On Using Machine Learning for Logic BIST

Christophe FAGOT Patrick GIRARD Christian LANDRAULT

Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier,
UMR 5506 UNIVERSITE MONTPELLIER II / CNRS
161 rue Ada, 34392 Montpellier Cedex 05, FRANCE.
Email: girard@lirmm.fr http://www.lirmm.fr/~w3mic

Abstract

This paper presents a new approach for designing test sequences to be generated on–chip. The proposed technique is based on machine learning, and provides a way to generate efficient patterns to be used during BIST test pattern generation. The main idea is that test patterns detecting random pattern resistant faults are not embedded in a pseudo–random sequence as in existing techniques, but rather are used to produce relevant features allowing to generate directed random test patterns that detect random pattern resistant faults as well as easy–to–test faults. A BIST implementation that uses a classical LFSR plus a small amount of mapping logic is also proposed in this paper. Results are shown for benchmark circuits which indicate that our technique can reduce the weighted or pseudo–random test length required for a particular fault coverage. Other results are given to show the possible trade off between hardware overhead and test sequence length. An encouraging point is that results presented in this paper, although they are comparable with those of existing mixed–mode techniques, have been obtained with a machine learning tool not specifically developed for BIST generation and therefore may significantly be improved.

1. Introduction

An efficient test pattern generator which guarantees complete fault coverage while minimizing test application time, area overhead, and test data storage is essential for any successful built–in self test (BIST) scheme. Many different generation schemes have been proposed to accomplish various tradeoffs between these parameters [1,3,22]. The solutions range from pseudo–random techniques that do not use any storage but take a long application time and often do not detect some faults to deterministic techniques that may require significant test data storage but achieve complete fault coverage in a relatively short time.

Mixed–mode test pattern generation is an attractive alternative to the above scenarios. It uses pseudo–random patterns to cover easy–to–test faults and deterministic patterns to target the remaining hard–to–test faults. As opposed to other approaches, such as test point insertion [23], mixed–mode techniques can reach complete fault coverage without imposing circuit modifications and causing performance degradation. However, storing compressed deterministic patterns in a ROM often requires a prohibitive amount of hardware overhead.

To solve this problem, a number of mixed–mode BIST architectures have been proposed in the recent literature. Some of them are based on reseeding an LFSR to efficiently encode deterministic patterns during application of the pseudo–random test sequence [32,13,14]. Others are based on fixing certain bit positions to specific logic values during the test such that deterministic test cubes are imbedded in the pseudo–random sequence [27,26,2,8]. The last ones use a bit–flipping function such that pseudo–random patterns that do not detect faults are modified just at a few bit positions to be compatible with a precomputed deterministic pattern [31]. Advantages and limitations of these techniques will be discussed in section 2 of this paper.

The new test pattern generation (TPG) technique presented in this paper is different from existing mixed–mode techniques. The proposed technique uses the concept of machine learning developed in Artificial Intelligence, and provides a way to generate efficient patterns to be used during BIST test pattern generation. The main idea is that test patterns detecting random pattern resistant (r.p.r.) faults are not directly embedded in a pseudo–random sequence as in existing techniques, but rather are used to produce relevant features allowing to generate directed random test patterns that detect r.p.r. faults as well as easy–to–test faults. Relevant features are represented by test cubes (test vectors with unspecified inputs), and only a few test cubes are needed to achieve 100% fault coverage with short test sequences. A BIST implementation of this technique has been carried out by using a classical LFSR combined with a combinational logic of low hardware overhead.

In the proposed generation process, the way to determine the test cubes is the following. A set of random test patterns is initially generated to detect easy–to–test faults. A deterministic ATPG is further used to detect remaining faults, considered as r.p.r. faults and also called

hard–to–test faults. This set of deterministic patterns is then used as input by a machine learning tool to represent positive examples from which the tool has to learn. Common characteristics among all these patterns are further identified by the learning tool so that it can provide strategic features on certain bit positions. These strategic features are represented by a limited set of test cubes (called "skeletals" in the original machine learning method) containing specified and unspecified bits. These test cubes are then utilized by a directed random test pattern generator to produce test vectors in which some bits are fixed at 0 or 1, and other bits remain unspecified.

The main feature of this technique is that it provides 100% fault coverage of detectable stuck–at faults with test sequences of length smaller than that of weighted or pseudo–random generation techniques. Another important feature is that test patterns can be generated with a BIST TPG of low silicon area cost because the number of test cubes to be stored on–chip is in most cases very low. A last point is that results obtained with this technique, although they are comparable with those of existing mixed–mode techniques [14,31], have been obtained with a machine learning tool not specifically developed for BIST generation and therefore may significantly be improved.

The rest of the paper is organized as follows. In the next section, we give an overview of the work done previously in the area of BIST test pattern generation. Next, we present the procedure that allows to obtain and evaluate the test cubes provided by the learning tool. The method to learn from instances of test patterns and the directed random generation process are detailed too. In section 4, a solution for the hardware implementation of the BIST TPG is proposed. All the experiments performed on ISCAS benchmark circuits are presented in section 5.

2. Motivation and previous work

Studies done using ***random testing*** techniques have shown fault coverage obtainable with the use of uniformly distributed random test patterns to be unacceptably low. It was observed that most practical circuits containing random pattern resistant (r.p.r.) structures are very hard to test using uniformly distributed random sequences. An example of such a structure is an n–input AND gate, for which the probability of detecting stuck–at one faults at the inputs as well as stuck–at zero fault at the output falls exponentially with the growth of n. This observation prompted an improvement to random testing methods where non–uniform probability distributions of test patterns were used and proved to be effective in numerous cases [3]. Random testing with non–uniform input signal distribution is called ***weighted random testing***, and the probabilities of input signals being equal to 1 are known as input weights.

