VLSI Design
1994, Vol. 2, No. 1, pp. 69–80
Reprints available directly from the publisher
Photocopying permitted by license only

Temporal Logic Based Hierarchical Test Generation for Sequential VLSI Circuits

ANAND V. HUDLI
Department of Computer and Information Science, Purdue School of Science, Indiana University—Purdue University, Indianapolis, IN 46202, U.S.A.

RAGHU V. HUDLI
IBM Corporation, 11400 Burnett Road, Austin TX 78758, U.S.A.

(Received May 21, 1990, Revised January 6, 1993)

Test generation for sequential VLSI circuits has remained a difficult problem to solve. The difficulty arises because of reasoning about temporal behavior of sequential circuits. We use temporal logic to model digital circuits. Temporal Logic can model circuits hierarchically. A set of heuristics is given to aid during test generation. A hierarchical test generation algorithm is proposed.

Key Words: *Temporal Logic; Test Generation; Sequential VLSI Circuits; Hierarchical Test Generation*

1. INTRODUCTION

Digital circuits are tested during manufacture and field operation by application of a sequence of test vectors. A failure is indicated if the circuit response does not match the expected response of the "good circuit." The quality of the produced circuit (which could be a chip, a board, or a whole system) is determined by "test escapes," that is, the fraction of bad units which pass the test. These units eventually cause field failures which are not only expensive to diagnose and repair but may also adversely affect the image of the company in terms of its product quality. Accurate measurement of test escapes is very difficult for a variety of reasons, hence manufacturers rely on "fault coverage" of the test-vector sequence as an indirect measure of the quality of the tested product. The fault coverage is the fraction of all the modeled faults detected by the sequence. With increasing emphasis on product quality, the minimum required fault coverage values are climbing into the 99% range. The purpose of test generation is to develop a test vector sequence with an acceptable fault coverage.

While the test generation problem is NP-complete even for combinational circuits [11], there is some evidence that the average-time complexity may not be exponential [24]. This theoretical result is supported by recent advances in test generation algorithms for combinational logic with impressive results reported on benchmark circuits [21]. It is now possible to complete test generation for a combinational circuit with several thousand gates in a matter of few seconds on standard work stations. Comparable test generators for sequential circuits do not exist because of the inherent greater complexity of the problem. For reasons of initialization, multiple-time-frames, asynchronous nature, and problems of "hazards" and "races," test generation for even a moderate size circuit of a thousand gates could consume more than 24 hours of VAX-8650 CPU time [1].

We believe the formalism chosen to represent sequential circuits is an important factor that determines the complexity of the test generation problem. A representation in which a circuit is just an interconnection of logic gates and latches is bound to extract unacceptable time penalties in search problems related to circuit initialization and justification of internal line values. In this paper we propose a temporal logic [26] based representation which can model the timing behavior of circuit elements of arbitrary complexity in a uniform manner. We extend

the linear time temporal logic [26] so as to be able to reason about the likely past events causing the known current events and develop specific heuristics to accelerate the search process implied by such reasoning. Both synchronous and asynchronous circuits can be represented by our formalism. Further, the circuits could be built from arbitrary components described by temporal logic formulas.

2. PROBLEM DEFINITION

We will first illustrate the test generation problem for combinational circuits to introduce some basic concepts in a simpler setting. The fault model we choose is the *stuck-at* model [18]. According to this model, faulty lines in a circuit have eternally constant logical value. For example, in Fig. 1, the output of gate H is permanently 0. This fault is represented as sa-0—read stuck at 0. Faults can be either sa-0 or sa-1. The test generation problem for combinational circuits is to find an input vector that will detect a given fault. A fault is said to be detected by a vector if the faulty circuit and fault-free circuit have opposite logical values at the output. We have to choose a vector that puts a logical value which is opposite to the faulty value at the site of the fault—this is called excitation of the fault—and the input vector must also be able to propagate the effect of the fault to the output. For example, in Fig. 1, for the fault output of H *sa*-0, we need $\langle A, B, C, D\rangle = \langle 0, 0, 1, 1\rangle$ to excite the fault and E has to be 0 to propagate the faulty value to Z. If E is 1, then $Z = 1$ regardless of the fault on H. So E has to be 0. The output is 1 in the fault-free case and 0 in the faulty case. Therefore the vector $\langle 0, 0, 1, 1, 0\rangle$ detects output of H sa-0. The vector that detects a fault is called a test pattern. It is easy to verify that the vectors $\langle 0, 1, 1, 1, 0\rangle$ and $\langle 1, 0, 1, 1, 0\rangle$ also detect the fault output of H sa-0. The above three test patterns can be written as $\langle 0, X, 1, 1, 0\rangle$ and $\langle X, 0, 1, 1, 0\rangle$, where X is used to denote 0 or 1.

Though the test generation problem for combinational circuits is NP complete, there are many algorithms that generate tests for combinational circuits efficiently [9, 10, 20, 21]. Some AI based programs also have been developed recently [2, 22, 23].

The problem is more complicated for sequential circuits. We need a sequence of vectors rather than one vector to detect a fault. For example, in Fig. 2, to detect Q sa-0, the sequence needed is $[A, Clk, B\rangle$:$\langle 1, \uparrow, X\rangle, \langle X, X, 1\rangle$. The $\uparrow$ is used to denote a rising clock, that is, a $0 \rightarrow 1$ transition. Z will be 0 in the faulty case and 1 in the fault-free case. In this case we need a vector to excite the fault and a vector to propagate the effect of the fault. In general, we will need a sequence of vectors to excite a fault and a sequence of vectors to propagate the effect of the fault.

