Short Papers

DYTEST: A Self-Learning Algorithm Using Dynamic Testability Measures to Accelerate Test Generation

WEIWEI MAO AND MICHAEL D. CILETTI

Abstract—**This paper presents a self-learning algorithm using a dynamic testability measure to accelerate test generation. It also introduces the concepts of full logic value label backward implication, the dependent backtrack, and *K*-limited backtracks. Experimental results indicating a high fault coverage are presented for ten benchmark combinational circuits.**

I. Introduction

Test generation algorithms [1], [5] include some [3], [5] that use static testability measures (STM) to guide the test generation (TG) process. These measures, like those in SCOAP [6], cannot account for backtrack-causing conflicts that can occur dynamically between the primary input assertions required for line justification and for path sensitization. Use of STM's can result in costly mistakes in the test generation process because they can lead the search down blind alleys in the decision tree. When this happens, repetitious backtracking can occur. Ivanov [7] introduced the use of a dynamic testability measure (DTM) that recalculates the static testability measures under the consideration of some lines having specific logic value during test generation. This method can also be costly because it involves time-consuming recalculation of testability measures.

The approach presented herein is to reduce backtracking by detecting and limiting its occurrence. Our design-aid program, DYTEST, employs a self-learning algorithm using a DTM to accelerate test generation. DYTEST's DTM guides the choice of a "potential main sensitizing path" during TG. A "full logic value label backward implication" creates *estimates* of logic values for the primary inputs. Dependent and *k*-limited backtracks are used to further reduce the number of backtracks.

II. DYTEST—an Overview

The combinational circuit to be tested is composed of connected AND, NAND, OR, NOR, and XOR gates having stuck-at-0/1 faults (S-a-0, S-a-1). All signal lines in the circuit are assigned a distinct index number according to the following rules: the index number of a gate's output line is larger than that of any input of the gate; the index number of a fan-out stem is smaller than that of any fan-out branch of the stem; and the index number of any fan-out branch of a stem is assigned according to SCOAP's static observability. At a stem, the index number of the fan-out branch having a lower observability value is larger than that of the other fan-out branches whose observability values are higher.

DYTEST begins by identifying and sensitizing a given fault. Then, it determines the unique logic implications—forward and backward—that result from the sensitization condition. This process reveals "MUST-BE" value labels that indicate, but do not effect, the logic values that must be assigned to some lines. Then, a *potential main sensitizing path* (PMSP) is chosen according to a testability measure that is calculated dynamically on the basis of the TG experience.

Manuscript received March 8, 1989; revised September 5, 1989 and October 20, 1989. This paper was recommended by Associate Editor S. C. Seth.

The authors are with the Department of Electrical and Computer Engineering, University of Colorado, Colorado Springs, CO 80907.

IEEE Log Number 9034775.

In order to sensitize the PMSP, a *full logic value label backward implication* (FLVLBI) is done. This provides a basis for estimating the logic values of the primary inputs. Then a PODEM-like implicit enumeration process is done. At each time when a failure occurs, the dependent backtrack and *K*-limited deep backtracks are used to reduce the number of backtracks.

III. Dynamic Testability Measure

When a TG algorithm using STM's chooses a sensitizing path, it sometimes correctly indicates a path that is relatively easy to observe. However, if one choice is misled by the STM, the wrong choice may be made repeatedly during subsequent stages of the sensitization process. In contrast, DYTEST avoids repetitive incorrect choices by adjusting its DTM according to the experience acquired *during* the TG process.

Consider the situation in Fig. 1. The line LS is a fan-out stem in a circuit; its fan-out branches are LB_1, LB_2, $\cdots$, LB_k. Let $F(LS)$ be a fault set containing faults in the shadow area, including faults at LS. If a TG algorithm selects a sensitizing path through LS, it faces the decision of choosing one of the branches emanating from the stem. If the algorithm remembers the past experience of the TG for the faults belonging to $F(LS)$, it can choose the sensitizing branch having the highest success-over-failure rate, because that choice has the highest likelihood of leading to TG success. Based on this idea, a DTM can be defined as follows:

Definition 1: Let each fan-out branch j in a circuit have success and failure records $n_s(j)$ and $n_f(j)$. During TG, if a chosen sensitizing path successfully leads to a valid test, the success record for each of the fan-out branches in the path is incremented, otherwise the failure record is incremented. The dynamic testability measure for a fan-out branch j is formed as $\mathrm{DTM}(j) = n_s(j) * W_s - n_f(j) * W_f$, with weights W_s, W_f subject to choice. (Experience suggests a 2 : 1 ratio in favor of successes.)