Several procedures have been developed for calculating optimal weights [3,16,28,29]. Some of them are heuristic, based on circuit structure analysis [3,28]. Others are based on analytical calculation of fault detection probabilities and use optimization techniques to reach a maximum of these probabilities in the n–dimensional space of input weights [16,29]. Attempts were also made to obtain optimal weights by randomizing the set of functional test vectors for a circuit [24]. For many circuits, the results of weighted random testing are significantly better than those obtained with uniformly distributed random tests both in terms of fault coverage and test length. However, it was further observed that some structures in logic circuits are resistant even to weighted random testing. An example of such a structure is an n–input AND gate and an n–input OR gate which have their corresponding inputs connected [4].

Based on this experience, it was conjectured in [7,10,17,18,30] that high fault coverage can only be obtained by weighted random tests using multiple sets of weights. However, these methods require a considerable amount of additional hardware to be implemented and some of them add delay between the input flip–flops and the inputs to the circuit during normal operation.

The use of ***biased pseudo–random patterns*** [11] requires only a small amount of additional hardware, but similar to weighted random patterns with a single weight set, requires long test times for r.p.r. circuits.

Another widely investigated approach for Built–In Self Test concerns mixed–mode techniques. In such techniques, deterministic patterns are used to detect the faults that the pseudo–random patterns miss [9]. However, storing deterministic patterns in a ROM requires a large amount of hardware overhead. To solve this problem, a technique based on ***reseeding*** an LFSR was proposed in [15] to reduce the storage requirements. The LFSR used for generating the pseudo–random patterns is also used to generate deterministic ***test cubes*** by loading it with computed seeds. The number of bits that need to be stored is reduced by storing a set of seeds instead of a set of deterministic patterns. In [12,13], an improved technique is proposed that uses a multiple–polynomial LFSR for encoding a set of deterministic test cubes and reduce the number of bits that need to be stored. Although reseeding techniques are the most area–efficient solutions for Built–in self test, they may produce test sequences of excessive length for circuits with a high number of r.p.r. faults.

Another mixed–mode approach is presented in [26] in which deterministic test cubes are embedded in the pseudo–random sequence of bits. Logic is added at the serial output of the LFSR to alter the pseudo–random bit sequence so that it contains deterministic patterns that detect the r.p.r. faults. This is accomplished by "fixing" certain bits in the pseudo–random test sequence. The idea of

fixing certain bit positions to specific logic values during the test is called ***bit–fixing*** and was first used in [18,19]. As in [26], the results presented in [18,19] showed that for most circuits, a high fault coverage can be achieved in competitive test times. However, these methods require a large number of bit positions to be fixed [18,19] or complex routing paths to eliminate logic between the input flip–flops and the CUT [26], and therefore do not lead to efficient hardware solutions. Bit–fixing was also used in [2,27,8,25]. The test per clock BIST technique presented in [2] uses multiple idler register segments with selective bit–fixing driven by multiple biased pseudo–random pattern generators. Although it provides 100% fault coverage of detectable single stuck–at faults, the technique does not lead to efficient solutions in terms of hardware overhead cost. The issue of complex routing and high overhead cost was adressed by [8]. But to limit overhead, a compromise was made with respect to fault coverage. The result is that many of the r.p.r. faults remained undetected.The most recent bit–fixing technique is presented in [27], where it applies to built–in self test of circuits with scan. The technique performs wery well, except for circuits with a high number of r.p.r. faults that require a high number of deterministic test cubes. Authors proposed to consider bit correlation among the test cubes to minimize the number of bit–fixing sequences and the hardware overhead. Authors suggested again to combine the bit–fixing technique with reseeding techniques to achieve a better overhead reduction.

Another interesting mixed–mode approach, called ***Bit–flipping***, was proposed in [31]. Authors showed that rather then store deterministic patterns on the chip, it is sufficient to alter just a few bits of the generated pseudo random sequence. The efficiency of this scheme relies on the fact that only a relatively small amount of the generated pseudo–random patterns are useful for fault detection, and all the others are candidates for altering. Moreover, as a deterministic test pattern usually contains many unspecified bits, there is a very high chance that one of the useless patterns can be modified just at a few bit positions such that it is compatible with a precomputed deterministic pattern. Results are provided to show that this scheme represents the most area–efficient solution up to now.

The new 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 is different from existing mixed–mode techniques because it does not try to directly embed deterministic patterns in the pseudo–random sequence. The proposed technique is based on machine learning from features of deterministic patterns, and provides a way to generate efficient patterns from a *limited* number of test cubes. The main idea is that test patterns detecting r.p.r. faults are not embedded in a pseudo–random sequence as in existing techniques, but rather are used to produce relevant features allowing to generate directed random test patterns that detect r.p.r. faults as well as easy–to–test faults. The main features of this technique are that (1) it produces test sequences shorter than those of weighted or pseudo–random techniques while achieving the same stuck–at fault coverage, (2) results are comparable with those of existing mixed–mode techniques, and (3) the number of test cubes to be stored on–chip is most of the time very low.

3. The test generation procedure

Given a pattern generating circuit, a test length, and a fault coverage requirement, a procedure is described in this section for finding a limited set of test cubes that will be used to generate test patterns satisfying the fault coverage and test length requirements.

