Low Cost Convolutional Code Based Concurrent Error Detection in FSMs

Konstantinos Rokas & Yiorgos Makris
Electrical Engineering Department
Yale University
{konstantinos.rokas, yiorgos.makris}@yale.edu

Dimitris Gizopoulos
Department of Informatics
University of Piraeus
dgizop@unipi.gr

Abstract

We discuss the use of convolutional codes to perform concurrent error detection (CED) in finite state machines (FSMs). We examine a previously proposed methodology, we identify its limitations, and we outline two improvements that reduce its cost and enhance its effectiveness. More specifically, we demonstrate how the existing FSM hardware can be reused for computing the convolutional code keys and we formulate the optimization problem of maximizing hardware reusability. Additionally, we extend the proposed methodology to detect errors that only affect the output logic but not the next state of the FSM, which are not detected by the previously proposed methodology.

1. Introduction

Several research efforts have been expended in concurrent error detection for FSMs over the last three decades [1, 2, 3]. Proposed solutions typically explore the trade-off between the achieved *coverage* and the incurred *overhead*, while guaranteeing latency-free error detection. Several intrusive redesign and resynthesis methods are described in [4, 5, 6], wherein parity or various unordered codes are employed to encode the states of the circuit. Limitations of [6], such as structural constraints requiring an inverter-free design, are alleviated in [7], where partitioning is employed to reduce the incurred hardware overhead. Utilization of multiple parity bits, first proposed in [8], is examined in [9] within the context of FSMs. A general algebraic model for non-intrusive CED is introduced in [10]. Implementations based on Bose-Lin and Berger codes are presented in [11] and [12], respectively. Finally, parity-based CED methods are described in [1, 13, 14].

The aforementioned methods guarantee latency-free detection of all errors resulting from prescribed fault models. In contrast, a method that explores the trade-off between error detection latency and overhead, while guaranteeing coverage is described in [15, 16]. To our knowledge, this is the only previously proposed method that provides an upper bound to the detection latency and is based on the use of convolutional codes [17, 18]. In this method, additional logic is utilized to generate *key* bits during every FSM transition, such that these bits are valid sequences in a convolutional code *if and only if* the FSM is operating correctly. Any erroneous transition in a prescribed model will be detected with latency which will not exceed the latency of the convolutional code.

In this work, we extend the original method proposed in [15, 16] along two directions. First we demonstrate that the hardware necessary for computing the convolutionally encoded keys during every FSM transition can be reduced by reusing the next state logic of the FSM. Moreover, we show that the problem of maximizing hardware reuse can be formulated as a judicious selection among alternative convolutional codes and assignments of keys to FSM transitions. Second, we address the issue of faults in the output logic, which cannot be detected by the original method. In particular, by reusing the output logic for calculating the keys of the convolutional code, many faults in the output logic can also be detected. Alternatively, by adding D flip-flops, FSM outputs can be treated as state-bits and the methodology can be applied directly to detect output faults.

Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03)
1063-6722/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

2. Convolutional Code Based CED in FSMs

In this section, we describe briefly the originally proposed CED method for FSMs based on convolutional codes [15, 16]. We summarize the principles of convolutional codes that are relevant to the proposed CED method, which we then present and demonstrate through a small example.

2.1. Convolutional Codes

An (n, k, m) convolutional code is a linear code with k-bit inputs, n-bit outputs and a memory of depth m. Using a convolutional encoder, sequences of k-bit inputs are encoded as sequences of n-bit outputs, where in general, $n > k$. Each encoded output is a function of the current input and the previous m inputs. The output sequences that are generated by a convolutional code are called *code sequences*. The original inputs can be recovered from the code sequences by using a convolutional decoder. Both the convolutional encoder and the convolutional decoded may be easily realized in canonical form using k linear feed-forward shift registers and combinational logic [17, 18]. Similar to the original method proposed in [15, 16], we will focus on $(n, k, 1)$ codes, in which the output depends on the current input as well as the immediately preceding input.

