# 6. TESTING FOR SINGLE STUCK FAULTS

**About This Chapter**

In this chapter we examine test generation methods for SSFs. In Chapter 4 we discussed the wide applicability and the usefulness of the SSF model. Many TG methods dealing with other fault models extend the principles and techniques used for SSFs.

The TG process depends primarily on the type of testing experiment for which stimuli are generated. This chapter is devoted to off-line, edge-pin, stored-pattern testing with full comparison of the output results. On-line testing and compact testing are discussed in separate chapters.

## 6.1 Basic Issues

Test generation is a complex problem with many interacting aspects. The most important are

- the cost of TG;
- the quality of the generated test;
- the cost of applying the test.

The cost of TG depends on the complexity of the TG method. *Random TG* (RTG) is a simple process that involves only generation of random vectors. However, to achieve a high-quality test — measured, say, by the fault coverage of the generated test — we need a large set of random vectors. Even if TG itself is simple, determining the test quality — for example, by fault simulation — may be an expensive process. Moreover, a longer test costs more to apply because it increases the time of the testing experiment and the memory requirements of the tester. (Random vectors are often generated on-line by hardware and used with compact testing, but in this chapter we analyze their use only in the context of stored-pattern testing with full output comparison.)

RTG generally works without taking into account the function or the structure of the circuit to be tested. In contrast, *deterministic TG* produces tests by processing a model of the circuit. Compared to RTG, deterministic TG is more expensive, but it produces shorter and higher-quality tests. Deterministic TG can be manual or automatic. In this chapter we focus on *automatic TG* (ATG) methods.

Deterministic TG can be fault-oriented or fault-independent. In a *fault-oriented* process, tests are generated for specified faults of a fault universe (defined by an explicit fault model). *Fault-independent* TG works without targeting individual faults.

Of course, the TG cost also depends on the complexity of the circuit to be tested. Methods of reducing the complexity for testing purposes — referred to as *design for testability* techniques — form the subject of a separate chapter.

Figure 6.1 shows a general view of a deterministic TG system. Tests are generated based on a model of the circuit and a given fault model. The generated tests include

both the stimuli to be applied and the expected response of the fault-free circuit. Some TG systems also produce diagnostic data to be used for fault location.

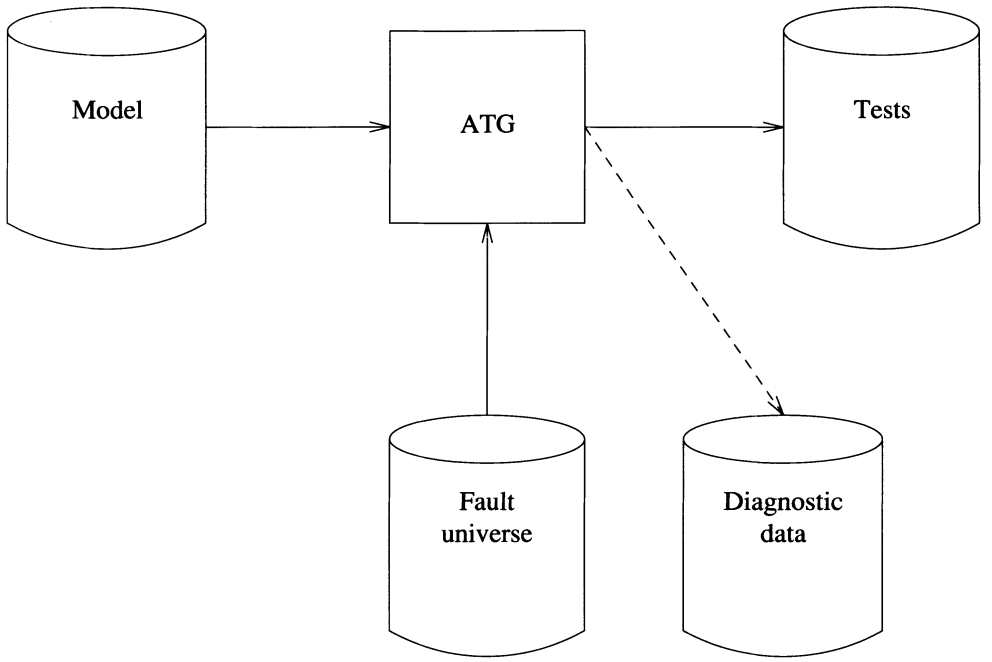



**Figure 6.1** Deterministic test generation system

## 6.2 ATG for SSFs in Combinational Circuits

In this section we consider only gate-level combinational circuits composed of AND, NAND, OR, NOR, and NOT gates.

### 6.2.1 Fault-Oriented ATG

#### Fanout-Free Circuits

We use fanout-free circuits only as a vehicle to introduce the main concepts of ATG for general circuits. The two fundamental steps in generating a test for a fault $l$ *s-a-v* are first, to *activate* (excite) *the fault*, and, second, to *propagate the resulting error* to a primary output (PO). Activating the fault means to set primary input (PI) values that cause line $l$ to have value $\bar{v}$. This is an instance of the **line-justification** problem, which deals with finding an assignment of PI values that results in a desired value setting on a specified line in the circuit. To keep track of error propagation we must consider values in both the fault-free circuit $N$ and the faulty circuit $N_f$ defined by the target fault $f$. For this we define *composite logic values* of the form $v/v_f$, where $v$ and $v_f$ are values of the same signal in $N$ and $N_f$. The composite logic values that represent errors — 1/0 and 0/1 — are denoted by the symbols $D$ and $\bar{D}$ [Roth 1966].

The other two composite values — 0/0 and 1/1 — are denoted by 0 and 1. Any logic operation between two composite values can be done by separately processing the fault-free and faulty values, then composing the results. For example, $\overline{D} + 0 = 0/1 + 0/0 = 0+0/1+0 = 0/1 = \overline{D}$. To these four binary composite values we add a fifth value ($x$) to denote an unspecified composite value, that is, any value in the set $\{0,1,D,\overline{D}\}$. In practice, logic operations using composite values are defined by tables (see Figure 6.2). It is easy to verify that $D$ behaves consistently with the rules of Boolean algebra, i.e., $D+\overline{D} = 1$, $D.\overline{D} = 0$, $D+D = D.D = D$, $\overline{D}.\overline{D} = \overline{D}+\overline{D} = \overline{D}$.

| $v/v_f$ | |
|---|---|
| 0/0 | 0 |
| 1/1 | 1 |
| 1/0 | $D$ |
| 0/1 | $\overline{D}$ |

(a)

| AND | 0 | 1 | $D$ | $\overline{D}$ | $x$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | $D$ | $\overline{D}$ | $x$ |
| $D$ | 0 | $D$ | $D$ | 0 | $x$ |
| $\overline{D}$ | 0 | $\overline{D}$ | 0 | $\overline{D}$ | $x$ |
| $x$ | 0 | $x$ | $x$ | $x$ | $x$ |

(b)

| OR | 0 | 1 | $D$ | $\overline{D}$ | $x$ |
|---|---|---|---|---|---|
| 0 | 0 | 1 | $D$ | $\overline{D}$ | $x$ |
| 1 | 1 | 1 | 1 | 1 | 1 |
| $D$ | $D$ | 1 | $D$ | 1 | $x$ |
| $\overline{D}$ | $\overline{D}$ | 1 | 1 | $\overline{D}$ | $x$ |
| $x$ | $x$ | 1 | $x$ | $x$ | $x$ |

(c)

**Figure 6.2** Composite logic values and 5-valued operations

Figure 6.3 shows the structure of an algorithm for generating a test for *l s-a-v*. It initializes all values to $x$ and it performs the two basic steps, represented by the routines *Justify* and *Propagate*.

