. Overview

Most approaches to test generation (e.g., the D-Algorithm [11] or PODEM [7]) are based on combinatorial search. Given a fault, the program searches for an assignment to the primary inputs that both excites the fault and sensitizes a path to a primary output. While such generators can produce test sets with good fault coverage for some circuits, they suffer from several drawbacks. Most importantly, the computational complexity for sequential circuits with search-based algorithms is usually high, since the search space consists of both space and time. In practice, MOS circuits are often sequential because of the following design tendencies: many memory elements are interspersed in the circuit since storage is cheap; pipelining is often used to enhance circuit performance; and precharged logic (e.g., Domino logic) is common.

There have been many efforts in utilizing *design for testability* methodologies to reduce the computational complexity of test generation for sequential circuits. One proven method is to use a complete scan design methodology to reduce the test generation problem of sequential circuits to that of combinational circuits. For many circuits, however, the cost of linking all registers into a scan chain may be too high. Many such systems can be made testable with partial scan chains and by exploiting the bus structure. A test generation tool, such as ours, that can handle arbitrary sequential design has great potential value.

Most previously reported test pattern generators for MOS circuits worked only for static, combinational circuits [6, 8, 9, 10]. Chen, *et al* [5] developed a program that could handle precharged logic as well, but not true sequential behavior. Furthermore, none of these programs has seen widespread usage, due to their limited generality and their poor performance.

As a further limitation of search-based techniques, if the circuit contains an undetectable fault, the program may exhaustively try all possible input combinations until it determines that the fault is undetectable, expending enormous effort on fruitless searches. For gate-level combinational circuits, undetectable faults occur only if the circuit contains redundant logic and hence can be minimized by careful logic design. At the switch level, however, undetectable faults may arise due to intermediate voltages or undefined initial states. Hence, a switch-level test generator must handle undetectable faults efficiently.

As an alternative, we propose *symbolic fault simulation.* This approach eliminates the need for combinatorial search. Instead, three sets of Boolean variables are introduced to encode many combinations of input, fault, and initial state. The circuit is then simulated, where the node states are given by Ordered Binary Decision Diagrams (OBDD's) [2] representing Boolean functions over the variables. These node states encode the behaviors of the good and many faulty circuits for many possible input sequences and for many possible initial states. From this information, a set of test patterns can be obtained by deriving a Boolean function representing the difference between the good and faulty circuits. This *test* function is computed by first exclusive-OR'ing the functions for each good and faulty circuit output and then OR'ing the functions for the different outputs. The test set is then derived by calculating a set of input sequences satisfying the test function for all possible combinations of fault and initial state.

Since the interface to our program is based on simulation, the user remains in control of the overall testing strategy. That is, he or she determines how many patterns to simulate and hence the length of each test. Furthermore, by simulating patterns with some inputs (e.g., control signals) set to constants, the user can, in effect, provide guidance to

*This research was supported in part by the Defense Advanced Research Projects Agency, ARPA Order Number 4976, and in part by the Semiconductor Research Corporation under contract 88-DC-068.

the program on how to transfer data from the primary inputs into internal nodes of the circuit, or from internal nodes to observation points. This results in a reasonable division of labor, with the user providing an overall testing strategy, and the program determining the detailed patterns to detect the faults.

This paper describes the prototype implementation of a switch-level test generator based on symbolic fault simulation. The program was created by extending the COSMOS switch-level simulator [3] with efficient symbolic Boolean manipulation based on OBDD's. We have successfully generated tests for sequential MOS circuits containing up to 770 transistors. This is more than twice the size of any previously reported automatic test generation at the switch level.

2. Symbolic Simulation

Conventional simulators model the functionality of the circuit only for the particular data given by the user. A large amount of input data is usually required to fully exercise a circuit. The complexity grows exponentially in the number of inputs. Symbolic simulation reduces the number of patterns simulated by evaluating a circuit for many input combinations simultaneously. A symbolic simulator resembles a conventional simulator, except that the input sequence can contain input variables in addition to the constants 1 and 0. During simulation, the circuit node states are computed as Boolean functions of the past and present input variables. These Boolean functions describe the behavior of the circuit for the set of all possible data represented by the variables. A symbolic simulator can also represent the behavior of a sequential circuit by computing the sequence of Boolean functions that would appear at each output as a function of the sequences of variables that have been applied to each input.

