5. FAULT SIMULATION

About This Chapter

First we review applications of fault simulation. Then we examine fault simulation techniques for SSFs. We describe both general methods — serial, parallel, deductive, and concurrent — and techniques specialized for combinational circuits. We also discuss fault simulation with fault sampling and statistical fault analysis. This chapter emphasizes gate-level and functional-level models and SSFs. Fault simulation for technology-specific faults models and fault simulation for bridging faults are dealt with in separate chapters.

5.1 Applications

Fault simulation consists of simulating a circuit in the presence of faults. Comparing the fault simulation results with those of the fault-free simulation of the same circuit simulated with the same applied test T, we can determine the faults detected by T.

One use of fault simulation is to **evaluate (grade) a test** T. Usually the grade of T is given by its *fault coverage*, which is the ratio of the number of faults detected by T to the total number of simulated faults. This figure is directly relevant only to the faults processed by the simulator, as even a test with 100 percent fault coverage may still fail to detect faults outside the considered fault model. Thus the fault coverage represents only a lower bound on the *defect coverage*, which is the probability that T detects any physical fault in the circuit. Experience has shown that a test with high coverage for SSFs also achieves a high defect coverage. Test evaluation based on fault simulation has been applied mainly to the SSF model, both for external testing and for self-testing (i.e., evaluation of self-test programs).

The quality of the test greatly influences the quality of the shipped product. Let Y be the manufacturing *yield*, that is, the probability that a manufactured circuit is defect-free. Let DL denote the *defect level*, which is the probability of shipping a defective product, and let d be the defect coverage of the test used to check for manufacturing defects. The relation between these variables is given by

$$DL = 1 - Y^{1-d}$$

[Williams and Brown 1981]. Assuming that the fault coverage is close to the defect coverage, we can use this relation, illustrated in Figure 5.1, to determine the fault coverage required for a given defect level. For example, consider a process with 0.5 yield. Then to achieve a 0.01 defect level — that is, 1 percent of the shipped products are likely to be defective — we need 99 percent fault coverage. A test with only 95 percent fault coverage will result in a defect level of 0.035. If, however, the yield is 0.8, then 95 percent fault coverage is sufficient to achieve a defect level of 0.01. Other aspects of the relation between product quality and fault coverage are analyzed in [Agrawal *et al.* 1981], [Seth and Agrawal 1984], [Daniels and Bruce 1985], and [McCluskey and Buelow 1988].

Fault simulation plays an important role in **test generation**. Many test generation systems use a fault simulator to evaluate a proposed test T (see Figure 5.2), then

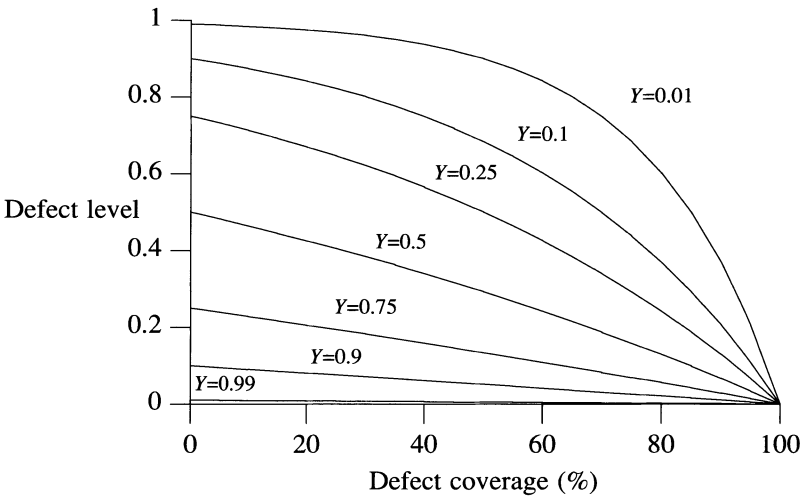


Figure 5.1 Defect level as a function of yield and defect coverage

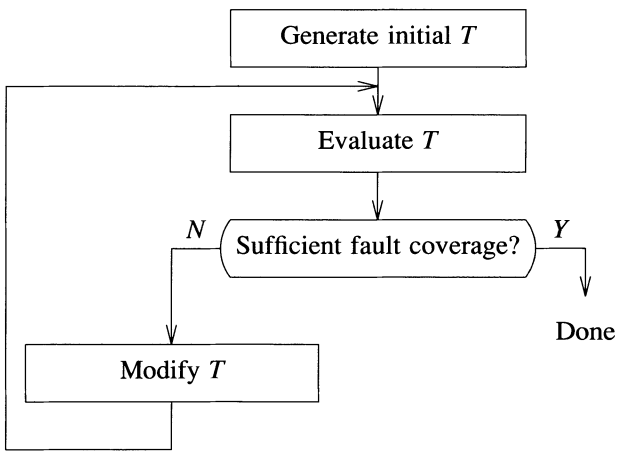


Figure 5.2 General use of fault simulation in test generation

change T according to the results of the fault simulation until the obtained coverage is considered satisfactory. The test T is modified by adding new vectors and/or by discarding some of its vectors that did not contribute to achieving good coverage. These changes may be made by a program or by a test designer in an interactive mode.

Another use of fault simulation in test generation is illustrated in Figure 5.3. Many test generation algorithms are fault-oriented; that is, they generate a test for one specified fault, referred to as the *target fault*. Often the same test also detects many other faults that can be determined by fault simulation. Then all the detected faults are discarded from the set of simulated faults and a new target fault is selected from the remaining ones.

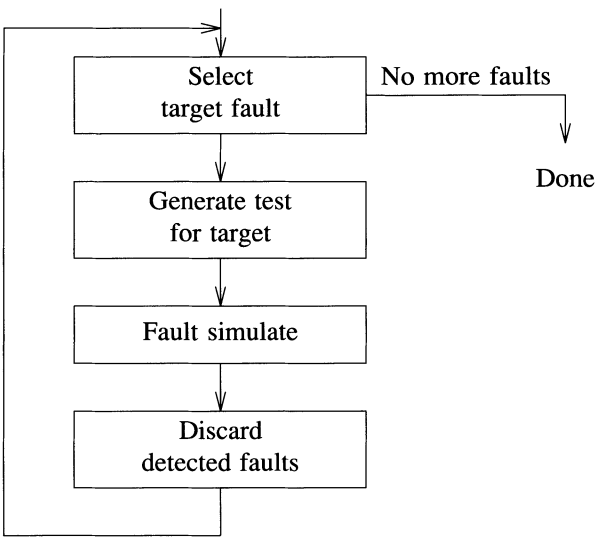


Figure 5.3 Fault simulation used in the selection of target faults for test generation

Fault simulation is also used to **construct fault dictionaries**. Conceptually, a fault dictionary stores the output response to T of every faulty circuit N_f corresponding to a simulated fault f. In fact, a fault dictionary does not store the complete output response R_f of every N_f, but some function $S(R_f)$, called the *signature* of the fault f. The fault location process relies on comparing the signature obtained from the response of the circuit under test with the entries in the precomputed fault dictionary.

Constructing a fault dictionary by fault simulation means, in principle, computing the response in the presence of every possible fault *before testing*. A different approach, referred to as **post-test diagnosis**, consists of first isolating a reduced set of "plausible" faults (i.e., faults that may be consistent with the response obtained from the circuit under test), then simulating only these faults to identify the actual fault. Diagnosis techniques will be discussed in more detail in a separate chapter.

Another application of fault simulation is to **analyze the operation of a circuit in the presence of faults**. This is especially important in high-reliability systems, since some faults may drastically affect the operation. For example:

- A fault can induce races and hazards not present in the fault-free circuit.
- A faulty circuit may oscillate or enter a deadlock (hang-up) state.
- A fault can inhibit the proper initialization of a sequential circuit.
- A fault can transform a combinational circuit into a sequential one or a synchronous circuit into an asynchronous one.

Fault simulation of self-checking circuits is used to verify the correct operation of their error-detecting and error-correcting logic.

5.2 General Fault Simulation Techniques

5.2.1 Serial Fault Simulation

Serial fault simulation is the simplest method of simulating faults. It consists of transforming the model of the fault-free circuit N so that it models the circuit N_f created by the fault f. Then N_f is simulated. The entire process is repeated for each fault of interest. Thus faults are simulated one at a time. The main advantage of this method is that no special fault simulator is required, as N_f is simulated by a fault-free simulator. Another advantage is that it can handle any type of fault, provided that the model of N_f is known. However, the serial method is impractical for simulating a large number of faults because it would consume an excessive amount of CPU time.

The other general fault simulation techniques — parallel, deductive, and concurrent — differ from the serial method in two fundamental aspects:

- They determine the behavior of the circuit N in the presence of faults without explicitly changing the model of N.
- They are capable of simultaneously simulating a set of faults.

These three techniques will be described in the following sections.

5.2.2 Common Concepts and Terminology

All three techniques — parallel, deductive, and concurrent — simultaneously simulate the fault-free circuit N (also called the *good circuit*) and a set of faulty (or *bad*) circuits $\{N_f\}$. Thus any fault simulation always includes a fault-free simulation run. If all the faults of interest are simulated simultaneously, then fault simulation is said to be done in *one pass*. Otherwise the set of faults is partitioned and fault simulation is done as a *multipass* process, in which one subset of the total set of faults is dealt with in one pass. In general, large circuits require multipass fault simulation.

In addition to the activities required for fault-free simulation (described in Chapter 3), fault simulation involves the following tasks:

- fault specification,
- fault insertion,
- fault-effect generation and propagation,
- fault detection and discarding.

Fault specification consists of defining the set of modeled faults and performing fault collapsing. **Fault insertion** consists of selecting a subset of faults to be simulated in one pass and creating the data structures that indicate the presence of faults to the simulation algorithm. These data structures are used to **generate effects of the inserted faults** during simulation. For example, let f be a s-a-1 fault inserted on line i. Then whenever a value 0 would propagate on line i (in the circuit N_f), the simulator changes it to 1. Most of the work in fault simulation is related to the **propagation of fault effects**. Whenever an effect of a fault f propagates to a primary output j (such that the values of j in the good and the faulty circuit are both binary), the simulator marks f as detected. The user may specify that a fault detected k times should be discarded from the set of simulated faults (usually k=1). **Fault discarding** (also called *fault dropping*) is the inverse process of fault insertion.

5.2.3 Parallel Fault Simulation

In parallel fault simulation [Seshu 1965] the good circuit and a fixed number, say W, of faulty circuits are simultaneously simulated. A set of F faults requires $\lceil F/W \rceil$ passes.[1] The values of a signal in the good circuit and the values of the corresponding signals in the W faulty circuits are packed together in the same memory location of the host computer. Depending on the implementation, a "location" consists of one or more words.

For example, if we use 2-valued logic and a 32-bit word, then W=31 (see Figure 5.4). Consider an AND gate with inputs A and B. Each bit in the word associated with a signal represents the value of that signal in a different circuit. Traditionally, bit 0 represents a value in the good circuit. Then using a logical AND instruction between the words associated with A and B, we evaluate (in parallel) the AND gate in the good circuit and in each of the 31 faulty circuits. Similarly we use an OR operation to evaluate an OR gate, and a bit complement operation for a NOT. A NAND gate requires an AND followed by a complement, and so on. A sequential element is represented by a Boolean expression; for example, for a JK F/F

$$Q^+ = J\bar{Q} + \bar{K}Q$$

where Q^+ and Q are, respectively, the new and the current state of the F/F. Thus the evaluation can be carried out using AND, OR, and complement operators.

Let v_i be the value propagating onto line i in the faulty circuit N_f, where f is the fault j s-a-c. Every line $i \neq j$ takes the value v_i, but the value of j should always be c. The new value of line i, v_i', can be expressed as

$$v_i' = v_i\overline{\delta_{ij}} + \delta_{ij}c$$

where

1. $\lceil x \rceil$ denotes the smallest integer greater than or equal to x.

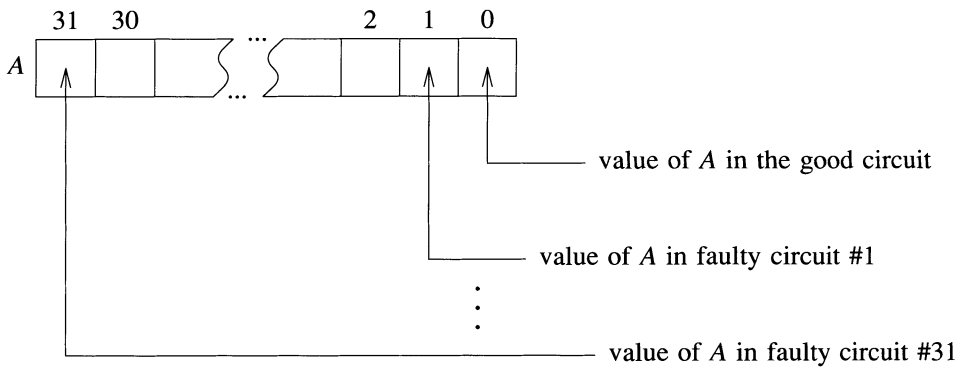


Figure 5.4 Value representation in parallel simulation

$$\delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

The above equations represent the process of fault insertion for one fault *f (j s-a-c)*. For *W* faults, this process is carried on in parallel using two *mask words* storing the values δ_{ij} and *c* in the bit position corresponding to fault *f*. Figure 5.5 shows a portion of a circuit, the masks used for fault insertion on line *Z*, and the values of *Z* before and after fault insertion. The first mask — *I* — associated with a line indicates whether faults should be inserted on that line and in what bit positions, and the second — *S* — defines the stuck values of these faults. Thus after evaluating gate *Z* by $Z = X1.Y$, the effect of inserting faults on *Z* is obtained by

$$Z' = Z.\bar{I}_Z + I_Z.S_Z$$

Fault insertion for *X1* and *Y* is similarly done before evaluating *Z*.

The above technique has several possible implementations [Thompson and Szygenda 1975].

For 3-valued logic (0,1,*u*), one bit is not sufficient to represent a signal value. The coding scheme shown in Figure 5.6 uses two words, *A1* and *A2*, to store the *W* values associated with signal *A*. Since the codes for the values 0 and 1 are, respectively, 00 and 11, the logical AND and OR operations can be applied directly to the words *A1* and *A2*. Hence, to evaluate an AND gate with inputs *A* and *B* and output *C*, instead of $C = A.B$ we use

$$C1 = A1.B1$$
$$C2 = A2.B2$$

The complement operator, however, cannot be applied directly, as the code 01 will generate the illegal code 10. An inversion *B* = NOT (*A*) is realized by

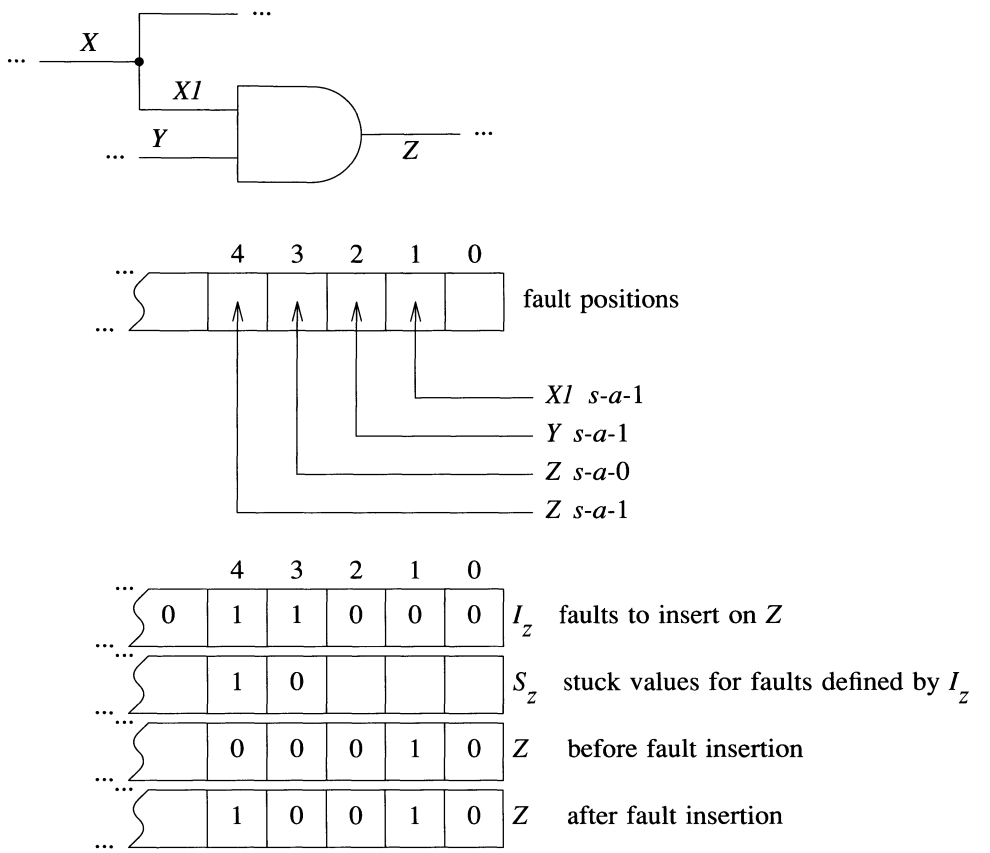


Figure 5.5 Fault insertion on Z

	Value of A		
	0	1	u
$A1$	0	1	0
$A2$	0	1	1

Figure 5.6 Coding for 3-valued logic

$$B1 = \overline{A2}$$
$$B2 = \overline{A1}$$

that is, we complement $A1$ and $A2$ and interchange them. Figure 5.7 shows a sample computation for $Z = \overline{X.Y}$ (with a 3-bit word).