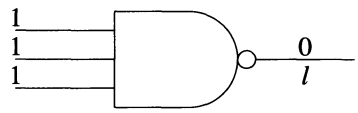
```
begin
    set all values to x
    Justify(l, v̄)
    if v = 0 then Propagate (l, D)
    else Propagate (l, D̄)
end
```

**Figure 6.3** Test generation for the fault *l s-a-v* in a fanout-free circuit

Line justification (Figure 6.4) is a recursive process in which the value of a gate output is justified by values of the gate inputs, and so on, until PIs are reached. Let us consider a NAND gate with $k$ inputs. There is only one way to justify a 0 output

value, but to justify a 1 value we can select any one of the $2^k-1$ input combinations that produce 1. The simplest way is to assign the value 0 to only one (arbitrarily selected) input and to leave the others unspecified. This corresponds to selecting one of the $k$ primitive cubes of the gate in which the output is 1.

```
Justify (l, val)
begin
    set l to val
    if l is a PI then return
    /* l is a gate (output) */
    c = controlling value of l
    i = inversion of l
    inval = val⊕i
    if (inval = c̄)
       then for every input j of l
          Justify (j, inval)
       else
          begin
             select one input (j) of l
             Justify (j, inval)
          end
end
```



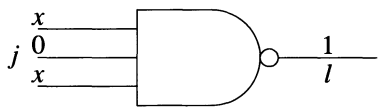




**Figure 6.4** Line justification in a fanout-free circuit

To propagate the error to the PO of the circuit, we need to sensitize the unique path from $l$ to the PO. Every gate on this path has exactly one input sensitized to the fault. According to Lemma 4.1, we should set all the other inputs of $G$ to the noncontrolling value of the gate. Thus we *transform the error-propagation problem into a set of line-justification problems* (see Figure 6.5).

**Example 6.1:** Let us generate a test for the fault $f$ *s-a*-0 in the circuit of Figure 6.6(a). The initial problems are *Justify(f,*1*)* and *Propagate(f,D)*. *Justify(f,*1*)* is solved by $a$=$b$=1. *Propagate(f,D)* requires *Justify(g,*0*)* and *Propagate(h,D)*. We solve *Justify(g,*0*)* by selecting one input of $g$ — say, $c$ — and setting it to 0. *Propagate(h,D)* leads to *Justify(i,*1*)*, which results in $e$=0. Now the error reaches the PO $j$. Figure 6.6(b) shows the resulting values. The generated test is 110$x$0 ($d$ can be arbitrarily assigned 0 or 1). □

It is important to observe that *in a fanout-free circuit every line-justification problem can be solved independently of all the others*, because the sets of PIs that are eventually assigned to justify the required values are mutually disjoint.

### Circuits with Fanout

Now we consider the general case of circuits with fanout and contrast it with the fanout-free case. We must achieve the same two basic goals — fault activation and error propagation. Again, fault activation translates into a line-justification problem. A first difference caused by fanout is that now we may have several ways to propagate an

```
Propagate (l, err)
/* err is D or D̄ */
begin
    set l to err
    if l is PO then return
    k = the fanout of l
    c = controlling value of k
    i = inversion of k
    for every input j of k other than l
        Justify (j, c̄)
    Propagate (k, err⊕i)
end
```

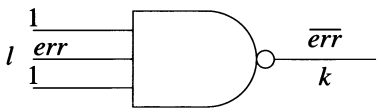
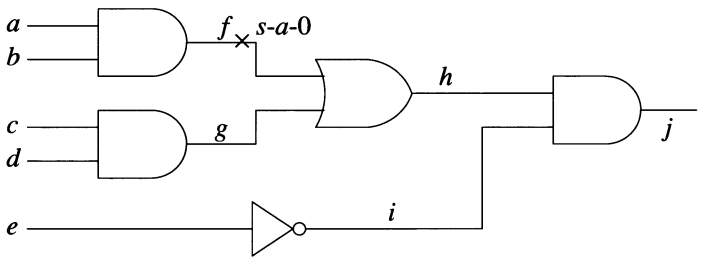
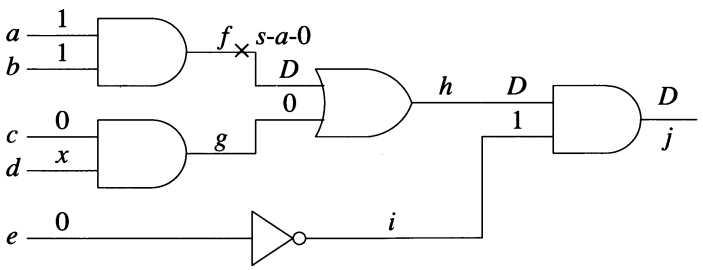



**Figure 6.5** Error propagation in a fanout-free circuit




(a)




(b)

**Figure 6.6**

error to a PO. But once we select a path, we again reduce the error-propagation problem to a set of line-justification problems. *The fundamental difficulty caused by*

*(reconvergent) fanout is that, in general, the resulting line-justification problems are no longer independent.*

**Example 6.2:** In the irredundant circuit of Figure 6.7 consider the fault $G_1$ *s-a-*1. To activate it we need to justify $G_1$=0. Now we have a choice of propagating the error via a path through $G_5$ or through $G_6$. Suppose we decide to select the former. Then we need to justify $G_2$ =1. The resulting set of problems — *Justify*($G_1$,0) and *Justify*($G_2$,1) — cannot be solved simultaneously, because their two unique solutions, $a$=$b$=$c$=1 and $a$=$d$=0, require contradictory values for $a$. This shows that the decision to propagate the error through $G_5$ was wrong. Hence we have to try an alternative decision, namely propagate the error through $G_6$. This requires $G_4$ = 1, which is eventually solved by $c$=1 and $e$=0. The resulting test is 111$x$0. □