Three points should be noted here.

i) The testability measure is updated according to the TG experience, thereby providing dynamic guidance to the path sensitization process.

ii) The DTM implicitly accounts for both the observability and the controllability of the circuit, even though it is only considered at the fan-out branches—because the DTM is updated by the dynamic success and failure experience of the TG process;

iii) The DTM is easy to calculate.

IV. Forced Forward and Backward Implication

An initial step in TG sensitizes the fault for which a test is to be generated. For example, if L_4 in Fig. 2(a) has an S-a-0 fault, a logic value of 1 *must be* assigned to inputs L_1, L_2, and L_3 in order to sensitize the fault. After a faulty line is sensitized, other lines *must be* assigned to logic value (0 or 1) to propagate the fault to a primary output. If fan-out branch LB_1 has an S-a-1 fault in Fig. 2(b), then to sensitize it and propagate it to the output, a logic value of 0 *must be* assigned to LS, and a logic value of 1 *must be* assigned to L_1. Also, a logic value of 0 *must be* assigned to LB_2 and LB_3 because 0 is assigned to LS. We say that these lines have "MUST-BE," or constrained, values. Other lines are assigned MUST-BE value if their logic value is uniquely implied.

In DYTEST, forced implications are done if there are lines to which MUST-BE values are uniquely implied. Its implication procedure is similar to the complete implication used in the *D*-algorithm and FAN, but it differs in the following important ways: (1) DYTEST conducts the forced forward and backward implications only once—after a faulty line is sensitized. (2) After conducting

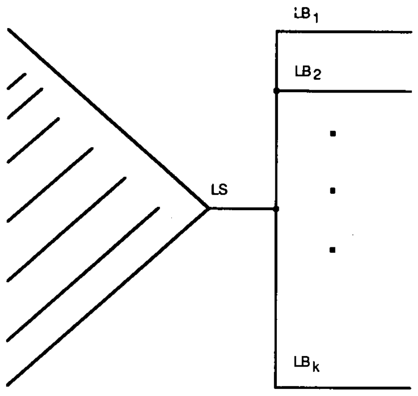


Fig. 1. A test generation algorithm propagating a sensitizing path to the fan-out stem LS must choose one of the emanating paths.

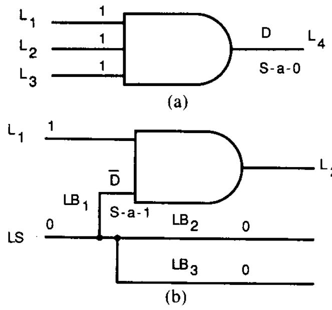


Fig. 2. (a) Lines L_1, L_2, and L_3 must be set to logic value 1 to sensitize an S-a-0 fault at line L_4. (b) A test for LB_1 S-a-1 results in LS, LB_2, and LB_3 having logic value 0.

the forced forward and backward implications, DYTEST creates MUST-BE value labels for lines whose logic values are uniquely implied. These labels are used in the failure check and report phases of the algorithm. The failure report is used in the dependent backtrack. The MUST-BE values determined by DYTEST are maximized by virtue of its using forced implications.

V. The Potential Main Sensitizing Path

In general, many paths exist for propagating a sensitized fault to a primary output. DYTEST chooses a potential main sensitizing path (PMSP) according to its DTM.

Definition 2: A path P is called a PMSP for a given fault if it satisfies the following conditions.

i) The path P begins at a faulty line and ends at a primary output.

ii) Each line in the path P has logic value D, $\overline{D}$ or X (don't care).

iii) If several paths pass through a fan-out stem and its branches separately, and all of them satisfy the conditions listed above, then the branch which the path P passes through is selected according to the highest dynamic testability value, i.e., the path remembered to have led to the most success in past choices.

If a gate input along the PMSP has a value of x, then its logic value label is assigned in order to propagate the fault through the gate.

The logic value label for a line L is denoted as $LVL(L)$. The label can be ZERO or ONE. The actual logic value for a line L is denoted as $V(L)$. The logic value can be 0, 1, D, $\overline{D}$ or X. For example, in Fig. 3, the PMSP for a fault at L_2 is composed of L_2, L_4, L_5, and L_8; lines L_1 and L_3 have logic value X. DYTEST assigns the logic value label ONE to L_1 and ZERO to L_3.

It should be noted that the concept of logic value label is different from that of logic value. Logic value label ZERO (ONE) for a line l means that a logic value of 0 (1) *may* be required for the line in order to actually sensitize the PMSP.

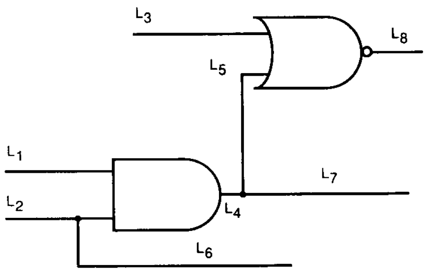


Fig. 3. A potential main sensitizing path (PMSP) for a fault at L_2 is composed of L_2-L_4-L_5-L_8.

