Synthesis of ASM-based Self-Checking Controllers[1]

Ilya Levin
Tel-Aviv University
School of Education
ilia1@post.tau.ac.il

Vladimir Sinelnikov
Academic Technological Institute, Department of Computer Science

Mark Karpovsky
Boston University
Department of Electrical, Computer and System Engineering

Abstract

In this paper we present a new technique for on-line checking of FPGA-based sequential devices defined by their algorithmic state machines (ASMs). The technique utilizes specific properties of ASMs for achieving the totally self-checking goal with a low hardware overhead. This technique is based on the architecture that consists of two portions: a self-checking sequential device and a separate totally self-checking (TSC) checker. Each of these portions is implemented as a combination of an "evolution" block and an "execution" block. Comparison of code vectors transferred between these blocks provides for the totally self-checking property. The proposed technique does not require any redundant encoding of output words and uses a one-rail design, thereby drastically decreasing the required overhead. The paper presents overhead estimations and results for benchmarks for the proposed architecture.

1. Introduction

Techniques for concurrent error detection for finite state machine (FSM) controllers have received a wide attention, since the control part of a digital system is usually the most critical part from the testability point of view. Irregularity and complexity of the control structure on one hand, and its central role in functioning of the whole controlled digital system on the other hand, puts the problem of synthesis of self-checking FSM controllers onto the theoretical and practical agenda. Most of the faults that occur in VLSI circuits and systems are transient/intermittent in nature. The self-checking property allows both the transient/intermittent and permanent faults to be detected on-line, thus preventing data contamination.

Existing approaches for design of self-checking FSMs are based on either duplication, or application to them of specific error detecting codes (Berger code, constant weight code, etc.). In most cases, these approaches require a hardware overhead of more than 100 percent.

Major difficulties in designing of self-checking devices are related to the complexity of decoding (i.e. verification that a given output is a codeword). Outputs of a self-checking circuit are usually encoded by codewords of a code, which detects unidirectional errors [8, 15, 19]. For example, in [8, 15] it was shown that stuck-at fault, cross-point faults and shorts in MOS PLAs and ROMs result in unidirectional errors at their outputs.

Applications of the self-checking concept to Control Units and microprocessors were presented in [19]. Several works deal with synthesis of totally self-checking (TSC) Control Units [4, 10, 16, 17], design for testability of controllers [6, 7] and self-checking control networks [3, 9, 14]. Paper [16] presents several schemes for on-line checking of microprogrammed control units, which are based on computing of a set of signatures and inserting of these signatures in a microprogram code at specific locations. Papers [10] and [17] are also dedicated to the problem of synthesis of self-checking microprogrammed control units. In [10] a design of a special monitor circuit enabling to detect a specified fault set is proposed. In [17], duplication of a microprogram sequencer was proposed to achieve the totally self-checking property.

Paper [9] describes a special technique for decomposing the initial FSM to achieve both on-line checking and on-line testing. The concurrent testing and checking allows decreasing the overhead in comparison to traditional on-line checking approaches. The technique, which is presented in [15] allows detection of input faults by providing so-called bi-orthogonality of input vectors.

[1] This research was supported by BSF under grant No. 9800154.

To the best of our knowledge in the above-mentioned works the authors did not try to design efficient controllers with on-line checking ability by utilizing specific ASM properties.

In this paper we deal with one specific but widely used class of controllers, so-called Algorithmic State Machines (ASMs) [1] based controllers. This particular class of FSM-based controllers has several important properties that can be utilized for designing self-checking controllers. We investigate these properties from the point of their influence on the self-checking ability of the controller. We propose a novel *ASM based self-checking controller* architecture.

Properties of ASM controllers that make them differ from conventional FSM-based controllers are the following.

P1. ASM controllers have orthogonal systems of product terms. A product of any two product terms corresponding to transitions from the same internal state is equal to 0. In the general case of the FSM based controllers we have only an orthogonal system of transition functions, but a non-orthogonal system of product terms.

P2. ASM controllers have "complete" systems of product terms. The sum of product terms, corresponding to transitions from any state is equal to 1.

P3. Number of different output vectors for ASM controller are much smaller than 2^N, where N is a number of output lines and also smaller than a number H of transitions.