	Values	Encoding	
X	0 1 u	$X1$ $X2$	0 1 0 0 1 1
Y	0 1 1	$Y1$ $Y2$	0 1 1 0 1 1
$X.Y$	0 1 u	$X1.Y1$ $X2.Y2$	0 1 0 0 1 1
$Z = \overline{X.Y}$	1 0 u	$Z1$ $Z2$	1 0 0 1 0 1

Figure 5.7 Example of NAND operation in 3-valued parallel simulation

Other coding techniques for parallel fault simulation with three or more logic values are described in [Thompson and Szygenda 1975] and [Levendel and Menon 1980].

It is possible to reduce the number of passes by simulating several *independent faults* simultaneously. Let S_i denote the set of lines in the circuit that can be affected by the value of line i. Faults defined on lines i and j, such that $S_i \cap S_j = \varnothing$, are said to be independent. Independent faults cannot affect the same part of the circuit, and they can be simulated in the same bit position. The least upper bound on the number of independent faults that can be processed simultaneously in the same bit position is the number of primary outputs — p — of the circuit. Hence the minimum number of passes is $\lceil F/(W.p) \rceil$. The potential reduction in the number of passes should be weighed against the cost of identifying independent faults. Algorithms for determining subsets of independent faults are presented in [Iyengar and Tang 1988].

To discard a fault f, first we have to stop inserting it. This is simply done by turning off the corresponding bit position in the mask word used for fault insertion. However, the effects of the fault f may have been propagated to many other lines, and additional processing is needed to stop all the activity in the circuit N_f [Thompson and Szygenda 1975].

Limitations

In parallel fault simulation several faulty circuits can be simulated in parallel, provided that for evaluation we use only operations that process each bit independently of all others. This requires that we model elements by Boolean equations, and therefore we cannot directly use evaluation routines that examine the input values of the evaluated elements, or routines based on arithmetic operations. But these types of evaluations are convenient for evaluating functional elements, such as memories and counters. To integrate these techniques into a parallel fault simulator, the individual bits of the faulty circuits are extracted from the packed-value words, the functional elements are individually evaluated, and then the resulting bits are repacked. Thus *parallel simulation is compatible only in part with functional-level modeling.*

Evaluation techniques based on Boolean equations are adequate for binary values, but they become increasingly complex as the number of logic values used in modeling increases. Hence, *parallel simulation becomes impractical for multivalued logic.*

In parallel fault simulation, an event occurs when the new value of a line differs from its old value in at least one bit position. Such an event always causes W evaluations, even if only one of the W evaluated elements has input events. Although it may appear that the unnecessary evaluations do not take extra time, because they are done in parallel with the needed ones, they do represent wasted computations. W evaluations are done even after all the faults but one have been detected and discarded. Thus *parallel fault simulation cannot take full advantage of the concept of selective trace simulation, or of the reduction in the number of faults caused by fault dropping.*

5.2.4 Deductive Fault Simulation

The deductive technique [Armstrong 1972, Godoy and Vogelsberg 1971] simulates the good circuit and deduces the behavior of *all* faulty circuits. "All" denotes a theoretical capability, subject in practice to the size of the available memory. The data structure used for representing fault effects is the **fault list**. A fault list L_i is associated with every signal line i. During simulation, L_i is the set of all faults f that cause the values of i in N and N_f to be different at the current simulated time. If i is a primary output and all values are binary, then L_i is the set of faults detected at i.

Figure 5.8 illustrates the difference between the value representation in parallel and in deductive simulation. Suppose that we have F faults and a machine word with $W \geq F + 1$, hence we can simulate all the faults in one pass. Then in parallel simulation the word associated with a line i stores the value of i in every faulty circuit. During simulation, however, the value of i in most faulty circuits is the same as in the good circuit. This waste is avoided in deductive simulation by keeping only the bit positions (used as fault names) that are different from the good value.

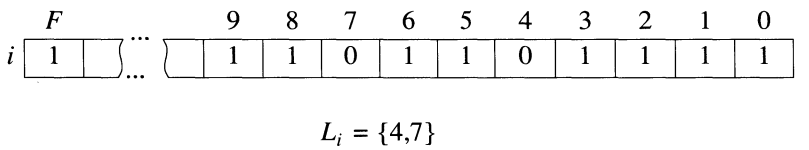


Figure 5.8 Fault-effects representation in parallel and deductive fault simulation

Given the fault-free values and the fault lists of the inputs of an element, the basic step in deductive simulation is to compute the fault-free output value and the output fault list. The computation of fault lists is called *fault-list propagation.* Thus in addition to the logic events which denote changes in signal values, a deductive fault simulator also propagates *list events* which occur when a fault list changes, i.e., when a fault is either added to or deleted from a list.

5.2.4.1 Two-Valued Deductive Simulation

In this section we assume that all values are binary. Consider, for example, an AND gate with inputs A and B and output Z (Figure 5.9). Suppose A=B=1. Then Z=1 and any fault that causes a 0 on A or B will cause Z to be erroneously 0. Hence

$$L_Z = L_A \cup L_B \cup \{Z\ s\text{-}a\text{-}0\}$$

Figure 5.9

Now suppose that A=0 and B=1. Then Z=0 and any fault that causes A to be 1 without changing B, will cause Z to be in error; i.e., Z=1. These are the faults in L_A that are not in L_B:

$$L_Z = (L_A \cap \overline{L_B}) \cup \{Z\ s\text{-}a\text{-}1\} = (L_A - L_B) \cup \{Z\ s\text{-}a\text{-}1\}$$

where $\overline{L_B}$ is the set of all faults *not* in L_B. Note that a fault whose effect propagates to both A and B does not affect Z.

Let I be the set of inputs of a gate Z with controlling value c and inversion i. Let C be the set of inputs with value c. The fault list of Z is computed as follows:

$$\textbf{if } C = \varnothing \textbf{ then } L_Z = \{\bigcup_{j \in I} L_j\} \cup \{Z\ s\text{-}a\text{-}(c \oplus i)\}$$

$$\textbf{else } L_Z = \{\bigcap_{j \in C} L_j\} - \{\bigcup_{j \in I-C} L_j\} \cup \{Z\ s\text{-}a\text{-}(\overline{c} \oplus i)\}$$

In other words, if no input has value c, any fault effect on an input propagates to the output. If some inputs have value c, only a fault effect that affects all the inputs at c without affecting any of the inputs at $\overline{c}$ propagates to the output. In both cases we add the local fault of the output.

Example 5.1: Consider the circuit in Figure 5.10. After fault collapsing (see Example 4.8), the set of faults we simulate is $\{a_0, a_1, b_1, c_0, c_1, d_1, e_0, g_0, h_0, h_1\}$ (α_v denotes α *s-a-v*). Assume the first applied test vector is 00110. The computation of fault lists proceeds as follows:

$$L_a = \{a_1\},\ L_b = \{b_1\},\ L_c = \{c_0\},\ L_d = \varnothing,\ L_e = \varnothing$$

$$L_f = L_a \cap L_b = \varnothing,\ L_g = L_c \cup \{g_0\} = \{c_0, g_0\},\ L_h = L_c \cup \{h_0\} = \{c_0, h_0\}$$

$L_j = L_g - L_f = \{c_0, g_0\}$, $L_i = L_d \cup L_h = \{c_0, h_0\}$

$L_k = L_i - L_e = \{c_0, h_0\}$

$L_m = L_k - L_j = \{h_0\}$

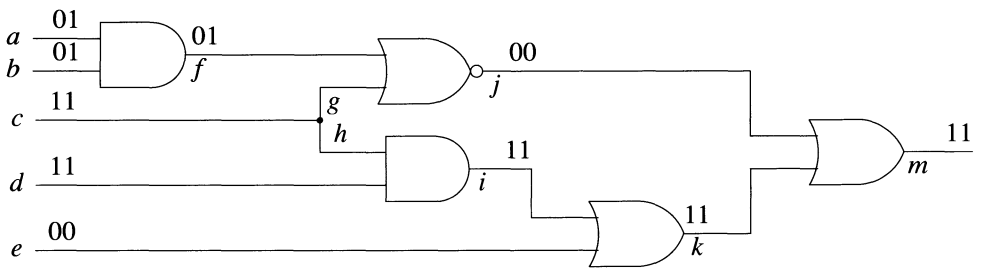


Figure 5.10

Since h_0 is detected, we drop it from the set of simulated faults by deleting h_0 from every fault list where it appears, namely L_h, L_i, L_k, and L_m. (Note that c_0 is not detected.)

Now assume that both inputs a and b change to 1. Then $L_a = \{a_0\}$, $L_b = \varnothing$, $f = 1$, and $L_f = \{a_0\}$. The evaluation of gate j generates no logic event, but now $L_j = L_f \cap L_g = \varnothing$. This shows that a list event may occur even without a corresponding logic event. Propagating this list event to gate m, we obtain $L_m = L_k - L_j = \{c_0\}$. Hence c_0 is now detected. □

Note that when L_α is computed, to determine whether a list event has occurred, the new L_α must be compared with the old L_α (denoted by $\tilde{L}_\alpha$), before the latter is destroyed; i.e., we must determine whether $L_\alpha = \tilde{L}_\alpha$.

Fault propagation becomes more involved when there is feedback. Care must be exercised when the effect of a fault, say α_0 or α_1, feeds back onto the line α itself. If the fault list propagating to line α contains an α_0 and if the good value of α is 0, then α_0 should be deleted from L_α because the values of α in the fault-free circuit and the circuit with the fault α_0 are the same. Similarly α_1 should be deleted from L_α if the good value of α is 1.

Additional complexities occur in propagation of fault lists through memory elements. Consider the SR latch shown in Figure 5.11. Let the state at time t_1 be $(y_1, y_2) = (0,1)$, and the input be $(S,R) = (1,1)$. If at time t_2 R changes to 0, the outputs should remain the same. Let L_S and L_R be the input fault lists at time t_2 associated with lines S and R, and let $\tilde{L}_1$ and $\tilde{L}_2$ be the fault lists associated with lines y_1 and y_2 at time t_1. The new fault lists L_1 and L_2 associated with lines y_1 and y_2 resulting from the input logic event at t_2 can be computed as follows (faults internal to the latch will be ignored).

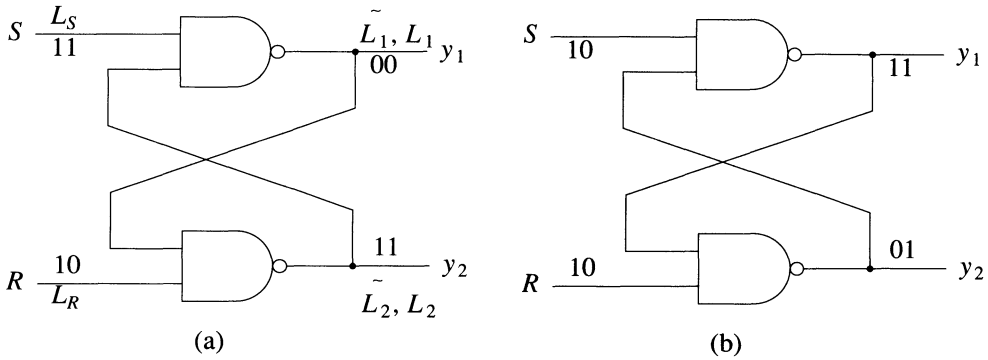


Figure 5.11 Propagation of fault lists through an *SR* latch (a) Good circuit (b) Faulty circuit for some fault f in the set $\{L_S \cap \bar{L}_R \cap \tilde{L}_1 \cap \tilde{L}_2\}$

Initially set $L_1 = \tilde{L}_1$ and $L_2 = \tilde{L}_2$. The new values for L_1 and L_2 are given by the expressions $L_1 = L_S \cup L_2$ and $L_2 = L_R \cap L_1$. Since a change in the fault lists for y_1 and y_2 may have occurred, this calculation must be repeated until the fault lists stabilize. Under some conditions, such as a critical race, the lists may not stabilize and special processing is required. This problem is dealt with later in this section.

A second and faster approach to propagation of fault lists through a latch is to consider this element to be a primitive and calculate the rules for determining the steady-state fault lists for y_1 and y_2 in terms of L_S, L_R, $\tilde{L}_1$, and $\tilde{L}_2$. These rules can be derived by inspecting all 16 minterms over the variables L_S, L_R, $\tilde{L}_1$, and $\tilde{L}_2$, i.e., $(L_S \cap L_R \cap \tilde{L}_1 \cap \tilde{L}_2)$, $(L_S \cap L_R \cap \tilde{L}_1 \cap \bar{\tilde{L}}_2)$, ..., $(\bar{L}_S \cap \bar{L}_R \cap \bar{\tilde{L}}_1 \cap \bar{\tilde{L}}_2)$.

For example, consider again the initial state condition $(S,R,y_1,y_2) = (1,1,0,1)$. Any fault f in the set (minterm) $L_S \cap \bar{L}_R \cap \tilde{L}_1 \cap \tilde{L}_2$ causes both outputs to be incorrect at time t_1 (due to $\tilde{L}_1$ and $\tilde{L}_2$), and input S to be incorrect at time t_2 (see Figure 5.11(b)). Hence at t_2 we have that $(S,R) = (0,0)$ in the faulty circuit, and this fault produces $(y_1,y_2) = (1,1)$ and hence is an element in L_1 but not L_2 (note that $y_1 \neq \bar{y}_2$, hence the outputs are not labeled y_1 and $\bar{y}_1$). Similarly, each of the remaining 15 minterms can be processed and the rules for calculating L_1 and L_2 developed. These rules become more complex if internal faults are considered or if an internal fault propagates around a loop and appears in L_S or L_R. Finally, the rules for L_1 and L_2 must be developed for each initial state vector (S,R,y_1,y_2).

Let z be the output of a combinational block implementing the function $f(a,b,c,...)$. A general formula for computing the fault list L_z as a function of the variables $a,b,c,...$ and their fault lists $L_a,L_b,L_c,...$ was obtained by Levendel [1980]. Let us define an exclusive-OR operation between a variable x and its fault list L_x by

$$x \oplus L_x = \begin{cases} L_x & \text{if } x = 0 \\ \bar{L}_x & \text{if } x = 1 \end{cases}$$

Since L_x is the list of faults that cause x to have a value different from its fault-free value, it follows that $x \oplus L_x$ is the list of faults that cause x to take value 1. Let us denote by $F(A,B,C,...)$ the set function obtained by replacing all AND and OR operations in $f(a,b,c,...)$ by $\cap$ and $\cup$, respectively. Then the list of faults that cause z to take value 1 (ignoring z s-a-1) is given by

$$z \oplus L_z = F(a \oplus L_a,\ b \oplus L_b,\ c \oplus L_c, ...)$$

(see Problem 5.6). Hence

$$L_z = f(a,b,c,...) \oplus F(a \oplus L_a,\ b \oplus L_b,\ c \oplus L_c, ...).$$

For example, let us apply the above equation to compute the output fault list of a JK flip-flop modeled by its characteristic equation $Y = J\bar{y} + \bar{K}y$, where Y and y are the new and the current output values. The new output fault list L_y is given by

$$L_y = (J\bar{y} + \bar{K}y) \oplus [(J \oplus L_J) \cap (\overline{Y \oplus L_y}) \cup (\overline{K \oplus L_K}) \cap (y \oplus L_y)]$$

For $J = 0$, $K = 0$, and $y = 0$, L_Y becomes

$$L_Y(J = 0,\ K = 0,\ y = 0) = (L_J \cap \bar{L}_y) \cup (\bar{L}_K \cap L_y).$$

The fault list of the complementary output $\bar{Y}$ is $L_{\bar{Y}} = L_Y$.