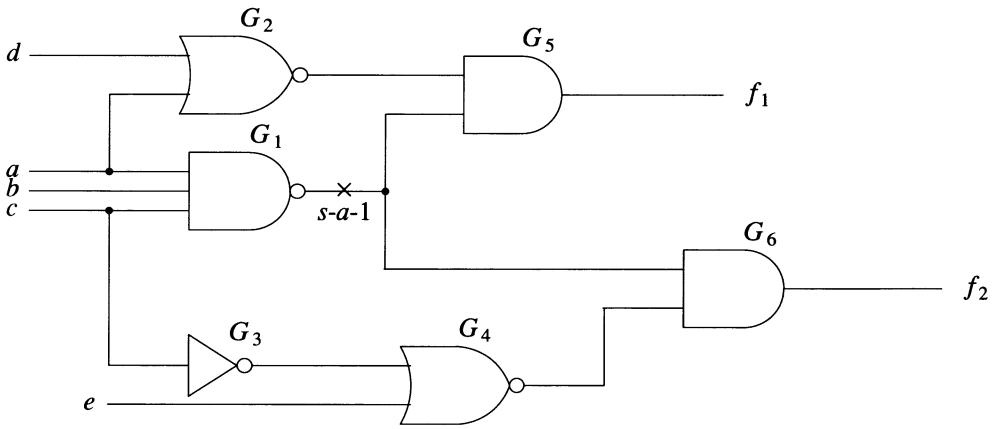



**Figure 6.7**

This example shows the need to explore alternatives for error propagation. Similarly, we may have to try different choices for line justification.

**Example 6.3:** Consider the fault $h$ *s-a-*1 in the circuit of Figure 6.8. To activate this fault we must set $h$=0. There is a unique path to propagate the error, namely through $p$ and $s$. For this we need $e$=$f$=1 and $q$=$r$=1. The value $q$=1 can be justified by $l$=1 or by $k$=1. First, let us try to set $l$=1. This leads to $c$=$d$=1. However, these two assignments, together with the previously specified $e$=1, would imply $r$=0, which leads to an inconsistency. Therefore the decision to justify $q$=1 by $l$=1 has been incorrect. Hence we must choose the alternative decision $k$=1, which implies $a$=$b$=1. Now the only remaining line-justification problem is $r$=1. Either $m$=1 or $n$=1 leads to consistent solutions. □

### Backtracking

We have seen that the search for a solution involves a *decision process.* Whenever there are several alternatives to justify a line or to propagate an error, we choose one of them to try. But in doing so we may select a decision that leads to an inconsistency

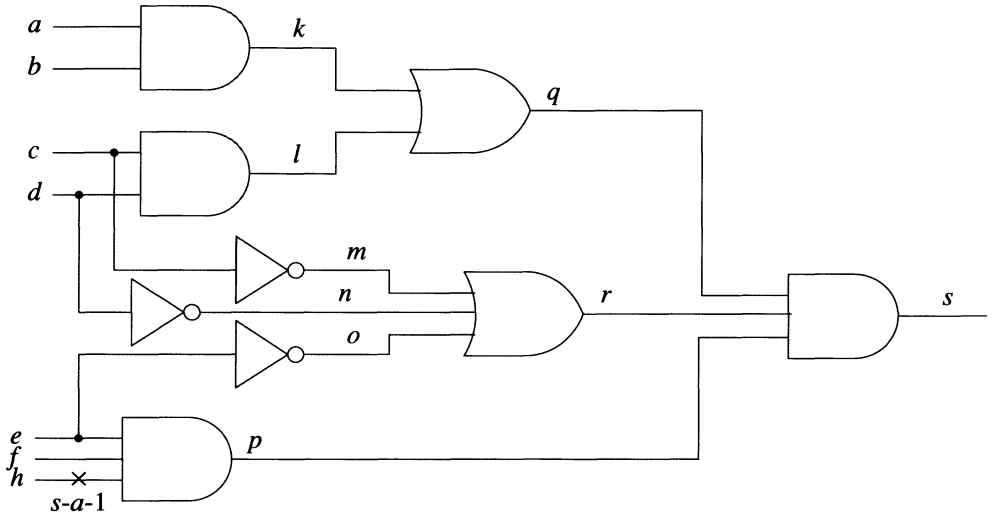



**Figure 6.8**

(also termed *contradiction* or *conflict*). Therefore in our search for a test we should use a **backtracking strategy** that allows a systematic exploration of the complete space of possible solutions and recovery from incorrect decisions. Recovery involves restoring the state of the computation to the state existing before the incorrect decision.

Usually the assignments performed as a result of a decision uniquely determine (imply) other values. The process of computing these values and checking for their consistency with the previously determined ones is referred to as **implication.** Figure 6.9 shows the progression of value computation for Example 6.3, distinguishing between values resulting from decisions and those generated by implication. The initial implications follow from the unique solutions to the fault-activation and error-propagation problems.

In most backtracking algorithms we must record all values assigned as a result of a decision, to be able to erase them should the decision lead to an inconsistency. In Example 6.3 all values resulting from the decision $l$=1 are erased when backtracking occurs.

Figure 6.10 outlines a recursive scheme of a backtracking TG algorithm for a fault. (This description is quite abstract, but more details will be provided later.) The original problems to be solved in generating a test for the fault $l$ *s-a-v* are to justify a value $\bar{v}$ on $l$ and to propagate the error from $l$ to a PO. The basic idea is that if a problem cannot be directly solved, we recursively transform it into subproblems and try to solve these first. Solving a problem may result in SUCCESS or FAILURE.

First the algorithm deals with all the problems that have unique solutions and hence can be solved by implication. These are processed by the procedure *Imply_and_check*,

| Decisions | Implications | |
|---|---|---|
| | $h=\bar{D}$ | Initial implications |
| | $e=1$ | |
| | $f=1$ | |
| | $p=\bar{D}$ | |
| | $r=1$ | |
| | $q=1$ | |
| | $o=0$ | |
| | $s=\bar{D}$ | |
| $l=1$ | | To justify $q=1$ |
| | $c=1$ | |
| | $d=1$ | |
| | $m=0$ | |
| | $n=0$ | |
| | $r=0$ | Contradiction |
| $k=1$ | $a=1$ | To justify $q=1$ |
| | $b=1$ | |
| $m=1$ | | To justify $r=1$ |
| | $c=0$ | |
| | $l=0$ | |

**Figure 6.9** Computations for Example 6.3

which also checks for consistency. The procedure *Solve* reports FAILURE if the consistency check fails. SUCCESS is reported when the desired goal is achieved, namely when an error has been propagated to a PO and all the line-justification problems have been solved. Even in a consistent state, the algorithm may determine that no error can be further propagated to a PO; then there is no point in continuing the search, and FAILURE is reported.

If *Solve* cannot immediately determine SUCCESS or FAILURE, then it selects one currently unsolved problem. This can be either a line-justification or an error-propagation problem. At this point there are several alternative ways to solve the selected problem. The algorithm selects one of them and tries to solve the problem at the next level of recursion. This process continues until a solution is found or all possible choices have failed. If the initial activation of *Solve* fails, then the algorithm has failed to generate a test for the specified fault.

