Path Delay Fault Test Generation for Standard Scan Designs Using State Tuples *

Yun Shao[1], Irith Pomeranz[2] and Sudhakar M. Reddy[1]

1: Electrical and Computer Engineering Department
University of Iowa, Iowa City, IA 52242, U.S.A.

2: School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907, U.S.A.

Abstract

In this work we propose a novel concept called state tuple to represent the states of lines in a circuit for the generation of two pattern tests. The proposed approach is described in detail for generating robust two pattern tests for path delay faults in standard scan designs. Using the proposed approach we also report experimental results of a test generator for robust path delay faults in standard scan designs. The results show that the test generator achieves high efficiency with reduced implementation complexity.

1. Introduction

The objective of delay fault testing is to verify the timing correctness of manufactured circuits. Two main delay fault models, the *gate delay fault* model [2-4, 14] and the *path delay fault* model [1, 5, 6, 14, 18] have been used for modeling delay defects. The path delay fault model is used throughout this paper.

In order to facilitate efficient test generation for delay faults, several multiple-valued logic systems, which can distinguish different logic states of circuit lines, were proposed earlier [1, 5, 7, 9, 10, 11-13, 17]. Each logic system is designed to target a specific problem. For a particular delay fault testing problem, there usually exists a minimum set of logic states or values that enables accurate implication of the signal attributes that are relevant to the problem. Different logic systems differ in the numbers of logic values and the states of the signal lines those logic values represent. It is desirable for a delay fault testing tool to deal with different classes of tests (e.g., robust, non-robust) and different testing environments (e.g., enhanced-scan, standard scan). An existing tool may also need to be extended to solve new problems. Much work is required to implement a new multiple-valued logic algebra, especially when the number of logic values is high. As a result, direct application of multiple-valued logic lacks flexibility and extendibility.

To alleviate the problem above, we propose in this paper an extendible representation for the logic states of signal lines called *state tuple*. States of lines are flexibly represented by state tuples. The elements included in the state tuples can be adjusted for a specific problem. Instead of implementing another logic algebra for each new problem, the same form of a tuple can be expanded or contracted by adding new elements or suppressing some elements to accommodate relevant attributes of the states of circuit lines. Thus, the implementation effort and the system complexity are reduced by using state tuples.

The paper is organized as follows. In Section 2, some background on path delay fault testing and an overview of the proposed approach are given. In Section 3, the state tuples proposed for use in path delay fault test generation are discussed. In Section 4 the implementation of a test generator for path delay faults in a standard scan environment based on state tuples is described. Experimental results for ISCAS89 benchmark circuits are given in Section 5. Section 6 concludes the paper.

2. Preliminaries

2.1 Path sensitization

Path delay faults model defects that cause the cumulative propagation delay of paths to exceed a specified value. A circuit connection f_{on} is called an *on-input* of a path P if it belongs to P. A circuit connection f_{side} is called a *side-input* of path P if it is connected to a gate G_i which belongs to path P and f_{side} is not on path P. For combinational or scanned designs, a two-pattern test $<V_1, V_2>$ is used to test a path delay fault. The first pattern V_1 is applied at time t_0. After the circuit-under-test has stabilized, the second pattern V_2 is applied at time t_1 to propagate a rising or falling transition from the input of the path-under-test to the output of the path. The output of the path is sampled at t_2, where $(t_2 - t_1)$ corresponds to the desired time interval for propagation of input changes to the outputs. The stabilized logic value of a line after V_1 is applied is called the *initial value*, and the logic value that is expected after V_2 is applied is called the *final value*.

The following necessary and sufficient path sensitization conditions for a path to be robustly tested were given by Lin and Reddy [5].

(1) $<V_1, V_2>$ will launch the desired transition at the input of the path.
(2) At each gate on the path, the side-inputs have logic values that are covered by the values indicated in Table 2.1.

	AND/NAND	OR/NOR
Rising	U1	S0
Falling	S1	U0

Table 2.1 Sensitizing input values

*Research reported was supported in part by SRC Grant 98-TJ-645 and NSF Grant MIP-9725053.