We shall now consider several aspects related to the efficiency of deductive simulators.

Fault Storage and Processing

Fault lists can be stored as linked lists, sequential tables, or characteristic vectors. These three structures are illustrated in Figure 5.12. For a list structure, the faults are stored in an ordered sequence to simplify the computation of set union and intersection. List structures require the overhead of an available space list, as well as a pointer to find the next element in a list. However, insertion and deletion of elements is a simple task. Lists can be easily destroyed by assigning them to the available space list. Using a sequential table organization leads to faster processing and eliminates the need for pointers. However, repacking of storage is required occasionally to make a section of memory reusable.

The computation of set union and intersection for unordered and ordered sequential tables is about of the same complexity. It is easier to add a single element to an unordered set, since it can be placed at the end of the table. In an ordered table it must be inserted in its correct position, and all entries below it repositioned in memory. It is easier to delete an arbitrary element in an ordered table than in an unordered one, since its position can be found by a binary search process.

Using the characteristic-vector structure, all list operations are simple and fast. A fault is inserted or deleted by storing a 1 or 0 in the appropriate bit. Set union and intersection are carried out by simple OR and AND word operations. However, this structure typically requires more storage than the preceding two, though it is of fixed size. An exact comparison of the storage requirements requires knowledge of the average size of a fault list.

For large circuits, it is possible to run out of memory while processing the fault lists, which are dynamic and unpredictable in size. When this occurs, the set of faults must

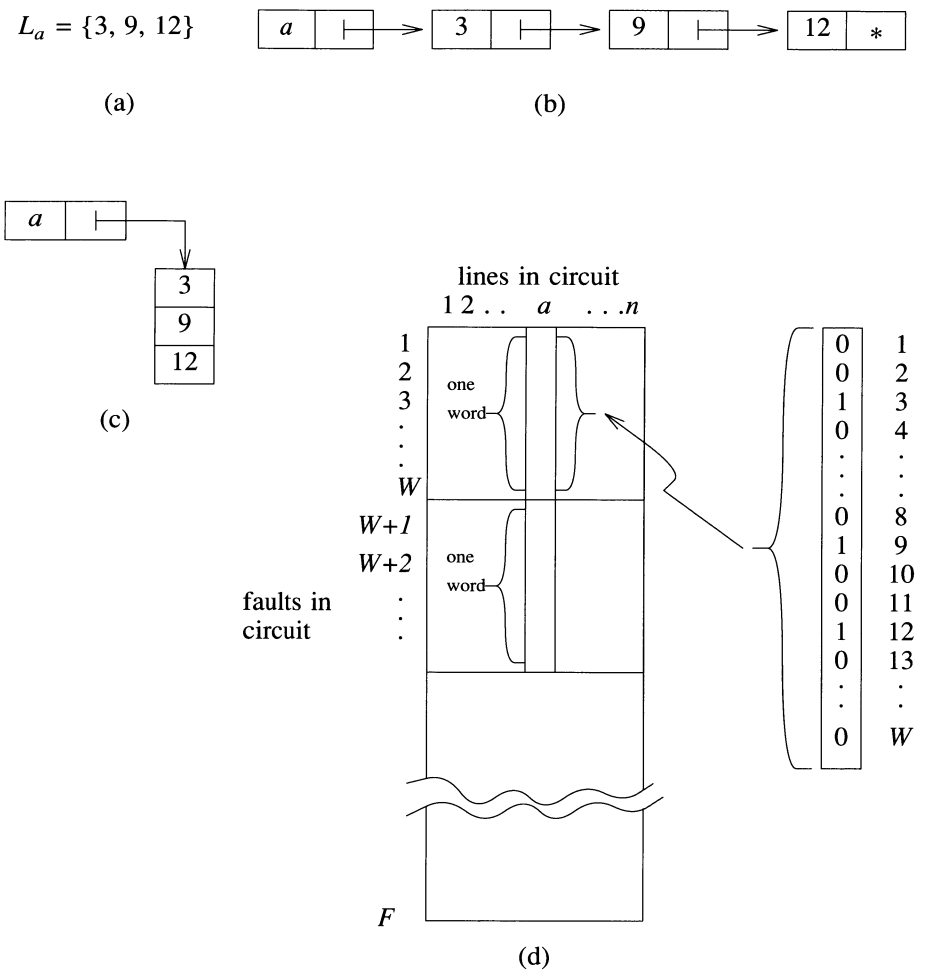


Figure 5.12 Three storage structures for lists (a) Fault list (b) Linked list (c) Sequential table (d) Characteristic vector

be partitioned and each subset processed separately. This process can be done dynamically and leads to a multipass simulation process.

Oscillation and Active Faults

It is possible that the fault-free circuit will stabilize while the circuit with some fault f oscillates. For this case there will be an arbitrarily long sequence of list events. Unfortunately, even though only one fault may be causing the oscillation, repeated complex processing of long lists may be required, which is time-consuming. Faults that produce circuit oscillation, as well as those that achieve stability only after a great amount of logical activity, should be purged from fault lists whenever possible.

5.2.4.2 Three-Valued Deductive Simulation

When 3-valued logic is employed, the complexity of deductive simulation greatly increases. In this section we will briefly outline two approaches of varying degree of accuracy (pessimism). We refer to these as third-order analysis (least pessimistic) and second-order analysis (most pessimistic).

Third-Order Analysis

Since each line α in the fault-free and faulty circuits can take on the logic values 0,1 and u, two lists L_α^δ and L_α^ε will be associated with a line α whose normal value is v, where $\{\delta,\varepsilon\} = \{0,1,u\} - \{v\}$. For example, if $v = 1$, then the lists L_α^0 and L_α^u are associated with line α; L_α^0 (L_α^u) represents the set of all faults that cause α to have value $0(u)$. Since the set of all faults currently processed equals the set $L_\alpha^0 \cup L_\alpha^1 \cup L_\alpha^u$, we have that $L_\alpha^v = \overline{L_\alpha^\varepsilon \cup L_\alpha^\delta}$.

Example 5.2: Consider the gate and input values shown in Figure 5.13. A fault that changes d to 0 must change a and b to 1, and it must not change c. Hence

$$L_d^0 = (L_a^1 \cap L_b^1 \cap (\overline{L_c^0 \cup L_c^u})) \cup d_0$$

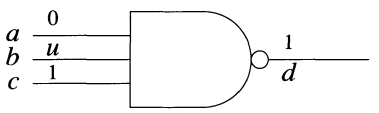


Figure 5.13

A fault that changes d to u must cause a, b, and c to have any combination of u and 1 values, except $a = b = c = 1$. Then

$$L_d^u = (L_a^u \cup L_a^1) \cap \overline{L_b^0} \cap \overline{L_c^0} - L_a^1 \cap L_b^1 \cap \overline{L_c^0 \cup L_c^u}$$

□

The disadvantage of this approach is that two lists exist for each line, and the complexity of the processing required to propagate the fault lists through an element is more than doubled.

Second-Order Analysis

To reduce computational complexity, we can associate a single fault list L_α with each line α. If the value of line α in the fault-free circuit is u, then $L_\alpha = \varnothing$. This means that if the value of the line is u, we will make no predictions on the value of this line in any of the faulty circuits. Hence some information is lost, and incorrect initialization of some faulty circuits is possible. If f is a fault that produces a u on a line α whose fault-free value is 0 or 1, then the entry in L_α corresponding to fault f is flagged (denoted by $*f$) and is called a *star fault*. This means that we do not know whether f is in L_α. If α is a primary output, then f is a *potentially detected* fault.

The rules for set operations for star faults are given in the table of Figure 5.14; here λ denotes an entry different from f or $*f$.

A	B	$A \cup B$	$A \cap B$	$A - B$	$B - A$
$*f$	λ	$\{*f,\lambda\}$	$\varnothing$	$*f$	λ
$*f$	f	f	$*f$	$\varnothing$	$*f$
$*f$	$*f$	$*f$	$*f$	$*f$	$*f$

Figure 5.14 Operations with star faults

Whenever a race or oscillation condition caused by fault f is identified on line α, f is entered as a star fault on the associated list. If it is decided that line α is oscillating, then those faults causing oscillation are precisely those entries in the set $(\tilde{L}_\alpha - L_\alpha) \cup (L_\alpha - \tilde{L}_\alpha)$ where $\tilde{L}_\alpha$ and L_α are the old and new fault lists. By changing these entries to star faults, the simulation can be continued and the simulation oscillation should cease [Chappell *et al.* 1974].

An interesting open problem is: What should be the initial contents of the fault lists? It is often not known how the initial state of the circuit was arrived at, and therefore it is not apparent whether a fault f would have influenced the initializing of line α. It is erroneous to set all fault lists initially to the empty set, and it is too pessimistic to place all faults into the set L_α^u. The former approach, however, is usually taken.

A more detailed discussion of multiple logic values in deductive simulation can be found in [Levendel and Menon 1980].

Limitations

The propagation of fault lists through an element is based on its Boolean equations. Thus *deductive fault simulation is compatible only in part with functional-level modeling*; namely it is applicable only to models using Boolean equations [Menon and Chappell 1978]. In principle, it can be extended to multiple logic values by increasing the number of fault lists associated with a line [Levendel and Menon 1980], but the corresponding increase in the complexity of the algorithm renders this approach impractical. Hence *deductive simulation is limited in practice to two or three logic values.*

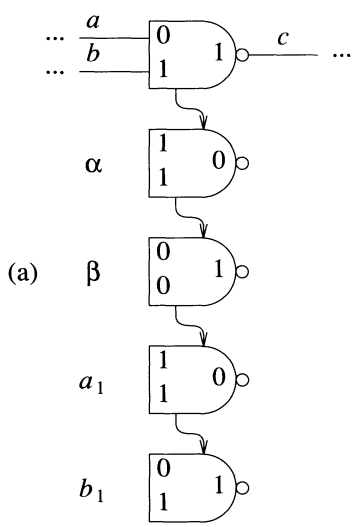
The fault-list propagation mechanism cannot take full advantage of the concept of activity-directed simulation. For example, suppose that at a certain time we have activity only in one faulty circuit. Propagating this list event may generate many long fault-list computations. But most of the fault-list entries involved in this computation correspond to faulty circuits without activity at that time. This is especially time-consuming when a faulty circuit oscillates.

5.2.5 Concurrent Fault Simulation

Concurrent fault simulation [Ulrich and Baker 1974] is based on the observation that most of the time during simulation, most of the values in most of the faulty circuits agree with their corresponding values in the good circuit. The concurrent method simulates the good circuit N, and for every faulty circuit N_f, it simulates only those elements in N_f that are different from the corresponding ones in N. These differences

are maintained for every element x in N in the form of a **concurrent fault list**, denoted by CL_x. Let x_f denote the replica of x in the circuit N_f. Let V_x (V_{x_f}) denote the ensemble of input, output, and (possibly) internal state values of x (x_f). During simulation, CL_x represents the set of all elements x_f that are different from x at the current simulated time. Elements x and x_f may differ in two ways. First, we may have $V_{x_f} \neq V_x$; this occurs when a fault effect caused by f has propagated to an input/output line or state variable of x_f. Second, f can be a *local fault* of x_f, that is, a fault inserted on an input/output line or state variable of x_f. A local fault f makes x_f different from x, even if $V_{x_f} = V_x$; this occurs when the input sequence applied so far does not activate f.

An entry in CL_x has the form (f,V_{x_f}). Figure 5.15(a) illustrates a concurrent fault list in pictorial form. The gates "hanging" from the good gate c are replicas of c in the faulty circuits with the faults α, β, a_1, and b_1. Here α and β are faults whose effects propagate to gate c; they cause, respectively, a=1 and b=0. Faults a_1 and b_1 are local faults of gate c. Note that b_1 appears in CL_c even if the values of a, b, and c in the presence of b_1 are the same as in the good circuit. Figure 5.15(b) shows CL_c in tabular form. By contrast, the fault list of c in deductive simulation is $L_c = \{\alpha, a_1\}$.



f	a	b	c
α	1	1	0
β	0	0	1
a_1	1	1	0
b_1	0	1	1

(b)

Figure 5.15 Concurrent fault list for gate c (a) Pictorial representation (b) Tabular representation

A fault f is said to be *visible* on a line i when the values of i in N and N_f are different. Among the entries in the concurrent fault list of a gate x, only those corresponding to faults visible on its output appear also in the deductive fault list L_x. (In Figure 5.15, α and a_1 are visible faults.) In this sense, a deductive fault list is a subset of the corresponding concurrent fault list. Thus concurrent simulation requires more storage than deductive simulation.

Most of the work in concurrent simulation involves updating the dynamic data structures that represent the concurrent fault lists. Fault insertion takes place during the initialization phase (after fault collapsing); the initial content of every list CL_x consists of entries corresponding to the local faults of x. A local fault of x remains in CL_x until it is dropped. During simulation, new entries in CL_x represent elements x_f, whose values become different from the values of x; these are said to *diverge from x*. Conversely, entries removed from CL_x represent elements x_f, whose values become identical to those of x; these are said to *converge to x*. A fault is dropped by removing its entries from every list where it appears.

The following example illustrates how the concurrent fault lists change during simulation. For simplicity, we will refer to the circuit with the fault f as "circuit f."

Example 5.3: Figure 5.16(a) shows logic values and concurrent fault lists for a circuit in a stable state. Now assume that the primary input a changes from 1 to 0. This event occurs not only in the good circuit but (implicitly) also in all the faulty circuits that do not have entries in CL_c; the entries in CL_c have to be separately analyzed. In the good circuit, c is scheduled to change from 0 to 1. The event on a does not propagate in the circuit a_1; hence that gate is not evaluated. Evaluating gate c_α does not result in any event.

When the value of c is updated, the fault a_1 becomes visible. In the good circuit, the change of c propagates to the gate e. At the same time, we propagate a "list event" to indicate that a_1 is a newly visible fault on line c (see Figure 5.16(b)).

The evaluation of gate e in the good circuit does not produce an output event. An entry for the newly visible fault a_1 is added to CL_e. The other entries in CL_e are analyzed as follows (see Figure 5.16(c)):

- c does not change in the circuit c_1; the entry for c_1 remains in the list because c_1 is a local fault of gate e.
- Gate e in the circuit d_1 is evaluated and generates a 1/0 event; the same event occurs in the circuit β.
- c does not change in the circuit α; the entry for α is deleted from the list. □

An important feature of the concurrent fault simulation mechanism is that it individually evaluates elements in both the good and the faulty circuits. For evaluation, an entry in a fault list denoting a replica of a gate in the good circuit is just a gate with a different set of input/output values.

A line i in a faulty circuit may change even if i is stable in the good circuit. This is illustrated in Figure 5.16(c), where line e changes to 0 in circuits d_1 and β but remains stable at 1 in the good circuit. Figure 5.17 shows that a line i in the good circuit and some faulty circuits may also have simultaneous but different events. An event generated by an element in a faulty circuit propagates only inside that circuit.

