VLSI DESIGN
1996, Vol. 4, No. 3, pp. 149–165
Reprints available directly from the publisher
Photocopying permitted by license only

PGEN: A Novel Approach to Sequential Circuit Test Generation

WEN-BEN JONE[a,*], NIGAM SHAH[b,†], ANITA GLEASON[c] and SUNIL R. DAS[d,‡]

[a]*Department of Computer Science and Information Engineering, National Chung-Cheng University, Chiayi, Taiwan, ROC;* [b]*Department of Computer Science, New Mexico Tech, Socorro, NM 87801, USA;* [c]*Department of Computer Science, Bloomsburg University, Bloomsburg, PA 17815, USA;* [d]*Department of Electrical Engineering, University of Ottawa, Ottawa, ON K1N 6N5, Canada*

A novel approach, called PGEN, is proposed to generate test patterns for resettable or nonresettable synchronous sequential circuits. PGEN contains two major routines, Sequential PODEM (S-PODEM) and a differential fault simulator. Given a fault, S-PODEM uses the concept of multiple time compression supported by a pulsating model, and generates a test vector in a single (yet compressed) time frame. Logic simulation (included in S-PODEM) is invoked to expand the single test vector into a test sequence. The single test vector generation methodology and logic simulation are well coordinated and significantly facilitate sequential circuit test generation. A modified version of differential fault simulation is also implemented and included in PGEN to cover other faults detected by the expanded test sequence. Experiments using computer simulation have been conducted, and results are quite satisfactory.

Keywords: Resettable or nonresettable synchronous sequential circuits, PODEM and sequential PODEM or S-PODEM, pulsating test generation or PGEN, differential fault simulator, multiple time compression.

1. INTRODUCTION

Sequential circuit testing has been recognized as the most difficult problem in the area of fault detection. The difficulty comes from the existence of memory elements. With memory elements, such as latches or flip-flops, the circuit output depends not only on the current inputs but also on the operation history (circuit states). Of course, it is possible to facilitate sequential circuit testing by adding some extra hardware, which enhances the controllability and observability of the circuit [22]. However, the test hardware increases hardware overhead and can degrade circuit performance. Thus, before using valuable chip space, test generation without adding extra hardware should be tried.

In this work, a novel approach called PGEN is proposed to generate test patterns for synchronous sequential circuits with or without a reset line. Instead of unfolding the sequential circuit into an iterative

*E-mail: csiwbj@ccunix.ccu.edu.tw
†Present address: Information Technology Services, AT & T Universal Card Services, Jacksonville, FL 32220, USA.
‡E-mail: srdpb@acadvml.uottawa.ca

logic array, for a given fault, PGEN uses the concept of time compression and synthesizes in one time frame a single test vector representing the compressed form of multiple time frames. The single vector is then expanded into a test sequence by a logic simulator guided by dynamic or static cost analysis. In general, when a set of test patterns is applied to a sequential circuit, each signal line of the machine can be static or pulsating (changing logic values). The static line values can be represented by conventional logic and fault models such as logic 1, 0, D, $\bar{D}$ [20], while the pulsating line values are represented with the model value P to reflect the circuit behavior in a single and yet compressed time frame. For this reason, the name of the proposed test generation method is PGEN (Pulsating Test Generation). Using the pulsating logic model, the sequential circuit behavior under a test sequence can be faithfully described in a single time frame.

The PODEM algorithm [10] used for combinational circuit testing is transplanted and upgraded to support the pulsating model for test pattern determination. PGEN contains two major parts. The first part, S-PODEM is used to synthesize a single test vector based on the compressed time frame, and expand the single vector into a test sequence. The second portion of PGEN is a modified version of the differential fault simulator [8] and is implemented to cover all faults detected by the test sequence developed by S-PODEM. In fact, the PGEN approach is a compromise between simulation-based and iterative logic array test methods. It utilizes the benefits of deterministic test generation methods to ascertain the required input signals for sensitizing and propagating the faults; however, the search for test patterns is greatly simplified by the pulsating model as will be shown later. PGEN also needs simulation; but, unlike conventional simulation-based methods which determine test patterns entirely by simulation, PGEN uses simulation only for test pattern expansion. In summary, the philosophy of PGEN is to utilize the advantages of simulation-based and iterative logic array testing, and avoid the difficulties of both approaches.

The paper is organized as follows: Section 2 gives background on sequential circuit testing, and Section 3 explains the pulsating model. The PGEN algorithm and its special attributes are presented in Section 4, and Section 5 describes S-PODEM and its routines. Simulation and results are given in Section 6. Lastly, conclusions are given in Section 7.

2. BACKGROUND

To increase the efficiency of sequential test generators, algorithms proposed have utilized different techniques such as backward justification [4] [14–16]; concurrent fault simulation [1]; and use of previous state information [2] [14]. Three different approaches have been considered for sequential circuit test generation. They are

1. iterative test generation method;
2. simulation-based method; and
3. functional test generation method.

In the *iterative test generation* approach [2] [4–5] [9] [13–17] [19], the combinational model for a sequential circuit is constructed by regenerating the feedback signals from previous time copies of the circuit. Thus, the timing behavior of the circuit is approximated by iterative combinational levels. Topological analysis algorithms that activate faults and propagate the effect through these multiple copies of the combinational circuit are used to generate tests. This approach can be further divided as forward time, reverse time or a combination of forward and reverse time test generation.

In the *simulation-based* approach [1] [6] [21] algorithms start with a random vector and simulate the circuit. From the simulation result, a cost function is computed. This cost is defined to be below a threshold only if the simulated vector is a test. If the vector is not a test, i.e., the cost is high, then cost reduction by gradual changes in the vector leads to a test.

All methods described above perform test generation based on circuit structures. Sequential circuit testing based on functions, especially using state tables, can also be found in [7] [12] [18] and will not be further discussed since the scope of this paper is

mainly restricted to test generation based on circuit structure.

3. MOTIVATION AND MODELING

The PGEN approach considers single stuck-at fault detection with a special 11-value model, denoted as the P-model. The values of the P-model reflect the logic values (or circuit behavior) of different lines for both the fault-free and faulty circuits, when test patterns are applied. Section 3.1 justifies the motivation behind the P-model, and details of the P-model are given in Section 3.2.

3.1 Motivation

The basic idea of PGEN originated from the desire to compress the time frame by a reasonable method, rather than unfolding the sequential circuit. After careful study of a number of sequential circuits, it was observed and concluded that most of the primary inputs remain at some static values during test generation for a particular fault. Only a few primary inputs exhibit pulsating behavior. That is, a considerable number of lines will remain at some stationary value during the entire test generation process for a particular fault.

Static line set (SLS): A line which remains at a static value (both fault-free and faulty circuits) during the entire test operation is regarded as a static line. The set of all such static lines constitutes a static line set, *SLS*.

There will be other lines which will experience different logic values, when test patterns are applied. To represent such a behavior in a compressed time frame, a new symbol is required. In the proposed P-model, this symbol is represented by P which hints that a particular line will change its value during the test generation process. Thus, the standard fault model is enhanced by this addition.

Pulsating line set (PLS) : When a line L experiences a change in its logic value (either fault-free or faulty circuit, or both) during the test experiment, L is known as a pulsating line. The set of all such pulsating lines is regarded as a pulsating line set, *PLS*.

