A BIST Pattern Generator Design for Near-Perfect Fault Coverage

Mitrajit Chatterjee and Dhiraj K. Pradhan, *Fellow*, *IEEE*

Abstract—A new design methodology for a pattern generator is proposed, formulated in the context of on-chip BIST. The design methodology is circuit-specific and uses synthesis techniques to design BIST generators. The pattern generator consists of two components: a pseudorandom pattern generator (like an LFSR or, preferably, a GLFSR) and a combinational logic to map the outputs of the pseudorandom pattern generator. This combinational logic is synthesized to produce a given set of target patterns by mapping the outputs of the pseudorandom pattern generator. It is shown that, for a particular CUT, an area-efficient combinational logic block can be designed/synthesized to achieve 100 (or almost 100) percent single stuck-at fault coverage using a small number of test patterns. This method is significantly different from weighted pattern generation and can guarantee testing of all hard-to-detect faults without expensive test point insertion. Experimental results on common benchmark netlists demonstrate that the fault coverage of the proposed pattern generator is significantly higher compared to conventional pattern generation techniques. The design technique for the logic mapper is unique and can be used effectively to improve existing pattern generators for combinational logic and scan-based BIST structures.

Index Terms—Linear feedback shift registers, built-in self-test, scan, synthesis, test pattern generation, fault coverage, core logic, SOC.

1 Introduction

BUILT-IN-SELF-TEST (BIST) has been widely adopted in the industry at the board level and is gaining increasing acceptance at the IC level. Having a small number of test patterns in a BIST environment results in a reduced test time. This is particularly important when BIST techniques are used for fast online diagnosis and reconfiguration. Though efficient methods have been developed to generate exhaustive and pseudoexhaustive patterns [2], many large circuits require unacceptably large numbers of vectors to attain acceptable test coverage. Consequently, a number of methods have been proposed in the literature to reduce the number of required patterns with additional hardware [1].

One of the most popular alternatives to using equiprobable pseudorandom testing is weighted random pattern testing (WRPT) [1]. WRPT is a technique in which the design of the pseudorandom pattern generator (PRPG) is altered so as to produce a desired distribution of 1s and 0s for each primary input of the circuit under test (CUT). Either a single set or multiple sets of weights can be used. Several procedures have been developed for determining the optimal set of weights [4], [5], [6], [7], [8], [9], [10]. These are based on analytical calculation of fault detection probabilities and certain heuristics. Multiple sets of weights are not always practical for on-chip IC BIST.

Though the WRPT technique can reduce the test length and be quite effective at board or module level, for large ICs, the number of test patterns required can still be excessively large. A technique which can achieve high fault coverage using fewer patterns is the cube-contained random patterns technique [22]. Here, reductions in test length are achieved by successively assigning temporary fixed values to the selected inputs during the random pattern generation process. The whole test set is divided into a number of partitions and, at each partition, a particular set of inputs is assigned some fixed values. However, a generator may need a control circuit to switch from one test set partition to another and may result in an increased hardware overhead. In the context of scan-based logic BIST, two of the recently available techniques involve using a "PRPG + Phase-shifter" [11], [12], [13], [14] or a "PRPG + Bit-fixing-sequence-generator" [38]. Here, the basic philosophy is to achieve reduction of test lengths by altering the scan-based PRPG vectors by phase shifters or synthesized logic. Both these methods rely on partitioning test sets and each partition is optimized to reduce additional hardware.

To further reduce the test length, designs may embed the test patterns of the hard-to-detect-faults or a compact test set in the data path of the pseudorandom pattern generator [17], [18], [19], [20], [21], [23]. However, embedding can be expensive in terms of area; therefore, it may not be always suitable for IC-BIST.

This paper proposes a new methodology of developing pattern generators which can achieve almost 100 percent single stuck-at fault coverage requiring, typically, a modest number of patterns while guaranteeing detection of all hard-to-detect faults. The area overhead of the proposed scheme is comparable to that of a WRPT generator. The proposed method aims at generating test patterns using a GLFSR [47] as the basic pattern generator (PRPG), whose outputs are mapped by a mapping logic and input to CUT.

- *M. Chatterjee is with the Internetworking Products Division, Integrated Device Technology Inc., Santa Clara, CA 95054. E-mail: mitrajit@idt.com.*
- *D.K. Pradhan is with the Department of Computer Science, University of Bristol, Bristol BS8 1UB, UK. E-mail: pradhan@cs.bris.ac.uk.*

Manuscript received 8 Feb. 2001; revised 18 Dec. 2002; accepted 24 Dec. 2002. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 113579.

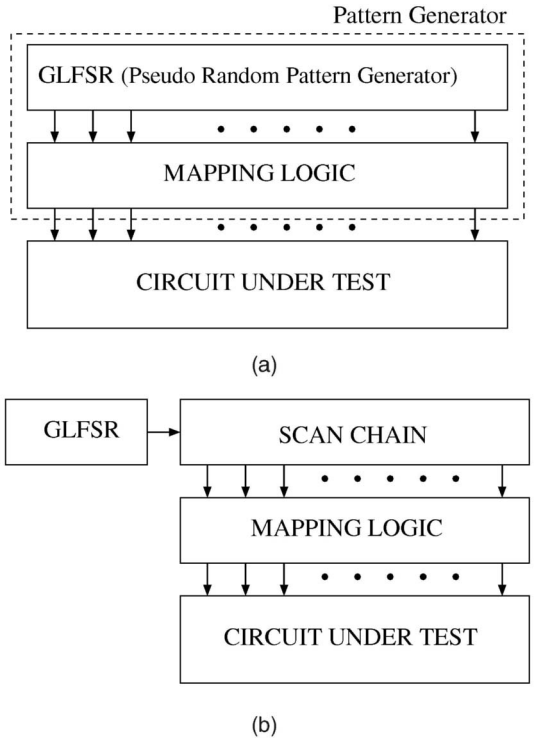

Fig. 1. Proposed test pattern generator: (a) parallel input BIST (b) scan chain-based design.

Essentially, the mapping logic transforms the outputs of the GLFSR into test vectors for the CUT, as shown in Fig. 1. The technique proposed is equally applicable when the mapper logic outputs are directly connected to CUT inputs, as shown in Fig. 1a, or through a scan chain, as shown in Fig. 1b. For the purpose of illustration, we use the organization shown in Fig. 1a throughout the paper. However, when tests are applied as shown in Fig. 1b, the scan chain may have to be segmented. Thus, the proposed pattern generator consists of two distinct components: a GLFSR and a combinational block. During the BIST mode, the GLFSR is loaded with an initial seed and is clocked to generate patterns for the mapping logic. We propose using GLFSRs as the built-in PRPG as they are known to provide better coverage of majority of the faults [47].

The other contribution of this paper is a new synthesis methodology to design the mapping logic. The synthesis procedure presented here is different from standard synthesis procedures [34]. Here, we exploit the fact that the input-output relationship of the logic is *not* fixed and is dependent on a chosen matching between the PRPG patterns and the target patterns. This synthesis procedure is an iterative process involving incremental mapping of a given set of PRPG patterns to a set of test patterns designed to detect certain target faults. This target set of faults may dynamically increase during the synthesis procedure. The performance of the proposed scheme is evaluated using fault simulation.

The proposed work has good relevance to some of the most recently published works on logic BIST [11], [12], [13], [14], [15], [38]. Methods where pattern generator outputs were applied on the CUT on the same cycle (as in Fig. 1) are directly related to the proposed method. In such cases, the proposed method can be used either as a stand-alone procedure or to enhance them. In comparison to scan-based BIST (or STUMPS [1] architecture-based approach) [11], [12], [38], GLFSR and the proposed synthesis method can be easily incorporated to improve fault-coverage or area. For example, one may try to improve the "LFSR + Phase-shifter" logic in [12], [14] by using "GLFSR + Mapping-logic." Also, one can find a way to improve the "LFSR + Bit-fixing-sequence-generator" [38] using the synthesis method proposed here.

This paper is organized as follows: The next section reviews GLFSR and its implementation as a PRPG. Section 3 describes the central idea of the proposed scheme, depicting how the tests of the hard-to-detect faults are determined. Section 4 describes the design methodology for the combinational logic and is one of the main contributions of the paper. Section 5 presents experimental results which compare the performance of our pattern generator with other state-of-the-art pattern generator designs. We conclude in Section 6. Details of the synthesis and the matching procedure are discussed in the Appendix.

2 GLFSR—Review

GLFSR (Generalized LFSR) was first proposed for test response compaction [26] and was later shown as an effective test pattern generator [25], [47] and is capable of better fault coverage than LFSR. Fig. 2 illustrates the basic structure of GLFSR. The circuit under test (CUT) is assumed to have $n = (\delta \times m)$ inputs which form the outputs of the GLFSR, the test pattern generator. The inputs and outputs of the storage elements, D_i, $0 \leq i \leq m-1$, are δ bit binary numbers, interpreted as elements over GF(2^δ). The GLFSR, designed over $\mathrm{GF}(2^\delta)$, has all its elements belonging to $\mathrm{GF}(2^\delta)$. The multipliers, adders, and storage elements are all δ bit components. It may be noted that the adder, $\bigoplus$, is simply a collection of δ EX-OR gates and the multiplier, $\bigotimes$, also uses only EX-OR gates. These multipliers require only EX-OR gates because they are *not* true Galois field multipliers where both inputs are variables. The multipliers in GLFSR simply multiply the δ bit feedback input with a *fixed* constant Φ_i. The feedback polynomial can be represented as

$$\Phi(x) = x^m + \Phi_{m-1}x^{m-1} + \ldots + \Phi_1 x + \Phi_0. \qquad (1)$$

Here, the GLFSR has m stages, $D_0, D_1..D_{m-1}$, where each stage has δ storage cells. Each shift shifts δ bits from one stage to the next. The feedback from the $D_{m-1}\mathrm{th}$ stage consists of δ bits and is sent to all the stages. The coefficients of the polynomial, $\Phi(x)$ are over $\mathrm{GF}(2^\delta)$ and define the feedback connections. The ith coefficient, Φ_i, multiplies the feedback input over a Galois field, which can be realized using only XOR gates. As observed in [26] and illustrated in Fig. 3, the GLFSR represents a general structure [26] and all known structures like the LFSR, MISR, multiple MISR, etc., being special cases, were treated uniformly for the study of aliasing. In addition, the GLFSR with a primitive feedback polynomial when both $\delta > 1$ and $m > 1$, termed MLFSR [26], represents a structure that has been shown to be a very