A convolutional code can be represented by its transition matrix. Figure 1 shows an example of a transition matrix of a $(3, 2, 1)$ convolutional code. In this example, $k = 2$ so the inputs are 2-bit wide and the valid words in the input code space are $(0, 1, 2, 3)$ or $(00, 01, 10, 11)$ in binary. Similarly, $n = 3$ so the outputs are 3-bit wide and the valid words in the output code space are $(0, 1, 2, 3, 4, 5, 6, 7)$ or $(000, 001, 010, 011, 100, 101, 110, 111)$ in binary. The memory of this code is $m = 1$, so the response of a convolutional encoder for this code depends on both the current input U_2 and the previous input U_1. In essence, the output depends on the transition from U_1 to U_2. For example, a sequence of input 2 followed by input 0, denoted as $2 \rightarrow 0$, will produce output 7, since this is the corresponding entry in the matrix of the convolutional code. Notice that by looking at output 7 we can not decode and obtain the original sequence, as it might map to either $2 \rightarrow 0$ or to $1 \rightarrow 1$. A memory of $m = 1$ is required for decoding and this is also the latency of the convolutional code. For example, if the input sequence is $2 \rightarrow 0 \rightarrow 1$, then the output sequence is $7 \rightarrow 3$. This information is now adequate to decode, since by construction of the transition matrix of the convolutional code there is a unique way to produce this output sequence, namely the given input sequence $2 \rightarrow 0 \rightarrow 1$. The reason for this is that the 8 possible output words are divided into two disjoint groups, which are assigned to two rows each, rows 1 and 4, and rows 2 and 3 in our example. The same property also holds for the columns of the transition matrix.

2.2. Previously Proposed CED Method

An overview of the CED method for FSMs based on convolutional codes, as proposed in [15, 16], is depicted in Figure 2. This CED method targets a predefined set of Single Event Upsets (SEUs), i.e. transient errors that have a duration of exactly one clock cycle and, consequently, affect only one FSM transition. The methodology is non-intrusive, in the sense that it does not interfere with the original FSM implementation and it only adds circuitry in parallel to it.

U1 \ U2	0	1	2	3
0	0	3	5	6
1	4	7	1	2
2	7	4	2	1
3	3	0	6	5

Or Equivalently in Binary

U1 \ U2	0	1	2	3
0	000	011	101	110
1	100	111	001	010
2	111	100	010	001
3	011	000	110	101

Figure 1. Transition Matrix for Example (3,2,1) Convolutional Code

Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03)
1063-6722/03 $17.00 © 2003 **IEEE**
IEEE COMPUTER SOCIETY

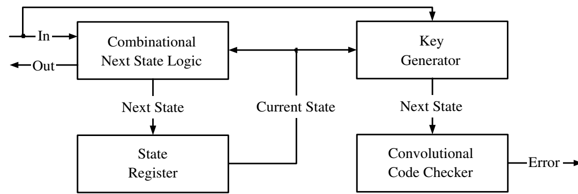


Figure 2. Convolutional Code Based CED for FSMs [15, 16]

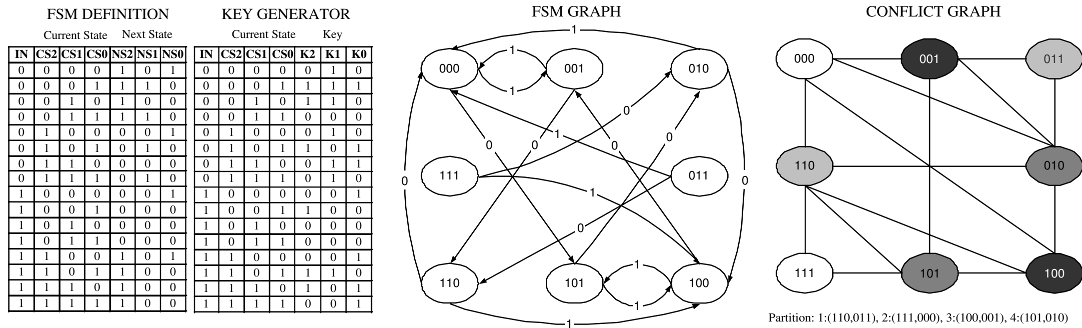
More specifically, given an FSM that consists of a state register and the next state combinational logic, two components are added in order to perform CED based on convolutional codes. The first component is a *Key Generator*, a combinational circuit that generates a key for every transition in the FSM (i.e. for every combination of current state and input). The keys are selected such that during error-free operation of the FSM, the generated key sequences are valid code sequences of a convolutional code. In essence, the FSM transitions are convolutionally encoded *a priori* by assigning appropriate keys, which are then generated during normal FSM operation through simple combinational logic. The second component is a simple *Convolutional Code Checker*. Keys are selected such that any SEU in the prescribed model, resulting in an erroneous FSM transition, will yield a sequence of keys that is not a valid code sequence for the convolutional code. This will be detected by the checker within the latency of the convolutional code.