P4. A number of input variables corresponding to any one of the transitions is much smaller than the total number of input variables.

It will be shown in the paper that these properties can be used for reduction of overheads.

Known approaches for synthesis of Mealy-type FSMs are based on Berger encoding of outputs and m-out-of-n state assignment [11, 18]. For these architectures checkers detect presence of a fault by examining whether an output vector belongs to the corresponding code.

The property P3 was used in [12] for designing of checkers. The authors show that the checker of an ASM controller can be efficiently implemented in the form of "sum of minterms" (SOM) of output functions of the controller. An unordered code for output vectors is used because for these codes any unidirectional error cannot transform one codeword into another codeword. In [12], the Berger code was used as an unordered code. Note that SOM-checker examines whether an output codeword belongs to the set of microinstructions, (and not to the Berger code as in the case of standard design [2, 11]. This allows a reduction of the required overhead.

We will use Field Programmable Gate Arrays (FPGAs) as a basis for ASM controller's implementations. The approach described in [11] for synthesis of self-checking circuits by FPGAs is based on dual-rail implementations of the hardware to be checked. Using this approach FPGA-based ASM controller can be implemented as a combination of the dual-rail controller and the dual-rail SOM-based checker (SOM-checker). In this case, output vectors of the ASM controller have to be encoded by a code detecting all unidirectional errors (such as the Berger code). Needless to say, that such an implementation is critical from the point of view of resulting overhead due to both the dual-rail design and the Berger encoding.

In present paper we propose a new architecture that does not require any encoding of output vectors and allows a single-rail design of controllers. In this work we investigate the proposed architecture from the point of view of the required overhead. Estimations of the overhead, presented in this paper, provides for guidelines for synthesis of ASM controllers with required parameters.

This paper is organized as follows. Section 2 introduces basic definitions and a review of related works. In Section 3 we describe the proposed architecture of the self-checking ASM controller. Evaluations of the required overhead for the proposed architecture are presented in Section 4. Benchmarks results are presented in Section 5. Conclusions are presented in Section 6.

2. Frameworks

In this section we remind basic definitions related to ASMs and fault models.

2.1. ASM Controllers

We will consider a system as a combination of a control unit (controller) and a data-path. A microoperation be an elementary step of data processing in the data-path, we denote by $Y = \{y_1,..., y_N\}$ the set of microoperations. The microoperations are initiated by the corresponding binary signals from of the controller. In other words, to perform the microoperation y_i signal $y_i = 1$ has to appear at the output y_i of the control unit. Sometimes several microoperations are executed simultaneously in the data-path. A set of microoperations executed simultaneously is called a microinstruction.

An Algorithmic State Machine (ASM) [1] is a directed connected graph containing an initial vertex, a final vertex, a finite set of operator vertices and conditional vertices.

Every operator vertex is labelled by a microinstruction. We denote a set of microinstructions as $\{Y_1,\ldots,Y_M\}$, where M is a number of microinstructions. The "*1-out-of-M*" code of microinstruction Y_m will be denoted by $\delta(Y_m)$.

In the vector $\delta(Y_m)$ the m-th bit is equal to 1 while all others are equal to 0.

The concept of ASM is very close to the concept of a finite state machine (FSM). For ASMs each path containing conditional vertices and one operational vertex can be interpreted as a transition within the FSM [1]. Each transition is associated with a product term that equals to 1 when this transition occurs. The set of product terms we will denote as: $P = \{ p_1, \dots, p_H \}$. We will represent a FSM as the list of transitions, which correspond to paths in the ASM.

An example of ASM with logical conditions $x_1, \dots, x_4$, micro-operations $y_1, \dots, y_6$, microinstructions $Y_0, \dots, Y_6$ and states of FSM $a_1, \dots, a_5$ is shown in Figure 1.