A circuit can be viewed as a finite state machine. Consider a circuit having n primary inputs $\vec{x} = \langle x_1, \ldots, x_n \rangle$, m primary outputs $\vec{C} = \langle C_1, \ldots, C_m \rangle$, and p state variables $\vec{S} = \langle S_1, \ldots, S_p \rangle$. At time t (measured in clock cycles), if we apply input variables $\vec{x}^t = \langle x_1^t, \ldots, x_n^t \rangle$ to the primary inputs, this circuit is represented by the following two equations:

$$\begin{aligned} \vec{S}^t &= next(\vec{S}^{t-1}, \vec{x}^t) \qquad (1) \\ \vec{C}^t &= out(\vec{S}^t) \end{aligned}$$

where $t = 1, 2, \ldots$. Note that these functions $\vec{S}^t$ and $\vec{C}^t$ should be represented as ternary functions to cover the ternary behavior of switch-level networks. Initially, all elements of $\vec{S}$ equal X, an uninitialized state:

$$\vec{S}^0 = \langle X, \ldots, X \rangle$$

According to the above equations, the present states are determined by the previous states and the present input variables. The present outputs are directly derived from the present states. Hence, $\vec{C}^t$ and $\vec{S}^t$ are ternary functions of the past and present input variables.

To cast the switch-level model in terms of Boolean operations, a logic value $y \in \{0, 1, X\}$ is represented by a "dual rail" Boolean encoding, $y.1, y.0 \in \{0, 1\}$ as follows:

y	$y.1$	$y.0$
1	1	0
0	0	1
X	1	1

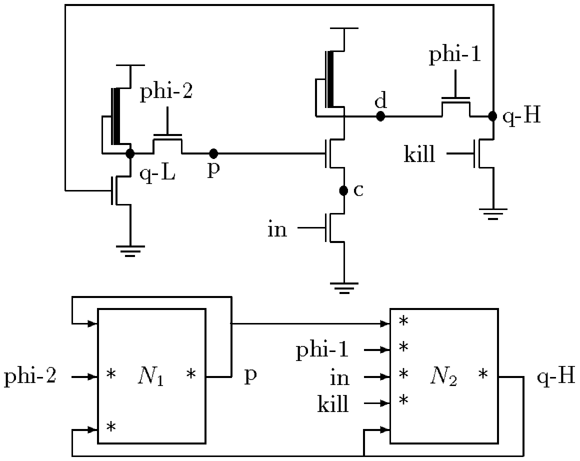


Figure 1: Network Partitioning and Fault Model

To extend this encoding to symbolic simulation, each circuit node state is represented by a pair of Boolean functions, represented as OBDD's. In this way we can operate in the Boolean domain instead of the ternary domain. The symbolic simulator derives two Boolean functions $C.1_i^t$ and $C.0_i^t$, for each output i. Equation 1 is expressed in these terms as:

$$\begin{aligned} \vec{S.1}^t &= next.1(\vec{S.1}^{t-1}, \vec{S.0}^{t-1}, \vec{x}^t) \qquad (2) \\ \vec{S.0}^t &= next.0(\vec{S.1}^{t-1}, \vec{S.0}^{t-1}, \vec{x}^t) \\ \vec{C.1}^t &= out.1(\vec{S.1}^t, \vec{S.0}^t) \\ \vec{C.0}^t &= out.0(\vec{S.1}^t, \vec{S.0}^t) \end{aligned}$$

where $t = 1, 2, \ldots$. Since the X state is represented by the encoding 1, 1, the initial values of $\vec{S.1}^t$ and $\vec{S.0}^t$ are

$$\vec{S.1}^0 = \vec{S.0}^0 = \langle 1, \ldots, 1 \rangle$$

The preprocessing methods used by the COSMOS switch-level simulator [4] make it possible to simulate MOS circuits symbolically. The COSMOS preprocessor applies symbolic analysis to convert the switch-level representation of a circuit into a Boolean representation. This analysis captures all details of the switch-level model and works for arbitrary networks. The resulting Boolean description is converted into a set of C language modeling procedures. By using different macro expansions, these procedures can implement the Boolean operations using machine-level logic instructions (for conventional simulation) or by calls to OBDD manipulation routines (for symbolic simulation).