The PMSP used in DYTEST is similar to the X-path checks used by PODEM. However, unlike PODEM, DYTEST uses the historical information summarized by its dynamic testability measures in constructing the path.

VI. Full Logic Value Label Backward Implication

After the PMSP is chosen and the logic value labels have been assigned to the other inputs of gates along the path, the FLVLBI is done for each gate type. Let X_i denote gate inputs, $1 \le i \le n$, and let Y denote gate output. The assignment is made as follows:

AND, OR Gate:
If $LVL(Y)$ = ZERO (ONE) and $V(X_i) = X$, then $LVL(X_i)$ = ZERO (ONE).

NAND, NOR Gate:
If $LVL(Y)$ = ZERO (ONE) and $V(X_i) = X$, then $LVL(X_i)$ = ONE (ZERO).

NOT Gate:
If $LVL(Y)$ = ZERO (ONE) and $V(X_i) = X$, then $LVL(X_i)$ = ONE (ZERO).

XOR Gate:

a) if $LVL(Y)$ = ZERO and $V(X_1) = V(X_2) = X$ then $LVL(X_1) = LVL(X_2)$ = ZERO or $LVL(X_1) = LVL(X_2)$ = ONE;

b) if $LVL(Y)$ = ONE and $V(X_1) = V(X_2) = X$ then $LVL(X_1)$ = ZERO, $LVL(X_2)$ = ONE or $LVL(X_1)$ = ONE, $LVL(X_2)$ = ZERO;

c) if $LVL(Y)$ = ZERO (ONE) and $V(X_2)$ (or $V(X_1)$) = 0 then $LVL(X_1)$ (or $LVL(X_2)$) = ZERO (ONE);

d) if $LVL(Y)$ = ONE (ZERO) and $V(X_2)$ (or $V(X_1)$) = 1 then $LVL(X_1)$ (or $LVL(X_2)$) = ZERO (ONE);

e) if $LVL(Y)$ = ONE or ZERO and $V(X_2)$ (or $V(X_1)$) = D or $\overline{D}$ then $LVL(X_1)$ (or $LVL(X_2)$) = ONE or ZERO.

At a fan-out stem, the FLVLBI is done as follows:

Let the fan-out stem be LS; its branches are $LB_1, LB_2, \cdots, LB_K$, and N_{one} (N_{zero}) are the number of branches having logic value label ONE (ZERO). The label choice is decided by one of two "majority rules:"

MR1: $LVL(LS)$ = ZERO if $N_{\text{zero}} \ge N_{\text{one}}$ and $V(LS) = X$; $LVL(LS)$ = ONE if $N_{\text{zero}} < N_{\text{one}}$ and $V(LS) = X$.

MR2: $LVL(LS)$ = ZERO if $N_{\text{zero}} > N_{\text{one}}$ and $V(LS) = X$; $LVL(LS)$ = ONE if $N_{\text{zero}} \le N_{\text{one}}$ and $V(LS) = X$.

If a faulty line L is an output of an AND (or NAND) gate that is S-a-1 (or S-a-0), then the FLVLBI is done for the gate as if $LVL(L)$ = ZERO (or ONE) is assigned to it. If a faulty line L is an output of an OR (or NOR) gate that is S-a-0 (or S-a-1), then the FLVLBI is done for the gate as if $LVL(L)$ = ONE (or ZERO) is assigned to it.

If a faulty line L is an output of an XOR gate that is S-a-1 (S-a-0), the FLVLBI is done as if $LVL(L)$ = ZERO (ONE) is assigned to it. If a MUST-BE 0 (or MUST-BE 1) label is assigned to an output L of a gate and some of the gate's inputs have logic value X, then the FLVLBI is done as if $LVL(L)$ = ZERO (ONE) is assigned to it.

When a primary input is met in the FLVLBI, it is pushed into a first-in-last-out stack for later use in constructing a binary decision tree. The FLVLBI is used to estimate the logic value requirement for the primary inputs.

The FLVLBI used by DYTEST is similar to other backtrace techniques, such as the multiple backtrace of FAN, but DYTEST does not explicitly guide its backward implication by calculating controllability measures. However, the controllability of the circuit is used implicitly because it affects the success and failure experience recorded during the backtraces. Thus DYTEST avoids the extra computation required by controllability measures, yet learns from its experience. We also note that FLVLBI sets all gate inputs to values. For example, if an AND gate's output label is to be a ZERO, then all input labels are set to ZERO.

