SPaRe: Selective Partial Replication for Concurrent Fault-Detection in FSMs

Petros Drineas and Yiorgos Makris, *Member, IEEE*

***Abstract*—We discuss SPaRe: selective partial replication, a methodology for concurrent fault detection in finite state machine (FSMs). The proposed method is similar to duplication, wherein a replica of the circuit acts as a predictor that immediately detects errors by comparison to the original FSM. However, instead of duplicating the FSM, SPaRe selects a few prediction functions which only partially replicate it. Selection is guided by the objective of minimizing the incurred hardware overhead without compromising the ability to detect all faults, yet possibly introducing fault-detection latency. SPaRe is nonintrusive and does not interfere with the encoding and implementation of the original FSM. Experimental results indicate that SPaRe achieves significant hardware overhead reduction over both duplication and test vector logic replication (TVLR), a previously reported concurrent fault-detection method. Moreover, as compared to TVLR, SPaRe also reduces the average fault-detection latency for detecting all permanent faults.**

***Index Terms*—Concurrent test, finite state machine (FSMs).**

I. Introduction

ELECTRONIC circuits are employed in a wide variety of modern life activities, ranging from mission critical applications to simple commodity devices. As a result, circuit designers are faced with a broad spectrum of dependability, reliability, and testability requirements, which necessitate a range of concurrent test methods of various cost and efficiency levels. Concurrent test provides circuits with the ability to self-examine their operational health during normal functionality and indicate potential malfunctions. While such an indication is highly desirable, designing a concurrently testable circuit which conforms to the rest of the design specifications is not a trivial task. Issues to be addressed include the hardware cost and design effort incurred, potential performance degradation due to interaction between the circuit logic and the concurrent test logic, as well as the level of required assurance.

In this paper, we focus on finite state machines (FSMs) and we explore the tradeoffs between the aforementioned parameters, in order to devise a cost-effective, nonintrusive, concurrent fault-detection method. Nonintrusiveness implies that hardware may only be added in parallel to the FSM, which may not be modified. The additional logic is expected to detect all faults. Moreover, test has to be performed concurrently with the FSM operation and may not degrade its performance.

Manuscript received November 30, 2002; revised July 2, 2003.

P. Drineas is with the Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12181 USA.

Y. Makris is with the Departments of Electrical Engineering and Computer Science, Yale University, New Haven, CT 06520-8285 USA.

Digital Object Identifier 10.1109/TIM.2003.818733

Concurrent test is based on the addition of hardware that monitors the inputs and generates an *a priori* known property that should hold for the outputs. A property verifier is utilized to indicate any violation of the property, thus detecting circuit malfunctions. The simplest approach is to duplicate the circuit, imposing an identity property between the original output and the replica output, which may be simply examined by a comparator, as shown in Fig. 1(a). With the exception of common-mode failures [1], duplication will immediately detect all errors. However, it incurs significant hardware overhead that exceeds 100% of the cost of the original circuit.

While expensive schemes such as duplication detect all functional *errors*, simpler properties detecting only structural *faults* in a prescribed fault model exist. For example, the method proposed in [2], [3] reduces the functionality of the duplicate so that it only predicts the output of the circuit for a set of test vectors adequate to detect all stuck-at faults. As a result, the hardware overhead of the prediction logic is also reduced. However, a functional error is allowed to go undetected by this method until the structural fault that causes it is eventually detected. The concept of *fault-detection latency*, the time difference between appearance of an error and detection of the causing fault is thus introduced.

We propose a concurrent fault-detection method that further explores the tradeoff between hardware overhead and fault-detection latency. The paper is organized as follows: related work in concurrent test is reviewed extensively in Section II. SPaRe: selective partial replication, the proposed concurrent fault-detection method based on selective partial replication is described in Section III. Finally, experimental results assessing SPaRe in terms of hardware overhead, fault coverage, and fault-detection latency are provided in Section IV.

II. Related Work

To motivate the proposed methodology, we first examine related work in the areas of concurrent self-test (CST), concurrent error detection (CED), and concurrent fault detection (CFD) [4]–[6]. Almost all previous research efforts in these areas share the objective of being able to detect *all* faults. What typically distinguishes them, however, is their position within the tradeoff space between hardware overhead and fault-detection latency. Most approaches fall in one of the two ends of this space.

Toward the low end, low-cost CST approaches have been proposed for combinational circuits. C-BIST [7] employs input monitoring and existing off-line built-in self-test hardware to perform CST. While hardware overhead is very low, the method relies on an ordered appearance of all possible input vectors before a signature indicating circuit correctness can be calculated,