In Table 2.1, U1 (U0) represents the signal on a line with an unknown initial value and final value 1 (0). S1 (S0) represents a stable signal for which both the initial and final values are 1 (0). Signal S1 or S0 is free of hazards, i.e., there are no temporary transitions to the opposite logic value between the initial and final values. The robust tests discussed here are also called *general robust tests* in [9]. More restricted robust tests are defined by adding additional constrains on the transitions propagated along the path [9].

2.2 Test Application for Scan Designs

Three test application methods were proposed to apply a two-pattern test to a scan design: *enhanced scan, functional justification* (also called *broadside*) and *scan-shifting justification* (also called *skewed-load*) [8, 10, 11]. For the enhanced scan test application method, each scan element in the scan chain is designed to store two bits by including an extra holding latch. Thus, two test patterns with arbitrary values can be shifted into the scan chain and applied to the combinational part of the circuit consecutively. Although enhanced scan enables the highest fault coverage among the three test application schemes, it has limited use because of the area and performance overhead incurred by the special scan elements. Functional justification and scan-shifting justification are used for standard scan designs, where only one bit can be stored in a scan element. For both methods, the first pattern V_1 of the test is loaded into the scan chain and held until the circuit stabilizes. In functional justification, a system clock is applied to load the present-state values of the second pattern V_2. These values are generated on the inputs of the flip-flops by the application of V_1. In scan-shifting justification, the state values of V_2 are obtained through shifting of the scan chain by one bit. Issues related to the use of scan justification and functional justification are discussed in [8, 11]. It is argued in [11] that functional justification is the preferred method for single clock standard scan circuits. The test generator discussed in this work uses functional justification for standard scan designs.

2.3 Overview of the proposed approach

Using a multiple-valued logic system leads to more accurate implications of the states of lines in the circuit, and allows more efficient decision-making during test generation. Large sets of logic values allow for higher precision, however, they also tend to be difficult to implement. In many cases, a compromise between the complexity of the chosen logic system and the implication power of the system has to be made. Furthermore, rewriting all the procedures for a new logic system is not an economic way to expand the capabilities of existing tools.

Our objective is to provide a consistent mechanism to represent the states of lines in a circuit for a wide range of delay fault testing problems, while providing a simple, extendible way to imply the states of lines. A representation called *state tuples* is proposed for this purpose. A state tuple is composed of elements. Each element depicts a certain attribute of a signal. When the state tuple is expanded to add new elements (corresponding to new signal attributes), the procedures for manipulating the already existing elements can be reused.

The focus of this paper is on generation of robust tests for standard scan circuits. In addition to the elements to represent the logic state of a circuit line, elements that depict the hazard status of a line are included in the state tuple of a line. As shown later, the implication of hazard values can be conducted in a very simply and efficient way. We developed a test generator based on state tuples with minimal implementation effort, while having the same implication power provided by the 20-valued logic system proposed earlier [9].

3. State tuples

The state of a line can be described by an n-tuple ($e_1, e_2, ..., e_n$). An element e_i in the state tuple represents one aspect of the state of a line. The initial and final values, as well as the hazard conditions on a line are examples of elements that are of interest for path delay fault test generation. We use a 4-tuple (v_1, v_2, h_1, h_0) to represent the state of a line for test generation. Logic status elements v_1 and v_2 are the initial and final logic values on the line, respectively. In this work they can have one of the three logic values {0, 1, X}. The hazard status element h_1 (h_0) is called the *1-hazard value* (*0-hazard value*). h_1 and h_0 can have one of three values {0, 1, X}. The definition of h_1 and h_0 are given next.

Definition 3.1: For a given line l, the 1-hazard value h_1 is 0 if and only if there is a stable 1 on l during the application of a two-pattern test. If it is known that l cannot assume a stable 1, the value of h_1 is 1. Otherwise h_1 is X.

Definition 3.2: For a given line l, the 0-hazard value h_0 is 0 if and only if there is a stable 0 on l during the application of a two-pattern test. If it is known that l cannot assume a stable 0, the value of h_0 is 1. Otherwise h_0 is X.