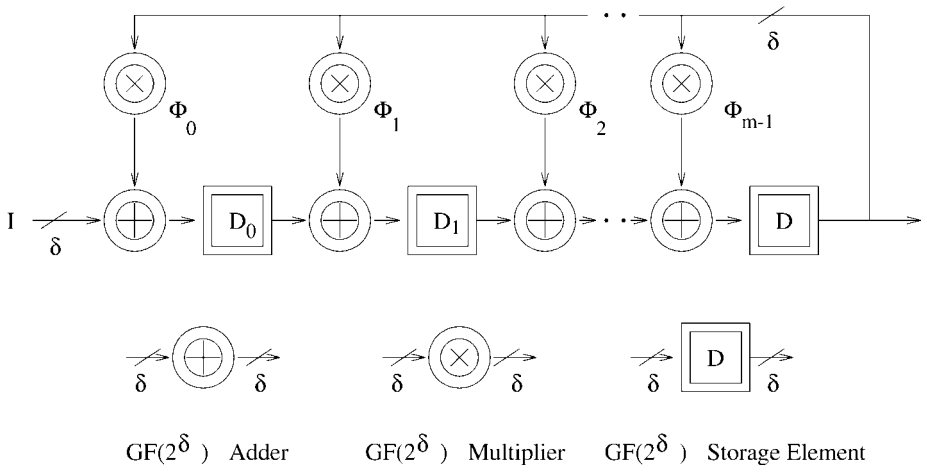

Fig. 2. The Generalized LFSR.

effective PRPG. What we propose is the use of this MLFSR version as the embedded PRPG in Fig. 1.

As an example, Fig. 4 illustrates [47] the structure of GLFSR(2,3) defined over $\mathrm{GF}(2^2)$. Any Galois field $\mathrm{GF}(2^m)$ can be defined by using a primitive polynomial over GF(2) of degree m. The field $\mathrm{GF}(2^2)$ used in this example is defined by using the primitive polynomial $p(x) = x^2 + x + 1$ of degree 2. The feedback polynomial used in constructing this GLFSR is $\Phi(x) = (x^3 + x^2 + \alpha^2 x + \alpha)$, where α is the primitive polynomial of $\mathrm{GF}(2^2)$. Therefore, the coefficients of the feedback polynomial are $\phi_0 = \alpha, \phi_1 = \alpha^2$, and $\phi_2 = 1$. The three storage elements of the GLFSR are represented using polynomials over GF(2) as $D_0 = (a_1 x + a_0)$, $D_1 = (a_3 x + a_2)$, and $D_2 = (a_5 x + a_4)$, respectively. At the beginning of each cycle, those values that are to be fed back into the storage elements are $(a_5 x + a_4)\phi_0$, $(a_5 x + a_4)\phi_1 + (a_1 x + a_0)$, and $(a_5 x + a_4)\phi_2 + (a_3 x + a_2)$, respectively. Putting in the values of the coefficients in polynomial form, the products are

$$\begin{aligned}(a_5 x + a_4)\phi_0 &= ((a_5 x + a_4)x) \bmod p(x)\\ &= (a_5 + a_4)x + (a_5)(a_5 x + a_4)\phi_1\\ &= ((a_5 x + a_4)x^2) \bmod p(x)\\ &= (a_4)x + (a_5 + a_4)(a_5 x + a_4)\phi_2 = (a_5)x + (a_4).\end{aligned}$$

The bits to be generated to XOR with the earlier values of the registers are (a_4), (a_5), and $(a_4 + a_5)$. Generating these values requires only one extra XOR gate; the values can now be inserted to be added to the corresponding registers preceding the $\mathrm{GF}(2^2)$ storage element. The transition matrix [1] for the GLFSR can be expressed as

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}.$$

Table 1 shows the first 15 states of the GLFSR(2, 3) and the GLFSR(1, 6) (which is the standard LFSR) as a comparison. It can be seen below that any two consecutive states of

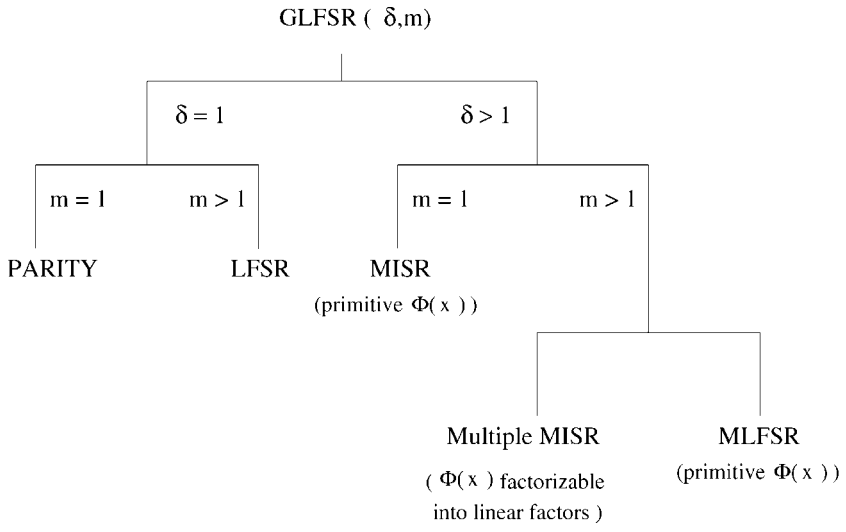

Fig. 3. Special cases of GLFSR.

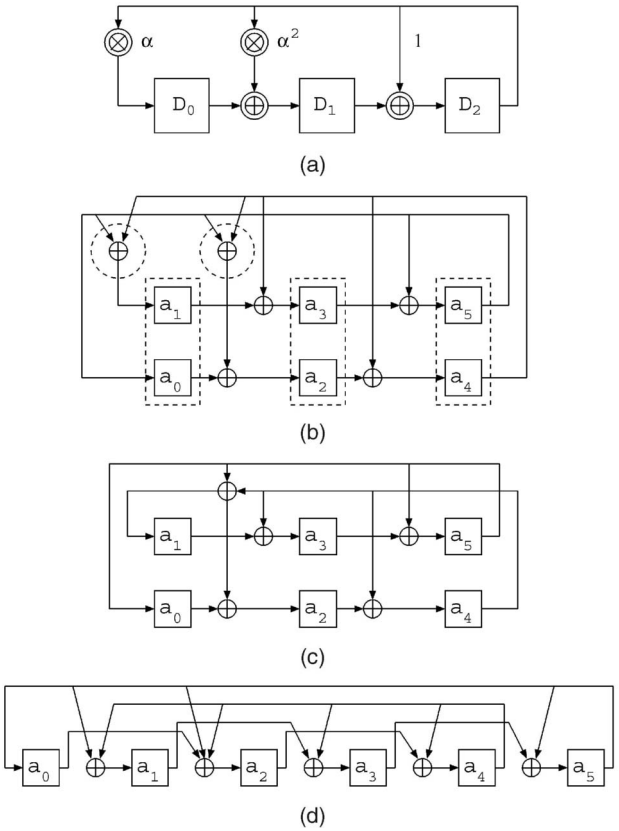

Fig. 4. Structure of the GLFSR: (a) representation, (b) initial structure to be implemented, (c) optimized implementation, and (d) 6-bit pattern generator (single register array implementation).

GLFSR have greater distance than the corresponding pair of consecutive states in LFSR.

The GLFSR, when used to generate patterns for a circuit of n inputs, can have m stages, each element belonging to $\mathrm{GF}(2^{\delta})$, where $(m \times \delta)$ is at least equal to n. To use the GLFSR as a test pattern generator, a nonzero seed is loaded into the GLFSR and is clocked autonomously to produce test patterns. Different values of δ and m, where $\delta m \geq n$, create different types of GLFSRs, capable of generating different types of patterns for the same n-input circuit. For example, for $n = 36$, we can choose the values of (δ, m) to be one of the following combinations: (2, 18), (3, 12), (4, 9), (6, 6), (9, 4), etc. As the value of δ increases, the number of XOR gates needed to realize the generator increases as well. A lower value of δ can, therefore, in general, be more cost-effective. Fortunately, as seen later with regard to the test pattern generator, only a small value of δ actually is optimal in terms of providing maximal coverage. For pattern generation, we propose the use of degree m primitive polynomial over $\mathrm{GF}(2^{\delta})$.

3 Random Pattern Generation with Combinational Gates

The basic structure of our proposed pattern generator has two components: a GLFSR (as a PRPG) and a combinational logic block, as shown in Fig. 1. It may be noted that this design methodology can also cover all LFSR-based designs as a special case, since LFSR is a special case of a GLFSR. For

TABLE 1
States of GLFSR(2, 3) and 6-Bit LFSR(GLFSR(1, 6))

GLFSR(2,3)	GLFSR(1, 6) = Standard LFSR
111111	111111
101000	101111
101010	100111
010100	100011
000101	100001
111000	100000
001111	010000
100100	001000
001001	000100
111011	000010
101001	000001
110011	110000
101011	011000
101101	001100
110010	000110

the sake of simplicity, the following discussion uses the standard LFSR, which is a GLFSR, with $\delta = 1$ to motivate our design. The goal here is to detect *all s-a faults* using small, but not necessarily minimal, numbers of patterns. However, we do allow the flexibility that, for some circuits, almost 100 percent coverage may be an acceptable goal. This flexibility allows for formulation of an efficient synthesis procedure. It can be observed that most of the pseudorandom pattern generators can detect all the easy-to-detect faults using a fairly small number of random patterns; this is because a large number of tests can detect each easy-to-detect fault [43]. The *detectability* of a fault we define in terms of the fraction of all possible test vectors that can detect the particular fault. Thus, an easy-to-detect fault has high *detectability*; a hard-to-detect fault has a low *detectability*. Hence, a hard-to-detect fault usually requires a longer sequence of pseudorandom patterns to be applied.

