Bit-Fixing in Pseudorandom Sequences for Scan BIST

Nur A. Touba, *Member, IEEE,* and Edward J. McCluskey, *Life Fellow, IEEE*

***Abstract*—A low-overhead scheme for achieving complete (100%) fault coverage during built-in self test of circuits with scan is presented. It does not require modifying the function logic and does not degrade system performance (beyond using scan). Deterministic test cubes that detect the random-pattern-resistant (r.p.r.) faults are embedded in a pseudorandom sequence of bits generated by a linear feedback shift register (LFSR). This is accomplished by altering the pseudorandom sequence by adding logic at the LFSR's serial output to "fix" certain bits. A procedure for synthesizing the bit-fixing logic for embedding the test cubes is described. Experimental results indicate that complete fault coverage can be obtained with low hardware overhead. Further reduction in overhead is possible by using a special correlating automatic test pattern generation procedure that is described for finding test cubes for the r.p.r. faults in a way that maximizes bitwise correlation.**

***Index Terms*—Design for testability, digital system testing, logic circuit testing, self-testing, sequences.**

I. Introduction

As THE density and complexity of integrated circuits continue to increase enabling whole systems to be integrated on a single chip, economical built-in self-test (BIST) approaches are needed. BIST involves using on-chip hardware to apply test patterns to the circuit under test and to analyze its output response. BIST techniques reduce the burden on external automatic test equipment (ATE) and allow concurrent testing of multiple modules.

A low-overhead approach for BIST in circuits with scan is to use a linear feedback shift register (LFSR) to shift a pseudorandom sequence of bits into the scan chain. When a pattern has been shifted into the scan chain, it is applied to the circuit under test and the response is loaded back into the scan chain and shifted out into a serial signature register for compaction as the next pattern is shifted into the scan chain. Fig. 1 shows a block diagram for this "test-per-scan" BIST scheme. Unfortunately, many circuits contain random-pattern-resistant (r.p.r.) faults [1], which limit the fault coverage that can be achieved with this approach.

One method for improving the fault coverage for a test-per-scan BIST scheme is to modify the circuit under test by either inserting test points [2]–[4] or by redesigning it [5]–[8] to improve the fault detection probabilities. The drawback of these techniques is that they generally add extra levels of logic to the circuit that can degrade system performance. Moreover, in some cases, it is not possible or not desirable to modify the function logic (e.g., macrocells, cores, legacy designs).

Another method for improving the fault coverage is to use a weighted pseudorandom sequence. Logic is added to weight the probability of each bit in the sequence being a "1" or a "0" in a way that biases the patterns that are generated toward those that detect the r.p.r. faults. The weight logic can be placed either at the input of the scan chain [9] or in the individual scan cells themselves [10]–[12]. Multiple weight sets are usually required due to conflicting input values needed to detect r.p.r. faults [13]. The weight sets need to be stored on chip and control logic is required to switch between them, so the hardware overhead can be large.

A third method to improve the fault coverage is to use a "mixed-mode" approach where deterministic patterns are used to detect the faults that the pseudorandom patterns miss. Storing deterministic patterns in a read-only memory (ROM) requires a large amount of hardware overhead. Koenemann [14] proposed a technique based on reseeding an LFSR that reduces the storage requirements. The LFSR that is used for generating the pseudorandom patterns is also used to generate deterministic *test cubes* (test patterns with unspecified inputs) by loading it with a computed seed. The number of bits that need to be stored is reduced by storing a set of seeds instead of a set of deterministic patterns. Hellebrand *et al.* [15]–[17] proposed an improved technique that uses a multiple-polynomial LFSR for encoding a set of deterministic test cubes. By "merging" and "concatenating" the test cubes, they further reduce the number of bits that need to be stored. Even further reduction can be achieved by using variable-length seeds [18] and a special ATPG algorithm [19]. More recently, techniques for generating the deterministic test cubes using BIST control logic have been studied [20]–[22]. Also, techniques for using special functional hardware (e.g., processors) when possible to generate deterministic test cubes have been investigated [23], [24].

This paper presents a mixed-mode approach in which deterministic test cubes are embedded in the pseudorandom sequence of bits itself (preliminary results were presented in [25] and described in U.S. Patent #6 061 818). Logic is added at the serial

Manuscript received November 14, 1999; revised June 17, 2000. This work was supported in part by the Ballistic Missile Defense Organization Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy Office of Naval Research under Grant N00014-92-J-1782, by the National Science Foundation under Grant MIP-9107760, and by the Advanced Research Projects Agency under Prime Contract DABT63-94-C-0045. This paper was recommended by Associate Editor K.-T. Cheng.

N. A. Touba was with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305 USA. He is now with the Computer Engineering Research Center, Department of Electrical and Computer Engineering, University of Texas, Austin, TX 78712 USA (e-mail: touba@ece.utexas.edu).

E. J. McCluskey is with the Center for Reliable Computing, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305 USA.

Publisher Item Identifier S 0278-0070(01)01944-3.

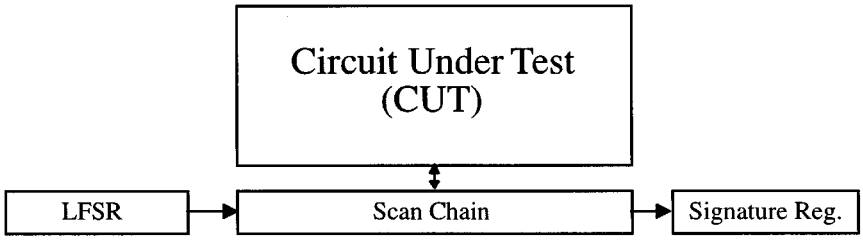


Fig. 1. Block diagram for a "test-per-scan" BIST scheme.

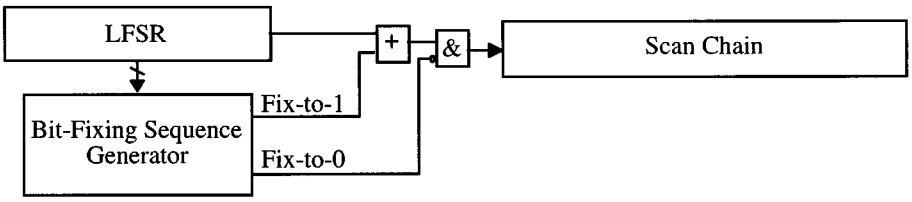


Fig. 2. Logic for altering the pseudorandom bit sequence.

output of the LFSR to selectively alter the pseudorandom bit sequence so that it will contain patterns that detect the r.p.r. faults. This is accomplished by "fixing" certain bits in the sequence. As illustrated in Fig. 2, logic is added to generate a bit-fixing sequence that alters the pseudorandom sequence by causing certain bits to be fixed to either a "1" or a "0." A procedure is described for designing the bit-fixing sequence generator in a way that minimizes area overhead. The procedure can embed any set of test cubes for the r.p.r. faults; however, the more correlated the test cubes for the r.p.r. faults are, the less the overhead is. A special correlating ATPG procedure is presented for finding test cubes for the r.p.r. faults that can be very efficiently encoded (preliminary results were presented in [26]).