VII. Dependent Backtrack

When a failure occurs in the TG process, it is important that the algorithm backtracks as little as possible. Most algorithms, including PODEM and FAN, only backtrack through the decision tree to the last choice and then make an alternative choice. For example, the binary tree for the backtracking of the circuit of Fig. 4(a) is illustrated in Fig. 4(b), where $X_1 = 1$, $X_2 = 1$, $X_3 = 1$, and $X_4 = 0$ are hypothetical initial choices for the primary input logic values. These result from FLVLBI, where a TG failure is assumed to occur at $X_4 = 0$. Choosing $X_4 = 1$ also fails due to violation of the MUST-BE 0 condition shown. A backtracking algorithm that examines the choices $X_3 = 0$ and $X_2 = 0$ will do so needlessly because these primary inputs have no influence on the line having the MUST-BE 0 requirement.

If a line having a MUST-BE 0 (1) label changes its logic value to 1 (0), or if a line at the PMSP has logic value 1 or 0, then a failure line report is given. If there is more than one failure line, the failure line with lowest index number is reported.

Definition 3: Let L_{fail} be a line whose failure has been reported. A primary input set $PI(L_{fail})$ is called the *dependent primary input set* of line L_{fail} if for any $PI_j \in PI(L_{fail})$, there is at least one path from PI_j to L_{fail} and no line in the path is a faulty line or has a MUST-BE value label. The backtrack in which the alternative choices are all members of $PI(L_{fail})$ is called the *dependent backtrack*.

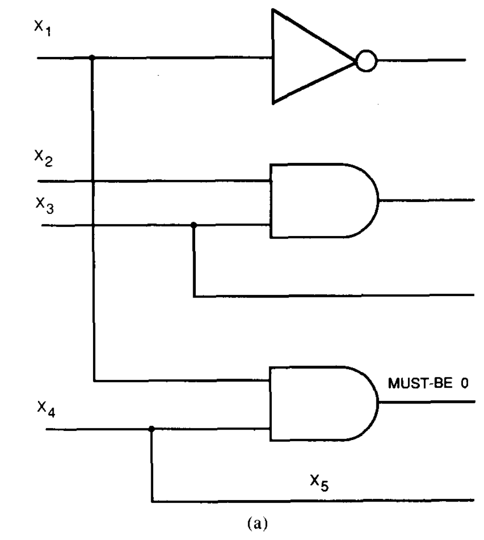
In the example of Fig. 4(a), the dependent primary input set for the MUST-BE 0 line condition failure consists of X_1 and X_4. Backtracking on the basis of this set leads to the efficient backtrack shown in Fig. 4(c).

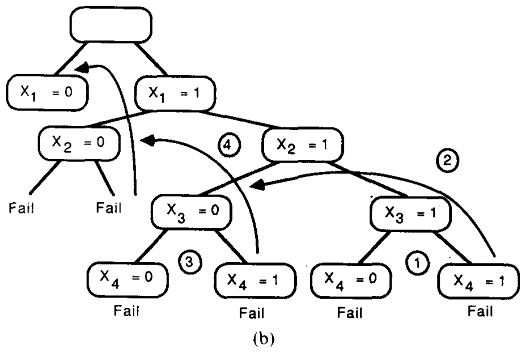
The binary decision tree used in our backtrack scheme is similar to that used in PODEM. When a failure is encountered, backtracking leads to change of a PI value. We note that X_2 and X_3 in Fig. 4(a) have no influence on the failure at the "MUST-BE 0" site, but may, in general, influence the detection of a fault in the circuit.

VIII. K-limited Deep Backtracks

Definition 4: Let A and B be primary inputs. Let node $A(V(A) \in \{0, 1\})$ have descendents $B(V(B) = 0)$ and $B(V(B) = 1)$ in the TG binary decision tree shown in Fig. 5. If a failure appears at $B(V(B) = 0)$, and if the algorithm backtracks to A and chooses $B(V(B) = 1)$, then primary input A is said to have a *one-level backtrack*. If a failure appears at a level lower than $B(V(B) = 0)$, and if the algorithm backtracks to A and chooses $B(V(B) = 1)$, then the primary input A is said to have a *deep backtrack*, to denote that the backtracking originates deep within the decision tree.

To avoid too many local backtracks, DYTEST uses a heuristic method to limit the depth of the backtrack at each primary input to a specified value, K. For example, if $K = 1$ for the binary decision tree shown in Fig. 6, the algorithm initially chooses nodes $A(V(A) = 1)$, $B(V(B) = 1)$, $D(V(D) = 1)$, $E(V(E) = 0)$, $G(V(G) =$





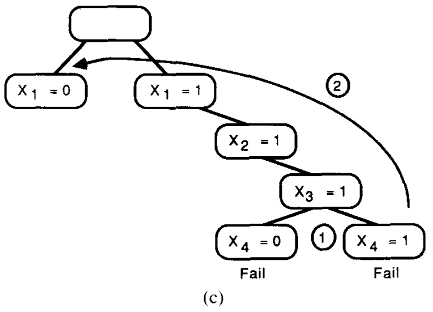


Fig. 4. (a) A circuit fragment that leads to backtracking. (b) Backtracking paths on the binary tree for the circuit fragment without dependent backtrack. (c) Backtracking paths on the binary tree for the circuit fragment when "dependent" backtracking is used.