It should be noted that if either v_1 or v_2 of a line is 0(1) then h_1 (h_0) of that line is 1, because there cannot be a stable 1 (0) on the line. Out of the 9 combinations of (h_1, h_0) only six, shown in Table 3.1, occur during test generation. The remaining three combinations (0, 0), (0, X), and (X, 0) are not meaningful. For example (0, 0) is not meaningful since a line cannot have both a stable 1 and a stable 0 at the same time. When h_1 (h_0) is zero, it implies that a stable 1 (0) is present on the corresponding line and hence h0 (h1) would be 1 (1). As a result, hazard status combinations (0, X) and (X, 0) do not occur.

hazard status on the line	h_1	h_0
unknown hazard status	X	X
hazard 0	X	1
hazard 1	1	X
stable 0	1	0
stable 1	0	1
not a stable value	1	1

Table 3.1 Hazard status indicated by (h_1, h_0)

The implication rules of h_1 and h_0 for an AND gate are given as Lemma 3.1.

Lemma 3.1: For an AND gate with m input lines $k_1, ..., k_m$ and output line k, let the hazard status value h_1 (h_0) of line l be $H_1(l)$ ($H_0(l)$). The hazard values of line k are calculated using the formulas,

$$H_1(k) = \mathrm{OR}(H_1(k_1), H_1(k_2),\ldots,H_1(k_m)) \quad (3.1)$$
$$H_0(k) = \mathrm{AND}(H_0(k_1), H_0(k_2),\ldots,H_0(k_m)) \quad (3.2)$$

Proof: The output line k has a stable 1 if and only if all the input lines of the AND gate are at a stable 1. Therefore, $H_1(k) = 0 \Leftrightarrow \mathrm{OR}(H_1(k_1),\ldots,H_1(k_m)) = 0$. The output line k has a stable 0 value if and only if at least one of the input lines of the AND gate is at a stable 0. Therefore $H_0(k) = 0 \Leftrightarrow \mathrm{AND}(H_1(k_1),\ldots,H_1(k_m)) = 0$.

For an OR gate the following formulas can be derived similarly,

$$H_1(k) = \mathrm{AND}(H_1(k_1), H_1(k_2),\ldots,H_1(k_m)) \quad (3.3)$$
$$H_0(k) = \mathrm{OR}(H_0(k_1), H_0(k_2),\ldots,H_0(k_m)) \quad (3.4)$$

For an inverter the following formulas can be derived,

$$H_1(k) = H_0(k_1)) \quad (3.5)$$
$$H_0(k) = H_1(k_1)) \quad (3.6)$$

Implication formulas for NAND and NOR gates can be derived by combining the formulas for AND and OR gates with the formulas for the inverter. It can be seen that the implication rules for hazard status values are very similar to those for logic values. Therefore, the same implication tables can be used for both logic and hazard implications.

Schulz *et al.* [15] proposed a tuple representation for parallel path delay fault simulation, in which the logic values and hazard status are separately depicted. This representation is not optimal for test generation purposes, for the following reason. The hazard status of a gate output in [15] is a function of both the logic status and the hazard status of the fan-in lines. Therefore, the implication formulas of the hazard status are more complicated than the ones for the 4-tuple representation proposed here. The proposed 4-tuple format enables simpler and more efficient ways to calculate the hazard values of lines.

4. Test generation based on state tuples

We developed a test generator for path delay faults based on the state tuples proposed in this work. It can generate either robust or nonrobust tests for path delay faults in combinational circuits (and hence enhanced scan designs) and in standard scan designs. Our discussion is focused on robust path delay fault test generation in standard scan designs using functional justification. As mentioned before, 20 logic values are required to form a complete logic system for this problem [9]. Due to its complexity, no implementation of the 20-valued system is given in [9]. A 29-valued logic system was used in the test generator Fastpath [10] for industrial scan designs. In the experimental results section, we will compare the test generator reported here with the one reported in [10].

In this section the implementation of some key parts of the test generator based on the state tuple representation are described.

4.1 Implication procedures

The efficiency of forward and backward implication procedures are critical in a test generator. In previous works where multiple-valued logic systems are used, forward implication is performed using implication tables for logic gates. The size of an implication table of a two-input gate is quadratic in the number of logic values in the logic system, which can be quite large. Furthermore, different implication tables must be built for different logic systems. During the implication procedure, new implications must be checked to determine if they are consistent with the previously assigned values. For a logic system with a large number of values, consistency checking can be quite complicated. For example, the value H1 in the 20-valued logic system conflicts with 10 different logic values in that system.