Due to the presence of a fault, several effects may take place. If $L \in SLS$ then there are two possibilities:

1. L gets the same static value for both faulty and fault-free circuits. By the definition of *SLS*, this value does not change with time.
2. L has a different static value for faulty and fault-free circuits. In the fault-free circuit it may get 0 (1), whereas in the faulty circuit it may receive a 1 (0).

Assume that $L \in PLS$, then L may experience pulsating behavior in either the faulty or fault-free circuit or both. There are three possibilities:

1. L experiences pulsating behavior in both the fault-free and faulty circuits.
2. L experiences pulsating behavior in the fault-free circuit, but due to the fault effect it remains at a constant value in the faulty circuit. Thus, in the fault-free circuit it has value P, but in the faulty circuit it may have 0 or 1.
3. L keeps a static value (0 or 1 logic value) in the fault-free circuit, but it experiences changes in line value for faulty circuit.

3.2 Logic Model and Operations

The logic model of PGEN consists of 11 values: 0, 1, D (1/0), $\bar{D}$(0/1), X, P, $P0$ (P/0), $P1$ (P/1), $1P$ (1/P), $0P$(0/P), and PP(P/P). Here, D and $\bar{D}$ have the same meaning as the one used in conventional methods [20], i.e., D denotes logic 1 in the good circuit and logic 0 in the faulty circuit. Similarly, $\bar{D}$ indicates logic value 0 in the good circuit and 1 in the faulty circuit. The P value is used for any pulsating signal. $P0$ ($P1$) indicates a pulsating signal in the good circuit and static value 0(1) in the faulty circuit, and $0P$ ($1P$) indicates 0(1) in the good circuit and P in the

faulty circuit. Furthermore, *PP* represents a signal pulsating in the good machine with another signal pulsating (assuming different pulsating behavior) in the faulty machine. As usual, *X* denotes an unknown value. Once the logic model is determined, different logic operations need to be redefined. Table I determines the relationship between inputs and outputs for a universal logic gate, a two-input NAND gate, from which all other combinational logic operations can be generated.

4. TEST GENERATION

4.1 The PGEN Algorithm

The main structure of PGEN is similar to any standard test generation algorithm which starts with initialization of data structures. It reads the circuit description file, in ISCAS 1989 benchmark circuit format [3], and prepares the internal data model. From the fault list file, it prepares the fault list for the circuit under test (CUT). Next, controllability and observability analysis using a modified SCOAP [11] is performed. Afterwards, a fault is selected from the fault list and S-PODEM is invoked for test generation. If a single test vector is successfully synthesized, then logic simulation (also included in S-PODEM) driven by cost functions is immediately invoked to expand the test vector into a test sequence. For example, if S-PODEM generates a test vector 10*PP*0 for fault *f*, the logic simulator will expand 10*PP*0 into test sequence 10000, 10110, 10100 which detects *f*. Finally, the fault simulator is invoked and all other faults detected by the generated test sequence are removed from the fault list. The S-PODEM algorithm is called repeatedly to synthesize single test vectors for faults remaining on the fault list. All undetected faults are marked as hard-to-detect. At the end of PGEN, all hard-to-detect faults are assigned single test vector *PP*...*PP* to pursue possible test sequence expansions with maximal degree of freedom.

4.2 Determination of the Fault Site

According to the ISCAS format, each node *N* can have only one output and the output line of *N* is indicated by *N* as well. In context to fanout lines, a fault can occur either on a stem line or on a (fanout) branch line. As shown in Fig. 4.1a fault *F* occurs on stem line *L*, thus *F* can be represented by (*N*, *v*) where *N* is the input node of the faulty line *L* and *v* is the fault value. However, representation of a branch line fault needs the use of both input and output nodes. As shown in Fig. 4.1b, fault *F* is represented as ([N_1, N_2], *v*). It requires two nodes to uniquely specify a fanout branch. Accordingly, in_fault_node and out_fault_node can be defined to respectively indicate the input and output nodes of the faulty line for fault *F*. For example, in Fig. 4.1a the in_fault_node

TABLE I Input-Output Relationship for Two-Input NAND Gate

NAND	0	1	D	$\bar{D}$	P	P0	P1	0P	1P	PP	X
0	1	1	1	1	1	1	1	1	1	1	1
1	1	0	$\bar{D}$	D	P	P1	P0	1P	0P	PP	X
D	1	$\bar{D}$	$\bar{D}$	1	P1	P1	P1	1	$\bar{D}$	P1	X
$\bar{D}$	1	D	1	D	1P	1	D	1P	1P	1P	X
P	1	P	P1	1P	P	P1	PP	1P	PP	PP	X
P0	1	P1	P1	1	P1	P1	P1	1	P1	P1	X
P1	1	P0	P1	D	PP	P1	P0	1P	PP	PP	X
0P	1	1P	1	1P	1P	1	1P	1P	1P	1P	X
1P	1	0P	$\bar{D}$	1P	PP	P1	PP	1P	0P	PP	X
PP	1	PP	P1	1P	PP	P1	PP	1P	PP	PP	X
X	1	X	X	X	X	X	X	X	X	X	X

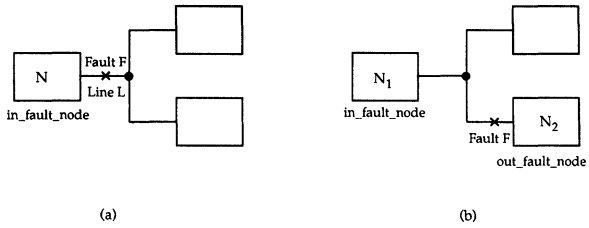

FIGURE 4.1 In_fault_node and out_fault_node with respect to fault site.

of fault F is N; while in Fig. 4.1b, the in_fault_node is N_1 and out_fault_node is N_2.

4.3 Pseudo Input Node

In synchronous sequential circuits there is at least one flip-flop in every single feedback loop, and all flip-flops are identified as *pseudo input nodes*. The rationale behind the name "pseudo input" is that implication always starts from primary input nodes; however, it is also necessary to consider logic values on the flip-flops to initiate the implication process for loops. Although a flip-flop is not a real primary input, it works as a primary input to start implication. Thus, it is called a pseudo input node. In summary, the reason for the pseudo input node is to break the feedback loop by assigning an initial value to the flip-flop, such that the implication process can be initiated.

Consider an example shown in Fig. 4.2; the implication process starts with the primary input node, A, assigned a logic value by the backtracing process and the flip-flop identified as a pseudo input node. At the beginning of the implication process, all pseudo input nodes are assigned value 0. This strategy is consistent with the assumption that all flip-flops are resettable, and hence initially all of them will provide value 0.

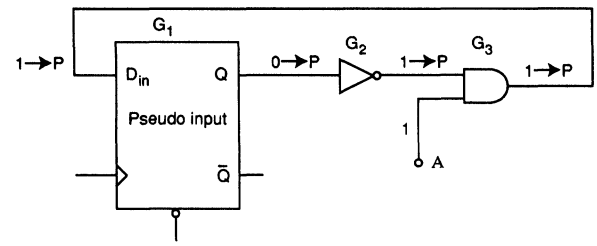

FIGURE 4.2 Resolving input/output values for a pseudo input node.

Extension to the general case can be achieved by removing the reset constraint to the CUT, and will be discussed later.