The selection of an unsolved problem to work on and the selection of an untried way to solve it, can be — in principle — arbitrary. That is, if the fault is detectable, we

```
Solve()
begin
   if Imply_and_check() = FAILURE then return FAILURE
   if (error at PO and all lines are justified)
     then return SUCCESS
   if (no error can be propagated to a PO)
     then return FAILURE
   select an unsolved problem
   repeat
      begin
         select one untried way to solve it
         if Solve() = SUCCESS then return SUCCESS
      end
   until all ways to solve it have been tried
   return FAILURE
end
```

**Figure 6.10** General outline of a TG algorithm

will generate a test for it independent of the order in which problems and solutions are attempted. However, the selection process may greatly affect the efficiency of the algorithm as well as the test vector generated. Selection criteria are discussed in Section 6.2.1.3.

There are several fault-oriented TG algorithms whose structure is similar to that of *Solve*. In the following we first analyze concepts common to most of them, then we discuss specific algorithms in more detail.

#### 6.2.1.1 Common Concepts

**Decision Tree**

The execution of the backtracking TG algorithm can be visualized with the aid of a *decision tree* (see Figure 6.11). A decision node (shown as a circle) denotes a problem that the algorithm is attempting to solve. A branch leaving a decision node corresponds to a decision, i.e., trying one of the available alternative ways to solve the problem. A FAILURE terminal node of the tree (shown as a square labeled F) indicates the detection of an inconsistency or encountering a state that precludes further error propagation. A SUCCESS terminal node (shown as a square labeled S) represents finding a test. The execution of the algorithm can be traced by a depth-first traversal of the associated decision tree. For example, in Figure 6.11(b), starting at the decision node $q$=1, we first follow the branch $l$=1 which reaches an F terminal node. Then we backtrack to the last decision node and take the other branch ($k$=1), which leads to a new decision node ($r$=1). Here the first decision ($m$=1) reaches an S terminal node.

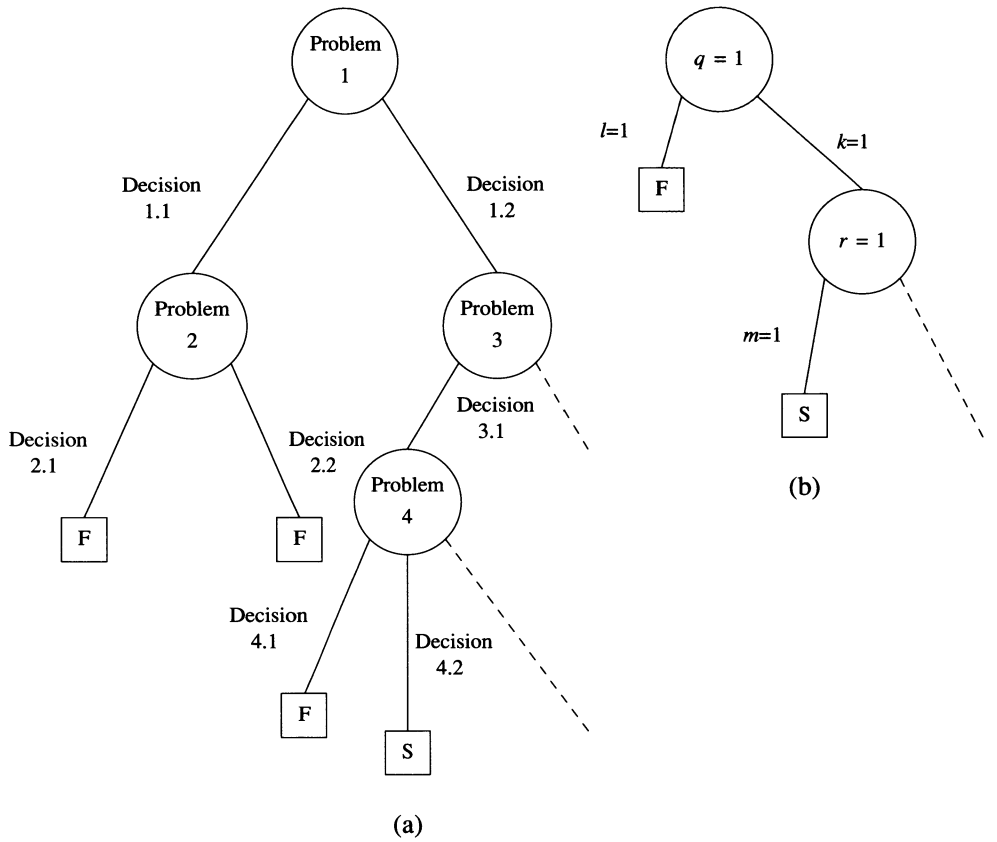



**Figure 6.11** Decision trees (a) General structure (b) Decision tree for Example 6.3

### Implicit Enumeration

An important property of the algorithm of Figure 6.10 is that it is *exhaustive*, that is, it is guaranteed to find a solution (test) if one exists. Thus if the algorithm fails to generate a test for a specified fault, then the fault is undetectable.

**Example 6.4:** Let us try to generate a test for the fault $f$ *s-a*-0 in the circuit of Figure 6.12(a). To justify $f$=1 we first try $b$=0 (see the decision tree in Figure 6.12(b)). But this implies $e$=1, which precludes error propagation through gate $h$. Trying $c$=0 results in a similar failure. We can conclude that no test exists for $f$ *s-a*-0. □

The algorithm is guaranteed to find a test, if one exists, because it can *implicitly enumerate all possible solutions.* The concept of implicit enumeration is best understood by contrasting it to explicit enumeration, which (in this context) means to repeatedly generate an input vector and to check whether it detects the target fault. Using implicit enumeration we direct the search toward vectors that can satisfy the set

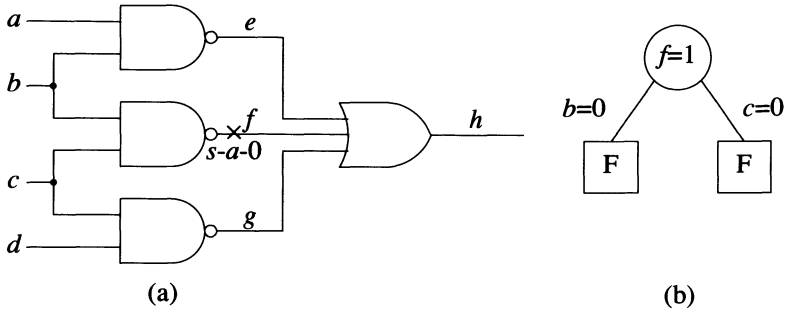



**Figure 6.12** TG failure for an undetectable fault (a) Circuit (b) Decision tree

of constraints imposed by the set of lines whose values must be simultaneously justified. As the set of constraints grows during the execution of the algorithm, the set of vectors that can satisfy them becomes smaller and smaller. The advantage of implicit enumeration is that it bounds the search space and begins to do so early in the search process.