The approach taken in this work is to evaluate logic values and hazard values separately. The events in the circuit can be classified into logic events and hazard events. A logic event on a line indicates a change of the logic values (v_1, v_2) of a line, while a hazard event indicates a change of its hazard values (h_1, h_0). A line may have logic events without hazard events, and vice versa. Each element in the tuple of a line is checked separately for consistency. Since the elements in the 4-tuple can only assume 3 values {0, 1, X}, consistency checking is much simpler than the procedure for multiple-value logic systems.

In the forward implication procedure, logic and hazard events are processed from the circuit inputs to its outputs. Logic events are processed as in any conventional test generator. Hazard values are computed by applying the formulas described before. An update of the hazard values on a line may trigger new hazard events on its fanout lines. On the inputs of the circuit, the hazard values are derived from the logic values on them using Table 4.1. The first column presents the initial value v_1 and the final value v_2 on an input line, and the second column presents the corresponding hazard values. It is assumed that there are no static hazards between the initial and final values applied to the input lines.

v_1, v_2	h_1, h_0
(0, 0)	(1, 0)
(1, 1)	(0, 1)
(1, 0), (0, 1)	(1, 1)
(X, 0), (0, X)	(1, X)
(1, X), (X, 1)	(X, 1)

Table 4.1 Relations between logic values and hazard values on inputs

The backward implication procedure for hazard events is similar to the one for logic values because of the similarity between their implication formulae. A conflict occurs when there is either a contradictory logic value assignment or a contradictory hazard value assignment on a line. By performing forward and backward implications for hazard values, those logic assignments that can cause hazard value conflicts can be avoided or discovered earlier.

An example of implication based on 4-tuples is shown in Figure 4.1. In Figure 4.1, the current logic and hazard values of a line are denoted in the form $v_1, v_2 / h_1, h_0$ in plain letters and the newly implied values are given in bold letters. Consider

justifying line *g* to 0, 0/1, 0 (i.e., a stable 0). This creates the backward implications, indicated by the arrows, on lines *e* and *b*.

It can be seen that even if the initial or final value of a line is X, it is still possible to determine the hazard status of the line. Line *f* is such an example. Failing to determine the hazard status on line *f* can cause a wrong choice to justify line *g* through line *f*.

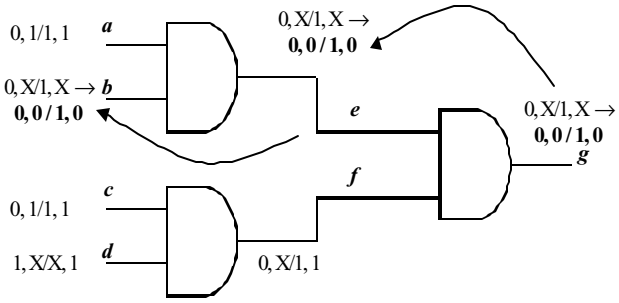

Figure 4.1 Illustration of backward implication

4.2 Backtracing procedure

During test generation, there may be unjustified lines such that logic and/or hazard values in their state tuples are not implied by the values on their fan-in lines. We call the set of lines on which only the logic values are unjustified the *J-frontier*, and the set of lines on which hazard values equal to 0 are unjustified (i.e., lines where stable values are required) the *SJ-frontier*. A backtracing procedure described in PODEM [16] can be used to justify a line in the J-frontier. A path consisting of lines with X values is backtraced from the objective line in the J-frontier to an input of the circuit. To justify a line objective in the SJ-frontier, a similar backtracing procedure can be used to backtrace through a path consisting of lines on which at least one of the hazard values (h_1, h_0) is X.