The basic question, therefore, becomes: Can a pattern generator be designed so that these rare tests can be generated by using a short sequence in a sequence which is also inherently rich in randomness so that the easy-to-detect faults are also detected as usual? Our approach is to develop precisely such a design methodology using a PRPG, followed by a mapper logic. The number of vectors required by the proposed generator to generate the target patterns will depend on the number of target patterns and the number of inputs to the CUT.

It is important to recognize now that there is a fundamental difference between our technique and WRPT. The WRPT is designed to enhance the probability of generating tests for the hard-to-detect faults; ours, though, is aimed at producing tests for these faults with certainty. Example 1, below, illustrates this difference.

Given a CUT, we outline five major steps involved in the design procedure:

1. Select a GLFSR which will be the embedded PRPG engine driving the composite pattern generator.
2. Determine the set of hard-to-detect faults that are difficult to test by the chosen PRPG.

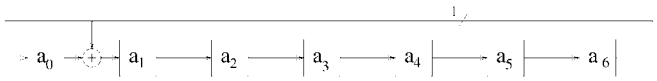

Fig. 5. GLFSR($\delta = 1$) with seven stages.

3. Determine a minimal set of target patterns which test the determined fault list.
4. Design a minimal area overhead combinational circuit with the PRPG so that the combined pattern generator generates all of the target patterns.
5. Evaluate the fault coverage of the developed pattern generator by simulation.

The basic LFSR can also be any one of the conventional pseudorandom pattern generators used in BIST. The randomness of the patterns and experimental results indicate the proposed choice of GLFSR is quite effective for our approach (we feel cellular automata [31], [32] can, as well, prove to be highly effective). The second step is implemented using the procedure outlined below (Section 3.1). The third step is implemented using a standard *automatic test pattern generation*, ATPG tool [29]. The fourth step and the fifth step are described in Section 4 and constitute the main contribution of the paper. Presented below is an example which illustrates the above outlined steps.

Example 1. Let us assume a circuit with seven primary inputs. For the purpose of this example, we use the standard linear feedback shift register (LFSR), GLFSR with $\delta = 1$, with a primitive polynomial given by $(X^7 + X + 1)$, as the basic pseudorandom pattern generator (Fig. 5). Let the patterns shown in Table 2 be those patterns required to detect the hard-to-detect faults. Let us assume that our goal is to design a pattern generator that can generate these six target patterns using a maximum of 20 patterns. The proposed pattern generator, as illustrated in Fig. 6, is an augmentation of the LFSR shown in Fig. 5. With the initial seed of $\{0011100\}$, the patterns transformed by the combinational logic, as shown, generate all the target patterns, as illustrated in Fig. 7. The proposed technique may now be compared to a weighted pattern generator. Consider targeting the same patterns by assigning weights to the GLFSR($\delta = 1$) output; the weight set would be $\{0.25, 0.25, 0.5, 0.25.0.5, 0.25, 0.75\}$ [1]. In the figures we use the term LFSR to denote GLFSR($\delta = 1$). The 20 generated patterns derived from assigning these weights are shown in Fig. 8. It may be seen that, in this case, *only two* of the target patterns were generated. Various other seeds result in a similar disparity between the weighted pattern technique and the proposed technique. As indicated earlier, the difference between

TABLE 2
List of the Hard-to-Detect Faults in Example 1

1.	0	0	1	1	0	0	1
2.	0	1	1	0	1	0	1
3.	0	0	0	0	1	0	0
4.	0	0	1	0	0	1	1
5.	1	1	0	0	0	0	0
6.	0	0	0	1	1	1	1

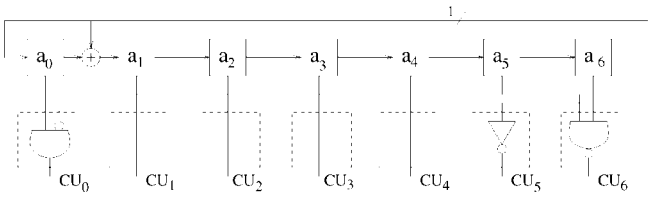

Fig. 6. Structure of the proposed pattern generator for Example 1.

our technique and the weighted pattern technique is that the latter enhances the possibility of generating the targeted patterns, our technique guaranteeing the generation. It may also be noted that, for both techniques, the area overhead is comparable.

3.1 Determining Hard-to-Detect Faults

The procedure used to determine hard-to-detect faults has an impact on the complexity of our generator. Though there are a number of methods to identify random pattern resistant faults [1], the method we choose is summarized as follows: Several different sets of pseudorandom patterns are generated, corresponding to different initial seeds in the chosen PRPG. We perform simulation of the CUT to determine all of the faults detected by the CUT. Those faults detected by each set of patterns are dropped from the fault set as easy-to-detect faults. The remaining faults are then selected as the hard-to-detect faults. It has been observed experimentally that seven to 10 simulations, corresponding to seven to 10 different seeds, are sufficient to determine this hard-to-detect fault set.

In the second step, we generate a minimal set of test patterns, referred to as target patterns, that can test the given hard-to-detect fault list. We use in our experiments the ATPG tool given in [29], although any other tool will work as well.

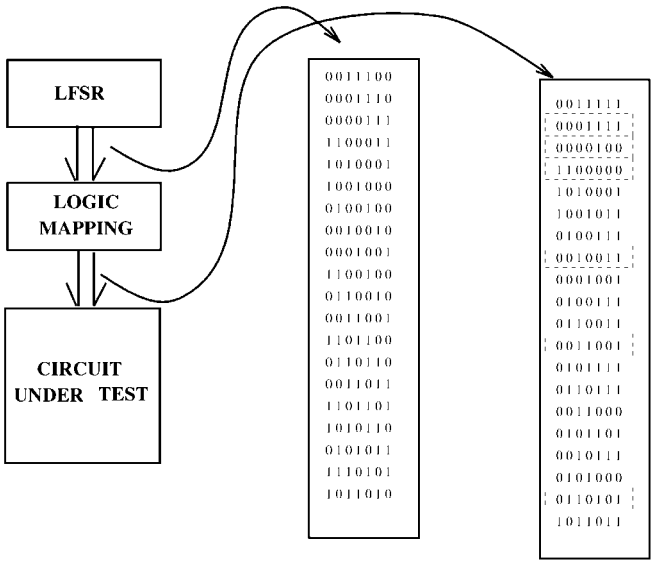

Fig. 7. Patterns transformed by combinational gate design.

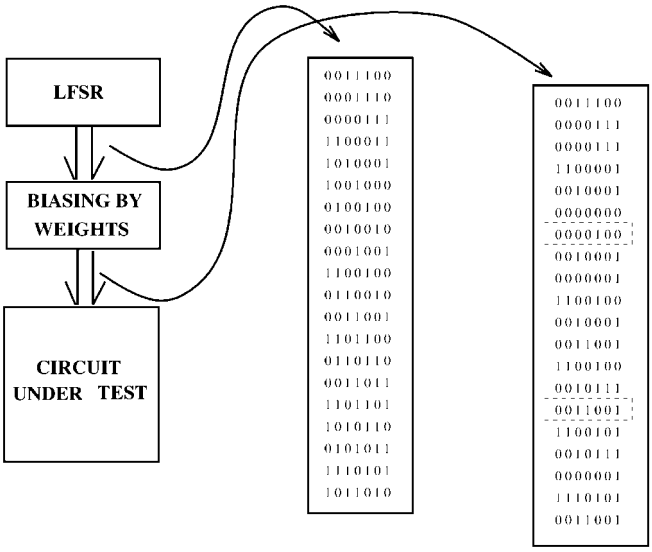

Fig. 8. Patterns transformed by weighted pattern generator.

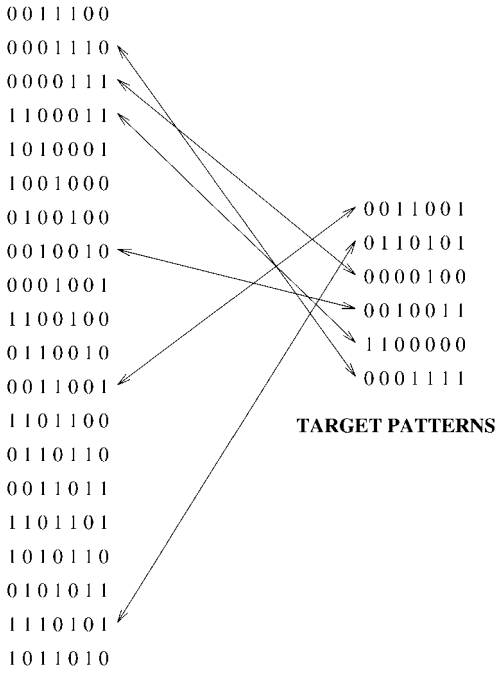

Fig. 9. Matching between the target vectors and the $\mathrm{GLFSR}(\delta = 1)$ vectors in Example 1.

4 Synthesis of Mapping Logic

The final steps of the pattern generator design involve the synthesis of a combinational network to map the outputs of PRPG to a set of patterns to include a specific set of target patterns This is a subproblem of the general synthesis problem [34] and can be formulated as follows: Given an initial seed to the PRPG and a specified test pattern length, design an area-efficient combinational logic block which *ensures* that the combined structure will generate all of the *target* patterns within the specified length.