0018-9456/03$17.00 © 2003 IEEE

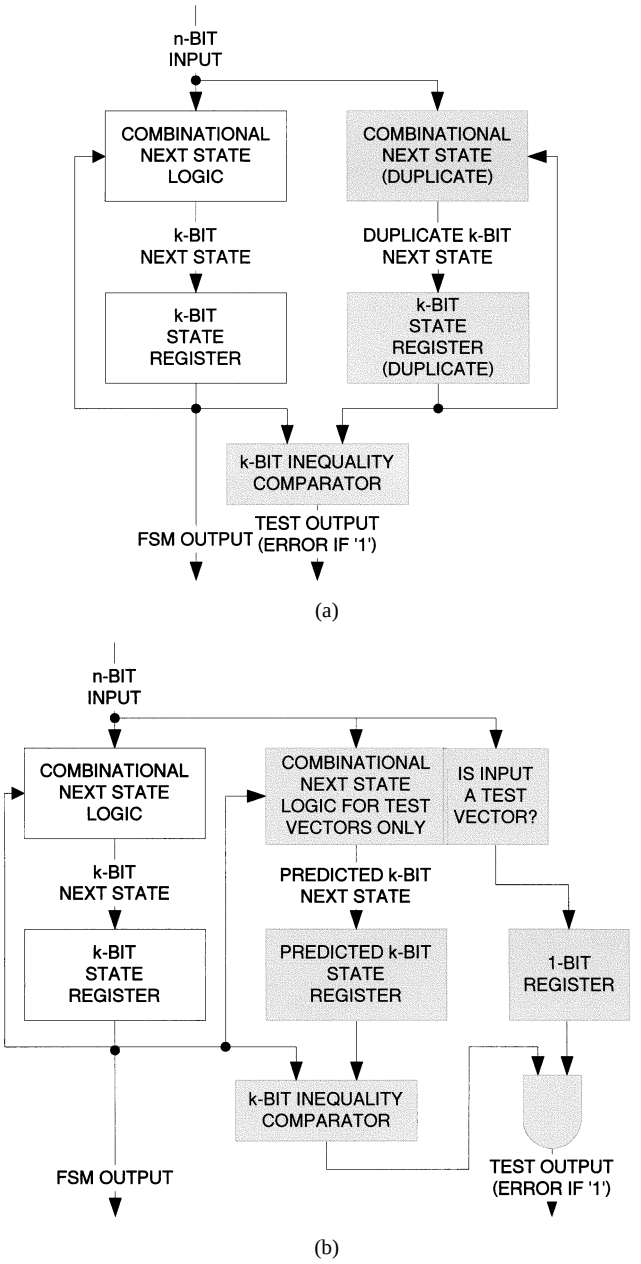

Fig. 1. (a) Duplication for CED. (b) Test vector logic replication for CFD.

resulting in very long fault-detection latency. This problem is alleviated in the R-CBIST method described in [8], where the requirement for a uniquely ordered appearance of all input combinations is relaxed at the cost of a small RAM. Nevertheless, all input combinations still need to appear before any indication of circuit correctness is provided.

Toward the high end, we find expensive CED methods for sequential circuits that check the circuit functionality at every clock cycle, therefore guaranteeing zero error detection latency. Reducing the area overhead below the cost of duplication typically requires redesign of the original circuit, thus leading to intrusive methodologies. Several redesign and resynthesis methods are described in [9]–[12], wherein parity or various unordered codes are employed to encode the states of the circuit. Limitations of [12], such as structural constraints requiring an inverter-free design, are alleviated in [13], where partitioning is employed to reduce the incurred hardware overhead. Utilization of multiple parity bits, first proposed in [14], is examined in [15] within the context of FSMs. All these methods render totally self-checking circuits and guarantee error detection with zero latency; on the down side, they are intrusive and relatively expensive. Nonintrusive CED methods have also been proposed for FSMs. The general algebraic model is introduced in [16]. implementations based on Bose–Lin and Berger codes are presented in [17] and [18], respectively. Finally, compression-based CED for combinational circuits is described in [19].

Among the few existing CFD approaches, a method that exploits properties of nonlinear adaptive filters is proposed in [20]. A similar technique is proposed in [21], where the frequency response of linear filters is used as an invariance property, achieving cost reduction but introducing fault-detection latency. A method exploiting transparency of RT-level components is described in [22]. Finally, a concurrent fault-detection method is proposed for combinational logic in [2] and extended to FSMs in [3]. Since this method, which we refer to as test vector logic replication (TVLR), is similar to the method proposed herein, we briefly describe it below.

A. Test Vector Logic Replication (TVLR)

In order to reduce the overhead of duplication, TVLR replicates only a portion of the original FSM, capable of detecting all faults in the design. More specifically, ATPG is performed on the combinational next state logic of the original FSM, treating the previous state bits as primary inputs, and a complete set of test vectors is obtained. These test vectors are subsequently synthesized into a prediction logic that generates the expected next state of the FSM when an input/previous state combination matches a test vector. The outputs of the prediction logic for input/previous state combinations that are not included in the test vector set are treated as *don't cares*. Thus, during synthesis, these outputs are chosen to minimize the required hardware. As a result, the prediction logic is less expensive than the duplicate next state logic.

However, since the output prediction logic will only generate the correct next state for input/previous state combinations included in the test vector set, the issue of *false alarms* needs to be addressed. More specifically, the concurrent test output should not be asserted during normal functionality, unless a fault has been detected. Therefore, an additional function is now required, indicating whether an input/previous state combination is a test vector. In the opposite case, the comparison outcome is not a valid indication of operational health of the FSM and is, therefore, masked through the AND gate in Fig. 1(b). Notice also that the predicted next state calculation is driven by the original FSM state register and not by the predicted state register, since the latter may not contain the correct value after an input/previous state combination that is not a test vector. The test vector set detects faults in the combinational next state logic. In order to also detect the faults in the state register, the comparison of the predicted next state is delayed by one clock cycle, similarly to [15]. If test responses comprise both a logic "1" and a logic "0" at every output, all faults in the state register will also be detected.

As shown in Fig. 1(b), TVLR is nonintrusive, since it leaves the original FSM intact. Despite the addition of one extra function (IS INPUT A TEST VECTOR), a considerable hardware overhead reduction is expected. On the down side, faults remain undetected until an appropriate test vector appears, thus introducing latency. However, given a sizeable test set, tests are performed frequently and low average fault-detection latency is expected.

III. PROPOSED METHOD

While TVLR trades off hardware for fault-detection latency, it is only one possible solution from a wide array of choices. Minimality of neither the incurred hardware overhead nor the introduced fault-detection latency can be ensured and superior methods may exist. In this section, we propose SPaRe, a CFD method based on selective partial replication. The key idea is presented through a small example, followed by an extensive description of the method.

A. Motivation

Consider the 2-bit up/down counter described in Fig. 2(a). If the objective is to detect all *errors* occurring during normal operation, the duplication-based CED scheme will achieve this by comparing the two outputs of the FSM to the two outputs of its replica. If, however, the objective is to detect all *faults*, allowing possible fault-detection latency, it is not necessary to compare both FSM outputs at every clock cycle. When we implemented the counter we noticed that by observing only one bit per state transition [shown in boldface in Fig. 2(a)], we are able to detect all faults in the FSM. Therefore, for the purpose of CFD, it is sufficient to replicate only partially the FSM, appropriately selecting which bits to predict for each state transition in order to detect all faults. Partial FSM replication implies cost reduction over duplication.