Unlike for combinational circuits, effective test generation algorithms for sequential circuits, have yet to become a reality. However, recently various new algorithms have been proposed [1, 8, 13, 14, 19]. The Iterative Test Generator (ITG) [19], that transforms a sequential circuit into an iterative array of combinational time frames, can be thought of as an extension of combinational test generation. The Extended Back Trace (EBT) [14] algorithm, that is a deliberate single path sensitizer, is also an extension of the classical D-algorithm. CONTEST (CONcurrent TEST generation) [1] uses simulation to generate test sequences. As faults are simulated in a concurrent fault simulator [15], a cost function is simultaneously computed. The input vector is then modified to reduce the cost function until a test is found. BACK [8] algorithm is similar to EBT with different heuristics. Ma et al. use state transition

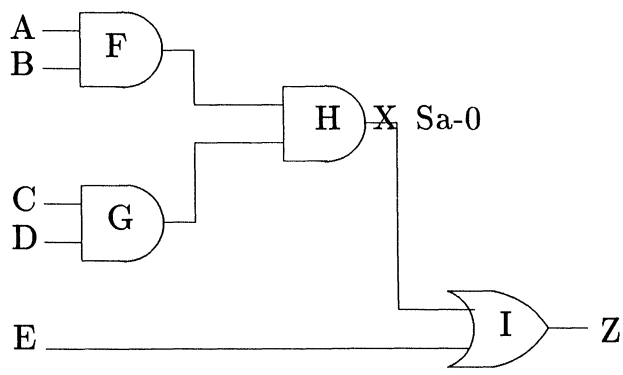

FIGURE 1 A simple combinational circuit.

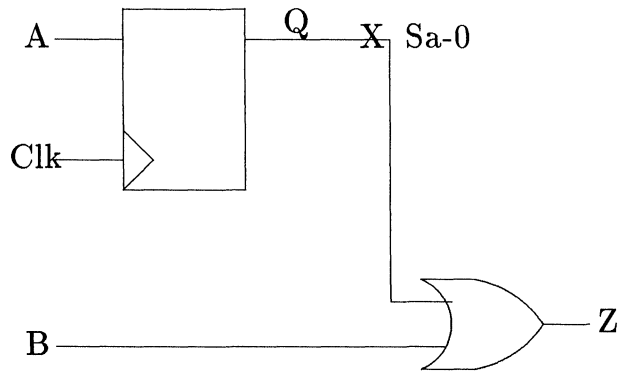

FIGURE 2 A simple sequential circuit.

graphs for test generation [13]. The drawbacks of this algorithm are excessive storage requirement for the state transition graph, and long fault simulation times for all the potential set up sequences.

Our goal is to develop an efficient scheme for test generation for sequential circuits. We choose temporal logic as a formalism for describing sequential circuits. We also give some heuristics to guide the search for test sequences. Temporal logic can be used for multilevel specification of digital circuits [17]. Since our algorithm works on temporal logic specification, it is possible to to hierarchical test generation.

3. REVIEW OF TEMPORAL LOGIC

In this section, we give a brief description of temporal logic [26]. Traditional logic uses operators such as $\vee$, $\wedge$, $\neg$, $\supset$, which stand for logical AND, OR, negation and implication. Temporal logic introduces additional operators for dealing with temporal sequences. While expressions in traditional predicate calculus specify properties of the system state at some given instant of time, which we will call the *present time,* temporal logic formulas specify properties of possible sequences of states that may evolve from present time. Temporal logic retains the logical connectives of predicate logic mentioned above and uses the following temporal operators.

1. Always Operator—$\Box$
   The expression $\Box\omega$ means that the formula ω is true at the present time and at all future times.

   Example: $B \supset \Box(A \wedge C)$

   The above formula means that if B is true, then A and C are true eternally in future.
2. Sometimes Operator—$\Diamond$
   The formula $\Diamond\omega$ is used to specify that ω will be true some time in future. In the literature this is treated as a primitive temporal operator. However, we note that $\Diamond$ can be derived from $\Box$ operator as below.

   $$\Diamond\omega \equiv \neg\Box\neg\omega$$

3. Next Operator—$\bigcirc$
   This is the most useful operator for test generation applications of temporal logic. The formula $\bigcirc\omega$ means that ω will be true in the next instant of time. This operator introduces the concepts of discrete time and transition that occurs between subsequent instants of time.
   It is important to note that the transition times of circuits are short compared to the time intervals of $\bigcirc$ operator. This means that the next transition of the input signals occur only when output signals have assumed stable value. For example, in Fig. 2, the temporal behavior of Q can be expressed as

   $$(A = 1 \wedge \uparrow Clk) \supset \bigcirc(Q = 1)$$

   $$(A = 0 \wedge \uparrow Clk) \supset \bigcirc(Q = 0)$$

4. Until Operator—$\mathcal{U}$
   $\omega\mathcal{U}\beta$ means that A is true till the time instant B is true. This operator can be used for specifying asynchronous behavior of digital circuits. For example to specify that the interrupt acknowledge signal—INTACK—does not rise till an interrupt request signal—INTREQ—rises, we write

   $$\neg(\uparrow \text{INTACK})\mathcal{U}(\uparrow \text{INTREQ})$$

Temporal logic in various forms—linear [3, 26], interval [17] and branching time [6]—has been used for specifying digital hardware and also for formal verification of digital circuits [3, 6, 7]. Our research is the first attempt of using temporal logic for test generation.

3.1 Modeling Sequential Circuits in Temporal Logic

Temporal logic can be used to model circuits at multiple levels. We give examples in this section that illustrate modeling at gate level and functional level.

Consider Fig. 3. The circuit can be specified as a set of temporal logic statements, as below. There are six combinational gates and two D flip-flops. Specifications for combinational gates look like traditional predicate calculus statements, since they do not exhibit temporal behavior.

1. $(G2 = 0 \wedge \uparrow ClK) \supset \bigcirc(Q0 = 0)$
2. $(G2 = 1 \wedge \uparrow ClK) \supset \bigcirc(Q0 = 1)$
3. $(G6 = 0 \wedge \uparrow Clk) \supset \bigcirc(Q1 = 0)$
4. $(G6 = 1 \wedge \uparrow Clk) \supset \bigcirc(Q1 = 1)$
5. $(G3 \vee G4 \vee G5) \supset G6$
6. $(Q0 \wedge \neg Q1) \supset G5$
7. $(D \wedge Q1 \wedge \neg Q0) \supset G4$

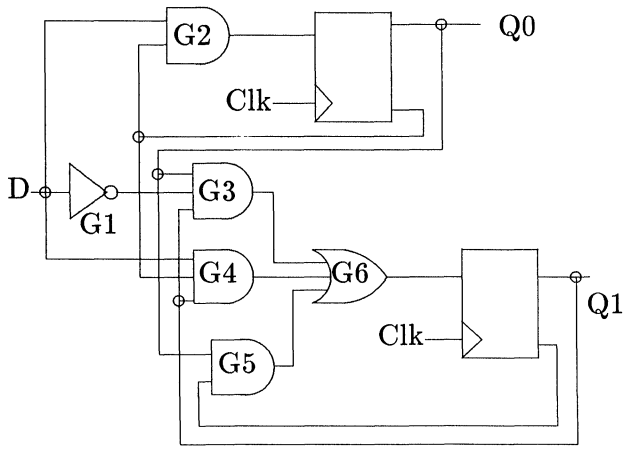

FIGURE 3 An example sequential circuit.