The mapping logic design thus has to ensure that all the target patterns are generated as mapped patterns within a specified test length. Unlike other synthesis problems, our mapper inputs can be assumed to be random; the basic challenge, therefore, is how to exploit the randomness of the input sequence advantageously. Also, the other difference from a standard synthesis problem is that our mapping is inherently free from the restriction of any ordering; therefore, the synthesis procedure must explore the large space of mappings defined by all possible permutations of the set of target patterns. For example, the mapping which corresponds to the generator designed in Example 1 is shown in Fig. 9. The ordering shown here is not unique since the target patterns are allowed to be generated in any order. Any general synthesis procedure will not yield an area-efficient mapper. The following procedure is, therefore, tailor-made to the framework outlined.

4.1 Preliminaries

Given below are a few preliminaries which have been used in the proposed algorithms.

1. n is the number of inputs to the CUT.
2. Let P be the set of vectors generated by the PRPG.
3. Let P_i be the ith vector in the set P.
4. Similarly, let T be the set of all *target* patterns and T_i be the ith *target* pattern.
5. For any vector A_i, let $A_i(j)$ be the jth bit.
6. We denote the length of a vector A by $\mid A \mid$. Hence, $\mid P_i \mid = \mid T_i \mid = n$.
7. A one-to-one matching from the set of *target* patterns, T, to the PRPG patterns, P, is expressed by M. The PRPG pattern, P_j, matched with T_i, is denoted by $M(P_j)$.
8. We denote the combinational unit, CU, for the ith input of the CUT by CU_i. Each combinational unit has one output with single or multiple inputs. We denote the function of the ith unit, CU_i, by F_i. In Example 1, Fig. 6, F_0 is the logical AND operation and F_5 is a logical NOT operation.
9. The input connection to a combinational gate CU_i is described by an array, I_i. The array, I_i, contains the list of the output indexes of the PRPG that form the inputs to the combinational gate, CU_i. Thus, the number of elements in I_i, denoted by $\mid I_i \mid$, correspond to the number of inputs to the combinational unit. In Example 1, Fig. 6, for CU_0, we have $\mid I_0 \mid = 2$, $I_0 = \{0, 4\}$. Similarly, for CU_1, we have $\mid I_1 \mid = 1$, $I_1 = \{1\}$.
10. The fanout of a PRPG output, i, denoted by $CF(i)$, is equal to the number of different CUs that the ith PRPG output is connected to. Thus, $CF(i)$ can vary from 0 to n. Note that $CF(i)$ is simply the number of combinational blocks to which the output is connected and may not be equal to the actual fanout of the output. In Example 1, Fig. 6, $CF(4) = 2$ and $CF(5) = 1$.
11. The *Hamming distance* between two vectors v and u, denoted by $d(u, v)$, is defined as the number of bits where they differ. For example, if $u = (1001011)$ and $v = (0100011)$, $d(u, v)$ is equal to 3.

TABLE 3
Output Transformations for Example 1

Input Vector	Output values required						
	CU_0	CU_1	CU_2	CU_3	CU_4	CU_5	CU_6
0 0 1 1 0 0 1	0	0	1	1	0	0	1
1 1 1 0 1 0 1	0	1	1	0	1	0	1
0 0 0 0 1 1 1	0	0	0	0	1	0	0
0 0 1 0 0 1 0	0	0	1	0	0	1	1
1 1 0 0 0 1 1	1	1	0	0	0	0	0
0 0 0 1 1 1 0	0	0	0	1	1	1	1

4.2 Matching and Synthesis Procedure

The procedure of finding the matching between the *target* patterns, T_i, and the PRPG patterns, P_i, is closely linked to the procedure of designing the combinational units based on the matching. Different matchings can result in different complexities of the mapper unit. In the proposed method, we associate a matching, M, with a *cost function*, C, which is related to the gate count of the resulting mapper. It is assumed that area overhead of a combinational unit is proportional to its number of inputs. In other words, while matching *target* patterns, T with PRPG patterns, P, we associate a larger area to combinational units which have more input bits. The cost function of a matching, M, with n *target* vectors and n PRPG vectors is the weighted sum of the number of inputs of each combinational unit involved realizing the transformations. The cost function used here is defined in the equation below and is rooted as a standard metric in logic synthesis [34].

$$C(M) = \sum_{i=0}^{n} (W(\mid I_i \mid)). \tag{2}$$

The cost function, C, is to be minimized when we find a matching, M, between the *target* vectors and the PRPG vectors. The initial problem to be solved can be expressed as follows:

Given: A set of Target Vectors, T and PRPG Vectors, P.

Problem: Find a one-to-one mapping, M, between T and P such that $C(M)$ is minimum.

This problem is NP-hard [30] as there are an exponential number of matching combinations possible. We propose a polynomial time procedure here to arrive at a *minimal solution*. Experimental results demonstrate that our procedure is indeed area-efficient and computationally tractable.

It can be observed that a weight, $W(i)$, is associated with the cost of a CU having i inputs. The weights are used in the expression for C to take into account the nonlinear relation between the number of inputs, i, and the area overhead of a combinational unit having i inputs. Thus, we associate the area of a 2 input combinational unit to $W(2)$ and the one with 4 inputs to $W(4)$. When we match k *target vectors* with k PRPG vectors, the cost function, though a crude approximation, gives a quick estimate of the area overhead without actually synthesizing the combinational logic circuit. Our method consists of three steps:

1. First, find a matching between *target* vectors with PRPG vectors.
2. Determine the minimum number of the inputs and the contents of the I_i array for each CU_i.
3. Design an optimized combinational logic circuit for each CU_i.

Once we have determined a matching, M, between the target patterns and the PRPG patterns, the combinational units are designed one at a time. Table 3 illustrates the matching used in Example 1. Given this, an input set, I_i, has to be determined for each CU_i. This input set is designed independently of all other combinational units. First, it may be noted that a small fraction of all outputs of PRPG is used in designing any particular combinational unit. This is a desired consequence of our cost function which is proportional to the number of inputs to the combinational units. The essential factors that are considered while determining I_i include: What is the essentially minimum number of input bits required to realize the particular function?

Steps 1 and 2 are closely related and, hence, are interdependent. Our matching procedure is outlined in Fig. 10. The procedure chooses one target vector, T_i, at a time and matches it to a PRPG vector, P_j. After adding each additional matched vector pair, the cost of the combinational logic, C (as in (2)), due to the mapping, is recomputed. Matchings are chosen so that there is minimum increment to C with each additional vector in the matched list. After each target vector is mapped to a unique PRPG vector and the I_i arrays of each CU_i are determined, a standard two-level logic design tool is used to design the logic mapper so as to keep the number of levels between the PRPG and CUT to a minimum. This is an important consideration in high-speed and at-speed testing. The Appendix gives a detailed description and analysis of Steps 1, 2, and 3 and more comprehensive version of the above **Find_Match** algorithm.

```
Find_Match(Target vectors, PRPG vectors)
Begin
1. Sort target vectors ascendingly based on # of Don't Care bits;
2. Match some initial target vectors to PRPG vectors using minimum Hamming distance;
3. Estimate cost of the matching using the initial target vectors matched;
4. For the rest of the target vectors, do
5.     Match that PRPG vector for the next target vector s.t the added cost is minimum;
6.     Estimate the cost of matching done so far;
End
```

Fig. 10. Overview of the algorithm matching *Target* patterns to PRPG patterns.

4.3 Important Design Constraints

The primary aim of the procedure which matches *target* vectors with PRPG vectors and designs the combinational unit design is the reduction of the cost function, C. However, to preserve much of the randomness in the PRPG vectors during the mapping process, we introduce some *randomness preserving techniques* to ensure fault coverage for the mapped vectors with respect to easy-to-detect faults. We also add some additional *area-saving* design constraints to the development procedure.

4.3.1 Randomness Preserving Heuristics

We have outlined three techniques, as follows:

1. *Use all PRPG outputs*: It was observed that, in most combinational mapper logic, there were a few PRPG outputs which are not used by any of the combinational units, CF, being zero for those PRPG outputs. This implies that the combinational logic utilizes only a partial set of the PRPG outputs and the rest of the PRPG outputs are not reaching any of the circuit-under-test inputs. Replacing a PRPG output with high CF values by a PRPG output with zero CF value (which maintains the same cost function) will result in a larger set of PRPG outputs used by the combinational logic. This will reduce the correlation between the outputs of the CUs, as well as improve the randomness.
2. *Upper bound on the CF values*: In the logic mapper design process, we restrict the CF of a particular PRPG output to either 2 or 3. This puts a restriction on the PRPG output so that it cannot be fed to more than two or three CU designs. If we *assume* that the PRPG outputs are uncorrelated, a PRPG output of $CF = k$ will result in k pattern generator outputs correlated due to a common PRPG output. The lower the value of k for a combinational mapper design, the better will be the randomness of the mapped vectors.
3. *Use EX-OR gates in CUs*: Though the application of EX-OR gates may contribute to a larger area, replacing other 2-input gates by EX-OR or EX-NOR gates wherever possible will improve the signal probability of the CU outputs.

4.3.2 Area Reducing Heuristics

Presented below are heuristics aimed at reducing the area of the logic mapper during *placement* and *routing*.

1. To provide less *routing* area for the combinational units, the additional constraint on choosing the input index array, I_i, of CU_i is that all the indices should not be much further away from the ith bit position. This would ensure that for all of input indices $I_i[j]$, the index number $I_i[j]$ lies in the range, $(i + R)$ and $(i - R)$. For choosing inputs to CU_i, the design procedure also gives a bit position nearer to i, a higher priority, to one away from i.
2. It is always preferable that the maximum number of inputs to a combinational unit is three, four or, at most, five (in very few cases). In a situation where the number of inputs becomes more than four or five, the number of PRPG patterns has to be increased to provide a larger space for finding suitable matches.

4.4 Overall Design Flow

Although one could achieve a very high fault coverage using the initial logic mapper design, it may not always lead to 100 percent fault coverage. The faults that escape the mapped vectors can be classified into two sets:

- Faults that have not yet been identified as hard-to-detect faults.
- New faults that are undetected when the mapped PRPG vectors were used to test the CUT.