This observation is the basis for the SPaRe methodology which is shown on Fig. 2(b) for the 2-bit up/down counter. A combinational prediction logic is used to implement the 1-bit function that generates for each state transition the value shown in boldface in Fig. 2(a). This value is stored in a D flip-flop and compared to the corresponding bit of the FSM state register one clock cycle later. A MUX is used to drive the appropriate FSM output to the comparator. The select line of the MUX is driven by a function of the previous state and the inputs of the FSM, in this case a simple XOR between PS1 and PS0, delayed by one clock cycle. All faults in the next state logic are, thus, detected. Additionally, by postponing the comparison by one clock cycle, faults in the register are also detected.

B. SPaRe: Selective Partial Replication

The optimization objective of SPaRe is to minimize the output width of the prediction logic. Based on the observation that a subset of output bits per state transition is typically sufficient to detect all faults, SPaRe aims at identifying a minimal such set. The general version of SPaRe is depicted in Fig. 3. For every $(n+k)$-bit input combination, the prediction logic generates ℓ outputs that match a subset of ℓ out of the k FSM outputs. A selection logic is required to choose which

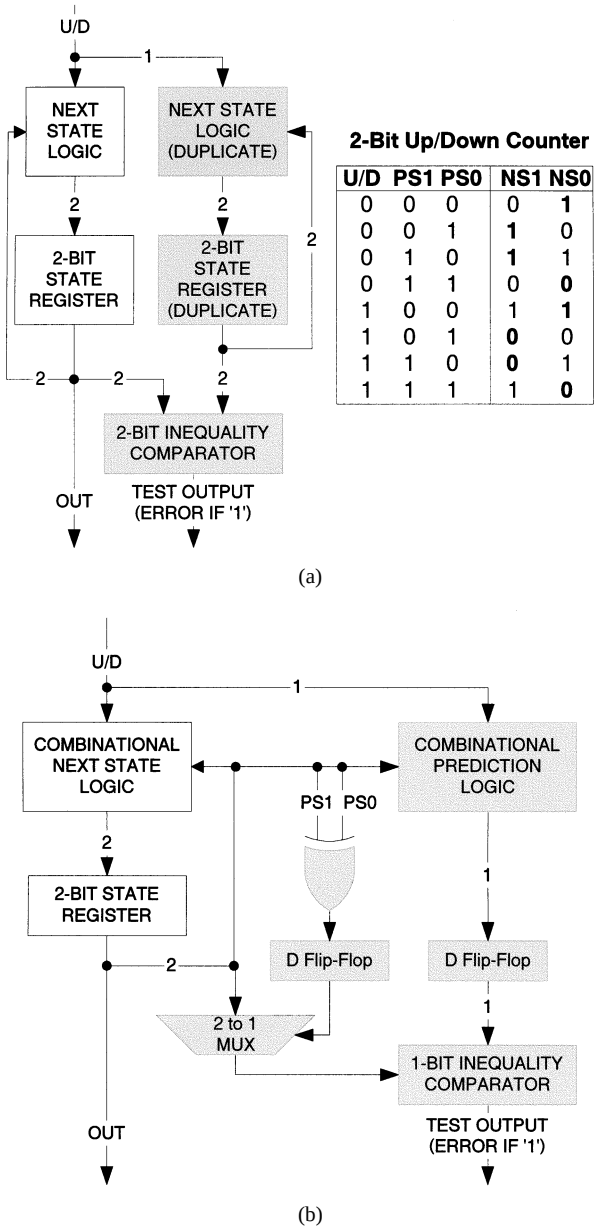

Fig. 2. (a) Duplication and (b) SPaRe on a 2-bit up/down counter.

FSM outputs to drive to the comparator for each $(n+k)$-bit input combination. Similarly to [15], comparison is delayed by one clock cycle to also detect faults in the state register.

Success of SPaRe relies on efficient solutions to two key issues: identification of appropriate output values to be replicated by the prediction logic and cost-effective selection of circuit outputs to which they should be compared. Regarding the first issue, an ATPG tool capable of generating all test vectors and reporting both the good and faulty circuit outputs for every fault in the combinational next state logic is required. This information indicates the faults that can be detected at each output for each input vector and may be used to construct a matrix similar to the one shown in Fig. 4. SPaRe seeks a set of columns that covers all faults, such that the maximum number of output bits to be observed for any input vector is minimized. However, the exact selection of columns impacts directly the cost of the selection logic. More specifically, since the prediction logic

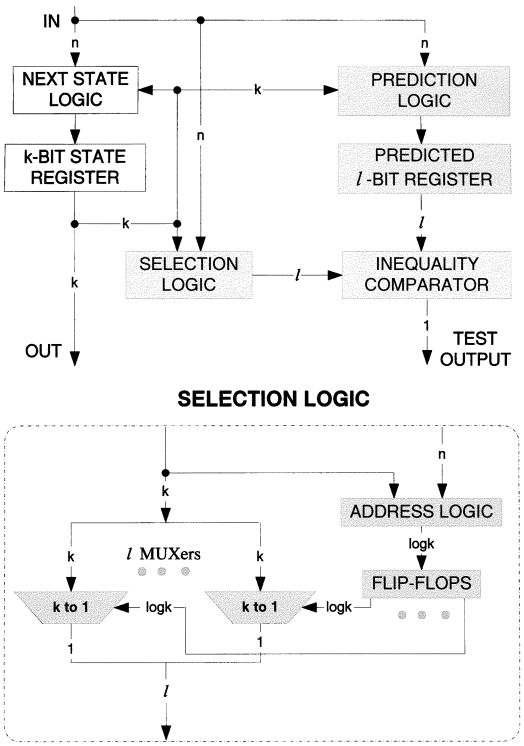

Fig. 3. SPaRe: selective partial replication.

only generates an ℓ-bit function, additional logic is necessary to select ℓ among the k circuit outputs to which the predicted ℓ bits will be compared. As shown in Fig. 3, this can be viewed as ℓ k-to-1 MUXs, each of which requires $\log k$ address bits. Therefore, if we allow any possible subset of size ℓ for every $(n+k)$-bit input combination, the address logic will generate $\ell \cdot \log k\,(n+k)$-input functions. Compared to duplication, SPaRe implements $k-\ell$ fewer $(n+k)$-input functions for the prediction logic, at the cost of implementing $\ell \cdot \log k\,(n+k)$-input functions and ℓ k-to-1 MUXs for the selection logic. The cost of the prediction logic is linear in ℓ; the cost of the selection logic, however, increases almost linearly in ℓ up to $\ell = k/2$, at which point it starts decreasing, eventually becoming zero at $\ell = k$. Therefore, if $\ell > k/(\log k + 1)$, the total size of the selection logic and the prediction logic exceeds the cost of duplication.