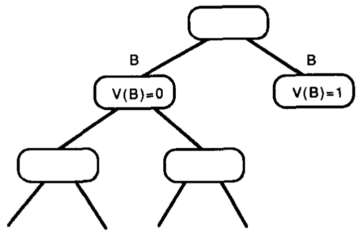


Fig. 5. Binary tree used to define one-level and deep backtracks.

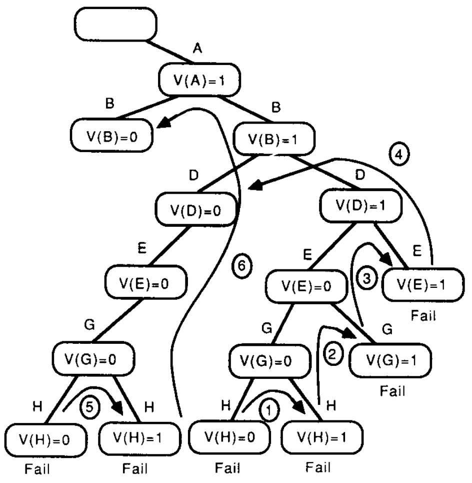


Fig. 6. Binary tree demonstrating DYTEST's avoidance of repetitive local backtracks.

0), and $H(V(H) = 0)$. A failure appears at $H(V(H) = 0)$ and again at $H(V(H) = 1)$. DYTEST then backs to $E(V(E) = 0)$ and chooses $G(V(G) = 1)$. If a failure appears at $G(V(G) = 1)$, then the algorithm backs to $D(V(D) = 1)$ and chooses $E(V(E) = 1)$. If a failure also appears at $E(V(E) = 1)$, then the algorithm backtracks to $B(V(B) = 1)$ and chooses $D(V(D) = 0)$, $E(V(E) = 0)$, $G(V(G) = 0)$, and $H(V(H) = 0)$. If a failure appears at $H(V(H) = 0)$ and $H(V(H) = 1)$ again, then the algorithm backs directly to $A(V(A) = 1)$ and chooses $B(V(B) = 0)$ because primary inputs D and E have been limited to one deep backtrack.

The objective in using K-limted backtracks is to avoid fruitless local backtracking and hasten the search. If the number of deep backtracks is limited to a chosen number K, the TG algorithm may fail to generate a test for a detectable fault and is said to be "incomplete." However, if the selected number K is large enough, then the completeness of the algorithm is not affected. The number K can be determined according to the maximum backtrack number for practical TG. Our work keeps K fixed to ensure fast backtracking. Increasing K might improve the coverage, provided that detectable faults were not found with $K = 1$. (There is no *a priori* method for determining the number of faults that remain to be found by changing K.) We note, however, that the coverage obtained in our experiments agrees closely with that reported by others. This suggests that increasing K would be futile for the circuits. In fact, increasing K without increasing the total number of backtracks may actually lower the fault coverage while simultaneously increasing the time expended.

IX. Test Procedures and Test Modes

DYTEST is configured to support two test procedures, TP1 and TP2, both of which may use four test modes. TP1 is relatively simple and allows only one backtrack. Imposing this constraint on the backtracking in TP1 allows the dynamic testability measures to acquire successful experience. TP2 is more complete, and it allows the user to select the number of backtracks and the value of the K-limit. Both procedures allow choice of all four test modes and the order in which they are performed. The flowchart of DYTEST is shown in Fig. 7. Its steps are explained as follows.

First, the fault is sensitized in a manner similar to the sensitization steps in the D-algorithm and FAN. Then the forced forward and backward implications are executed, followed by selection of a PMSP according to the DYTEST's DTM. If no PMSP exists, the fault is untestable. Next, the full logic value label backward implication is made. Then, one of four test modes is invoked. These modes differ in their choice of label assignment at conflict fan-out stems and in the order of primary inputs in the stack. The test modes are denoted by TM(i, j) with $i, j \in \{0, 1\}$. Each mode makes