3. Fault Model

Our preprocessor accepts as input the switch-level representation of a MOS circuit, and partitions the network into channel-connected subnetworks [4]. Each subnetwork consists of a set of storage nodes connected by transistor sources and drains. This partitioning describes the static connections in the network, i.e., those independent of transistor state. The user can inject stuck-at-1 and stuck-at-0 faults on the subnetwork inputs and outputs. Single faults are injected by default, but multiple faults are also allowed.

Figure 1 shows an example of network partitioning and the fault model. This circuit partitions into 2 subnetworks: N_1 and N_2. The local feedback shown for each subnetwork represents the nodes that may store charge dynamically.

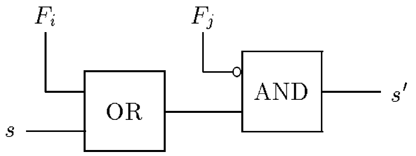


Figure 2: Injection of Stuck-at Faults on Node n with State s

The global feedback represents cyclic connections. Stuck-at faults can be injected at locations marked with asterisks (*). Faults cannot be injected on the subnetwork inputs formed by local feedback, since these are only implicit inputs. Observe that nodes q-L, c, and d are not shown in the subnetwork structure. These nodes are eliminated during preprocessing since they serve only as interconnection points within a subnetwork, i.e., they neither control any transistors nor form part of the circuit's dynamic memory. Hence, no faults can be injected on these nodes. Two distinct faults can be injected for node q-H. One is on the output of subnetwork N_2, while the other is on an input to subnetwork N_1. Assuming q-H is the only primary output of the circuit, the effect of a fault on node q-H in N_2 can be observed directly. The effect of a fault on the fanout of node q-H to N_1 can be observed only if it propagates through both subnetworks.

Our fault model is chosen as a compromise between modeling detail and efficiency. By injecting faults only at subnetwork boundaries, we avoid the need to symbolically analyze circuits containing fault effects. Furthermore, experience has shown that many detailed faults such as stuck-closed transistors cannot be reliably modeled using switch-level models. That is, the fault causes the simulator to set a node to X, to represent a potentially intermediate voltage. The simulator cannot determine whether the faulty circuit would ever produce the opposite digital (non-X) value from the good circuit on some primary output. Nonetheless, our model generates patterns that exercise every signal connection in the circuit and propagates the effects to the primary outputs. We anticipate that such patterns will reliably detect most chip defects.

4. Fault Injection

Suppose we wish to evaluate N different faults numbered from 0 to $N-1$. Figure 2 shows the idea of how we inject stuck-at faults on a line with good value s concurrently and symbolically, where fault i causes this line to be stuck at 1, and fault j causes this line to be stuck at 0. By introducing two Boolean *fault injection signals*, F_i and F_j, a "qualified" line value s' is created. Note that the OR and AND gates in Figure 2 are just conceptual representations of how faults can be injected in an algorithmic way. We do not actually introduce any physical gates or transistors into the circuit. Setting one of the fault injection signals to 1 causes the corresponding fault to appear. Setting both to 0 makes the line behave as in the good circuit. This idea is extended to cover the ternary behavior of MOS circuits as follows:

$$\begin{aligned} s'.1 &= (s.1 \vee F_i) \wedge \overline{F_j} \\ s'.0 &= (s.0 \vee F_j) \wedge \overline{F_i} \end{aligned}$$

To inject a multiple stuck-at fault, the values representing all of the faulted lines are qualified by a single fault injection signal.

