Copyright © 1974 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 53, No. 8, October 1974
Printed in U.S.A.

LAMP:

Automatic Test Generation for Asynchronous Digital Circuits

By S. G. CHAPPELL

(Manuscript received February 28, 1974)

An automatic test generation system has been developed to detect faults in combinational and sequential circuits. The circuit model treats logic circuits as interconnections of unit- and zero-time-delay gates. A series of time-dependent Boolean equations are derived from the logic network (starting from the network inputs) in terms of sequences of signals (input vectors) on the circuit input leads. These equations account for the effect of specific circuit faults. Many tests, each consisting of a sequence of input signals (input vectors), are needed to detect all single faults in a circuit. Tests are generated from the time-dependent equations using two different strategies: (i) a maximum-cover approach to detect a large number of faults quickly by generating tests for the faults on the circuit-input leads. The fault-detection level achieved by the maximum-cover tests is then evaluated using fault simulation; (ii) tests for individual faults not detected by the maximum-cover approach. ATG has been implemented on the IBM 360, Model 67, and IBM 370, Model 168, computers.

I. INTRODUCTION

The automatic test generation system (ATG) was designed to provide fault-detection tests for single stuck-at faults in combinational and sequential circuits. Since this problem has essentially been solved for combinational circuits,[1–3] this paper concentrates on aspects of automatic test generation for sequential circuits.

The ATG algorithms presented attempt to account for actual circuit behavior as closely as possible. Hence, it is necessary to create a computer model of the actual gates in the logic circuit. The circuit description used by ATG will utilize a unit/zero time-delay model, where a gate can assume one of three values: logical 0, logical 1, and don't-know X. This model has been widely used for logic-circuit simulation.[4,5] Because the test-generation algorithms described use the same model as many simulators, there are parallels between the simulation and test-generation techniques. These result from the effort to increase the accuracy of test generation to achieve the accuracy of current simulation techniques.

The major drawback of previous algorithms[6–8] for test generation for sequential circuits is the lack of a satisfactory model for the sequential circuit. Previous algorithms use either the Huffman model or an iterative combinational circuit model for sequential circuits. While these models are mathematically convenient, they are hardly accurate representations of real logic circuits. The system to be presented here has the following features:

(*i*) Requires no identification of feedback lines.
(*ii*) Allows gates to have time delays associated with their response to input stimuli.
(*iii*) Resolves races on flip-flops and detects circuit oscillations.
(*iv*) Assumes that an unknown circuit state corresponds to each gate having the unknown value X. [The X corresponds to value 3 in Ref. (5).]
(*v*) Generates a test for a single stuck-at or open-gate input fault, if it exists. The test is guaranteed to detect the fault (subject to the circuit-model assumptions).
(*vi*) Handles gate-level models of sequential circuits containing up to approximately 1000 gates.

For economy, the system allows test generation using two strategies. The first strategy (maximum cover) generates a set of tests designed to detect a large number of single faults without ever explicitly considering a specific fault. The second strategy generates tests for specified single faults. To allow rapid evaluation of the set of tests derived by the first strategy, a fault simulator is needed to simulate all single stuck-at faults. This simulator identifies the undetected set of faults that must be considered by the strategy-2 test generator. To keep the computation time reasonable, a user-specified parameter sets the maximum sequence length that will be considered by the system. The use and operation of the system is shown in the flow diagram in Fig. 1.

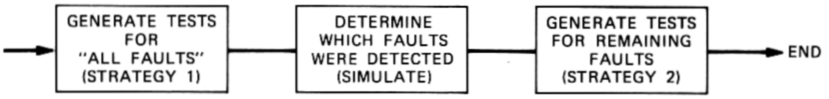

GENERATE TESTS FOR "ALL FAULTS" (STRATEGY 1)
DETERMINE WHICH FAULTS WERE DETECTED (SIMULATE)
GENERATE TESTS FOR REMAINING FAULTS (STRATEGY 2)
END

Fig. 1—Overall strategy.

II. MATHEMATICAL BASIS

This section builds the framework for the remainder of the paper. The behavior of some gate G will be described by two equations G^0 and G^1, where G^0 (G^1) describes the input conditions that set gate $G = 0$, ($G = 1$). The gates can assume one of three logical values: logical 0, logical 1, or the don't-know value X. Equations G^0 and G^1, however, are strictly Boolean equations in that the constituent variables of G^0 and G^1 can assume only values of 0 or 1. Similarly, G^0 and G^1 are Boolean variables.

2.1 Definitions

The following definitions are used in this discussion.

(*i*) Input vector: A string of n logical values (0, 1, and X, where X is a don't-know value) that applied to the n corresponding input leads of a circuit. The effect of these values is allowed to propagate through the circuit before the next input vector is applied to the circuit.

(*ii*) Test: A series of input vectors applied in a specific order to the circuit inputs. A test is also sometimes called a sequence. The first vector in each test assumes the circuit is in a completely unknown state. The nth vector ($N > 1$) assumes the state produced by the preceding $N - 1$ input vectors. Many tests may be required to detect all of the detectable faults in a logic circuit. Notice that it is not necessary to allow the circuit to stabilize between input vectors.

(*iii*) Sequence length: The number of input vectors in a test.

(*iv*) Input variables: Associated with each circuit input lead a are two binary input variables a^0 and a^1. The variables a^0 and a^1 can each take on Boolean values 0 and 1 (or "false" and "true"). Together, a^0 and a^1 define the logical value (0, 1, or X) of input lead a as shown in Table I. Hence, if $a^0 = 1$ (disallowing $a^0 = a^1 = 1$), then the logical value of lead a is 0. If $a^1 = 1$, then the logical value of lead a is 1. If neither $a^0 = 1$

Table I — Definition of a^0 and a^1

a^1 Value	a^0 Value	Lead a Logical Value
0	0	X
0	1	0
1	0	1
1	1	Impossible

nor $a^1 = 1$, then the value of input lead a is unknown or X. It is clearly impossible for input lead a to simultaneously have a logical value of 1 and 0. Therefore, $a^0 = a^1 = 1$ is an impossible situation. The variables a^0 and a^1 will often be used as an ordered pair (a^1, a^0). For example, (1, 0) represents the gate value of logical 1.

(*v*) Sequence of input variables: Let a^0i (a^1i), $i = 1, 2, \cdots$, represent the fact that $a = 0$ $(a = 1)$ during the ith input vector of a sequence. If no subscript is used (e.g., a^0 is written), then it is assumed that a represents the first input vector.

(*vi*) Notation: As is traditional, "+" represents logical OR, and "·" represents logical AND. The symbol "–" will be used to represent NOT or complement.

(*vii*) Unknown state: If the circuit is in an unknown state, it is assumed that each gate in the circuit is assigned the unknown output value X.

2.2 *Properties of the equations*

Some of the properties of input variables a^0 and a^1 are described in this section. Consider a circuit consisting of a two-input AND gate c with inputs a and b. Input leads a and b have associated with them (a^1, a^0) and (b^1, b^0), respectively. The problem is to compute (c^1, c^0). The usual truth table for AND is shown below.

			a	
	AND	0	1	X
	0	0	0	0
b	1	0	1	X
	X	0	X	X

Translating this to the ordered-pair notation, we have:

	AND	(a^1, a^0) (0, 1)	(1, 0)	(0, 0)
	(0, 1)	(0, 1)	(0, 1)	(0, 1)
(b^1, b^0)	(1, 0)	(0, 1)	(1, 0)	(0, 0)
	(0, 0)	(0, 1)	(0, 0)	(0, 0)

Examining these ordered pairs, one finds that $c^0 = 1$ if and only if (iff) $a^0 = 1$ or $b^0 = 1$. Similarly, $c^1 = 1$ iff both $a^1 = 1$ and $b^1 = 1$. Hence, the following relations hold for a two-input AND gate c with inputs a and b:

$$c^0 = a^0 + b^0$$
$$c^1 = a^1 \cdot b^1$$

or

$$(c^1, c^0) = (a^1 \cdot b^1, a^0 + b^0). \tag{1}$$

It is important to note that c^0 is not necessarily the complement of c^1. For example, if $(a^1, a^0) = (0, 0)$ and $(b^1, b^0) = (1, 0)$, then $(c^1, c^0) = (0 \cdot 1, 0 + 0) = (0, 0)$.