The test-per-scan BIST scheme presented in this paper is sort of a hybrid approach. It is different from weighted pattern testing because it is not based on probability. It guarantees that certain test cubes will be applied to the circuit under test during a specified test length. Also, it does not require a multiphase test in which control logic is needed to switch to different weight sets for each phase. The control is very simple because there is only one phase.

In the proposed scheme, no data is stored in a ROM, but rather a multilevel circuit is used to dynamically fix bits in a way that exploits bit correlation (same specified values in particular bit positions) among the test cubes for the r.p.r. faults. Small numbers of correlated bits are fixed in selected pseudorandom patterns to make the pseudorandom patterns match the test cubes. So rather than trying to compress the test cubes themselves, the proposed scheme essentially compresses the bit differences between the test cubes and a selected set of pseudorandom patterns. Since there are so many pseudorandom patterns to choose from, a significant amount of compression can be achieved resulting in reduced overhead.

The approach described here and the "bit-flipping" approach presented by Kiefer and Wunderlich [20]–[22] share some similar characteristics in that both alter the serial sequence generated by an LFSR. However, the procedures for designing the sequence altering logic differ greatly. Kiefer and Wunderlich target a program logic array (PLA) implementation and use ESPRESSO-like procedures to minimize the number of minterms [22]. The scheme described here targets a sequential multilevel logic implementation and inherently factors and decomposes the sequence altering logic by construction to minimize the overhead. A special ATPG procedure is also described here for reducing overhead by maximizing bitwise correlation in the test cubes for the r.p.r. faults.

Schemes based on reseeding an LFSR require that the LFSR have at least as many stages as the maximum number of specified bits in any test cube. A hardware tradeoff that is possible in the scheme presented in this paper is that a smaller LFSR can be used for generating the pseudorandom bit sequence. This may cause some faults to not be detected because of linear dependencies in the patterns that are generated, but deterministic test cubes for those faults can be embedded at the expense of additional logic in the bit-fixing sequence generator. Data is presented showing how much additional logic is required for different size LFSRs.

The paper is organized as follows. In Section II, the architecture of the bit-fixing sequence generator is described. In Section III, the procedure for designing the bit-fixing sequence generator is presented. In Section IV, the special correlating ATPG procedure for maximizing bitwise correlation is described. In Section V, experimental results are shown for benchmark circuits. Section VI is the conclusion.

II. Architecture of Bit-Fixing Sequence Generator

The purpose of the bit-fixing sequence generator is to alter the pseudorandom sequence of bits that is shifted into the scan chain in order to embed deterministic test cubes in the sequence. This is done by generating a sequence of *fix-to-1* and *fix-to-0* control signals that fix certain bits to either "1" or "0." The architecture of the bit-fixing sequence generator is shown in Fig. 3. For a scan chain of length m, there is a Mod-$(m+1)$ *Counter* that counts the number of bits that have been shifted into the scan chain. After m bits, the scan chain is full, so when the counter reaches the $(m+1)$ state, the pattern in the scan chain is applied to the circuit under test and the response is loaded back

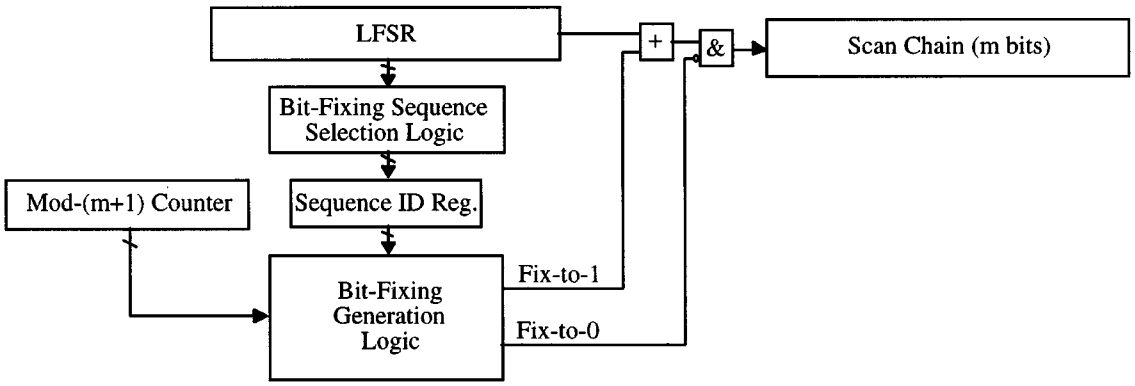


Fig. 3. Architecture of bit-fixing sequence generator.

into the scan chain. At this point, the LFSR contains the starting state for the next pattern that will be shifted into the LFSR. The *Bit-Fixing Sequence Selection Logic* decodes the starting state in the LFSR and selects the bit-fixing sequence that will be used for the next pattern. The selected bit-fixing sequence identifier is loaded into the *Sequence ID Register*. As the counter counts through the next m bits that are shifted into the scan chain, the *Bit-Fixing Sequence Generation Logic* generates the *fix-to-1* and *fix-to-0* control signals based on the bit-fixing sequence identifier stored in the *Sequence ID Register* and the value of the counter (see Fig. 7 for a specific example).

One thing that should be pointed out is that the Mod-$(m+1)$ *Counter* is not additional overhead. It is needed in the control logic for any test-per-scan BIST technique to generate a control signal to clock the circuit under test when the scan chain is full. Thus, this scheme takes advantage of existing BIST control logic.

For each pattern that is shifted into the scan chain, the bit-fixing sequence generator is capable of generating one of 2^n different bit-fixing sequences, where n is the size of the *Sequence ID Register*. A deterministic test cube for an r.p.r. fault can be shifted into the scan chain by generating an appropriate bit-fixing sequence for a pseudorandom pattern generated by the LFSR. The bit-fixing sequence fixes certain bits in the pseudorandom pattern such that the resulting pattern that is shifted into the scan chain detects the r.p.r. fault. The bit-fixing sequence generator must be designed so that it generates enough deterministic test cubes to satisfy the fault coverage requirement. The key to minimizing the area overhead for this approach is careful selection of the bit-fixing sequences that are generated.

One characteristic of the test cubes for r.p.r. faults is that subsets of them often have the same specified values in particular bit positions (this will be referred to as "bit correlation"). For example, the test cubes 11011, $11X00$, and $1X0X0$, are correlated in the first, second, and third bit positions, but not the fourth and fifth. That is because all of the specified bits in the first and second bit positions are ones and all the specified bits in the third bit position are zeros. However, the fourth and fifth bit positions have conflicts because some of the specified values are ones and some are zeros. Note that the unspecified values (Xs) do not matter. The reason why a significant amount of bit correlation often exists among the test cubes for the r.p.r. faults is probably due to the fact that several r.p.r. faults may be caused by a single random pattern resistant structure in the circuit. For example, if there is a large fan-in AND gate in a circuit, then that may cause all of the input stuck-at one faults and the output stuck-at zero fault of the gate to be r.p.r. Many of the specified values in particular bit positions of the test cubes for these r.p.r. faults will be the same. Thus, there will be a significant amount of bit correlation among the test cubes. This phenomenon is seen in weighted pattern testing, where biasing certain bit positions results in detecting a significant number of r.p.r. faults.

In the scheme presented in this paper, bit correlation among the test cubes for the r.p.r. faults is used to minimize both the number of different bit-fixing sequences that are required and the amount of decoding logic. A procedure for designing the bit-fixing sequence generator is described in Section III.