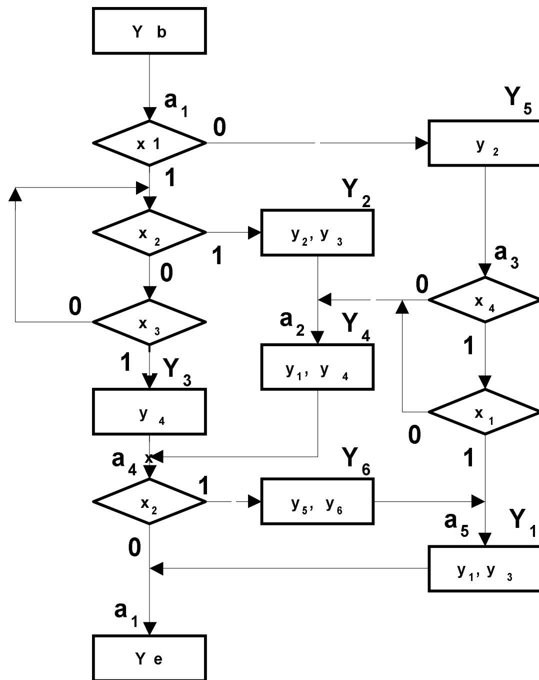

Figure 1. The Example of an Algorithmic State Machine

The transition table of the corresponding FSM is presented in Table 1.

In this table:

a_m and a_s present state and next state correspondingly,

$X(a_m, a_s)$ - transition function, i.e. a Boolean function which is equal to 1 when FSM makes the transition from state a_m to state a_s.

$Y(a_m, a_s)$ - list of output signals which are equal 1 on the transition of the FSM from a_m to a_s,

Table 1. Table of FSM corresponding to the ASM from Figure 1

a_m	a_s	$X(a_m, a_s)$	$Y(a_m, a_s)$	h
a_1	a_2	$x_1 x_2$	y_2, y_3	1
	a_4	$x_1 \bar{x}_2 x_3$	y_4	2
	a_1	$x_1 \bar{x}_2 \bar{x}_3$	–	3
	a_3	$\bar{x}_1$	y_2	4
a_2	a_4	1	y_1, y_4	5
a_3	a_1	$x_4 x_1$	y_1, y_3	6
	a_4	$x_4 \bar{x}_1$	y_1, y_4	7
	a_4	$\bar{x}_4$	y_1, y_4	8
a_4	a_5	x_2	y_5, y_6	9
	a_1	$\bar{x}_2$	–	10
a_5	a_1	1	y_1, y_3	11

We will use the following notations:

L - number of input variables (L=4 for our example);

N - number of output variables (N=6 for our example);

H - number of transitions (product terms) (H=11 for our example);

M - number of microinstructions (M=7 for our example);

R - number of states (R=5 for our example).

2. 2. Definitions and Assumptions

We now recall some basic definitions from the theory of design of Totally Self Checking (TSC) sequential circuits.

A state machine (ASM or FSM) is self-testing if, for every fault in a fault's set, there is such an input/state pair in the circuit that a non-code output is produced. A state machine is fault-secure if, for every fault from a faults set, the machine never produces an incorrect code output for a code input. A state machine is totally self-checking if it is both self-checking and fault-secure [11].

2. 3. Fault Model

As it has been mentioned, the basis of target implementation of the ASM controller is LUT-based FPGA comprising Configurable Logic Blocks (CLBs). The fault model used in this paper is general model of single cell faults. We assume that at most one CLB can produce a faulty output. The circuit primary inputs are considered to be fault-free.

3. Match Detector based architecture of a self-checking FSM

We propose a new architecture for ASM controllers that does not require any encoding of output vectors and consequently allows reduction of the required overheads. We call this architecture Match Detector (MD) architecture since it is based on using a Match Detector within the checker. The FSM for this architecture consists of two portions: a self-checking FSM and a MD-checker. In turn, each of these portions contains two main blocks: the "evolution" block and "execution" block [13]. Additionally the FSM contains a Product Terms Compressor (PTC), while the checker contains a Math Detector (MD). A schematic diagram of the MD-architecture is shown in Figure 2.