A similar set of relations can be derived for a two-input OR gate f with inputs d and e.

$$(f^1, f^0) = (d^1 + e^1, d^0 \cdot e^0). \tag{2}$$

The interpretation of this is that $f = 1$ if either $d = 1$ or $e = 1$ or both. Similarly, $f = 0$ if both $d = 0$ and $e = 0$.

Another relation can be derived for the NOT gate (or inverter) h with input g as follows. Note that the complement of X is still X.

$$(h^1, h^0) = (g^0, g^1). \tag{3}$$

For later use, the relations governing the NAND gate are also presented here. The NAND gate is simply an AND gate followed by a NOT gate. Hence, we have, for a two-input NAND gate w with input y and z:

$$(w^1, w^0) = (y^0 + z^0, y^1 \cdot z^1). \tag{4}$$

The above definitions have been presented for two-input gates. However, since the functions AND and OR are associative, the equations for an N-input gate can easily be derived. For example, for a three-input NAND gate w with inputs p, y, and z, we have:

$$(w^1, w^0) = (p^0 + y^0 + z^0, p^1 \cdot y^1 \cdot z^1). \tag{5}$$

Notice that since a^0 and a^1 are binary variables, they obey all the laws of Boolean algebra. However, the interactions of a^0 and a^1 are not so obvious and are of interest here.

It is significant that in the algorithms presented for computing the output equations for a gate [eqs. (1) through (5)], we have never produced a $\bar{G}^0$, $\bar{G}^1$, or $\bar{G}$ expression, where G is any gate in the circuit. This has occurred for two reasons. First, because gate G can assume three values $\bar{G}$ is not particularly useful. For example, if $G = 1$, then $\bar{G} = 0 + X$. Second, as a practical matter, the computation of $\bar{G}^0$ or $\bar{G}^1$, given G^0 or G^1, is quite time consuming if both input and output are to be in sum-of-products form.

2.3 Some identities and nonidentities

After the operations AND, OR, and NOT have been defined, further properties can be investigated. By simple examination of the definitions for AND, OR, and NOT, the following identities are obvious. Let a represent any gate in the circuit. For ease of understanding, the corresponding theorem of Boolean algebra is written on the same line as the identities but enclosed in brackets.

(*i*) $(0, 1)\cdot(a^1, a^0) = (0, 1) \quad [0\cdot a = 0]$.

(*ii*) $(1, 0)\cdot(a^1, a^0) = (a^1, a^0) \quad [1\cdot a = a]$.

(*iii*) $(a^1, a^0)\cdot(a^1, a^0) = (a^1, a^0) \quad [a\cdot a = a]$.

(*iv*) $(1, 0) + (a^1, a^0) = (1, 0) \quad [1 + a = 1]$.

(*v*) $(0, 1) + (a^1, a^0) = (a^1, a^0) \quad [0 + a = a]$.

(*vi*) $(a^1, a^0) + (a^1, a^0) = (a^1, a^0) \quad [a + a = a]$.

(*vii*) $(a^1, a^0)\cdot(b^1, b^0) = (b^1, b^0)\cdot(a^1, a^0)$ [Commutative].

Proof: $(a^1, a^0)\cdot(b^1, b^0) = (a^1\cdot b^1, a^0 + b^0)$
$= (b^1\cdot a^1, b^0 + a^0) = (b^1, b^0)\cdot(a^1, a^0)$ QED.

Similarly,

(*viii*) $(a^1, a^0) + (b^1, b^0) = (b^1, b^0) + (a^1, a^0)$ [Commutative].

(*ix*) $[(a^1, a^0)\cdot(b^1, b^0)]\cdot(c^1, c^0) = (a^1, a^0)\cdot[(b^1, b^0)\cdot(c^1, c^0)]$
[Associative].

Proof: $[(a^1, a^0)\cdot(b^1, b^0)]\cdot(c^1, c^0)$
$= ([a^1\cdot b^1]\cdot c^1, [a^0 + b^0] + c^0)$
$= (a^1\cdot[b^1\cdot c^1], a^0 + [b^0 + c^0])$
$= (a^1, a^0)\cdot[(b^1, b^0)\cdot(c^1, c^0)]$ QED.

Similarly,

(*x*) $[(a^1, a^0) + (b^1, b^0)] + (c^1, c^0)$
$= (a^1, a^0) + [(b^1, b^0) + (c^1, c^0)]$ [Associative].

(*xi*) $(a^1, a^0)\cdot(b^1, b^0) + (a^1, a^0) = (a^1, a^0) \quad [ab + a = a]$.

Proof: $(a^1, a^0) \cdot (b^1, b^0) + (a^1, a^0)$
$= (a^1 \cdot b^1, a^0 + b^0) + (a^1, a^0)$
$= (a^1 \cdot b^1 + a^1, [a^0 + b^0] \cdot a^0)$
$= (a^1, a^0 + a^0 b^0) = (a^1, a^0)$ QED.

(*xii*) $a^0 \cdot a^1 = a^1 \cdot a^0 = 0$.

This is obviously true if a is a circuit input lead. Because the computation of the equations proceeds from gate inputs to gate outputs, this result can be shown inductively. For any valid circuit state (gates have logical values 0, 1, or X), the theorem is true. It is also true for input leads, as mentioned earlier. Then, by examination of eqs. (1) through (5) above, we see that the relationship is preserved when the new gate output equations are computed. Hence, by induction, it follows that the relationship holds for every gate in the circuit.

(*xiii*) $\overline{[(a^1, a^0) \cdot (b^1, b^0)]} = (a^0, a^1) + (b^0, b^1) \quad [\overline{(a \cdot b)} = \bar{a} + \bar{b}]$.

Proof: $\overline{[(a^1, a^0) \cdot (b^1, b^0)]} = \overline{(a^1 \cdot b^1, a^0 + b^0)}$
$= (a^0 + b^0, a^1 \cdot b^1) = (a^0, a^1) + (b^0, b^1)$ QED.

(*xiv*) $\overline{[(a^1, a^0) + (b^1, b^0)]} = (a^0, a^1) \cdot (b^0, b^1) [\overline{(a + b)} = \bar{a} \cdot \bar{b}]$.

Proof: $\overline{[(a^1, a^0) + (b^1, b^0)]} = \overline{(a^1 + b^1, a^0 \cdot b^0)}$
$= (a^0 \cdot b^0, a^1 + b^1) = (a^0, a^1) \cdot (b^0, b^1)$ QED.

Again, it is clear that identities (*xiii*) and (*xiv*) can easily be extended to several variables (e.g., $\overline{(a \cdot b \cdot c)} = \bar{a} + \bar{b} + \bar{c}$). These are simply DeMorgan's theorems.

The identities above simply follow the Boolean algebra. The following set of nonidentities results primarily from the three values used to model the gate behavior.

(*i*) $a^0 + a^1 \neq 1$.

Proof by example: $(a^1, a^0) = (0, 0)$.
Clearly, $0 \cdot 0 \neq 1$.
This is not unexpected since the only relation between a^0 and a^1 is that $a^0 \cdot a^1 = 0$.

(*ii*) $(a^1, a^0) \cdot (b^1, b^0) + (a^1, a^0) \cdot (b^0, b^1) \neq (a^1, a^0) \quad [a \cdot b + a \cdot \bar{b} \neq a]$.

(*iii*) $a \cdot c + \bar{a} \cdot b \cdot c \neq b \cdot c + a \cdot c$.

(*iv*) $a \cdot b + \bar{a} \cdot c + b \cdot c \neq a \cdot b + \bar{a} \cdot c$.

Nonidentities (*ii*), (*iii*), and (*iv*) are easily proved by examining the truth tables, where the variables are allowed to assume three values: logical 0, logical 1, and the don't-know value X.