3.1 Description of the procedure

The first step of the procedure consists in generating a given number of random test patterns to detect easy–to–test faults (see Figure 1). Details will be given in section 5 about the way to obtain this number. Fault simulation with fault dropping is then performed on the circuit for the random patterns to see which faults are detected and which are not.

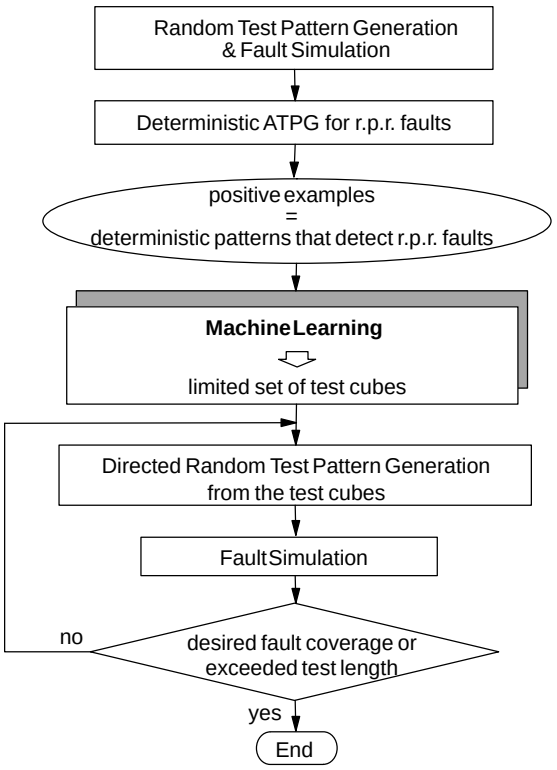

Figure 1: The flow of the test generation algorithm

The faults that are not detected after application of a given number of random patterns are classified as r.p.r. faults, and require deterministic test patterns to be detected. An ATPG tool is then used to determine these patterns, which form the

set of positive examples required by the machine learning tool. Positive examples are further used by the machine learning tool to produce relevant features of a positive test pattern. A positive test pattern is a test pattern that detects at least one r.p.r. fault and a number of easy–to–test faults. A relevant feature is a combination of the values of some bits (a test cube) which allows to predict with a good confidence level if any given test pattern is positive or not. As a test cube is a mapping of information learned from a subset of the deterministic patterns, the number of test cubes produced by the learning tool is always orders of magnitude smaller than the number of deterministic patterns. A directed random test pattern generation is then carried out from the limited set of test cubes, where the new test vectors have a structure that is consistent with the cubes provided by the learning tool. Details on the test generation procedure are given in the following subsection. The last step of the procedure is to simulate the generated test patterns so as to evaluate the fault coverage and the test sequence length. If the desired fault coverage is not achieved and the maximum length of the test sequence is not exceeded, then a set of new test vectors is generated from the same test cubes. This process continues until the fault coverage and test length requirements are satisfied.

In the following, a test cube is a combination of *n* variables, where *n* is the number of primary inputs to the circuit under test. A test cube is represented by a vector in $\{0,1,X\}^n$ where a 0 (1) indicates that the variable has the value '0' ('1') in the cube, and an 'X' indicates that the variable does not appear in the cube. The number of test patterns with all bits specified that can be generated from a test cube is 2^k where *k* represents the number of X's in the test cube.

3.2 Obtaining the test cubes

In order to achieve a maximum fault coverage with a reasonable test sequence length, we use a machine learning technique providing a way to direct the generation of test patterns during on–chip test generation. This technique is called CNN for Conceptual Nearest Neighbors [20], and is derived from ANNA which has been successfully applied in various domains [21]. This technique provides a set of test cubes (called "skeletals" in the original version of CNN) having a high probability to detect r.p.r. faults. Some bits in these skeletals are fixed at 0 or 1, while the others remain unspecified (X value). From this set of skeletals, a directed random test pattern generation is then applied to the circuit, that minimizes the number of test patterns required to achieve the desired stuck–at fault coverage.

CNN's input is a learning set E composed of positive examples provided by the ATPG tool. In the first step, CNN produces all possible skeletals by pairwise comparison of every examples. For instance, comparing the vectors (0 1 0 1 0) and (0 1 1 0 0) will produce the skeletal (0 1 X X 0). In the second step, CNN evaluates the quality of the skeletals found during the first step. For this purpose, two quantities are determined for each skeletal: the number of positive examples recognized (denoted α) and the number of positive examples for which the skeletal is the most representative among all possible skeletals (denoted β). A constraint in this evaluation is that each positive example must have at least one representative skeletal in the final set of selected test cubes. Considering the first number, a skeletal S recognizes an example description if this description contains the features of S. For instance, skeletal (X 1 0 X 1) recognizes vector (0 1 0 1 1). Considering the second number, we can better explain with an example how it is calculated. Let us consider the set of deterministic patterns (positive examples) given in Figure 2 with the partial list of skeletals found after pairwise comparison of every examples.