8. $(G1 \wedge Q1 \wedge Q0) \supset G3$
9. $(D \wedge \neg Q0) \supset G2$
10. $\neg D \supset G1$

According to the first formula, if the output of gate $G2$ is 0, and the clock rises now, then the output of the D flip-flop $Q0$ will also be 0, at the next instant of time. The instant of time that is implicit is one clock cycle. So the flip-flops are *edge-triggered* flip-flops. The formulas 2–4 are similar to formula 1. The remaining formulas, model combinational gates, and hence their specifications look like boolean logic formulas.

Consider Fig. 4. The figure is a serial adder. The temporal specification for the two output ports are

1. $((A \oplus B \oplus Cin) = 1) \supset Sum = 1$
2. $((A \oplus B \oplus Cin) = 0 \supset Sum = 0$
3. $((A\ B + B\ Cin + A\ Cin) = 1) \supset Cout = 1$
4. $((A\ B + B\ Cin + A\ Cin) = 0) \supset Cout = 0$
5. $Cout = 1 \supset \bigcirc(Cin = 1)$
6. $Cout = 0 \supset \bigcirc(Cin = 1)$

The adder part of the circuit is strictly combinational, hence the formulas 1–4 do not have any temporal operators. However, the carry-in (Cin) for the next time instant is the same as the carry-out for the current time. Hence formulas 5 and 6 have the next ($\bigcirc$) operator.

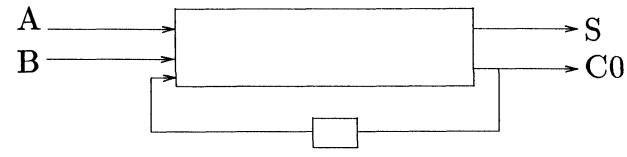

FIGURE 4 A serial adder.

4. OUR APPROACH

Our approach to the test generation problem is based on cause-effect type of reasoning. This technique is used to solve the line justification problem. The line justification problem is to find what inputs cause a particular logic value to appear on a line in the circuit. The line justification problem is solved by solving a set of reverse temporal-$\triangle$-formulas. To find a test sequence, a path along which the effect of the fault can be propagated is determined. This creates several line justification problems. Then, we solve the line justification problems. When all the problems have been solved, a test sequence is obtained. For economy, we limit the search to solutions which can be found by sensitizing a single path to a primary output though this is not a fundamental limitation of our technique.

4.1 Reverse Temporal Operators

Forward temporal logic operators, are useful in reasoning about future events in time. So, these operators are useful for simulation where we have to find the response of digital circuits to input stimulus. However, in test generation, we need tp reason about the past events also. For example, to initialize a line to a certain value at the present time, we need to find what events in the *past* could have resulted in the desired value on the line of interest at the *present time*. Such needs in test generation lead to the definition of two reverse temporal operators, that are similar to the forward operators.

1. Previous Operator—$\triangle$
   This operator is the reverse of the $\bigcirc$ (next) operator.
   $\triangle(w)$—read *previous* w is true if w was true in the previous state/time interval.
   $\triangle^n(w)$ is true if w was true in the nth previous time instant. $\triangle$ is cumulative.

$$\triangle^i(\triangle^j)(w)) = \triangle^{i+j}(w)$$

   We will show the utility of this operator in the solution of the line justification problem.
2. Hither-to Operator λ
   This operator is the reverse of the $\Box$ (henceforth) operator. $\lambda_k(w)$—read *hither to* w is true if w has been true in the previous k time interval.

λ models the memory state of the flip-flops. λ is also cumulative.

$$\lambda_1(w) \equiv \triangle(w)$$

We now illustrate the use of both the above operators by writing the causal formula for the D flip-flop. Reverse/Causal Temporal Formula for the D-flip-flop in Fig. 2 is

$$(Q = 1) \supset (n \geq 1 \wedge \triangle^n(D = 1 \wedge \uparrow Clk) \wedge \lambda_{n-1}(Q))$$

Explanation: If Q is 1 now, it was set to 1 due to a rising clock and $D = 1$ n instants of time before the present one and has been stable for the last $(n - 1)$ instants of time.

4.2 Modeling Rising Signals and Faults

The reverse operators can be used to model clock signals and stuck-at faults very efficiently. We use the $\triangle$ operator rather than the $\bigcirc$ operator to model rising signals. Our definition of a rising signal is

$$\uparrow X \equiv \triangle(X = 0) \wedge (X = 1)$$

This way we can model signals that make transitions in the present instant of time. Falling signals are similarly modeled as

$$\downarrow X \equiv \triangle(X = 1) \wedge (X = 0)$$

Some authors represent transitions using the $\bigcirc$ operator, as

$$\uparrow X \equiv (X = 0) \wedge \bigcirc(X = 1)$$

This way the signal rises in the next instant of time rather than the present one. If the previous operator is used, then the signal rises in the present instant.

Stuck-at faults can be modeled using λ and $\Box$ operators. For example to model H sa-0, we write

$$\lambda(H = 0) \wedge \Box(H = 0)$$

A line which has a stuck-at fault has the same value in past and future.

4.3 Modeling Circuits with Reverse Temporal Formula

In the previous section, examples were given to show how circuits could be modeled using forward temporal operators. Now, we consider the same examples and model them by specifying reverse temporal formulas for them.

The formulas below are the reverse temporal formulas for the circuit of Fig. 3.