We shall describe the concurrent simulation mechanism by reference to Figure 5.18. Line i is an output of A, and B is one of its fanouts. At a certain simulation time we may have an event on a line i in the good circuit and/or in several faulty circuits. The set of the simultaneous events occurring on the same line i is called a *composed event* and has the form (i,L), where L is a list of pairs (f,v_f'), in which f is a fault name

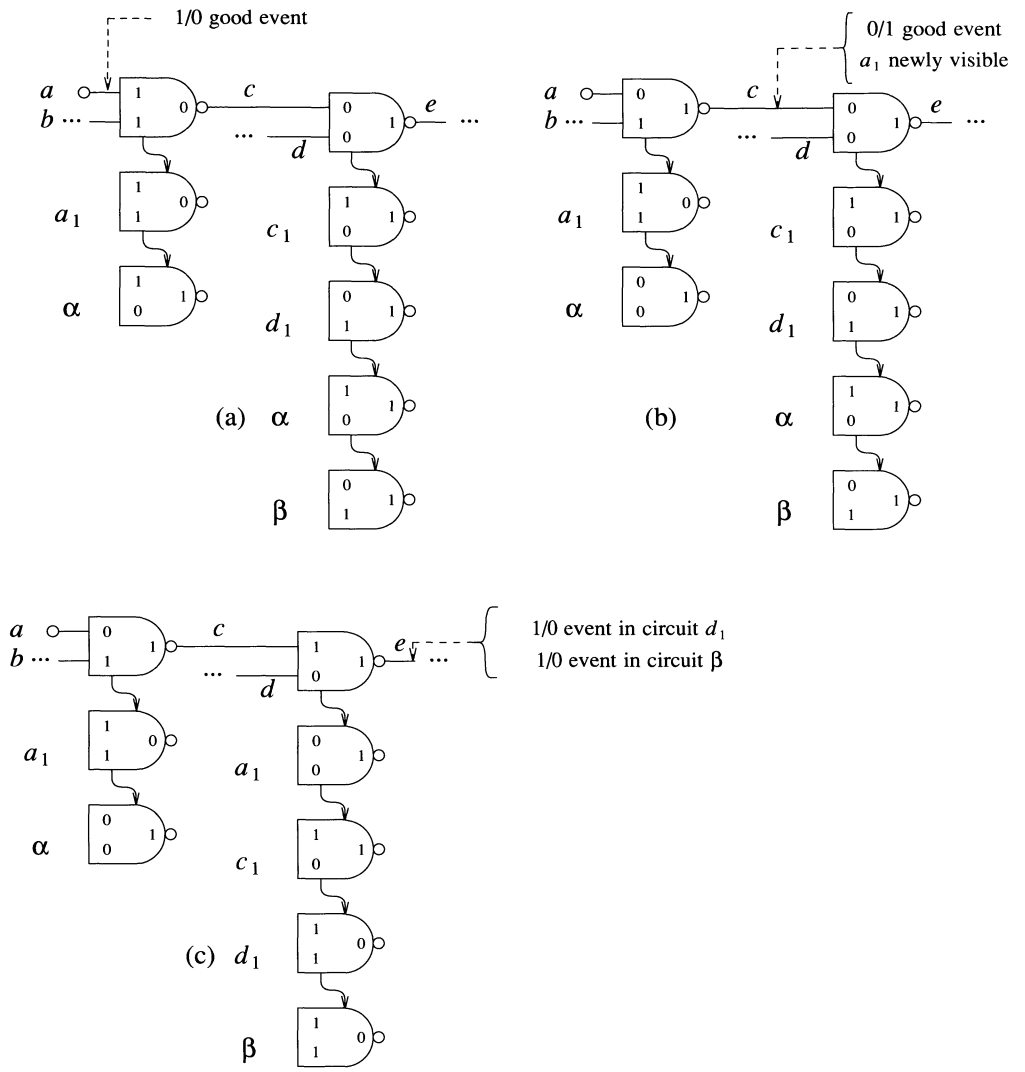


Figure 5.16 Changes in fault lists during simulation

(index) and v_f' is the scheduled value of line i in the circuit f. (The good circuit has f=0). The good-circuit event also occurs (implicitly) in all the faulty circuits that do *not* have an entry in CL_A.

The overall flow of event-directed logic simulation shown in Figure 3.12 is valid for concurrent fault simulation with the understanding that the events processed are composed events. Suppose that a composed event (i,L) has just been retrieved from the event list. First we update the values and the concurrent fault list of the source element A where the event originated. Then we update the values and the concurrent

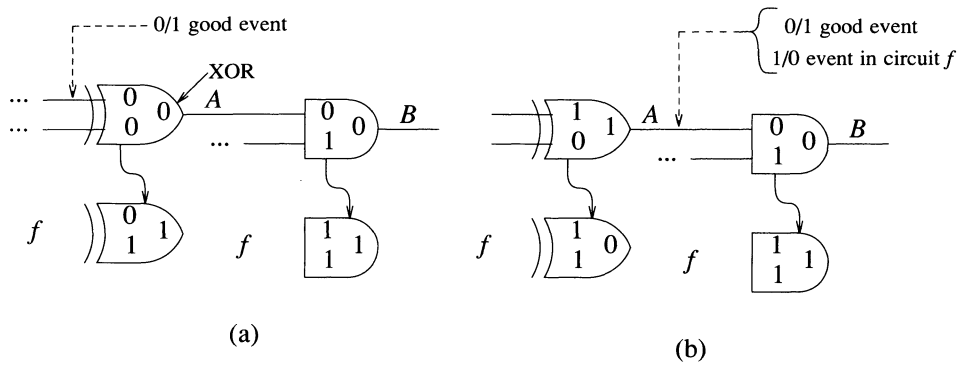


Figure 5.17

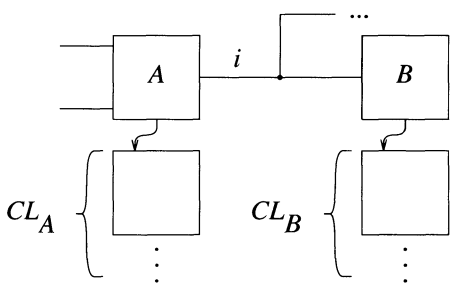


Figure 5.18

fault list of every element B on the fanout list of i and evaluate the activated elements (in the good and faulty circuits).

Figure 5.19 outlines the processing of the composed event at the source element A. Let v (v_f) be the current value of line i in the good circuit (in the circuit f). If an event v/v' occurs in the good circuit, then we have to analyze every entry in CL_A; otherwise we analyze only those entries with independent events. In the former case, if in a circuit f line i stays at value $v_f = v$ (i.e., the value of i in the circuit f is the same as the value of i in the good circuit before the change), then f is a newly visible fault on line i; these faults are collected into a list NV. The values of every analyzed entry f in CL_A, except for the newly visible ones, are compared to the values of the good element A, and if they agree, f is deleted from CL_A. (Practical implementation of the processing described in Figure 5.19 is helped by maintaining entries in concurrent fault lists and in lists of events ordered by their fault index.)

```
NV = ∅
if i changes in the good circuit then
  begin
    set i to v' in the good circuit
    for every f ∈ CL_A
        begin
          if f ∈ L then
              begin
                set i to v_f' in circuit f
                if V_{A_f} = V_A then delete f from CL_A
              end
          else /* no event in circuit f */
              if v_f = v then add newly visible fault f to NV
              else if V_{A_f} = V_A then delete f from CL_A
        end
  end
else /* no good event for i */
  for every f ∈ L
    begin
        set i to v_f' in circuit f
        if V_{A_f} = V_A then delete f from CL_A
    end
```

Figure 5.19 Processing of a composed event (i, L) at the source element A

Next, the composed event (i,L), together with the list NV of newly visible faults on line i, is propagated to every fanout element B. If a good event exists, then it activates B (for simplicity, we assume a two-pass strategy; thus evaluations are done after all the activated elements are determined). The processing of an element B_f depends on which lists (CL_B, L, NV) contain f. The way NV is constructed (see Figure 5.19) implies that f cannot belong to both NV and L. The different possible cases are labeled 1 through 5 in the Karnaugh map shown in Figure 5.20. The corresponding actions are as follows:

Case 1: (B_f exists in CL_B and no independent event on i occurs in N_f.) If a good event exists and it can propagate in N_f, then activate B_f. The good event on line i can propagate in the circuit f if $v_f = v$ and f is not the local *s-a-v* fault on the input i of B_f. For example, in Figure 5.16(b) the change of c from 0 to 1 propagates in the circuits d_1 and β but not in the circuits c_1 and α.

Case 2: (B_f exists in CL_B and an independent event on i occurs in N_f.) Activate B_f. Here we have independent activity in a faulty circuit; this is illustrated in Figure 5.17(b), where the event 1/0 activates the gate B_f.

Case 3: (An independent event on i occurs in N_f, but f does not appear in CL_B.) Add an entry for f to CL_B and activate B_f. This is shown in Figure 5.21. Here A and A_f

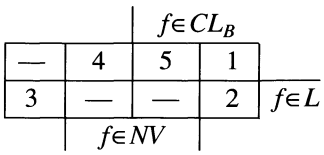


Figure 5.20 Possible cases in processing a composed event propagated to a fanout element B

have been evaluated because the input b changed from 0 to 1, and now i changes from 0 to 1 in the good circuit and from 0 to u in the circuit f. An entry for f is added to CL_B (with the same values as B) and activated.

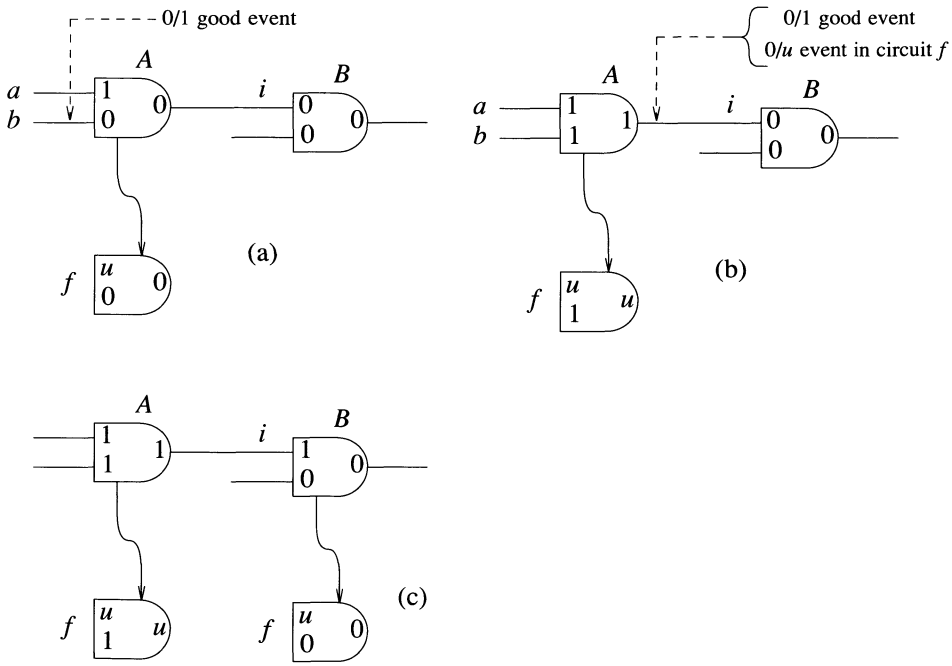


Figure 5.21

Case 4: (f is newly visible on line i and does not appear in CL_B.) Add an entry for f to CL_B. This is illustrated in Figure 5.16 by the addition of a_1 to CL_e.

Case 5: (f is a newly visible fault on line i, but an entry for f is already present in CL_B.) No action. In a combinational circuit this may occur only when there is reconvergent fanout from the origin of the fault f to the element B. Figure 5.22 provides such an example.

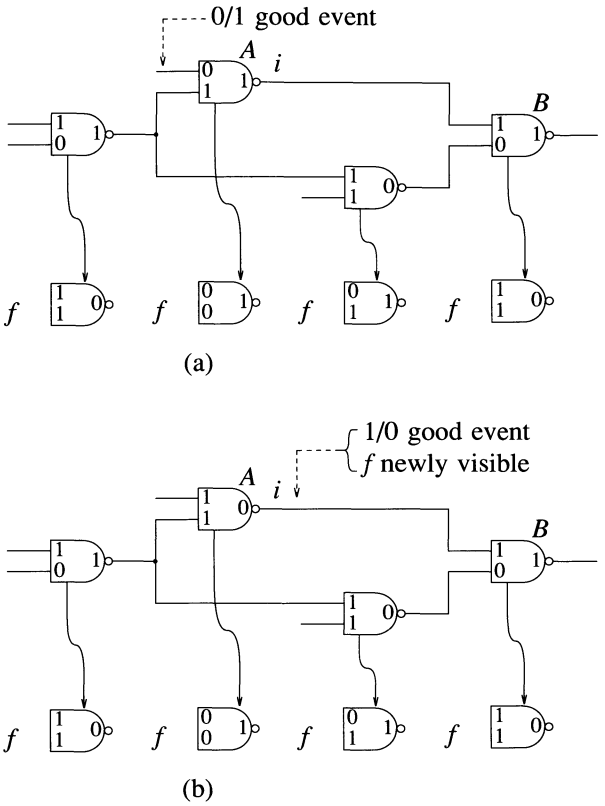


Figure 5.22

The activated elements are individually evaluated and resulting events for their outputs are merged into composed events. If a good event is generated, then all the comparisons for deletions from the fault list will be done when the composed event will have been retrieved from the event list (see Figure 5.19). If no good event is generated, however, any activated element B_f that does *not* generate an event should be immediately compared to the good element (they are both stable), and if their values agree, f should be deleted from CL_B.

Since the concurrent fault simulation is based on evaluation of individual elements, it is directly applicable to functional-level modeling [Abramovici *et al.* 1977]. In functional simulation, an event may also denote a change of an internal state variable of a functional element. Such a state event has the form (M, i, v'), where M is the element whose state variable i is scheduled to get value v'. In concurrent simulation we propagate *composed state events* of the form (M, i, L), where L has the same form as for line events. A composed state event is processed like a composed line event, with the only difference that both the source (A) and the fanout (B) of the event are the same element M.

Our definition of a concurrent fault list CL_x implies that for any entry x_f we store all the values of its inputs, outputs, and state variables. But this would waste large amounts of memory for a functional element x with a large number of state variables, because most of the time during simulation, most of the values of the elements in CL_x would be the same as the good values. Then it is preferable to maintain only the differences between the values of every x_f in the fault list and the good element values. (This represents a second-level application of the principle of concurrent simulation!) Memories are typical examples of elements of this category. We can view a memory as being "composed" of addressable words, and we can keep associated with every good word a concurrent list of faults that affect the contents of that word [Schuler and Cleghorn 1977].

5.2.6 Comparison

In this section we compare the parallel, the deductive, and the concurrent techniques with respect to the following criteria:

- capability of handling multiple logic values,
- compatibility with functional-level modeling,
- ability to process different delay models,
- speed,
- storage requirements.

While parallel and deductive simulation are reasonably efficient for two logic values, the use of three logic values (0, 1, u) significantly increases their computational requirements, and they do not appear to be practical for more than three logic values. Both methods are compatible only in part with functional-level modeling, as they can process only components that can be entirely described by Boolean equations.

In both parallel and deductive simulation, the basic data structures and algorithms are strongly dependent on the number of logic values used in modeling. By contrast, the concurrent method provides only a mechanism to represent and maintain the differences between the good circuit and a set of faulty circuits, and this mechanism is independent of the way circuits are simulated. Thus concurrent simulation is transparent to the system of logic values used and does not impose any limitation on their number. Being based on evaluation of individual elements, concurrent simulation does not constrain the modeling of elements, and hence it is totally compatible with functional-level modeling and can support mixed-level and hierarchical modeling [Rogers *et al.* 1987]. It has also been applied to transistor-level circuits [Bose *et al.* 1982, Bryant and Schuster 1983, Saab and Hajj 1984, Kawai and Hayes 1984, Lo *et al.* 1987].

We have described the three methods by assuming a transition-independent nominal delay model. The use of more accurate timing models — such as rise and fall delays, inertial delays, etc. — leads to severe complications in the parallel and deductive methods. In contrast, because of its clean separation between fault processing and element evaluation, a concurrent simulator can be as sophisticated as the underlying good-circuit simulator. This means that detailed delay models can be included in concurrent simulation without complications caused by the fault-processing mechanism.

Theoretical analysis done by Goel [1980] shows that for a large combinational circuit of n gates, the run times of the deductive and parallel methods are proportional to n^2 and n^3 respectively. It is unlikely that any fault simulation algorithm can have worst-case linear complexity [Harel and Krishnamurthy 1987]. Experimental results presented in [Chang *et al.* 1974] show that (using 2-valued logic) deductive simulation is faster than parallel simulation for most sequential circuits, except small (n<500) ones.