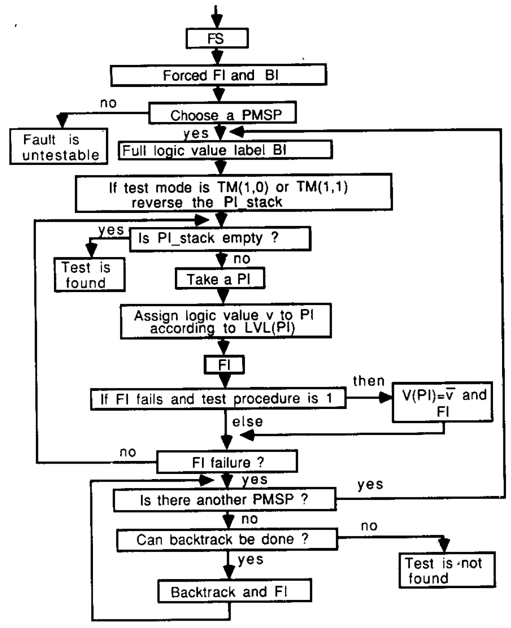


Fig. 7. Flowchart for DYTEST's algorithm.

choices in the FLVLBI when a conflict fan-out stem is met. These are given below.

TM(0, 0): The majority rule MR1 is used at fan-out stems. No change is made in the order of primary inputs in the stack.
TM(0, 1): The majority rule MR2 is used. No change is made in the order of the PI's in the stack.
TM(1, 0): The majority rule MR1 is used. The order of the PI's in the stack is reversed.
TM(1, 1): The majority rule MR2 is used. The order of PI's in the PI_stack is reversed.

After the test mode is chosen, the forward logic implication is made. It should be noted that in Test Procedure 1, if the forward implication fails for an assignment of logic value, the complementary logic value is tried immediately. This step will cut the branches of the binary decision tree quickly.

The binary decision tree in DYTEST is similar to PODEM's. However, DYTEST uses the dependent backtrack and K-limited deep backtrack techniques to reduce the number of backtracks.

X. Experimental Results

DYTEST is implemented on a VAX8600 in the C programming language. Ten benchmark combinational circuits [8] were tested using only DYTEST—no fault simulation was used.

Table I shows a comparison of the fault coverage provided by the STM's and DTM's after one pass, with the DTM using Test Procedure 1 with the test mode TM (0, 0). The maximum backtrack number is 1. In every case the DTM achieves better fault coverage.

Next, all four passes of TG with test modes TM(0, 0), TM(0, 1), TM(1, 0), and TM(1, 1) were done using Test Procedure 1. The maximum backtrack number is 1. After each pass the detected faults are deleted from the fault list and the result is shown in Table II. For the remaining undetected faults, another four passes with test modes TM(0, 0), TM(0, 1), TM(1, 0), and TM(1, 1) were done using Test Procedure 2 with 50 maximum backtracks and 1 limited deep backtrack for each primary input in a binary decision tree. This results in the final fault coverage shown in Table III. The achieved fault coverages are better than those reported in [5]. Table I also shows that a single pass of the DTM requires less time than the STM.

The computation time required for DYTEST in these experi-

TABLE I
A COMPARISON OF FAULT COVERAGES AFTER ONE PASS USING TP1 AND TM(0, 0).

CIRCUIT	NUMBER OF FAULTS	FAULT COVERAGE IN %		NORMAL RUN TIME	
		STM	DTM	STM	DTM
C432	524	86.45	86.83	1.04	1.00
C499	758	67.94	78.63	1.16	1.00
C880	942	100.00	100.00	1.15	1.00
C1355	1574	76.87	79.54	1.09	1.00
C1908	1879	88.56	91.33	1.12	1.00
C2670	2747	71.20	83.91	1.02	1.00
C3540	3428	51.20	69.19	1.18	1.00
C5315	5350	92.58	94.52	0.98	1.00
C6288	7744	54.48	55.09	1.32	1.00
C7552	7550	73.17	77.97	1.15	1.00

TABLE II
ACCRUED FAULT COVERAGES USING TP1 WITH TM(0, 0), TM(0, 1), TM(1, 0), AND TM(1, 1).

CIRCUIT	NUMBER OF FAULTS	FAULT COVERAGE IN %			
		T(0,0)	T(0,1)	T(1,0)	T(1,1)
C432	524	86.83	98.09	98.66	----
C499	758	78.63	86.94	98.15	98.42
C880	942	100.00	----	----	----
C1355	1574	79.54	87.04	97.27	98.79
C1908	1879	91.33	98.72	99.10	99.15
C2670	2747	83.91	90.57	91.92	93.52
C3540	3428	69.19	81.13	91.57	92.36
C5315	5350	94.52	96.90	97.85	98.19
C6288	7744	55.09	79.60	89.62	94.56
C7552	7550	77.97	83.60	94.97	96.19

TABLE III
ACCRUED FAULT COVERAGES USING TP2 AND TM(0, 0), TM(0, 1), TM(1, 0), AND TM(1, 1) WITH 50 BACKTRACKS AND 1 LIMITED DEEP BACKTRACK.

