SYNTHESIS OF SELF-TESTING FINITE STATE MACHINES FROM HIGH-LEVEL SPECIFICATION

Vishwani D. Agrawal  
Bell Labs, Lucent Technologies  
Murray Hill, NJ 07974  
va@research.bell-labs.com

R.D. (Shawn) Blanton  
Carnegie Mellon University  
Pittsburgh, PA 15213-3890  
blanton@ece.cmu.edu

Maurizio Damiani  
Synopsys, Inc.  
Mountain View, CA 94086  
maurizio@synopsys.com

Abstract

Current approaches to self test consist of adding hardware to the already synthesized circuits to transform them into autonomous finite state machines. If the circuit's own function is used for test generation and/or data compression, then the fault coverage and aliasing properties have to be obtained by simulation. In this paper, we give a function-level specification for the self-test problem. In the self-test mode, all primary inputs and outputs are latched. The circuit behaves as an autonomous finite-state machine, which executes an Euler walk of all states. Thus, each state is visited exactly once, with all states forming a closed path in the state transition graph of the test machine. This function is embedded in the high-level description of the given finite state machine. The self-test hardware thus undergoes the same optimization process as the machine hardware, with a chance of better area/timing optimization. Up to 100% fault coverage against all single/multiple faults can be achieved if the appropriate synthesis/optimization tools are used. On completion of self-test the signature, consisting of the states of all flip-flops, is shown to have an aliasing probability $2^{-m}$ when the circuit has $m$ flip-flops and the fault corrupts a single state transition.

1 Introduction

A synchronous circuit is said to be self-testing if it can operate in an autonomous mode, termed self-test. Its general architecture is shown in Figure 1. In the test mode, the combinational portion of the circuit is exercised by a sequence of input vectors, that are assumed to be a sufficient test set. The decision on the circuit correctness is based uniquely on the final state of the circuit. In other words, the circuit performs test generation and response analysis at least for its combinational part [1, 2, 3].

To obtain a self-testing circuit, we must synthesize a circuit that: 1) realizes the desired normal-mode function; 2) activates all modeled faults of the combinational part in the self-test mode; and 3) propagates the effects of each fault to state variables by some predefined test length such that the final state of the faulty machine is always different from that of the correct machine. In practice, requirement 3 is hardest to satisfy (unless a large memory is used to store the circuit response) and is usually compromised. Fault coverage in this context then represents the fraction of all faults that have been activated during the test, while aliasing is the fraction of the activated faults that are not observable (i.e., are masked) in the final state (signature).

To date, mostly heuristic approaches are used for the synthesis of self-testing circuits. An exception is the synthesis of Eschermann and Wunderlich [4] discussed in the next section. Most approaches consist of modifying an already synthesized normal-mode machine by augmenting it with a set of standard building blocks for test generation and/or response analysis. The structure shown in Figure 1(b) corresponds to the conventional test generator - data compressor (signature analyzer) approach. The tasks of test generation and data compression are separated. They are usually performed by linear feedback shift registers (LFSR) used as a pseudorandom pattern generator and a parallel signature analysis device, respectively. A reconfigurable register array (BILBO [5]) realizes these functions. This approach is used mostly for testing large combinational blocks, typically embedded in a data path. The hardware overhead required for testing a finite state machine, however, may be quite high although a high fault coverage can be guaranteed by implicitly performing exhaustive testing.

Alternatively, the use of the circuit's own function has been tried for test generation [6, 7]. Test generation and data compression circuitry are thus merged. The major drawback of these methods is that the circuit function can affect self-test in a rather unpredictable way, so that there is no a priori guarantee of adequate fault coverage. The coverage is obtained by simulation...
whenever the circuit structure is modified.

The general concept of synthesis for testability involves adding the test requirements to the function before it is implemented [8]. Synthesis allows optimization of the test hardware. Very large circuits have been synthesized through partitioning [9]. While this concept has been successfully applied to designs where tests are generated for external application, there is no insight available for a self-test design. The addition of a self-test specification at high level has several advantages. The self-test hardware is incorporated as part of the machine, and thus undergoes the same area, timing and power optimization steps as the normal-mode machine hardware.

The specification we present is at the finite-state machine description level. Along with the next-state function, a suitable next-input function is specified. The circuitry generating the next-input function takes control of the circuit's primary inputs during self-test. The architecture of the resulting self-test circuit is given in the next section. We show that any combinationally redundant implementation derived from such a specification can result in 100% fault coverage of single stuck-at faults, with a chance of being 100% self-testable against multiple faults if appropriate combinatorial synthesis (optimization) tools are used.