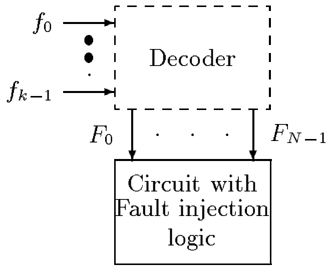


Figure 3: Control of Fault Injection with Fault Variables

Circuit	f_1	f_0	description	*out*
0	0	0	in_1 s.a.0	1
1	0	1	in_2 s.a.1	$\overline{a_1}$
2	1	0	*out* s.a.1	1
3	1	1	no fault	$\overline{a_1 \wedge a_2}$

Figure 4: NAND Circuit with 3 Faults

A circuit with N faults requires N fault injection signals. Since only one fault injection signal is active for a given faulty circuit, only $N+1$ combinations out of 2^N will be used. In other words, if each fault injection signal were represented by an independent variable, symbolic simulation would effectively evaluate all possible multiple faults. We only want to evaluate single faults plus the good circuit. To control the injection of faults, we generate the fault injection signals in a manner analogous to the outputs of a decoder, as shown in Figure 3. Again, the decoder is only a conceptual representation. We do not model any extraneous circuitry. A set of $k = \lceil log(N+1) \rceil$ Boolean *fault variables*, $\vec{f} = \langle f_{k-1}, \ldots, f_0 \rangle$ is introduced to control fault injection. Each fault injection signal F_i is then given by the OBDD for the function having value 1 when the fault variables are assigned values corresponding to the binary representation of i. No fault injection signal is activated when the fault variables all equal 1, and hence we model the good circuit as well. By symbolically simulating the circuit as a function of these fault variables, we effectively compute the behavior of the good and all faulty circuits concurrently. Due to the sharing of common subgraphs within OBDD's, this approach exploits the commonality between all good and faulty circuit functions. In this sense, our approach improves on conventional concurrent fault simulation, which exploits only the commonality between each faulty circuit function and the good circuit function.

5. Symbolic Fault Simulation

Symbolic fault simulation combines symbolic logic simulation with the fault injection described in Section 4. The circuit model for symbolic fault simulation is an extension of the symbolic simulation model described earlier. A circuit contains n primary inputs $\vec{x} = \langle x_1, \ldots, x_n \rangle$, m primary outputs $\vec{C} = \langle C_1, \ldots, C_m \rangle$, p state variables $\vec{S} = \langle S_1, \ldots, S_p \rangle$, and k fault variables $\vec{f} = \langle f_{k-1}, \ldots, f_0 \rangle$. The symbolic fault simulator derives two circuit output functions , $C.1_i^t$ and $C.0_i^t$, for each output i in a similar manner to Equation 2, with the extension that the functions depend on the fault variables $\vec{f}$.

As an example of symbolic fault simulation, consider a two-input NAND circuit having inputs in_1 an in_2 and output *out*. Suppose we wish to generate tests to detect stuck-

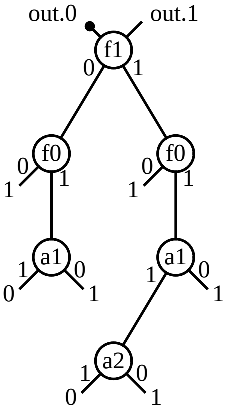


Figure 5: OBDD for output functions $out.1$ and $out.0$

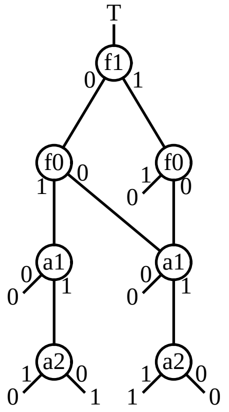


Figure 6: OBDD for test function T for circuit of Figure 7

at-1 faults on in_2 and out, and a stuck-at-0 fault on in_1. To model the 3 faulty plus the good circuit, we introduce 2 fault variables: f_1 and f_0 as listed in Figure 4. To simulate this circuit, we set in_1 to Boolean variable a_1 and in_2 to variable a_2. The simulator then computes two functions for the output: $out.1$ and $out.0$. Figure 5 shows the OBDD representation of these functions. Function $out.1$ represents the circuit output, while $out.0$ is its complement (denoted by the solid dot on the arc pointing to the root). In general the functions describing the state of a node will be complements of one another, unless some assignment to the variables could produce node state X. Figure 4 shows Boolean expressions corresponding to the output functions for the 4 circuits simulated.