Note that while the scheme is described for a single scan chain, the extension to multiple scan chains is straightforward. Fig. 4 shows how the scheme can be applied to the STUMPS [27] architecture. Multiple *fix-to-1* and *fix-to-0* control lines are generated. The procedure for designing the bit-fixing sequence generator for multiple short scan chains is exactly the same as for one long scan chain except that the bit-fixing control lines would be distributed among the multiple scan chains.

III. Designing Bit-Fixing Sequence Generator

For a given LFSR and circuit under test, this section describes an automated procedure for designing a bit-fixing sequence generator to satisfy test length and fault coverage requirements. The bit-fixing sequence generator is designed to alter the pseudorandom bit sequence generated by the LFSR to achieve the desired fault coverage for the given test length (number of scan patterns applied to the circuit under test).

A. Obtaining Test Cubes

The first step is to simulate the r-stage LFSR for the given test length L to determine the set of pseudorandom patterns that are applied to the circuit under test. For each of the L patterns that are generated, the starting r-bit state of the LFSR is recorded (i.e., the contents of the LFSR right before shifting the first bit of the pattern into the scan chain). Fault simulation is then performed on the circuit under test for the pseudorandom patterns to see which faults are detected and which are not. The pattern that *drops* each fault from the fault

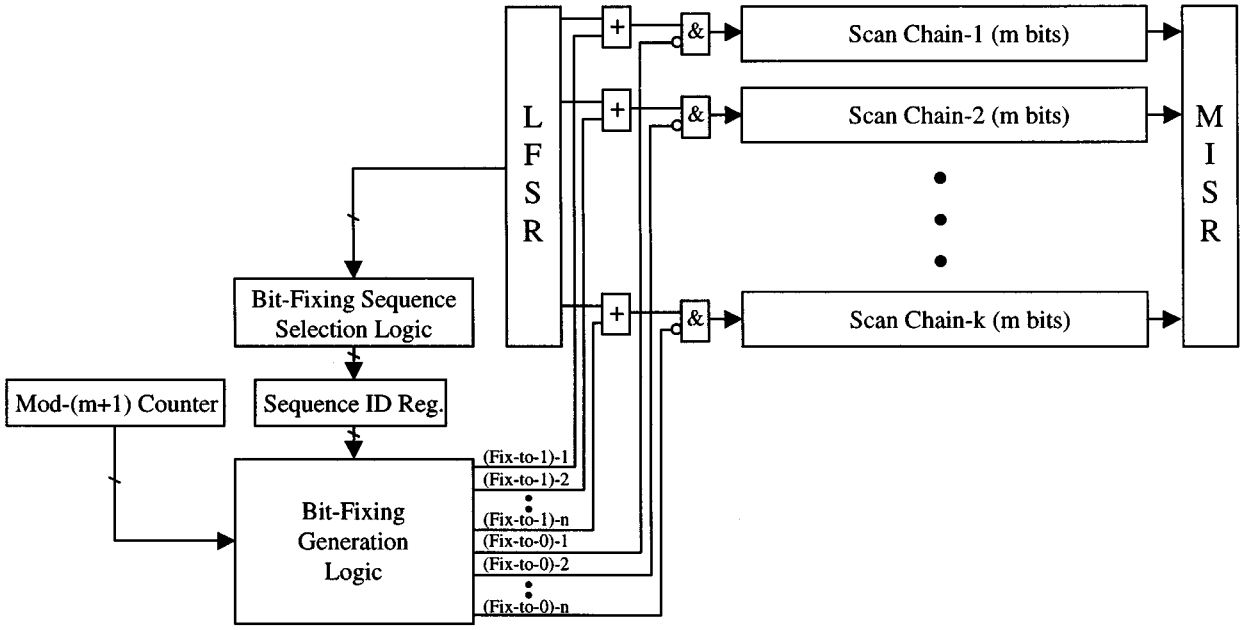
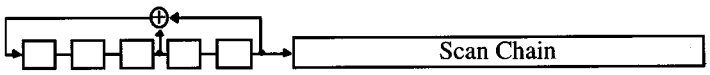


Fig. 4. Bit-fixing sequence generator for STUMPS architecture.



Starting State		Pattern Shifted into Scan Chain	Faults Dropped
0 1 0 1 1	→	0 1 0 0 0 0 1 0 1 0 1 1	7
1 1 0 1 0	→	1 1 1 1 1 0 0 1 1 0 1 0	5
1 1 0 0 0	→	0 1 0 1 1 1 0 1 1 0 0 0	2
0 0 0 0 1	→	1 1 0 1 0 0 1 0 0 0 0 1	0
1 1 1 0 0	→	1 1 0 0 0 1 1 1 1 1 0 0	1
0 1 1 1 0	→	0 0 0 0 1 0 1 0 1 1 1 0	0
0 1 0 0 1	→	1 1 1 0 0 1 1 0 1 0 0 1	0
0 0 0 1 1	→	0 1 1 1 0 1 1 0 0 0 1 1	0
0 0 1 0 1	→	0 1 0 0 1 0 0 0 0 1 0 1	0
1 0 0 1 1	→	0 0 0 1 1 1 1 1 0 0 1 1	1
1 1 0 1 1	→	0 0 1 0 1 0 1 1 1 0 1 1	0
0 0 1 0 0	→	1 0 0 1 1 0 1 0 0 1 0 0	0
			16/20

Fault Coverage = 80%

Test Cubes for Undetected Faults:

1 1 1 X 0 0 X X X X 0 0
1 0 1 X 1 0 X X X X 0 X
0 0 0 X X 1 X X X X 0 0
0 1 X X 0 1 X X X X 1 0

Fig. 5. Design example: obtaining the test cubes.

list (i.e., detects the fault for the first time) is recorded. The faults that are not detected are the faults that require altering of the pseudorandom bit sequence. The pseudorandom bit sequence must be altered to generate test cubes that detect the undetected faults. An automatic test pattern generation (ATPG) tool is used to obtain test cubes for the undetected faults by leaving the unspecified inputs as Xs.

A simple contrived design example will be used to illustrate the procedure described in this paper. A bit-fixing sequence generator will be designed to provide 100% fault coverage for a test length of 12 patterns $(L = 12)$ generated by a five-stage LFSR $(r = 5)$. Fig. 5 shows the 12 patterns that are generated by the LFSR and applied to the circuit under test through the scan chain. For each pattern, the starting state of the LFSR is shown and the number of faults that are dropped from the fault list is shown. Five of the patterns drop faults while the other seven do not. The pseudorandom patterns detect 16 out of 20 possible faults giving a fault coverage of 80%. An ATPG tool is used to obtain test cubes for the four undetected faults. The bit-fixing sequence generator must be designed so that it alters the pseudorandom bit sequence in a way that all four test cubes are generated in the scan chain.