A separate signature register can be used for output compression, to insure low aliasing. Alternatively, the final state of the machine can be used as signature. We explore the link between aliasing and the structure of the machine state diagram, and develop heuristics to reduce the impact of aliasing on self-test if the latter approach is taken.

2 Self-Test Specification

The synthesis of a finite state machine typically starts from the specification of the next state and output functions at a symbolic level. The fundamental synthesis steps are symbolic minimization, state assignment (with a possible input/output encoding), and optimization of the combinational circuit. State assignment can be specialized to two-level [10, 11] or multiple-level combinational logic implementation [12]. Current two-level logic optimizers insure 100% testability against multiple stuck-at faults of the resulting logic. This property can be maintained through subsequent multiple-level algebraic factorization. Boolean optimization can also insure 100% testability against single stuck-at faults, as well as a generally smaller final area.

Eschermann and Wunderlich [4] synthesize the structure of Figure 1(b). They recognize that the next-state function of the object machine and the test machine may overlap. The next-state logic of the object machine is reduced by treating the overlap transitions as don't cares. This results in a self-test machine that is smaller than the traditional post-synthesis implementation. Our proposal differs in several significant ways. First, pattern generator in [4] is restricted to be a LFSR. In our approach, no such restriction is used, allowing a better overlap with the object function. Second, the multiple input LFSR used in [4] for response compaction is not included in the optimization process. In our approach, the entire test function is merged with the object function before synthesis.

Our target architecture is shown in Figure 2. In normal mode, the input lines are externally controlled, while in the self-test mode internal control takes over. In self-test mode the combinational block is only required to generate the next-input entry, that is, we
need a latch for any external input. All inputs of the combinational logic come from latches and all outputs of combinational logic feed into latches. However, no separation between the input and output latches is required. This architecture differs from the circular self-test method of forming a ring of flip-flops [13]. As will become clear in the following discussion, our method forms a ring in the state diagram. Thus, the test covers all states. Such coverage may not be guaranteed in the circular self-test. The architecture of Figure 2 may also appear similar to another known technique in which the pattern generator and signature register are combined into one set of flip-flops [6]. In that method, however, no functional changes are made to the combinational logic to ascertain that all tests will be generated. In fact, the pattern generator behaves as a non-linear feedback shift register, whose capability of generating patterns strongly depends on the feedback logic. To correct this situation, a modification has been suggested in the literature [14]. That technique, known as ring-like testing, adds a separate combinational logic block whose outputs are exclusive-OR-ed with the logic under test before feeding back into flip-flops. The function of the added logic is to linearize the feedback. Our technique has similarity to this method. However, there are significant differences. We add a test function, and not an explicit logic block. Our function is added even prior to the state assignment. Based on the optimality of the synthesis system and the given object function, a variety of test mode functions, such as a feedback shift register or a counter, are possible. Combinational logic, including the implicit test logic, can be exhaustively tested and theoretical estimates on the aliasing probability have been obtained.

Figure 3 shows the state diagram of a simple synchronous machine. A possible approach to the self-test of the machine is as follows. After an implementation is chosen, a test generator is used to produce a test set for the combinational part. Each test vector for the combinational portion is a state-input combination for the circuit, and is represented by an arc in the state transition diagram of the implemented machine.

A test set is thus represented by a set of arcs in the implemented machine, and a test sequence is a path in the state diagram that includes the test set. The major disadvantages of this approach are: 1) it requires a prior implementation of the functional logic (very little can be said about a test set from the high-level description); 2) test generation has to be used as an intermediate step; 3) the functional logic and the next-input logic cannot be easily merged without the possibility of invalidating the previously derived test set. The obtained results are, in this sense, sub-optimal. Furthermore, the description does not support design improvements, i.e., any modifications may result in coverage loss.

2.1 High-Level Specification of Self-Test

In a state diagram, the state/input combinations that are possible at the input of the combinational logic represent its care set. A semi-symbolic description of this care set is provided by the state transition table. It is known from combinational optimization theory [15] that if the combinational circuit is irredundant with respect to this care set, then a complete test set exists for the circuit, and it is contained in the care set. We thus formulate the following proposition:

**Proposition 2.1** For a combinatorially irredundant implementation of a finite state machine, there is an
arc set $E_t$ contained in the symbolic state diagram that is a complete test for the combinational logic.