6. Test Pattern Generation

Given a set of functions $\vec{C.1}^t$ and $\vec{C.0}^t$ representing the good and faulty circuit behaviors, we derive a set of test patterns to detect the faults. The good circuit function for each output i is obtained by restricting the fault variables to 1's:

$$\begin{aligned} G.1_i^t &= C.1_i^t\big|_{\vec{f} = \langle 1,\ldots,1\rangle} \\ G.0_i^t &= C.0_i^t\big|_{\vec{f} = \langle 1,\ldots,1\rangle} \end{aligned}$$

As is illustrated by Figure 5, computing this restriction is a simple matter of following the path from the OBDD root through the fault variable vertices, always following arcs labeled by 1. For each output i, we then derive a function H_i^t indicating the conditions under which the faulty circuits produce the opposite digital (non-X) value from the good circuit:

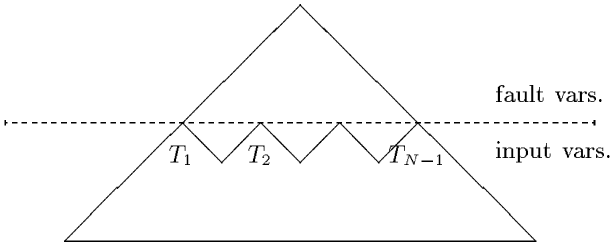
$$H_i^t \;=\; \left(\overline{G.1_i^t} \wedge \overline{C.0_i^t}\right) \;\vee\; \left(\overline{G.0_i^t} \wedge \overline{C.1_i^t}\right)$$

This equation exploits the property that the two ternary signals x and y are opposite only if either $x.1 = y.0 = 0$, or $x.0 = y.1 = 0$.

Next, we derive a test function T^t, describing all ways of testing every fault with a t cycle test. No fault can be detected for $t = 0$, and hence $T^0 = 0$. For other values of t, a fault can be detected if it has been detected earlier, or if one of the faulty circuit outputs at time t is different from the corresponding good circuit output:

$$T^t \;=\; T^{t-1} \;\vee\; \bigvee_{1 \leq i \leq m} H_i^t$$

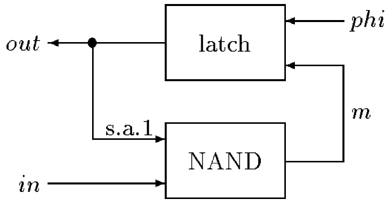
By ordering the fault variables before the input variables, the topology of the OBDD representing T^t has the following structure:



We derive a test function for faulty circuit i by traversing the path from the root through the fault variable vertices following the labels corresponding to the binary representation of i. Each of the resulting functions T_i denotes the set of all tests for fault i. That is, any assignment to the variables that causes T_i to evaluate to 1 defines a test for fault i. If the fault cannot be detected, the corresponding test function will equal the constant function 0. Hence, undetectable faults are easily identified. To generate a complete set of tests, we attempt to maximize the number of faults covered by each test by intersecting the test functions for a number of faults (using the AND operation). For example, the function $T_i \wedge T_j$ denotes the set of all input sequences that test both faults i and j.

Returning to the example of Figure 4, the OBDD for the test function T^1, derived from $out.1$ and $out.0$, is shown in Figure 6. This graph contains the test functions of all faulty circuits, i.e., $T_0 = a_1 \wedge a_2$, $T_1 = a_1 \wedge \overline{a_2}$, and $T_2 = a_1 \wedge a_2$. The test set is then derived by calculating a set assignments to the variables satisfying the three test functions. In this case, the test set is uniquely determined as $\{(in_1 = 1, in_2 = 1),\ (in_1 = 1, in_2 = 0)\}$. For larger circuits, the program has more flexibility in selecting tests and can achieve higher test compaction.