Reverse Temporal Formulas

1. $(Q0 = 0) \supset \triangle(G2 = 0 \wedge \uparrow Clk)$
2. $(Q0 = 1) \supset \triangle(G2 = 1 \wedge \uparrow Clk)$
3. $(Q1 = 0) \supset \triangle(G6 = 0 \wedge \uparrow Clk)$
4. $(Q1 = 1) \supset \triangle(G6 = 1 \wedge \uparrow Clk)$
5. $G6 \supset (G3 \vee G4 \vee G5)$
6. $G5 \supset (Q0 \wedge \neg Q1)$
7. $G4 \supset (D \wedge Q1 \wedge \neg Q0)$
8. $G3 \supset (G1) \wedge Q1 \wedge Q1)$
9. $G2 \supset (D \wedge \neg Q0)$
10. $G1 \supset \neg D$

The first four formulas are for the two D flip-flops in the Fig. 3. We note that they look similar to the forward formulas, except that these formulas are now expressed so that one may reason about the past, rather than predict the future, as in the case of forward formulas. The reverse formulas help us to do causal reasoning. The above set of formulas are written so that we find the cause for an event in the most recent time instant. So they appear as $\triangle$ formulas. For example if $Q0$ is 1, then we can infer that in the previous time instant $G2$ was 1, and the clock rose. If $Q1$ is 0, then in the previous time instant $G6$ was 0, and the clock rose. The combinational gates have formulas which look like the traditional predicate logic formulas. The reason is that they do not have temporal behavior.

We also now model the serial adder, (with reverse temporal operators), that is shown in Fig. 4. The sum and carry output depend on the previous value of the carry output. The previous value can be easily represented using the $\triangle$ operator. The following formulas model the serial adder.

1. $(A \oplus B \oplus \triangle(C_0)) \supset S$
2. $(AB + B\triangle(C_0) + A\triangle(C_0)) \supset C_0$

5. HEURISTICS FOR TEST GENERATION

Since the test generation problem is NP-complete, an optimal polynomial time solution does not exist. Any solution that is obtained in polynomial time is sub-optimal and will depend on heuristics to make

decisions that will (hopefully) yield good average-time solutions. In this section, we give heuristics that can be used for test generation for sequential circuits. The heuristics are given for both parts of the test generation problem, namely, line justification and fault propagation. The heuristics depend on temporal parameters.

We classify the lines in a circuit into

1. State Lines
   (a) Dependent State Lines
   (b) Independent State Lines
2. Sequential Lines
3. Combinational Lines

State lines are outputs of sequential blocks viz. outputs of flip-flops, registers, counters, etc. Dependent state lines are those state lines that can be reached by *other* state lines. Independent state lines are state lines that are not reachable by other state lines. So these state lines are directly controllable by combinational logic blocks.

Sequential lines are all lines reachable by a state line. In the subcircuit influencing these lines—called the cone of influence—there is at least one sequential block.

Lines of the circuit not reachable by any state line are called combinational lines. The cones of influence of these lines do not have any sequential block.

For example in Fig. 3, $Q0$ and $Q1$ are state lines, D, output of $G1$, and CLK are combinational lines. All other lines are sequential. $Q0$ is an independent state line and $Q1$ is a dependent state line.

We also define two "depth" factors (corresponding to the *next* ($\bigcirc$) and *delta* ($\triangle$) temporal operators) that indicate the sequential depth from the primary input side and also in the reverse side, the primary output side.

Next Depth of a line i is defined as

min{next depth of inputs of line i}
+ next exponent of i

Similarly *Delta Depth* of a line i is defined as

min{(delta depth + next exponent)
of all fanout elements}

The next depth of all primary inputs is 0 and the delta depth of all primary outputs is 0.

We use the above classification of lines in developing the following heuristics.

1. If there is *only one* state line to be justified, and if a choice is available, do not solve the $\triangle$ formula that needs the state line to be set to the same value.
   To set a flip-flop to a 1, it may be wrong to choose a $\triangle$ formula that needs the same flip-flop to be set to 1! If a choice is not available, then eventually the flip-flop may be set to 1, using other heuristics, or, may be impossible to set to 1, depending on the circuit structure, in such cases the impossibility should be detectable. For example, in Fig. 3, to set $Q1$ to 1, we have to make $G6$ equal to 1. $G6$ is 1 if *at least* one of $G3$, $G4$, or $G5$ is 1. For $G3$ or $G4$ to be 1, $Q1$ is required to be 1. So in order to set $Q1$ to 1, we require $Q1$ to be 1 in the previous state! This means that we must solve the same $\triangle$ formula again. Instead, we choose to make $G5$ 1, which creates $\triangle$ formula for $Q1 = 0$.
2. If a choice is available, choose to justify through a combinational line.
   This heuristic is very logical, since what can be achieved with one vector is to be preferred to that achievable by a sequence of vectors. In Fig. 5, to set the output of the AND gate to 0, we can set any one or more of its inputs to 0. We choose to set the combinational line to 0, because one pattern is needed to justify the 0 on the combinational line. If we choose the sequential line we will need *at least* two input patterns.
3. If a choice is available, choose a *non-state line*.
   This may help us to find a shorter input vector sequence which sets a desired value to a line. If we choose a state line, it means we have *at least* one more $\triangle$ equation to solve. If we choose a non-state line, it is possible that we may be able to avoid additional $\triangle$ equations, especially if Heuristic 2 is applicable to the non-state line.

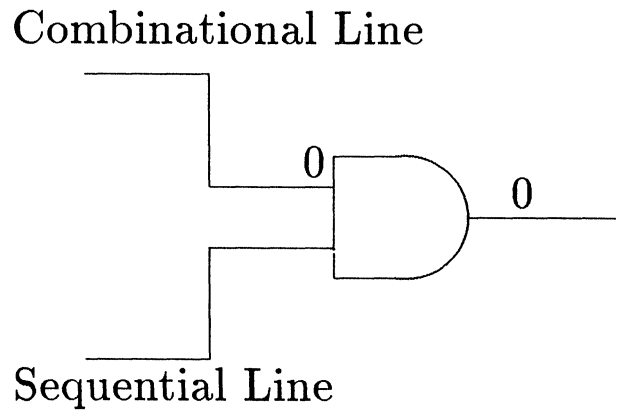

FIGURE 5 Heuristic 2.

4. Hardest to set and easiest to set heuristics are to be used whenever necessary and possible. PODEM [10] also uses similar heuristics. However, here we use next depth instead of "distances" from primary input.

5. If an assignment to a state line is going to lead to a "state"—which is the vector of all state lines—being justified, do not make the assignment, since this corresponds to a loop in the state diagram of the circuit. If a choice exists, choose another assignment or else backtrack to the most recent decision and choose an alternative.
   During the solution of $\triangle$ formulas, more $\triangle$ formulas evolve. The set of $\triangle$ formulas we have to solve at a particular instant of time corresponds to a state of the circuit at that instant of time. The line justification problem is equivalent to the question *What state resulted in the present state?* In other words, what was the previous state that resulted in the present state? So the $\triangle$ formulas give us a state which we should visit in order to solve the line justification problem. This heuristic helps us detect loops in the visiting of states in order to solve a problem. If the loop is inevitable, then we conclude that the problem is unsolvable.