Although no direct comparisons between the deductive and concurrent methods have been reported, qualitative arguments suggest that concurrent simulation is faster. One significant difference between the two methods is that a concurrent simulator processes only the active faulty circuits. This difference becomes more apparent when changes occur only in faulty circuits and especially when a faulty circuit oscillates. The reason is that a deductive simulator always recomputes a complete fault list (even to add or to delete a single entry), while a concurrent simulator propagates changes only in the active circuit. Another difference is that a concurrent simulator, being based on evaluation of individual elements, can make use of fast evaluation techniques, such as table look-up or input counting.

The main disadvantage of concurrent simulation is that it requires more memory than the other methods. The reason is that the values V_{x_f} of every element x_f in a concurrent fault list must be available for the individual evaluation of x_f. However, this disadvantage can be overcome by partitioning the set of faults for multipass simulation. Even if unlimited memory is available, partitioning the set of faults improves the efficiency of concurrent simulation by limiting the average size of the concurrent fault list [Henckels *et al.* 1980]. The most important advantages of concurrent simulation are its compatibility with different levels of modeling and its ability to process multiple logic values; these factors make it suitable for increasingly complex circuits and evolving technology.

The *parallel-value list* (PVL) is a fault simulation method [Moorby 1983, Son 1985] which combines several features of the parallel, deductive, and concurrent techniques. Consider the fault list representation based on a characteristic-vector structure, illustrated in Figure 5.12(d). Assume that the faulty values of every line are stored in another vector parallel to the characteristic vector (or in several parallel vectors for multivalued logic). The fault set is partitioned into groups of W faults. When all the characteristic-vector values in a group are 0, all the faulty values are the same as the good value and are not explicitly represented. The remaining groups, in which at least one faulty value is different, are maintained in a linked list, similar to a concurrent fault list. Given the groups in fault lists of the inputs of a device, the groups of the output fault list are determined by set-union and set-intersection operations similar to those used in deductive simulation. The computation of faulty values for the output groups proceeds as in parallel simulation. The PVL method requires less storage than concurrent simulation.

5.3 Fault Simulation for Combinational Circuits

Specialized fault simulation methods for combinational circuits are justified by the widespread use of design for testability techniques (to be described in a separate chapter) that transform a sequential circuit into a combinational one for testing

purposes. For static testing, in which the basic step consists of applying an input vector and observing the results after the circuit has stabilized, we are interested only in the final (stable) values. In a combinational circuit the final values of the outputs do not depend on the order in which inputs or internal lines change. Thus a fault simulation method for combinational circuits can use a simplified (zero or unit) delay model.

5.3.1 Parallel-Pattern Single-Fault Propagation

The *Parallel-Pattern Single-Fault Propagation* (PPSFP) method [Waicukauski *et al.* 1985] combines two separate concepts — single-fault propagation and parallel-pattern evaluation.

Single-fault propagation is a serial fault simulation method specialized for combinational circuits. After a vector has been simulated using a fault-free simulator, SSFs are inserted one at a time. The values in every faulty circuit are computed by the same fault-free simulation algorithm and compared to their corresponding good values. The computation of faulty values starts at the site of the fault and continues until all faulty values become identical to the good values or the fault is detected.

Parallel-pattern evaluation is a simulation technique, introduced in Section 3.5, which simulates W vectors concurrently. For 2-valued logic, the values of a signal in W vectors are stored in a W-bit memory location. Evaluating gates by Boolean instructions operating on W-bit operands generates output values for W vectors in parallel. Of course, this is valid only in combinational circuits, where the order in which vectors are applied is not relevant.

A simulator working in this way cannot be event-driven, because events may occur only in some of the W vectors simulated in parallel. This implies that all the gates in the circuit should be evaluated in every vector, in the order of their level (for this we can use a compiled simulation technique). Let $a < 1$ denote the average activity in the circuit, that is, the average fraction of the gates that have events on their inputs in one vector. Then parallel-pattern evaluation will simulate $1/a$ more gates than an event-driven simulator. However, because W vectors are simultaneously simulated, parallel-pattern evaluation is more efficient if $W > 1/a$. For example, $W \geq 20$ will compensate for a value a of 5 percent. The overall speed-up is Wa. The implementation described in [Waicukauski *et al.* 1985] uses $W = 256$.

The PPSFP method combines single-fault propagation with parallel-pattern evaluation. First it does a parallel fault-free simulation of a group of W vectors. Then the remaining undetected faults are serially injected and faulty values are computed in parallel for the same set of vectors. Comparisons between good and faulty values involve W bits. The propagation of fault effects continues as long as faulty values are different from the good values in at least one vector. Detected faults are dropped and the above steps are repeated until all vectors are simulated or all faults are detected.

The PPSFP method has been successfully used to evaluate large sets (containing 1/2 million) of random vectors. However, it cannot support an algorithmic test generation process of the type illustrated in Figure 5.3, in which we need to know the faults detected by one vector before we generate the next one.

5.3.2 Critical Path Tracing

In this section we present the main concepts of the method called *critical path tracing* [Abramovici *et al.* 1984], which includes and extends features of earlier fault simulation techniques for combinational circuits [Roth *et al.* 1967, Ozguner *et al.* 1979, Su and Cho 1972, Hong 1978].

For every input vector, critical path tracing first simulates the fault-free circuit, then it determines the detected faults by ascertaining which signal values are *critical.*

Definition 5.1: A line l has a *critical value* v in the test (vector) t iff t detects the fault l *s-a-*$\bar{v}$. A line with a critical value in t is said to be *critical* in t.

Finding the lines critical in a test t, we immediately know the faults detected by t. Clearly, the primary outputs are critical in any test. (We assume completely specified tests; hence all values are binary.) The other critical lines are found by a backtracing process starting at the primary outputs. This process determines paths composed of critical lines, called *critical paths.* It uses the concept of *sensitive inputs.*

Definition 5.2: A gate input is *sensitive* (in a test t) if complementing its value changes the value of the gate output.

The sensitive inputs of a gate with two or more inputs are easily determined as follows:

1. If only one input j has the controlling value of the gate (c), then j is sensitive.
2. If all inputs have value $\bar{c}$, then all inputs are sensitive.
3. Otherwise no input is sensitive.

The sensitive inputs of a gate can be easily identified during the fault-free simulation of the circuit, as scanning for inputs with controlling value is an inherent part of gate evaluation.

The following lemma provides the basis of the critical path tracing algorithm.

Lemma 5.1: If a gate output is critical, then its sensitive inputs, if any, are also critical.

The next example illustrates critical path tracing in a fanout-free circuit.

Example 5.4: Figure 5.23(a) shows a circuit, its line values for the given test, and the sensitive gate inputs (marked by dots). Critical path tracing starts by marking the primary output as critical. The other critical lines are identified by recursive applications of Lemma 5.1. Figure 5.23(b) shows the critical paths as heavy lines. □

It is important to observe that critical path tracing has completely avoided the areas of the circuit bordered by B and C, because by working backward from the output, it first determined that B and C are not critical. In contrast, a conventional fault simulator would propagate the effects of all faults in these areas (to B and C), before discovering (at the output gate) that these faults are not detected.

For a fanout-free circuit, critical path tracing can be done by a simple depth-first tree-traversal procedure that marks as critical and recursively follows in turn every sensitive input of a gate with critical output. The next example illustrates the problem

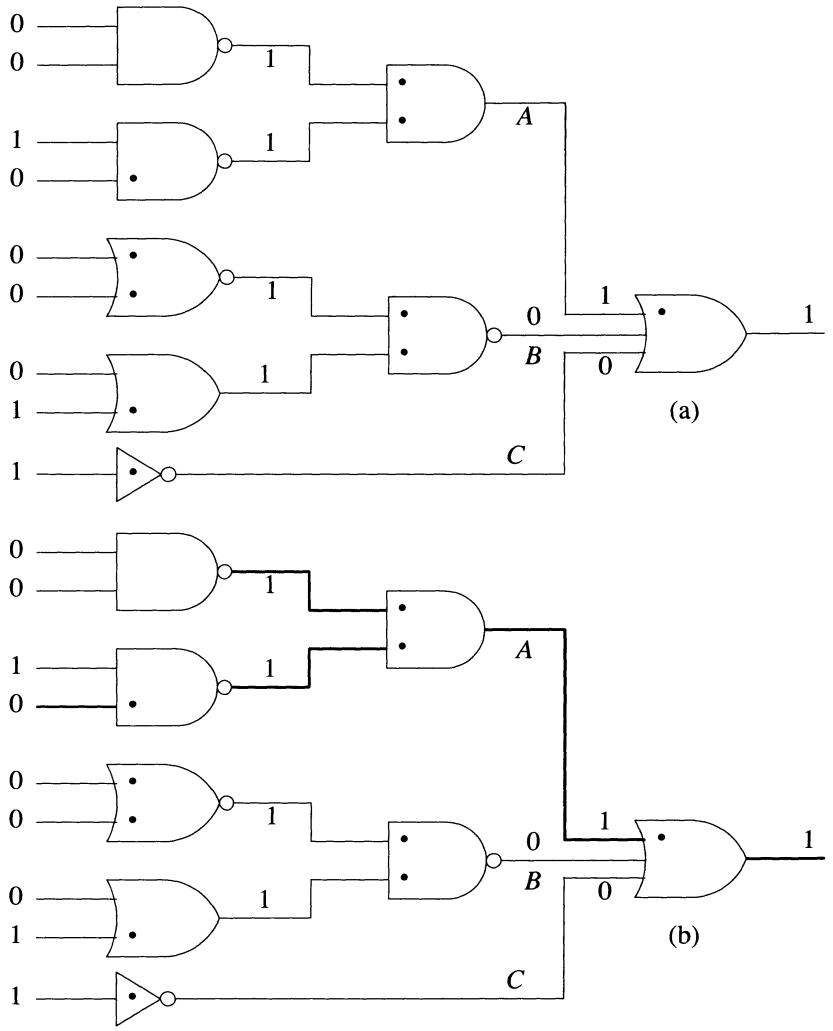


Figure 5.23 Example of critical path tracing in a fanout-free circuit

that appears in extending critical path tracing to the general case of circuits with reconvergent fanout.

Example 5.5: For the circuit and the test given in Figure 5.24(a), we start at the primary output, and by repeatedly using Lemma 5.1, we identify *F, D, A,* and *B1* as critical. We cannot, however, determine whether the stem *B* is critical without additional analysis. Indeed, the effects of the fault *B s-a-0* propagate on two paths with different inversion parities such that they cancel each other when they reconverge at gate *F*. This phenomenon, referred to as *self-masking*, does not occur for the test shown in Figure 5.24(b), because the propagation of the fault effect along the path starting at *B2* stops at gate *E*. Here *B* is critical. □

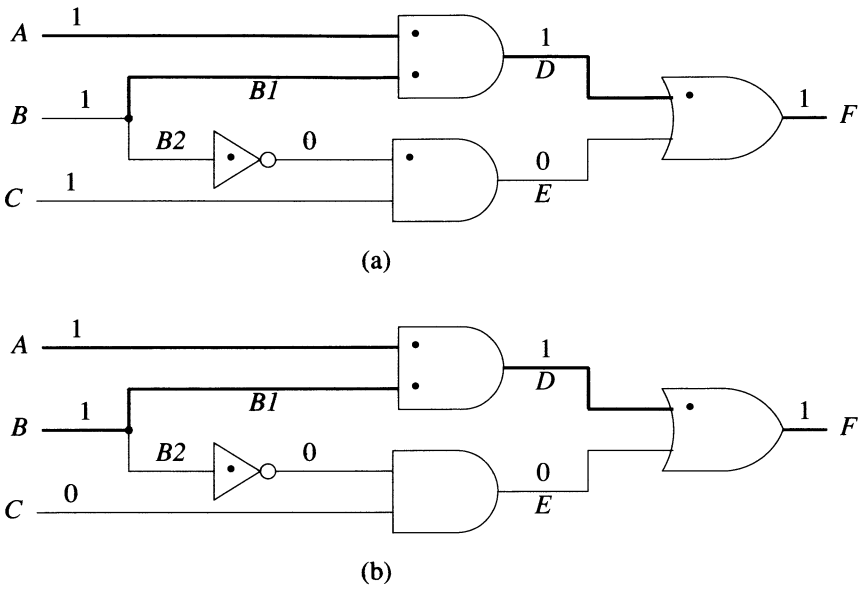


Figure 5.24 Example of stem analysis (a) B is self-masking (b) B is critical

Thus the main problem is to determine whether a stem x is critical, given that some of its fanout branches are critical. Let v be the value of x in the analyzed test t. One obvious solution is to explicitly simulate the fault x s-a-$\bar{v}$, and if t detects this fault, then mark x as critical. Critical path tracing solves this problem by a different type of analysis, which we describe next.

Definition 5.3: Let t be a test that activates fault f in a single-output combinational circuit. Let y be a line with level l_y, sensitized to f by t. If every path sensitized to f either goes through y or does not reach any line with level greater than l_y, then y is said to be a *capture line* of f in test t.

A capture line, if one exists, is a bottleneck for the propagation of fault effects, because it is common to all the paths on which the effects of f can propagate to the primary output in test t. If t detects f, then there exists at least one capture line of f, namely the primary output itself. If the effect of f propagates on a single path, then every line on that path is a capture line of f.

Example 5.6: Consider the circuit in Figure 5.25 and let f be b s-a-0. The capture lines of f in the test shown are e, j, and q. Lines n and p are sensitized to f but are not capture lines of f. □

Let y be a capture line of fault f in test t and assume that y has value v in t. It is easy to show that any capture line of y s-a-$\bar{v}$ is also a capture line of f. In other words, the capture lines of a fault form a "transitive chain." For example, in Figure 5.25, j and q are also capture lines of e s-a-0, and q is also a capture line of j s-a-1.

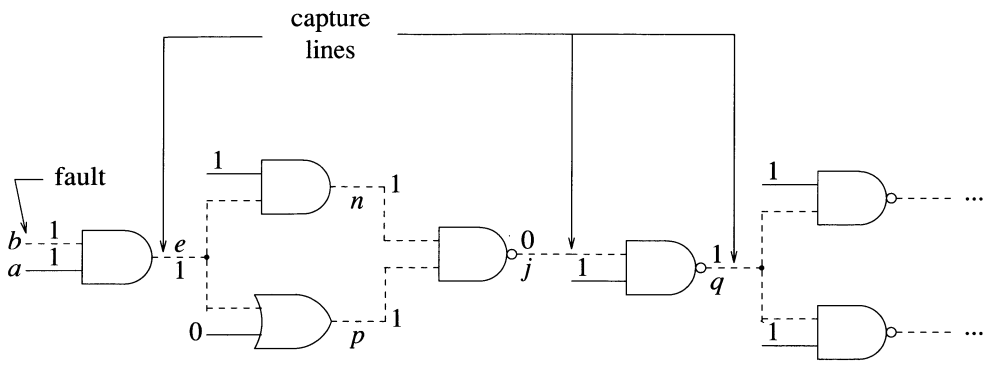


Figure 5.25 Example of capture lines

Theorem 5.1: A test t detects the fault f iff all the capture lines of f in t are critical in t.

Proof: First note that if no capture line exists, then t does not detect f (because otherwise there exists at least one capture line). Let y be an arbitrary capture line of f in t and let v be its value. The value of y in the presence of f is $\overline{v}$, and no other effect of f but $y=\overline{v}$ can reach any line whose level is greater than the level of y.

1. Let us assume that all the capture lines of f in t are critical. Hence, the error $y=\overline{v}$ caused by y s-a-$\overline{v}$ propagates to the primary output. Then the same is true for the error $y=\overline{v}$ caused by f. Therefore t detects f.

2. Let us assume that t detects f. Hence, the error $y=\overline{v}$ caused by f at y propagates to the primary output. Then the same is true for the error $y=\overline{v}$ caused by y s-a-$\overline{v}$. Thus t detects y s-a-$\overline{v}$ and therefore all the capture lines of f are critical in t. □

Theorem 5.1, together with the transitivity property of a set of capture lines, shows that to determine whether a stem is critical, we may not need to propagate the effects of the stem fault "all the way" to a primary output as done in explicit fault simulation. Rather, it is sufficient to propagate the fault effects only to the first capture line of the stem fault (i.e., the one closest to the stem). Then the stem is critical iff the capture line is critical.