In the functional justification test environment for standard scan designs, a scanned circuit can be conceptually expanded into two time frames as illustrated in Figure 4.2. The primary inputs (outputs) are denoted as "PIs" ("POs") and the pseudo-primary inputs (outputs) are denoted as "PPIs" ("PPOs"). The states of the flip-flops in the second time frame are justified by applying the first test pattern to the first time frame. Each line in the second time frame has a duplicate in the first time frame. The duplicated line in the first time frame of a line *l* in the second time frame is denoted as *l'*. Backtracing for an objective line in the SJ-frontier starts from the line in the second time frame. Let (*L*, *V*) be an input assignment returned by the backtracing procedure. Here *L* is an input line and *V* is the logic value to be assigned to *L*. A full description of the backtracing procedure for a line in the SJ-frontier is given in Procedure 4.1.

Procedure 4.1: Backtracing for a line in the SJ-frontier

(1) Suppose the hazard value h_v of line *l* is to be justified to 0 (i.e., a stable *v* is to be justified on line *l*). Denote the gate that drives *l* as G_l. Denote the current line as *CL*, current value as *CV*, current gate as *CG*. Initially *CL* = *l*, *CV* = *v*, and *CG* = G_l.

(2) If *CG* has an inverted output, *CV* is set to its complement value. Select a fan-in line *f* of *CG* whose state tuple hazard value h_{cv} is an X value. Set *CL* to be *f* and set *CG* to be the gate driving *f*.

(3) If the current line *CL* is an input line of the second time frame (a primary input or a pseudo-primary input), then choose one of the following actions. Otherwise go to (2)

(a) If *CL* is a primary input line, return input assignments (*CL*, *CV*) and (*CL'*, *CV*) (i.e., the same value is assigned to the primary input in both time frames; the hazard status corresponds to a stable value *CV*).

(b) If *CL* is a pseudo-primary input line then perform a second backtrace from *CL* with the logic objective *CV* on it. Let the input line reached by the second backtrace be CL_1' with a backtraced value CV_1. If *CL'* = CL_1' return one input assignment (*CL'*, *CV*). Otherwise, return two input assignments (*CL'*, *CV*) and (CL_1', CV_1).

It can be seen from the above procedure that two input lines can be assigned logic values after a backtrace from a line in the SJ-frontier. An illustration of the backtrace procedure is given in Figure 4.2. In this example a stable value on line *l* is to be justified. The pseudo-primary input l_2 is reached by the first backtrace with the logic value *v*. The second backtrace is started from l_2 and reaches an input l_1' in the first time frame with value v_1. The backtrace procedure returns the value *v* on l_2' and the value v_1 on l_1'.

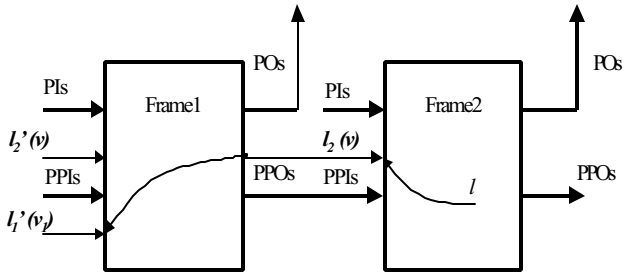

Figure 4.2 Backtrace from a line in the SJ-frontier

4.3 Overall procedure

The overall procedure of robust test generation is based on a PODEM-like search strategy. For a given path, all its on-inputs and side-inputs are assigned appropriate state tuples according to the requirements of a robust test. After that, forward and backward implications are performed to compute the logic and hazard values on the circuit lines and check for consistency. Many untestable paths are discovered at this stage. The unjustified lines are put into either the SJ-frontier or the J-frontier. The objective lines in the SJ-frontier have a higher priority for justification. Backtracing is performed to find a set of input assignments, which may help to justify an objective. During test generation, new implications of logic values and hazard values must be stored for each input assignment. When backtracking is used to resolve a conflict, the circuit is restored to the state before the last input assignment.

5. Experimental Results

The test generator described in the previous section was implemented in C++. Experiments were conducted on a set of ISCAS89 benchmark circuits using a Pentium IV 1.4 GHz PC with a Linux operating system. The results for robust test generation are presented in Table 5.1 and Table 5.2. All the experiments were done for the functional justification test mode with a backtrack limit of 100 per path.