6. If multiple state lines have to be justified, then justify in the sequence:
   (a) Independent state lines.
   (b) State lines, which are not dependents of a state line, whose value is yet to be justified.

7. When we have to propagate the fault effect, we propagate to the line of least delta depth. The line with the least delta depth *may* help us to propagate the effect of the fault to a primary output quickly.

These heuristics are geared towards generating short test sequences. To measure the effectiveness of the heuristics, we have computed the temporal logic parameters for sequential benchmark circuits [5]. Table I lists the various parameters for the benchmark circuits. There is a considerable number of combinational and sequential lines. The heuristics that use the line classifications will help during line justification. Most circuits also have a few independent state lines. Justifying lines through independent state lines is easy.

6. LINE JUSTIFICATION

In this section, we give the solution to the line justification problem. Consider the circuit in Fig. 3, and

TABLE I
Temporal Logic Data for ISCAS-89 Benchmark Circuits

Ckt Name	Num. of Input Lines	Comb. Lines	Seq Lines	Ind. State Lines	Dep. State Lines	Ave. Next Depth	Ave. Delta Depth
s298	3	12	116	3	11	1.0775	1.2606
s349	9	10	171	0	15	0.7500	0.9183
s382	3	14	153	2	19	0.8511	1.7074
s386	7	14	159	0	6	0.2067	0.4805
s510	19	30	207	0	6	0.6872	0.6461
s526	3	13	189	3	18	0.9865	1.7309
s641	35	90	348	0	19	0.1072	0.2604
s820	18	54	272	0	5	0.2719	0.6344
s838	35	38	389	1	31	0.2963	1.7112
s953	16	83	351	3	26	0.2786	1.1123
s1238	14	344	192	12	6	0.0523	0.1643
s1423	17	98	581	2	72	0.2337	1.0186
s1488	8	31	649	0	6	0.3513	0.3994
s5378	35	327	2536	33	146	0.9602	0.3264
s35932	36	1327	15093	0	1728	0.2736	3.0138

TABLE II
Delta Table for $Q1 = 1$

Δ	0	1	2	3	4
D		X	1	0	0
Clk		↑	↑	↑	↑
G1					
G2			1	0	0
G3			0	0	
G4			0	0	
G5		1	0	0	
G6		1	0	0	
Q0		1	0	0	
Q1	1	0	0	X	

its temporal formulas as given in Section 4.3. We find the input sequence that will set $Q1$ to 1, and construct a Δ table for it. In this table, column number n gives the line values in the nth previous time instant. Column 0 specifies the problem. Below we give the Δ table for the problem $Q0 = 1$.

In order to set $Q1$ to 1, according to the reverse temporal formula 2 (Section 4.3), we need $G6 = 1$ and a rising clock in the previous time instant. $G6$ can be set to 1 by setting $Q0$ to 1 and $Q1$ to 1 (using heuristic 1) in the previous time instant. Thus we get column 1 in the above table. We now need to solve two Δ formulas viz. $Q0 = 1$ and $Q1 = 0$. The other columns are similarly constructed. We stop when there are no other Δ formulas to be solved. In column 3, we need $Q0 = 1$. This can be achieved by setting $D = 0$, using heuristic 2. This does not give rise to any more Δ formulas. So we stop. Reading from right to left the rows which correspond to the primary inputs, we get the input sequence needed to set $Q1$ to 1. The sequence is

$$\langle D, Clk\rangle = \langle 0, \uparrow\rangle\langle 0, \uparrow\rangle\langle 1, \uparrow\rangle\langle X, \uparrow\rangle$$

7. TEST GENERATION ALGORITHM AND EXAMPLE

In this section, we describe the test generation algorithm. For generating tests for faults in sequential circuits, we first excite the fault. Then we find line values that are necessary to propagate the effect of the fault to a primary output. We use the ○ formulas to determine the propagating line values. This will create line justification problems spread over several time instants. We create a ○ table similar to the Δ table when we find the propagating line values. We then solve all the line justification problems. The two phase test generation algorithm may be described as

1. Excite and propagate the fault. In the process we build a constraint table containing the line justification problems that need to be solved in different time frames. This table is built using the forward temporal formulas and is called the ○ table (read *next table*).
2. Transform the table into a Δ table, by copying the table backwards. Set the line value for the faulty line to 0 or 1, depending on the type of fault, in all *new* columns of the Δ table. Using the above heuristics solve the Δ formulas until no more Δ formula exists. In case of failure, choose a different path to propagate the fault, by going back to step 1.

Consider for example the fault $G5$ sa-0 in the circuit of Fig. 3. In order to excite the fault, we need $Q0 = 1$ and $Q1 = 0$. To propagate the fault effect, we need $G3 = G4 = 0$ and a rising clock. The effect is propagated to the output $Q1$. The ○ table for this fault is shown below. The numbers in parentheses indicate the faulty value. To find the test we have to justify the state $Q0 = 1$ and $Q1 = 0$. We can do that by constructing the Δ table as shown in Table IV.

The test sequence is

$$\langle D, Clk\rangle = \langle 0, \uparrow\rangle\langle 0, \uparrow\rangle\langle 1, \uparrow\rangle\langle X, \uparrow\rangle$$

Similarly, it is possible to generate test sequences for other faults in the circuit. However, two faults D sa-1 and $G3$ sa-1 are undetectable by ATPG pro-

TABLE III
Next-Table for $G5$ sa-0

○	0	1
D	X	
Clk	↑	
G1		
G2		
G3	0	
G4	0	
G5	1(0)	
G6	1	
Q0	1	1
Q1	0	1(0)

TABLE IV
Delta-Table for $G5$ sa-0

Δ	0	1	2	3
D	X	1	0	0
Clk	↑	↑	↑	↑
G2		1	0	0
G3	0	0		
G4	0	0		
G5	1(0)	(0)	(0)	
G6	1(0)	0	0	
Q0	1	0	0	
Q1	0	0	X	

grams and our algorithm also fails to derive test sequences for these faults.