To incorporate these faults in the logic mapper design, proposed is a design flow which can guarantee 100 percent fault coverage. The flow is presented in Fig. 11. Such a design flow will achieve 100 percent fault coverage at the end of the design procedure. In the worst case, the design might have a very large fault list. However, our experimental results show that the procedure converges in an average of two to three iterations. If fault resimulation becomes very costly for very large designs, one can create a set of test-patterns, and check how many of the test-patterns are generated by simulating the pattern generator.

As far as synthesis runtime is concerned, finding a match between the target vectors and the PRPG vectors (Step 1) is most time-consuming. The order of the algorithm is proportional to the square of the number of target vectors, $|T|$ (see the Appendix), and is proportional to the cube of the number of inputs to the CUT, n (see the Appendix). Hence, we do not expect the runtime to increase with larger designs and larger numbers of target vectors. Also, if we keep the number of inputs to each CU_i to a small number (up to four or five), the cost of the logic mapper remains less.

5 Experimental Results

The proposed algorithm for pattern generation was implemented to show the performance of the new method. Experiments were conducted on the ISCAS85 [35] benchmark combinational circuits and some combinational profiles of ISCAS89 benchmark circuits; only those circuits which were not easily randomly testable were used. The results were compared with the single set WRPT techniques in [4] and [6] and other types of pattern generation techniques such as [27], [36], [28], and [40].

Results have been compared based on the fault coverage and extra gate count overhead of the pattern biasing hardware.

The results presented here have been significantly improved from the work published in [24]. This is primarily due to improvements in two areas: 1) an improved **Find_Match** algorithm which reduced the cost, and 2) randomness preserving heuristics and the overall design flow, which increased the fault coverage.

5.1 Experimental Framework

We have implemented the proposed algorithm and the algorithm in [4] for comparison. For the sake of fairness,

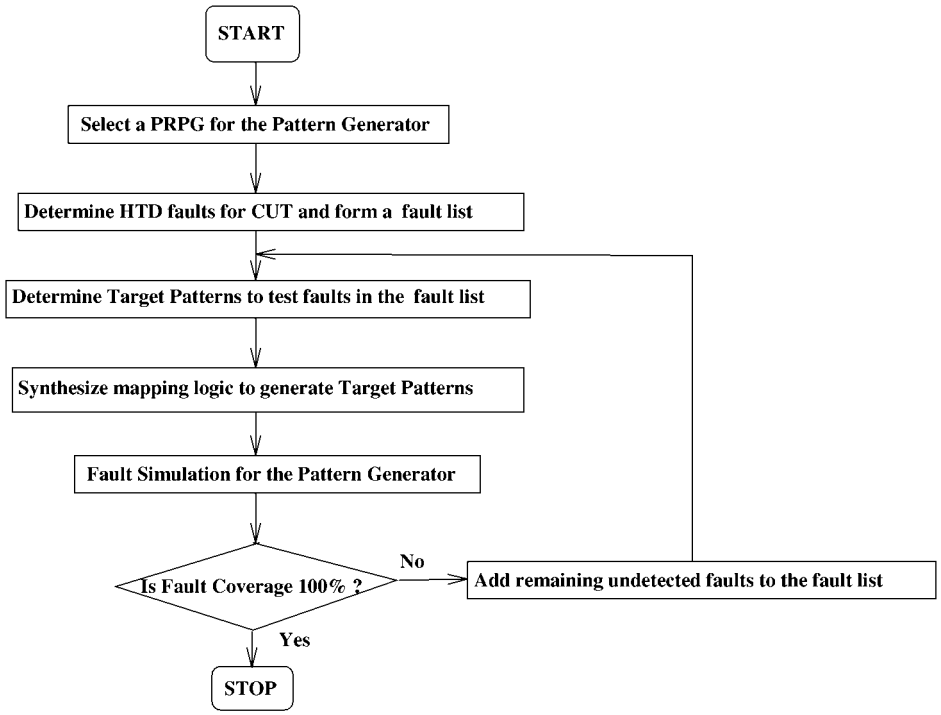

Fig. 11. Design flow for the pattern generator.

GLFSR was chosen as the PRPG for our technique, as well as the weighted pattern technique, the primary reason (as mentioned earlier) being that the GLFSR exhibits better randomness than the LFSR. However, we tried to use an LFSR which is basically GLFSR$(1, n)$ wherever possible. Also, to make the comparison unbiased, we have selected the same set of hard-to-detect faults for both our technique and the weighted pattern technique. The number of GLFSR patterns chosen to search for a suitable mapping of the given *target* patterns was equal to the number of unmapped GLFSR patterns required to detect 90-95 percent of the single stuck-at-faults in the circuit. The combinational units designed for the proposed pattern generator were implemented in a *two-level* realization. Only the tabular method [34] was used for logic minimization for the individual combinational units. As mentioned, we also implemented the WRPT technique proposed in [4], where we processed the target patterns instead of the "Tail Vectors." The weights were limited to 0.125, 0.25, 0.5, 0.75, and 0.875 and were generated by using AND and OR gates at the outputs of the shift registers. Finally, during fault simulation, the number of patterns generated is applied until a predetermined number of consecutive patterns failed to detect any of the remaining undetected faults. An interesting optimization problem will be to search for optimal p, q such that GLFSR(p, q) provides the best results where $n = p \cdot q$.

5.2 Area Overhead

We have compared area overhead in terms of the number of *equivalent 2-input gates* (GE) to realize the combinational logic component of the test pattern generator, a standard technique for comparing area in logic synthesis [34]. Using this metric, the proposed scheme is compared with that in [4]. Since the embedded PRPG is GLFSR for both the proposed method and the WRPT method, we compare the extra area required for augmenting the GLFSR with the combinational logic. The results have been presented in Table 4. The third column provides the name of which generator was used as the PRPG in the combined pattern generator. The fourth column here gives the number of *target* patterns that were selected to design the pattern generators. The fifth column corresponds to the number of GLFSR patterns potentially matched with the given *target* patterns. The last two columns provide the number of two-input gates required to realize the transformation. It should be noted that the number of *equivalent two-input gates* is comparable to the WRT technique [4].

The mappers synthesized for the ISCAS circuits shown above have the following characteristics: First, most of the GLFSR outputs are directly connected, therefore not needing any mapping logic at all. The other outputs require only two or three inputs. Very few combinational units at GLFSR outputs require four or five inputs.

5.3 Fault Coverage

The fault coverage of the proposed pattern generator is compared to that of the WRPT-implemented, based on [4] and the results from [6]. The generators used for the proposed technique and the WRPT technique from [4] are the same as described in Table 4. Table 5 presents the results of the fault coverage on the ISCAS85 benchmark circuits and some ISCAS89 circuits. Columns 3 and 4 depict the achieved fault coverage with respect to the test length for the pattern generator. Columns 4 and 5 depict the same for the WRPT technique implemented, which is based on [4], the only difference being that we used GLFSR instead of an LFSR. Finally, Columns 6 and 7 give the results from [6]. While measuring the fault coverage, only detectable faults

TABLE 4
Area Overhead Comparisons

Circuit Name	# of Inputs	PRPG Used	# of Targets	PRPG patterns	2-i/p Gates	
					Prop.	WRPT, [4]
s641	54	GLFSR(1,54)	18	10000	6	16
s713	54	GLFSR(1,54)	13	5000	4	24
s1196	32	GLFSR(2,16)	30	10000	36	15
c880	60	GLFSR(1,60)	35	1000	21	28
c1355	41	GLFSR(1,41)	4	2000	0	25
c1908	33	GLFSR(1,33)	25	4500	8	19
c2670	233	GLFSR(1,233)	122	5000	119	102
c3540	50	GLFSR(3,17)	18	4500	4	18
c7552	207	GLFSR(1,204)	127	8000	297	149

TABLE 5
Fault Coverage Comparison for the Proposed Technique

Circuit Name	Proposed Method		WRPT [4]		WRPT [6]	
	Test Length	# Undetected	Test Length	# Undetected	Test Length	# Undetected
s641	7700	0	10000	25	-	-
s713	4800	0	5000	28	-	-
s1196	10000	7	10000	123	-	
c880	640	0	1000	21	272	0
c1355	1760	0	3700	3	-	-
c1908	4700	0	5000	47	6629	0
c2670	6128	0	7000	213	7554	31
c3540	4828	0	4500	284	8553	0
c7552	8000	65	10000	307	19000	91

TABLE 6
Comparison with the Mixed Mode Scheme [27]

Circuit Name	Random Test Length	Proposed Method			Mixed-Mode Scheme		
		Test Length	Coverage	GE	Test Length	Coverage	GE
c880	15K	640	100%	21	1K	100%	255
c1355	4K	1.8K	100%	0	3K	100%	59
c1908	10K	4.7K	100%	8	4K	99%	121
c2670	4.5M	6K	100%	119	5K	99%	621
c3540	20K	4.8K	100%	4	4.5K	99%	65
c7552	> 100M	8K	99%	297	8K	99%	1228

were considered. The number of undetected faults listed in Columns 3, 5, and 7 is the number of undetectable faults left among all of the detectable ones.

The results show that, in almost all cases, the proposed pattern generator *outperforms* those pattern generators designed using single weight sets. It has also been observed that the fault coverage based on the proposed generator is better than some of the published results where multiple sets of weights have been used [3], [9], [10]. Any weighted pattern generation technique identifies a set of hard-to-detect faults in the circuit and assigns weights to the bits to increase the probability of generating tests for the faults. The proposed technique, on the contrary, targets the same set of hard-to-detect faults, guaranteeing the generation of the test patterns for those faults. Thus, the proposed pattern generator tests all of the hard-to-detect faults in the identified set, resulting in much higher fault coverage.

An alternative approach to achieve almost perfect fault coverage is to resynthesize the circuit itself, as done, for example, by the tool LOT [45]. However, the test lengths for the resynthesized circuit can be longer than the approach proposed here to attain the near-perfect fault coverage. Therefore, a hybrid approach that combines resynthesis [44], [45], [46] with mapping can yield a better result than those reported here. Further research needs to be carried out to explore this approach.