It is interesting to note that if we required the circuit input leads to have only values 0 and 1, the system presented here would reduce to Boolean algebra with $a^0 = \bar{a}^1$ and $a^1 = \bar{a}^0$. This is a reasonable restriction, since we could always require that any X values generated for the input leads be arbitrarily set to logical 0 or 1. However, it would then be necessary to treat input leads differently from other gates in the circuit, since it is clearly not possible to force every gate in the circuit to a known value (logical 0 or 1). Hence, the generality of allowing circuit input leads to assume the value X is retained in this paper and all gates are treated identically.

III. EQUATION DERIVATION FOR LOGIC NETWORKS

The operation of the ATG has two well-defined steps. The first step is to derive a set of relations (equations) that represent the behavior of the logic circuit. The second step is to derive a set of tests for the circuit based on the equations derived in the first step. In this section the equation-derivation process is described.

The equation derivation process essentially reduces the behavior of a logic circuit to a series of equations. Hence, this reduction process is quite critical. These equations must reflect the true circuit behavior as closely as is possible (or economical). This means that the time delay of gates must be accounted for during the equation-generation process. The equation-generation process will first be presented using a fault-free, unit/zero, time-delay model for each gate. The model will then be extended to account for single stuck-at-one and stuck-at-zero faults.

The method is essentially a dynamic equation-generation process that determines exactly those input sequences that will force each gate to a logical 0 or 1 at each instant of time. The equation-derivation process begins with the circuit inputs and continues through the circuit until the equations are stable; that is, until the output equation on each gate is consistent with the input equation on that gate. The equations are derived in terms of circuit input variables only; no feedback lines need be identified. The input variables may change several times before the circuit finally reaches a stable state.

Since the objective is to generate tests to detect faults in a circuit, the result of this process will be a series of logical values 0, 1, and X (don't know) to be applied to the input leads of the circuit. The output of the circuit will then be observed to determine which classical faults have been detected. That is, the output of the real (perhaps faulty) circuit will be compared to the expected result to determine if the real circuit is performing correctly.

3.1 Fault-free-equation derivation

In this section, the problem of generating equations that represent the behavior of the fault-free circuit is discussed. Because certain simplifications are possible, the equation-derivation process for combinational circuits is discussed first. This is followed by the equation-derivation process for sequential circuits.

3.1.1 Equation derivation for combinational circuits

The derivation of the fault-free equations will be considered here. Consider the NAND gate G shown in Fig. 2. If we assume both inputs to the gate are circuit inputs or other gate outputs, then we have:

$$G^0 = A^1 \cdot B^1$$
$$G^1 = A^0 + B^0.$$

Equation G^0 denotes exactly those input conditions to gate G that force (or set) gate G to logical 0. Implicit in this equation is the unit-delay assumption. If inputs $A = 1$ and $B = 1$ are applied at time t, then the output of G is forced to logical 0 at time $t + 1$. A similar situation exists for G^1. Either $A = 0$ or $B = 0$ (or both) applied at time t to the inputs of G forces its output to be logical 1 at time $t + 1$. This is similar to eqs. (1) through (5) in the previous section, except that the element of time has been added. For most gates, the output of the gate responds to the input stimuli one unit of time later. The gates with one unit of delay are "real" gates, e.g., those containing an active semiconductor device.

In some logic families it is possible to directly connect two (or more) gate output leads together. This connection (called a TIC here, for tied collector) performs a logic function. If the ground level is logical 0, then the TIC function is AND. If the ground level is logical 1, then the TIC function is OR. The TICs may be considered zero-delay gates except for the wire-propagation delay, which is not considered here.

If computation begins at the circuit inputs, which are assumed to be applied at time t, the output of each gate driven by a primary input is reevaluated and the new equation is assigned to the gate output at time $t + 1$. Every gate whose input equation changed at time $t + 1$ is reevaluated and its new output is assigned at time $t + 2$. This process continues until the computation reaches the circuit output

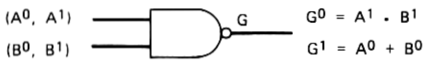


Fig. 2—Equations based on input equations.

gates. At any time $t + i$ after the processing begins, the equations denote those input conditions that force each gate to logical 0 and 1 at i gate delays after application of the vector. In particular, when the circuit has settled to a stable state, the input values that set each gate to logical 0 or 1 are specified. Notice that no assumptions have been made that would preclude the application of this argument to sequential circuits.

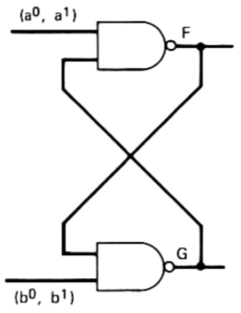
The similarity between this procedure and the actual propagation of electrical signals through the circuit should be evident. In both cases, the input stimuli are applied to the circuit inputs and are allowed to propagate through the circuit.

3.1.2 Equation derivation for sequential circuits

Three points are significant in the discussion of equation derivation for combinational circuits: (*i*) the process assumes all gates have either unit or zero delay, (*ii*) the process starts from the circuit inputs and proceeds through the circuit much as a signal would propagate through the circuit, and (*iii*) the equations define, for each time $t + i$, exactly those input conditions that cause each gate in the circuit to be forced to logical 0 and 1 at that time from the specified initial state. Again, there are no assumptions that limit this technique to combinational circuits.

The primary addition, which must be made to allow the same algorithm to be applied to sequential circuits, is some provision for deciding when to stop the computation. For combinational circuitry, the computation stops when the circuit outputs are reached. However, this is not satisfactory for sequential circuits. The equation derivation yields G^0 and G^1 for each gate G for each time $t + i$. If both G^0 and G^1 at time $t + i$ are equal to G^0 and G^1 at time $t + i + 1$, the gate is in a stable state. Otherwise, each gate driven by gate G must be reevaluated since G changed values (output equations). A detailed flow chart of the equation computation process will be presented later.

Let $a.i$ represent the value of circuit input lead a during the ith vector of the test (or sequence). Similarly, a^0i (a^1i) means make input lead a logical 0 (logical 1) during the ith input vector of the sequence. The first vector in each sequence is number one. If the sequence number is missing, then it is assumed to represent the first vector of the sequence. An example of the application of this algorithm is shown in Fig. 3 where the equations for a flip-flop are calculated. Time runs down the page. The flip-flop is assumed to start from the unknown state since $F^0 = F^1 = G^0 = G^1 = 0$. The inputs are assumed to be



F^0	F^1	TIME
0	0	t
0	a^0	t + 1
$a^1 \cdot b^0$	a^0	t + 2
$a^1 \cdot b^0$	$a^0 + a^0 \cdot b^1 = a^0$	t + 3

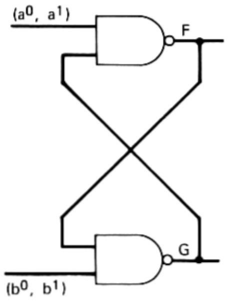
G^0	G^1	TIME
0	0	t
0	b^0	t + 1
$a^0 \cdot b^1$	b^0	t + 2
$a^0 \cdot b^1$	$b^0 + a^1 \cdot b^0 = b^0$	t + 3

Fig. 3—Equations for NAND flip-flop from an unknown state.

applied at time t. At time $t + 1$, only F^1 and G^1 changed values so at time $t + 2$ only G^0 and F^0 are calculated. At time $t + 3$, none of the new output equations changed, so the circuit is stable and computation stops.

A similar computation can be carried out if the circuit is in some known initial state. This is illustrated in Fig. 4 where the circuit is initially set at $F = 0$ ($F^0 = 1$, $F^1 = 0$) and $G = 1$ ($G^0 = 0$, $G^1 = 1$).

The above procedure finds the next state function for a combinational or sequential circuit. That is, given a circuit state (possibly unknown), we can find all possible next states resulting from the application of one input vector.



F^0	F^1	TIME
1	0	t
a^1	a^0	t + 1
a^1	a^0	t + 2
a^1	$a^0 + a^0 \cdot b^1 = a^0$	t + 3

G^0	G^1	TIME
0	1	t
0	1	t + 1
$a^0 \cdot b^1$	$a^1 + b^0$	t + 2
$a^0 \cdot b^1$	$a^1 + b^0$	t + 3

Fig. 4—Equations from $F = 0$, $G = 1$ state.