More formally, given an FSM S, let T_f be the set of targeted erroneous transitions, and let each element of T_f be of the form (q, s, s', x), where q is the current state, s is the correct next state, s' is the faulty next state, and x is the input. Now consider a partition τ on the states of S such that, for any erroneous transition in T_f, s and s' belong to separate blocks of the partition τ. The FSM will be encoded using an $(n, k, 1)$ convolutional code, such that $n = k + 1$ and the number of partitions is 2^k. Consequently, the Key Generator will provide an n-bit codeword for every transition. The Key Generator does not need any memory, as it is not a convolutional encoder itself. It merely uses the information of the current state and the current input, to deduce the current partition, the next correct partition and, thus, the n-bit codeword corresponding to this transition. The Convolutional Code Checker will check, at each time, if the n-bit codeword at time t, the n-bit codeword at time $t - 1$ and the n-bit codeword at time $t - 2$ constitute a valid sequence of codewords. A sequence of three n-bit codewords is valid *if and only if* it can be the output of the convolutional code. For example, for the convolutional code of Figure 1, the sequence $0 \rightarrow 5 \rightarrow 1$ is a valid sequence that occurs when the input to the encoder is $(0, 2, 3)$ whereas the sequence $0 \rightarrow 5 \rightarrow 3$ is not.

2.3. Example

An example will help clarify how the keys are generated such that the above condition is met. Consider the FSM with 8 states and 1 input shown in Figure 3. For the purpose of the example, also assume that the targeted set of SEUs consists of all possible erroneous transitions due to single stuck-at faults. The conflict graph shown in Figure 3 is subsequently created. In this graph, nodes represent the states of the FSM. An edge between two nodes implies that there is an error in the targeted set, as well as a transition in the FSM, such that the error-free and the erroneous next state correspond to the states represented by the two nodes. For example, an edge between nodes 000 and 001 implies that there exists a stuck-at fault such that for some FSM transition the error-free next state is 000 and the erroneous response is 001, or vice-versa. The graph can be easily constructed using a fault simulator. The next step is to create a partition τ on this graph, such that no two nodes connected to each other belong to the same partition block. Essentially, we need to color the graph. In this example, we need at least 4 colors, with one of the many possible colorings shown in Figure

IEEE COMPUTER SOCIETY



FSM DEFINITION

IN	CS2	CS1	CS0	NS2	NS1	NS0
	Current State			Next State		
0	0	0	0	1	0	1
0	0	0	1	1	1	0
0	0	1	0	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	0	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	1	0	1
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	1	0	0

KEY GENERATOR

IN	CS2	CS1	CS0	K2	K1	K0
	Current State			Key		
0	0	0	0	0	1	0
0	0	0	1	1	1	1
0	0	1	0	1	1	0
0	0	1	1	0	0	0
0	1	0	0	0	1	0
0	1	0	1	1	0	1
0	1	1	0	0	1	1
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	1	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	0	0	1
1	1	0	1	1	1	0
1	1	1	0	1	0	1
1	1	1	1	0	0	1