During implication, it may be observed that input and output values of a pseudo input node do not match. In general, this mismatch can be resolved by representing the behavior using a pulsating signal which may or may not contain the fault effect. For instance, consider the circuit in Fig. 4.2. Assume that primary input A is assigned value 1, and node G_1 has been recognized as a pseudo input node. When implication starts, line G_1 will have value 0 and G_2 will have value 1. Since both inputs of G_3 have value 1, the output of G_3 will be 1. Now, node G_1 has different values on its input and output lines. It is possible to continue the implication process by passing value 1 from the pseudo input node, but in that case implication will continue forever and oscillate implication values between 0 and 1 in the feedback loop.

So, in the S-PODEM algorithm whenever there is a mismatch between values at the input and output of a pseudo input node, the output will be assigned a value which contains some information regarding pulsating behavior. In this case, the output of G_1 is assigned P. Implication continues, and nodes G_2 and G_3 will also be assigned value P. At this point, pseudo input node G_1 will have the same value at the input and output, and the implication stabilizes. As shown in Fig. 4.2, the test behavior of multiple time frames has been compressed and faithfully represented by the pulsating model. Any faulty line assigned a pulsating value P, after the implication process, will be detected (Fig. 4.2). More details will be given in Section 5 which describes the implication process.

4.4 Implication Types

It is important to distinguish implication types for different nodes. Consider the circuit in Fig. 4.3, and assume that primary input A is not assigned any logic value. At the time of implication, initially G_1 will be assigned value 0, and a logic 1 will be assigned to node G_2. Further implication is not possi-

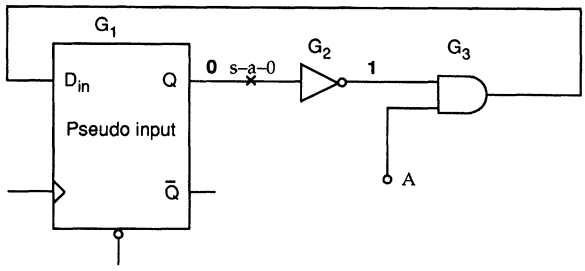

FIGURE 4.3 Differentiating implication types.

ble. Thus, the implication algorithm stabilizes, and concludes that the fault can not be sensitized. According to the standard PODEM algorithm, the implication fails and backtracking will be performed. However, this is not a desirable action. By assigning value 1 to input A, pseudo input node G_1 will get value P and the fault will be sensitized. Consequently, to resolve this implication failure, backtracing and more primary input assignments are required instead of backtracking.

Careful observation reveals that the erroneous backtracking choice arose due to the temporary and local effect of the implication value generated by the pseudo input node. The decision in favor of backtracking was mainly based on the pseudo input node behavior at the first time-frame, rather than the complete (and compressed) time-frame behavior. The difference between the *first-time-frame-info* and *all-time-frame-info* arises only with pseudo input nodes. For primary input nodes, implication values (0, 1 or P) are stabilized in the first time frame and the values remain the same for the entire test experiment. So, the first-time-frame-info is equivalent to the all-time-frame-info for the implication values of primary inputs. Based on this discussion, two different implication types can be introduced, *INPUT-IMPLIED* (*IIMP*), and *PSEUDO-IMPLIED* (*PIMP*). In short, an IIMP line (or node) is one which is stabilized over all time frames; however, a PIMP line is not stabilized. Note that flip-flops are the only source of the PIMP implication type.

Implication of a node can be represented by a pair (imp-value, imp-type) in which imp-value indicates the implication value, while imp-type denotes the implication type. The implication value can be easily generated using Table I (Section 3). The implication type of a given node N is determined according to the following rules.

1. All assigned primary inputs are IIMP.
2. When *any controlling input* of N is IIMP, the output of N is IIMP.
3. When *all controlling inputs* of N are PIMP, the output of N is PIMP.
4. When all inputs of N are noncontrolling inputs, the output has a non-X value and at least one input is PIMP, then the output of N is PIMP.
5. When all inputs of N are noncontrolling IIMP and output has a non-X value, then the output of N is IIMP.
6. For any pseudo input node N, if its input and output have the same value, then the output becomes IIMP.
7. For any pseudo input node N, if the output of N is the in_fault_node, plus, the new and old implication values of N are the same, then the output of N is IIMP.
8. When the output of N is X, then N is NIMP (*NON-IMPLIED*).

Rules 1 to 5 are easy to understand, once the proper relation of IIMP and PIMP with *first-time-frame-info* and *all-time-frame-info* is clear. For a given fault, a primary input will exhibit the same kind of behavior (with logic value 0, 1, or P) for the entire test experiment; hence it is IIMP. Besides, when a controlling input is IIMP, it is going to control the behavior of the gate output, and hence the output is also IIMP. Similarly, when the behavior of a node is controlled only by PIMP node(s), the output is also PIMP.

According to rules 2 and 5, a node can be IIMP only when at least one of the controlling inputs is IIMP or all (noncontrolling) inputs are IIMP. Since pseudo input nodes initially have PIMP type, at least one of the inputs of the nodes in the feedback loop will be PIMP; and according to rules 3 and 4, their output will always be PIMP except for a controlling

IIMP input. So nodes in a feedback loop may never be INPUT IMPLIED (Fig. 4.4) unless rule 6 is considered. As shown in Fig. 4.5, implication types of all nodes are changed to IIMP by applying rule 6; and this reflects that the circuit behavior is stable under the current input assignment.

If N is the in_fault_node, then the implication of pseudo input node N, only using rule 6, may not be stable. Assume that D_{in} of N keeps receiving implication value $P0$, but Q of N is stuck-at-1. Then the resolved implication value (see Section 5.4.2) of N is determined as $P1$, and the implication values of D_{in} and Q will never be the same. Consequently, the implication type of N will never be changed to IIMP. With the addition of rule 7, the implication type of node N is resolved to IIMP.

The last rule deals with *NON-IMPLIED* or NIMP type. The NIMP type is introduced for sake of completeness only. Any line which has value X is not implied, and the implication type is NIMP.

5. THE S-PODEM ALGORITHM

S-PODEM is a test generation algorithm characterized by a direct search process, in which decisions consist only of primary input assignments. As discussed before, S-PODEM generates single test vectors based on the concept of time compression which compacts the sequential circuit behavior using a pulsating model. Thus, instead of unfolding the sequential circuit into combinational logic array, S-PODEM

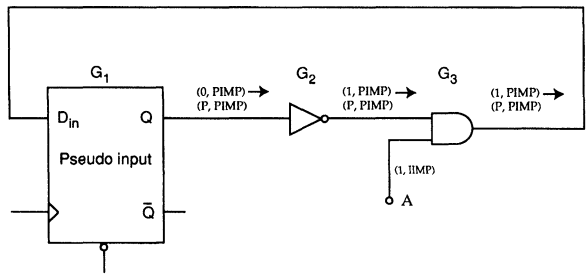

FIGURE 4.4 Implication without considering rule 6 of implied type.