Clearly, the problem is to determine the state of the circuit as the result of each possible sequence of vectors. If this can be done, then there is no need to select a particular next state since they are all considered simultaneously. A method for doing this is described in the next section.

3.1.3 Sequence derivation for sequential circuits

The algorithm for sequence derivation is based only on the input behavior of the circuit. There is no need to consider any feedback variables. Again, the derivation assumes either unit- or zero-delay gates.

This algorithm is based on the explanation presented in the previous section. The derivation proceeds as follows for a sequence of length M.

(*i*) To the circuit inputs $[a, b, c, \cdots] = I$ apply the variables $[a^*1, b^*1, c^*1, \cdots] = I.1$ (a^* means apply a^1 or a^0) and derive the equations for the circuit (starting from any initial state).

(*ii*) Let $j = 1$.

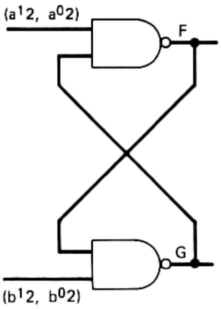
(*iii*) From the circuit "state," as defined by the application of the input vector of variables $I.j = [a^*j, b^*j, c^*j, \cdots]$, apply the input vector of variables $I.j + 1$ and propagate these variables through the circuit, i.e., derive the "equations" for the circuit in terms of $I.j + 1$ and $I.k$ for all $k \leq j$. The effect of $I.j$ need not stabilize before applying $I.j + 1$.

(*iv*) If $j < M$, then let $j = j + 1$ and go to step (*iii*). Otherwise, exit.

This procedure models the behavior of a logic circuit. The input stimuli (variables) are applied to the inputs of the circuit and allowed to propagate through the circuit. The input vector $I.1$ assumes the circuit is in some initial state, which is probably unknown. Input vector $I.2$ produces equations from the initial state produced by $I.1$. In general, the vector $I.j$ starts from the state produced by all $I.k$ where $k < j$.

After application of $I.j$, the effect on the circuit of any sequence of j vectors is known. This is obvious since we have already shown that the application of $I.1$ from any state produces the equations G^0 and G^1 for every gate in the circuit as a result of $I.1$. This extension makes the initial state for $I.j$, $j \geqq 2$, a function of all $j - 1$ vectors.

An application of this algorithm is shown in Figs. 3 and 5 for the NAND flip-flop. Here $I.j = (a^*j, b^*j)$ and the sequence derivation is carried out for sequences of length 2 or less. Figure 3 represents sequences of length 1. Figure 5 represents sequences of length 2. The



F^0	F^1	TIME
$a^11 \cdot b^01$	a^01	t + 3
~~$a^12 \cdot b^01$~~	$a^01 \cdot b^11 + a^02$	t + 4
$a^11 \cdot a^12 \cdot b^01 + a^12 \cdot b^01 \cdot b^02$		RACE
$a^11 \cdot a^12 \cdot b^01 + a^12 \cdot b^02$	$a^01 \cdot b^11 \cdot b^12 + a^02$	t + 5

G^0	G^1	TIME
$a^01 \cdot b^11$	b^01	t + 3
~~$a^01 \cdot b^12$~~	$a^11 \cdot b^01 + b^02$	t + 4
$a^01 \cdot b^11 \cdot b^12 + a^01 \cdot a^02 \cdot b^12$		RACE
$a^01 \cdot b^11 \cdot b^12 + a^02 \cdot b^12$	$a^11 \cdot a^12 \cdot b^01 + b^02$	t + 5

Fig. 5—Result of second vector of sequence.

computation for sequences of length 2 in Fig. 5 begins from the final state of the computation for sequences of length 1 shown in Fig. 3. To illustrate the interpretation of the equations, consider the final state of $G^0 = a^02 \cdot b^12 + a^01 \cdot b^11 \cdot b^12$ shown in Fig. 5. This means that the sequence of length 1, $a = 0$ and $b = 1$ (for $a^02 \cdot b^12$), or the sequence of length 2, $a = 0$ and $b = 1$ followed by $a = X$ and $b = 1$ (for $a^01 \cdot b^11 \cdot b^12$), will set gate G to logical 0.

3.1.4 Equation race analysis

A race occurs on the simple two-NAND-gate flip-flop shown in Fig. 3 when the output state of the flip-flop is unpredictable from the input conditions. Under these circumstances, the outputs of the flip-flop must be set to the unknown value X. Let us examine $F^0 = b^01 \cdot a^12$ and $G^0 = a^01 \cdot b^12$ at time $t + 4$ in Fig. 5. Since $F^0 \cdot G^0 \neq 0$, then both $b^01 \cdot a^12$ and $a^01 \cdot b^12$ could be simultaneously applied to the circuit inputs producing the sequence $a^01 \cdot b^01 \cdot a^12 \cdot b^12$. This represents the application to the flip-flop of the sequence $a = 0$ and $b = 0$ followed by $a = 1$ and $b = 1$. This produces the race state (unpredictable output conditions) for the NAND flip-flop and must therefore be eliminated. The race state for our example is $a = b = F = G = 1$ at some time t. If a race occurs, the values computed and saved for time $t + 1$ are $F = G = 0$. In addition, when unknown states are allowed, a race is also declared if $F = 0$ and $G = X$ or $F = X$ and $G = 0$ at the same instant of time.[5] This implies that when $F = 0$, G cannot be 0 or X. Thus, to eliminate races, it is necessary to demand that if $F = 0$ then $G = 1$ and, similarly, if $G = 0$ then $F = 1$ at the same instant of time. This is accomplished in our example by forming the new equation F^0n at time $t+4$ as $F^0n(t+4) = F^0(t+4)\cdot G^1(t+4) = a^11\cdot a^12\cdot b^01 + a^12\cdot b^01\cdot b^02$. Similarly, $G^0n(t+4) = a^01\cdot a^02\cdot b^12 + a^01\cdot b^12\cdot b^11$. This process is called race analysis since it prevents the equations from causing simple flip-flops (basically, two cross-coupled NAND or NOR gates) to race.

Race analysis must be performed at time t if both F^0 and G^1 changed at time t, where F and G are the two gates in a simple flip-flop. While this result is shown here for the NAND flip-flop, the proof can easily be extended to NOR flip-flops. The primary difference is that F^1 and G^1 must be modified for NOR flip-flops while F^0 and G^0 must be modified for NAND flip-flops.

(*i*) If F^0 does not change, then gate F cannot change to logical 0 at time t; therefore, there can be no race.

(*ii*) If G^1 (see Fig. 5) does not change at time t, then $F^0(t)$ was formed by ANDing together $G^1(t-1)$ and $a^1(t-1)$. That is, $F^0(t) = G^1(t-1)\cdot a^1(t-1)$. But $G^1(t-1) = G^1(t)$ by assumption. Race analysis would form

$$\begin{aligned} F^0n(t) &= F^0(t)\cdot G^1(t) = F^0(t)\cdot G^1(t-1) \\ &= a^1(t-1)\cdot G^1(t-1)\cdot G^1(t-1) \\ &\qquad = a^1(t-1)\cdot G^1(t-1) = F^0(t). \end{aligned}$$

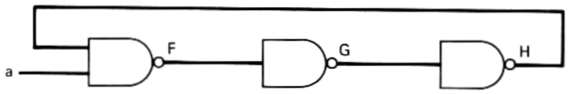
Therefore, the new F^0n resulting from race analysis is the same as the original F^0. Then, there can be no race.

Earlier it was shown that $F^0\cdot F^1 = 0$ for any gate F at any time. It is easily seen that race analysis does not destroy this property since, if $F^0(t)\cdot F^1(t) = 0$ and $F^0n(t) = F^0(t)\cdot G^1(t)$, then $F^0n(t)\cdot F^1(t) = F^0(t)\cdot G^1(t)\cdot F^1(t) = 0$.

3.1.5 *Equation oscillations*

It is possible that the equation computation process will never terminate. That is, the old equations on some gate are always different from the new equations on that gate. This situation is known as equation oscillation. If the computation described in Section 3.1.3 proceeds through an arbitrary number (user declared) of timing lists, then an oscillation is declared and the message "equation oscillation" is printed for the user.