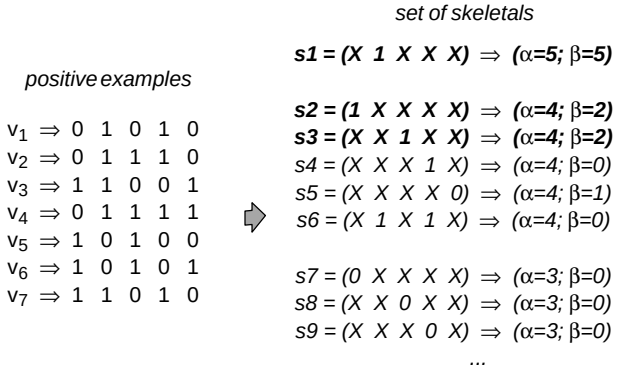

Figure 2: Result of the machine learning tool

Skeletals are ordered according to their number of positive examples recognized, showing that, for instance, skeletal s1 recognizes example vectors v_1, v_2, v_3, v_4 and v_7. To calculate β for each skeletal, the learning tool proceeds as follows. As s1 covers the largest number of positive examples (it has the highest α value), the five vectors v_1, v_2, v_3, v_4 and v_7 select s1 to be their representative skeletal during the directed random test pattern generation. Skeletal s1 is therefore the most representative skeletal for vectors v_1, v_2, v_3, v_4 and v_7 (β=5). The machine learning consists now in selecting a representative skeletal for not yet assigned vectors v_5 and v_6. Five skeletals each recognize four vectors in the learning set (α=4). However, only s2 or s3 can be selected by both vectors v_5 and v_6 to be their representative skeletal in the next test generation (s5 recognizes only vector v_5, s4 and s6 recognize neither v_5 nor v_6). Hence, the set of relevant skeletals in this example is composed of s1 and either s2 or s3. Note that (1) this set of relevant skeletals is minimum and covers all the vectors of the initial learning set, (2) β can also be defined as the number of positive examples recognized by the skeletal and not yet recognized by the skeletals already selected for test generation.

In the final step, CNN outputs a set of *p* skeletals explaining which are the combinations of the bit values that have a high probability to detect r.p.r. faults. Using the *p* found skeletals, we perform a new directed random generation of test patterns. For each found skeletal, we generate *k* test patterns which are recognized by this skeletal. Values specified in the skeletal are conserved, while the others (X's) are randomly chosen with probability 0.5 to be 0 or 1. We then obtained a new set of $N=k.p$ test vectors which are simulated by a fault simulator. The process of generating new directed random test patterns is iterated until the maximum or the desired stuck–at fault coverage has been achieved. Although the directed random test sequences may be efficient in terms of fault coverage and test length, their main feature is that they can be produced by a low–cost TPG in a BIST environment. This point is addressed in the following section.

Note: As illustrated in section 5 of this paper, the number of test patterns needed to achieve 100% fault coverage may sometimes be considerable with the CNN program described in this section. For this reason, an extended version of the program in which the number of selected skeletals may be greater than *p* (where *p* is the minimum number of skeletals that is needed to cover all positive examples) has been experimented. In this version, skeletals with a number of fixed bits greater than a given threshold are selected in addition to the *p* skeletals for insertion in the set of test cubes to be used during test generation. This threshold is chosen to ensure the best tradeoff between minimizing the number of test cubes (i.e. minimizing hardware overhead) and satisfying the test length and fault coverage requirements. Results obtained in this way are given in section 5.

4. Architecture of the BIST TPG

As claimed at the beginning of this paper, the number of test cubes required for test pattern generation in each circuit is generally very low (see results given in section 5). For this reason, a solution to materially generate test patterns has been to use a low–cost mapping logic in the BIST hardware environment.

After a set of test cubes has been determined such that the test length and fault coverage requirements are satisfied, an implementation of the BIST test pattern generator can be performed. As in any BIST hardware implementation, the main objective is to implement the test generator with a low cost of area overhead. In the proposed technique, no storage of deterministic patterns, seeds, characteristic polynomials, or weight sets is required. In fact, no additional sequential logic needs to be added. As illustrated in Figure 3, a purely combinational logic block is added between the pattern generator and the CUT to map the original set of patterns into a transformed set of patterns that provides the desired fault coverage. A key issue in such a BIST scheme is to design the mapping logic so that it uses only a small number of gates. This is accomplished as explained in the following. Note that the pattern generator we used in our BIST scheme is a classical primitive polynomial LFSR in which the number of stages is imposed by the maximum number of X's in the test cubes.

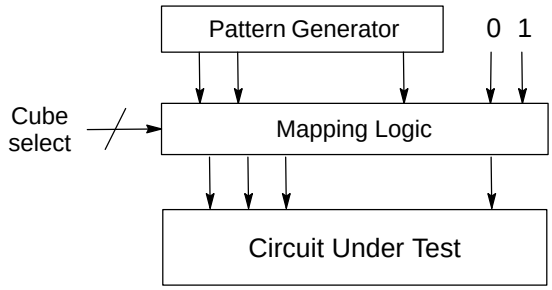

Figure 3: Block diagram for generating patterns

The mapping logic has inputs coming from the pattern generator and outputs connected to the CUT. A *cube selection input* is also connected to the logic to allow switching from a test cube to another during testing. The internal structure of the mapping logic can be viewed as composed of as many cells as the number of inputs to the CUT (see Figure 4). In fact, this number can be drastically reduced after logic optimization as explained at the end of this section.