Unfortunately, from the symbolic description it is not possible to determine an implementation-independent test set $E_t$ such that $|E_t| < |E|$. Therefore, any self-test specification aiming at 100% fault coverage should contain all arcs of the symbolic description. We require the following definitions.

**Definition 2.1** An exhaustive walk of a directed graph is a walk that traverses each arc at least once.

**Proposition 2.2** Any exhaustive walk of the state transition graph describes an implementation independent sufficient self-test.

**Proof:** From Proposition 3.1, an exhaustive walk covers at least one test set of any irredundant implementation.

**Definition 2.2** An Euler walk of a directed graph is a walk that traverses each arc exactly once.

Any exhaustive walk can be regarded as an Euler walk in an extended graph, in which each arc traversed $m > 1$ times is replaced by $m$ arcs, each traversed once. A graph that has an Euler walk is termed an Eulerian graph. Not all digraphs, however, are Eulerian. The above observation shows that any exhaustive walk is just an Euler walk in some extended graph. Ways of augmenting a graph to make it Eulerian stem from the following theorem from graph theory [16, 17]:

**Theorem 2.1** Let $\text{indegree}(v)$ and $\text{outdegree}(v)$ denote the number of fanin and fanout arcs of a vertex $v$ of a digraph. A graph is Eulerian if and only if for each vertex $v$ $\text{indegree}(v) = \text{outdegree}(v)$, with the possible exception of two vertices $v_1$ and $v_2$, for which $\text{indegree}(v_1) = \text{outdegree}(v_1) - 1$ and $\text{indegree}(v_2) = \text{outdegree}(v_2) + 1$, respectively.

If for all vertices $v$ $\text{indegree}(v) = \text{outdegree}(v)$ all Eulerian walks are cyclic, i.e., they end at the vertex where they started. An arc could be removed from such a graph (the arc joining the last vertex of the walk to the first vertex) and it would still be possible to perform a complete graph traversal. This corresponds to the situation in Theorem 2.1 where two vertices have excess indegree and outdegree, respectively. The resulting walk, however, will no longer be cyclic.

The simplest way to make a graph Eulerian is to add extra arcs from the vertices having $\text{outdegree} < \text{indegree}$ to those having $\text{outdegree} > \text{indegree}$. Note that, in general, a) there is no unique way to make a graph Eulerian, and b) for a given Eulerian graph, there is no unique Euler walk.

**Example 1.** The graph of the machine in Figure 3 is Eulerian. Any walk must begin in $s_0$ and end in state $s_2$. If a complete cycle is to be performed, then an arc must be added, joining $s_2$ to $s_0$. In this case, however, this is not necessary. Two possible Euler walks for the machine are:

<table>
<thead>
<tr>
<th>Present state,</th>
<th>Next state,</th>
<th>Present state,</th>
<th>Next state,</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>input</td>
<td>input</td>
<td>input</td>
</tr>
<tr>
<td>st0, in0</td>
<td>st0, out0, in1</td>
<td>st0, in0</td>
<td>st0, out0, in1</td>
</tr>
<tr>
<td>st0, in1</td>
<td>st1, ---, in1</td>
<td>st0, in1</td>
<td>st1, ---, in1</td>
</tr>
<tr>
<td>st1, in1</td>
<td>st1, out1, in2</td>
<td>st1, in1</td>
<td>st1, out1, in0</td>
</tr>
<tr>
<td>st1, in2</td>
<td>st1, out1, in0</td>
<td>st1, in0</td>
<td>st2, out1, in0</td>
</tr>
<tr>
<td>st1, in0</td>
<td>st2, out1, in0</td>
<td>st2, in0</td>
<td>st2, out1, in1</td>
</tr>
<tr>
<td>st2, in0</td>
<td>st2, out1, in3</td>
<td>st2, in1</td>
<td>st3, out1, in1</td>
</tr>
<tr>
<td>st2, in3</td>
<td>st2, out1, in1</td>
<td>st3, in1</td>
<td>st3, out1, in0</td>
</tr>
<tr>
<td>st2, in1</td>
<td>st3, out1, in1</td>
<td>st3, in0</td>
<td>st0, ---, in2</td>
</tr>
<tr>
<td>st3, in1</td>
<td>st3, out1, in0</td>
<td>st0, in2</td>
<td>st1, ---, in2</td>
</tr>
<tr>
<td>st3, in0</td>
<td>st0, ---, in2</td>
<td>st1, in2</td>
<td>st1, out1, in3</td>
</tr>
<tr>
<td>st0, in2</td>
<td>st1, ---, in3</td>
<td>st1, in3</td>
<td>st2, out1, ---</td>
</tr>
<tr>
<td>st1, in3</td>
<td>st2, out1, ---</td>
<td>st2, in3</td>
<td>st2, out1, ---</td>
</tr>
</tbody>
</table>