An example of an oscillation is shown in Fig. 6. In general, the objective is to stop the oscillation by selecting a stable set of equations. This can usually be done by setting the new equation on an oscillating



F^0	F^1	G^0	G^1	H^0	H^1	TIME
0	1	1	0	0	1	t
a^1	a^0	1	0	0	1	$t + 1$
a^1	a^0	a^0	a^1	0	1	$t + 2$
a^1	a^0	a^0	a^1	a^1	a^0	$t + 3$
0	$a^0 + a^1$	a^0	a^1	a^1	a^0	$t + 4$
0	$a^0 + a^1$	$a^0 + a^1$	0	a^1	a^0	$t + 5$
0	$a^0 + a^1$	$a^0 + a^1$	0	0	$a^0 + a^1$	$t + 6$
a^1	a^0	$a^0 + a^1$	0	0	$a^0 + a^1$	$t + 7$
a^1	a^0	a^0	a^1	0	$a^0 + a^1$	$t + 8$

Fig. 6—Equation oscillation.

gate, say $F^0(t)$ equal to $F^0(t) \cdot F^0(t - 1)$. This is intended to force the equations on gate F to stabilize by generating equations that make $F^0(t) = F^0(t - 1)$. This technique is not guaranteed to resolve all oscillations.

3.1.6 Complete description of equation derivation

The complete algorithm for generating the equations for a sequential circuit is shown in Fig. 7. Only two parts of the flow chart have not been explained previously in this section. One of these parts is the method of handling the zero-delay gates. The output of all the zero-delay gates are calculated before the next list of unit-delay gates is processed. These output equations are assigned to the zero-delay gates immediately.

The remaining unexplained part is the initial-state pass. This pass simply examines the circuit and propagates forward (before the input variables are applied) the effect of any gates set to logical 0 or 1 and any faults. For example, if gate G drives gate H and gate G is set to logical 0, this pass determines that the output of H should be logical 1.

This completes the description of the equation-generation process for fault-free sequential circuits. Next, the algorithm for generating the equations for a sequential circuit in the presence of a single fault is described.

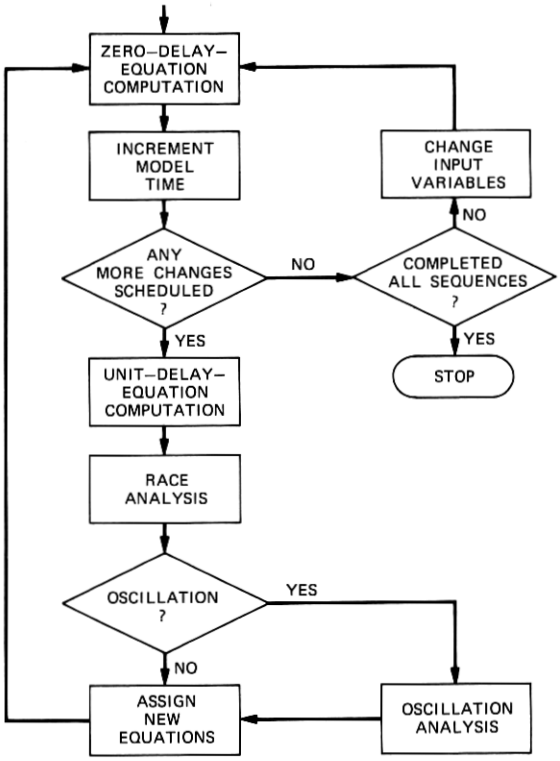


Fig. 7—Equation computation flow chart.

3.2 Equations containing faults

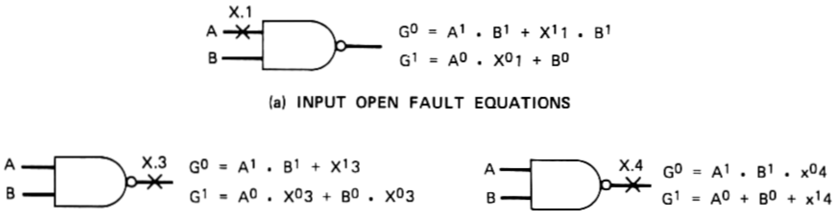
Equations for circuits containing faults may be derived in a way similar to those used for fault-free circuits. This method allows tests to be generated that can detect a specific fault. For efficiency, it is possible to consider several single faults simultaneously. The single faults considered here are the gate outputs stuck-at-one and stuck-at-zero as well as gate inputs open (e.g., open diode or emitter). The input open on a NAND or AND gate will be treated as stuck-at-one while the input open on a NOR or OR will be treated as stuck-at-zero.

Let the variables x^0i and x^1i represent fault variables. Let x^1i mean the fault $x.i$ is present in the circuit. Similarly, x^0i means the fault is not present in the circuit. For the fault variables, the i does not represent the ith vector in a sequence; rather, it represents the ith fault being considered. (Faults are always denoted by $x.i$ and the associated variables by x^0i or x^1i.) Since the single faults are assumed to be permanent, any fault $x.i$ will be present during the entire test sequence. The two states for $x.i$ allow the comparison of the faulty and fault-free circuit behavior to derive a test to detect the presence or absence of the fault in the circuit.

Consider the gate shown in Fig. 2. The fault-free equations are shown. If, however, the input-open fault on gate G from A is being examined, then the equations for gate G are shown in Fig. 8a. It is possible to set gate G to logical 0 either by applying A^1 and B^1 in the presence or absence of fault $x.1$ or by applying B^1 in the presence of fault $x.1$. It is also possible to set G to logical 1 by applying B^0 in the presence or absence of fault $x.1$ or by applying A^0 in the absence of fault $x.1$. Similar analysis for the output stuck-at-zero fault $x.3$ and the output stuck-at-one fault $x.4$ can easily be performed in the manner shown in Figs. 8b and 8c.

Now assume there is only one fault in the circuit and consider the case where the fault propagates around a loop and returns to the site of the failure. If the fault is the input open on gate G from A, then the equations shown in Fig. 8a can be rewritten as shown in Fig. 9a where fault $x.1$ is explicitly considered and D, E, F, $\cdots$ represent sum-of-products equations. Computing G^0 and G^1 yields the equations shown in Fig. 9a. Figure 9b considers the case where the fault exists (x^11) and Fig. 9c considers the case where the fault does not exist (x^01). Comparison of Figs. 9b and 9c with 9a shows that the computations proposed in this section for combinational circuits are also applicable to sequential circuits for the input-open case.

A similar analysis can be carried out for the gate output stuck-at-one and the output stuck-at-zero faults. This demonstrates that the equations shown in Fig. 8 for handling faults in combinational circuits are also applicable to sequential circuits.



$$G^0 = A^1 \cdot B^1 + X^11 \cdot B^1$$
$$G^1 = A^0 \cdot X^01 + B^0$$

(a) INPUT OPEN FAULT EQUATIONS

$$G^0 = A^1 \cdot B^1 + X^13$$
$$G^1 = A^0 \cdot X^03 + B^0 \cdot X^03$$

(b) OUTPUT STUCK–AT–ZERO EQUATIONS

$$G^0 = A^1 \cdot B^1 \cdot x^04$$
$$G^1 = A^0 + B^0 + x^14$$

(c) OUTPUT STUCK–AT–ONE EQUATIONS

Fig. 8—Equations for handling faults in combinational circuits.