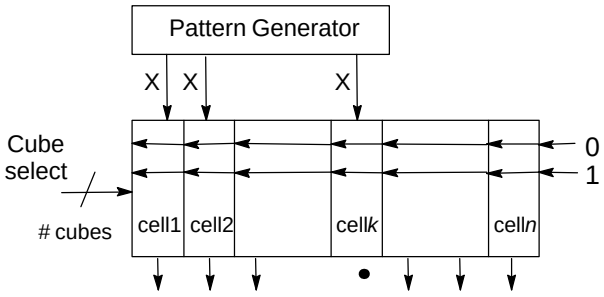

Figure 4: Overview of the mapping logic

During testing, each test pattern applied to the CUT has a structure that is consistent with the considered test cube, so that each bit in the pattern can take the value '0', '1' or 'X'. Don't care values (X's) are provided by the LFSR, '0' and '1' are provided by the supply and ground of the circuit. The *cube select* input allows to select the test cube from which a number of test patterns will be generated. Each cell of the mapping logic has therefore three inputs and one output, plus a limited number of selection inputs.

The gate implementation of the mapping logic can be better explained with an example. Assume a circuit consisting of six primary inputs for which three test cubes have been constructed from the machine learning technique described in the previous section (see Figure 5). Cell 1 of the mapping logic has to implement the fact that bit 1 has the

value ’1’ in Cube 1, the don’t care value in Cube 2 and the value ’0’ in Cube 3. Four NAND gates are needed for this purpose as illustrated in Figure 5 (note that inverting gates are used to minimize the number of transistors per gate and, hence, the area overhead). Bit 2 at the output of cell 2 in the mapping logic is always fixed at the don’t care value, thus resulting in a logic cell with no gate. Bit 5 at the output of cell 5 is fixed at ’X’ when Cube 1 or Cube 3 are selected, and is fixed at ’1’ when Cube 2 is selected. Three NAND gates and one OR gate are needed in this case.

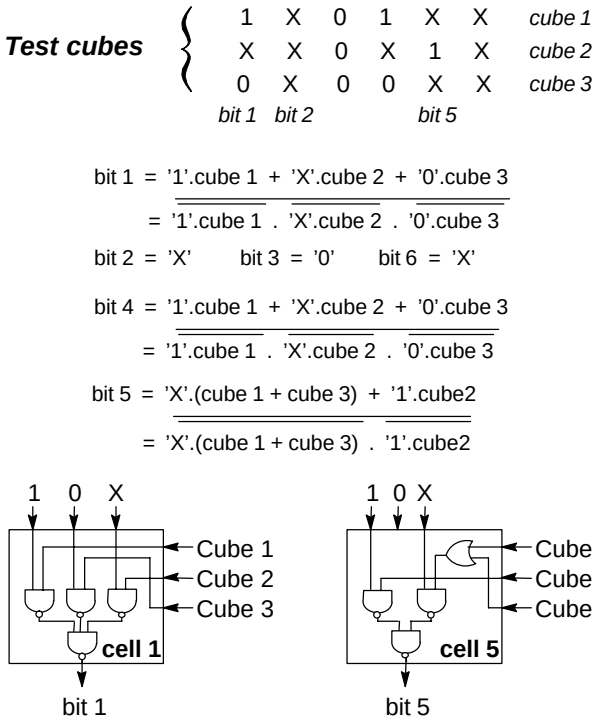

Fig. 5: Design example of cells in the mapping logic

From a general point of view, each cell in the mapping logic is design as described in the previous example. After that, the circuit is synthesized with a logic synthesis tool to minimize the silicon area of the mapping logic and to control the delay between the input flip–flops and the CUT. As the number of test cubes required for test generation in each circuit is most of the time very low (see results given in section 5), the number of additional gates in the mapping logic is never prohibitive. Moreover, the most important feature in this scheme is that this number can be drastically reduced after logic optimization. For example, bits 1 and 4 in Figure 5 are both fixed at ’1’ when Cube 1 is selected for test generation. Hence, one NAND gate can be saved in one of the two cells, thus reducing the total number of gates needed. This reduction is obviously applicable a lot of times since all the cells in the mapping logic have the same ’1’ and ’0’ inputs. Note that this type of reduction does not apply for input ’X’ so as to avoid having the same random logic value on several bit positions of a selected test cube. In order to illustrate the effect of logic optimization on the mapping logic, the complete logic for the example detailed in Figure 5 is shown in Figure 6. Remember that the number of outputs of the LFSR is given by the maximum number of X’s in each test cube (the third ’X’ at the logic input has two connections because ’X’ for bit 4 is used when Cube 2 is selected, while ’X’ for bit 5 is used when Cube 1 and Cube 3 are selected).

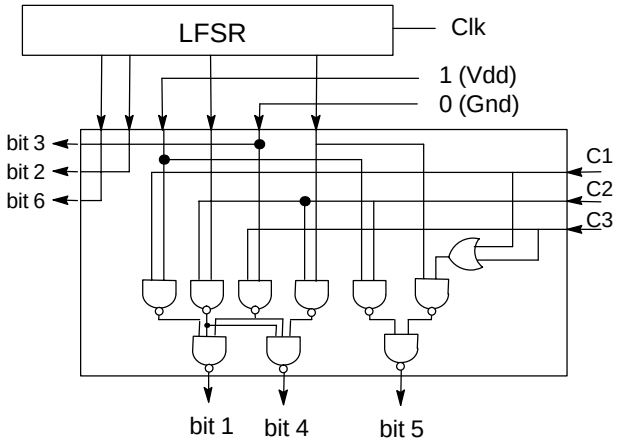

Figure 6: The mapping logic after optimization

An obvious concern about inserting gates between the input flip–flops and the CUT is that the delay through the circuit will be a problem. As in [27], a solution to eliminate this is to add multiplexers to bypass the mapping logic during the normal mode of operation. In the same way, a control logic is needed to activate one test cube at a time and generate *k* vectors from each test cube. This control logic can be reduced by using existing counters and clock signals in the circuit as in [27].