It is hard to predict the cost of a particular walk, since it strongly depends on the synthesis process used. In case the input field is already encoded, however, a simple heuristic is to "mimic" the behavior of a shift register (possibly, with linear feedback). Thus, most next-input functions are simply reduced to wires, substantially reducing the amount of extra hardware required for the next-input function generation.

3 **Aliasing**

The Euler walk approach is used to guarantee the built-in generation of a sufficient self-test. A separate signature register can be used to determine the circuit correctness. If the state of the machine is to be used as signature, then it is obviously necessary to latch the circuit outputs, or otherwise the faults only observable at primary outputs will be unobservable from the final signature. It is also necessary that a functional dependence be established between the next state of the machine and the latched outputs. Otherwise, the latched information will be lost.

3.1 **Single Corrupted Arc (SCA)**

Consider an $m$-bit self-testing FSM, and assume first that, during self-test, it cycles through all $N = 2^m$ states. Under a SCA model, aliasing can occur only
when the state diagram is transformed as shown in Figure 4, i.e., the starting (and end) state are included in a cycle of length dividing $2^m$. There are $2^m \times (2^m - 1)$ possible faulty machines with a single corrupted edge. Those resulting in a cycle of length $2^i$ are exactly $2^i$. Therefore, assuming each corrupted edge equally likely, the aliasing probability is

$$P_{al} = \frac{1}{2^m(2^m - 1)} \sum_{i=0}^{m-1} 2^i = \frac{2^m - 1}{2^m(2^m - 1)} = \frac{1}{2^m}$$  \hspace{1cm} (1)$$

Surprisingly, we obtain the same result as in the case of signature analysis.

Figure 4: The state diagram of a modulo-8 counter with a single corrupted (dashed) edge. The faulty state diagram represents a modulo-4 counter. The fault is aliased because 4 divides 8.

### 3.2 Multiple Corrupted Arcs (MCA)

Suppose that multiple arcs of a sequencer are corrupted. We assume that all corruptions are equally likely. Since each entry may be corrupted in $N - 1$ ways, and there are $N$ entries, there are

$$N_F = N^N - 1$$  \hspace{1cm} (2)$$
corrupted FSMs. Aliasing occurs whenever the initial state is in a cycle that divides the test length. Let $L$ denote the length of a cycle. We now count the cycles of length $L$ that can be made out of $N$ states. For each cycle, we count the number of faulty FSMs containing that cycle.

There are $(N - 1; L - 1)$ ways of selecting $L$ states out of $N$, if we make sure the start state is included. For each selection, one can form $(L - 1)!$ cycles starting with the start state. Hence, one can form in all $(L - 1)!(N - 1)/(N - L)!$ distinct cycles. Each cycle, however, may belong to several distinct faulty machines. It is possible to count such machines. In the next-state table of the FSM, the cycle identifies uniquely $L$ entries. The remaining $N - L$ entries can take $N$ symbolic values each. There are thus $N^{N-L}$ FSMs containing one prescribed cycle of length $L$. The number of machines that can cause aliasing is:

$$N_{al} = \sum_{L; L \text{ divides } N} N^{N-L} \frac{(N-1)!}{(N-L)!}$$  \hspace{1cm} (3)$$

Note that the original machine is included in this count. By extracting the term $N^N$ from the right-hand side of Eq. (3), and using the identity $(N-1)! = N!/N$ the aliasing probability is then expressed by the ratio:

$$P_{al} = \frac{N_{al}}{N_F} = \frac{N^N}{N^N - 1} \frac{1}{N} \sum_{L; L \text{ divides } N} \frac{(N)!}{N^L(N-L)!}$$  \hspace{1cm} (4)$$