Figure 3. Convolutionally Encoded FSM Example

3. Subsequently, we need an $(n, k, 1)$ convolutional code, where $n = k + 1$ and the number of partitions is smaller or equal to 2^k, resulting in this case to $n = 3$ and $k = 2$. So we are going to use a $(3, 2, 1)$ convolutional code, and more specifically, the one shown in Figure 1. Creating the key generator is now a very simple process. Every partition is assigned to one of the columns and one of the rows of the transition matrix of the convolutional code. An FSM transition from a state belonging to partition A to a state belonging to partition B will then result in the generation of a key equal to the entry (A, B) in the transition matrix. For example, on input 0 and current state 000 the next state of the FSM is 101. Since 000 belongs to the second partition and 101 belongs to the fourth partition, the key generated during this transition is going to be the entry $(2, 4)$ in the transition matrix, i.e. 010. The truth table of the key generator is shown in Figure 4. As explained in detail in [15, 16], by construction, the keys generated through the above process guarantee that any erroneous FSM transition in the prescribed error model will result in a sequence of keys that is not a valid sequence of the convolutional code and will be detected through the convolutional code checker within latency of $m = 1$ clock cycle.

3. Cost Reduction Through Hardware Reuse

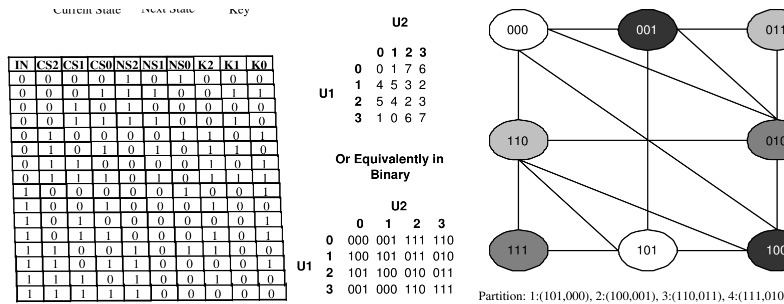
In this section, we propose a method for reducing the key generator cost by reusing the hardware that calculates the next FSM state. We outline the underlying principle, we demonstrate it through an example and we formulate the corresponding optimization problem.

3.1. Underlying Principle

As shown in the method depicted in Figure 2, the key generator calculates the key based on the current FSM state and the input. Notice that the key corresponds to a transition from the current state to the next state, but the calculation does not use the actual next state. This is computed concurrently with the generation of the key, through the combinational next state logic of the FSM. Both the key bits and the next state bits are functions of the same variables, namely the current state and the input. Additionally, while assigning the necessary keys to perform CED, there is a lot of flexibility as to the selection of the actual convolutional code and the actual partition of the conflict graph. Conceivably, by choosing an appropriate convolutional code and an appropriate partition of the conflict graph, some of key bits may become *identical* to some of the next state bits. In this case, the same hardware can be used for calculating both the key and the next state bit, thus reducing the cost of the key generator and the overall CED method.

1063-6722/03 $17.00 © 2003 **IEEE**

IEEE COMPUTER SOCIETY



IN	CS2	CS1	CS0	NS2	NS1	NS0	K2	K1	K0
0	0	0	0	1	0	1	0	0	0
0	0	0	1	1	1	0	0	1	1
0	0	1	0	1	0	0	0	0	0
0	0	1	1	1	1	0	0	1	0
0	1	0	0	0	0	1	1	0	1
0	1	0	1	0	1	0	1	1	0
0	1	1	0	0	0	0	1	0	1
0	1	1	1	0	1	0	1	1	1
1	0	0	0	0	0	1	0	0	1
1	0	0	1	0	0	0	1	0	0
1	0	1	0	0	0	0	0	0	1
1	0	1	1	0	0	0	1	0	1
1	1	0	0	1	0	1	1	0	0
1	1	0	1	1	0	0	0	0	1
1	1	1	0	1	0	0	1	0	0
1	1	1	1	1	0	0	0	0	0