As capture lines are defined for a single-output circuit, we partition a circuit with m primary outputs into m single-output circuits called *cones*. A cone contains all the logic feeding one primary output. To take advantage of the simplicity of critical path tracing in fanout-free circuits, within each cone we identify *fanout-free regions* (FFRs). Figure 5.26 shows these structures for an adder with two outputs. The inputs of a FFR are checkpoints of the circuit, namely fanout branches and/or primary inputs. The output of a FFR is either a stem or a primary output. Constructing cones and FFRs is a preprocessing step of the algorithm.

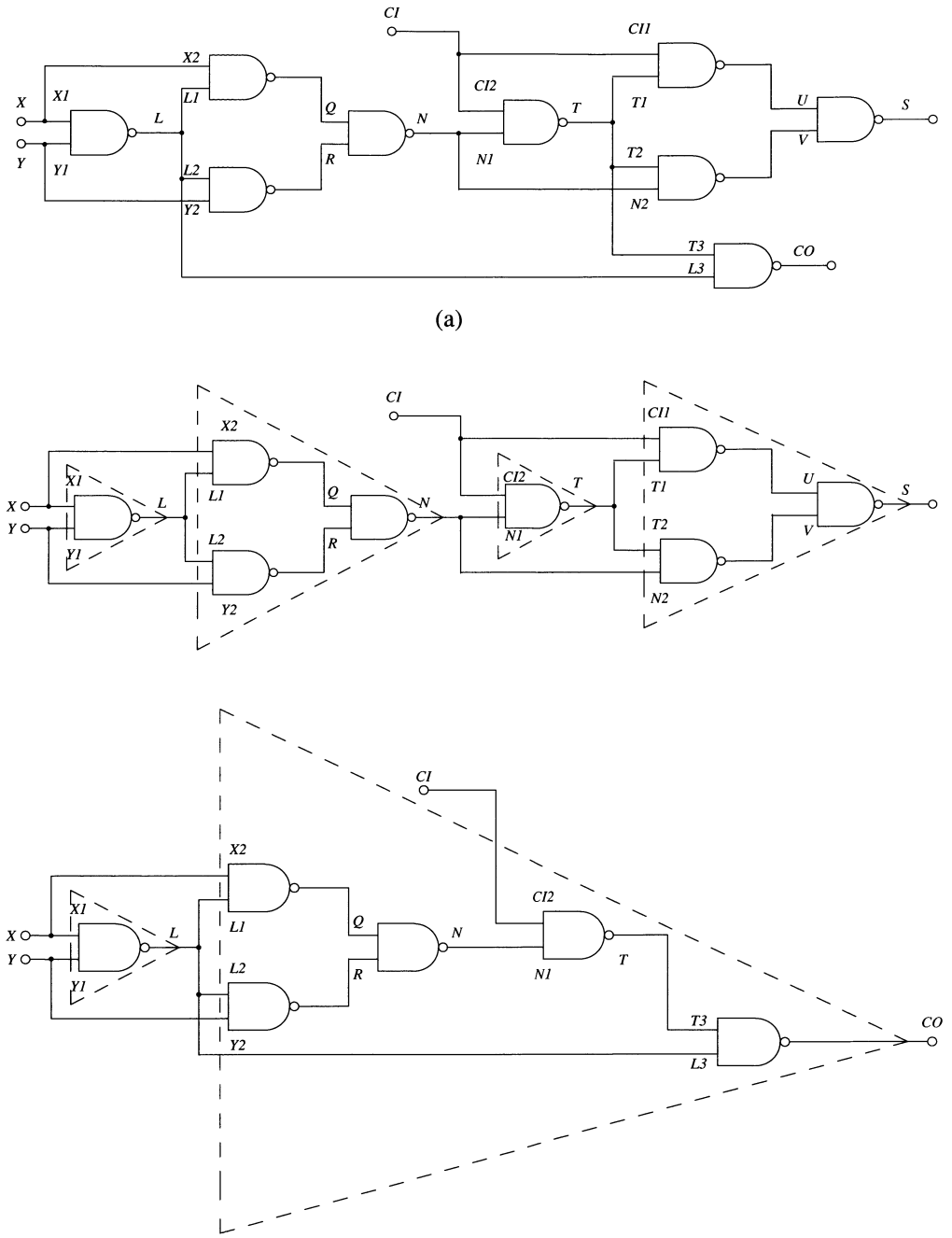


Figure 5.26 (a) Full-adder circuit (b) Cones for *S* and *CO* partitioned into FFRs

Figure 5.27 outlines the critical path tracing algorithm for evaluating a given test. It assumes that fault-free simulation, including the marking of the sensitive gate inputs, has been performed. The algorithm processes every cone starting at its primary output and alternates between two main operations: critical path tracing inside a FFR, represented by the procedure *Extend*, and checking a stem for criticality, done by the function *Critical*. Once a stem j is found to be critical, critical path tracing continues from j.

```
for every primary output z
  begin
    Stems_to_check = ∅
    Extend(z)
    while (Stems_to_check ≠ ∅)
      begin
        j = the highest level stem in Stems_to_check
        remove j from Stems_to_check
        if Critical(j) then Extend(j)
      end
  end
```

Figure 5.27 Outline of critical path tracing

Figure 5.28 shows the recursive procedure *Extend(i)*, which backtraces all the critical paths inside a FFR starting from a given critical line i by following lines marked as *sensitive*. *Extend* stops at FFR inputs and collects all the stems reached in the set *Stems_to_check*.

```
Extend(i)
begin
  mark i as critical
  if i is a fanout branch then
    add stem(i) to Stems_to_check
  else
    for every input j of i
      if sensitive(j) then Extend(j)
end
```

Figure 5.28 Critical path tracing in a fanout-free region

Figure 5.29 outlines the function *Critical(j)*, which determines whether the stem j is critical by a breadth-first propagation of the effects of the stem fault. *Frontier* is the set of all gates currently on the frontier of this propagation. A gate in *Frontier* has

been reached by one or more fault effects from j, but we have not yet determined whether these propagate through the gate. This analysis is represented by the function *Propagates*. Because fault effects are propagated only in one cone, *Frontier* is bound to become empty. If the last gate removed from *Frontier* propagates fault effects, then its output is the first capture line of the stem fault and the result depends on the status (critical or not) of the capture line.

```
Critical(j)
begin
  Frontier = {fanouts of j}
  repeat
    begin
      i = lowest level gate in Frontier
      remove i from Frontier
      if (Frontier ≠ ∅) then
        begin
          if Propagates(i) then add fanouts of i to Frontier
        end
      else
        begin
          if Propagates(i) and i is critical then return TRUE
          return FALSE
        end
    end
end
```

Figure 5.29 Stem analysis

The function *Propagates(i)* determines whether gate i propagates the fault effects reaching its inputs based on the following rule.

Lemma 5.2: A gate i propagates fault effects iff:

1. either fault effects arrive only on sensitive inputs of i,
2. or fault effects arrive on all the nonsensitive inputs of i with controlling value and only on these inputs.

Proof: Based on Lemma 4.1 and the definition of sensitive inputs. □

Figure 5.30 shows different cases of propagation of fault effects through an AND gate. Fault effects are indicated by arrows.

Example 5.7: The table in Figure 5.31 shows the execution trace of critical path tracing in the circuit of Figure 5.26(a) for the test 111. Figure 5.32 shows the obtained critical paths. Note that the stem L is critical in the cone of S, but it is self-masking in the cone of CO. □

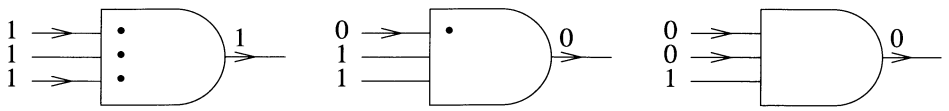


Figure 5.30 Propagation of fault effects through an AND gate

FFR traced	Critical lines	*Stems_to_check*	Stem checked	Capture line
S	*S, U, CI1, T1*	*T, CI*		
		CI	*T*	*U*
T	*T, N1*	*CI, N*		
		CI	*N*	*U*
N	*N, Q, R, L1, L2*	*CI, L*		
		CI	*L*	*N*
L	*L, X1, Y1*	*CI, X, Y*		
		X, Y	*CI*	*U*
CI	*CI*	*X, Y*		
		Y	*X*	*R*
X	*X*	*Y*		
		∅	*Y*	*Q*
Y	*Y*	∅		
CO	*CO, L3*	*L*		
		∅	*L*	—

Figure 5.31 Execution trace of critical path tracing in the circuit of Figure 5.26 for the test 111

When we analyze a stem, we have already backtraced a critical path between a primary output and one of the fanout branches of the stem. The effect of the stem fault would propagate on the same critical path, unless self-masking occurs. This may take place only when effects of the stem fault reach a reconvergence gate on paths with different inversion parities (see Figure 5.24). If we examine the circuit of Figure 5.26, we can observe that this situation can never occur for the stems T and L in the cone of S. A simple preprocessing technique to identify this type of stem is described in [Abramovici *et al.* 1984]. The resulting benefit is that whenever such a stem is reached by backtracing, it can be immediately identified as critical without any additional analysis.

Other techniques to speed up critical path tracing are presented in [Abramovici *et al.* 1984].

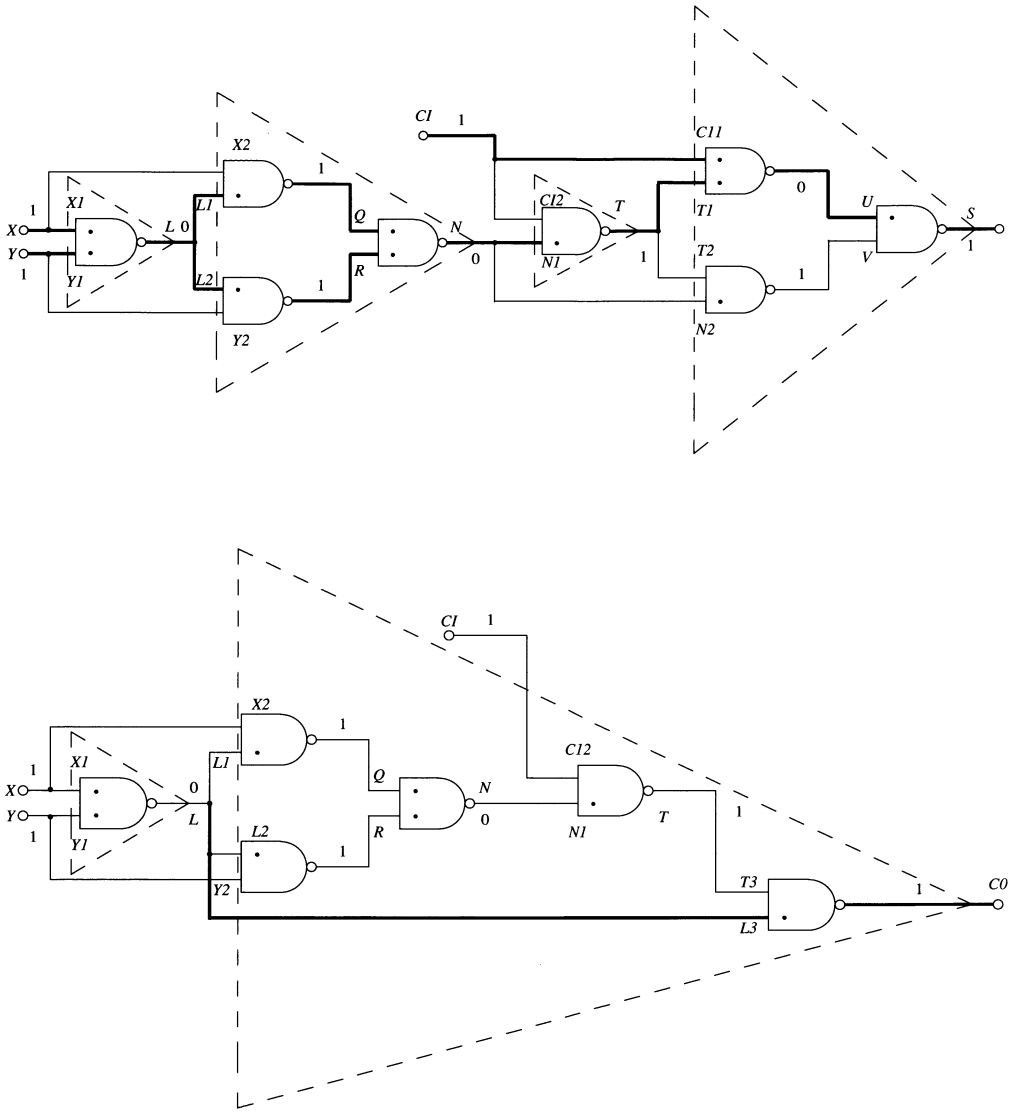


Figure 5.32 Critical paths in the circuit of Figure 5.26 for the test 111

Sometimes critical path tracing may not identify all the faults detected by a test. This may occur in a test t that propagates the effect of a fault on multiple paths that reconverge at a gate without sensitive inputs in t. For example, in Figure 5.33 the effects of j s-a-0 propagate on two paths and reach the reconvergence gate on nonsensitive inputs. In this case critical path tracing would not reach j. This situation occurs seldom in practical circuits. In the following we shall show that even when it does occur, its impact is negligible.

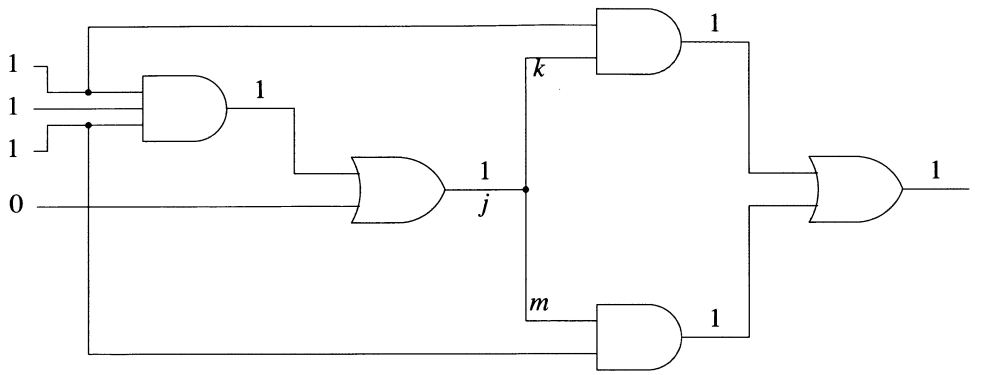


Figure 5.33

For *test grading*, it does not matter which test detects a fault but whether the fault is detected by any of the tests in the evaluated test set. Even if a fault is not correctly recognized as detected in one test, it is likely that it will be detected in other tests. Eventually the only faults incorrectly flagged as not detected are those that are detected *only* by multiple-path sensitization with reconvergence at a gate with nonsensitive inputs. If this unlikely situation occurs, then the computed fault coverage will be slightly pessimistic.

In the context of test generation, the function of critical path tracing is to aid in the selection of the next target fault (see Figure 5.3). Again, for the overall process it does not matter which test detects a fault but whether the fault is detected by any of the tests generated so far. The only adverse effect on the test generation process will occur if critical path tracing incorrectly identifies a fault f as not detected by any of the currently generated tests *and* f is selected as the next target fault. In this unlikely situation the test generator will needlessly generate a test for the already detected fault f. As we will show in the next chapter, usually f will be detected by sensitizing a single path, and then critical path tracing will mark f as detected. If the test generator has obtained a test for f, but f is not marked as detected by critical path tracing, then we directly mark the corresponding line as critical and restart critical path tracing from that line. Thus in practical terms there is no impact on the test generation process.

Compared with conventional fault simulation, the distinctive features of critical path tracing are as follows:

- *It directly identifies the faults detected by a test*, without simulating the set of all possible faults. Hence the work involved in propagating the faults that are not detected by a test towards the primary outputs is avoided.
- *It deals with faults only implicitly.* Therefore fault enumeration, fault collapsing, fault partitioning (for multipass simulation), fault insertion, and fault dropping are no longer needed.