Imposing such an upper bound on ℓ could reduce fault coverage. Instead, we impose restrictions on the complexity of the address logic and by extension, on the acceptable solutions on the matrix of Fig. 4. SPaRe *eliminates the address logic all together*, therefore allowing that the $\log k$ select inputs of each multiplexer may only be driven directly by any $\log k$ out of the $(n+k)$ previous state and input bits. The form of acceptable solutions under this constraint, as well as a selection algorithm for identifying an appropriate set of columns that detects all faults are discussed next.

C. Selection Algorithm

We focus on the next state logic of the FSM, which, given a previous state and an input generates the next state. The inputs to this component are $I_1 \ldots I_k$ (the previous state) and $I_{k+1} \ldots I_{n+k}$ (the FSM inputs). The outputs of this component

	VECTOR 0			...	VECTOR 2^{n+k}-1		
	OUT_0	...	OUT_{k-1}	...	OUT_0	...	OUT_{k-1}
$Fault_1$	1						
$Fault_2$					1		1
. . .	. . .	. . .	. . .	...	. . .	. . .	. . .
$Fault_m$			1	...	1		

Fig. 4. Fault-detection matrix.

are $O_1 \ldots O_k$ (the next state). We denote the set of the 2^{n+k} possible previous state/input combinations by V.

Assume for the moment that we are given the matrix of Fig. 4, say A. We remind that SPaRe eliminates the ADDRESS LOGIC component of Fig. 3. For simplicity, we assume that 2 *specific* input bits, denoted by I_1 and I_2, drive *all* ℓ MUXs *and* also that ℓ is given. As a result, each MUX selects only among four of the FSM outputs; we remove this assumption promptly. Thus, the selection logic component of the diagram is fully specified. The selection logic splits the input vectors to four disjoint groups, each corresponding to a possible value for the pair $I_1 I_2 \in \{00, 01, 10, 11\}$; for all vectors in each group the *same* ℓ output bits are observed at the output of the selection logic. We denote the groups by G_1, G_2, G_3, G_4. We now state the problem formally: given A, the groups G_1, G_2, G_3, G_4, and ℓ pick ℓ output bits for each group so that the number of covered faults is maximized.

Prior to presenting an algorithm to solve the above problem, we revoke the simplifications we made earlier, starting with the assumption that ℓ is given. In practice, we seek the minimum ℓ for which we can detect all faults. Finding such an ℓ though is trivial; since $1 \leq \ell \leq k$, use binary search and solve the above problem $\log k$ times. We also assumed that the addressing bits (I_1 and I_2) were given; in practice, we try all possible 2-bit addressing schemes ($\approx (n+k)^2/2$). If we were to use $c > 2$ bits to feed the MUXs, the number of possible addressing schemes increases; however, since we only allow up to $\log k$ addressing bits, it is always a small number. We note that in this case the number of groups would increase to 2^c instead of 4. Finally, we assumed that A is fully constructed; obviously, for large circuits, time/space constraints render this assumption infeasible. Thus, in large circuits, the following strategy is employed: for every fault, generate a large number (say r) of input vectors detecting it. Thus, assuming m faults in our circuit, at most, mr vectors are generated. We subsequently identify the faults detected by each of these vectors, construct an $m \times mr$ matrix A' and solve the aforementioned problem in A' instead of A. Generally, A' admits less efficient solutions than A; as r increases the two solutions converge.

The size of the solution space for the above problem, assuming that ℓ and c are fixed, is $\binom{n+k}{c}\binom{k}{l}^{2^c}$. If ℓ and c are small constants, the size of the solution space is polynomial in both n and k. In practice, though, ℓ might be close to $k/2$, in which case the size of the solution space grows exponentially in k and it is impossible to explore it exhaustively. To understand its size, if $n = 2, k = 6, l = 3$, and $c = 2$, there are $4.5 \cdot 10^6$ possible solutions, while, if $c = 3$, there are more than

$14 \cdot 10^{11}$ possibilities. Thus, we describe an algorithm to explore the space of possible solutions efficiently; given infinite time, the algorithm would explore the whole state space. In practice, we explicitly limit its running time. We note that it is not necessary to drive all MUXs with the same bits; indeed, better fault coverage might be achieved by using different bits. Yet, the state space increases with the number of addressing schemes.

Our algorithm is simple: it randomly decides which ℓ output bits to generate for each group of input vectors; we denote by R_i the set of output bits that we generate for group G_i. Initially all the R_is are empty. The algorithm essentially picks a group and decides which output bit to generate for this group; we decide which group to pick using biased sampling and favoring groups whose corresponding R_i contains fewer elements. Biased sampling is also used to decide which output bit to include in R_i. We assign a **score** to every output bit not already included in R_i: this score reflects the **significance** of this particular output bit for fault detection. Intuitively, the **significance** of an output bit is a function of the number of faults it detects, and, in particular, faults that are not detected by a large number of vectors in V. As an example, we tend to favor an output bit that detects two faults that no other input vector can detect over an output bit that detects five faults, each detected by ten other input vectors as well. Every time an output bit is selected to be included in R_i, we remove all faults covered by that bit for any input vector in G_i. The above process is repeated until all R_i contain exactly ℓ elements and the fault coverage is reported. If the result is unsatisfactory, we repeat the process until either a satisfactory result emerges or a fixed number T of iterations is exceeded; if the result is still unsatisfactory, we try a different addressing scheme. The **SPaRe** algorithm calls the **BasicSPaRe** algorithm with different G_1, G_2, G_3, G_4 until a target fault coverage is attained or the run time limit of the scheme is exceeded.