Figure 4. Convolutionally Encoded FSM Example with Hardware Reuse

GENERAL FORM

	P1	P2	P3	P4
P1	0	A3	A5	A6
P2	A4	A7	A1	A2
P3	A7	A4	A2	A1
P4	A3	0	A6	A5

CODE A

	1	2	3	4
1	0	1	7	6
2	4	5	3	2
3	5	4	2	3
4	1	0	6	7

CODE B

	1	2	3	4
1	0	3	5	6
2	4	7	1	2
3	7	4	2	1
4	3	0	6	5

Figure 5. General Form of (3,2,1) Convolutional Code and Equivalence Example

3.2. Example

Consider once again the FSM defined in Figure 3, along with the conflict graph for the targeted errors. Only this time, a different partition (coloring) is chosen, as depicted in Figure 4. Moreover, an alternative convolutional code is selected, which is also shown in Figure 4. Following the exact same process as detailed in the previous section, the truth table for the new key generator is derived. We emphasize that the two alternative key generators are both capable of detecting all targeted errors. In the latter case, however, one may observe that the key bit K_1 is identical to the next state bit NS_1. Therefore, no additional hardware is necessary for generating K_1, as it can be tapped directly from NS_1. Thus, the key generator is reduced from 3 bits to 2 bits, without compromising its ability to detect all prescribed errors.

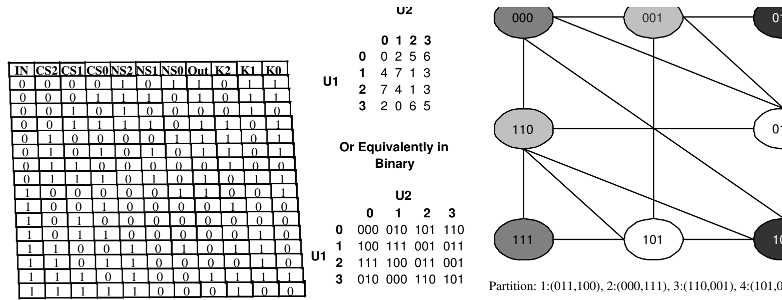
3.3. Problem Formulation

As mentioned above, there are two dimensions along which there is flexibility in developing the proposed CED method. The first dimension is the selection of the convolutional code to be used, while the second dimension is the selection of the partitioning (coloring) of the conflict graph. The previous example demonstrated that by choosing judiciously, some of the key generator bits may become identical to some of the next state logic bits, thus reducing the hardware. In this section we formulate the optimization problem of maximizing the number of identical bits.

Figure 5 shows the general form of a (3,2,1) convolutional code. Given this general form, several equivalent convolutional codes exist, among which we are called to choose the appropriate one. More specifically, two convolutional codes A and B are equivalent *if and only if* we can derive the transition matrix of B by consistently interchanging some or all of the elements in the transition matrix of A and vice versa. An example is given in Figure 5, where the transition matrix of B is derived by interchanging entry 1 with entry 3 and entry 5 with entry 7 in the transition matrix of A.

Regarding the selection of partitions and the assignment of keys, consider the example of Figure 4, where 4 distinct partitions are required. Let us define these partitions as P_1, P_2, P_3, and P_4. Moreover, let us define the eight states of the FSM, i.e. 000 through 111, as C_1, C_2, C_3, C_4, C_5, C_6, C_7, and C_8. Let C_iC_j be binary variables such that $C_iC_j = 1$ *if and only if* there is a valid transition

IEEE COMPUTER SOCIETY



IN	CS2	CS1	CS0	NS2	NS1	NS0	Out	K2	K1	K0
0	0	0	0	1	0	1	1	0	1	1
0	0	0	1	1	1	0	1	0	1	1
0	0	1	0	1	0	0	0	0	1	0
0	0	1	1	1	1	0	1	1	0	1
0	1	0	0	0	0	1	1	1	0	1
0	1	0	1	0	1	0	1	1	0	1
0	1	1	0	0	0	0	0	1	0	0
0	1	1	1	0	1	0	1	0	1	1
1	0	0	0	0	0	1	1	0	0	1
1	0	0	1	0	0	0	0	1	0	0
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	1	0
1	1	0	0	1	0	1	0	1	1	0
1	1	0	1	1	0	0	0	0	1	0
1	1	1	0	1	0	0	1	1	1	1
1	1	1	1	1	0	0	0	1	0	0