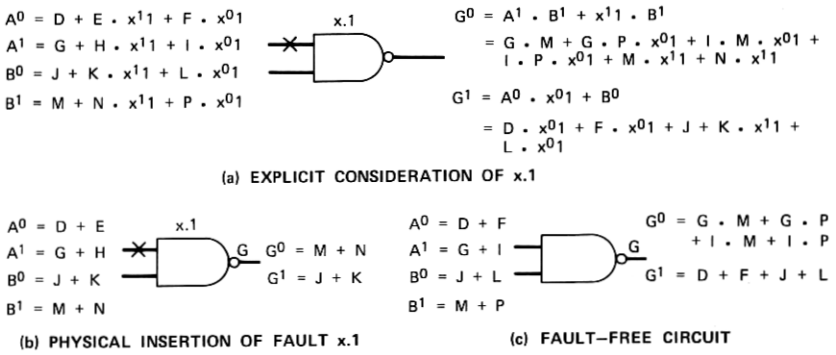

A⁰ = D + E · x¹1 + F · x⁰1
A¹ = G + H · x¹1 + I · x⁰1
B⁰ = J + K · x¹1 + L · x⁰1
B¹ = M + N · x¹1 + P · x⁰1
x.1
G⁰ = A¹ · B¹ + x¹1 · B¹
= G · M + G · P · x⁰1 + I · M · x⁰1 + I · P · x⁰1 + M · x¹1 + N · x¹1
G¹ = A⁰ · x⁰1 + B⁰
= D · x⁰1 + F · x⁰1 + J + K · x¹1 + L · x⁰1
(a) EXPLICIT CONSIDERATION OF x.1
A⁰ = D + E
A¹ = G + H
B⁰ = J + K
B¹ = M + N
x.1
G
G⁰ = M + N
G¹ = J + K
(b) PHYSICAL INSERTION OF FAULT x.1
A⁰ = D + F
A¹ = G + I
B⁰ = J + L
B¹ = M + P
G
G⁰ = G · M + G · P + I · M + I · P
G¹ = D + F + J + L
(c) FAULT–FREE CIRCUIT

Fig. 9—Equations for handling faults in sequential circuits.

3.3 The halting problem

One problem that must be discussed is how to determine when sequences of sufficient length have been generated. That is, given the equations that represent sequences of length N and the equations that represent sequences of length $N + 1$, will more information be gained by generating sequences of length $N + 2$? The question is answerable[9] if the feedbacks have been identified; however, the maximum sequence length contains factors of the form 2 to the power m, where m is the number of circuit inputs. For 500-gate, 40-input circuits, this is an absurd number.

There does not appear to be any practical method of determining when to halt the equation-generation process. In practice, the maximum sequence length to be considered is supplied by the user. The usual procedure is to start with sequences of length 1 and increase the sequence length until an acceptable level of undetected faults remains using the test-generation schemes presented in the next section. As might be expected, the run time increases significantly with increasing sequence length such that, even if it were simple to determine when to halt, it would probably not be economical. In practice, the halting problem has presented no difficulties. It is, however, an interesting theoretical problem.

In practice, the maximum sequence length required to detect all faults in the circuit provides some measure of the ease with which the circuit can be tested. The shorter the sequence length required, the more easily the circuit can be tested. This fact could be used as a circuit-design constraint by requiring that all circuits be testable with sequences of N or less where N is small. In fact, a 1000-gate, 11-state sequencer was designed so that the flip-flops representing the state could be written and read directly from circuit inputs and outputs. This produced an easily testable sequential circuit.

3.4 Clocked circuits

The algorithms that have been presented allow the circuit input leads to be treated as variables [e.g., (a^1, a^0)] or as logical values where logical 0 is (0, 1) and logical 1 is (1, 0). It is possible to allow some circuit inputs to be represented by variables and others by logical values. Clearly, it is possible to change the logical values between logical 0 and 1. Then we have the ability to apply a sequence of logical values to an input lead.

For example, suppose the circuit being considered has a clock lead whose normal operating waveform is 1–0–1–0 and all other input leads are static during this cycle. Then it is possible to apply variables to all but the clock lead and to supply the waveform (1, 0) – (0, 1) – (1, 0) – (0, 1) to the clock lead. In this way, ATG does less work since we have considered a sequence of length 4 on the clock lead and sequences of length 1 on all other leads. This is considerably more economical than computing sequences of length 4 over all input leads.

In a similar way, user-specified initialization sequences can be applied to the circuit to place it in some desired state before allowing ATG to select the next input sequence. This is an effective way of using ATG.

3.5 Self-initializing circuits

Certain classes of sequential circuits are self-initializing in that, regardless of the initial state, the circuit always assumes a known state when power is applied. A simple example of such a circuit is shown in Fig. 10. Because the flip-flop always initializes to $C = 0$, $D = 1$ or $C = 1$, $D = 0$, gate F will always be logical 0 forcing the flip-flop to the $C = 1$, $D = 0$ state.

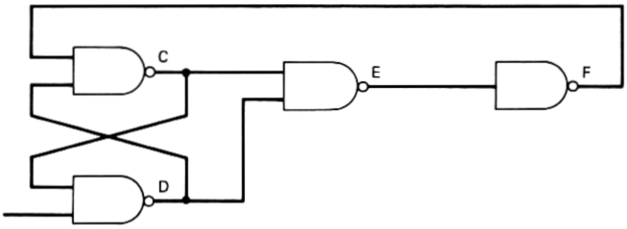


Fig. 10—Self-initializing circuit.

If this circuit is assumed to be in an unknown state ($B = C = D = E = F = X$), then because $\bar{X} = X$, the basic ATG algorithm does not determine the required initial state. The operation of ATG requires that gates be forced to some initial state by the application of input vectors from some initial state. Hence, self-initializing circuits require that the proper initial state be specified by the user. This has not proved to be a problem in practice since most circuits contain initializing leads.

IV. TEST GENERATION FROM THE EQUATIONS

Two different schemes for generating tests are described in this section. The first scheme described is the generation of tests to detect single faults, where the equations are derived in terms of these faults. However, in a 500-gate, 2000-fault circuit it is not economical to attack all 2000 faults on a one-at-a-time basis. The second method for test generation is aimed at detecting large numbers of faults as easily as possible. It attacks the problem by essentially attempting to detect the stuck-at-one and stuck-at-zero fault at each circuit input lead by observing each circuit output lead. This is called the maximum-cover strategy. This scheme typically detects around 90 percent of the classical faults if the equations reasonably describe the circuit—that is, if the sequence length used is long enough.

4.1 Generating a test to detect a fault

To detect fault $x.i$, it is necessary to select a test (input sequence) that will force some output of the circuit to have the value k for $k = 0, 1$ in the presence of the fault $x.i$ and to have the value $\bar{k}$ in the absence of fault $x.i$ starting from the given initial state. Let one output gate G of a circuit have the following equations (by simple factoring):

$$\begin{aligned} G^0 &= A + B \cdot x^1 i + C \cdot x^0 i \\ G^1 &= D + E \cdot x^1 i + F \cdot x^0 i, \end{aligned} \tag{6}$$

where $A, B, \cdots, F$ are also sum-of-products expressions. This means that the terms in $A(D)$ are the only terms that set $G = 0$ ($G = 1$) regardless of the presence or absence of fault $x.i$. The tests to detect fault $x.i$ at gate G are given by $B \cdot F + C \cdot E$. This is proven as follows.

Since the fault either exists or does not exist, $x^1 i \cdot x^0 i = 0$. First consider the case in which $k = 0$. Since $G^1(G^0)$ represents exactly those conditions that set $G = 1$ ($G = 0$), then $G^1(x^0 i = 1) = D + F$ represents those conditions that set $G = 1$ in the absence of fault $x.i$. Similarly, $G^0(x^1i = 1) = A + B$ represents those conditions that set $G = 0$ in the presence of the fault. Hence, every condition (input vector) that makes the good output of $G = 0$ and makes the faulty output of $G = 1$ is given by $(A + B)\cdot(D + F) = A\cdot D + B\cdot D + A\cdot F + B\cdot F$. Examination of the terms of this equation reveals $A\cdot D = B\cdot D = A\cdot F = 0$. Term $A\cdot D = 0$ because, if it were not zero, then there would be some term in $A\cdot D$ that could set $G = 1$ and $G = 0$ simultaneously. This is clearly impossible. Similarly, $B\cdot D \neq 0$ $(A\cdot F \neq 0)$ implies that in the presence (absence) of the fault, there is some term in $B\cdot D(A\cdot F)$ that can set $G = 1$ and $G = 0$ simultaneously. Therefore, any term that can set $G = 0$ in the presence of the fault and $G = 1$ in the absence of the fault must be in $B\cdot F$.

For the case in which $k = 1$, the test must be a term of $(D + E)\cdot(A + C) = D\cdot A + D\cdot C + E\cdot A + E\cdot C$. By similar analysis, $A\cdot D = D\cdot C = A\cdot E = 0$. Therefore, a term that sets $G = 1$ in the presence of the fault and $G = 0$ in the absence of the fault must be in $E\cdot C$.