It is possible to find a bound to $P_{al}$ by observing that each term in the sum of Eq.(4) is upper-bounded by 1. Hence,

$$P_{al} \leq \frac{N^N}{N^N - 1} \frac{N_d(N)}{N}$$  \hspace{1cm} (5)$$

where $N_d(N)$ represents the number of divisors of $N$. In the present case, $N = 2^m$, there are then $m + 1$ divisors of $N$, namely, $L = 2^i; i = 0, \ldots, m$. Hence,

$$P_{al} \leq \frac{N^N}{N^N - 1} \frac{(m+1)/N}{N}$$  \hspace{1cm} (6)$$

A slightly more accurate bound can be obtained by observing that

$$\frac{N!}{(N-L)!} = N \cdot (N-1) \cdots (N-L+1) \leq N^{L-1}(N-L+1)$$  \hspace{1cm} (7)$$

Hence,

$$P_{al} \leq \frac{N^N}{N^N - 1} \frac{1}{N} \sum_{L \text{ divides } N} \frac{N - L + 1}{N}$$  \hspace{1cm} (8)$$

$$P_{al} \leq \frac{N^N}{N^N - 1} \left( \frac{m - 1}{N} + \frac{m + 2}{N^2} \right)$$  \hspace{1cm} (9)$$

Figure 5 plots the actual behavior of $N \cdot P_{al}$ for integers $m \leq 24$. The plot indicates that the actual $P_{al}$ behaves roughly as

$$P_{al} \approx \frac{1}{2} \frac{(m+1)}{N}$$  \hspace{1cm} (10)$$

A breakdown of the sources of aliasing for different counts of corrupted arcs looks complex. Moreover, without an actual implementation, it is difficult to predict the impact of a hardware fault on the state diagram and, consequently, on the count of corrupted arcs.
3.3 Non-Maximum Length Machines

In this case, other aliasing sources need to be considered. Figure 6 shows the state diagram of a 3-bit implementation of a modulo-6 counter. States $G, H$ in particular belong to the implementation only. Figure 6 also shows two corrupted-arc fault mechanisms, resulting in a multiple corrupted arc fault. The fault introduces aliasing because it creates a loop $(A, B, G)$ whose length divides the original counter period. Unlike previous cases, the new loop uses states $(G)$ not belonging to the original circuit. A second source of aliasing may be caused by the corruption only of arcs not belonging to the original state diagram. This would be represented by a fault consisting only of the dotted arc in Figure 6. In the light of Section 2, however, we remark that one such event cannot be caused by single or multiple stuck-at faults.

We now modify the analysis of the previous sections to account for the presence of extra states in the implementation of non-maximum length machines. We consider only the case of multiple corrupted arcs. Let $C$ denote the counter period, and let $m$ denote the number of bits used in its implementation. Again, $N = 2^m$. The number of faulty machines is still given by Eq. (2). We now count the number of faulty machines that alias with the original machine. A faulty machine is aliased if it cycles through $L$ states from the start state, where $L$ divides $C$. The number of cycles of length $L$ that can be constructed by choosing $L - 1$ out of $N - 1$ states is given again by $(N - 1)!/(N - L)!$. and the total number of machines that alias the original counter is

$$N_{ai} = \sum_{L; L \text{ divides } C} N^{N-L} \frac{(N-1)!}{(N-L)!}$$  \hfill (11)

The sum (11) is more complex than the one of Eq. (5), because now $C \neq N$. By transformations similar to those leading to Eq. (6), however, one gets:

$$P_{al} = \frac{N^N}{N^N - 1} \frac{1}{N} \sum_{L; L \text{ divides } C} \frac{N!}{N^L(N - L)!}$$  \hfill (12)

and the bound

$$P_{al} \leq \frac{N^N}{N^N - 1} \frac{N_d(C)}{N}$$  \hfill (13)

Equation (13) yields several suggestions. First, since the numerator depends only on $C$, aliasing is certainly made smaller by choosing large values of $N$. This result is not obvious, because adding registers implies adding states and therefore sources of aliasing. Second, it indicates that choosing a prime number for $C$ would minimize, for a given value of $N$, the probability of aliasing. This suggestion is confirmed numerically by the plots of Figure 7, indicating the actual aliasing probability for different values of $C$, when $m = 14; N = 8192$. Unfortunately, the function $N_d(C)$ is not simple to compute, and it is not very informative about the worst-case behavior of aliasing. We conjecture the bound to be

$$P_{al} \leq \frac{N^N}{N^N - 1} \cdot \frac{1}{N} \cdot \sqrt{2} \left(1 + \frac{\log(C)}{2}\right)^2$$  \hfill (14)

Figure 7 shows this bound for all values of $C \leq N$, assuming $N = 8192$. 
4 Optimization

4.1 State-Output Merging

It was observed in Section 2 that, if a circuit is to be self-testing, its outputs must be latched. The state and output registers can thus be viewed as an extended state for the machine, the actual output being now just a portion of the machine's future state. A Moore description of the machine is therefore most appropriate.