8. MULTI-LEVEL TEST GENERATION

The same algorithm can be used for test generation at multiple levels of description of the circuit. At higher levels of description, only input/output pin faults are considered. Further, the pin faults are assumed to be stuck-at faults. The modules at higher levels can be described by functional specifications as was done for the serial adder in Section 4.3. Hierarchical specification can also be extracted from the gate level description by using the state transition information. The description thus obtained, uses *assigned state values* for the equivalent state machine. This information can be extracted in a straightforward and mechanical way from the state table. The same heuristics specified in the previous section also apply at this level of description of the circuit. Later, an example of deriving the high level description of a circuit from its gate level description is given.

There have been some attempts in the past to develop high level test generation. Breuer and Friedman have extended D algorithm to work at the level of registers and counters [4]. They have defined a set of functional operations for left shift, right shift, loading the registers, etc. Using these functions as operations, a calculus has been developed for test generation. However, a main drawback is that they have not done any computational feasibility studies, and more importantly, no algorithmic method is used for D-propagation through the constructs of the calculus developed. Levendel and Menon have also extended the classical D-algorithm to work at a higher level of description of circuits [12]. They use an equation method for finding tests for the simple examples demonstrated in the paper. They have specified a high level algorithm for test generation. But the computational feasibility is unknown. Thatte and Abraham have proposed a test generation method for instruction level representations of hardware [25]. The operations of the hardware are mapped to graph transitions, modeling register transfers. The main limitation of this method is its restriction to instruction-level hardware descriptions. Min and Su also have proposed a test generation algorithm based on register transfer language description of hardware [16]. This method is very similar to that of Thatte and Abraham. The difference is that some of the decoding fault classes are separated. For example, the READ and WRITE faults are combined in Thatte and Abraham's method while Min and Su consider them separately.

The method described in this paper is significantly different from all of the above schemes. It uses a new set of heuristics. The algorithm works at *multiple levels*. The above algorithms all work at a high level of description of the hardware.

High level information about circuits can be obtained is a straightforward manner from state tables. Consider the circuit in Fig. 6(a) that is the same one as in Fig. 3, except that it is now depicted as a functional block. The state diagram corresponding to the circuit is shown in Fig. 6(b). The following temporal logic statements describe the functional behavior of the circuit.

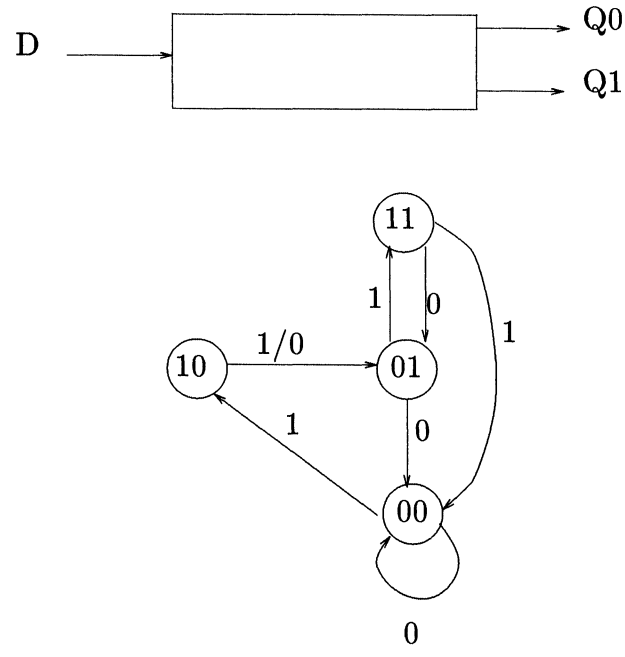

FIGURE 6 Black-box for circuit in Fig. 3 and state diagram.

logic statements describe the functional behavior of the circuit.

1. $D = 0 \wedge \uparrow Clk \supset \bigcirc(Q0 = 0)$
2. $D = 1 \wedge Q0 = 1 \wedge \uparrow Clk \supset \bigcirc(Q0 = 0)$
3. $D = 1 \wedge Q0 = 0 \wedge \uparrow Clk \supset \bigcirc(Q0 = 1)$
4. $D = 0 \wedge Q0 = 0 \wedge \uparrow Clk \supset \bigcirc(Q1 = 0)$
5. $D = 0 \wedge Q0 = 1 \wedge Q1 = 1 \wedge \uparrow Clk \supset \bigcirc(Q1 = 1)$
6. $D = 0 \wedge Q0 = 1 \wedge Q1 = 0 \wedge \uparrow Clk \supset \bigcirc(Q1 = 1)$
7. $D = 1 \wedge Q0 = 0 \wedge Q1 = 0 \wedge \uparrow Clk \supset \bigcirc(Q1 = 0)$
8. $D = 1 \wedge Q0 = 0 \wedge Q1 = 1 \wedge \uparrow Clk \supset \bigcirc(Q1 = 1)$
9. $D = 1 \wedge Q0 = 1 \wedge \uparrow Clk \supset \bigcirc(Q1 = 0)$

The formulas are easily obtainable from the state table description of the circuit. Consider the state table for $Q0$ as given below.

State Table for $Q0$

Q_0Q_1	$D = 0$	$D = 1$
0 0	0	1
0 1	0	1
1 1	0	0
1 0	0	0

The first column gives formula 1. The first 2 rows in the second column give formula 3, and the last two rows in column 2 give formula 2. The procedure of getting the temporal logic statements from the state table is straightforward, and the minimum set of formulas needed is also unique.

Consider the state table for $Q1$, given below.

State Table for $Q1$

Q_0Q_1	$D = 0$	$D = 1$
0 0	0	0
0 1	0	1
1 1	1	0
1 0	1	1

The first two rows in the first column, give formula 4. The third row in column 1, gives formula 5. Formula 6, is obtained from the last row in column 1 of the state table. The first row in column 2, results in formula 7. The second row in column 2, defines formula 8, and finally, the last 2 rows in the second column, result in formula 9. It can be easily seen that we can obtain a high level description of a sequential Moore machine, from the state table description of the machine.

It is evident that these formulas are similar in *form* to the formulas describing the same circuit at the gate level. The test generation at the functional level can therefore be done the same way as it was at gate level. The reverse temporal formulas (at functional level) for the circuit are given below.