Since the problem is to detect fault $x.i$ without regard to the output value of G, any term in $B\cdot F + E\cdot C$ is a valid test. Therefore, all tests to detect fault $x.i$ at gate G can be expressed as

$$\text{Detection Tests} = B\cdot F + E\cdot C.$$

If $B\cdot F + E\cdot C = 0$, there is no test that will detect fault $x.i$ at gate G. It is then necessary to examine each remaining circuit output to determine if $x.i$ is detectable. If $x.i$ is not detectable at any circuit output, then there exists no test to detect $x.i$ for the sequence length specified.

Clearly, this algorithm generates every test that will detect fault $x.i$ at each output. Since it is probably necessary to detect the fault only once, the first valid test found usually terminates the process.

4.2 The maximum-cover strategy

The maximum-cover strategy has been quite successful. In most cases, it has detected from 85 to 100 percent of the faults in the circuit that are detectable with the maximum sequence length specified. For highly sequential circuits, a short-maximum-sequence length may detect few faults because the circuit cannot be exercised completely without using a long sequence of input vectors.

The maximum-cover strategy operates on the fault-free equations derived for the circuit according to the maximum sequence length specified. The basic idea is simply to attempt to detect each primary circuit input fault at each circuit output. Factoring the output equations as before yields:

$$\begin{aligned} F^0 &= A + B\cdot a^1j + C\cdot a^0j \\ F^1 &= D + E\cdot a^1j + F\cdot a^0j, \end{aligned} \tag{8}$$

where A, B, C, D, E, and F are sum-of-product terms. This case attempts to detect the input faults on a at the output F. A test is formed in a manner similar to that used for detecting faults, except that it is necessary here to specify the value to be assigned to input lead $a.j$.

$$\text{Maximum-Cover Test} = (E\cdot C + F\cdot B)\cdot(a^1j + a^0j).$$

This process is repeated until an attempt has been made to test each circuit input fault at each output lead. This scheme actually produces every test that satisfies the above equation. The shortest test (fewest input leads set to logical 0 or 1) is selected in each case.

The time spent performing this computation is usually much less than that required to derive the equations. Also the time and results of the maximum-cover operation must be weighed against the cost of detecting additional faults on a one-at-a-time basis. Thus, while maximum cover is an expensive heuristic (when compared to, say, random-number test generation), it provides a set of tests that is usually good enough so that one can economically attack the remaining faults on a one-at-a-time basis. As a general rule, about 5 to 10 faults can be detected using the one-at-a-time strategies for the same cost as one pass of the maximum-cover strategy which inherently tries to detect all faults.

V. EXPERIMENTAL RESULTS

The final measure of an automatic test generation system is how well it does its job on real circuits. The ATG system has been programmed and is being used at several locations in Bell Laboratories. The algorithms presented here are generally not useful for hand computation. The version of ATG used by Bell Laboratories on the IBM 360, Model 67, collects certain data each time it runs successfully. The data collected include the execution CPU time, number of test vectors generated, number of faults detected, number of gates in the circuit, and number of flip-flops in the circuit.

This implementation of ATG requires about 100,000 bytes for program storage. Other storage, used during execution, depends on the characteristics of the circuit being run. As the equations get longer, the storage requirements increase. Generally speaking, ATG requires from one to five megabytes of virtual storage. This implementation allows only unit- and zero-gate delays, handles single stuck-at-one and stuck-at-zero faults, and generates fault-detection tests for single faults as well as the maximum-cover test-generation strategy.

The data that have been collected indicate that ATG has been primarily used to generate tests via the maximum-cover strategy. In a few uses of ATG, the user attempted to detect only specified faults; these data are not included in this paper.

The data collected represent only successful ATG runs. If the same circuit was run several times, then only the run that produced the fewest undetected faults (e.g., used the longest sequence length) is included. This is consistent with the recommended operational procedure, which starts with a short sequence length and increases it until an acceptable level of fault detection is reached. Faults in unused gates are included both in the undetected faults and in the total number of faults in the circuit.

The results of 300 ATG runs on 120 circuits using the maximum-cover strategy are summarized in Figs. 11 through 15. The average circuit contained about 270 gates including about 10 flip-flops in the sequential circuits. Thirty-two circuits were combinational. ATG produced an average of 94 vectors in an average of 43 seconds of IBM 360, Model 67, CPU time resulting in an average detection level of 88 percent of the total number of faults in the circuit. However, the median percentage of undetected faults was only 7 to 8 percent. The longest sequence length used for these circuits was 5. Unfortunately, there is almost no correlation between the five parameters plotted in Figs. 11 through 15. The data correlate only in the extreme cases. For example, the circuit with 32 flip-flops produced a large

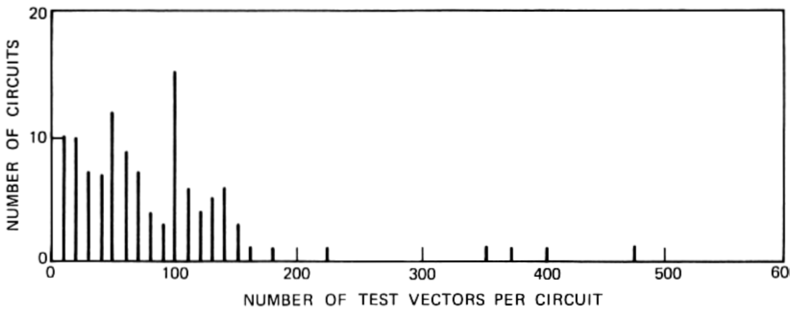


Fig. 11—Distribution of number of test vectors per circuit.

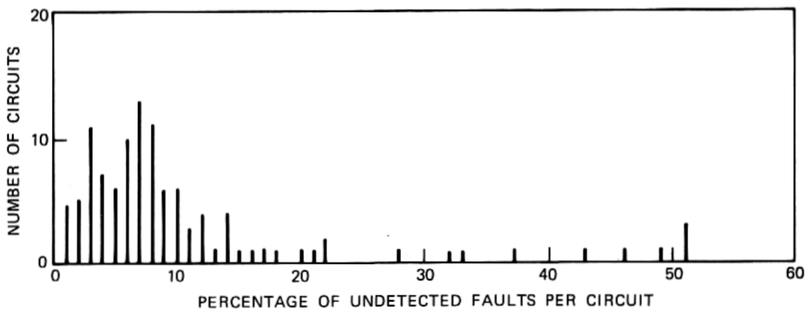


Fig. 12—Distribution of percentage of undetected faults per circuit.

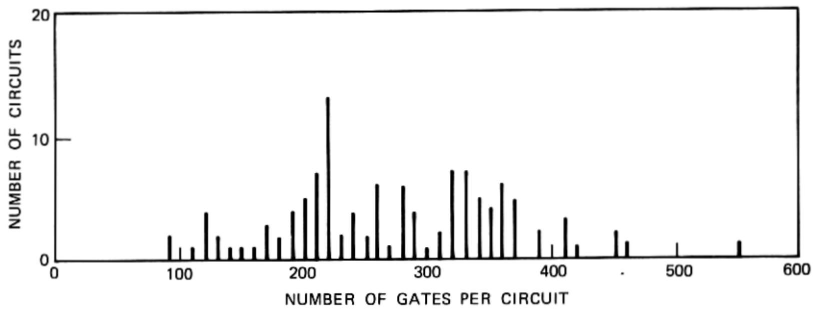


Fig. 13—Distribution of number of gates per circuit.