A brief note on x: while in our experiments a value of $x = 1$ returned acceptable solutions fast (typically, after trying at most 10 addressing schemes with $T = 100$), one could try different values of x to fine tune the algorithm. As an example, as x increases, our search becomes greedier: the output bit with the highest score is picked with very high probability. We prefer to present our algorithm using generic values for x; in practice, one could potentially use training data to learn the "best" value of x for the circuits at hand.

IV. Experimental Results

In this section, we compare SPaRe to TVLR and duplication, in terms of hardware overhead, fault coverage, and fault-detection latency. In order to preserve generality, we employ randomly generated FSMs of $K = 2^k$ states and n inputs. We start by building the connected component (a tree) of the FSM, to guarantee that there exists a path from some ROOT node to every state; we denote by $s_i, i = 1 \ldots 2^k$, the states of the FSM. Starting from the ROOT, we add a random number r_i of children to each state node s_i; r_i is picked uniformly at random from $0 \ldots 2^n$ and independently for each state node. We visit the states nodes in a breadth-first search order and we stop when a full tree with all 2^k states is built. Let r_i denote the number of children of s_i; we add $2^n - r_i$ edges from state node s_i to other nodes in the tree. We pick these nodes uniformly at random with replacement. Finally, we label the 2^k states-nodes using a random permutation of $1 \ldots 2^k$; we also label the 2^n out edges from each s_i using random permutations of $1 \ldots 2^n$. The objective of this process is to build complex FSMs in order to assess the proposed method. Although alternative methods may be suggested, we emphasize that this process has the ability to generate *all possible* FSMs of $K = 2^k$ states and n inputs. We experimented with ten different types of (K, n) FSMs, namely (8, 1), (8, 2), (16, 1), (16, 2), (32, 1), (32, 2), (32, 3), (64, 1), (64, 2), and (64, 3).

Algorithm SPaRe

Input: A, ℓ

Output: R_1, R_2, R_3, R_4 (initially empty).

(a) Create candidate G_1, G_2, G_3, G_4.

(b) BasicSPaRe$(A, G_1, G_2, G_3, G_4, \ell)$

(c) Repeat (a)–(b) until the fault coverage is above target or the running time limit is exceeded.

Algorithm BasicSPaRe

Input: $A, G_1, G_2, G_3, G_4, \ell$

Output: R_1, R_2, R_3, R_4, initially empty.

Preprocessing: Assign a score to each fault in A,

$$\mathbf{score}(\mathbf{F_j}) = \mathbf{nnz}(\mathbf{A_{(j)}}), \quad \mathbf{j = 1 \ldots m}$$

$\mathbf{nnz}(\mathbf{A_{(j)}})$ denotes the number of nonzero elements in the i-th row of A.

(a) Randomly pick one of the R_i, with probability

$$\Pr(\text{picking } R_i) = (\ell - |R_i|) / \sum_{i=1}^{4} (\ell - |R_i|)$$

Denote the one picked by R_i.

(b) Assign a score to each output bit $\mathbf{O_p} \in \mathbf{R_i}$, $\mathbf{p = 1 \ldots k}$ ($x \in \mathcal{R}$, usually $x = 1$).

$$\mathbf{score}(\mathbf{O_p}) = \left(\sum_{\mathbf{F_j} \in \mathbf{S}} \mathbf{score}(\mathbf{F_j}) \right)^{\mathbf{x}}$$

$$S = \{\mathbf{F_j}: \mathbf{O_p} \text{ and any vector in } \mathbf{G_i} \text{ cover } \mathbf{F_j}\}$$

(c) Randomly pick one of the $\mathbf{O_p}$, with probability

$$\Pr(\text{picking } \mathbf{O_p}) = \frac{\mathbf{score}(\mathbf{O_p})}{\sum_{\mathbf{p}:\ \mathbf{O_p} \in \mathbf{R_i}} \mathbf{score}(\mathbf{O_p})}$$

Denote the one picked by $\mathbf{O_p}$.

(d) $R_i = R_i \cup \{\mathbf{O_p}\}$

(e) Remove all faults (rows of A) covered by $\mathbf{O_p}$ and any vector in G_i.

(f) Repeat steps (a)–(e) until all the R_is contain exactly ℓ elements and report the fault coverage.

(iter) Repeat steps (a)–(f) T times.

A. Hardware Overhead

In terms of incurred hardware overhead, the major difference between TVLR, SPaRe, and duplication is in the prediction logic. Duplication employs a replica of the combinational next state logic of the original FSM, while TVLR employs a predictor which is accurate only for test vectors, as well as an additional function indicating whether the current input is a test

FSM (States, Inputs)	Duplication	TVLR		SPaRe		Cost Comparison		
	Cost	Vectors	Cost	Bits	Cost	TVLR vs. Duplication	SPaRe vs. Duplication	SPaRe vs. TVLR
(8, 1)	33408	10 / 16	33872	2 / 3	28734	101.39 %	86.00 %	84.83 %
(8, 2)	70374	19 / 32	65578	2 / 3	52846	93.18 %	75.09 %	80.58 %
(16, 1)	95275	20 / 32	83210	2 / 4	52515	87.33 %	55.11 %	63.11 %
(16, 2)	186219	35 / 64	153429	2 / 4	102294	82.39 %	54.93 %	66.67 %
(32, 1)	222411	38 / 64	178794	2 / 5	100303	80.38 %	45.09 %	56.09 %
(32, 2)	423014	69 / 128	325882	2 / 5	204086	77.03 %	48.24 %	62.62 %
(32, 3)	832571	128 / 256	615573	2 / 5	369779	73.93 %	44.41 %	60.07 %
(64, 1)	504368	70 / 128	384346	3 / 6	276888	76.20 %	54.89 %	72.04 %
(64, 2)	937744	129 / 256	688112	3 / 6	522590	73.37 %	55.72 %	75.94 %
(64, 3)	1809757	237 / 512	1227689	3 / 6	1022400	67.83 %	56.49 %	83.27 %

Fig. 5. Hardware overhead comparison of CED based on duplication, CFD based on TVLR, and CFD based on SPaRe.

vector. In contrast, SPaRe employs a predictor that generates only a subset of the output bits of the circuit. As a result, SPaRe uses a narrower state register and a narrower comparator than duplication and TVLR. However, SPaRe employs a few additional MUXs, balancing the cost savings of these modules. Essentially, in order to compare the three methods, it is adequate to compare the cost of the prediction logic employed by each of them.