1. $Q0 = 0 \supset \Delta(D = 1 \wedge Q0 = 1 \wedge \uparrow Clk)$
2. $Q0 = 0 \supset \Delta(D = 0 \wedge \uparrow Clk)$
3. $Q0 = 1 \supset \Delta(D = 1 \wedge Q0 = 0 \wedge \uparrow Clk)$
4. $Q1 = 0 \supset \Delta(D = 0 \wedge Q0 = 0 \wedge \uparrow Clk)$
5. $Q1 = 0 \supset \Delta(D = 1 \wedge Q0 = 1 \wedge Q1 = 1 \wedge \uparrow Clk)$
6. $Q1 = 0 \supset \Delta(D = 1 \wedge Q0 = 0 \wedge Q1 = 0 \wedge \uparrow Clk)$
7. $Q1 = 1 \supset \Delta(D = 0 \wedge Q0 = 1 \wedge Q1 = 1 \wedge \uparrow Clk)$
8. $Q1 = 1 \supset \Delta(D = 1 \wedge Q0 = 1 \wedge Q1 = 0 \wedge \uparrow Clk)$
9. $Q1 = 1 \supset \Delta(D = 1 \wedge Q0 = 0 \wedge Q1 = 1 \wedge \uparrow Clk)$

We will illustrate the functional level test generation for this circuit. At the functional level, we consider only input/output faults. Consider the fault $Q1$ sa-0. We excite the fault, which gives us two delta formulas to solve, $Q0 = 1$ and $D = 1$ (from reverse temporal formula 8). The faulty line $Q1$ remains at 0. Because of the fault, formulas 7 and 9 cannot be used. These delta formulas, give rise to the delta formula $Q0 = 0$ and $D = 1$ (two time units in the past). Solving until we no longer have any more delta formulas, we get $\langle 0, 0, 1, 1\rangle$ as the test sequence. Table V illustrates the test generation at functional level for the fault $Q0$ sa-0.

Test generation can be done when circuits are described so that some modules are described functionally while some of them are described at the gate level. This kind of description may be useful when test generation is being done for a fault at a particular site. The module containing the fault is represented at the gate level. The other modules can be repre-

TABLE V
Test Generation for $Q0$ sa-0

Δ	0	1	2	3	4
Q1	1(0)	0	0	X	X
Q0		1	0	0	x
D		1	1	0	0

sented at higher levels. Through an example, we show how test generation can be done at mixed level; when some components are described at higher level and some are described at the gate level.

Consider the circuit in Fig. 7 which is constructed by augmenting the circuit in Fig. 6 with a 4-bit shift register and an AND gate. The temporal specification for the circuit is augmented by the following formulas for the lines S and Z.

1. $D = 1 \wedge \uparrow Clk \supset \bigcirc^4(S = 1)$
2. $D = 0 \wedge \uparrow Clk \supset \bigcirc^4(S = 0)$
3. $S = 0 \supset \triangle^4(D = 0 \wedge \uparrow Clk)$
4. $S = 0 \supset \triangle^4(D = 0 \wedge \uparrow Clk)$
5. $Q0 \wedge Q1 \wedge S \supset Z$

The next exponent is 4, because it takes 4 clock pulses for the D input to be observable at S. We will consider the input fault S sa-1 for the AND gate. To excite the fault, we need a 0 on S, which forces that we need a 0 on the input D 4 units of time before the present one. And in order to be able to propagate the effect of the fault to the output Z, we need propagating values on $Q0$ and $Q1$, which are both 1. These give rise to more delta formulas, which are solved to get the complete test sequence. The test sequence for this fault is $\langle 0, 0, 1, 0, 1, X\rangle$. Table VI shows the derivation of the test sequence for the fault S sa-1. Similarly, the test sequence for S sa-0 can be derived, which is $\langle 0, 0, 1, 0, 1, 1, 1, 0, 1, X\rangle$.

As the last example, we consider the serial adder of Fig. 4. The temporal logic formulas are reproduced here again.

1. $(A \oplus B \oplus \triangle(C_0)) \supset S$
2. $(AB + B\triangle(C_0) + A\triangle(C_0)) \supset C_0$

We generate test sequence for the fault S sa-0. To excite the fault, we need a 1 on S, which will enforce that one or all three of A, B and $\triangle(C_0)$ to be 1. We choose to make $A = 1$ and B and $\triangle(C_0)$ to be 0. The requirement of $C_0 = 0$ in the previous time frame can be easily satisfied by making A or B (or both) 0. So one test sequence—(A, B) are the two inputs of the adder—for S sa-0 is $\langle(0, 0), (1, 0)\rangle$. The

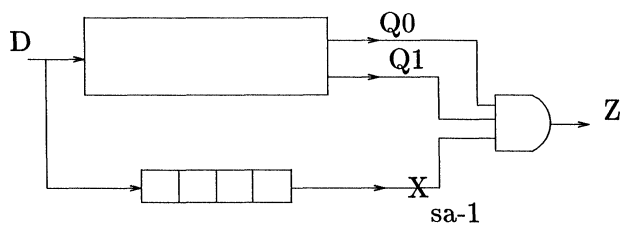

FIGURE 7 A mixed-level circuit with a sa-1 fault.

TABLE VI
Test Generation for S sa-1

Δ	0	1	2	3	4	5
S	0 (1)					
Q0	1	0	1	0	0	X
Q1	1	1	0	0	X	X
D	X	1	0	1	0	0

other test sequences are $\langle(X, 0), (1, 0)\rangle$, $\langle(0, X), (1, 0)\rangle$, $\langle(0, X), (0, 1)\rangle$, $\langle(X, 0), (0, 1)\rangle$, $\langle(0, 0), (0, 1)\rangle$, $\langle(1, 1), (1, 1)\rangle$. Table VII shows the derivation of the test sequence $\langle(0, 0), (1, 0)\rangle$.

A test sequence that can similarly be derived for A sa-1 is $\langle(1, 1), (0, 1)\rangle$, for A sa-0 is $\langle(0, 0), (1, 0)\rangle$. Similar tests work for faults on B. C sa-1 has a test vector $\langle(0, 0)\rangle$, which can be derived from formula 2 above. C sa-0 can be detected by the vector $\langle(1, 1)\rangle$.