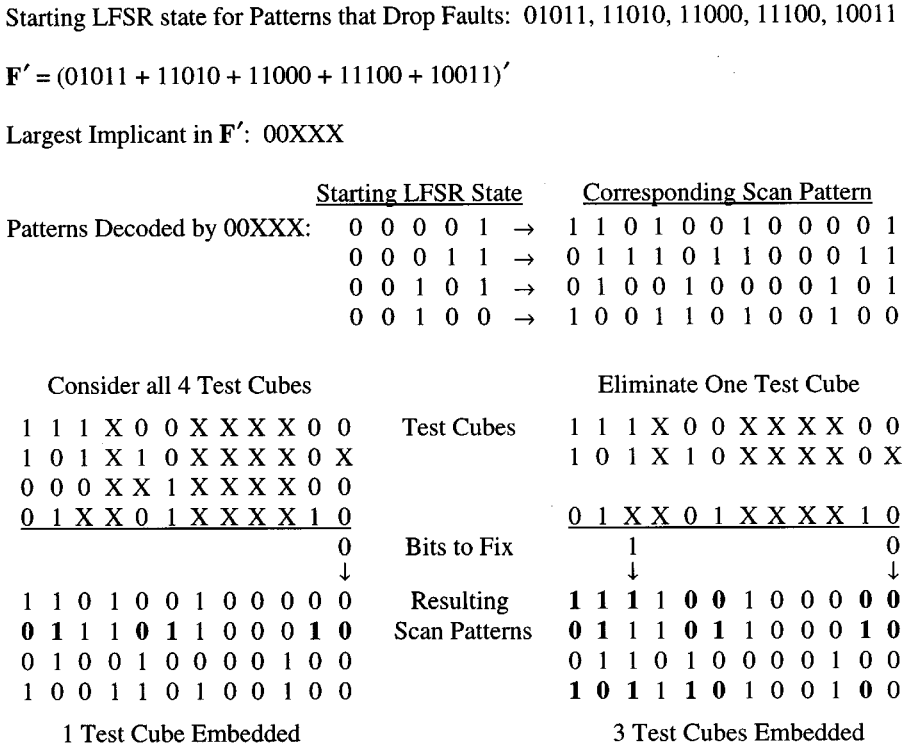


Fig. 6. Design example: finding decoding function and set of bits to fix for the new *Sequence ID Register* bit.

B. Embedding Test Cubes

Once the set of test cubes for the undetected faults has been obtained, the bit-fixing sequence generator is then designed to embed the test cubes in the pseudorandom bit sequence. The test cubes are embedded in a way that guarantees that faults that are currently detected by the pseudorandom bit sequence will remain detected after the test cubes are embedded. This is done by only altering patterns that do not drop any faults. As long as the patterns that drop faults are not altered, the dropped faults are guaranteed to remain detected. This ensures that fault coverage will not be lost in the process of embedding the test cubes.

The goal in designing the bit-fixing sequence generator is to embed the test cubes with a minimal amount of hardware. A hill-climbing strategy is used in which one bit at a time is added to the *Sequence ID Register* based on maximizing the number of test cubes that are embedded each time. Bits continue to be added to the *Sequence ID Register* until a sufficient number of test cubes have been embedded to satisfy the fault coverage requirement. Complete fault coverage can be obtained by embedding test cubes for all of the undetected faults.

For each bit that is added to the *Sequence ID Register*, the first step is to determine which patterns the bit will be active for (i.e., which patterns it will alter). In order not to reduce the fault coverage, it is important to choose a set of patterns that do not currently drop any faults in the circuit under test. In order to minimize the *Bit-Fixing Sequence Selection Logic*, it is important to choose a set of patterns that are easy to decode. The set of patterns for which the new *Sequence ID Register* bit will be active are decoded from the starting state of the LFSR for each pattern. Let $\mathbf{F}$ be a Boolean function equal to the sum of the minterms corresponding to the starting state for each pattern that drops faults. Then, an implicant in $\mathbf{F}'$ corresponds to a set of patterns that do not drop faults and can be decoded by an n-input AND gate, where n is the number of literals in the implicant. A binate covering procedure can be used to choose the largest implicant in $\mathbf{F}'$ (see [28]). The largest implicant requires the least logic to decode and corresponds to the largest set of pseudorandom patterns that do not drop any faults. These are the patterns that will activate and, hence, be altered by the new *Sequence ID Register* bit.

In the design example, there are five starting LFSR states that correspond to the patterns that drop faults. They are listed at the top of Fig. 6. The function $\mathbf{F}$ is formed and the largest implicant in the complement of $\mathbf{F}$ is found. The largest implicant is $00XXX$. Whenever the first two bits in a starting state of the LFSR are both "0," then the new *Sequence ID Register* bit is activated. Thus, there are four patterns for which the new *Sequence ID Register* bit will be activated.

After the set of patterns that activate the new *Sequence ID Register* bit have been determined, the next step is to determine which bits in the patterns will be fixed when the new *Sequence ID Register* bit is activated. The goal is to fix the bits in a way that embeds as many test cubes as possible. The strategy is to find some good candidate sets of bits to fix and then compute how many test cubes would be embedded if each were used. The candidate that embeds the largest number of test cubes is then selected.

The candidate sets of bits to fix are determined by looking at bit correlation among the test cubes. For example, if the two test cubes $1010X$ and $00X11$ are to be embedded, then fixing the second bit position to a "0," the third bit position to a "1," and the fifth bit position to a "1" would help to embed both test cubes in the pseudorandom bit sequence. However, fixing the

first bit to a "1" or fixing the fourth bit to a "0" would only help to embed the first test cube; it would prevent the second test cube from being embedded. The reason for this is that the two test cubes have conflicting values in the first and fourth bit. So given a set of test cubes to embed, the best bits to fix are the ones in which there are no conflicting values among the test cubes. The procedure for selecting the set of bits to fix is as follows (the procedure is illustrated for the design example at the bottom of Fig. 6).

1) Place all test cubes to be embedded into the initial set of test cubes.

   Begin by considering all of the test cubes that need to be embedded.

   [In the design example in Fig. 6, all 4 test cubes are initially considered.]
2) Identify bits where there are no conflicting values among the test cubes.

   Look at each bit position. If one or more test cubes has a "1" and one or more test cubes has a "0" in the bit position, then there is a conflict. If all of the test cubes have either a "1" ("0") or an "X," then the bit can be fixed to a "1" ("0").

   [In the design example in Fig. 6, when all 4 test cubes are considered, only the last bit position has no conflicting values. All 4 of the test cubes have either a "0" or an "X" in the last bit position.]
3) Compute the number of test cubes that would be embedded by fixing this candidate set of bits.

   For each pattern that activates the new *Sequence ID Register* bit, fix the set of bits that was determined in Step 2. Count the number of test cubes that are embedded in the resulting patterns.

   [In the design example in Fig. 6, when the last bit position is fixed to a "0" in the 4 scan patterns that activate the new *Sequence ID Register* bit, it enables the test cube $01XX01XXXX10$ to be embedded in the pseudorandom pattern 011101100011.]
4) If the number of test cubes embedded is larger than that of the best candidate, then mark this as the best candidate.

   The goal is choosing the set of bits to fix is to embed as many test cubes as possible.
5) Remove the test cube that will eliminate the most conflicts.

   One test cube is removed from consideration in order to increase the number of bits that can be fixed. The test cube that is removed is chosen based on reducing the number of conflicting bits in the remaining set of test cubes.