5.4 Comparison with the Mixed-Mode Scheme

We now compare the proposed method with a pattern generator hardware designed for a mixed mode testing scheme [27]. Here, an LFSR was used to generate the pseudorandom vectors and was then reconfigured as a ring counter to generate deterministic vectors through a network of OR gates. Unlike the proposed method, the deterministic vectors were delivered using a minimal number of vectors and thus can incur a larger area overhead. The comparison is presented in Table 6. For the mixed mode scheme, the

TABLE 7
Comparison with Cube/Rectangular Mapping [36], [37]

Circuit Name	Random Test Length	Proposed Method			Cube/Rectangular Mapping		
		Test Length	Coverage	GE	Test Length	Coverage	GE
s641	1.0M	7.7K	100%	6	10K	100%	11
s713	1.2M	4.8K	100%	4	10K	100%	11
s1196	2.1M	10K	99%	36	10K	100%	21
c880	15K	640	100%	21	1K	100%	27
c1355	4K	1.8K	100%	0	3K	100%	11
c1908	10K	4.7K	100%	8	4K	100%	12
c2670	4.5M	6K	100%	119	5K	100%	121
c3540	20K	4.8K	100%	4	4.5K	100%	13
c7552	> 100M	8K	99%	297	8K	100%	256

"decoding logic + MUX" is counted as the hardware overhead.

5.5 Comparison with Cube/Rectangular Mapping

Another method [36], [37], developed independently, uses LFSR and a mapping logic to generate tests. The procedure described in [36] is based on a class of transformations called "cube mappings." Here, each cube mapping transforms those patterns in the original pattern set (pattern set generated by the LFSR) which do not detect any new faults into a set of patterns that detects the targeted hard-to-detect faults. The method guarantees 100 percent fault coverage since only those patterns that do not detect any new faults are transformed. An enhanced method of the logic mapping synthesis [37] uses a mapping function which corresponds to a minimum rectangular cover in a binate matrix. Though this method compares favorably with recently published weighted pseudorandom pattern methods [3], [10], certain differences with the proposed synthesis method can be mentioned:

- Identifying cubes or rectangles and then transforming only a selected set of vectors may need larger numbers of gates for certain circuits, when compared to the proposed method.
- These kinds of transformations often lead to mapper designs resulting in *more than two levels of logic*, unlike the proposed method.
- Although no comparison of interconnections or routing is done, our approach is based on localized interconnections, whereas the procedures in [36], [37] require more global interconnections.

In summary, there are can be distinctions between the two methods for a specific CUT. Table 7 compares results for cube/rectangular mapping with the proposed method (former old versions of s420 and s838 were not available to us for comparison). Nevertheless, the proposed method compares favorably with [36], [37] in all circuits except s1196 and c7552. Also note that [36], [37] measure an n-input NAND or NOR gate as $(n/2)$ GEs, compared to our measurement of $(n-1)$ GEs.

Another method similar to [36], [37] is proposed in [41]. Here, the target cubes are derived by a machine learning procedure. Due to unavailability of area overhead data, we could not compare the numbers for the benchmark netlists. Nevertheless, the method seems to give similar numbers to [36], [37] and the proposed method.

5.6 Comparison with Minimal Stage Test Pattern Generator Design

The test pattern design approach in [28] partitions circuit inputs into groups, each group corresponding to a test signal. The *extra hardware* incurred in this design is the interconnection between the PRPG and the CUT. This method not only combines unrelated inputs (inputs that do not belong to the same cone) into a test signal, but also determines compatible inputs for the particular CUT, even if the inputs belong to the same cone. Compatibilities between circuit inputs are determined such that all detectable single stuck-at faults in the circuit are guaranteed to be detected. The number of stages of the resultant test pattern generator determines a test length for 100 percent single stuck-at fault coverage. Table 8 compares the test lengths of these test pattern generators with the proposed method for some random-pattern resistant circuits. It can be observed that an addition of a few gates can provide a significant reduction in the test length. The results can be improved further by combining the two techniques.

5.7 Comparison with Ring Architecture Design

The ring architecture [40] is composed out of a set of masks that are cyclically used to transform the PRPG patterns. The set of masks are predetermined, given a particular CUT. Given a set of l masks, say $R_0, R_2, \ldots R_{l-1}$, the cyclic use of the masks consists of the modification of P_0 by R_0, P_1 by $R_1, \ldots, P_{l-1}$ by R_{l-1}, P_l by R_0, P_{l+1} by R_1, and so on. The masks are designed to contain deterministic test cubes and may contain an "all X" mask. The advantage of this method is that the ring architecture has a simple design and does not need a synthesis method to design the extra gates. The determination of the masks and PRPG seed can be a challenge. When the number of masks is low, the ring architecture can be area efficient. When the number of masks is high the area overhead can increase a lot. Table 9 compares test-lengths and the area overheads of the ring architecture method with the proposed method. The proposed method compares favorably with most of the published cases in [40]. There were no published numbers for c2670 and c7552—the two most random-fault-resistant netlists for the ring architecture method to compare.

TABLE 8
Fault Coverage Comparison with Minimal TPG Design [28]

Circuit Name	Random Test Length	Proposed Method			Minimal TPG design		
		Test Length	Coverage	Extra GE	Test Length	Coverage	Extra GE
s641	1.0M	7.7K	100%	6	32K	100%	0
s713	1.2M	4.8K	100%	4	32K	100%	0
s1196	2.1M	10K	99%	36	32K	100%	0
c880	15K	640	100%	21	8K	100%	0
c1355	4K	1.8K	100%	0	2.4K	100%	0
c1908	10K	4.7K	100%	8	8K	100%	0
c2670	4.5M	6K	100%	119	2000K	100%	0
c3540	20K	4.8K	100%	4	128K	100%	0
c7552	> 100M	8K	99%	297	256000K	100%	0

TABLE 9
Fault Coverage Comparison with Ring Architecture Design [40]

Circuit Name	Random Test Length	Proposed Method			Ring Architecture design		
		Test Length	Coverage	Extra GE	Test Length	Coverage	Extra GE
s641	1.0M	7.7K	100%	6	10K	100%	21
s713	1.2M	4.8K	100%	4	10K	100%	13
s1196	2.1M	10K	99%	36	10K	100%	25
c880	15K	640	100%	21	1K	100%	11.5
c1355	4K	1.8K	100%	0	2K	100%	0
c1908	10K	4.7K	100%	8	4K	100%	13
c2670	4.5M	6K	100%	119	-	-	-
c3540	20K	4.8K	100%	4	4.5K	100%	16.5
c7552	> 100M	8K	99%	297	-	-	-

6 Conclusions

Proposed in this paper is a methodology for synthesizing effective pattern generators for BIST applications with small overhead. Our goal here is to detect all single stuck-at faults in combinational circuits. However, this can be extended to non-stuck-at fault models, as well. The comparisons with the weighted random pattern technique and other test pattern generator methods show that the proposed method has significant advantages in the context of IC-BIST where added gate count and delays in the TPG can be a major concern. Our scheme has the potential to achieve 100 percent fault coverage with an area comparable to WRPT. Although we have used a GLFSR here, the method is general and one can be used for standard LFSR [1] or cellular arrays [31], [32]. The methodology also can be extended to multichip modules, where there can be a single GLFSR with different sets of combinational units for different modules. Finally, the proposed method can easily be integrated with any general pattern generation technique or a scan-based technique for performance enhancement.

Appendix A

The following is the description and analysis of the matching and the synthesis procedure (briefly described before in Section 4.2). The matching procedure (Step 1) is closely related to the procedure of determining the input arrays to the CUs (Step 2). We first present the procedure for Step 2 and then use this procedure to formulate Step 1. This section also includes a more comprehensive version of the **Find_Match** algorithm presented earlier.

A.1 Determining Input Arrays of the CUs

Input arrays of the CUs are designed independently of each other. The combinational unit, CU_i, can take as its input all the n output bits of the PRPG. The combinational unit, CU_i, for the *i*th bit is designed such that, for all the *target* patterns, T_j, $F_i(M(T_j))$ is equal to $T_j(i)$. Note that $M(T_j)$ is the PRPG pattern mapped with T_j. Hence, to satisfy the first condition for CU design, we have to satisfy the relation

$$F_i(M(T_j)) = T_j(i) \quad \text{for } i = 0 \text{ to } (n-1), \quad j = 0 \text{ to } (\mid T \mid -1). \tag{3}$$

As mentioned before, F_i will not need all the output bits from the PRPG vectors and the relation to those PRPG output bits will appear as *Don't Cares*, Xs. The proposed method eliminates as many PRPG output bits as possible and lists the rest of the required bit indices in the input array I_i. If we observe the *i*th bit of the *target* vectors, some of the bits have the value 1, some have 0, and the rest are Xs. If $M(T_j)$ is the input to the combinational unit CU_i, $F_i(M(T_j))$ should be a 0 if $T_j(i)$ is a 0 and a 1 if $T_j(i)$ is a 1. CU_i need not care about the target patterns whose *i*th bit is an X. Thus, CU_i is only concerned with the *target* vectors whose *i*th value is a 1 or a 0. We divide these *target* vectors into two sets—*Set1* and *Set0*. *Set1* (*Set0*) contains those *target* vectors whose *i*th value is equal to 1 (0). These sets are a function of the mapping, M, and the bit i for which the CU is being designed. The contents of the sets will be completely different when a different combinational unit is being designed. CU_i, when designed, should be able to distinguish a pattern, $M(T_1)$, where $T_1 \in Set1$ and a

Find_Input_index(Targets,PRPG vectors,Matching)
Begin
1. For each bit i, from 0 to n do
2. Create $Set1 = \{M(T_j) | \ T_j(i) = 1\}$;
3. Create $Set0 = \{M(T_j) | \ T_j(i) = 0\}$;
4. Choose bits from 0 to n for I_i in a greedy manner until (For all $M(T_{j1}) \in Set1$ & $M(T_{j0}) \in Set0$ it satisfies $V(M(T_{j1}) \star I_i) \neq V(M(T_{j0}) \star I_i)$);
5. Remove extra bits, if any in I_i without which the above condition holds;
End

Fig. 12. Algorithm to generate input indexes for the combinational units.

pattern, $M(T_0)$, where $T_0 \in Set0$. The minimum number of bits required to distinguish between $M(T_1)$ and $M(T_0)$ should be used as the inputs to CU_i. Given a set of $M(T_{j1})$ vectors, where $T_{j1} \in Set1$, and a set of $M(T_{j0})$ vectors, where $T_{j0} \in Set0$, a greedy algorithm is used to choose the *minimal number* bit indices that can distinguish between the two sets of vectors. The bit indices found are then listed in the array, I_i.