At the gate level, the algorithm behaves like a typical time-frame expansion algorithm. The advantages however are, in the modeling aspects. As seen by examples, it is possible to do hierarchical test generation using the temporal logic model. Thus the algorithm is able to *work at multiple levels without any change required in it*. The added advantage is that the various modules of the circuit can be specified at various levels; some of them at the gate level, and others at the functional level. A unified formalism has been presented and used for test generation.

9. CONCLUSION

In this paper, we defined two reverse temporal operators. These operators are useful in reasoning about the past. The finest clock cycle is the unit of time for the temporal operators. Examples were given to illustrate that circuits can be modeled using the temporal operators. Each line has a temporal

TABLE VII
Test Generation for a Full-Adder Fault

Δ	0	1
S	1(0)	
A	1	0
B	1	0
C_o		0

specification. Their utility in line justification was also demonstrated. A test generation algorithm for synchronous sequential circuits based on temporal logic was described. A set of heuristics were given to guide the algorithm during line justification, and fault propagation. The algorithm considers single stuck-at faults. Faults on clock lines are not considered by the algorithm. The basic sequential block at the gate level is a flip-flop. Since temporal logic can model circuits at multiple levels, the test generation algorithm works hierarchically. Examples were given to illustrate hierarchical level test generation.

References

[1] V.D. Agrawal, K.T. Cheng, and P. Agrawal. CONTEST: A Concurrent Test Generator For Sequential Circuits. In *Proc. ACM/IEEE Design Automation Conference,* pages 84–89, 1988.

[2] M.J. Bending. HITEST: A Knowledge-based Test Generation System. *IEEE Design And Test of Computers,* pages 83–92, May 1984.

[3] G.V. Bochman. Hardware Specification With Temporal Logic: An Example. *IEEE Transactions on Computers,* C-31:223–231, March 1982.

[4] M.A. Breuer and A.D. Friedman. Functional Level Primitives in Test Generation. *IEEE Transactions on Computers,* C-29(3):527–535, March 1980.

[5] F. Brglez, D. Bryan, and K. Koźmiński. Combinational Profiles of Sequential Circuits. In *Proc. of the Intl. Symposium on Circuits and Systems,* 1989.

[6] M. Browne, . Clarke, D. Dill, and B. Mishra. Verification of Sequential Circuits using Temporal Logic. *IEEE Transactions on Computers,* pages 1035–1044, December 1986.

[7] P. Camurali and P. Prinetto. Formal Verification of Hardware Correctness: Introduction and Survey of Current Research. *IEEE Computer,* pages 8–19, July 1988.

[8] W.-T. Cheng. The back algorithm for sequential test generation. In *Proc. of IEEE Int. Conf. on Computer Design,* pages 66–69, October 1988.

[9] H. Fujiwara and T. Shimono. On the Acceleration of Test Generation Algorithms. *IEEE Transactions on Computers,* pages 1137–1144, December 1983.

[10] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Circuits. *IEEE Transactions on Computers,* pages 215–222, March 1981.

[11] O.H. Ibarra and S.K. Sahani. Polynomially Complete Fault Detection Problems. *IEEE Transactions on Computers,* pages 242–249, 1975.

[12] Y.H. Levendel and P.R. Menon. Test Generation Algorithms for Computer Hardware Description Languages. *IEEE Transactions on Computers,* C-31(7):577–588, July 1982.

[13] H.-K.T. Ma, S. Devadas, A.R. Newton, and A. Sangiovanni-Vincentilli. Test generation for sequential circuits. *IEEE Transactions on Computer Aided Design,* 7(10):1081–1093, October 1988.

[14] R.A. Marlett. An Effective Test Generation System for Sequential Circuits. In *Proc. ACM/IEEE Design Automation Conference,* pages 250–256, 1986.

[15] A. Miczo. *Digital Logic Testing and Simulation.* Harper Row, 1986.

[16] Y. Min and S.Y.H. Su. Testing Functional Faults. In *Proc. ACM/IEEE Design Automation Conference,* pages 384–392, 1982.

[17] B. Moszkowski. A Temporal Logic for Multilevel Reasoning About Hardware. *IEEE Computer,* pages 10–19, 1985.

[18] J.F. Poage. Derivation of Optimum Tests to Detect Faults in Combinational Circuits. In *Proc. Symposium on Mathematical Theory of Automata,* pages 483–528, 1963.

[19] G. Putzolu and J.P. Roth. A Heuristic Algorithm for Testing Asynchronous Circuits. *IEEE Transactions on Computers,* pages 643–647, June 1971.

[20] J.P. Roth. Diagnosis of Automata Failures. *IBM Journal of Research and Development,* pages 278–291, July 1968.

[21] M.H. Schultz, T. Trischler, and T. Starfet. SOCRATES: A Highly Efficient Automatic Test Pattern Generation System. In *Proc. ACM/IEEE Design Automation Conference,* pages 1016–1025, 1987.

[22] M.H. Shirley. Generating Tests by Exploiting Designed Behavior. In *Proc. of AAAI Conf.,* pages 884–890, 1986.

[23] N. Singh. SATURN: An Automatic Test Generation System for Digital Circuits. In *Proc. of AAAI Conf.,* pages 778–783, 1986.

[24] H.S. Stone and P. Sipla. The Average Complexity of Depth First Search with Backtracking Cutoff. *IBM Journal of Research and Development,* pages 242–258, May 1986.

[25] S.M. Thatte and J. Abraham. Test Generation for Microprocessors. *IEEE Transactions on Computers,* C-29:429–441, June 1980.

[26] P. Wolper. Temporal Logic can be more Expressive. In *Proc. 22nd Annual Symposium on Foundation of Computer Science,* pages 340–348, 1981.

Biographies

ANAND V. HUDLI received B.E. degree in Electronics and Communications, from University of Mysore, India, in 1982, M.Tech in Computer Science from Indian Institute of Technology, Bombay, in 1985, and Ph.D. in Computer Science from University of Nebraska, in 1989. Since 1989, he is an Assistant Professor in the Department of Computer Science at Purdue School of Science, Indianapolis. His current research interests include Artificial Intelligence, Programming Languages, and Databases.

RAGHU V. HUDLI received B.E. degree in Electronics and Communications, from University of Mysore, India, in 1984, M.E. in Electrical Communications Engineering, from Indian Institute of Science, in 1986, and Ph.D. in Computer Science from University of Nebraska, in 1990. Since 1990, he is at IBM Corp. His research interests include all aspects of VLSI design automation, theory of algorithms, distributed and parallel computing.