In order to obtain these costs, the next state function of the FSMs generated through the above process is converted to *pla* format, synthesized using the *rugged* script of SIS [23], and mapped to a standard cell library comprising only 2-input gates. Since the proposed methodology is nonintrusive, no assumptions are made as to how the FSMs are encoded or optimized. For TVLR, ATPG is performed using ATALANTA [24]. The test vector set is subsequently converted to *pla* format, synthesized using the *rugged* script of SIS [23], and mapped to a standard cell library comprising only 2-input gates. For input combinations that are not in the test set, the output of the circuit is set to *don't care*, thus allowing SIS [23] to minimize the hardware. The additional function indicating whether a current input is a test vector is also synthesized together with the predictor. For SPaRe, ATALANTA [24] is used to generate all vectors detecting each fault, and HOPE [25] is employed to provide both the good machine and the bad machine responses for every *(vector, fault)* pair. This reveals the output bits at which each fault may be detected for every vector. This information is used to construct the matrix A shown in Fig. 4, through which the prediction logic functions for SPaRe are identified. These functions are subsequently converted to *pla* format, synthesized using the *rugged* script of SIS [23], and mapped to a standard cell library comprising only 2-input gates.

In all three cases, the cost of the prediction logic is reported by SIS [23] through the *print_map_stats* command. The results are summarized in Fig. 5. The cost of duplication is provided first, followed by the number of test vectors required by TVLR and the cost of the synthesized TVLR prediction logic. Subsequently, the number of prediction logic bits generated through the algorithm of Section III-C is reported, along with the cost of the synthesized SPaRe prediction logic. Finally, the three rightmost columns indicate the cost of TVLR as a percentage of the cost of duplication, and the cost of SPaRe as a percentage of the cost of duplication and TVLR, indicating the hardware savings of SPaRe over these approaches. As may be observed, the hardware overhead of SPaRe is, on average, 45% less than duplication and 30% less than TVLR. Furthermore, as the size of the circuit increases, the percentage of predicted output bits for SPaRe is expected to decrease, thus resulting in even higher hardware savings.

B. Fault Coverage

By construction, both TVLR and SPaRe are expected to detect all faults in the original FSM. In order to demonstrate this, we construct the FSM with duplication-based CED, the FSM with TVLR-based CFD and the FSM with SPaRe-based CFD in ISCAS89 [26] format. The next state logic, the prediction logic for TVLR and the prediction logic for SPaRe are obtained as described in the previous section. Two copies of the original FSM and a comparator are used for duplication. One copy of the original FSM, the TVLR prediction logic and a comparator are used for TVLR. One copy of the original FSM, the SPaRe prediction logic, a narrower comparator and a few MUXs for the selection logic are used for SPaRe.

Two experiments are performed employing these circuits. In the first experiment, we compare the number of faults in the *original* FSM detectable by SPaRe to those detectable by TVLR and duplication. HITEC [27] is used to perform ATPG on the three constructed FSMs. In all three, ATPG runs only the faults in the original FSM are targeted and only the Test Output is made observable. The results are summarized in Fig. 6. Duplication detects all testable faults in the original FSM, the number of which is reported in the second row. As expected, all faults testable by duplication are also detected by both the TVLR-based CFD method and the SPaRe-based CFD method.

In the second experiment, we demonstrate the ability of SPaRe to also detect all testable faults in the hardware added for CFD. Two ATPG runs are performed using HITEC [27] on the FSM with SPaRe-based CFD, targeting *all* circuit faults. Both the test output and the original FSM outputs are made observable in the first ATPG run, while only the test output is made observable in the second ATPG run. The results are summarized in Fig. 7. As demonstrated, all testable faults in the additional hardware are also detected by SPaRe-based CFD.

FSM Type	(8, 1)	(8, 2)	(16, 1)	(16, 2)	(32, 1)	(32, 2)	(32, 3)	(64, 1)	(64, 2)	(64, 3)
Faults Detected by Duplication	53	123	143	286	362	649	1312	788	1433	2751
Faults Detected by TVLR	53	123	143	286	362	649	1312	788	1433	2751
Faults Detected by SPaRe	53	123	143	286	362	649	1312	788	1433	2751

Fig. 6. Fault coverage of duplication, TVLR, and SPaRe on original FSM.

FSM Type	(8, 1)	(8, 2)	(16, 1)	(16, 2)	(32, 1)	(32, 2)	(32, 3)	(64, 1)	(64, 2)	(64, 3)
Testable Faults	177	285	326	552	603	1074	1964	1340	2384	4496
Detected by SPaRe	177	285	326	552	603	1074	1964	1340	2384	4496

Fig. 7. Fault coverage of SPaRe on all faults.

TESTABLE FAULTS	STATISTIC	RANDOM 10		RANDOM 50		RANDOM 100		RANDOM 500		RANDOM 1000		RANDOM 5000	
		TVLR	SPaRe	TVLR	SPaRe	TVLR	SPaRe	TVLR	SPaRe	TVLR	SPaRe	TVLR	SPaRe
2751	REMAINING	1970	1970	1081	1081	640	640	91	91	25	25	0	0
	DETECTED	589	669	1286	1525	1669	1983	2492	2577	2657	2680	2751	2751
	MISSED	192	112	384	145	442	128	168	83	69	46	0	0
	MAX LAT	7	8	41	46	94	94	451	466	936	947	4203	2714
	AVG LAT	0.18	1.32	2.97	2.16	7.86	4.02	38.94	11.66	60.75	20.51	91.05	28.35

Fig. 8. Comparison of fault-detection latency of TVLR-based CFD and SPaRe-based CFD on (64, 3) FSM.