3.1. Self-checking FSM

Inputs of the evolution block of the FSM (EvFSM) comprise working inputs X of the FSM and output memory signals $T = (t_1,\ldots,t_r)$. Outputs of the EvFSM correspond to product terms $P = \{p_1,\ldots,p_H\}$. At each clock, one and only one product term is equal to 1, which means that EvFSM outputs are codewords of the *"1-out-of-H"* code. The EvFSM is denoted in Figure 2 as $X \times T \rightarrow \delta(P)$. The Product Terms Compressor (PTC) transforms *"1-out-of-H"* code δ *(P)* into *"1-out-of-M"* code δ *(Y)*. As mentioned in Section 1, the number of microinstructions is essentially smaller than the number of product terms for the typical ASM controller. PTC comprises CLBs that are programmed for implementation of "*1-out-of-2g*" functions.

The EvFSM is implemented as a tree, wherein each of the nodes is either a pre-designed Configurable Logic Block (CLB), or a fan-out. Each CLB is designed for implementation of either a sum of two product terms of *g* variables, or an AND-function of *2g* variables. We use the Xilinx-4000 series FPGAs [20] for implementation of the proposed self-checking scheme. In this case the number of inputs of the CLB is equal to 8, which means $g = 4$.

The execution block of the FSM (ExFSM) implements OR-assembling of EvFSM outputs. Outputs of ExFSM are output signals Y of the FSMand input memory signals $D = (d_1 ,..., d_R)$. The memory signals are coded by codewords of the "*1-out-of-R*" code.

3.2. TSC MD-checker

The MD-checker consists of the evolution block (EvCh), the execution block (ExCh) and the Math Detector (MD) between them. EvCh implements all minterms, while ExCh assembles these minterms to implement the checker's function.

The EvCh is built as a self-checking tree with "AND" nodes for implementation of "long" product terms and "fork" nodes actually implemented by regular fan-outs. The ExCh comprises either "*1-out-of-2g*", or *"(2g -1)-out-of-2g"* cells (CLBs) combining all minterms, coming from the EvCh.

It is proposed to use the following pre-designed A-Cells and O-cells for implementation of the above-mentioned nodes of the checker.

A-Cell implements nodes of the "AND" type. It has two inputs, four functional inputs for implementation of minterms, and two cascade outputs.

O-Cell implements nodes of either *"g-out-of-2g"* or *"(2g -1)-out-of-2g"* types. These cells have 8 inputs and 1 output.

EvCh is a self-checking two-rail tree comprising a number of A-Cells. This tree is constructed in such a way that in the case of proper functioning of both the controller and the checker one and only one dual-rail output (S_i,V_i) will have value (1,0). All the remaining outputs will have value (0,1). Outputs of the EvCh tree are inputs of the ExCh of the checker. Each of the two-rail outputs of the EvCh corresponds to a certain microinstruction of the original FSM.

ExCh comprises two components consisting of O-Cells. All S-outputs of EvCh serve as inputs of the first component of the ExCh. This component is implemented as a converging "*1-out-of-M*" multilevel tree. All V-outputs of the EvCh are inputs to the second component of the ExCh, which is implemented as a converging "*(M-1)-out-of-M*" multilevel tree.

The Match Detector compares outputs of the PTC and outputs of the evolution block of the checker (EvCh). Any output vector of PTC is formed by M binary one-rail outputs. Output vectors of EvCh are M dual-rail-coded outputs. In Figure 2, the checker is shown as MD-checker. If the two compared vectors are equal, the resulting vector will be equal to the EvCh output vector. If they are not, the ExCh will receive a predetermined faulty dual-rail vector.

Table 2. Truth table of the Match Detector

$\delta(i)$	$S^1(i)$, $V^1(i)$		$S^0(i)$, $V^0(i)$	
1	**1**	**0**	**1**	**0**
0	**0**	**1**	**0**	**1**
1	**0**	**1**	**1**	**1**
0	**1**	**0**	**0**	**0**
-	**0**	**0**	**0**	**0**
-	**1**	**1**	**1**	**1**

An example of the match detection function is shown in Table 2. In this table:

$S^1(i)$, $V^1(i)$ - dual-rail code of bit i of an output vector of the EvCh;

$S^0(i)$, $V^0(i)$ - dual-rail code of bit i of the corresponding output vector the match detector (MD);

$\delta(i)$ - the state of bit i of the PTC single-rail output vector $\delta(Y)$.