- *It is based on a path tracing algorithm* that does not require computing values in the faulty circuits by gate evaluations or fault list processing.
- *It is an approximate method.* The approximation seldom occurs and results in not marking as detected some faults that are actually detected by the evaluated test. We have shown that even if the approximation occurs, its impact on the applications of critical path tracing is negligible.

Consequently, critical path tracing is faster and requires less memory than conventional fault simulation. Experimental results presented in [Abramovici *et al.* 1984] show that critical path tracing is faster than concurrent fault simulation.

Additional features of critical path tracing and their use in test generation are described in Chapter 6.

Antreich and Schulz [1987] developed a method that combines features similar to those used in critical path tracing with parallel-pattern evaluation.

5.4 Fault Sampling

The computational requirements (i.e., run-time and memory) of the general fault simulation methods increase with the number of simulated faults. Let M be the number of (collapsed) SSFs in the analyzed circuit and let K be the number of faults detected by the evaluated test sequence T. Then the fault coverage of T is $F=K/M$. *Fault sampling* [Butler *et al.* 1974, Agrawal 1981, Wadsack 1984] is a technique that reduces the cost of fault simulation by simulating only a random sample of $m<M$ faults.

Let k be the number of faults detected by T when simulating m faults. There exists an obvious trade-off between the cost of fault simulation and the accuracy of the estimated fault coverage $f=k/m$. This trade-off is controlled by the size m of the sample. The problem is to determine a sample size m such that we can be confident (with a specified confidence level c) that the error in the estimated fault coverage is bounded by a given e_{max}. In other words, we want to determine m such that the estimated fault coverage f belongs to the interval $[F-e_{max},F+e_{max}]$ with a probability c.

Since the m faults in the sample are randomly selected, we can regard the number of detected faults k as a random variable. The probability that T will detect k faults from a random sample of size m, given that it detects K faults from the entire set of M faults, is

$$P_k(m,M,K) = \frac{\binom{K}{k}\binom{M-K}{m-k}}{\binom{M}{m}}$$

This is a hypergeometric distribution with mean

$$\mu_k = m\,\frac{K}{M} = mF$$

and variance

$$\sigma_k^2 = m\frac{K}{M}\left[1 - \frac{K}{M}\right]\frac{M-m}{M-1} \simeq mF(1-F)(1 - m/M)$$

where σ_k denotes the standard deviation. For large M, this distribution can be approximated by a normal distribution with mean μ_k and standard deviation σ_k. Hence the estimated fault coverage f can be considered to be a random variable with normal distribution and mean

$$\mu_f = \mu_k/m = F$$

and variance

$$\sigma_f^2 = \sigma_k^2/m^2 = \frac{1}{m}F(1-F)(1 - m/M)$$

With a confidence level of 99.7 percent, we can claim that the estimated fault coverage f is in the interval $[F-3\sigma_f, F+3\sigma_f]$. Therefore, it is almost certain that the estimation error is bounded by $3\sigma_f$; i.e., the maximum error $e_{\max}$ is given by

$$e_{\max} = 3\sqrt{F(1-F)(1-m/M)\ 1/\ m}$$

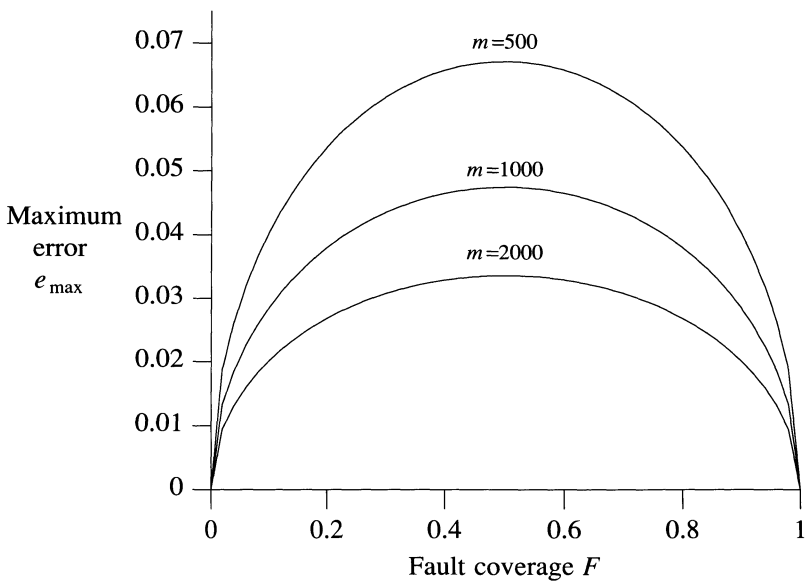


Figure 5.34 The maximum error in the estimated fault coverage

To reduce significantly the cost of fault simulation, the sample size m should be much smaller than the total number of faults M. With $m \ll M$, we can approximate $(1-m/M) \simeq 1$. Then the error becomes independent of the total number of faults M. Figure 5.34 illustrates how $e_{\max}$ depends on F for several values of m. Note that the

worst case occurs when the true fault coverage F is 50 percent. This analysis shows that a sample size of 1000 ensures that the estimation error is less than 0.05.

Fault sampling can provide an accurate estimate of the fault coverage at a cost substantially lower than that of simulating the entire set of faults. However, no information is available about the detection status of the faults not included in the sample; hence it may be difficult to improve the fault coverage.

5.5 Statistical Fault Analysis

Because exact fault simulation is an expensive computational process, approximate fault simulation techniques have been developed with the goal of trading off some loss of accuracy in results for a substantial reduction in the computational cost. In this section we examine the principles used in STAFAN, a *Sta*tistical *F*ault *An*alysis method which provides a low-cost alternative to exact fault simulation [Jain and Agrawal 1985].

Like a conventional fault simulator, STAFAN includes a fault-free simulation of the analyzed circuit N for a test sequence T. STAFAN processes the results of the fault-free simulation to estimate, for every SSF under consideration, its probability of being detected by T. The overall fault coverage is then estimated based on the detection probabilities of the individual faults.

First assume that N is a combinational circuit. STAFAN treats T as a set of n independent random vectors. Let d_f be the probability that a randomly selected vector of T detects the fault f. Because the vectors are assumed independent, the probability of not detecting f with n vectors is $(1-d_f)^n$. Then the probability d_f^n that a set of n vectors detects f is

$$d_f^n = 1-(1-d_f)^n$$

Let Φ be the set of faults of interest. The expected number of faults detected by n vectors is $D_n = \sum_{f \in \Phi} d_f^n$, and the corresponding expected fault coverage is $D_n / \mid \Phi \mid$ (which is also the average detection probability).

The basis for the above computations is given by the detection probabilities d_f. Let f be the *s-a-v* fault on line l. To detect f, a vector must activate f by setting l to value $\overline{v}$ and must propagate the resulting fault effect to a primary output. STAFAN processes the results of the fault-free simulation to estimate separately the probability of activating f and the probability of propagating a fault effect from l to a primary output. These probabilities are defined as follows:

- *C1(l)*, called the *1-controllability* of l, is the probability that a randomly selected vector of T sets line l to value 1. The *0-controllability*, *C0(l)*, is similarly defined for value 0.
- *O(l)*, called the *observability* of l, is the probability that a randomly selected vector of T propagates a fault effect from l to a primary output (our treatment of observabilities is slightly different from that in [Jain and Agrawal 1985]).

STAFAN uses the simplifying assumption that activating the fault and propagating its effects are independent events. Then the detection probabilities of the *s-a-0* and *s-a-1*

faults on l can be computed by $C1(l)O(l)$ and $C0(l)O(l)$. (Some pitfalls of the independence assumption are discussed in [Savir 1983]).

To estimate controllabilities and observabilities, STAFAN counts different events that occur during the fault-free simulation of n vectors. A *0-count* and a *1-count* are maintained for every line l; they are incremented, respectively, in every test in which l has value 0 or 1. After simulating n vectors, the 0- and the 1- controllability of l are computed as

$$C0(l) = \frac{0\text{-}count}{n}$$

and

$$C1(l) = \frac{1\text{-}count}{n}$$

An additional *sensitization-count* is maintained for every gate input l. This count is incremented in every test in which l is a sensitive input of the gate (see Definition 5.2). After simulating n vectors, the probability $S(l)$ that a randomly selected vector propagates a fault effect from l to the gate output is computed as

$$S(l) = \frac{sensitization\text{-}count}{n}$$

The computation of observabilities starts by setting $O(i)=1$ for every primary output i. Observabilities for the other lines are computed via a backward traversal of the circuit, in which $O(l)$ is computed based on the observability of its immediate successor(s). Let l be an input of a gate with output m. To propagate a fault effect from l to a primary output, we need to propagate it from l to m and from m to a primary output. Using the simplifying assumption that these two problems are independent, we obtain $O(l)$ as

$$O(l) = S(l)O(m)$$

Now let l be a stem with k fanout branches $l_1, l_2, ..., l_k$ (see Figure 5.35). Assume that we have computed all the $O(l_i)$ values, and now we want to determine $O(l)$. Let L_i (L) denote the event whose probability is $O(l_i)$ ($O(l)$), that is, the propagation of a fault effect from l_i (l) to a primary output. Here we use two more simplifying assumptions: (1) the events $\{L_i\}$ are independent, and (2) the event L occurs if and only if any subset of $\{L_i\}$ occurs. Then

$$L = \bigcup_{i=1}^{k} L_i$$

Because of reconvergent fanout, L may occur even when none of the L_i events happens, and the occurrence of a subset of $\{L_i\}$ does not guarantee that L will take place. STAFAN treats the probability of $\bigcup_{i=1}^{k} L_i$ as an upper bound on the value of $O(l)$. The lower bound is obtained by assuming that L can be caused by the most probable L_i. Thus

$$\max_{1 \le i \le k} O(l_i) \le O(l) \le P\left(\bigcup_{i=1}^{k} L_i\right)$$

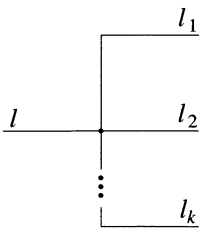


Figure 5.35

For example, for k=2, $P(L_1 \cup L_2) = O(l_1)+O(l_2)-O(l_1)O(l_2)$. Based on these bounds, STAFAN computes the observability of the stem l by

$$O(l)=(1-\alpha)\max_{1\le i\le k} O(l_i)+\alpha P(\bigcup_{i=1}^{k} L_i)$$

where α is a constant in the range [0,1]. The lower bound for $O(l)$ is obtained with α=0, and the upper bound corresponds to α=1. The experiments reported in [Jain and Agrawal 1985] show that the results are insensitive to the value of α.

STAFAN applies the same computation rules to estimate detection probabilities in a sequential circuit (some additional techniques are used to handle feedback loops). This results in more approximations, because both fault activation and error propagation are, in general, achieved by sequences of related vectors, rather than by single independent vectors. In spite of its approximations, the fault coverage obtained by STAFAN was found to be within 6 percent of that computed by a conventional fault simulator [Jain and Singer 1986].

Based on their detection probabilities computed by STAFAN, faults are grouped in the following three ranges:

- the high range, containing faults with detection probabilities greater than 0.9;
- the low range, containing faults whose detection probabilities are smaller than 0.1;
- the middle range, grouping faults with detection probabilities between 0.1 and 0.9.

The faults in the high range are considered likely to be detected by the applied test; those in the low range are assumed not detected, and no prediction is made about the detectability of the faults in the middle range. The results for sequential circuits reported in [Jain and Singer 1986] show that the accuracy of the predictions made by STAFAN is as follows:

- Between 91 and 98 percent of the faults in the high range are indeed detected.
- Between 78 and 98 percent of the faults in the low range are indeed undetected.
- The middle range contains between 2 and 25 percent of the faults (in most cases less than 10 percent).

The results presented in [Huisman 1988] also show that lack of accuracy in fault labeling is the main problem in applying STAFAN.

STAFAN requires the following additional operations to be performed during a fault-free simulation run. First, the 0-count and the 1-count of every gate output, and the sensitization-count of every gate input, must be updated in every vector. Second, controllabilities, observabilities, and detection probabilities are computed after simulating a group of n vectors. Since n can be large, the complexity of STAFAN is dominated by the first type of operations (updating counters); their number is proportional to Gn, where G is the gate count of the circuit.

5.6 Concluding Remarks

Fault simulation plays an important role in ensuring a high quality for digital circuits. Its high computational cost motivates new research in this area. The main research directions are hardware support, new algorithms, and the use of hierarchical models.

One way to use hardware support to speed up fault simulation is by special-purpose accelerators attached to a general-purpose host computer. The architectures for fault simulation are similar to those described in Chapter 3 for logic simulation. Although any hardware accelerator for logic simulation can be easily adapted for serial fault simulation, the serial nature of this process precludes achieving a significant speed-up. Hardware and microprogrammed implementations of the concurrent algorithm (for example, [Chan and Law 1986], [Stein *et al.* 1986]) obtain much better performance.

Interconnected general-purpose processors can also provide support for fault simulation by partitioning the set of faults among processors. Each processor executes the same concurrent algorithm working only on a subset of faults. The processors can be parallel processors connected by a shared bus or independent processors connected by a network [Goel *et al.* 1986, Duba *et al.* 1988]. With proper fault-partitioning procedures, the speed-up obtained grows almost linearly with the number of processors used.

Two new algorithms have been developed for synchronous sequential circuits using a 0-delay model. One is based on extending critical path tracing [Menon *et al.* 1988]. The other, called *differential fault simulation* (DSIM), combines concepts of concurrent simulation and single fault propagation [Cheng and Yu 1989]. DSIM simulates in turn every faulty circuit, by keeping track of the differences between its values and those of the previously simulated circuit.

Hierarchical fault simulation [Rogers *et al.* 1987] relies on a hierarchical model of the simulated circuit, whose components have both a functional model (a C routine) and a structural one (an interconnection of primitive elements). The simulator uses the concurrent method and is able to switch between the functional and the structural models of components. Faults are inserted in the components modeled at the lower level. Efficiency is gained by using the higher-level models for fault-effect propagation.

REFERENCES

[Abramovici *et al.* 1977] M. Abramovici, M. A. Breuer, and K. Kumar, "Concurrent Fault Simulation and Functional Level Modeling," *Proc. 14th Design Automation Conf.*, pp. 128-137, June, 1977.

[Abramovici *et al.* 1984] M. Abramovici, P. R. Menon, and D. T. Miller, "Critical Path Tracing: An Alternative to Fault Simulation," *IEEE Design & Test of Computers*, Vol. 1, No. 1, pp. 83-93, February, 1984.

[Abramovici *et al.* 1986] M. Abramovici, J. J. Kulikowski, P. R. Menon, and D. T. Miller, "SMART and FAST: Test Generation for VLSI Scan-Design Circuits," *IEEE Design & Test of Computers*, Vol. 3, No. 4, pp. 43-54, August, 1986.

[Agrawal 1981] V. D. Agrawal, "Sampling Techniques for Determining Fault Coverage in LSI Circuits," *Journal of Digital Systems*, Vol. 5, No. 3, pp. 189-202, Fall, 1981.

[Agrawal *et al.* 1981] V. D. Agrawal, S. C. Seth, and P. Agrawal, "LSI Product Quality and Fault Coverage," *Proc. 18th Design Automation Conf.*, pp. 196-203, June, 1981.

[Antreich and Schulz 1987] K. J. Antreich and M. H. Schulz, "Accelerated Fault Simulation and Fault Grading in Combinational Circuits," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-6, No. 9, pp. 704-712, September, 1987.

[Armstrong 1972] D. B. Armstrong, "A Deductive Method of Simulating Faults in Logic Circuits," *IEEE Trans. on Computers*, Vol. C-21, No. 5, pp. 464-471, May, 1972.

[Bose *et al.* 1982] A. K. Bose, P. Kozak, C-Y. Lo, H. N. Nham, E. Pacas-Skewes, and K. Wu, "A Fault Simulator for MOS LSI Circuits," *Proc. 19th Design Automation Conf.*, pp. 400-409, June, 1982.

[Bryant and Schuster 1983] R. E. Bryant and M. D. Schuster, "Fault Simulation of MOS Digital Circuits," *VLSI Design*, Vol. 4, pp. 24-30, October, 1983.