Table 5.1 presents the experimental results of the proposed test generator and the industrial tool Fastpath described in [10]. Fastpath is a test generator capable of generating robust tests for standard scan designs. It uses a 29-valued logic algebra to perform implications. Results of robust test generation for 1000 longest paths per circuit under the functional justification mode, using a backtrack limit of 10,000 per path, were given in [10]. The paths targeted in [10] were selected using a commercial timing analyzer. For the experiments on the proposed test generator reported in Table 5.1, the paths targeted during test generation are selected in the following way. Let the maximum propagation delay from the inputs of the circuit-under-test to the outputs of the circuit be T_{max}. The start lines of these paths are picked randomly out of the set of circuit inputs such that the longest propagation delays from these inputs to the circuit outputs are greater than or equal to $0.8T_{max}$. Each path is selected to be one of the longest paths from the selected input to the outputs of the circuit. Thus all the selected paths have propagation delays that are greater than or equal to $0.8T_{max}$.

In Table 5.1, the test generation results using the proposed test generator and the results reported in [10] are listed in the columns with the headings "prop." and "[10]", respectively. The numbers of detected paths, untestable paths and aborted paths are reported under the columns with the headings "detected", "untest." and "aborted", respectively. For each circuit, the total number of backtracks taken by the proposed test generator for 1000 paths is reported in column "backtrack". The runtimes for test generation using the proposed method are reported in the last column. It can be seen that for all the circuits reported, test generation is done in a short time by the proposed test generator. With a backtrack limit of 100, no path is aborted for the reported circuits except for s5378. Although the results of our experiments cannot be directly compared to the data in [10] because of the different paths selected and the different experimental environments, it can be argued that the proposed test generator is significantly faster than Fastpath based on the number of backtracks. With a backtrack limit of 10,000 per path, Fastpath aborts more paths for the reported circuits. For example, 281 paths are aborted by Fastpath for s5378, whereas only 83 paths are aborted by the proposed test generator using a much smaller backtrack limit of 100. For circuit s5378, no path is aborted when the backtrack limit is raised to 3,000.

To better demonstrate the effectiveness of the test generation procedure, experiments were conducted on five different sets of 1000 paths for each circuit. The start lines of these paths are picked randomly out of the set of inputs of the circuit. Each path is selected to be one of the longest paths from the selected input to the outputs of the circuit. It should be noted that the selected paths are not necessarily the global longest paths in the circuit. Each path is only the longest one among the paths that start from the same input. The objective is to select a set of long paths, while trying to ensure that the selected paths cover different areas of the circuit. The average numbers over the five sets of paths for each circuit are reported in Table 5.2. The data in Table 5.2 provides further evidence that the test generator achieves high efficiency for a large number of paths in the circuit. On the average very few backtracks are done per path. For example, for circuit s38417 an average of 1.1 backtracks per path are done by the proposed test generator.

6. Concluding remarks

In this work we proposed the use of state tuples to describe the state of a line in a circuit. A 4-tuple representation was used for robust path delay fault test generation in scan designs. The first two elements of a tuple stand for logic values, and the next two elements stand for hazard states. The effectiveness of the test generation procedure based on state tuples was supported by experimental results.

References

[1] G. L. Smith, "Model for Delay Faults Based Upon Paths", *Proc ITC, pp. 342-349*, September 1985.

[2] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, "Transition Fault Simulation", *IEEE Design and Test, pp. 32-38*, April 1987.

[3] A. K. Pramanick and S.M. Reddy, "On the Computation of the Ranges of Detected Delay Fault Sizes," *Proc. ICCAD, pp. 126-129*, November 1989.

[4] V. S. Iyengar, B. K. Rosen, and J. A. Waicukauski, "On computing the sizes of detected delay faults", *IEEE TCAD, pp. 299-312*, March 1990.

[5] C. J. Lin and S. M. Reddy, "On Delay Fault Testing in Logic Circuits", *IEEE TCAD, pp. 694-703*, September 1987.

[6] K. T. Cheng and H-C Chen, "Delay Testing For Non-Robust Untestable Circuits", *Proc. ITC., pp. 954-961*, October 1993.