As an example of sequential test generation, consider the circuit in Figure 7 with a stuck-at-1 fault on one input of the NAND gate. Assume that the Boolean functions representing the good circuit output functions at time t are denoted $G.1^t$ and $G.0^t$, while the faulty circuit output functions at time t are denoted $F.1^t$ and $F.0^t$. For $t = 0$, these functions all equal the constant value 1 denoting state X. The initial



Function	Cycle t			
	0	1	2	3
$in.1$	–	a_1	a_2	a_3
$in.0$	–	$\overline{a_1}$	$\overline{a_2}$	$\overline{a_3}$
$G.1^t$	1	1	$a_1 \vee \overline{a_2}$	$a_2 \vee \overline{a_3}$
$G.0^t$	1	a_1	a_2	$(a_1 \vee \overline{a_2}) \wedge a_3$
$F.1^t$	1	$\overline{a_1}$	$\overline{a_2}$	$\overline{a_3}$
$F.0^t$	1	a_1	a_2	a_3
T^t	0	0	0	$\overline{a_1} \wedge a_2 \wedge a_3$

Figure 7: Sequential Test Generation Example

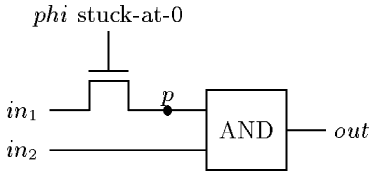


Figure 8: Example Circuit for Initialization Problem

test function T^0 equals the constant 0. The NAND gate output is updated as follows:

$$\begin{aligned} m.1 &\leftarrow in.0 \vee out.0 \\ m.0 &\leftarrow in.1 \wedge out.1 \end{aligned}$$

The above two equations cover the ternary behavior of NAND circuit. To detect this fault, we introduce an input variable a_1 and run one clock cycle. As indicated in Figure 7, T^1 is computed as 0, indicating that this fault is not yet detectable. We introduce a second input variable a_2 and simulate a second cycle. T^2 also equals 0, and hence we introduce a third variable a_3 and simulate one more cycle. Finally, T^3 is computed as $\overline{a_1} \wedge a_2 \wedge a_3$, denoting a test with $a_1 = 0$, $a_2 = 1$, and $a_3 = 1$. The test sequence is then derived by repeating the simulation sequence with these constant values substituted for the variables, as indicated by the following table:

	cycle 1	cycle 2	cycle 3
in	0	1	1
out_g	1	0	1
out_f	1	0	0

The resulting test sequence detects the fault in 3 cycles. Setting in to 0 for the first cycle initializes the circuit, causing a 1 on out for both the good and faulty circuits. The second input pattern excites the fault by setting out to 0. The third pattern makes this fault effect observable by sensitizing a path to out.

7. Initialization Variables

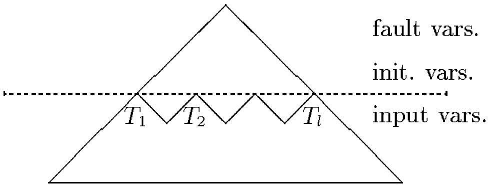
In switch-level simulation, all circuit node states are normally set to X at the beginning of simulation. This causes some faults to be declared undetectable. Consider a simple AND gate and pass transistor, as shown in Figure 8 with the pass transistor gate *phi* stuck at 0. We consider a fault truly detected only if the faulty circuit produces the opposite digital value from the good circuit on some primary output. The state of node p in the faulty circuit remains at X, because it cannot be controlled by the inputs. Using the algorithm described thus far, there is no way to generate a test for this fault.

As a compromise, most fault simulators and test generators classify a fault as "soft" or "possibly" detected if the good circuit output toggles between 0 and 1 while the faulty circuit output remains at X. This compromise is not very satisfactory. The user has no way of determining the true coverage of a set of test patterns when many faults are labeled as only possibly detected.

In an actual circuit, we may reasonably assume that each node will initialize to either 0 or 1, but in an unpredictable way. We can exploit this property by introducing a set of *initialization* variables to represent the arbitrary initial state of the circuit. The user can specify that some nodes should be set to initialization variables, rather than to X. The symbolic fault simulator then derives a test function in terms of the past and present input variables, the fault variables, and the initialization variables. The test set is then derived to cover all combinations of fault and initial state.