[Butler *et al.* 1974] T. T. Butler, T. G. Hallin, J. J. Kulzer, and K. W. Johnson, "LAMP: Application to Switching System Development," *Bell System Technical Journal*, Vol. 53, pp. 1535-1555, October, 1974.

[Chan and Law 1986] T. Chan and E. Law, "MegaFAULT: A Mixed-Mode, Hardware Accelerated Concurrent Fault Simulator," *Proc. Intn'l. Conf. on Computer-Aided Design*, pp. 394-397, November, 1986.

[Chang *et al.* 1974] H. Y. Chang, S. G. Chappell, C. H. Elmendorf, and L. D. Schmidt, "Comparison of Parallel and Deductive Simulation Methods," *IEEE Trans. on Computers*, Vol. C-23, No. 11, pp. 1132-1138, November, 1974.

[Chappell *et al.* 1974] S. G. Chappell, C. H. Elmendorf, and L. D. Schmidt, "LAMP: Logic-Circuit Simulators," *Bell System Technical Journal*, Vol. 53, pp. 1451-1476, October, 1974.

[Cheng and Yu 1989] W.-T. Cheng and M.-L. Yu, "Differential Fault Simulation — A Fast Method Using Minimal Memory," *Proc. 26th Design Automation Conf.*, pp. 424-428, June, 1989.

[Daniels and Bruce 1985] R. G. Daniels and W. C. Bruce, "Built-In Self-Test Trends in Motorola Microprocessors," *IEEE Design & Test of Computers*, Vol. 2, No. 2, pp. 64-71, April, 1985.

[Davidson and Lewandowski 1986] S. Davidson and J. L. Lewandowski, "ESIM/AFS — A Concurrent Architectural Level Fault Simulation," *Proc. Intn'l. Test Conf.*, pp. 375-383, September, 1986.

[Duba *et al.* 1988] P. A. Duba, R. K. Roy, J. A. Abraham, and W. A. Rogers, "Fault Simulation in a Distributed Environment," *Proc. 25th Design Automation Conf.*, pp. 686-691, June, 1988.

[Gai *et al.* 1986] S. Gai, F. Somenzi and E. Ulrich, "Advanced Techniques for Concurrent Multilevel Simulation," *Proc. Intn'l. Conf. on Computer-Aided Design*, pp. 334-337, November, 1986.

[Godoy and Vogelsberg 1971] H. C. Godoy and R. E. Vogelsberg, "Single Pass Error Effect Determination (SPEED)," *IBM Technical Disclosure Bulletin*, Vol. 13, pp. 3443-3344, April, 1971.

[Goel 1980] P. Goel, "Test Generation Costs Analysis and Projections," *Proc. 17th Design Automation Conf.*, pp. 77-84, June, 1980.

[Goel *et al.* 1986] P. Goel, C. Huang, and R. E. Blauth, "Application of Parallel Processing to Fault Simulation," *Proc. Intn'l. Conf. on Parallel Processing*, pp. 785-788, August, 1986.

[Harel and Krishnamurthy 1987] D. Harel and B. Krishnamurthy, "Is There Hope for Linear Time Fault Simulation?," *Digest of Papers 17th Intn'l. Symp. on Fault-Tolerant Computing*, pp. 28-33, July, 1987.

[Henckels *et al.* 1980] L. P. Henckels, K. M. Brown, and C-Y. Lo, "Functional Level, Concurrent Fault Simulation," *Digest of Papers 1980 Test Conf.*, pp. 479-485, November, 1980.

[Hong 1978] S. J. Hong, "Fault Simulation Strategy for Combinational Logic Networks," *Digest of Papers 8th Annual Intn'l Conf. on Fault-Tolerant Computing*, pp. 96-99, June, 1978.

[Huisman 1988] L. M. Huisman, "The Reliability of Approximate Testability Measures," *IEEE Design & Test of Computers*, Vol. 5, No. 6, pp. 57-67, December, 1988.

[Iyengar and Tang 1988] V. S. Iyengar and D. T. Tang, "On Simulating Faults in Parallel," *Digest of Papers 18th Intn'l. Symp. on Fault-Tolerant Computing*, pp. 110-115, June, 1988.

[Jain and Agrawal 1985] S. K. Jain and V. D. Agrawal, "Statistical Fault Analysis," *IEEE Design & Test of Computers*, Vol. 2, No. 1, pp. 38-44, February, 1985.

[Jain and Singer 1986] S. K. Jain and D. M. Singer, "Characteristics of Statistical Fault Analysis," *Proc. Intn'l. Conf. on Computer Design*, pp. 24-30, October, 1986.

[Kawai and Hayes 1984] M. Kawai and J. P. Hayes, "An Experimental MOS Fault Simulation Program CSASIM," *Proc. 21st Design Automation Conf.*, pp. 2-9, June, 1984.

[Ke *et al.* 1988] W. Ke, S. Seth, and B. B. Bhattacharya, "A Fast Fault Simulation Algorithm for Combinational Circuits," *Proc. Intn'l. Conf. on Computer-Aided Design*, pp. 166-169, November, 1988.

[Levendel 1980] Y. H. Levendel, private communication, 1980.

[Levendel and Menon 1980] Y. H. Levendel and P. R. Menon, "Comparison of Fault Simulation Methods — Treatment of Unknown Signal Values," *Journal of Digital Systems*, Vol. 4, pp. 443-459, Winter, 1980.

[Lo *et al.* 1987] C-Y. Lo, H. H. Nham, and A. K. Bose, "Algorithms for an Advanced Fault Simulation System in MOTIS," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-6, No. 3, pp. 232-240, March, 1987.

[McCluskey and Buelow 1988] E. J. McCluskey and F. Buelow, "IC Quality and Test Transparency," *Proc. Intn'l. Test Conf.*, pp. 295-301, September, 1988.

[Menon and Chappell 1978] P. R. Menon and S. G. Chappell, "Deductive Fault Simulation with Functional Blocks," *IEEE Trans. on Computers*, Vol. C-27, No. 8, pp. 689-695, August, 1978.

[Menon *et al.* 1988] P. R. Menon, Y. H. Levendel, and M. Abramovici, "Critical Path Tracing in Sequential Circuits," *Proc. Intn'l. Conf. on Computer-Aided Design*, pp. 162-165, November, 1988.

[Moorby 1983] P. R. Moorby, "Fault Simulation Using Parallel Value Lists," *Proc. Intn'l. Conf. on Computer-Aided Design*, pp. 101-102, September, 1983.

[Narayanan and Pitchumani 1988] V. Narayanan and V. Pitchumani, "A Parallel Algorithm for Fault Simulation on the Connection Machine," *Proc. Intn'l. Test Conf.*, pp. 89-93, September, 1988.

[Ozguner *et al.* 1979] F. Ozguner, W. E. Donath, and C. W. Cha, "On Fault Simulation Techniques," *Journal of Design Automation & Fault-Tolerant Computing*," Vol. 3, pp. 83-92, April, 1979.

[Rogers *et al.* 1987] W. A. Rogers, J. F. Guzolek, and J. Abraham, "Concurrent Hierarchical Fault Simulation," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-6, No. 9, pp. 848-862, September, 1987.

[Roth *et al.* 1967] 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*, Vol. EC-16, No. 10, pp. 567-579, October, 1967.

[Saab and Hajj 1984] D. Saab and I. Hajj, "Parallel and Concurrent Fault Simulation of MOS Circuits," *Proc. Intn'l. Conf. on Computer Design*, pp. 752-756, October, 1984.

[Savir 1983] J. Savir, "Good Controllability and Observability Do Not Guarantee Good Testability," *IEEE Trans. on Computers*, Vol. C-32, No. 12, pp. 1198-1200, December, 1983.

[Schuler and Cleghorn 1977] D. M. Schuler and R. K. Cleghorn, "An Efficient Method of Fault Simulation for Digital Circuits Modeled from Boolean Gates and Memories," *Proc. 14th Design Automation Conf.*, pp. 230-238, June, 1977.

[Seshu 1965] S. Seshu, "On an Improved Diagnosis Program," *IEEE Trans. on Electronic Computers*, Vol. EC-12, No. 2, pp. 76-79, February, 1965.

[Seth and Agrawal 1984] S. C. Seth and V. D. Agrawal, "Characterizing the LSI Yield Equation from Wafer Test Data," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-3, No. 2, pp. 123-126, April, 1984.

[Shen *et al.* 1985] J. P. Shen, W. Maly, and F. J. Ferguson, "Inductive Fault Analysis of MOS Integrated Circuits," *IEEE Design & Test of Computers*, Vol. 2, No. 6, pp. 13-26, December, 1985.

[Silberman and Spillinger 1986] G. M. Silberman and I. Spillinger, "The Difference Fault Model — Using Functional Fault Simulation to Obtain Implementation Fault Coverage," *Proc. Intn'l. Test Conf.*, pp. 332-339, September, 1986.

[Son 1985] K. Son, "Fault Simulation with the Parallel Value List Algorithm," *VLSI Systems Design*, Vol. 6, No. 12, pp. 36-43, December, 1985.

[Stein *et al.* 1986] A. J. Stein, D. G. Saab, and I. N. Hajj, "A Special-Purpose Architecture for Concurrent Fault Simulation," *Proc. Intn'l. Conf. on Computer Design*, pp. 243-246, October, 1986.

[Su and Cho 1972] S. Y. H. Su and Y-C. Cho, "A New Approach to the Fault Location of Combinational Circuits," *IEEE Trans. on Computers*, Vol. C-21, No. 1, pp. 21-30, January, 1972.

[Thompson and Szygenda 1975] E. W. Thompson and S. A. Szygenda, "Digital Logic Simulation in a Time-Based, Table-Driven Environment — Part 2. Parallel Fault Simulation," *Computer*, Vol. 8, No. 3, pp. 38-49, March, 1975.

[Ulrich and Baker 1974] E. G. Ulrich and T. G. Baker, "Concurrent Simulation of Nearly Identical Digital Networks," *Computer*, Vol. 7, No. 4, pp. 39-44, April, 1974.

[Wadsack 1984] R. L. Wadsack, "Design Verification and Testing of the WE32100 CPUs," *IEEE Design & Test of Computers*, Vol. 1, No. 3, pp. 66-75, August, 1984.

[Waicukauski *et al.* 1985] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, and T. McCarthy, "Fault Simulation for Structured VLSI," *VLSI Systems Design*, Vol. 6, No. 12, pp. 20-32, December, 1985.

[Williams and Brown 1981] T. W. Williams and N. C. Brown, "Defect Level as a Function of Fault Coverage," *IEEE Trans. on Computers*, Vol. C-30, No. 12, pp. 987-988, December, 1981.

PROBLEMS

5.1

a. Show that a single-output combinational circuit has no independent faults.

b. Construct a circuit that has two independent faults.

c. Show that the number of primary outputs is a least upper bound on the number of independent faults.

5.2 Simulate the latch shown in Figure 5.36, assuming $\bar{y} = 0$, $y = 1$, and input sequences **S** = 10 and **R** = 11. Use a simple event-directed unit delay simulation process with 0,1 logic states. Assume a parallel simulator with the following faults.

Show the value of y and $\bar{y}$ for each time frame.

5.3 For the circuit and the fault set used in Example 5.1, determine the faults detected by the test 11010 by deductive simulation.

5.4 Flow chart the following procedures for processing a sequential table structure used to store fault lists:

fault	*bit position*
fault-free	0
S s-a-1	1
S s-a-0	2
$\bar{y}$ s-a-1	3
$\bar{y}$ s-a-0	4

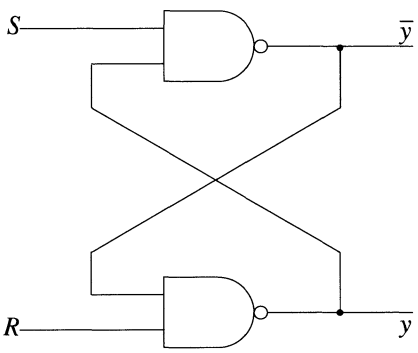


Figure 5.36

a. set intersection assuming ordered lists;

b. set intersection assuming unordered lists.

Compare the complexity of these two procedures.

5.5 For the latch shown in Figure 5.36, assume $\bar{y} = 0$, $y = 1$, and $S = R = 1$. Assume the initial fault lists associated with lines y, $\bar{y}$, R and S are L_y, $L_{\bar{y}}$, L_R, and L_S. Let S change to a 0. Determine the new output fault lists in terms of the given fault lists produced by this input event. Also, include all s-a-faults associated with this circuit.

5.6 Associate with each line α a list L_α^1, called the *one-list*, where fault $f \in L_\alpha^1$ if and only if line α in the circuit under fault f has the value 1. Note that $L_\alpha^1 = L_\alpha$ if line α in the fault-free circuit has the value 0, and $L_\alpha^1 = \bar{L}_\alpha$ if the line has the value 1. Show that for an AND (OR) gate with inputs a and b and output c, $L_c^1 = L_a^1 \cap L_b^1 \cup \{c\ s\text{-}a\text{-}1\}$ ($L_c^1 = L_a^1 \cup L_b^1 \cup \{c\ s\text{-}a\text{-}1\}$). What are the major advantages and disadvantages of carrying out fault analysis using L_α^1 rather than L_α?

5.7 Repeat the simulation carried out in Example 5.1 using the concurrent method. (Show the fault lists after simulating each vector.)

5.8 Consider the portion of a circuit shown in Figure 5.37, where $a = 0$.

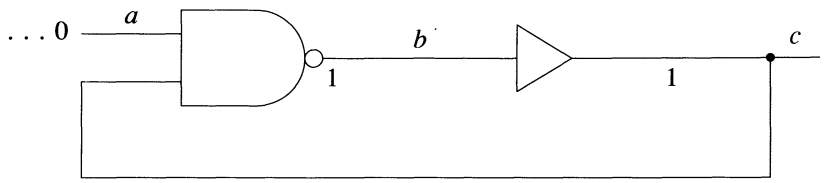


Figure 5.37 Simple circuit that oscillates for fault a s-a-1

Assume the initial conditions

$$L_a = L_A \cup \{a_1\},\ L_b = \varnothing,\ L_c = \varnothing,$$

where L_A is an arbitrary fault list and $b_o \notin L_A$. Note that the fault a s-a-1 causes the circuit to oscillate.

a. Determine the oscillatory values for the fault set L_b.

b. Simulate this same case using concurrent simulation.

c. Compare the complexity of these two simulation procedures for this case.

5.9 For the circuit in Figure 5.38, determine the faults detected by the test 111 by

a. concurrent fault simulation (start with a collapsed set of faults)

b. critical path tracing.

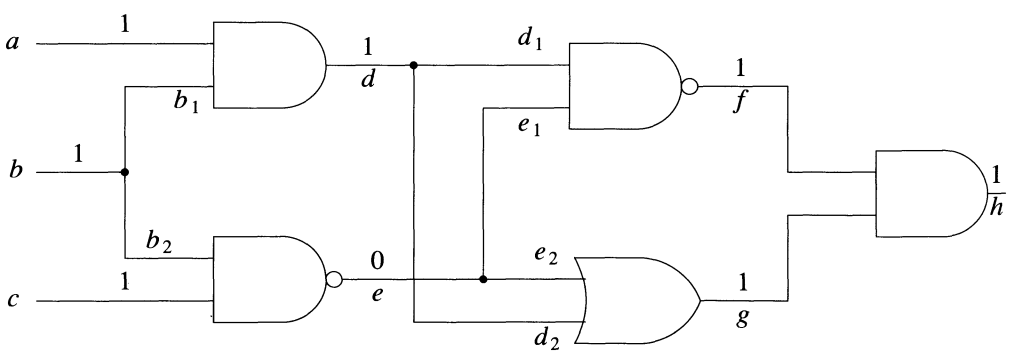


Figure 5.38

5.10 Let i denote a primary input of a fanout-free circuit and let p_i be the inversion parity of the path between i and the primary output. Let v_i be the value applied to i in a test t. Show that all primary inputs critical in t have the same sum $v_i \oplus p_i$.

5.11 Discuss the extensions needed to allow critical path tracing to handle partially specified vectors (i.e., any line may have value 0, 1 or x).