C. Fault-Detection Latency

The hardware savings achieved by TVLR and SPaRe come at the cost of introducing fault-detection latency, unlike duplication which immediately detects all errors. It is not possible to predict the exact latency of the method, since it depends on the values that appear at the FSM inputs during normal operation. Yet, an experimental indication of how much latency is introduced by TVLR and SPaRe is necessary for their evaluation.

We measure fault-detection latency based on fault simulation of randomly generated input sequences. More specifically, we use HOPE [25] to perform *two* fault simulations of the *same* sequence of randomly generated inputs, once observing both the test output and the FSM outputs, and a second time observing only the test output. The time step at which a fault is detected during the first fault simulation is the *fault-activation* time, while the time step at which it is detected during the second fault simulation is the *fault-detection* time. *Fault-detection latency* is the time difference between fault activation and fault detection, therefore we can calculate the fault-detection latency for each fault, as well as the average fault-detection latency.

Results on the largest example, the (64,3) FSM, are summarized in Fig. 8 for both TVLR and SPaRe. Similar results hold for all other circuits. We fault simulate a total of 5000 random patterns and snapshots of the results are shown after 10, 50, 100, 500, 1000, and finally, all 5000 patterns are applied. For each snapshot, we provide the number of faults *remaining* nonactivated, the number of faults activated and *detected*, and the number of faults activated but *missed* (not yet detected) by TVLR and SPaRe. We also provide the *maximum* and the *average* fault-detection latency for the faults that are both activated and detected. Based on these results we observe the following.

- While the MAX latency is significant, the AVG latency ranges only up to 92 vectors for TVLR and 29 vectors for SPaRe. For example, once all faults are detected, the MAX latency is 4203 vectors for TVLR and 2714 vectors for SPaRe. However, the AVG latency is 91.05 vectors for TVLR and 28.35 vectors for SPaRe, which is only 2.16% and 1.04% of the respective MAX latency.
- For both TVLR and SPaRe, most faults are detected quickly and a 90–10 rule applies for the AVG latency: 90% of the faults are detected within 50% of the AVG latency, while the other 50% is contributed by the remaining 10% of the faults. For example, once 500 vectors are applied, 2660 (i.e., 96.69%) of all faults are activated, out of which 2492 (i.e., 90.57%) are detected by TVLR and 2577 (i.e., 93.67%) by SPaRe. The AVG fault-detection latency at this point is 38.94 vectors for TVLR and 11.60 vectors for SPaRe, which represents 42.76% and 40.91% of the AVG latency when all faults are detected.

Furthermore, a comparative examination of TVLR and SPaRe leads to the following two observations.

- SPaRe detects more faults slightly faster than TVLR. A plot of the faults activated, faults detected by TVLR, and faults detected by SPaRe as the number of applied random patterns increases is given in Fig. 9 for circuit (64, 3). As demonstrated, SPaRe consistently detects more faults faster than TVLR, up to the convergence point where all faults are detected by both methods.
- SPaRe detects faults with significantly lower AVG latency than TVLR. A plot of the AVG fault-detection latency of SPaRe and TVLR as the number of applied random patterns increases is given in Fig. 10 for circuit (64, 3). As demonstrated, SPaRe consistently detects faults with lower AVG latency than TVLR.

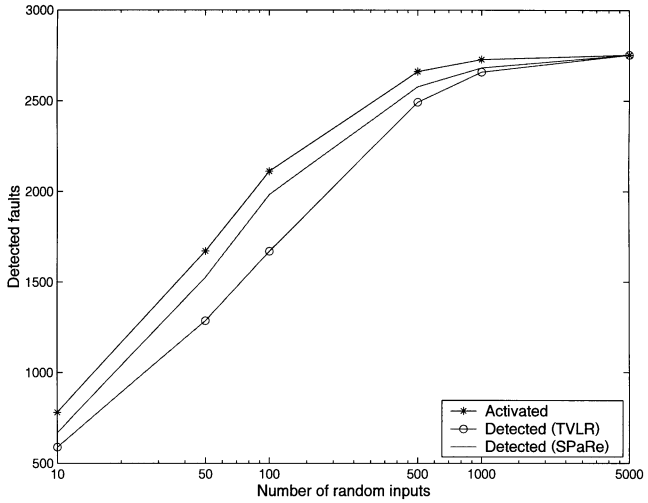

Fig. 9. Fault coverage versus number of random patterns for (64, 3) FSM.

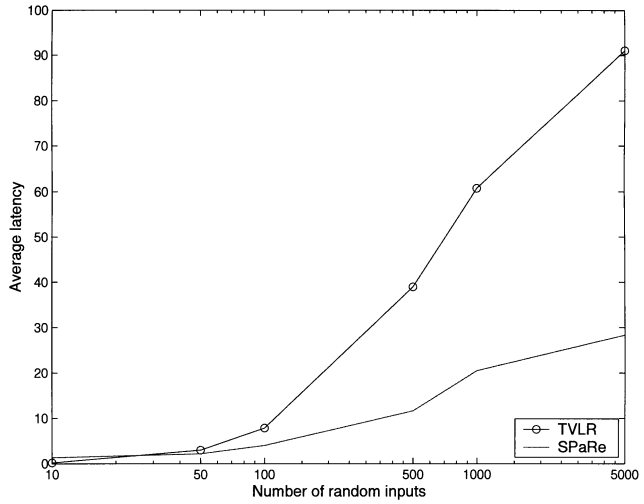

Fig. 10. Average latency versus number of random patterns for (64, 3) FSM.

V. Conclusion

Design of controller circuits with cost-effective concurrent test capabilities requires careful examination of the tradeoffs between the conflicting objectives of low hardware overhead, low fault-detection latency, and high fault coverage. SPaRe, the proposed CFD method, explores the tradeoff between fault-detection latency and hardware overhead, under the additional constraint that the original circuit design may not be altered. Thus, a comparison-based approach is employed, wherein the next state logic of the original FSM is partially replicated into a smaller prediction logic which selectively tests the circuit during normal operation. The problem of identifying the minimum number of adequate prediction functions is theoretically formulated and an algorithm for efficient selective partial replication is proposed. Experimental results demonstrate that SPaRe reduces significantly the incurred hardware overhead over both duplication and TVLR, a previously proposed CFD method, while preserving the ability to detect all permanent faults in the circuit. Further reduction of this overhead is anticipated as the size of the circuit increases. While these savings come at the cost of introducing fault-detection latency, the experimentally observed average latency is smaller than the average fault-detection latency of TVLR and scales favorably with the size of the circuit. In conclusion, when nonzero fault-detection latency can be tolerated, SPaRe constitutes a powerful alternative to both duplication and TVLR.