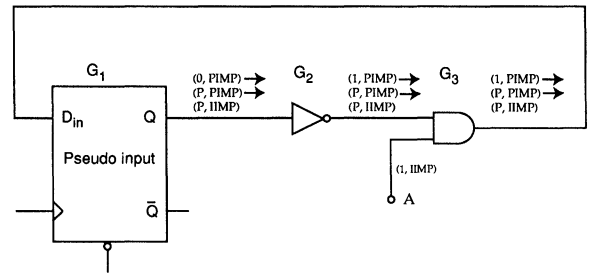

FIGURE 4.5 Implication after considering rule 6 of implied type.

generates a single test vector in a single (yet compressed) time frame. Algorithm 5.1 briefly discusses the major routines (or algorithms) used by S-PODEM, and detailed illustrations of these routines are given in the following sections.

```
Algorithm S-PODEM(fault)
begin
  Initialize decision_tree and other structures;
  repeat forever
    repeat forever
      if find_objective() fails then
        break inner repeat loop;
      if backtracing() fails then
        break inner repeat loop;
      if check_implication_and_justify() fails then
        break inner repeat loop;
      if check_test_and_simulate() succeeds then
        return SUCCESS
    end repeat
    repeat forever
      if backtracking() fails then return FAILURE;
        Increment backtracking_count;
      if check_implication_and_justify() succeeds
        then
        break inner repeat loop
    end repeat
    if check_test_and_simulate() succeeds then
        return SUCCESS
  end repeat
end algorithm
```

ALGORITHM 5.1 S-PODEM Algorithm.

5.1 Algorithm Find_Objective

The find_objective algorithm determines the next objective used by the backtracing routine. First, the routine checks whether or not fault f has been sensitized. The objective is set to sensitize f if: (1) faulty line L has value X; or (2) the logic value of L equals the fault value and L has a PIMP implication type. Then, the objective node and value are passed to the backtracing process.

If f has already been sensitized (regardless of the implication type of L), then the objective is to propagate the fault effect to a primary output. As in the case of the standard PODEM algorithm, a D-frontier is prepared for fault propagation. In PGEN, the D-frontier is implemented as a priority queue sorted on the ascending order of observability values. A lower observability value for node N indicates that N is easier to observe at a primary output; so the observability value serves as a good criterion of priority for the D-frontier.

As a member of the D-frontier, the basic requirements for node N are: (1) N has the fault effect on one of its inputs and X on its output; and (2) there exists an X-path from node N to a primary output. If no nodes can be included into the D-frontier using the above two criteria, it is still possible to find a potential D-frontier by loosening the requirements. Thus, node N can be a member of the potential D-frontier if: (1) N has the fault effect on one of its inputs and the implication type of N is PIMP; and (2) there exists an X-PIMP-path from N to a primary output. If no nodes can be found for the potential D-frontier, then the process of find-objective fails. Note that an X-PIMP-path is a path where all nodes have either value X on their outputs or have been assigned implication type PIMP.

The D-frontier is passed to the backtracing process to perform fault propagation. If backtracing from N fails, then N is discarded and another node is dequeued and backtraced. If the D-frontier becomes empty during backtracing, then the backtracing process returns failure and backtracking is performed. If backtracing from N succeeds, then implication based on the newly assigned primary input value is initiated. For fault propagation, the objective value of N is determined in such a way that all unassigned inputs of N will obtain a noncontrolling value.

5.2 Algorithm Backtracing

Given an initial objective, the backtracing process is employed to find a primary input assignment such that the objective can be accomplished. The backtracing algorithm used by PGEN considers D flip-flops as combinational nodes using a time compression technique, and must reach a primary input to terminate successfully. When backtracing reaches a primary input node N, it enters N into the decision tree along with the objective value of N. The objective value at primary input node N is known as the *preset value* of N, because the implication process starts with preset values at the primary input nodes. If the backtraced path is a feedback loop, then some nodes in the feedback loop may be traversed more than once.

Each time a D flip-flop is traversed during a backtracing, the clock line value is checked first. If the value is X then the next objective is to obtain value P for the clock input. The backtracing process is guided by controllability values derived using a modified SCOAP method. If the backtraced path is a feedback loop, then backtracing directed only by controllability values may never leave the loop. In order to solve this problem, the concept of *backtrace count* is utilized. The backtrace count of a node N is an enumeration of the number of times that N has been backtraced during a particular backtracing process. The controllabilities of each node are then weighted by the node's backtrace count, when a backtracing decision is made. If node N is overbacktraced then its controllabilities are weighted by a large number, and N will not be further backtraced. Therefore, infinite backtracing on a feedback loop can be avoided.

When a D flip-flop node FF is backtraced more than once for each backtracing process, the backtracing can be further continued or restarted from the initial objective node. If the initial objective node N,

which triggers the backtracing, has the objective value v recognized as the control value, then another backtracing path from N may be chosen to provide the control value. However, if v is not the control value of N, then the backtracing continues from FF. The reason behind this strategy is not difficult to understand. It is possible that all rebacktracing from N ultimately stop on D flip-flops, and could result in infinite switching on rebacktracing from N. To avoid this situation, backtracing from FF continues regardless of the initial objective (N and v), if the backtrace count of FF exceeds a threshold value TH(bc) (TH(bc) is assumed 8 in this work).

Another major difference in the S-PODEM backtracing process from the standard one is the selection of the input node. In PGEN, the easiest input of a combinational node is always selected (guided by the controllability/observability values) regardless of the objective value. This approach helps avoid pseudo input nodes, and generally guides the backtracing process towards primary inputs. If a pseudo input node is eventually necessary, then backtracing will lead to a pseudo input node and finally to a primary input.

Given pairs (obnode, ob_newvalue), the obnode input which is the easiest to control to the desired value (ob_newvalue), than all other unassigned input nodes, is selected. With testability analysis, controlling an input is quantified by the corresponding controllability value, and a node can have different 0 and 1 controllabilities. First chosen is an input node N with value X, when N has the lowest controllability value of being driven to the *new_obvalue*. If there is no input with the X value, then a PIMP input is selected. If backtracing fails, the initial objective (obnode, obvalue) is resumed and another backtracing is tried (at most 20 times in this experiment).

Finally, another backtracing threshold TH(bt) is added in the backtracing algorithm to assure termination of each backtracing process, whenever the total number of node traversals exceeds TH(bt). It fails if the backtracing process exceeds the backtracing threshold TH(bt), or is unable to find any input with X value or a PIMP type.

5.3 Algorithm Backtracking

The backtracking process is employed to explore the solution space and recover from incorrect decisions. If implication fails, or an objective can not be found, or backtracing is unsuccessful, then backtracking is invoked. This algorithm backtracks only on primary inputs, and pseudo input nodes are not involved. All implication values and types left by the last implication must be removed, before another new value is assigned to the backtracked primary input.

The backtracking process used in S-PODEM has three values since it allows assignment of 0, 1 and P to any input. The logic value of primary input N is inverted at the first backtracking, and N is assigned value P at the second backtracking. One more backtracking will assign value X to N and remove N from the decision tree. Again, we emphasize that an implication on event (N = X, NIMP) should be performed to remove the implication history left by the previous assignment on N. In addition, backtracking is a very time-consuming process, so a threshold is added to prevent backtracking from exploring the entire solution space. In PGEN, the backtracking threshold is set to $2^k w$, where w is the number of primary inputs and k is 2 or 3.

5.4 Algorithm Implication

The implication process determines the value at different nodes based on the primary input and pseudo input node values. It should be emphasized that there is a basic difference between implication and simulation, though the basic purpose of both is quite similar. In the case of simulation, real time behavior of the circuit is imitated and each line will have a value from the set 0, 1, X. In implication the complete fault model, which includes some variables to represent fault effects, is used. Thus, implication includes both faulty and fault-free behavior using model values such as D, $1P$, $P0$, etc. Conventionally, implication is used only for combinational circuits. For sequential circuits, all traditional approaches use simulation be-

cause they did not use the concept of compressed timing behavior. This is one of the major features of PGEN.