5. Experimental results

Experimental evaluation of the proposed method was conducted using the standard sets of combinational ISCAS’85 benchmark circuits [5] and sequential ISCAS’89 benchmark circuits [6] that contain r.p.r. faults. The flip–flops in the ISCAS’89 circuits were considered scannable and only detectable faults were considered in fault coverage calculations.

Table 1 presents the results of experimental runs carried out from the machine learning method described in section 3. The first column gives circuit name (letter ”c” after ISCAS’89 circuits indicates that they were converted into combinational circuits) and the second column contains the number of detectable faults (#fts) in each circuit. Columns 3 and 4 contain the number of gates required to implement the mapping logic (#add) and the percentage of area overhead of the BIST TPG (%area). This percentage is expressed in terms of gate count and represents the ratio between the number of additional gates in the mapping logic and the total number of gates in the circuit. An important feature

concerning the number of additional gates is that this number is a maximum since it was calculated before logic optimization (logic optimization is not yet implemented in the current version of the program). Since the mapping logic required to implement a set of test cubes can be correlated in certain bit positions to minimize hardware, important reduction can therefore be obtained after logic optimization as illustrated with the example in section 4 (see Figure 5). In this example, the maximum number of additional gates as counted in Table 1 is 25, while it is 10 after logic optimization (see Figure 6). Results given in Columns 3 and 4 can therefore be drastically reduced. Moreover, it is important to note that this percentage decreases when the size of the circuit increases.

CIRCUIT		AREA		100%MinC			
name	#fts	#add	%area	#cub	#fix	#vects	CPU
c432	520	6	3.75	2	1	752	0.12
c499	750	12	5.94	3	1	718	0.14
c880	942	20	5.22	2	4	3134	0.88
c1355	1566	62	11.0	6	3.8	2666	0.51
c1908	1870	12	1.36	3	1	4104	0.62
c3540	3291	26	1.56	2	5.5	7295	2.50
c6288	7710	15	0.62	3	1.33	196	0.03
s208c	215	18	19.0	2	3.5	1525	0.16
s344c	342	9	5.62	2	1.5	594	0.07
s382c	399	18	11.0	3	1.66	420	0.05
s420c	430	34	17.0	2	7.5	>50K	–
s444c	460	25	14.0	3	2.66	574	0.07
s510c	564	16	7.58	2	3	5190	0.62
s526c	554	48	25.0	6	2.5	7981	0.97
s641c	463	6	1.58	2	1	>50K	–
s713c	543	25	6.36	3	2.66	>20K	–
s820c	850	22	7.61	2	4.5	>30K	–
s838c	857	82	21.0	2	19.5	>50K	–
s953c	1033	6	1.51	2	1	>30K	–
s1196c	1240	6	1.13	2	1	>30K	–
s1238c	1283	9	1.77	2	1.5	>50K	–
s1423c	1501	6	0.91	2	1	>30K	–
s1488c	1486	12	1.84	3	1	7880	0.54
s1494c	1494	12	1.85	3	1	6222	0.43

Table 1: Results of directed random generation

Columns from the fifth to the last contain the results of directed random test pattern generation under the condition that initial random generation stops when the number of random patterns that do not detect any fault becomes equal to the number of random patterns that have detected faults. Columns 5 and 6 contain the number of test cubes used for directed random generation (#cub) and the average number of bits fixed in each cube (#fix). The number of test cubes represents the minimum number of cubes that is needed to exhaustively cover the set of positive examples (deterministic patterns). Columns 7 reports the number of directed random patterns that are needed to achieve 100% stuck–at fault coverage with a minimum number of test cubes (100%MinC), and column 8 gives the CPU time in SPARC 5 seconds for the entire process of generating test patterns. A first comment is that for some sequential benchmark circuits (s420c, s641c, s820c, ...), more than 30000 patterns are needed to achieve 100% fault coverage. The reason is that the minimum set of test cubes does not allow to easily produce test patterns for each r.p.r. fault, and that additional test cubes are necessary. By increasing the number of cubes from 2 to 6 for circuit s820c for example, the 100% fault coverage is achieved with 16847 patterns instead of more than 50000 (see Table 2, colum 100%Trade–off). Results for which shorter test sequence lengths have been obtained with a greater number of test cubes and bits fixed are listed in Table 2. Note that results for circuits with a large number of inputs are not provided because the current version of CNN [21] is unable to deal with test patterns having a number of bits greater than 200. An extended version of CNN is currently under implementation to solve this problem.

For the results listed in Table 1, experiments were performed in the following way. Each circuit was first simulated with uniformly distributed random test patterns until the number of patterns that do not detect any fault becomes equal to the number of random patterns that have detected faults. After that, test patterns for the remaining undetected faults were generated by a deterministic test pattern generator (GENTEST from System–HILO of Genrad) to form the learning set of positive examples in the subsequent. The number of deterministic test patterns depends on the size of the circuit and obviously on the random pattern testability of the circuit. This number ranges from 20 for smallest circuits to 150 for the biggest (s1423). In the next step, a minimum number of test cubes was produced by the machine learning tool in such way that every example in the learning set is recognized by at least one test cube. The CPU time taken by the process of generating test cubes (from the learning tool) is not reported in the table, but is less than one second for each circuit. A directed random test pattern generation was then carried out by using the selected test cubes with a classical LFSR to produce test patterns that detect random as well as r.p.r. faults. Although additional hardware is required for the mapping logic, it is important to note that smaller LFSR's than in pseudo–random pattern generation are needed in the

proposed technique. This point has to be taken into consideration when looking at the amount of hardware overhead. A simulation process (HISIM from System–HILO) was finally executed to evaluate the fault coverage and start again the test generation if necessary. Results reported in Table 1 show that using only two or three test cubes with a very small number of bits fixed allows most of the time to achieve 100% stuck–at fault coverage with test sequences of acceptable length.