Let us compare implicit and explicit enumeration for Example 6.4. Using implicit enumeration, we start by limiting the search to the set of vectors that satisfy $f$=1. From this set, we first reject the subset of all vectors with $b$=0. Then we reject the subset of all vectors with $c$=0, and this concludes the search. Using explicit enumeration, we would generate and simulate all $2^4$ input vectors.

**Complexity Issues**

Because of the exhaustive nature of the search process, the *worst-case complexity* of the algorithm in Figure 6.10 is *exponential*; i.e., the number of operations performed is an exponential function of the number of gates in the circuit. The worst-case behavior is characterized by many remade decisions; that is, much searching is done before a test is found or the target fault is recognized as undetectable. To minimize the total TG time, any practical TG algorithm is allowed to do only a limited amount of search; namely, the search is abandoned when the number of incorrect decisions (or the CPU time) reaches a specified limit. This may result in not generating tests for some detectable faults. The worst-case behavior has been observed mainly for undetectable faults [Cha *et al.* 1978].

The *best-case behavior* occurs when the result — generating a test or recognizing redundancy — is obtained without backtracking. This means either that the result is found only by implications (then the decision tree degenerates to one terminal node), or that only correct decisions are taken. Then the number of operations is a *linear* function of the number of gates. This is always the case for fanout-free circuits and for circuits without reconvergent fanout, because in these types of circuits no decision can produce a conflict. (Furthermore in such circuits all faults are detectable).

From analyzing the worst-case and the best-case behavior of the algorithm we can conclude that the key factor in controlling the complexity of our TG algorithm is to *minimize the number of incorrect decisions.* In Section 6.2.1.3 we shall discuss several heuristic techniques that help in achieving this goal. Here we introduce a simple principle that helps in minimizing the number of incorrect decisions by reducing the number of problems that require decisions (in other words, the algorithm will have fewer opportunities to make a wrong choice). This is the *maximum implications principle*, which requires one always to *perform as many implications as possible.* Clearly, by doing more implications, we either reduce the number of problems that otherwise would need decisions, or we reach an inconsistency sooner.

**Example 6.5:** Consider the circuit in Figure 6.13 and assume that at a certain stage in the execution of the TG algorithm we have to justify $f$=0 and $e$=0. Suppose that we justify $f$=0 by $c$=0. If we do not determine all the implications of this assignment, we are left with the problems of justifying $e$=0 and $c$=0. Both problems are solved, however, if we compute all implications. First $c$=0 implies $d$=1. This leaves only one way to justify $e$=0, namely by $b$=0, which in turn justifies $c$=0. □

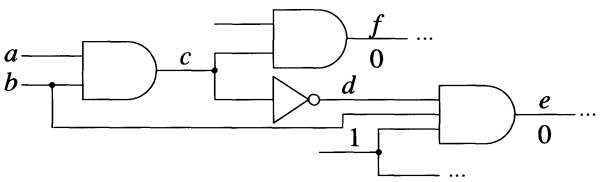



**Figure 6.13**

### The *D*-Frontier

The *D-frontier* consists of all gates whose output value is currently $x$ but have one or more error signals (either $D$'s or $\bar{D}$'s) on their inputs. Error propagation consists of selecting one gate from the *D-frontier* and assigning values to the unspecified gate inputs so that the gate output becomes $D$ or $\bar{D}$. This procedure is also referred to as the *D-drive* operation. If the *D-frontier* becomes empty during the execution of the algorithm, then no error can be propagated to a PO. Thus an empty *D-frontier* shows that backtracking should occur.

### The *J*-Frontier

To keep track of the currently unsolved line-justification problems, we use a set called the *J-frontier*, which consists of all gates whose output value is known but is not implied by its input values. Let $c$ be the controlling value and $i$ be the inversion of a gate on the *J-frontier*. Then the output value is $c \oplus i$, at least two inputs must have value $x$, and no input can have value $c$.

### The Implication Process

The tasks of the implication process (represented by the routine *Imply_and_check* in Figure 6.10) are

- Compute all values that can be uniquely determined by implication.
- Check for consistency and assign values.
- Maintain the *D-frontier* and the *J-frontier*.

We can view the implication process as a modified zero-delay simulation procedure. As in simulation, we start with some values to be assigned; these assignments may determine (imply) new values, and so on, until no more values are generated. All values to be assigned are processed via an *assignment queue* similar to the event queue used in simulation. Unlike simulation, where values only propagate forward (toward POs), here values may also propagate backward (toward PIs). An entry in the assignment queue has the form ($l$, $v'$, *direction*), where $v'$ is the value to be assigned to line $l$ and *direction* $\in$ {*backward, forward*}. To generate a test for the fault $l$ *s-a*-1, the initial two entries in the assignment queue are ($l$, 0, *backward*) and ($l$, $\overline{D}$, *forward*).

*Imply_and_check* retrieves in turn every entry in the assignment queue. The value $v'$ to be assigned to $l$ is first checked for consistency with the current value $v$ of $l$ (all values are initialized to $x$). An inconsistency is detected if $v \neq x$ and $v \neq v'$. (An exception is allowed for the faulty line, which gets a binary value to be propagated backward and an error value to be propagated forward.) A consistent value is assigned, then it is further processed according to its *direction*.

Backward propagation of values is illustrated in Figure 6.14. The right side of the figure shows the effects of the assignments made on the left side. An arrow next to a logic value shows the direction in which that value propagates. The assignment $a$=0 in Figure 6.14(c) causes $a$ to be added to the *J-frontier*. Figure 6.14(d) shows how backward propagation on a fanout branch may induce forward propagation on other fanout branches of the same stem.

Similarly, Figures 6.15 and 6.16 illustrate forward propagation of values. Note in Figure 6.15(d) how forward propagation on a gate input may induce backward propagation on another input of the same gate.

If after all values have been propagated, the *D-frontier* contains only one entry — say, $a$ — then the only way to propagate the error is through gate $a$. Figure 6.17 illustrates the implications resulting from this situation, referred to as *unique D-drive*.

### Global Implications

Let us consider the example in Figure 6.18(a). The *D-frontier* is {$d$,$e$}, so we do not have unique $D$-drive. We can observe, however, that no matter how we will decide to propagate the error (i.e., through $d$ or $e$ or both), eventually we will reach a unique $D$-drive situation, because the error must propagate through $g$. Then we can make the implications shown in Figure 6.18(b) [Akers 1976, Fujiwara and Shimono 1983].