Implication also uses an event list. The event list is implemented as a priority queue containing all the nodes which are already implied, but have not had their output nodes implied. Again observability values are used for priority determination; however, unlike the D-frontier, the priority queue is maintained according to the descending order of observability values. The nodes which are not easily observed from the primary outputs are thus given a higher priority for implication. Furthermore, implication of other nodes depending upon the values of these hard-to-observe nodes is delayed. This guarantees that implication values of the nodes closer to the primary inputs are settled, before these values are used for further implications.

As mentioned earlier, implication starts each time when a new preset value is assigned to a primary input by backtracing or backtracking. The primary input node is placed in the implied_list by the setup_implied_list routine. In the first implication of a given fault, all D flip-flops are also inserted into the implied_list for implication. The implication algorithm is designed as an event-driven one. For a fault f, implication values of all nodes in the CUT are only cleared in the initialization process of the first implication (for f). We do not clear implication values for the other implications for f (unless backtracking occurs) because:

1. there is no reason to destroy the implication history;
2. the implication speed is much faster with implication values retained;
3. the implication results of sequential circuits are quite implication-order dependent. If the implication values are erased, the implication order problem can further deteriorate.

The implication algorithm keeps iterating until the implied_list is empty. Counter imp_cycle is employed to keep track of the number of times that different nodes have been implied. When the value of imp_cycle exceeds the implication threshold value, the implication algorithm returns FAILURE. Since PGEN is not proved as a complete algorithm, this mechanism helps avoid infinite loops. At present, the implication threshold value is set to 4 times the number of nodes in the CUT.

Successively a node, called implied_node, is dequeued from the implied_list; and each node N immediately connected to the output of the implied_node is checked for implication by the imply_node routine (see Section 5.4.2). The imply_node routine also enqueues N into the implied_list, if N can be further implied and new events take place. A special flag, MORE_ASSIGN_REQ, signals an X-oscillation (see Section 5.4.2) and indicates a requirement for further backtracing. The objective node for backtracing is returned by parameter sp_node and the implication routine returns the same flag to the calling routine.

When all nodes are stabilized and no more events are left, a check is made to determine whether or not the faulty line receives a sensitizing value. If it does not have a sensitizing value and the in_fault_node is IIMP, failure is returned from the implication routine. Inhibition of fault effect propagation is also a cause of backtracking, but that is signaled by the find_objective routine.

5.4.1 Algorithm setup_implied_list

The setup_implied_list algorithm prepares an event list from the nodes in the decision tree. As discussed before, for the first implication of a fault f, the assigned primary input and all D flip-flops are initial entries of the implied_list. All D flip-flops are initially set to value 0, and the primary input node is set to its preset value. If f is on a primary input line, the fault effect will be taken into account immediately. On subsequent implications of f, only the newly assigned primary input is placed into the implied_list. Note that the implied_list provides event sources.

5.4.2 Algorithm imply_node

The imply_node algorithm is responsible for implying a node called imnode, and enqueuing the imnode into the implied_list if it is successfully implied.

First, pseudo input nodes are considered. The variable *result* in the imply_node routine represents the new implication value at the D flip-flop output in the next time frame, and depends upon the D_{in} value. To consider the pulsating behavior, the new output implication value of flip-flop *FF* is resolved with its previous output implication value and a resolved value is created. Some simple examples of the resolved output include: conversion from 0 to 1 changes into *P*, 0 to *D* changes into *P*0 and 0 to $\bar{D}$ into 0*P*; conversion from 1 to 0 changes into *P*, 1 to *D* changed into 1*P* and 1 to $\bar{D}$ into *P*1.

The reason for the conversions is very simple. For example, if there is a 0 to *D* implication change on the output of *FF*, i.e., in the fault-free circuit the signal has a 0 to 1 conversion, a pulsating effect (*P*) has occurred. However, if the circuit is faulty, the signal remains at 0. In summary, the new and old implication values on the output of *FF* are resolved and stored in the variable *new_imvalue*. For some signal conversions, the new implication value just overwrites the old implication value. For example, if the old implication value on the output of *FF* is *P*1 and the new value is 1*P*, then the new implication value is resolved as 1*P*. The implication value conversions are summaried in Table II. The first column (row) of Table II gives the old (new) implication values of *FF*, and resolved values can directly be obtained from the table entries. Note that if the output of *FF* is the fault site, then special implication value conversion must be considered. Assume that the output of *FF* is stuck-at-0, and the new_imvalue of *FF* is *P*1; it is necessary to resolve the new_imvalue as *P*0. The implication type of *FF* is changed to IIMP if: (1) The resolved implication value of *Q* is the same as the implication value of D_{in}; or (2) The output of *FF* is faulty, and the old_imvalue and new_imvalue of *FF* are the same.

The problem of X oscillation is very important. Consider the circuit in Fig. 5.1(a); flip-flop G_1 is recognized as a pseudo input node. Implication starts with value 0 on node G_1 and continues with the output value of G_2 implied to logic 1. At that moment,input and output of pseudo input node G_1 will be different and resolved to value *P*. As it can be observed from Fig. 5.1(b), it is no longer possible to propagate the value *P* through node G_2, and the output of G_2 will be set to X. Now, the input of node G_1 is X and, according to the implication rules, X can not be propagated through a pseudo input node. Hence, the output will again be set to 0. This will repeat the situation of Fig. 5.1(a), and the output of node G_1 will keep oscillating between values 0 and *P*.

Hence, each pseudo input node keeps an individual counter which counts the number of changes from a non-X value to X value. If there is an X-value oscillation occurring at pseudo input node *N*, then the X-oscillation counter of *N* will exceed the X-value change threshold. This indicates that there is one input (in the feedback loop) not assigned a value which provides the source of the X-value changes. So, this input should be found and backtracing continued until it is assigned an appropriate value. To indicate this, the imply_node routine returns a special value MORE_ASSIGN_REQ.

Implication on combinational nodes is much easier. If implication of the current node *N* is successful,

TABLE II Conversions of Implication Values

	0	1	D	$\bar{D}$	P	P0	P1	0P	1P	PP	X
0	0	P	P0	0P	P	P0	PP	0P	PP	PP	X
1	P	1	1P	P1	P	PP	P1	PP	1P	PP	X
D	P0	1P	D	PP	P	P0	PP	PP	1P	PP	X
$\bar{D}$	0P	P1	PP	$\bar{D}$	P	PP	P1	0P	PP	PP	X
P	0	1	D	$\bar{D}$	P	P0	P1	0P	1P	PP	X
P0	0	1P	D	0P	P	P0	PP	0P	1P	PP	X
P1	0P	1	1P	$\bar{D}$	P	PP	P1	0P	1P	PP	X
0P	0	P1	P0	$\bar{D}$	P	P0	P1	0P	PP	PP	X
1P	P0	1	D	P1	P	P0	P1	PP	1P	PP	X
PP	0	1	D	$\bar{D}$	P	P0	P1	0P	1P	PP	X
X	0	1	D	$\bar{D}$	P	P0	P1	0P	1P	PP	X

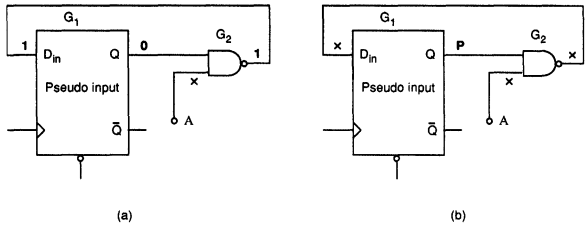

FIGURE 5.1 X oscillations with a pseudo input node.

then the implication value with type of N is updated. In addition, N is enqueued into the implied_list to trigger more implication events.