   [In the design example in Fig. 6, if third test cube is eliminated from consideration, the three remaining test cubes have two specified bit positions where there are no conflicts. The third bit can be fixed to a "1" in addition to fixing the last bit to a "0."]
6) If the number of test cubes that are embedded by the best candidate is greater than the number of test cubes that remain, then select the best candidate. Otherwise, loop back to Step 2.

   The next candidate set of bits to fix will only help to embed the remaining set of test cubes and, therefore, has limited potential. If it is not possible for the next candidate to embed more test cubes than the best candidate, then the best candidate is selected as the set of bits to fix.
7) Eliminate as many fixed bits as possible without reducing the number of embedded test cubes.

In order to minimize hardware area, it is desirable to fix as few bits as possible. It may be possible to embed the test cubes without fixing all of the bits in the selected set. An attempt is made to reduce the number of fixed bits by eliminating one bit at a time and checking to see if the same test cubes are embedded.

The bit-fixing sequence generator is designed so that when the new *Sequence ID Register* bit is activated, the set of bits selected by the procedure above is fixed. The pseudorandom patterns that are altered to embed each test cube are added to the set of patterns that drop faults (one pattern per embedded test cube). This is done to ensure that those patterns are not further altered such that they would no longer embed the test cubes. If the fault coverage is not sufficient after adding the new *Sequence ID Register* bit, then another *Sequence ID Register* bit is added to embed more test cubes.

In the design example in Fig. 6, when all four test cubes are considered, the only specified bit position in which there are no conflicts is the last bit position, which can be fixed to a "0." Fixing this bit enables one test cube to be embedded. However, when one of the test cubes is eliminated from consideration then the three remaining test cubes have two specified bit positions where there are no conflicts. Fixing these two bits enables all three of the remaining test cubes to be embedded. Thus, this is the selected set of bits to fix when the new *Sequence ID Register* bit is activated. There is still one test cube that has not been embedded. Since complete fault coverage is required, another bit must be added to the *Sequence ID Register*. The three pseudorandom patterns in which the three test cubes were embedded are added to the set of patterns that drop faults and the procedure for adding a new *Sequence ID Register* bit is repeated.

C. Synthesizing Bit-Fixing Sequence Generation Logic

When enough bits have been added to the *Sequence ID Register* to provide sufficient fault coverage, the remaining task is to synthesize the *Bit-Fixing Sequence Generation Logic*. The *Bit-Fixing Sequence Generation Logic* generates the *fix-to-1* and *fix-to-0* control signals to fix the appropriate bits in the sequence depending on which *Sequence ID Register* bits are active. For each *Sequence ID Register* bit that is active, control signals are generated when certain states of the counter are decoded.

The process of constructing the *Bit-Fixing Sequence Generation Logic* is best explained with an example. The *Bit-Fixing Sequence Generation Logic* for the design example is shown in Fig. 7. The first bit in the *Sequence ID Register* is activated whenever the first two bits in the starting seed for a pattern are both "0." This condition is decoded using a two-input AND gate and loading the *Sequence ID Register* right before shifting a new pattern into the scan chain. When the first bit in the *Sequence ID Register* is active, it fixes the first bit shifted into the scan chain to a "0" and the tenth bit shifted into the scan chain to a "1." This is done by generating a *fix-to-0* signal when the counter is in the "cnt-1" state and a *fix-to-1* signal when the counter is in

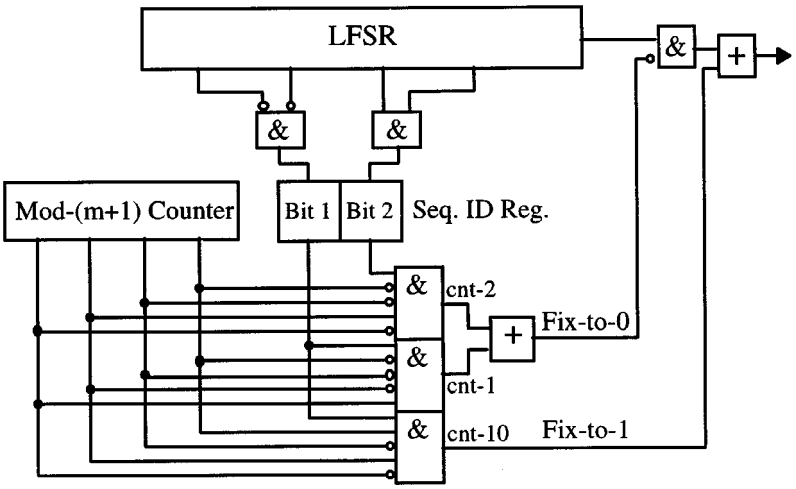


fig. 7. Design example: bit-fixing sequence generation logic prior to multilevel logic optimization.

"cnt-10" state. The second bit in the *Sequence ID Register* is activated whenever the third and fourth bit in the starting seed for a pattern are both "1." When the second bit in the *Sequence ID Register* is activated, it fixes the second bit shifted into the scan chain to a "0." This is done by generating a *fix-to-0* signal when the counter is in "cnt-2" state.

When constructing the *Bit-Fixing Sequence Generation Logic*, the states of the counter can be decoded by simply using n-input AND gates, where n is equal to the number of bits in the counter. However, once the logic has been constructed, it should be minimized using a multilevel logic optimization tool. The don't care conditions due to the unused states of the counter can be used to minimize the logic but, more importantly, the logic can be factored. Because the number of inputs to the logic is small, factoring is very effective for significantly minimizing the *Bit-Fixing Sequence Generation Logic*.

IV. Correlating ATPG Procedure

One way to reduce the overhead of the bit-fixing sequence generator is to use a special ATPG procedure to find test cubes for the r.p.r. faults that maximizes the amount of correlation in the test cubes. In this section, a special ATPG procedure for finding correlated test cubes is described. The starting point is the initial set test cubes for the r.p.r. faults. The correlated bit positions in the initial set of test cubes are identified. Then the correlating ATPG procedure is used to find a test cube for each r.p.r. fault that conflicts with as few of the correlated bit positions as possible. This allows the test cube to be embedded with the least amount of bit-fixing. This ATPG task is different from *dynamic compaction* [29], where an attempt is made to find a test cube for a fault by specifying the "don't cares" (Xs) in test cubes for others faults. Dynamic compaction looks for a test cube for a particular fault that has *no conflicts* with other test cubes, whereas the problem of interest here is to find a test cube for a particular fault that has the *fewest number of conflicts* with other test cubes.

A. Initial Input Assignments

The "Correlating ATPG" procedure presented here uses a PODEM-based [30] algorithm in which the inputs corresponding to the correlated bit positions are assigned initial values. Normally, the PODEM algorithm begins with all inputs having unassigned values (Xs). However, in the correlating ATPG procedure, the initial input assignments are made to begin in the part of the search space that would yield the most correlated test cube. If the fault can be detected by making further inputs assignments without backtracking on any of the initial input assignments (i.e., the correlated bit positions), then a test cube can be found with no conflicts in the correlated bit positions. In general, however, some backtracking on the initial input assignments will be necessary to detect the fault. The key to maximizing the bit correlation is to carefully select the order of the backtracking in order to minimize the number of initial assignments that are reversed.