While the implications previously described can be characterized as *local*, as they consist in propagating values from one line to its immediate successors or

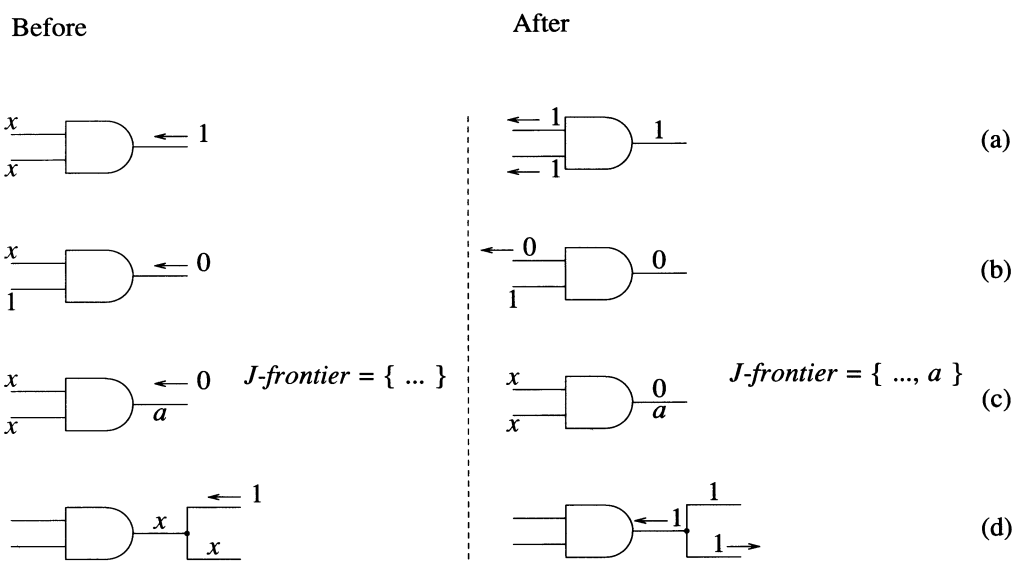



**Figure 6.14** Backward implications

predecessors, the implication illustrated in Figure 6.18 can be characterized as *global*, since it involves a larger area of the circuit and reconvergent fanout.

Next we analyze other global implications used in the SOCRATES system [Schulz *et al.* 1988, Schulz and Auth 1989]. Consider the circuit in Figure 6.19. Assume that $F = 1$ has just been assigned by backward propagation. No other values can be determined by local implications. But we can observe that, no matter how we decide to justify $F = 1$ (by $D = 1$ or $E = 1$), in either case we will imply $B = 1$. Thus we can conclude that $F = 1$ implies $B = 1$. This implication is "learned" during the preprocessing phase of SOCRATES by the following analysis. Simulating $B = 0$, we determine that it implies $F = 0$. Then $\overline{(F=0)}$ implies $\overline{(B=0)}$, that is, $F = 1$ implies $B = 1$.

The type of learning illustrated above is called *static*, because the implications determined are valid independent of other values in the circuit. SOCRATES also performs *dynamic learning* to determine global implications enabled by previously assigned values. For example, in the circuit in Figure 6.20, $F = 0$ implies $B = 0$ when $A = 1$ (because $B = 1$ implies $F = 1$ when $A = 1$).

**Reversing Incorrect Decisions**

Consider the problem of justifying a 0 on the output of an AND gate with three inputs — $a$, $b$, and $c$ — all currently with value $x$ (see Figure 6.21). Let us assume that the first decision — $a$=0 — has been proven incorrect. This shows that, independent of the values of $b$ and $c$, $a$ cannot be 0. Therefore, we can conclude that $a$ must be 1. Then before we try the next decision — $b$=0 — we should set $a$=1

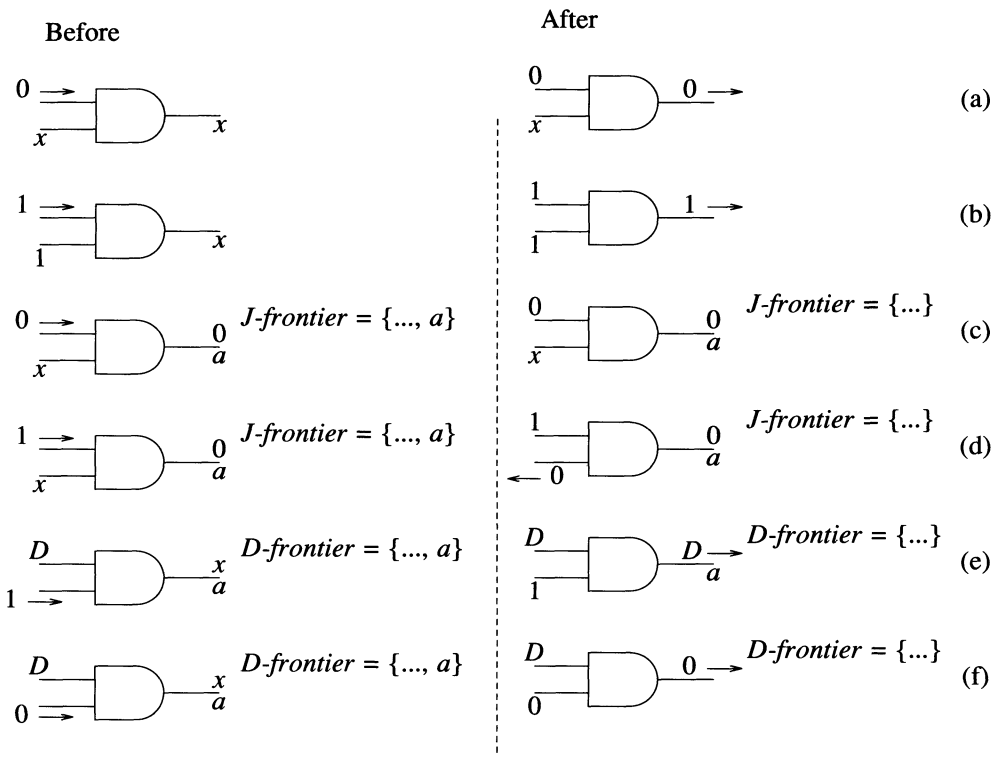



**Figure 6.15** Forward implications (binary values)

(rather than leave $a$=$x$) and process $a$=1 as an implication. Similarly, if the decision $b$=0 fails as well, we should set both $a$=1 and $b$=1 before trying $c$=0. The benefit of this technique of *reversing incorrect decisions* [Cha *et al.* 1978] is an increase in the number of implications.

**Error-Propagation Look-Ahead**

Consider the circuit and the values shown in Figure 6.22. The *D-frontier* is $\{a,b\}$. We can observe that, independent of the way we may try to propagate the errors, eventually the *D-frontier* will become empty, as $D$s cannot be driven through $e$ or $f$. This future state can be identified by checking the following necessary condition for successful error propagation.

Let an *x-path* denote a path all of whose lines have value $x$. Let $G$ be a gate on the *D-frontier*. The error(s) on the input(s) of $G$ can propagate to a PO $Z$ only if there exists at least one $x$-path between $G$ and $Z$ [Goel 1981].