Example 2. The following represent Moore machine descriptions of the state transition table of Example 1. Included is the next-input function as specified by the second Euler walk of Example 1:

<table>
<thead>
<tr>
<th>Present state, input</th>
<th>Next state, input</th>
<th>Present state, input</th>
<th>Next state, input</th>
</tr>
</thead>
<tbody>
<tr>
<td>st0, out0, in0</td>
<td>st0, out0, in1</td>
<td>st0, out0, in0</td>
<td>st0, out0, in1</td>
</tr>
<tr>
<td>st0, out0, in1</td>
<td>st1, out1, in1</td>
<td>st0, out0, in1</td>
<td>st1, out1, in1</td>
</tr>
<tr>
<td>st1, out1, in1</td>
<td>st1, out1, in2</td>
<td>st1, out1, in1</td>
<td>st1, out1, in2</td>
</tr>
<tr>
<td>st1, out1, in2</td>
<td>st2, out1, in0</td>
<td>st1, out1, in2</td>
<td>st2, out1, in0</td>
</tr>
<tr>
<td>st2, out1, in0</td>
<td>st2, out1, in3</td>
<td>st2, out1, in0</td>
<td>st2, out1, in3</td>
</tr>
<tr>
<td>st2, out1, in3</td>
<td>st2, out1, in1</td>
<td>st2, out1, in3</td>
<td>st2, out1, in1</td>
</tr>
<tr>
<td>st2, out1, in1</td>
<td>st3, out1, in1</td>
<td>st3, out1, in1</td>
<td>st3, out1, in1</td>
</tr>
<tr>
<td>st3, out1, in1</td>
<td>st3, out1, in0</td>
<td>st3, out1, in1</td>
<td>st3, out1, in0</td>
</tr>
<tr>
<td>st3, out1, in0</td>
<td>st0, out0, in0</td>
<td>st3, out1, in0</td>
<td>st0, out0, in0</td>
</tr>
<tr>
<td>st0, out0, in0</td>
<td>st1, out1, in3</td>
<td>st0, out0, in0</td>
<td>st1, out1, in3</td>
</tr>
<tr>
<td>st1, out1, in3</td>
<td>st2, out1, in3</td>
<td>st1, out1, in3</td>
<td>st2, out1, in3</td>
</tr>
<tr>
<td>st2, out1, in3</td>
<td>st3, out1, in3</td>
<td>st2, out1, in3</td>
<td>st3, out1, in3</td>
</tr>
</tbody>
</table>

Note that, for an arbitrary incompletely specified Mealy machine, there may be many Moore-style descriptions. An Euler walk of the Moore machine could be significantly longer than the Euler walk of the original machine. Therefore, the self-test specification should be carried out at the Mealy-machine description level. An optimal implementation of a Moore machine is known to take possibly fewer registers, and generally a smaller size than the original Mealy implementation, assuming latched outputs. This is because past outputs implicitly provide information on the current state.

4.2 Input-State Merging

Having reduced the machine to a Moore machine, it is clear that the only "wasted" function in this framework is the next-input function, which is used only in self-test. It is therefore desirable to keep the complexity of this function to a minimum. The optimum choice would be to make the next-input field coincide with the next-state (or next-output) field. For a given Euler walk, this is not generally possible.

Figure 8: An Euler walk for the Moore-style description of the machine of Example 1.

Example 3. The second Euler walk of Example 1 corresponds to the walk shown in Figure 8 for a corresponding Moore machine. As can be seen, state st0, out0 is exited with two different input symbols, in1 and in2. Therefore, a one-to-one correspondence between the state and the input entry is not possible.