5.5 Other Routines

The implication routine is actually invoked by the check_implication_and_justify routine. This routine simply invokes implication, and if implication returns the flag MORE_ASSIGN_REQ then it invokes the backtracing routine. Check_test_and_simulate is another simple algorithm. First, it determines whether or not the fault effect has reached a primary output by checking implication values on primary outputs. A fault is potentially detected if there exists at least one implication value of primary outputs containing $P1$, $P0$, $0P$, $1P$, PP, D, or $\bar{D}$ regardless of the implication types. If so, S-PODEM performs logic simulation to expand the single test vector into a test sequence. A successful result from the logic simulation indicates that the test sequence can detect the fault.

5.6 A Complete Example

Figure 5.2 gives the circuit diagram of ISCAS89 benchmark circuit S27 (resettable), and a stuck-at-0 fault is assumed to occur on line L8. S-PODEM starts with all nodes assigned implication value X and implication type NIMP. Then, the find-objective routine is called to determine the initial objective (L8, 1). The backtracing process backtraces nodes L14, L0, and 0 is assigned to primary input L0. Implications start with events (L0 = 0, IIMP), (L7 = 0, PIMP), (L6 = 0, PIMP), and (L5 = 0, PIMP) inserted into the implied-list. Then, (L0 = 0, IIMP) and (L6 = 0,

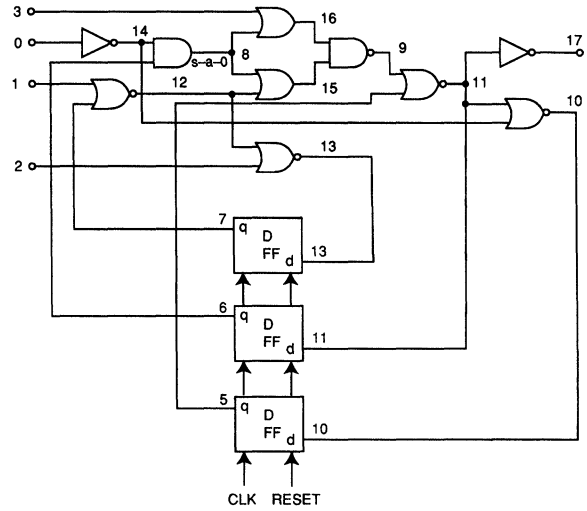

FIGURE 5.2 Benchmark circuit S27.

PIMP) imply (L14 = 1, IIMP), (L8 = 0, PIMP), (L10 = 0, IIMP), and (L5 = 0, IIMP); events L5 = 0 and L7 = 0 do not create any new events. Although implication value L8 = 0 masks fault L8 stuck-at-0, the PIMP implication type indicates that L8 = 0 is not a permanent logic value under the current input assignment; hence, implication does not fail.

Since the fault has not been sensitized, the initial objective is still (L8, 1). The backtracing process traverses nodes L8, L6, L11, L9, L16, and finally primary input L3 is assigned value 1. The implication process arises by entering (L3 = 1, IIMP) into the implied-list, and triggers implication (L16 = 1, IIMP). Again, the fault is not sensitized and the same initial objective is used. Nodes L8, L6, L11, L9, L15, L12 are then backtraced, and L1 is assigned value 0. By entering (L1 = 0, IIMP) into the implied-list, the following implications are performed: (L12 = 1, PIMP), (L15 = 1, PIMP), (L13 = 0, PIMP), (L7 = 0, IIMP), (L12 = 1, IIMP), (L15 = 1, IIMP), (L13 = 0, IIMP), (L9 = 0, IIMP), (L11 = 1, IIMP), (L6 = P, PIMP), (L17 = 0, IIMP), (L8 = $P0$, PIMP). The fault is not sensitized to the primary output. Since (L8 = $P0$, PIMP) hints that the stuck-at-0 fault on L8 is only activated temporarily (PIMP type), the initial objective is still (L8, 1). Backtracing starts from L8, then L6 and fails on line L11 which has IIMP implication type.

Backtracking occurs on primary input L1 which is thus assigned value 1. Before (L1 = 1, IIMP) implies, all implication history left by (L1 = 0, IIMP) must be removed. Thus, nodes L1, L12, L15, L13, L9, L11, L17 are assigned (X, NIMP), nodes L7, L6, and L8 are assigned (0, PIMP). Note that the implication history removal restores the implication history back to the state before L1 = 0 was implied. Implication is then activated by event (L1 = 1, IIMP), and results in the following implication values and types: (L12 = 0, IIMP), (L15 = 0, PIMP), (L9 = 1, PIMP), (L11 = 0, PIMP), (L6 = 0, IIMP), (L17 = 1, PIMP), (L8 = 0, IIMP), (L15 = 0, IIMP), (L9 = 1, IIMP), (L11 = 0, IIMP), and (L17 = 1, IIMP). Now, L8 is IIMP but the fault is not sensitized; thus, implication fails.

Failure of implication triggers one more backtracking on primary input L1. Implication history left by event (L1 = 1, IIMP) is erased. This causes the implication history of nodes L1, L12, L15, L9, L11, L17 to be replaced by (X, NIMP); nodes L6 and L8 by (0, PIMP). Primary input L1 is assigned value P after backtracking, and implication proceeds by entering (L1 = *P*, IIMP) into the implied-list. Ultimately, we have (L12 = *P*, PIMP), (L15 = *P*, PIMP), (L9 = *P*, PIMP), (L11 = *P*, PIMP), (L6 = *P*, IIMP), (L17 = *P*, PIMP), (L8 = *P*0, IIMP), (L15 = *PP*, PIMP), (L9 = *PP* , PIMP), (L11 = *PP* , PIMP), (L6 = *PP* , IIMP) and (L17 = *PP*, PIMP). The fault effect *PP* is propagated to primary output L17; thus the fault is potentially detectable by the single test vector 0*PX*1. The logic simulator (described in the next section) successfully expands the vector into test sequence (0001), (0101) which detects line L8 stuck-at-0.

6. SIMULATION AND RESULTS

Logic simulation is an integral part of the S-PODEM algorithm. Given a fault *f*, the test vector generation phase of S-PODEM produces a single test vector which should have a high probability of being expanded as a test sequence for *f*. However, the compression of multiple time frames for test generation serves as both an advantage as well as a disadvantage. In general, only logic simulation can provide accurate information about circuit behavior. One can not rely entirely on the information given by the single test vector, since the P-model contains vague signal representations as discussed in previous sections.

6.1 Logic Simulation and Sequence Expansion

In the end of S-PODEM, there might be primary inputs assigned X, i.e., S-PODEM does not assign values to them. To achieve the maximal freedom on sequence expansion, all primary inputs assigned X are assigned P. The logic simulator used in PGEN simulates both fault-free and faulty circuits simultaneously, and stops when different values are observed at the output of faulty and fault-free circuits, or when the circuit is simulated for the predefined number of clock cycles. To expand the *P* signals at primary inputs, the logic simulator uses *dynamic cost analysis* [1] to guide the sequence expansion.