Clearly, none of the gates on the *D-frontier* in Figure 6.22 satisfies this condition. The benefit of identifying this situation is that we can avoid all the decisions that are bound

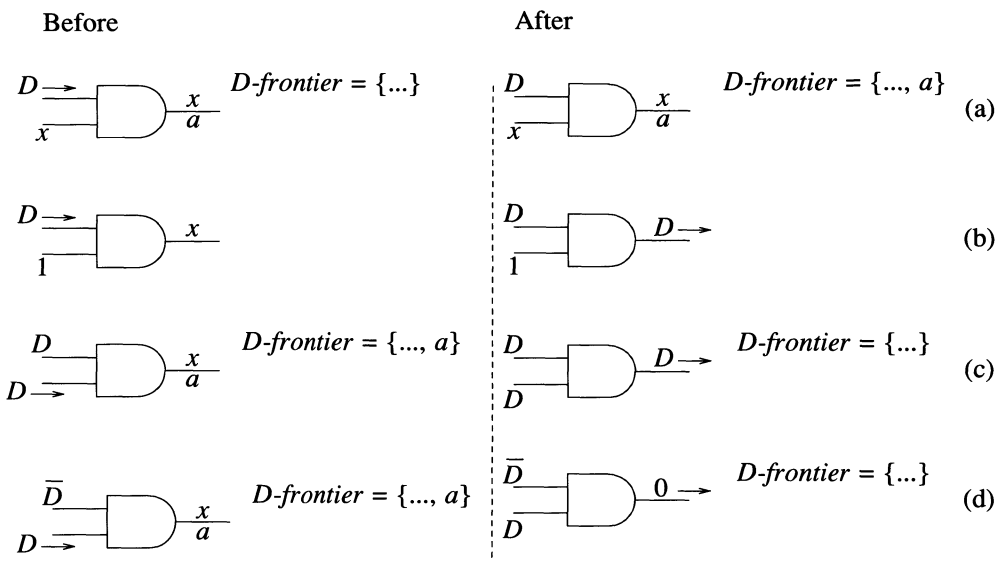



**Figure 6.16** Forward implications (error values)

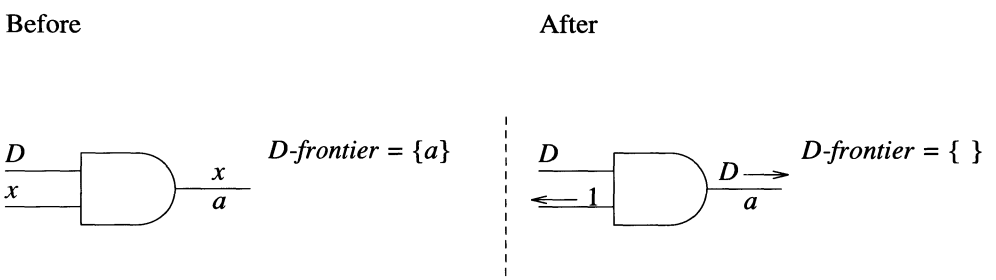



**Figure 6.17** Unique *D*-drive

eventually to fail and their associated backtracking. Thus by using this look-ahead technique we may prune the decision tree by recognizing states from which any further decision will lead to a failure.

### 6.2.1.2 Algorithms

Many of the concepts presented in the previous section are common to a class of TG algorithms generally referred to as *path-sensitization algorithms*. In this section we discuss specific algorithms of this class.

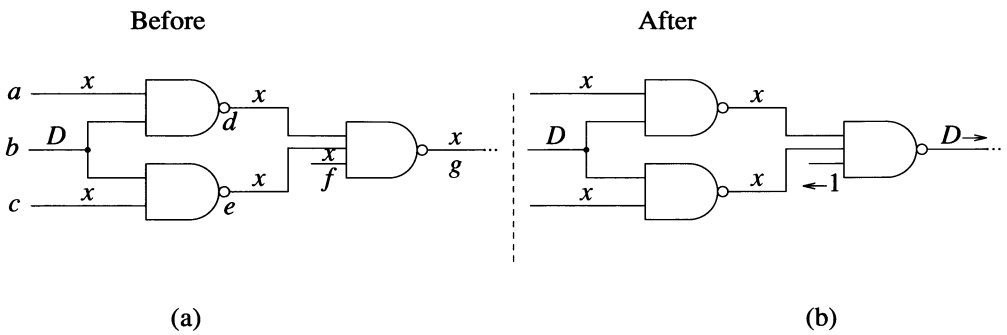



**Figure 6.18** Future unique $D$-drive

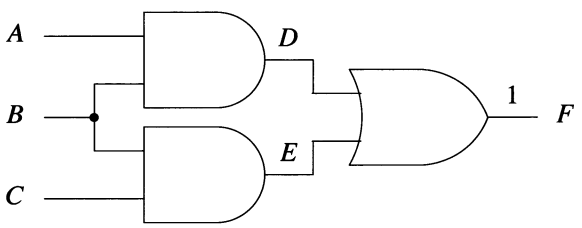



**Figure 6.19** Global implication: $F = 1$ implies $B = 1$

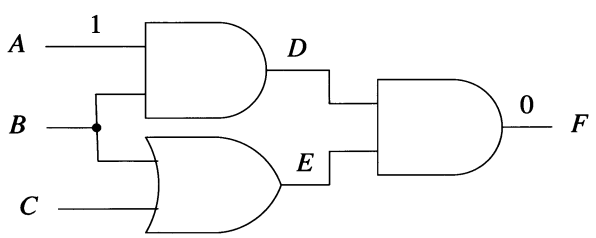



**Figure 6.20** Global implication: $F = 0$ implies $B = 0$ when $A = 1$

### The $D$-Algorithm

Figure 6.23 presents our version of the classical $D$-algorithm [Roth 1966, Roth *et al.* 1967]. It follows the general outline shown in Figure 6.10. For the sake of

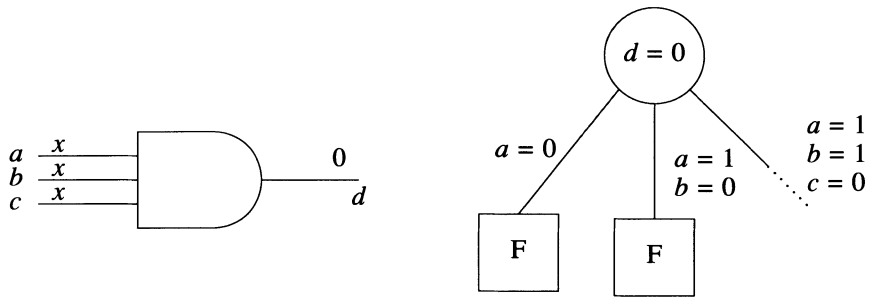



**Figure 6.21** Reversing incorrect decisions

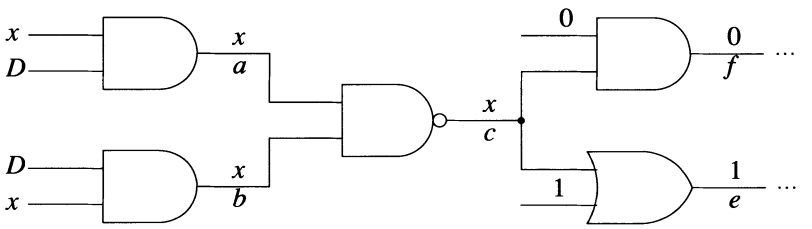



**Figure 6.22** The need for look-ahead in error propagation

simplicity, we assume that error propagation is always given priority over justification problems; however, this assumption is not essential (see Section 6.2.1.3).