CIRCUIT			100%MinC		100%Trade−off		
name	#fts	RAND	#cub	#vects	#cub	#fix	#vects
c432	520	1K	2	752	5	7.4	518
c499	750	754	3	718	7	3.7	480
c880	942	15K	2	3134	8	9.5	1593
c1355	1566	10K	6	2666	6	3.8	2666
c1908	1870	6112	3	4104	29	5.03	2827
c3540	3291	15K	2	7295	2	5.5	7295
c6288	7710	1K	3	196	3	1.33	196
s208c	215	2233	2	1525	14	13.03	740
s344c	342	1K	2	594	3	4	177
s382c	399	1K	3	420	6	2	276
s420c	430	1.1M	2	>50K	3	14.33	17488
s444c	460	810	3	574	6	3.5	267
s510c	564	12K	2	5190	2	3	5190
s526c	554	> 100K	6	7981	6	2.5	7981
s641c	463	1.0M	2	>50K	4	8.75	16711
s713c	543	> 100K	3	>20K	42	34	4090
s820c	850	> 100K	2	>30K	6	5.5	16847
s838c	857	> 100M	2	>50K	57	59.25	5415
s953c	1033	> 100K	2	>30K	4	4.5	15800
s1196c	1240	> 100K	2	>30K	71	17.07	13523
s1238c	1283	> 100M	2	>50K	3	5.33	19600
s1423c	1501	> 100K	2	>30K	5	10	17122
s1488c	1486	8950	3	7880	5	2	5289
s1494c	1494	20K	3	6222	6	2	3931

Table 2: Results of directed random generation

Another set of experiments was conducted to study the effect of the number of test cubes and bits fixed on the test efficiency. Experimental results are summarized in Table 2 for two different numbers of test cubes in each circuit. The first two columns give the circuit name and the number of detectable faults in each circuit. The third column reports the number of random patterns that is required to achieve 100% stuck–at fault coverage. The two groups of results that follow give a comparison of the test sequence length needed to achieve 100% fault coverage with respect to the number of test cubes used during directed random generation. A first comment in observing these results is that they indicate that our technique can drastically reduce the random pattern test length when the number of test cubes is increased from 2.5 to 16 in average. A counterpart of this reduction is that the amount of mapping logic required for 100% fault coverage goes up when the test length decreases. Nevertheless, one can see that it is very easy to trade off between test length, fault coverage and hardware overhead.

Compared with a number of existing methods, the proposed solution requires no additional flip–flops, only combinational logic between the LFSR and the CUT. A direct comparison between our results and those of existing mixed–mode techniques [27,31] is not provided in this paper. Nevertheless, it is not unfair to claim that they are in the same order of magnitude concerning the test length required to reach 100% fault coverage. Concerning hardware overhead, any objective comparison is possible before logic optimization has been implemented in the current version of the BIST hardware generator. We can however report that they are also in the same order of magnitude than those given in [27]. An encouraging point is that results presented in this paper, although they are comparable with those of existing mixed–mode techniques, have been obtained with a machine learning tool not specifically developed for BIST generation and therefore may significantly be improved by modifying the learning process and its parameters.

6. Conclusion

In this paper, the problem of improving fault coverage during pseudo–random pattern testing was thought of as transforming a pseudo–random pattern set into a better one. A machine learning technique has been used for this purpose, which provides a way to generate efficient patterns to be used during BIST test pattern generation. The main idea is that test patterns detecting random pattern resistant faults are not embedded in a pseudo–random sequence as in existing techniques, but rather are used to produce relevant features allowing to generate directed random test patterns that detect random pattern resistant faults as well as easy–to–test faults. A BIST implementation of the TPG is also proposed in this paper, that allows to achieve area–efficient solutions by trade off between the number of test cubes and the test sequence length.

One way to improve upon the results presented in this paper may be to use another version of CNN in which machine learning is carried out from positive and negative examples. Test cubes could be produced by learning from both deterministic and random patterns in this case, thus allowing to improve the performance of directed random

test pattern generation. This idea is currently being investigated.

References

[1] V.D. Agrawal, C.R. Kime and K.K. Saluja, "***A Tutorial on Built−In Self Test. Part 1: Principles***", IEEE Design & Test of Computers, pp. 73−82, March 1993.

[2] M.F. AlShaibi and C.R. Kime, "***MFBIST: A BIST Method for Random Pattern Resistant Circuits***", IEEE International Test Conference, pp. 176−185, 1996.

[3] P. Bardell, W. McAnney and J. Savir, "***Built−In Test for VLSI. Pseudorandom Techniques***", John Wiley & Sons, New York, 1987.

[4] M. Bershteyn, "***Calculation of Multiple Sets of Weights for Weighted Random Testing***", IEEE International Test Conference, pp. 1031−1040, 1993.