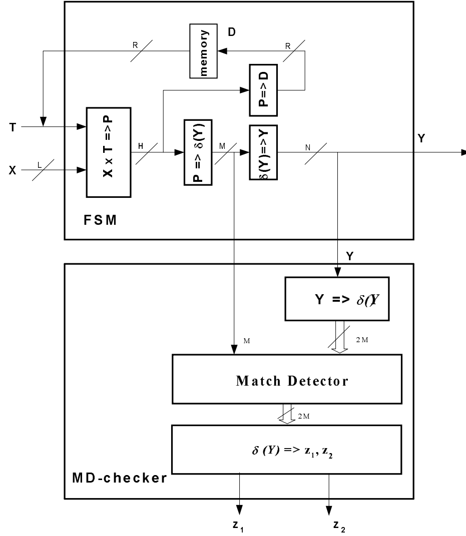

Figure 2. The MD-architecture of the totally self-checking controller

For the MD-architecture of at any clock, an input vector initiates one and only one product term. The "*1-out-of-H*" vector produced by the EvFSM is transformed by PTC block in to $\delta(Y_m)$ ("*1-out-of-M*" code of Y_m). This code is introduced both into the ExFSM and the match detector. Output vectors produced by ExFSM are transformed into the same $\delta(Y_m)$ code that has been produced by the PTC. The match detector checks whether these codes are the same, and if the codes differ from one another, the MD-checker will produce the error signal.

4. Estimations of the required hardware overheads

In this section we will estimate expected overheads for the proposed MD- architecture.

The MD-architecture does not require any redundancy for the FSM itself. Therefore, the difference between the two implementations with and without the self-checking ability equals to a complexity of the MD-checker (i.e. the number of CLBs). We assume that the PTC block of the MD-architecture does not introduce overhead, since it is present in the ExFSM of the basic architecture. The match detector (MD) also does not require any additional overhead, since MD can be implemented within the last level of EvCh. (It will require one additional input in each LUT of the last level of the EvCh.)

The goal of this section is to estimate the checker's complexity as a function of two parameters: number of microinstructions M and length N of microinstructions. The complexity of the MD-checker will be estimated as a sum of complexities of its two components: EvCh and ExCh.

4.1. Estimations of the EvCh complexity

The EvCh tree implements a system of M logical functions of N variables. Each of these M functions is a unique minterm. The EvCh tree comprises CLBs having g dual-rail inputs (g = 4) [20] and one dual-rail output.

The upper bound of the complexity (number of CLBs) $S_{max}(EvCh)$ of the Evolution block of the MD-checker is:

$$S_{max}(EvCh) \leq \left\lfloor \frac{N+1}{g-1} \right\rfloor M \qquad (1)$$

(here g is decreased by 1 in denominator since one dual-rail input is used for the cascade connection between CLBs).

This bound corresponds to the case of a disjoint implementation of the minterms.

For the lower bound on complexity we assume that:
the number of CLBs at the first level of the EvCh tree is equal to M.

Each level of the EvCh tree should comprise CLBs implementing the maximal number of different minterms of (g-1) variables. Then the lower bound on a number of CLBs at level i of the EvCh tree can be computed as:

$$A_{i+1} = \left\lfloor \frac{A_i}{2^{g-1}} \right\rfloor + 1 \qquad (2)$$

and we have:

$$S_{min}(EvCh) = \sum_{i=1}^{\left\lfloor \frac{N+1}{g-1} \right\rfloor} A_i \, . \qquad (3)$$

4.2. Estimations of the ExCh complexity

As have been mentioned above, ExCh consists of two equal converging multilevel trees. One of these trees implements the "*1-out-of-2g*" function, while the second implements a function "*(2g-1)-out-of-2g*".

The complexity of the ExCh can be computed as:

$$S(\,ExCh\,) = 2(B_1 + \ldots + B_K), \qquad (4)$$

where B_k is the number of CLBs at level k of the tree and $K = \left| log_{2g} \right|$. Then B_k can be computed using the following recursive expression:

$$B_1 = \left\lfloor \frac{M}{2g} \right\rfloor; \; B_k = \left\lfloor \frac{B_{k-1} + (M - 2gB_{k-1})}{2g} \right\rfloor,$$