Example 2. Let us design CU_6 of the pattern generator shown in Example 1. As derived from the matchings, for $i = 6$, we have *Set1* is equal to $\{0011001, 1110101, 0010010, 0001110\}$ and *Set0* is equal to $\{0000111, 1100011\}$. It was found that, to distinguish these two sets using the minimal bits, we can choose bit 5 and bit 6. Looking at the fifth and sixth bits of the two sets of vectors, we have $\{01, 01, 10, 10\}$ for *Set1* and $\{11, 11\}$ for *Set0*. The function can be realized by a NAND gate. The input array, I_6, would be equal to $\{5, 6\}$ and F_6 will be the NAND operation.

Given a set of input bit indices in I_i and the PRPG pattern P_j, the vector that appears as the input to CU_i is given by

$$[P_j(I_i(0)), P_j(I_i(1)), P_j(I_i(2)), \ldots] \equiv V(P_j(I_i(k)), k = 0 \text{ to } \mid I_i \mid).$$

Throughout the remainder of the paper, we will denote $V(P_j(I_i(k)), k = 0 \text{ to } \mid I_i \mid)$ as $V(P_j \star I_i)$. The length of the vector is equal to $\mid I_i \mid$. The procedure describing the scheme of generating the input bit indices for the CUs is presented in Fig. 12.

The *Find_Input_index* procedure begins with a given set of *Target Vectors* and its matched PRPG patterns. At the end, the procedure returns the contents of the input index arrays, Is. In Step 4, the procedure chooses one bit at a time, including it in the array, I_i. The bit whose addition in the array results in distinguishing more $M(T_{j1}) \in Set1$ and $M(T_{j0}) \in Set0$ vectors is chosen first. This method is greedy in nature, stopping when there are enough number of bits to satisfy the condition in Step 4.

Lemma 1. *The order of the algorithm Find_Input_index with* k target *vectors is* $O(k.n^3)$.

Proof. It can be observed that Steps 2-5 are repeated n times. Steps 2 and 3 require $O(n)$ time units. Step 4 is the key stage in the loop. In Step 4, in the worst case, all of the n bits have to be chosen. In such a case, the time complexity for checking whether the collected bits are sufficient to be the inputs to CU_i will be $(k.n)$. Thus, the worst case time complexity for Step 4 is $(k.n^2)$. The worst case time complexity for algorithm *Find_Input_index* will be $(k.n^3)$. □

Theorem 1. *The input set,* I_i*, of* CU_i*, derived from the Find_Input_index procedure, is capable of distinguishing any* $M(T_{j1})$*, where* $T_{j1}(i) = 1$ *from any* $M(T_{j0})$*, where* $T_{j1}(i) = 0$.

Proof. This can be proven by contradiction. Let us assume that the procedure which gives the input index, I_i, is not capable of distinguishing all of the PRPG vectors in *Set1* from the PRPG vectors in *Set0*. This would imply that there exists a pair $(M(T_{k1}), M(T_{k0}))$, where $T_{k1}(i) = 1$, $T_{k0}(i) = 0$, and that inputs to CU_i are the same and not distinguishable. Then, for this pair, we have $V(M(T_{k1}) \star I_i) = V(M(T_{k0}) \star I_i)$. This means that, in the procedure, either the sets were wrongly determined in Steps 2 and 3 or the condition was not satisfied in Step 4, neither of which is possible. □

A.2 Matching Target Vectors

The proposed matching scheme, as described in Fig. 13, utilizes the *Find_Input_ index* procedure to find a matching between the *target* vectors and the PRPG patterns with a minimal C. The procedure picks up one *target* vector, T_i, at a time and matches it with a PRPG vector, P_i. After matching each vector pair, the procedure determines the input indices for the CUs, based on the current matchings, and estimates the cost of the combinational unit area overhead. The cost of matching $(k + 1)$ vectors will be more than or equal to that with k vectors; this is because, with $(k + 1)$ vectors, there will be the existing k vectors, as well as one more vector to work upon and distinguish them. At an instant of time, when k vectors are matched $(k > \mid T \mid)$, let the cost of the matching be C_k. Now, the matching for the next *target* vector is determined so that the cost $C_{(k+1)}$ with the added matching is as little as possible. Ideally, it is best to try each P_i which has not yet been matched as a potential match for T_i and determine the cost with the added potential pair. Determining the cost with each potential pair will make the procedure too *time-intensive* and is not preferable for large T and P. In the proposed method, with each P_i not yet matched, we estimate the extra cost incurred, if P_i is chosen to match T_i. The extra cost will depend on how many combinational units will need extra inputs to realize the matchings of k vectors and $P_i - T_i$ matching. The extra cost, denoted by EC, is a function of the input index arrays with the existing k matchings, P_i and T_i. If the potential

```
Find_Match(Targets,PRPG vectors)
Begin
1. Matching, M = ∅;
2. Sort the vectors in T in an ascending order based on # of Don't Care bits;
3. Choose matchings of the initial T_i vectors based on minimum d(T_i,P_i);
4. Update M;
5. Find_Input_indexs with the current mapping;
6. For the rest of the target patterns, T_i in the ascending order, do
7.    Find that matching P_j for T_i with minimum EC (based on (4));
8.    Update M;
9.    Find_Input_indexs with the current mapping;
End
```

Fig. 13. Algorithm to match *Target* patterns to PRPG patterns.

matching, $T_i - P_i$, is accepted, the combinational unit, CU_j, may or may not need extra inputs. To check whether CU_j needs extra inputs, P_i has to be compared with the *Set1* vectors (or the *Set0* vectors) for CU_j, if $T_i(j)$ is a 0 (or a 1). If $T_i(j)$ is 0(1), for CU_j to need no extra input bits, $V(P_i \star I_j)$ should not be equal to any $V(P_k \star I_j)$, P_k belonging to *Set1* (*Set0*) for CU_j. If the combinational unit, CU_j, needs extra inputs, the extra cost associated with it will also depend on $\mid I_i \mid$ (refer to (2)). Thus, EC is expressed as:

$$EC(k, P_i, T_i) = \sum_{j=0}^{n} (W(\mid I_j \mid +1) - W(\mid I_j \mid)) \times B_j)$$
$$\text{where } B_j = \begin{cases} 0 & \text{No extra bit is required in } CU_j \\ 1 & \text{Otherwise.} \end{cases} \quad (4)$$

The calculation of EC can be used to quickly estimate the extra area incurred if the pair is matched. Our method calculates the EC with every potential $T_i - P_i$ matching and selects the matching with the least EC value. This procedure is the same as Fig. 10.

The procedure first sorts the *target* vectors on the basis of the number of Xs in the vectors. This will enable the vectors with fewer Xs to be matched first and the vectors with larger numbers of Xs to be matched later. It can be observed that the handful of initial matchings plays a significant role in determining the complexity of the combinational logic block. The synthesis is carried out incrementally. The matchings added later are selected based on the partially synthesized combinational block and the matchings are added in the order which will require minimal perturbations to the design. Only one or two matchings are made in Step 3, based on the minimum Hamming distance; the rest of the matchings are formulated using Steps 5-9. The procedure at its termination yields a complete matching between *target* patterns and a subset of PRPG patterns. As the procedure uses the *Find_Input_index* function after selecting each matching, the input index arrays, I_is, are also available at the end of this procedure.

Lemma 2. *The order of the algorithm Find_Match is* $O(\mid T \mid^2 (n^3 + \mid P \mid .n))$.

Proof. We can proceed step by step and analyze the time complexity. Sorting the vectors in Step 2 takes $O(\mid T \mid . \log \mid T \mid .n)$. Choosing the first few initial matchings, based on the Hamming distance, would take $(n. \mid P \mid)$ steps. But, the order of the algorithm is determined by Steps 6-9. The loop 7-9 is repeated $O(\mid T \mid)$ times. Calculation of *EC* takes $O(\mid T \mid .n)$ steps and, hence, Step 7 takes $O(\mid T \mid . \mid P \mid .n)$ time. Step 8 takes a single time unit and Step 9 has an order of $O(\mid T \mid .n^3)$ (*Lemma 1*). Thus, Steps 7-9, in the worst case, take a time complexity of $(\mid T \mid (n^3 + \mid P \mid .n))$. Hence, the order of Steps 6-9 and that of the algorithm is equal to $O(\mid T \mid^2 (n^3 + \mid P \mid .n))$. □

A.3 Synthesis of the Combinational Logic Block

The proposed scheme begins by designing each combinational unit, CU_i independently. Designing CU_i would require the input index array, I_i, and the truth table to be implemented. The truth table is derived from the matched PRPG vectors and the sets, *Set0* and *Set1*. If an input vector to CU_i is equal to $V(P_{j1} \star I_i)$, where $P_{j1} \in Set1$, the output of CU_i will be 1. If the input vector is $V(P_{j0} \star I_i)$, where $P_{j0} \in Set0$, the output of CU_i will be 0. If neither of these conditions is satisfied, the output will be X. Based on the above information, we have implemented the combinational units in a *two level realization*, using the tabular method of logic minimization described in [34]. Once all the combinational units are designed, the overall area can be further minimized, using other two-level logic optimization, as done by *ESPRESSO* [34].