B. Backtracking

Normally, backtracking in the PODEM algorithm is done in the reverse order in which the inputs are assigned (i.e., the last input assignment made is the one that is changed first). Backtracking in the correlating ATPG procedure is done in the same way except for when backtracking on the initial input assignments (i.e., the correlated bit positions). The order in which backtracking is performed on the initial input assignments is determined by using structural heuristics aimed at minimizing the number of initial input assignments that need to be reversed.

Backtracking on the initial input assignments is required when one of the line values implied by the initial input assignments must be complemented in order to allow the fault to be provoked or sensitized to a primary output by subsequent input assignments. If the value implied at the fault site is the same value as the fault polarity [i.e., if a one (zero) is implied at a stuck-at one (zero) fault site], then one or more initial input assignments must be reversed in order to either complement the value implied at the fault site or to imply an X at the fault site such that subsequent input assignment can provoke the fault. Backtracing is done to determine which initial input assignments to reverse. When there is a choice on which gate input to set to a controlling value, decisions are made based on minimizing the total number of initial input assignments that need to be reversed. If the fault site cannot be sensitized

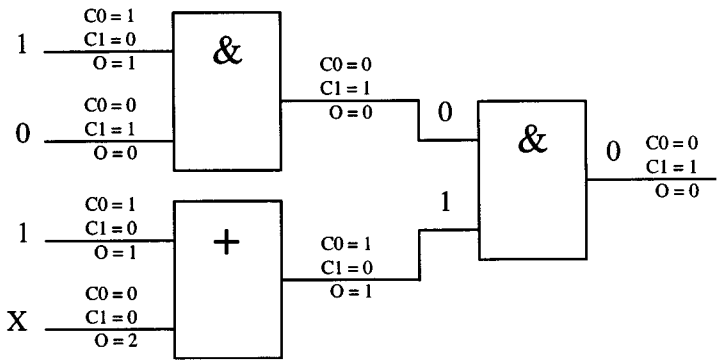


Fig. 8. Controllability and observability values.

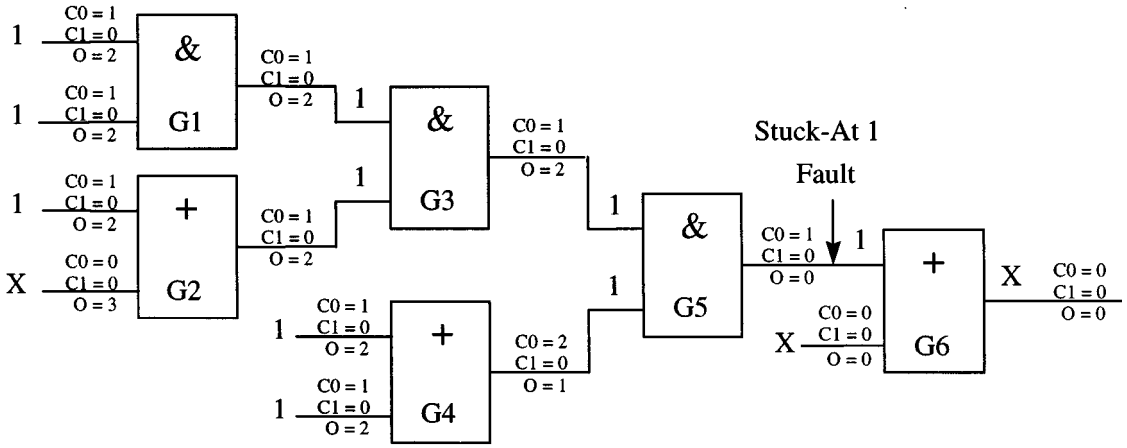


Fig. 9. Example of backtracking with controllability and observability values.

to a primary output with additional input assignments (i.e., no "X path" exists from the "D frontier" to a primary output), then line justification decisions for creating an X path are again based on minimizing the total number of initial input assignments that need to be reversed. These decisions can be made quickly using the controllability and observability cost functions described in Section IV-C

C. Controllability and Observability Cost Functions

In the correlating ATPG procedure, the goal is to minimize the number of initial input assignments that are reversed. Thus, the cost of justifying a line to a particular logic value or observing a line is the number of initial input assignments that need to be reversed. Controllability and observability values are computed to reflect this cost and used to guide line justification decisions. These values are computed when the initial input assignments are made and their implications are determined. The controllability values are determined by traversing the circuit from the primary inputs to the primary outputs. If no value is implied on a line (i.e., it is an X), then both the zero-controllability and one-controllability values for that line are zero since it can be justified to either logic value without reversing any of the initial input assignments. If the value implied on a line is a zero (one), then the zero-controllability (one-controllability) is set to zero and the one-controllability (one-controllability) is set to the number of initial input assignments that need to be reversed in order to complement the value implied on the line or to imply an X on the line. Once the controllability values have been computed, then the observability values can be determined by traversing the circuit from primary outputs to primary inputs and using the controllability values to determine the number of initial input assignments that need to be reversed in order to make the line observable. An example of computing controllability and observability values is shown in Fig. 8. $C0$, $C1$, and O denote the controllability-zero, controllability-one, and observability values, respectively, for each line. Note that there is no initial assignment for the fourth input (i.e., it is an X) so there is no cost for subsequent assignments to that input.

When making line justification decisions in correlating ATPG, the controllability and observability values based on the number of initial input assignments that need to be reversed are the primary criteria. Of course, in many cases, these values will be zero or multiple decisions will have the same value. In those cases, the conventional ATPG heuristics (to minimize ATPG runtime) or the heuristics described in [31] to maximize don't cares can be used.

Consider the example in Fig. 9. The fault being targeted is the output of gate $G5$ stuck-at one. Conventional ATPG would begin with all inputs initially unassigned (Xs). However, in correlating ATPG, the initial input assignments correspond to the correlated bit positions. Implications based on the initial input assignments are made and the controllability and observability values are computed based on the number of initial input assignments that need to be reversed as previously described. Since the

TABLE I
RESULTS FOR BIT-FIXING SEQUENCE GENERATORS

Circuit			Reseeding [19]		Bit-Fixing Sequence Generator		
Circuit Name	Scan Size	Max. Num. Specified Bits	LFSR Size	ROM Bits	LFSR Size	Seq. ID Reg. Size	Literal Count
s420	34	20	20	250	20	1	27
					14	3	70
					10	4	70
s641	54	22	22	183	22	2	63
					14	4	87
					9	6	109
s838	66	36	36	1623	36	5	168
					14	7	176
					12	7	199
s1196	32	17	17	267	17	4	67
					14	4	71
					12	8	102
s5378	214	19	27	726	19	3	163
					14	4	174
					12	9	367
C2670	233	48	60	3412	48	4	328
					16	5	334
					10	12	427
C7552	207	100	100	5241	100	7	741
					36	8	782
					17	13	828

value implied at the fault site is the same as the fault polarity, one or more of the initial input assignments must be reversed to justify a zero at the fault site. Backtracing is done to determine which initial input assignments to reverse. Backtracing can be done through either gate $G3$ or gate $G4$. Since the zero-controllability at the output of gate $G3$ is less than the zero-controllability at the output of gate $G4$, backtracing is done through gate $G3$. Next, there is a decision whether to backtrace through gate $G1$ or gate $G2$. The zero-controllability values are equal for gate $G1$ and gate $G2$ because in either case, one input assignment will need to be reversed. In this case, a secondary criteria can be used in making the decision. For example, if the secondary criteria was to maximize the don't cares (Xs), then backtracing would be done through gate $G1$ since going through gate $G2$ would require assigning a value to a currently unassigned input (in addition to reversing the input assigned to a one).