Returning to the example in Figure 8, we introduce an initialization variable y, to represent the initial state of node p. Let us assume that nodes in_1 and in_2 are set to input variables a_1 and a_2 respectively, and the clock node *phi* is set to 1. Then, the state of output node *out* will be $(a_1 \wedge a_2)$ for the good circuit and $(y \wedge a_2)$ for the faulty circuit. We then derive two test functions $a_1 \wedge a_2$ and $\overline{a_1} \wedge a_2$ to cover both possible initial states. By testing the circuit with the patterns $(in_1 = 1, in_2 = 1)$ and $(in_1 = 0, in_2 = 1)$, we can reliably detect the fault. Other programs would say these patterns possibly detect the fault, but our program can guarantee fault coverage.

The test function T^t is derived in the same way as that described in Section 6, except that initialization variables are also incorporated into this function. The topology of the OBDD representing T^t has the following structure:



Each vertex lying below the dotted line with a parent above the line represents an equivalence class of fault and initial state. That is, suppose two paths lead to one such vertex, where one path denotes fault i and initial state $\vec{y}$ while the other denotes fault i' and initial state $\vec{y}'$. Then these two fault and initial state combinations are detected by exactly the same set of tests. Thus, we can consider each such vertex to be the root of the OBDD for a test function T_j. By creating a test set that for every j contains at least one input assignment satisfying T_j, we are guaranteed to test for all combinations of fault and initial state. In the worst case, a circuit with v initialization variables and N faults could have $N \cdot 2^v$ test functions. In practice, the number is far smaller. We generate a compacted test set as before by intersecting the test functions as much as possible.

primary inputs	14
primary outputs	8
transistors	240
faults injected	266
faults detected	266
test patterns generated	12
CPU time (on VAX 8800)	69 sec.
memory required	4.5 MB

Table 1: Results of Test Pattern Generation for 74181 4-Bit ALU

primary inputs	7
primary outputs	5
transistors	206
faults injected	302
faults detected	274
undetectable faults	28
test patterns generated	24
CPU time (on VAX 8800)	90 sec.
memory required	3.7 MB

Table 2: Results of Test Pattern Generation for 4-Bit Accumulator

8. Testing Strategy

The computational complexity of our test generation program grows with the number of faults, the number of initialization variables, and the number of input variables. As the circuit size increases, these factors also increase. Fortunately, the test generator can operate over a spectrum of generality. By simulating patterns of constants, the program operates as a fault simulator, evaluating the coverage of the patterns. In fact, our ability to model arbitrary initial state provides a feature not found in any other fault simulator. By simulating patterns containing only variables, the program operates as a fully automatic test pattern generator. For larger circuits, the most desirable approach is a hybrid of the two, in which the patterns contain both constants and variables. For example, we can start by generating tests for easily detected faults using patterns containing mostly constants. Full symbolic simulation is applied only after the fault set has been reduced to the difficult ones. Alternatively, we can generate patterns for one block of the circuit at a time. First, a series of patterns is simulated to transfer symbolic data to the inputs of the block. The block is then exercised, and a series of patterns is simulated to transfer the block outputs to the primary outputs. In this manner, the user guides the program on an overall test strategy, while the program determines the detailed data values needed to test the block. A symbolic simulator provides a natural interface for specifying such test strategies. It gives the user total control over the nodes to set to initialization variables, the number of cycles to simulate, and the constants and variables to apply to the inputs.

9. Experimental Results

We have tested our program on a variety of combinational and sequential MOS circuits. In the following discussion, all measurements were taken on a Digital Equipment Corporation VAX 8800, an 8 MIP mainframe computer.

As a combinational benchmark circuit, we selected the 74181 4-bit Arithmetic Logic Unit (ALU) circuit. We implemented this circuit using 240 CMOS transistors. Table 1 summarizes the results of this experiment. We injected stuck-at-1 and stuck-at-0 faults on all subnetwork inputs and outputs. All the faults are detected with 12 test patterns. Akers and Krishnamurthy [1] have shown that 12 test patterns are the minimum required for a complete single stuck-at fault test set of this circuit. This demonstrates our success at test compaction.