Assume that S-PODEM generates a single test vector 10*PP* for fault *f*, dynamic cost analysis will determine the first test pattern as 1000; and only primary inputs assigned P can change logic value for sequence expansion. If *f* is not detected by 1000, further expansion is required. Based on the first *P* (of 10*PP*), two test patterns 1000 and 1010 are tried; and costs C_1 and C_2 are estimated respectively based on fault sensitization and propagation. If *f* is not detected by either test pattern and C_2 is smaller than C_1 (for example), then 1010 is a better choice than 1000. From 1010, two more test patterns 1010 and 1011 (based on the second P of 10*PP*) can be further tried. If 1011 has a lower cost than 1010, then the second test pattern is determined as 1011. Thus, multiple bit changes are possible. Note that the cost analysis of [1] allows only single bit change.

The sequence expansion process repeats until *f* is detected or a simulation threshold is exceeded. Once the single vector is successfully expanded as a test sequence by the logic simulation, a counter S-vec (number of successful vectors) is incremented by one.

6.2 Fault Simulation

When a test sequence T has been successfully expanded from the single test vector for fault f, using the dynamic cost analysis, fault simulation is performed to find all other faults covered by T. Differential fault simulation (Dsim) [8] has been recognized as very powerful in aspects of speed and memory requirement, and was implemented in PGEN with minor modification. If a sequence T is expanded and fails to detect the target fault f, it might be wasteful if T is abandoned immediately. Thus, fault simulation (called intermediate fault simulation) is performed to find other faults which are covered by T. If T covers other faults in the fault list, then T is incorporated into the test sequence; otherwise, T is abandoned. Note that if intermediate fault simulation (Isim) is successful, then the counter S-vec is incremented by one as well.

6.3 Nonresettable Synchronous Sequential Circuits

So far, discussions of PGEN are mainly restricted to the domain of resettable synchronous sequential circuits. The results can be further extended to a more general case: nonresettable synchronous sequential circuit testing. In the process of single test vector synthesis, S-PODEM assumes the unknown state (X) for all pseudo input nodes, when the implication process starts. In addition, the fault list is sorted such that the faults closer to outputs will be processed earlier. This strategy avoids the use of an initialization sequence. Assume that fault f is closer to primary outputs than other faults; in most cases, a sequence T can be generated to detect f from the unknown state. However, if f does not occur then T will drive the CUT to a known (or partially known) state from which test generation for other faults can be done smoothly. It has been found that all ISCAS89 benchmark circuits can be tested using the aforementioned philosophy except S510. It is almost impossible to drive S510 to a specific state since a synchronization sequence might not exist.

The test generation phase (S-PODEM) of PGEN is strongly incorporated with the fault simulation phase (Dsim), and their relationship is bidirectional. In PGEN, S-PODEM generates test patterns for Dsim; and Dsim provides S-PODEM with good and faulty circuit states from which the test generation phase can be resumed. Thus, starting from an unknown circuit state, S-PODEM generates a single test vector V by selecting a fault f_1 which is closest to primary outputs. Also, starting from an unknown state, test vector V is expanded into test sequence T using one of the three expanding methods. Test sequence T is then passed to Dsim to find all other faults covered by T. These faults are removed from the fault list. In addition, Dsim manipulates the faulty state information for all faults which can not be detected by T. S-PODEM then selects another fault f_2, and resumes single test vector generation using the faulty state of f_2 provided by Dsim. The process is repeated until all faults are detected, or no new faults can be detected. Note that the faulty state of f_2 discussed above is the circuit state derived by applying test sequence T starting from unknown state under fault f_2.

6.4 Simulation Results

The test generation algorithm described previously was implemented in the program PGEN, which consists of about 5000 lines of C code and runs in a SUN 3/260 environment. Results for several ISCAS89 sequential benchmark circuits are shown in Table III. For each circuit, circuit name (Circuit), test length (Length), number of faults (Faults), number of faults detected (Detected), number of faults detected by Dsim (Dsim), number of faults detected by Isim (Isim), number of total single vectors synthesized (T-Vec), number of successful single vectors (S-Vec), fault coverage (Cov), ATPG time (Time) and logic simulation threshold (TH(sim)) are provided. The fault coverage presented in this table is *sure* fault coverage, and possible fault detections are excluded in the calculation of Cov. The ATPG time is represented in hours of SUN 3/260 except S27 (in seconds). The simulation was conducted using the fol-

TABLE III Experimental Results on ISCAS89 Benchmark Sequential Circuits

Circuit	Length	Faults	Detected	Dsim	Isim	T-Vec	S-Vec	Cov	Time	TH(sim)
s27	16	32	32	25	0	7	7	100.00	1.8s	20
s208	190	215	137	28	98	342	23	63.92	0.40	20
s298	138	308	262	160	98	60	11	85.06	0.61	20
s344	128	324	307	240	59	93	20	94.75	1.05	20
s349	131	330	315	92	213	159	21	95.45	1.51	20
s382	311	399	348 + 5	256	93	49	12	88.47	2.45	50
s386	133	384	289 + 14	128	161	663	39	78.91	14.2	50
s400	276	424	368 + 2	235	131	52	9	87.26	2.79	50
s444	387	473	414	310	100	60	13	87.53	3.31	50
s526	904	555	439 + 2	285	152	130	26	79.46	8.18	50
s953	27	1078	89	67	0	22	22	8.26	0.06	20
s1196	365	1242	1218 + 10	883	182	699	197	98.87	9.19	20
s1488	476	1486	1280 + 63	595	732	59	45	90.38	13.81	20
s1494	528	1506	1373 + 26	628	754	140	62	92.90	13.21	20

lowing thresholds: backtracking—($PI2^2$), backtracing (TH(bt))—($N2^2$), X-oscillation—(20), implication—($N2^2$); where PI denotes the number of primary inputs of the CUT, N is the number of nodes.

In all benchmark circuit simulations, no reset line is assumed and all CUTs are tested starting from the unknown state. As expected, S510 can not be tested since no flip-flop can be initialized. The pulsating model used by S-PODEM is weak in pulsating representation, and there might be faults for which S-PODEM can not synthesize a test vector; but test patterns really exist. PGEN avoids this situation by expanding test patterns for all hard-to-detect faults using vector *PPP...PP* before the end of the test generation process. For example, PGEN detects 370 (368 + 2) faults for S400 as shown in Table III. Among these 370 faults detected, 2 faults are detected by using vector *PPP...PP*.

Threshold values have strong impact on the performance of PGEN. For example, PGEN can detect only 257 faults when the logic simulation threshold is set to 20. However, 353 faults can be detected if TH(sim) is increased to 50. It is worth noting that the summation of Dsim, Isim and S-Vec is greater than the number of detected faults (Detected). For example, there are totally 262 faults detected in circuit S298. Among these 262 faults, 160 of them are detected by Dsim, 98 detected by Isim and 4 (= 262 − 160 − 98) detected by logic simulation which is used to expand the test sequence. However, the number of successful vectors (S-Vec) are 11 which hints that 7 out of the 11 successful vectors are contributed by Isim. Table III demonstrates that Isim is very powerful, and it detects a majority of faults in some circuits (such as S208, S349).