D. Postprocessing

The last step after a test cube that detects the fault has been found is to try to complement the value of any bit positions that conflict with the correlated bit positions. For each bit position that conflicts with a correlated bit position, the value is complemented and the resulting test cube is simulated to see if the fault is still detected. If the fault is no longer detected, then the bit position is returned to its previous value. Unlike the "maximal compaction" procedure described in [32], if it is possible to complement the bit, then the bit is left at the complemented value rather than making it an X. This is done to maximize the possibility of complementing other bits since the goal is to minimize the number of conflicts.

E. Backtracking Limit

The goal of the correlating ATPG procedure is to maximize correlation as opposed to conventional ATPG procedures whose goal is to minimize execution time. One potential problem is that the heuristics used in the correlating ATPG procedure may result in more backtracking. However, a limit can be placed on the backtracking based on the minimum amount of correlation that is acceptable. Note that the correlating ATPG procedure need only be used for finding test cubes for the r.p.r. faults, not all faults.

V. EXPERIMENTAL RESULTS

The procedure described in this paper was used to design bit-fixing sequence generators for some of the ISCAS benchmark circuits [33], [34] that contain r.p.r. faults. The primary inputs and flip-flops in each circuit were configured in a scan chain. The bit-fixing sequence generators were designed to provide complete fault coverage of all detectable single stuck-at faults for a test length of 10 000 patterns.

In Table I, results are shown comparing the area of the bit-fixing generator for different size LFSRs. A conventional ATPG procedure was used to find the test cubes for the r.p.r. faults. The size of the scan chain is shown for each circuit followed by the maximum number of specified bits in any test cube contained in the test set reported in [19]. Results are shown for the bit-fixing sequence generator required for different size LFSRs. For each different size LFSR, the number of bits in the *Sequence ID Register* is shown along with the factored form literal count for the multilevel logic required to implement the bit-fixing sequence generator. For each circuit, results were shown for an LFSR with as many stages as the maximum number of specified bits as well

TABLE II
RESULTS USING BIT-CORRELATING ATPG

Circuit		Bit-Fixing Sequence Generator with Conventional ATPG			Bit-Fixing Sequence Generator with Bit Correlating ATPG		
Circuit Name	Scan Size	LFSR Size	Seq. ID. Reg Size	Literal count	LFSR Size	Seq. ID. Reg Size	Literal count
s420	34	14	3	70	14	1	36
		10	4	70	10	3	59
s641	54	14	4	87	14	2	80
		9	6	109	9	5	98
s838	66	14	7	176	14	5	164
		12	7	199	12	6	183
s1196	32	14	4	71	14	4	69
		12	8	102	12	7	97
s5378	214	14	4	174	14	4	164
		12	9	367	12	8	332
C2670	233	16	5	334	16	4	313
		10	12	427	10	8	385
C7552	207	36	8	782	36	7	753
		17	13	828	17	11	806

as smaller LFSRs. For smaller LFSRs, extra test cubes must be embedded in order to detect faults that are missed due to linear dependencies in the LFSR thereby resulting in an increase in the area of the bit-fixing sequence generator. As can be seen, in some cases adding just a small amount of logic to the bit-fixing sequence generator permits the use of a much smaller LFSR. Consider C2670, using a 16-stage LFSR instead of a 48-stage LFSR only requires an additional 6 literals. However, in some cases there is a large increase in the amount of logic required for using a smaller LFSR. Consider s5378, using a 12-stage LFSR instead of a 14-stage LFSR increases the amount of logic in the bit-fixing sequence generator by more than a factor of two.

Results for the reseeding method presented in [19] are shown in Table I for comparison. The size of the LFSR and the number of bits stored in the ROM are shown. Note that the reseeding method requires that the LFSR have at least as many stages as the maximum number of specified bits in any test cube. It is difficult to directly compare the two methods because they are implemented differently (ROM versus multilevel logic) and require very different control logic. The reseeding method requires that the LFSR have programmable feedback logic and parallel load capability as well as additional control logic for loading the seeds from the ROM.

Results are shown in Table II comparing the overhead for encoding test cubes generated with conventional ATPG and those generated using the special bit-correlating ATPG procedure described in Section IV. Note that both the amount of combinational logic (i.e., literal count) and more significantly the number of flip flops required (i.e., *Sequence ID Register Size*) are reduced.

VI. CONCLUSION

A synthesis procedure for generating sequence altering logic to embed deterministic test cubes in a pseudorandom sequence has been presented. It constructs a sequential multilevel circuit that very efficiently encodes the deterministic test cubes. This approach can achieve any desired fault coverage during BIST by detecting the r.p.r. faults missed by the pseudorandom patterns.

There are three important features of the mixed-mode scheme presented in this paper. The first is that test cubes for the r.p.r. faults are embedded in the pseudorandom bit sequence. Since there are so many possible pseudorandom patterns in which to embed each test cube, the bit fixing required to embed a set of test cubes can be correlated in certain bit positions to minimize hardware. The second feature is that a one-phase test is used. Having only one phase simplifies the BIST control logic. The third feature is that smaller LFSRs can be used. There is a tradeoff between the size of the LFSR and the amount of bit-fixing logic; therefore, the LFSR size can be chosen to minimize the overall area. These three features make the scheme presented in this paper an attractive option for BIST in circuits with scan.

ACKNOWLEDGMENT

The authors would like to thank V. Lo, M. Karkala, and Prof. H.-J. Wunderlich in helping with this paper.

REFERENCES

[1] E. B. Eichelberger and E. Lindbloom, "Random-pattern coverage enhancement and diagnosis for LSSD logic self-test," *IBM J. Res. Develop.*, vol. 27, no. 3, pp. 265–272, May 1983.

[2] K.-T. Cheng and C. J. Lin, "Timing-driven test point insertion for full-scan and partial-scan BIST," in *Proc. Int. Test Conf.*, Oct. 1995, pp. 506–514.

[3] N. A. Touba and E. J. McCluskey, "Test point insertion based on path tracing," in *Proc. VLSI Test Symp.*, 1996, pp. 2–8.

[4] N. Tamarapalli and J. Rajski, "Constructive multiphase test point insertion for scan-based BIST," in *Proc. Int. Test Conf.*, Oct. 1996, pp. 649–658.

[5] N. A. Touba and E. J. McCluskey, "Automated logic synthesis of random pattern testable circuits," in *Proc. Int. Test Conf.*, Oct. 1994, pp. 174–183.

[6] C.-H. Chiang and S. K. Gupta, "Random pattern testable logic synthesis," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1994, pp. 125–128.