[5] F. Brglez and H. Fujiwara, "***A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran***", IEEE International Symposium on Circuits and System, pp. 663−698, 1985.

[6] F. Brglez, F.D. Bryan and K. Kozminski, "***Combinational Profiles of Sequential Benchmark Circuits***", IEEE International Symposium on Circuits and System, pp. 1929−1934, 1989.

[7] F. Brglez, C. Gloster and G. Kedem, "***Built−In Self Test with Weighted Random Pattern Hardware***", IEEE International Conference on Computer Design, pp. 161−166, 1990.

[8] M. Chatterjee and D. Pradham, "***A Novel Pattern Generator for Near−Perfect Fault Coverage***", IEEE VLSI Test Symposium, pp. 417−425, 1995.

[9] C. Dufaza, H. Viallon and C. Chevalier, "***BIST Hardware Generator for Mixed Test Scheme***", IEEE European Test Conference, pp. 424−430, 1995.

[10] E. Eichelberger, E. Lindbloom, J. Waicukauski and T. Williams, "***Structured Logic Testing***", Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[11] M. Gruetzner and C.W. Starke, "***Experience with Biased Random Pattern Generation to Meet the Demand for a High Quality BIST***", IEEE European Test Conference, pp. 408−417, 1993.

[12] S. Hellebrand, S. Tarnick, J. Rajski and B. Courtois, "***Generation of Vector Patterns Through Reseeding of Multiple−Polynomial Linear Feedback Shift Registers***", IEEE International Test Conference, pp. 120−129, 1992.

[13] 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 Transactions on Computers, Vol.44, N°2, pp. 223−233, 1995.

[14] S. Hellebrand, B. Reeb, S. Tarnick and H.J. Wunderlich, "***Pattern Generation for a Deterministic BIST Scheme***", IEEE International Conference on CAD, pp. 88−94, 1995.

[15] B. Koenemann, "***LFSR−Coded Test Patterns for Scan Designs***", IEEE European Test Conference, pp. 237−242, 1991.

[16] R. Lisanke, F. Brglez and A. Degeus, "***Testability−Driven Random Test Pattern Generation***", IEEE Transactions on CAD, Vol.6, N°6, pp. 1082−1087, November 1987.

[17] F. Muradali, V. Agarwal and B. Nadeau−Dostie, "***A New Procedure for Weighted Random Built−In Self−Test***", IEEE International Test Conference, pp. 660−669, 1990.

[18] S. Pateras and J. Rajski, "***Cube−Contained Random Patterns and their Application to the Complete Testing of Synthesized Multi−Level Circuits***", IEEE International Test Conference, pp. 473−482, 1991.

[19] I. Pomeranz and S.M. Reddy, "***3−Weight Pseudo−Random Test Generation Based on a Deterministic Test Set for Combinational and Sequential Circuits***", IEEE Trans. on CAD, Vol.12, pp. 1050−1058, July 1993.

[20] J.C. Regin, "***Development of Algorithmic Tools for Artificial Intelligence. Application to Organic Chemistry***", PhD Thesis, University of Montpellier (France), 1995.

[21] J.C. Regin, O. Gascuel and C. Laurenço, "***Machine Learning of Strategic Knowledge in Organic Synthesis from Reaction Databases***", Proc. of Euro. Conf. on Computational Chemistry, pp. 618−623, 1994.

[22] J. Savir and W. McAnney, "***A Multiple Seed Linear Feedback Shift Register***", IEEE Trans. on Computers, Vol.41, N°2, February 1992.

[23] B.H. Seiss, P.M. Trousborst and M.H. Schulz, "***Test Point Insertion for Scan−Based BIST***", IEEE European Design & Test Conference, pp. 253−262, 1991.

[24] F. Siavoshi, "***WTPGA: A Novel Weighted Test−Pattern Generation Approach for VLSI Built−In Self Test***", IEEE International Test Conference, pp. 256−262, 1988.

[25] N.A. Touba and E.J. McCluskey, "***Synthesis of Mapping Logic for Generating Transformed Pseudo−Random Patterns for BIST***", IEEE International Test Conference, pp. 674−682, 1995.

[26] N.A. Touba and E.J. McCluskey, "***Transformed Pseudo−Random Patterns for BIST***", IEEE VLSI Test Symposium, pp. 2−8, 1995.

[27] N.A. Touba and E.J. McCluskey, "***Altering a Pseudo−Random Bit Sequence for Scan−Based BIST***", IEEE International Test Conference, pp. 167−175, 1996.

[28] J. Waicukauski, E. Lindbloom, E. Eichelberger and O. Forlenza, "***A Method for Generating Weighted Random Test Patterns***", IBM Journal of Research and Development, Vol.33, N°2, pp. 149−161, March 1989.

[29] H.J. Wunderlich, "***On Computing Optimized Input Probabilities for Random Tests***", ACM Design Automation Conference, pp. 392−398, 1987.

[30] H.J. Wunderlich, "***Multiple Distributions for Biased Random Test Patterns***", IEEE Transactions on CAD, Vol.9, N°6, pp. 584−593, June 1990.

[31] H.J. Wunderlich and G. Kiefer, "***Scan−Based BIST with Complete Fault Coverage and Low Hardware Overhead***", IEEE European Test Workshop, pp. 60−64, 1996.

[32] N. Zacharia, J. Rajski and J. Tyszer, "***Transformed Decompression of Test Data Using Variable−Length Seed LFSRs***", IEEE VLSI Test Symposium, pp. 426−433, 1995.