$$k = 2, \ldots, K. \qquad (5)$$

The sequence B_1, ..., B_K, which defines the ExCh complexity, converges rapidly to 1. All members of this sequence turn to 1 at step $j = \left\lfloor \frac{log_2 M}{g-1} \right\rfloor$ and stay equal to 1, for all $i>j$. This fact allows assessing an effectiveness of the proposed estimations. We note, that for any N and M, $\left\lfloor \frac{N+1}{g-1} \right\rfloor \geq \left\lfloor \frac{log_2 M}{g-1} \right\rfloor$. In the case of equality the gap between lower and upper bounds is minimal, which means that the accuracy of the proposed estimations is maximal. In this situation the real EvCh complexity can be estimated by its upper bound. The accuracy of the estimations decreases as the difference between the two sides of the inequality increases.

These bounds were used for estimation of complexities of checkers for benchmarks. Results of the computation are presented in Table 2, and enable comparison of the estimated complexities of the MD-architecture with the real ones.

5. Benchmarks results

We applied the synthesis approach described above to several MCNC benchmarks to compute FPGA implementations for Xilinx-4000 series FPGSs [20] Results for benchmarks are presented in Table 3. In this table:

L - number of input lines,

N - number of output lines,

H - number of product terms (transitions),

R - number of states of the FSM,

M - number of output vectors (microinstructions).

S^{FSM} and S^{MD} - complexities (numbers of CLBs) of the initial FSM and the MD-checker correspondingly.

S_{min} , S_{max} - minimal and maximal complexities of the MD checker calculated by (1) - (5).

$\Omega^{MD} = S^{MD}/S^{FSM}*100\%$;

S^B , Ω^B - complexity (numbers of CLBs) and overhead (in %) for FSM self-checking implementations based on the Berger coding architecture [2, 11].

Table 3. Overheads results for FSM benchmarks implemented by Xilinx-4000 series FPGAs

NAME	R	L	N	H	M	S^{FSM}	S^b	Ω^b	S^{MD}	$Smin$	$Smax$	Ω^{MD}
bbsse	13	7	7	53	14	37	38	103%	22	21	44	59%
cse	13	7	7	98	12	60	51	85%	24	21	42	40%
Dk-14	7	5	3	56	13	38	31	82%	22	22	22	58%
Dk-15	4	5	3	30	16	27	26	96%	21	20	22	78%
styr	32	10	9	161	25	110	77	70%	52	34	108	47%
saund	32	12	11	134	27	116	79	68%	58	38	116	50%
S1488	48	19	8	236	64	213	201	94%	153	93	210	72%
S1	20	6	8	109	20	116	38	33%	33	29	68	28%
pma	24	8	8	120	24	91	83	91%	49	41	80	54%
planet	51	19	7	118	54	82	151	184%	135	80	170	165%
S820	24	18	18	199	22	175	91	52%	57	32	162	33%
Ex6	8	8	5	36	14	45	46	102%	34	20	34	76%
Ex1	20	23	9	154	60	74	171	231%	133	87	258	180%
tav	4	4	4	49	11	26	23	88%	21	18	28	81%
big	18	28	17	185	17	124	87	70%	71	26	114	57%
bs	19	13	17	185	17	125	67	54%	43	21	91	34%
acdl	16	27	22	214	23	158	114	72%	89	61	214	56%
cow	49	24	24	261	18	262	111	42%	84	53	152	32%
v1_6	14	18	17	169	17	74	63	85%	45	42	124	61%
v1_10	15	18	18	264	18	102	70	69%	49	42	132	48%
v11_20	14	29	18	367	17	110	82	75%	71	61	176	65%

One can see from this table that the proposed approach for design of totally self-checking microcontrollers results in overheads, which are about 65% and our approach results in about 25-30% reduction of overhead as compared to known implementation based on Berger coding architecture [2, 11].

6. Conclusions

We have proposed a novel technique for the synthesis of self-checking FPGA-based controllers. By utilizing several intrinsic features of the corresponding ASMs, the proposed architecture allows implementation of controllers by a single-rail scheme without any additional encoding of output words. This results in considerable reduction of the required overhead. Benchmarks results indicate that the proposed approach for the design of self-checking controllers is efficient from the points of view of required overheads.

References

[1] S. Baranov. Logic Synthesis for Control Automata. Kluwer Academic Publisher, Dordrecht/Boston/London. 1994.