CIRCUIT	NUMBER OF FAULTS	FAULT COVERAGE IN %			
		T(0,0)	T(0,1)	T(1,0)	T(1,1)
C432	524	----	98.85	----	----
C499	758	98.94	----	----	----
C880	942	----	----	----	----
C1355	1574	99.24	----	99.49	----
C1908	1879	99.41	99.57	----	----
C2670	2747	95.23	95.63	95.74	95.78
C3540	3428	94.78	95.10	95.62	95.80
C5315	5350	98.58	98.64	98.73	98.77
C6288	7744	94.58	98.67	----	99.56
C7552	7550	97.01	97.31	98.11	98.19

TABLE IV
CUMULATIVE CPU TIME TO ATTAIN THE FINAL COVERAGES SHOWN IN TABLE III.

Circuit	CPU time (Sec./Per Fault)
C432	0.12
C499	0.16
C880	0.13
C1355	0.45
C1908	0.50
C2670	1.26
C3540	1.47
C5315	0.88
C6288	2.58
C7552	3.26

ments is shown in Table IV. It is stressed that the experiments reported here were designed to compare the coverage provided by STM's and DTM's, and the relative time required by their TG process—without fault simulation. Futhermore, the data structures employed by DYTEST have not been optimized for speed. Thus the time expended by DYTEST in these experiments is not necessarily optimal.

XI. CONCLUSION

DYTEST is a TG algorithm guided by a DTM. The experimental results show that DYTEST achieves higher fault coverage with a DTM than it does with an STM, and it does so in less time. Its fault coverage compares favorably with other STM-driven TG [5]. As noted by a reviewer, a dynamic DTM like ours can be misled by "wrong learning experience" in the early stage, or by having a "good learning experience" destroyed by encountering many "hard-to-detect" faults in the middle of the test process. (This liability is common to heuristic methods.) However, for the ten benchmark circuits examined, experience has shown that the performance of the DTM is at least as good as the performance of an STM, and it can be dramatically superior.

DYTEST uses the novel dependent backtrack and K-limited deep backtrack techniques. The experimental results show that high fault coverage can be achieved in a few backtracks. The experimental results also show that the combinational use of four test modes and two test procedures is a good TG strategy. Additional speedup can be achieved by combining DYTEST with fault simulation. Future work will examine the merits of varying K and the success/failure weights, and consider other strategies in the FLVLBI.

REFERENCES

[1] J. P. Roth, "Diagnosis of automata failures: A calculus and a method," *IBM J. Res. Develop.*, vol. 10, pp. 278-291, July 1966.
[2] C. W. Cha, W. E. Donath, and F. Ozguner, "9-V algorithm for test pattern generation of combinational digital circuits," *IEEE Trans. Comput.*, vol. C-27, pp. 193-200, Mar. 1978.
[3] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Trans. Comput.*, vol. C-30, pp. 215-222, Mar. 1981.

[4] H. Fujiwara, T. Shimono, "On the acceleration of test generation algorithm," *IEEE Trans. Comput.*, vol. C-32, pp. 1137-1144, Dec. 1983.
[5] S. Patel and J. Patel, "Effectiveness of heuristics for automatic test pattern generation," in *Proc. 23rd IEEE Design Automation Conf.*, June 1986, pp. 547-552.
[6] L. H. Goldstein, "Controllability/observability analysis of digital circuit," *IEEE Trans. Circuit, Syst.*, CAS-26, pp. 685-693, Sept. 1979.
[7] A. Ivanov and V. K. Agarwal, "Dynamic testability measures for ATPG," *IEEE Trans. on Computer-Aided Design*, pp. 598-608, Dec. 1988.
[8] F. Brglez and H. Fujiwara, "A neutral netlist of the combinational benchmark circuits and a target translator in FORTRAN," presented at IEEE Int. Symp. on Circuits and Systems, Kyoto, Japan, June 5-7, 1985.

Computing Signal Delay in General *RC* Networks by Tree/Link Partitioning

PAK K. CHAN AND KEVIN KARPLUS

Abstract—**Most *RC* simulators handle only tree networks, not arbitrary networks. We present an algorithm for computing signal delays in general *RC* networks using the *RC*-tree computation as the primary operation. We partition a given network into a spanning tree and links. Then we compute the signal delay of the spanning tree, and update the signal delay as we incrementally add the links back to reconstruct the original network. If m is the number of links, this algorithm requires $m(m + 1)/2$ updates and $m + 1$ tree delay evaluations. All the tree delay evaluations involve computing signal delays with the same resistive spanning tree, but with different values for the capacitors.**

I. Introduction

The linear *RC* model has become an acceptable and pragmatic approach for modeling digital MOS circuits in the past decade. Research has been carried out both in bounding the waveforms [17], [22], [24] and in estimating the signal delays [3], [7], [13], [14], [19] of *RC* networks. In particular, Elmore's notion of signal delay [8] has been used widely to approximate the time taken for a signal to start from an initial value and reach half of its final value.