[7] M. Chatterjee, D. K. Pradhan, and W. Kunz, "LOT: Logic optimization with testability—New transformations using recursive learning," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1995, pp. 318–325.
[8] Z. Zhao, B. Pouya, and N. A. Touba, "BETSY: Synthesizing circuits for a specified BIST environment," in *Proc. Int. Test Conf.*, Oct. 1998, pp. 144–153.
[9] F. Brglez, C. Gloster, and G. Kedem, "Hardware-based weighted random pattern generation for boundary scan," in *Proc. Int. Test Conf.*, Oct. 1989, pp. 264–274.
[10] F. Muradali, V. K. Agarwal, and B. Nadeau-Dostie, "A new procedure for weighted random built-in self-test," in *Proc. Int. Test Conf.*, Oct. 1990, pp. 660–668.
[11] M. F. AlShaibi and C. R. Kime, "Fixed-biased pseudorandom built-in self-test for random pattern resistant circuits," in *Proc. Int. Test Conf.*, Oct. 1994, pp. 929–938.
[12] "MFBIST: A BIST method for random pattern resistant circuits," in *Proc. Int. Test Conf.*, Oct. 1996, pp. 176–185.
[13] H.-J. Wunderlich, "Multiple distributions for biased random test patterns," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 584–593, June 1990.
[14] B. Koenemann, "LFSR-coded test patterns for scan designs," in *Proc. Eur. Test Conf.*, Mar. 1991, pp. 237–242.
[15] S. Venkataraman, J. Rajski, S. Hellebrand, and S. Tarnick, "An efficient BIST scheme based on reseeding of multiple polynomial linear feedback shift registers," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1993, pp. 572–577.
[16] S. Hellebrand, S. Tarnick, J. Rajski, and B. Courtois, "Generation of vector patterns through reseeding of multiple-polynomial linear feedback shift registers," in *Proc. Int. Test Conf.*, Oct. 1992, pp. 120–129.
[17] S. Hellebrand, J. Rajski, S. Tarnick, S. Venkataraman, and B. Courtois, "Built-in test for circuits with scan based on reseeding of multiple-polynomial linear feedback shift registers," *IEEE Trans. Comput.*, vol. 44, pp. 223–233, Feb. 1995.
[18] N. Zacharia, J. Rajski, and J. Tyszer, "Decompression of test data using variable-length seed LFSRs," in *Proc. VLSI Test Symp.*, Apr. 1995, pp. 426–433.
[19] S. Hellebrand, B. Reeb, S. Tarnick, and H.-J. Wunderlich, "Pattern generation for a deterministic BIST scheme," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1995, pp. 88–94.
[20] H.-J. Wunderlich and G. Kiefer, "Bit-flipping BIST," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1996, pp. 337–343.
[21] G. Kiefer and H.-J. Wunderlich, "Using BIST control for pattern generation," in *Proc. Int. Test Conf.*, Oct. 1997, pp. 347–355.
[22] ——, "Deterministic BIST with multiple scan chains," in *Proc. Int. Test Conf.*, Oct. 1998, pp. 1057–1064.
[23] S. Hellebrand, H.-J. Wunderlich, and A. Hertwig, "Mixed-mode BIST using embedded processors," in *Proc. Int. Test Conf.*, Oct. 1996, pp. 195–204.
[24] R. Dorsch and H.-J. Wunderlich, "Accumulator based deterministic BIST," in *Proc. Int. Test Conf.*, Oct. 1998, pp. 412–421.
[25] N. A. Touba and E. J. McCluskey, "Altering a pseudorandom bit sequence for scan-based BIST," in *Proc. Int. Test Conf.*, Oct. 1996, pp. 167–175.
[26] M. Karkala, N. A. Touba, and H.-J. Wunderlich, "Special ATPG to correlate test patterns for low-overhead mixed-mode BIST," in *Proc. Asian Test Symp.*, Dec. 1998, pp. 492–499.
[27] P. H. Bardell and W. H. McAnney, "Parallel pseudorandom sequences for built-in test," in *Proc. Int. Test Conf.*, Oct. 1984, pp. 302–308.
[28] N. A. Touba and E. J. McCluskey, "Transformed pseudorandom patterns for BIST," in *Proc. VLSI Test Symp.*, Apr. 1995, pp. 410–416.
[29] P. Goel and B. C. Rosales, "Test generation and dynamic compaction of tests," in *Proc. Int. Test Conf.*, Oct. 1979, pp. 189–192.
[30] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Trans. Comput.*, vol. C–30, no. 3, pp. 215–222, Mar. 1981.
[31] B. Reeb and H.-J. Wunderlich, "Deterministic pattern generation for weighted random pattern testing," in *Proc. Eur. Design Test Conf.*, Mar. 1996, pp. 30–36.
[32] I. Pomeranz, L. N. Reddy, and S. M. Reddy, "COMPACTEST: A method to generate compact test sets for combinational circuits," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 1040–1049, Jul. 1993.
[33] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in fortan," in *Proc. Int. Symp. Circuits and Systems*, May 1985, pp. 663–698.
[34] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Proc. Int. Symp. Circuits and Systems*, May 1989, pp. 1929–1934.

Nur A. Touba (S'88–M'96) received the B.S. degree in electrical engineering from the University of Minnesota at Twin Cities, Minneapolis, MN, in 1990 and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1991 and 1996, respectively.

Since 1996, he has been an Assistant Professor with the University of Texas, Austin. He serves on the Technical Program Committees of the International Test Conference, International Conference on Computer-Aided Design, International Conference on Computer Design, International Test Synthesis Workshop, and International On-Line Test Workshop. His current research interests include very large scale integration testing, computer-aided design, and fault-tolerant computing.

Dr. Touba received the National Science Foundation Early Faculty CAREER Award in 1997.

Edward J. McCluskey (S'51–M'55–SM'59–F'65–LF'94) received the A.B. degree in physics and math from Bowdoin College, Brunswick, ME, in 1953 and the B.S., M.S., and Sc.D. degrees in electrical engineering from the Massachusetts Institute of Technology, Cambridge, in 1953, 1953, and 1956, respectively.

He was with the Bell Telephone Laboratories from 1955 to 1959, where he worked on electronic switching systems. In 1959, he became a Professor of Electrical Engineering and the Director of the University Computer Center at Princeton University. In 1966, he joined Stanford University, where he is currently a Professor of Electrical Engineering and Computer Science as well as Director of the Center for Reliable Computing. He founded the Stanford Digital Systems Laboratory (now the Computer Systems Laboratory) in 1969 and the Stanford computer engineering program (now the computer science M.S. degree program) in 1970. He was Director of the Stanford Computer Forum (an industrial affiliates program) until 1978, which was founded by Dr. McCluskey and two of his colleagues in 1970. He developed the first algorithm for designing combinational circuits—the Quine–McCluskey logic minimization procedure—as a doctoral student at the Massachusetts Institute of Technology. At Bell Laboratories and Princeton University, he developed the modern theory of transients (hazards) in logic networks and formulated the concept of operating modes of sequential circuits. His Stanford University research focuses on logic testing, synthesis, design for testability, and fault-tolerant computing. He and his students at the Center for Reliable Computing worked out many key ideas for fault equivalence, probabilistic modeling of logic networks, pseudoexhaustive testing, and watchdog processors. He collaborated with Signetics researchers in developing one of the first practical multivalued logic implementations and then worked out a design technique for such circuitry. He has authored or coauthored several books, including two widely used texts.

Dr. McCluskey is a Fellow of the AAAS and the ACM and a Member of the NAE. He is the recipient of the 1996 IEEE Emanuel R. Piore Award and served as the first President of the IEEE Computer Society.