[2] C. Bolchini, R. Montandon, F. Salince, and D. Sciuto. "Design of VHDL-Based Totally Self-Checking Finite State Machines and Data-Path Descriptions", IEEE Transaction on Very Large Scale Integration (VLSI) Systems, Vol. 8, No. 1, 2000.

[3] C. Bolchini, F. Salice, D. Sciuto. "Fault Analysis in Networks with Concurrent Error Detection Properties", Proc. Design Automation Test Europe (DATE'98), Paris, F, 1998, pp. 957-958.

[4] C. Bolchini, R. Montandon, F. Salice, D. Sciuto, "Self-checking FSMs based on a constant distance state encoding", Proc. 1995 IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems (DFT '95), Lafayette, U.S.A., 1995, pp. 271-277.

[5] A. L. Burress and P. K. Lala. "On-line Testable Logic Design for FPGA Implementation", Proc. of 1997 International Test Conference, pp. 471-478.

[6] S. Dey, V. Gangaram and M. Potkonjak, "A Controller-Based Design-for-Testability Technique for Controller-datapath Circuits," Proc. Intl. Conf. on Comp. -Aided Design, pp. 534-540, October 1995.

[7] B. Eschermann and H. Wunderlich, "Optimized Synthesis of Self-Testable Finite State Machines," Intl. Conf. on Fault Tolerant Computation, 1990.

[8] W. K. Fuchs, C. R. Chen, J. A. Abraham. Error Detection in Highly Structured Logic Arrays. IEEE Journal of Solid-State Circuits, Vol. Sc-22, no.4, August 1987, pp. 583-594.

[9] A. Hertwig. "Utilizing Off-line BIST Circuitry for the On-line Test of FSMs", 4-th International On-line Testing Conference, Capri, 1998, Compendium of papers, pp. 42-46.

[10] V. S. Iyengar, L. L. Kinney. "Concurrent Fault Detection in Microprogrammed Control Units", IEEE Transactions on Computers, Vol. C-34, No. 9, September 1985, pp. 810-821.

[11] P. Lala. Self-checking and Fault-Tolerant Digital Design. Morgan Kaufmann Publishers, San-Francisco/San-Diego/New-York/Boston/London/Sydney/Tokyo, 2000.

[12] I. Levin, M. Karpovsky. "On-line Self-Checking of Microprogram Control Units", 4-th International On-line Testing Conference, Capri, 1998, Compendium of papers, pp. 153-159.

[13] I. Levin, V. Sinelnikov. "Self-checking of FPGA based Control Units", Proceedings of 9th Great Lakes Symposium on VLSI, Ann Arbor, Michigan, 1999, IEEE press, pp. 292-295.

[14] G. P. Mak, J.A. Abraham and E. S. Davidson. The Design of PLAs with concurrent Error Detection. Digest 12th Int. Symp. Fault-Tolerant Computing, 1982, pp. 303-310.

[15] A. Yu. Matrosova, S. A. Ostanin. "Self-Checking FSM Networks Design", 4-th International On-line Testing Conference, Capri, 1998, Compendium of papers, pp. 162-166.

[16] M. Namjoo. "Design of Concurrently Testable Microprogramming Control Units", Proc. Of the 15 Annual Workshop on Microprogramming, Palo Alto, CA., pp. 173-180, Oct. 1982.

[17] A. M. Paschalis, C. Halatsis, G. Philokyprou. "Concurrently Totally Self-Checking Microprogram Control Unit with Duplication of Microprogram Sequencer", Microprocessing and Microprogramming, 20, 1987, pp. 271-281.

[18] M. G. Sami, D. Sciuto, R. Stefanelli, "Concurrently Self-Checking Structures for FSMs". Microprocessing and Microprogramming, 39 (1993) 237-240, North-Holland.

[19] M. M. Yen, W. K. Fuchs, J. A. Abraham. Designing for Concurrent Error Detection in VLSI: Application to a Microprogram Control Unit. IEEE Journal of Solid-State Circuits, vol. Sc-22, no.4, August 1987, pp. 595-605.

[20] Xilinx, "The Programmable Logic", Data Book, 1996.