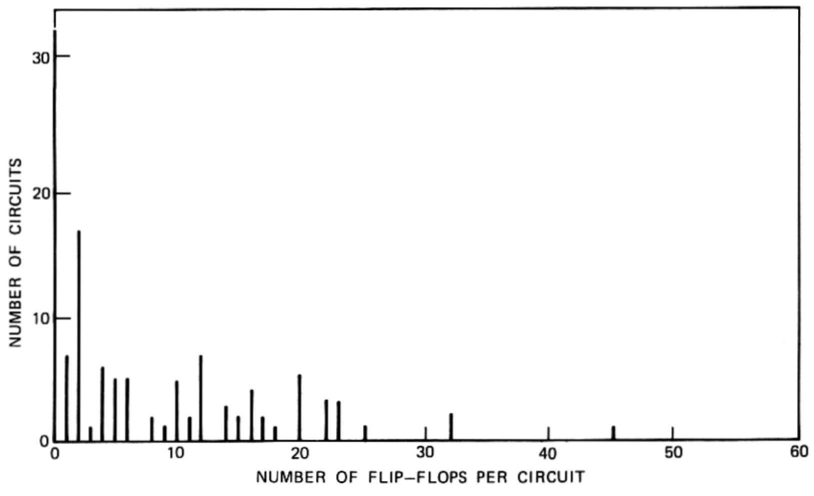


Fig. 14—Distribution of number of flip-flops per circuit.

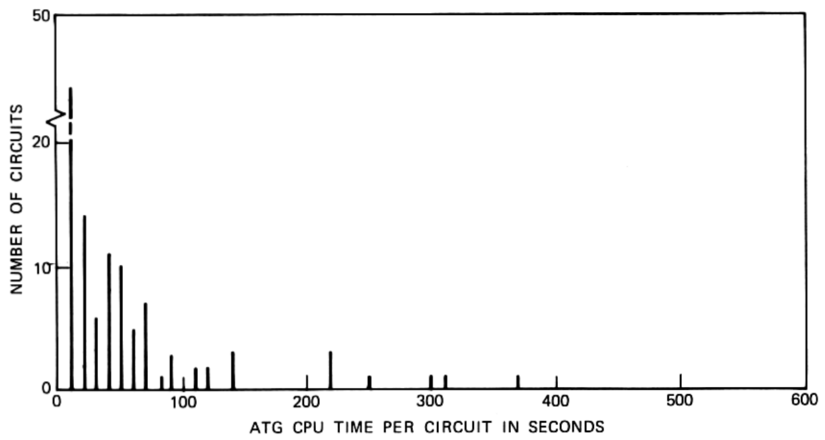


Fig. 15—Distribution of CPU time per circuit.

percentage of undetected faults. For most of the data, none of the parameters correlates significantly.

ATG did not produce acceptable results on all circuits. In general, ATG is limited by the length of the equations generated. As these equations become long, the execution time increases and ATG may not terminate successfully due to excessive run time and/or storage requirements. The equations can become long as a result of long sequence lengths (e.g., shift registers and counters) or as a result of the function of the logic circuit (e.g., parity trees and adders). In addition, circuits such as parity trees produce quite long equations and ATG generates more vectors than the minimum required.

One circuit recently run on ATG using the maximum cover strategy is worthy of special mention. The circuit is a 1000-gate, 11-state sequencer plus input, output, and transition logic. The sequencer state, represented by four *D*-flip-flops, can be read and written from circuit outputs and inputs respectively. Extensive use is made of the system clock to control transitions and gating. The clock waveforms were supplied to ATG by the user. ATG, using the clock and sequences of length one, generated 770 vectors in about 800 seconds, detecting about 95 percent of the faults in the circuit. The success of ATG here is partially due to the "easily testable design" which allows the sequencer state to be directly read and written.

In practice, while ATG will not efficiently handle all circuits, it appears to be an economical tool for automatic test generation for "mildly" sequential circuits containing around 500 gates. The design of circuits in an "easily testable" manner greatly eases the work required to automatically generate test vectors for the circuit.

VI. SUMMARY AND CONCLUSION

In summary, the method used to generate tests is as follows:

(*i*) Set the maximum sequence length $k = 1$.
(*ii*) Generate equations for the logic circuit with sequence length k.
(*iii*) Generate tests using maximum-cover strategy.
(*iv*) Simulate the tests. If the percentage of undetected faults is less than, say, 10 percent, proceed to step (*v*). Otherwise, set $k = k + 1$ and return to step (*ii*).
(*v*) Generate tests for remaining undetected faults (one fault at a time) detectable with sequence length k.
(*vi*) If an acceptable percentage of undetected faults remains, stop. Otherwise, set $k = k + 1$ and return to step (*v*).

In practice, most users of ATG have been satisfied with the ATG results without trying steps (*v*) or (*vi*).

The algorithms treat a logic network as an interconnection of gates which are assigned some fixed time delay. The technique generates two equations, $F^0(t)$ and $F^1(t)$, for each gate in the circuit. These equations denote the input conditions required to set gate F to logical 0 and 1 respectively at time t. Because the technique starts from the circuit inputs and proceeds forward through the circuit (like the signal flow), it is not necessary to identify feedback leads. Therefore, both combinational and sequential circuits can be handled by the same algorithm.

The primary difference between combinational and sequential-circuit test generation is that several input vectors may be required in a sequential circuit to set the desired state, detect some fault, and then propagate the fault to some output lead. The number of input vectors required to perform some test on the circuit is called the sequence length of the test. A sequence length of one is sufficient to generate all tests for a combinational circuit since it has no memory. The maximum sequence length to be considered is supplied by the user.

The test-generation algorithms first generate the equations for the circuit, taking into consideration the gate delays and the maximum sequence length specified. These equations also take into account the effect of various single stuck-at-one, stuck-at-zero, or open-gate input faults when tests are being generated for specific faults. Then, from these equations, the algorithms will generate a test for any of the above faults if such a test exists within the sequence length specified.

Equations may also be generated that represent only the fault-free circuit. It is then possible to generate tests from these equations which exercise the circuit in such a way that many faults are detected. This has been a popular feature because it produces good results economically.

These algorithms have been implemented and are currently being used to generate tests for circuits containing around 500 gates. Quite good results have been produced using the maximum-cover technique. A median of 7 to 8 percent undetected stuck-at faults was reached in less than 1 minute of IBM 360, Model 67, CPU time on a sample of some 120 circuits. Because of the success of the maximum-cover techniques, very little use has been made of the "single–fault" techniques.

In conclusion, ATG is a production system that has been found to be a valuable tool for the generation of circuit pack tests.

VII. ACKNOWLEDGMENTS

The author gratefully acknowledges the guidance of Professor S. S. Yau of Northwestern University during the course of this work. The work of G. F. Shuttleworth of Bell Laboratories on the nonvirtual memory version of ATG is also acknowledged. In addition, the support of W. Ulrich and R. W. Ketchledge during the course of this work is appreciated.

REFERENCES

1. D. B. Armstrong, "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets," IEEE Trans. on Computers, *EC-15*, No. 1 (February 1966), pp. 66–73.
2. F. F. Sellers, Jr., M. Y. Hsiao, and L. W. Bearnson, "Analyzing Errors With the Boolean Difference," IEEE Trans. on Computers, *EC-17*, No. 7 (July 1968), pp. 676–683.
3. J. P. Roth, W. G. Bouricius, and P. R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," IEEE Trans. on Computers, *EC-16*, No. 5 (October 1967), pp. 567–580.
4. S. A. Szygenda, D. W. Rouse, and E. W. Thompson, "A Model and Implementation of a Universal Time Delay Simulator for Large Digital Nets," Proc. AFIPS Spring Joint Computer Conference, 1970, pp. 207–216.
5. S. G. Chappell, C. H. Elmendorf, and L. D. Schmidt, "LAMP: Logic-Circuit Simulators," B.S.T.J., this issue, pp. 1451–1476.
6. S. S. Yau and Y. S. Tang, "Generation of Shortest Test Sequences for Individual Faults of Sequential Circuits," to be published.
7. G. R. Putzolu and J. P. Roth, "A Heuristic Algorithm for the Testing of Asynchronous Circuits," IEEE Trans. on Computers, *C-20*, No. 6 (June 1971), pp. 639–647.
8. M. Y. Hsiao and D. K. Chia, "Boolean Difference for Fault Detection in Asynchronous Sequential Machines," IEEE Trans. on Computers, *C-20*, No. 11 (November 1971), pp. 1356–1361.
9. E. F. Moore, "Gedanken Experiments on Sequential Machines," Automata Studies, Princeton: Princeton University Press, 1956, pp. 129–153.