7. CONCLUSIONS

In this paper, a novel sequential circuit testing method called PGEN has been introduced. Results of the proposed method might not be attractive when compared with existing solutions [5–6]. However, the motivation of this research is to unify the deterministic test generation method and simulation-based method for sequential circuit testing and to seek for the possibility of resulting in a better solution. Using a new logic model (P-model) and the concept of circuit time compression, the multiple time-frame circuit behavior can be efficiently represented by a single time frame. In addition, the concepts of pseudo input node and implication type support the implementation of PGEN. Thus, PODEM has been successfully extended to the domain of sequential circuit testing. The idea behind PGEN is to determine the input patterns which are fixed (and also can be easily generated) in the test process, then use logic simulation (guided by cost analysis) to expand the pulsating signals.

According to benchmark circuit simulation, single test vectors generated by S-PODEM have relatively

low probability of being expanded as *real* test patterns. This comes from the inadequacy of the pulsating model, and can be remedied using a more powerful model. Computing time used by PGEN is generally high, and the reasons are: (1) PGEN can just predict resettability of the circuits using SCOAP and wastes lots of CPU time on synthesizing single vectors for untestable faults; (2) the backtracing and implication processes are easy to be trapped in feedback loops (until threshold values are exceeded), and this wastes CPU time; (3) the PGEN program was not optimally coded; for example, a lot of events are unnecessarily activated in logic simulation and fault simulation. Test sequences are generally very compact, and fault coverage can be higher if larger values are set to the thresholds (but CPU time will be increased).

Acknowledgements

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada under Grant A 4750.

References

[1] V. D. Agrawal, K. T. Cheng and P. Agrawal,"A Directed Search Method for Test Generation Using a Concurrent Fault Simulator," *IEEE Trans. on Computer-Aided Design*, Vol. 8, pp. 131–138, Feb. 1989.

[2] M. J. Bending, "HITEST—A Knowledge-Based Test Generation System," *IEEE Design and Test of Computers*, Vol. 1, pp. 83–93, May 1984.

[3] F. Brglez, D. Bryan and K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," *Proc. of International Symp. on Circuits and Systems*, pp. 1929–1934, May 1989.

[4] W. T. Cheng and T. J. Chakraborty, "Gentest: An Automatic Test-Generation System for Sequential Circuits," *IEEE Computer*, Vol. 22, pp. 43–49, April 1989.

[5] W. T. Cheng and S. Davison, "Sequential Circuit Test Generator (STG) Benchmark Results," *Proc. of International Symp. on Circuits and Systems*, pp. 1939–1941, May 1989.

[6] K. T. Cheng and V. D. Agrawal, "Concurrent Test Generation and Design for Testability," *Proc. of International Symp. on Circuits and Systems*, pp. 1935–1938, May 1989.

[7] K. T. Cheng and J. Y. Jou, "Functional Test Generation for Finite State Machines," *Proc. of International Test Conf.*, pp. 162–168, 1990.

[8] W. T. Cheng and M. L. Yu, "Differential Fault Simulation for Sequential Circuits," *Journal of Electronic Testing: Theory and Applications*, Vol. 1, pp. 7–13, Feb. 1990.

[9] A. Ghosh, S. Devadas and A. R. Newton, "Test Generation and Verification for Highly Sequential Circuits," *IEEE Trans. on Computer-Aided Design*, Vol. 10, pp. 652–667, May 1991.

[10] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. on Computers*, Vol. C-30, pp. 215–222, March 1981.

[11] L. Goldstein, "Controllability/Observability Analysis of Digital Circuits," *IEEE Trans. on Circuits and Systems*, Vol. CAS-26, pp. 685–693, Sept. 1979.

[12] F. C. Hennie, *Finite-State Models for Logical Machines*, Wiley, 1968.

[13] T. P. Kelsey and K. K. Saluja, "Fast Test Generation for Sequential Circuits," *Proc. of IEEE International Conf. on Computer-Aided Design*, pp. 354–357, 1989.

[14] H-K. T. Ma, S. Devadas, A. R. Newton, A. Sangiovanni-Vincentelli, "Test Generation for Sequential Circuits," *IEEE Trans. on Computer-Aided Design*, pp. 1081–1093, Oct, 1988.

[15] S. Mallela and S. Wu, "Sequential Circuit Testing Generation System," *Proc. of International Test Conf.*, pp. 57–61, Nov. 1985.

[16] R. Marlett, "An Efficient Test Generation System for Sequential Circuits," *Proc. of 23rd Design Automation Conf.*, pp. 250-256, 1986.

[17] P. Muth, "A Nine-Valued Circuit Model for Test Generation," *IEEE Trans. on Computers*, Vol. C-25, pp. 630–636, June 1976.

[18] I. Pomeranz and S. M. Reddy, "On Achieving a Complete Fault Coverage for Sequential Machines Using the Transition Fault Model," *Proc. of 28th Design Automation Conf., pp. 341–346, 1991.*

[19] G. R. Putzolu and J. P. Roth, "A Heuristic Algorithm for the Testing of Asynchronous Circuits," *IEEE Trans. on Computers*, Vol. C-20, pp. 639–647, June 1971.

[20] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, Vol. 10, pp. 278–291, July 1966.

[21] T. J. Snethen, "Simulation-Oriented Fault Test Generator," *Proc. of 14th Design Automation Conf.*, pp. 88–93, 1977.

[22] T. W. Williams and K. P. Parker, "Design for Testability—A Survey," *IEEE Proc.*, pp. 98–112, Jan. 1983.

Authors' Biographies

Anita Gleason received her B.S., M.S., and Ph.D. in Computer Science from the New Mexico Institute of Mining and Technology, Socorro, NM. She is presently an Associate Professor in the Department of Mathematics and Computer Science, Bloomsburg University, Bloomsburg, PA. Her research interests include design for testability, built-in self-testing, and fault-tolerant computing.

Dr Gleason is a Member of the IEEE.

Nigam Shah received his B.E. in Computer Engineering from the University of Gujarat, Gujarat, India in 1989, and his M.S. in Computer Science from the

New Mexico Institute of Mining and Technology, Socorro, NM in 1991. He expects to receive his M.B.A. from the Nova Southwestern University, Fort Lauderdale, FL in spring of 1995.

Mr Shah is currently a Technical Manager and Senior Systems Specialist in Information Services at the AT & T Universal Card Services, Jacksonville, FL. His research interests include network protocols, systems interoperability, software engineering, and testing.

Mr Shah is a recipient of the President Circle Award at the AT & T Universal Card Services.

Wen-Ben Jone was born in Taipei, Taiwan, Republic of China. He received the B.S. degree in Computer Science in 1979, the M.E. degree in Computer Engineering in 1981, both from National Chiao-Tung University, Taiwan; and the Ph.D. degree in Computer Engineering and Science from Case Western Reserve University, Cleveland, OH, in 1987.

In 1987, he joined the Department of Computer Science at New Mexico Institute of Mining and Technology, Socorro, where he was promoted as an Associate Professor in 1992. He is currently a Visiting Associate Professor of the Department of Computer Engineering and Information Science, National Chung-Cheng University, Chiayi, Taiwan, R.O.C. His research interests include fault-tolerant computing. VLSI design and test, and computer architecture. He has published papers and served as a reviewer in these research areas in various technical journals and conferences. He has also served on the program committee for the 5th VLSI Design/CAD Symposium (in Taiwan).

Dr. Jone received the Best Thesis Award from the Chinese Institute of Electrical Engineering (Republic of China), in 1981. He is a member of the IEEE and the IEEE Computer Society Test Technology Technical Committee.