Figure 6. Convolutionally Encoded FSM Example with Output Logic Reuse

from state C_i to state C_j, otherwise $C_iC_j = 0$. Also, let P_iC_j be binary variables such that $P_iC_j = 1$ *if and only if* state C_j belongs to partition P_i, otherwise $P_iC_j = 0$. Since each state must belong to exactly one partition, $\forall j$, if $P_iC_j = 1$ for some i, then $P_kC_j = 0, \forall k \neq i$. Furthermore, if there is a transition between states C_i and C_j, then $P_kC_j \cdot P_kC_i = 0, \forall k$. Finally, let P_iP_j be the symbolic values of the entries in the transition matrix of the convolutional code, i.e., $P_1C_1 = 0$, $P_1C_2 = A_3$, $P_1C_3 = A_5$, $P_1C_4 = A_6$, $P_2C_1 = A_4$, $P_2C_2 = A_7$, $P_2C_3 = A_1$, $P_2C_4 = A_2$, $P_3C_1 = A_7$, $P_3C_2 = A_4$, $P_3C_3 = A_2$, $P_3C_4 = A_1$, $P_4C_1 = A_3$, $P_4C_2 = 0$, $P_4C_3 = A_6$, and $P_4C_4 = A_5$. We now need to assign a permutation of $(001, 010, 011, 100, 101, 110, 111)$ to variables A_1 through A_7, and to identify the appropriate values for every variable P_iC_j, so that a key bit K_t will be identical to a next state bit NS_t. Let $f(x, y)$ be a function returning the value of the y-th digit of x, written in binary form. Let also $g(k)$ be a function that returns the partition, j, where state k belongs to. For a key bit K_t to be identical to a next state bit NS_t, it must hold that:

$$\forall C_i, C_j, P_a, P_b, \text{ such that } P_aC_i = 1 \text{ and } P_bC_j = 1, C_iC_j \cdot f(P_aP_b, t) = C_iC_j \cdot f(C_j, t)$$

Or equivalently, the objective is to find the maximum number of variables t such that:

$$\sum_{\forall i,j} C_iC_j \cdot (f(P_{g(i)}P_{g(j)}, t) - f(C_j, t))^2 = 0$$

4. Detecting Errors in the Output Logic

The aforementioned CED method based on convolutional codes will detect the prescribed errors in the next state logic and the state register of the FSM. However, errors in the output logic of the FSM are not detected, since this information is not part of the key computation. In this section we demonstrate how these faults can also be detected, either by reusing the output logic for key generation, or by making the outputs part of the convolutional encoding and the key assignment.

4.1. Output Logic Hardware Reuse

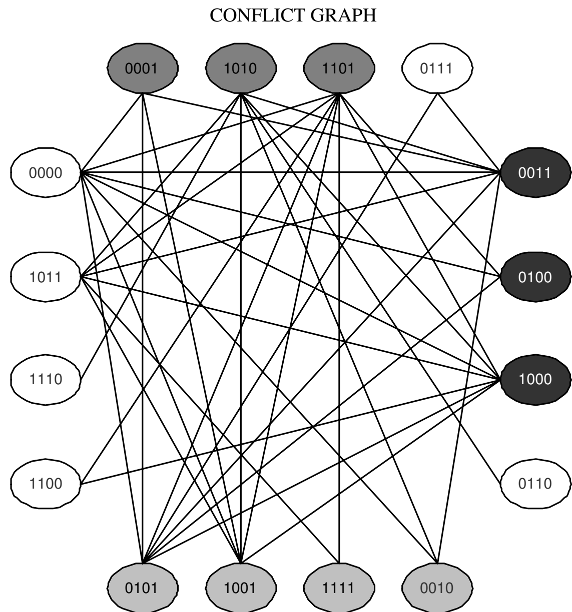
Consider once again the running example FSM, only this time an output has been added, as depicted in Figure 6. In order to detect errors affecting the output logic we add an extra constraint while selecting among alternative convolutional codes and graph partitions. More specifically, we require that the key generator functions that will be provided through hardware reuse, will include the output functions and not just the next state logic. In this way, errors affecting the outputs will also be detected, as they will result in an invalid sequence of key. For example, by choosing the convolutional code and the partitioning shown in Figure 5, the key generator bit K_0 is identical to the output bit Out. Thus, this particular portion of the key generator may be omitted and the corresponding bit will be tapped from the output logic of the FSM.