Theorem 2. *The functions implemented by the combinational units satisfy relation (3).*

Proof. The function, F_i for CU_i is implemented from the input index array and the truth table derived from the matchings. Let us assume that the functions are not properly realized and that the vector, $M(T_i)$, at the input of the combinational gates, is transformed to another vector, T_i'. There are two ways to achieve this. The first possibility is that $M(T_i)$ was not correctly identified by the CU and was transformed to T_i' instead of T_i. This would imply that the inputs at the combinational units were not sufficient to identify T_i. As this would contradict *Theorem 1*, this possibility is not valid. The second possibility is that the truth table has not been properly realized. This possibility would violate the methodology of realizing the truth table. Hence, the combinational units designed by the proposed method will satisfy relation (3). □

Theorem 3. *The proposed pattern generator generates all the given* target *patterns.*

Proof. The theorem follows from the description of the pattern generator, definition of the mapping function, and *Theorem 2*. □

ACKNOWLEDGMENTS

The authors would like to thank Subodh Reddy for providing his implementation of the ATPG tool [29].

REFERENCES

[1] P.H. Bardell, W.H. McAnney, and J. Savir, *Built-In Test for VLSI: Pseudo-Random Techniques.* New York: John Wiley & Sons, 1987.
[2] M. Abramovichi, M.A. Breuer, and A.D. Friedman, *Digital Testing and Testable Design.* Compter Science Press, 1993.
[3] M. Bershteyn, "Calculation of Multiple Sets of Weights for Weighted Random Testing," *Proc. Int'l Test Conf.,* pp. 1031-1040, 1993.
[4] F. Muradali, V.K. Agarwal, and B. Nadeau-Dostie, "A New Procedure for Weighted Random Built-In-Self-Test," *Proc. Int'l Test Conf.,* pp. 660-668, 1990,
[5] J.A. Waicukauski and E. Lindbloom, "Fault Detection Effectiveness of Weighted Random Patterns," *Proc. Int'l Test Conf.,* pp. 245-261, 1988.
[6] M.A. Miranda et al., "Generation of Optimized Single Distributions of Weights for Random BIST," *Proc. Int'l Test Conf.,* pp. 1023-1030, 1993.
[7] J. Hartmann and G. Kemnitz, "How to Do Weighted Random Testing for BIST," *Proc. Int'l Conf. Computer-Aided Design (ICCAD),* 1993.
[8] F. Saivoshi, "WTPGA: A Novel Weighted Test Pattern Generation Approach for VLSI BIST," *Proc. Int'l Test Conf.,* pp. 256-262, 1988.
[9] H. Wunderlich, "Multiple Distributions of Biased Random Test Patterns," *IEEE Trans. Computer-Aided Design,* vol. 9, no. 6, June 1990.
[10] M. AlShaibi and C. Kime, "Fixed-Biased Pseudorandom BIST for Random-Pattern-Resistant Circuits," *Proc. Int'l Test Conf.,* pp. 929-938, 1994.
[11] M. Cogswell, D. Pearl, J. Sage, and A. Troidl, "Test Structure Verification of Logical BIST: Problems and Solutions," *Proc. Int'l Test Conf.,* pp. 123-129, 2000.
[12] G. Kiefer and H.J. Wunderlich, "Deterministic BIST with Multiple Scan Chains," *J. Electronic Testing: Theory and Applications (JETTA),* pp. 85-93, Feb.-Apr. 1999.
[13] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan, and J. Rajski, "Logic BIST for Large Industrial Designs: Real Issues and Case Studies," *Proc. Int'l Test Conf.,* pp. 358-367, 1999.
[14] G. Kiefer, H. Vranken, E.J. Marinissen, and H.J. Wunderlich, "Application of Deterministic Logic BIST on Industrial Circuits," *Proc. Int'l Test Conf.,* pp. 105-114, 2000.
[15] D. Das and N.A. Touba, "Reducing Test Data Volume Using External/LBIST Hybrid Test Patterns," *Proc. Int'l Test Conf.,* pp. 115-122, 2000.
[16] V.K. Agarwal and E. Cerny, "Store and Generate Built-in Testing Approach," *Proc. Int'l Fault-Tolerant Computing Symp. (FTCS 11),* pp. 35-40, June 1981.
[17] B. Vasudevan et al., "LFSR Based Deterministic Hardware for At-Speed BIST," *Proc. VLSI Test Symp.,* pp. 201-207, 1992.
[18] S.J. Upadhyaya and L.C. Chen, "On-Chip Test Generation for Combinational Circuits by LFSR Modification," *Proc. Int'l Conf. Computer-Aided Design (ICCAD),* 1993.
[19] J.V. Sas, F. Catthoor, and H.D. Man, "Optimized BIST Strategies for Programmable Data Paths Based on Cellular Automata, " *Proc. Int'l Test Conf.,* pp. 110-119, 1992.
[20] S. Venkataraman et al., "A Efficient BIST Scheme Based on Reseeding of Multiple Polynomial LFSRs," *Proc. Int'l Conf. Computer-Aided Design (ICCAD),* 1993.
[21] S. Boubezari and B. Kaminska, "Cellular Automata Synthesis Based on Pre-Computed Test Vectors for BIST," *Proc. Int'l Conf. Computer-Aided Design (ICCAD),* 1993.
[22] S. Pateras and J. Rajski, "Cube-Contained Random Patterns and Their Application to the Complete Testing of Synthesized Multi-Level Circuits," *Proc. Int'l Test Conf.,* pp. 473-481, 1991.
[23] S. Akers and W. Jansz, "Test Set Embedding in a BIST Environment," *Proc. Int'l Test Conf.,* pp. 257-263, 1989.
[24] M. Chatterjee and D.K. Pradhan, "A Novel Pattern Generator for Near-Perfect Fault Coverage," *Proc. IEEE VLSI Test Symp.,* 1995.
[25] D.K. Pradhan and M. Chatterjee, "GLFSR—A New Pseudo-Random Pattern Generator for BIST," *Proc. Int'l Test Conf.,* pp. 481-490, 1994.
[26] D.K. Pradhan and S. Gupta, "A New Framework for Designing and Analyzing BIST Techniques and Zero Aliasing Compression," *IEEE Trans. Computers,* vol. 40, no. 6, June 1991.
[27] C. Dufaza et al., "BIST Hardware Generator for Mixed Test Scheme," *Proc. European Design and Test Conf.,* 1995.
[28] C.H. Chen and S.K. Gupta, "A Methodology to Design Efficient BIST Test Pattern Generators," *Proc. Int'l Test Conf.,* pp. 814-823, 1995.
[29] W. Kunz and D.K. Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solution to CAD Problems—Test, Verification and Optimization," *IEEE Trans. Computer-Aided Design,* pp. 1143-1158, Sept. 1994.
[30] M. Gary and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP Completeness.* Freeman, 1979.
[31] P.H. Bardell, "Analysis of Cellular Automata Used as a Pseudo-Random Pattern Generators," *Proc. Int'l Test Conf.,* pp. 762-768, 1990.
[32] P.D. Hortensius, R.D. McLeod, and H.C. Card, "Parallel Random Number Generation for VLSI Systems using Cellular Automata," *IEEE Trans. Computers,* vol. 38, no. 10, pp. 1466-1473, Oct. 1989.
[33] E.J. McCluskey, "Verification Testing—A Pseudoexhaustive Test Technique," *IEEE Trans. Computers,* vol. 33, no. 6, pp. 541-546, June 1984.
[34] G. De Micheli, *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.
[35] F. Brglez and H. Fujiwara, "A Neural Netlist of Ten Combinational Benchmark Circuits and a Target Translator in FORTRAN," *Proc. Int'l Symp. Circuits and Systems,* June 1985.
[36] N. Touba and E.J. McCluskey, "Transformed Pseudo-Random Patterns for BIST," *Proc. IEEE VLSI Test Symp.,* 1995.
[37] N. Touba and E.J. McCluskey, "Synthesis of Mapping Logic for Generating Transformed Pseudo-Random Patterns for BIST," *Proc. Int'l Test Conf.,* pp. 674-682, 1995.
[38] N. Touba and E.J. McCluskey, "Bit-Fixing in Pseudorandom Sequences for Scan BIST," *IEEE Trans. Computer-Aided Design,* vol. 20, no. 4, pp 545-555, Apr. 2001.
[39] C. Krishna, A. Jas, and N. Touba, "Test Vector Encoding Using Partial LFSR Reseeding," Proc. Int'l Test Conf., pp. 884-893, Oct. 2001.
[40] C. Fagot, O. Gascuel, P. Girard, and C. Landrault, "A Ring Architecture Strategy for BIST Test Pattern Generation," *Proc. IEEE Seventh Asian Test Symp.,* pp. 418-423, 1998.
[41] C. Fagot, P. Girard, and C. Landrault, "On Using Machine Learning for Logic BIST," *Proc. Int'l Test Conf.,* pp. 338-346, 1997.
[42] H. Liang, S. Hellebrand, and H.J. Wunderlich, "Two Dimensional Test Data Compression for Scan-Based Deterministic BIST," *Proc. Int'l Test Conf.,* pp. 894-902, Oct. 2001.
[43] Y.K. Malaiya and S. Yang, "A Coverage Problem for Random Testing," *Proc. IEEE Int'l Test Conf.,* pp. 237-245, Nov. 1984.
[44] C.-H. Chiang and S.K. Gupta, "Random Pattern Testable Logic Synthesis," *Proc. Int'l Conf. Computer-Aided Design,* pp. 125-128, Nov. 1994.
[45] M. Chatterjee, D.K. Pradhan, and W. Kunz, "LOT: Logic Optimization with Testability—New Transformations for Logic Synthesis," *IEEE Trans. Computer-Aided Design,* vol. 17, no. 5, pp. 386-399, May 1998.
[46] Z. Zhao, B. Pouya, and N.A. Touba, "BETSY: Synthesizing Circuits for a Specified BIST Environment," *Proc. Int'l Test Conf.,* pp. 144-153, Oct. 1998.
[47] D.K. Pradhan and M. Chatterjee, "GLFSR—A New Test Pattern Generator for Built-In-Self-Test," *IEEE Trans. Computer-Aided Design,* vol. 18, no. 2, pp. 238-247, Feb. 1999.