The above example shows, however, the reason why this merging is seldom possible, namely the existence of states that are crossed more than once by the walk. A solution to this problem is to augment the state diagram by creating equivalent states, and making each state crossed exactly once.
4.3 Register Sharing

The above machine transformations dealt especially with the optimization of combinational logic. The next-input registers, however, are left unused during normal operation. If, as most frequently occurs, the input of our target finite-state machine is the output of another machine, then it is possible to share those registers in the self-test mode. We may consider two possible architectures for sharing registers in self-test mode. The first architecture refers to the case in which the outputs of the first machine feed directly into inputs of the second machine. The second one refers to the case in which the communication occurs through pipeline latches.

5 A Simple Design Illustration

Consider incorporating the BIST function described in the last section into a 2-bit counter. Figure 9 shows the STG for the counter. It successively counts through states $st_0$, $st_1$, $st_2$, $st_3$, $st_0$, ... when the primary input $I = 1$ and retains its current state when $I = 0$. The counter produces an output $Z = 1$ only when entering or residing in the fourth count state $st_3$. The BIST function described above is incorporated into the counter by adding a primary input $TEST$. Figure 10 shows the STG for the resulting self-test counter. The new STG contains twice as many states as the original STG reflecting the additional state variable required to replace $I$ during self-test. Figure 10a corresponds to normal function of the counter. The pairing of states in Figure 10a is used to indicate their equivalence during normal operation. Because the original counter's STG is Eulerian, all edges can be traversed in a single cycle during self-test as shown in Figure 10b.

**Counter Implementation:** Using the SIS [15] synthesis tools, gate-level implementations of the original and self-testing counter of Figure 11 were obtained. Figure 11 also shows a traditional BIST architecture for the counter. (Note the shaded flip-flops of Figure 11c used for test pattern generation serves as the counter's state memory as well.) The total number of gate inputs and flip-flops for the three counter circuits are used as a rough measure of circuit size (see Table 1). Table 1 indicates that the self-testing counter has twice as many gate inputs and one more flip-flop than the counter without BIST function. But the traditional BIST architecture for the counter is more than twice the size of our synthesized self-testing counter. We believe this small example illustrates the potential advantages in terms of the overhead incurred over traditional BIST implementations.

It may appear that the structure of our self-testing counter resembles a Circular BIST (CBIST) structure. CBIST, developed by Kransniewski and Pilarski [13], converts a set of registers into a circular LFSR that is used for both test pattern generation and response compaction. In our approach, like CBIST, the next-state memory in the self-testing counter is used for both test
pattern generation and response compaction but the structure corresponding to these two functions is not specified. In CBIST, registers are converted to LSFRs with a predetermined structure without any regard for how its function may overlap with any other function of the object machine. Moreover, the test pattern generation characteristics of CBIST are highly dependent upon the function of the object machine. As a result, long test sequences (four times the size of an exhaustive set) are required to ensure that a good test set is generated by CBIST. Our method is superior in that a good test set is obtained through the traversal of the Eulerian cycle.

Self-Test Operation: Self-test operation is initiated by resetting the counter and then asserting the test input to \( TEST = 1 \). Clocking the machine eight times causes the counter to traverse the Eulerian cycle shown in Figure 10b, thereby causing all possible input patterns to be applied to the counter’s primary and state inputs.

To obtain a general measure of the counter’s testability, the self-test operation was fault simulated for manufacturing defects that can be modeled as single stuck-line (SSL) faults [18]. We assumed that the primary output \( Z \) was observable after every clock period and that the state of the counter was observable only at the end of self-test operation. The results of the simulation showed that 80% of all the SSL faults are detected by the self-test operation. The undetected faults are all dependent upon the asserted \( TEST \) input and are readily detectable by the normal operation of the counter. For any self-testing circuit, a set of such faults will always exist.

6 Conclusion

With the current advancements in synthesis, a self-test technique of the type we have presented should prove useful. Basically, we have an alternative to the conventional LFSR or counter type of pattern generators. Our pattern generators are functionally similar to those at a higher level, however, state assignment will usually make them quite different in implementation. Thus combinational logic is fully tested. Another innovation the new technique offers is in combining the signature function within the same set of flip-flops, for which we have analytic bounds on aliasing. The strength of this methodology lies in the use of synthesis tools, which would allow optimal sharing of the functional and test hardware.

The technique, as presented, is directly applicable to small control machines. For large systems, however, a partitioning approach needs to be evolved. The future
architectural investigation will consider system partitions, with each partition simultaneously running self-test. Signatures will have to be scanned out. For that, scan-like functions can be synthesized in the partitions as described in some recent work [9].

References