1063-6722/03 $17.00 © 2003 **IEEE**
IEEE COMPUTER SOCIETY

FSM & KEY GENERATOR

Current State					Next State/Output				Key		
IN	PS2	PS1	PS0	PREV. OUT	NS2	NS1	NS0	NEXT OUT	K2	K1	K0
0	0	0	0	0	1	0	1	1	0	0	0
0	0	0	0	1	1	0	1	1	0	1	0
0	0	0	1	0	1	1	0	1	0	1	1
0	0	0	1	1	1	1	0	1	0	0	1
0	0	1	0	0	1	0	0	0	0	1	1
0	0	1	0	1	1	0	0	0	0	0	1
0	0	1	1	0	1	1	0	1	1	1	0
0	0	1	1	1	1	1	0	1	1	1	0
0	1	0	0	0	0	0	1	1	0	1	1
0	1	0	0	1	0	0	1	1	0	0	1
0	1	0	1	0	0	1	0	1	0	0	0
0	1	0	1	1	0	1	0	1	0	1	0
0	1	1	0	0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0	0	0	1	0
0	1	1	1	0	0	1	0	1	0	1	0
0	1	1	1	1	0	1	0	1	1	1	1
1	0	0	0	0	0	0	1	1	1	0	1
1	0	0	0	1	0	0	1	1	1	1	0
1	0	0	1	0	0	0	0	0	1	0	0
1	0	0	1	1	0	0	0	0	1	1	1
1	0	1	0	0	0	0	0	0	1	1	1
1	0	1	0	1	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0	0
1	1	0	0	0	1	0	1	0	0	0	1
1	1	0	0	1	1	0	1	0	0	1	1
1	1	0	1	0	1	0	0	0	1	1	0
1	1	0	1	1	1	0	0	0	1	0	1
1	1	1	0	0	1	0	0	1	0	1	0
1	1	1	0	1	1	0	0	1	0	0	0
1	1	1	1	0	1	0	0	0	1	0	1
1	1	1	1	1	1	0	0	0	0	0	1

CONFLICT GRAPH



Partition: 1:(0111,0110,0000,1011,1110,1100), 2:(0101,1001,1111,0010), 3:(0011,0100,1000), 4:(0001,1010,1101)

Figure 7. FSM Example with Convolutionally Encoded Next State / Output Logic

However, we now run the danger of aliasing. This will happen when an error in the output logic results in an incorrect key sequence, which nevertheless is valid for some other sequence of FSM transitions. Although the probability is small, this problem will limit the effectiveness of the proposed method in detecting errors in the output logic. In this particular example, only 58% of the targeted errors in the output logic were detected through this method. We clarify that this aliasing problem does not exist for errors in the next state logic, since by construction of the code keys any error will result in an incorrect and invalid sequence of the convolutional code.

4.2. Combined Output/Next State Convolutional Encoding

An alternative way to detect not only errors affecting the next state but also the output logic of the FSM is to include the output logic in the actual assignment of the keys. This will resolve the aliasing problem of the previous method and will guarantee that any error in the either the next state or the output logic will result in an incorrect and invalid sequence of keys that will be detected by the convolutional code checker. On the down side, since computation of the keys is performed one clock cycle later, the outputs will also need to be latched for one clock cycle, incurring additional cost. More specifically, the keys will now be computed using the current inputs, the previous response of the next state logic, which is stored in the state register of the FSM, and the previous response of the output logic, which is stored in the newly added latches. Through this minor hardware addition, the method proposed in the previous sections may now be extended to detect errors in both the next state and the output logic of the FSM. Consider again the FSM example with one output bit which was used in the previous subsection. Following the method described above, the three state bits and the one output bit are all considered when assigning keys to convolutionally encode the next state and the output logic of the FSM. Figure 7 shows the conflict graph arising in this case and a coloring with 4 colors that will guarantee detection of all targeted errors. The same convolutional code shown in Figure 6 is employed and the assigned keys are shown in the table of Figure 7.