References

[1] A. Avizienis and J. P. J. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *IEEE Computer*, vol. 17, pp. 67–80, Aug. 1984.
[2] R. Sharma and K. K. Saluja, "An implementation and analysis of a concurrent built-in self-test technique," in *Proc. Fault Tolerant Computing Symp.*, 1988, pp. 164–169.
[3] P. Drineas and Y. Makris, "Non-intrusive design of concurrently self-testable FSMs," in *Proc. Asian Test Symp.*, 2002, pp. 33–38.
[4] M. Gossel and S. Graf, *Error Detection Circuits*. New York: McGraw-Hill, 1993.
[5] S. J. Piestrak, "Self-checking design in Eastern Europe," *IEEE Des. Test Comput.*, vol. 13, pp. 16–25, Jan. 1996.
[6] S. Mitra and E. J. McCluskey, "Which concurrent error detection scheme to choose?," in *Proc. Int. Test Conf.*, 2000, pp. 985–994.
[7] K. K. Saluja, R. Sharma, and C. R. Kime, "A concurrent testing technique for digital circuits," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 1250–1260, 1988.
[8] I. Voyiatzis, A. Paschalis, D. Nikolos, and C. Halatsis, "R-CBIST: An effective RAM-based input vector monitoring concurrent BIST technique," in *Proc. Int. Test Conf.*, 1998, pp. 918–925.
[9] G. Aksenova and E. Sogomonyan, "Synthesis of built-in test circuits for automata with memory," *Autom. Remote Control*, vol. 32, no. 9, pp. 1492–1500, 1971.
[10] ——, "Design of self-checking built-in check circuits for automata with memory," *Autom. Remote Control*, vol. 36, no. 7, pp. 1169–1177, 1975.
[11] S. Dhawan and R. C. De Vries, "Design of self-checking sequential machines," *IEEE Trans. Comput.*, vol. 37, pp. 1280–1284, Oct. 1988.
[12] N. K. Jha and S.-J. Wang, "Design and synthesis of self-checking VLSI circuits," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 878–887, June 1993.
[13] N. A. Touba and E. J. McCluskey, "Logic synthesis of multilevel circuits with concurrent error detection," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 783–789, July 1997.
[14] E. Sogomonyan, "Design of built-in self-checking monitoring circuits for combinational devices," *Automat. Remote Contr.*, vol. 35, no. 2, pp. 280–289, 1974.
[15] C. Zeng, N. Saxena, and E. J. McCluskey, "Finite state machine synthesis with concurrent error detection," in *Proc. Int. Test Conf.*, 1999, pp. 672–679.
[16] V. V. Danilov, N. V. Kolesov, and B. P. Podkopaev, "An algebraic model for the hardware monitoring of automata," *Autom. Remote Control*, vol. 36, no. 6, pp. 984–991, 1975.
[17] D. Das and N. A. Touba, "Synthesis of circuits with low-cost concurrent error detection based on Bose–Lin codes," *J. Electron. Testing: Theory Applicat.*, vol. 15, no. 2, pp. 145–155, 1999.
[18] R. A. Parekhji, G. Venkatesh, and S. D. Sherlekar, "Concurrent error detection using monitoring machines," *IEEE Design Test Comput.*, vol. 12, pp. 24–32, Mar. 1995.
[19] S. Tarnick, "Bounding error masking in linear output space compression schemes," in *Proc. Asian Test Symp.*, 1994, pp. 27–32.
[20] A. Chatterjee and R. K. Roy, "Concurrent error detection in nonlinear digital circuits with applications to adaptive filters," in *Proc. Int. Conf. Computer Design*, 1993, pp. 606–609.
[21] I. Bayraktaroglu and A. Orailoglu, "Low-cost on-line test for digital filters," in *VLSI Test Symp.*, 1999, pp. 446–451.
[22] Y. Makris, I. Bayraktaroglu, and A. Orailoglu, "Invariance-based on-line test for RTL controller-datapath circuits," in *Proc. VLSI Test Symp.*, 2000, pp. 459–464.
[23] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldahna, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Univ. California, Berkeley, ERL MEMO. no. UCB/ERL M92/41, 1992.
[24] ATALANTA Combinational Test Generation Tool. [Online]. Available: http://www.ee.vt.edu/ha/cadtools
[25] H. K. Lee and D. S. Ha, "HOPE: An efficient parallel fault simulator for synchronous sequential circuits," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1048–1058, Sept. 1996.
[26] ISCAS'89 Benchmark Circuits Information. [Online]. Available: http://www.cbl.ncsu.edu
[27] T. Niermann and J. H. Patel, "HITEC: A test generation package for sequential circuits," in *Proc. Eur. Conf. Design Automation*, 1992, pp. 214–218.

Petros Drineas received the Diploma degree in computer engineering and informatics from the University of Patras, Greece, in 1997, and the M.S. and Ph.D. degrees in computer science from Yale University, New Haven, CT, in 1999 and 2003, respectively.

He is currently an Assistant Professor of computer science at Rensselaer Polytechnic Institute, Troy, NY. His research interests lie in the area of design and analysis of algorithms and, in particular, randomized algorithms for matrix operations and their applications.

Yiorgos Makris (S'96–M'02) received the Diploma degree in computer engineering and informatics from the University of Patras, Greece, in 1995, and the M.S. and Ph.D. degrees in computer engineering from the University of California, San Diego, in 1997 and 2001, respectively.

He is currently an Assistant Professor of electrical engineering and computer science at Yale University, New Haven, CT. His research interests include test generation, testability analysis, DFT, and concurrent test for analog and digital circuits and systems.