As a sequential benchmark circuit, we selected a 4-bit CMOS accumulator circuit consisting of a 4-bit adder, latch, and overflow logic. This circuit contains a variety of structures unique to MOS, including transmission gates, dynamic latches, and a precharged Manchester carry chain. There would be no way to construct a gate-level equivalent. We introduced 5 initialization variables to set the internal nodes within the latches. Since this circuit is sequential, we need to run at least two clock cycles. We could generate tests for this circuit by introducing an input variable for each data input at each time point until maximum fault coverage is obtained. However, this approach requires too much computation. Instead, we used a hybrid approach applying restricted patterns first, and then more general patterns. As a restricted pattern, we set each data input to the same input variable, thereby evaluating the patterns 0000 and 1111. By running two clock cycles with this type of pattern, we obtained 75% fault coverage. To detect the remaining faults, we introduced a new input variable on each data input for two more cycles. The 28 remaining faults were determined to be inherently undetectable: 6 lie within logically redundant circuitry, while 22 cause one side of a transmission gate to remain off. The program then generated a set of 6 tests, each 4 cycles long. Table 2 summarizes the results.

We also generated tests a 16-bit version of this accumulator (770 transistors) following a similar testing strategy. For this circuit, the program required 3390 CPU seconds and 15 MB of virtual memory. This is the biggest circuit we have tried to date.

10. Conclusion

Our prototype program demonstrates that symbolic fault simulation is a promising approach to test generation. It can handle situations that other test generation programs find intractable or impossible, such as:

- Arbitrary MOS circuits
- Sequential circuits
- Undetectable faults
- Faults that prevent proper circuit initialization

Furthermore, since the program provides a continuous range between fault simulation and fully automatic test generation, the user has close control over the strategy used in generating tests.

We have successfully generated tests for a number of nontrivial benchmark circuits, including ones larger than any other switch-level test pattern generator has been able to handle. Much more work remains, however, to make this tool usable for genuine VLSI circuits. The performance of our program is currently limited by the high memory requirement needed to store the large OBDD's created. These graphs are accessed in a highly random fashion causing the computer system to thrash badly once we exceed the physical memory of the machine. We plan to investigate several ways to improve the performance of the program. Major improvements seem possible.

References

[1] S. B. Akers and B. Krishnamurthy, "On the Application of Test Counting to VLSI Testing," *1985 Chapel Hill Conference on Very Large Scale Integration*, Computer Science Press, 1985, pp. 343–360.

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

[3] R. E. Bryant, *et al*, "COSMOS: A Compiled Simulator for MOS Circuits," *24th Design Automation Conference*, 1987, pp. 9–16.

[4] R. E. Bryant. "Boolean Analysis of MOS Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, Vol. CAD-6, No. 4 (July, 1987), pp. 634–649.

[5] H. H. Chen, R. G. Mathews, and J. A. Newkirk, "An Algorithm to Generate Tests for MOS Circuits at the Switch Level," *International Test Conference*, 1986, pp. 304–312.

[6] R. I. Damper and N. Burgess. "MOS Test Pattern Generation Using Path Algebras," *IEEE Transactions on Computers*, Vol. C-36, No. 9 (Sept., 1987), pp. 1123–1128.

[7] P. Goel. "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, Vol. C-30, No. 3 (March, 1981), pp. 215–222.

[8] S. K. Jain and V. D. Agrawal. "Modeling and Test Generation Algorithms for MOS Circuits," *IEEE Transactions on Computers*, Vol. C-34, No. 5 (May, 1985), pp. 426–433.

[9] M. K. Reddy, S. M. Reddy, and P. Agrawal. "Transistor Level Test Generation for MOS Circuits," *22nd Design Automation Conference*, 1985, pp. 825–828.

[10] S. H. Robinson and J. P. Shen. "Towards a Switch-Level Test Pattern Generation Program," *International Conference on Computer-Aided Design*, 1985, pp. 39–41.

[11] J. P. Roth, *Computer Logic, Testing, and Verification*, Computer Science Press, 1980.