If $\boldsymbol{G}$ is the node conductance matrix of a given *RC* network and $\boldsymbol{C}$ is the capacitance matrix of the network, calculating Elmore's delay, t_d, can be as simple as evaluating the product of $\boldsymbol{G}^{-1}$, $\boldsymbol{C}$, and unit vector $\boldsymbol{1}$. Since $\boldsymbol{G}$ and $\boldsymbol{C}$ are given, the delay estimation problem amounts to computing the resistance matrix, $\boldsymbol{R} \equiv \boldsymbol{G}^{-1}$. Thus delay estimation in *RC* networks has been viewed as a numerical problem: inverting $\boldsymbol{G}$ [13]. The computational requirement of this numerical approach limits its applicability to general problems. However, if the *RC* network is a tree, then $\boldsymbol{R}$ can be determined by inspection, and t_d can be computed in linear time. Almost all MOS timing level simulators treat networks as if they were trees, trading off accuracy for simplicity [3], [14], [19].

Signal delays in tree networks are easy to evaluate; unfortunately, many practical MOS circuits are not trees. The Manchester adder with carry-bypass circuitry, as depicted in Fig. 1, is an example [20]. Since the bypass transistor B is connected to $\overline{C}_0$ and $\overline{C}_4$, when this transistor is ON and all the P_i are high, they form a closed loop of conducting transistors, and cannot be modeled as an *RC* tree.

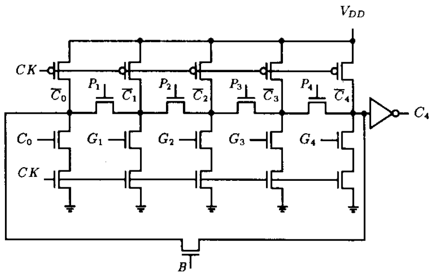


Fig. 1. Manchester adder with carry-bypass.

The goal of this paper is to provide an efficient way of computing signal delays in *RC* networks that do not necessarily form trees. We partition a given network into a spanning tree and links. We then start with the signal delay of the *spanning tree*, and gradually update the signal delay while incrementally piecing the *links* back to reconstruct the original network. The concept of circuit partitioning will be explained in Section III. A delay computation algorithm and its complexity will be explained in Section IV. Here, we shall consider only *RC* networks with grounded capacitors at each node of the network and no floating capacitors. Extensions to handle floating capacitors are discussed in [5].

A note on related work: Lin and Mead invented the technique of "tree decomposition" and "load redistribution" to calculate signal delay in general *RC* networks [13]. Their algorithm is relaxation-based, and the number of relaxation steps depends on the required accuracy. The algorithm that we are presenting here is based on dynamic programming, and terminates in a predetermined number of steps with the exact result.

II. Signal Delay in *RC* Networks

The *delay estimation problem* aims to find the time interval that it would take a signal to start from an initial value and reach a prescribed value. The most meaningful such value for digital circuits is the threshold voltage where the two logic states cross. However, locating this delay time precisely can be as hard as finding the exact waveform. A notion of delay defined in terms of the first moment of the impulse response was introduced by Elmore [8]. Many researchers have used the normalized Elmore delay to approximate such a delay time [2], [3], [13], [23]:

$$t_d \equiv \frac{\int_0^\infty h(t)t\,dt}{v(\infty) - v(0)} \tag{1}$$

where $v(\infty)$ and $v(0)$ are the final and initial voltages. If $v(t)$ is is the voltage response due to a unit step input, then an equivalent definition of Elmore delay is [17], [22], [23]:

$$t_d \equiv \frac{\int_0^\infty [v(\infty) - v(t)]\,dt}{v(\infty) - v(0)}. \tag{2}$$

Based on this notion of signal delay, closed-form delay expressions can be derived for *RC* networks without floating capacitors. We note that for a simple *RC* circuit with only one resistor and one grounded capacitor, Elmore delay is the same as the time constant of the circuit.

For a given *RC* network with n grounded capacitors, let

- $G_{i,j}$ be the branch conductance between nodes i and j (and by reciprocity, $G_{i,j} = G_{j,i}$);

Manuscript received August 28, 1988; revised July 1, 1989 and September 6, 1989. The work of P. K. Chan was supported in part by the University of California MICRO Program under Grant 89-136. This paper was recommended by Associate Editor R. K. Brayton.

The authors are with the Department of Computer Engineering, University of California, Santa Cruz, CA 95064.

IEEE Log Number 9035469.