IEEE COMPUTER SOCIETY

5. Conclusions

We presented two extensions to a previously proposed CED method based on convolutional codes for FSMs. The first extension provides a framework for reducing the hardware cost incurred for generating the sequences of convolutionally encoded keys during normal FSM operation. The second extension provides the ability to detect errors in the output logic of the FSM, which cannot be detected through convolutional encoding of FSM states. In conjunction, the proposed extensions yield a reduced-cost, enhanced-effectiveness, convolutional code based CED method for FSMs. While we only discussed error detection with latency of one clock cycle, further hardware reduction at the cost of greater latency may also be also possible and is currently under investigation.

References

[1] M. Gossel and S. Graf, *Error Detection Circuits*, McGraw-Hill, 1993.

[2] S. J. Piestrak, "Self-checking design in Eastern Europe," *IEEE Design and Test of Computers*, vol. 13, no. 1, pp. 16–25, 1996.

[3] S. Mitra and E. J. McCluskey, "Which concurrent error detection scheme to choose?," in *International Test Conference*, 2000, pp. 985–994.

[4] G. Aksenova and E. Sogomonyan, "Design of self-checking built-in check circuits for automata with memory," *Automation and Remote Control*, vol. 36, no. 7, pp. 1169–1177, 1975.

[5] S. Dhawan and R. C. De Vries, "Design of self-checking sequential machines," *IEEE Transactions on Computers*, vol. 37, no. 10, pp. 1280–1284, 1988.

[6] N. K. Jha and S.-J. Wang, "Design and synthesis of self-checking VLSI circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 6, pp. 878–887, 1993.

[7] N. A. Touba and E. J. McCluskey, "Logic synthesis of multilevel circuits with concurrent error detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 7, pp. 783–789, 1997.

[8] E. Sogomonyan, "Design of built-in self-checking monitoring circuits for combinational devices," *Automation and Remote Control*, vol. 35, no. 2, pp. 280–289, 1974.

[9] C. Zeng, N. Saxena, and E. J. McCluskey, "Finite state machine synthesis with concurrent error detection," in *International Test Conference*, 1999, pp. 672–679.

[10] V. V. Danilov, N. V. Kolesov, and B. P. Podkopaev, "An algebraic model for the hardware monitoring of automata," *Automation and Remote Control*, vol. 36, no. 6, pp. 984–991, 1975.

[11] D. Das and N. A. Touba, "Synthesis of circuits with low-cost concurrent error detection based on Bose-Lin codes," *Journal of Electronic Testing: Theory and Applications*, vol. 15, no. 2, pp. 145–155, 1999.

[12] R. A. Parekhji, G. Venkatesh, and S. D. Sherlekar, "Concurrent error detection using monitoring machines," *IEEE Design and Test of Computers*, vol. 12, no. 3, pp. 24–32, 1995.

[13] S. Tarnick, "Bounding error masking in linear output space compression schemes," in *Asian Test Symposium*, 1994, pp. 27–32.

[14] P. Drineas and Y. Makris, "Non-intrusive concurrent error detection in FSMs through State/Output compaction and monitoring via parity trees," in *Design Automation and Test in Europe Conference*, 2003, pp. 1164-1165.

[15] L. P. Holmquist and L. L. Kinney, "Error detection with latency in sequential circuits," in *International Test Conference*, 1988, pp. 926–933.

[16] L. P. Holmquist and L. L. Kinney, "Concurrent error detection for restricted fault sets in sequential circuits and microprogrammed control units using convolutional codes," in *International Test Conference*, 1991, pp. 926–935.

[17] J. Wakerly, *Error Detecting Codes, Self Checking Circuits and Applications*, Elsevier, 1978.

[18] S. Lin and D. J. Costello Jr., *Error-Control Coding: Fundamentals and Applications*, Prentice-Hall, 1983.

1063-6722/03 $17.00 © 2003 **IEEE**

IEEE COMPUTER SOCIETY