[7] K. Fuchs, F. Fink, and M. H. Schulz, "DYNAMITE: An Efficient Automatic Test Pattern Generation System for Path Delay Faults", *IEEE TCAD, pp. 1323-1335*, October 1991.

[8] K.-T. Cheng, S. Devadas, and K. Keutzer, "Delay-Fault Test Generation and Synthesis for Testability Under a Standard Scan Design Methodology", *IEEE TCAD, pp. 1217-1231*, August 1993.

[9] K. Fuchs, M. Pabst, and T. Rossel, "RESIST: A Recursive Test Pattern Generation Algorithm for Path Delay Faults Considering Various Test Classes", *IEEE TCAD, pp. 1550-1562*, December 1994.

[10] B. Underwood, W. O. Law, S. Kang, and H. Konuk, "Fastpath: A Path-Delay Test Generator for Standard Scan Designs", *Proc. ITC, pp. 154-163*, October 1994.

[11] P. Varma, "On Path Delay Testing in a Standard Scan Environment", *Proc. ITC., pp. 164-173*, October 1994.

[12] S. Bose, P. Agrawal, and V. D. Agrawal, "Generation of Compact Delay Tests by Multiple Path Activation", *Proc. ITC, pp. 714-723*, October 1993.

[13] S. Bose, P. Agrawal, and V. D. Agrawal, "Deriving Logic Systems for Path Delay Test Generation", *IEEE Trans. on Computers, pp. 829-846*, August 1998.

[14] A. K. Majhi and V. D. Agrawal, "Tutorial: Delay Fault Models and Coverage", *Proc. 11th Int'l Conf. on VLSI Design, pp. 364-369*, January 1998.
[15] M. H. Schulz, F. Fink, and K. Fuchs, "Parallel Pattern Fault Simulation of Path Delay Faults", *Proc. 26th DAC., pp. 357-363*, June 1989.
[16] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits", *IEEE Trans. on Computers, pp. 215-222*, March 1981.
[17] T. J. Chakraborty, V. D. Agrawal, and M. L. Bushnell, "Delay Fault Models and Test Generation", *Proc. 29th DAC, pp. 165-172*, June 1992.
[18] R. Desineni, K. N. Dwarkanath, R. D. Blanton, "Universal Test Generation Using Fault Tuples", *Proc. ITC, pp. 812-819*, October 2000.

Table 5.1 Robust test generation for 1000 paths

circuit	detected [10]	untest. [10]	aborted [10]	detected (prop.)	untest. (prop.)	aborted (prop.)	back-track (prop.)	runtime (prop.) [sec]
s1196	-	-	-	214	786	0	0	0.71
s1238	151	847	2	159	841	0	0	0.72
s1423	5	993	2	39	961	0	0	0.72
s1488	-	-	-	157	843	0	264	0.93
s1494	345	655	0	145	855	0	304	0.91
s5378	485	234	281	759	158	83	9384	4.98
s9234	0	1000	0	0	1000	0	0	3.30
s13207	0	1000	0	0	1000	0	0	5.85
s15850	0	1000	0	0	1000	0	0	4.80
s35932	0	1000	0	0	1000	0	0	10.77
s38417	185	732	83	0	1000	0	0	12.77
s38584	58	754	188	0	1000	0	0	12.92

Table 5.2 Average numbers over five sets of 1000 paths for robust test generation

circuit	detected (prop.)	untest. (prop.)	aborted (prop.)	backtrack (prop.)	runtime (prop.) [sec]
s1196	476.2	523.8	0.0	1772.8	0.88
s1238	487.8	512.2	0.0	777.6	0.84
s1423	178.6	821.4	0.0	14.0	0.56
s1488	180.8	819.2	0.0	716.2	1.07
s1494	165.8	834.2	0.0	773.6	1.15
s5378	780.2	207.6	12.2	1420.0	2.94
s9234	223.4	773	3.6	2099.4	3.92
s13207	320.2	679.8	0.0	1.6	4.87
s15850	188.8	810.8	0.4	52.4	5.73
s35932	172.4	827.6	0.0	0.0	13.20
s38417	305.8	686.6	7.6	1104.0	12.71
s38584	262.8	737.2	0.0	161.0	16.25