The term "assign" should be understood as "add the value to be assigned to the assignment queue" rather than an immediate assignment. Recall that all the assignments are made and further processed by *Imply_and_check*.

A characteristic feature of the *D*-algorithm is its ability to propagate errors on several reconvergent paths. This feature, referred to as *multiple-path sensitization*, is required to detect certain faults that otherwise (i.e., sensitizing only a single path) would not be detected [Schneider 1967].

**Example 6.6:** Let us apply the *D*-algorithm for the circuit and the fault shown in Figure 6.24(a). Figure 6.24(b) traces the value computation and Figure 6.24(c) depicts the decision tree. The content of a decision node corresponding to an error propagation problem shows the associated *D-frontier*. A branch emanating from such a decision node shows the decision taken, that is, the gate selected from the *D-frontier* for error propagation. (Remember that when backtracking occurs, the *D-frontier* should be restored to its state before the incorrect decision.) Note how the *D*-algorithm first tried to propagate the error solely through *i*, then through both *i* and

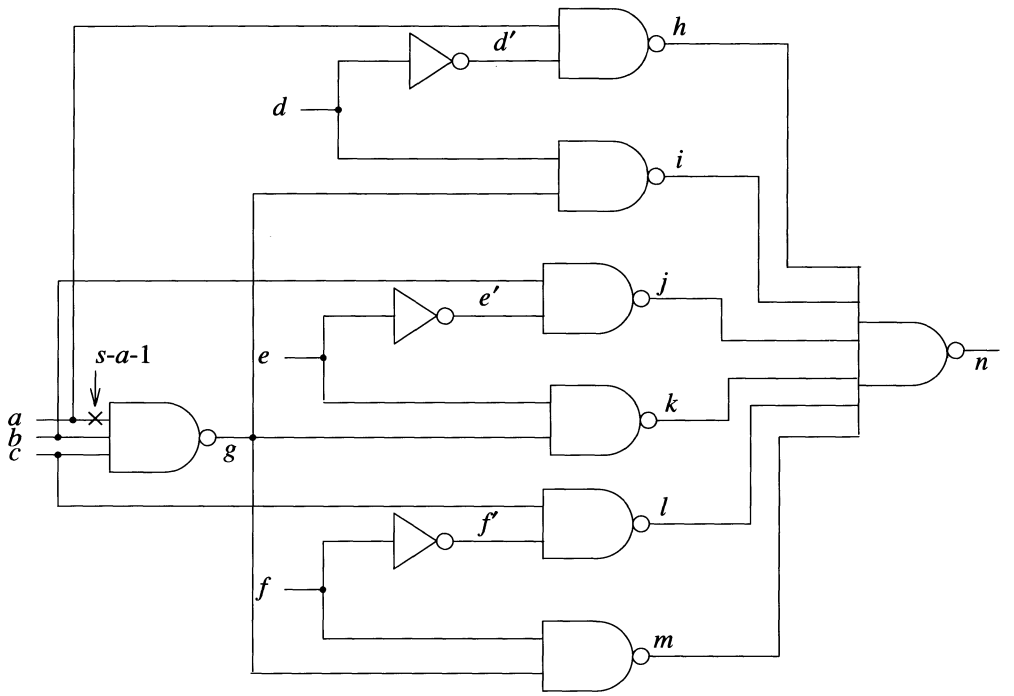
```
D-alg()
begin
   if Imply_and_check() = FAILURE then return FAILURE
   if (error not at PO) then
     begin
        if D-frontier = ∅ then return FAILURE
        repeat
           begin
              select an untried gate (G) from D-frontier
              c = controlling value of G
              assign c̄ to every input of G with value x
              if D-alg() = SUCCESS then return SUCCESS
           end
        until all gates from D-frontier have been tried
        return FAILURE
     end
   /* error propagated to a PO */
   if J-frontier = ∅ then return SUCCESS
   select a gate (G) from the J-frontier
   c = controlling value of G
   repeat
      begin
         select an input (j) of G with value x
         assign c to j
         if D-alg() = SUCCESS then return SUCCESS
         assign c̄ to j   /* reverse decision */
      end
   until all inputs of G are specified
   return FAILURE
end
```

**Figure 6.23** The *D*-algorithm

$k$, and eventually succeeded when all three paths from $g$ (through $i$, $k$, and $m$) were simultaneously sensitized. □

### The 9-V Algorithm

The 9-V algorithm [Cha *et al.* 1978] is similar to the *D*-algorithm. Its main distinguishing feature is the use of nine values [Muth 1976]. In addition to the five composite values of the *D*-algorithm, the 9-V algorithm employs four *partially specified composite values*. The $x$ value of the *D*-algorithm is totally unspecified, that is, neither $v$ nor $v_f$ is known. For a partially specified composite value $v/v_f$, either $v$ is binary and $v_f$ is unknown ($u$) or vice versa. For example, $1/u$ represents both 1/0 and 1/1. This means that $1/u$ can be either $D$ or 1. Figure 6.25 shows the partially




(a)

**Figure 6.24**

specified composite values and the sets of completely specified composite values they represent. The totally unspecified value $x$ is $u/u$ and represents the set $\{0,1,D,\overline{D}\}$.

A logic operation between two composite values can be carried out by separately processing the good and the faulty circuit values, and then composing the results. For example $D.x = 1/0 \;.\; u/u = (1.u)/(0.u) = u/0$. (In practice, logic operations using the nine composite values are defined by tables.) Note that using only the five values of the $D$-algorithm the result of $D.x$ is $x$. The 9-valued system provides more information as $D.x = u/0$ shows that the result is 0 or $D$.

When the 9-V algorithm tries to drive a $D$ through a gate $G$ with controlling value $c$, the value it assigns to the unspecified inputs of $G$ corresponds to the set $\{\overline{c},D\}$. Similarly, the propagation of a $\overline{D}$ is enabled by values corresponding to the set $\{\overline{c},\overline{D}\}$. For example, to drive a $\overline{D}$ through an AND gate, the unspecified inputs are assigned a $u/1$ value (which is 1 or $\overline{D}$), and it is the task of the implication process to determine whether this value eventually becomes 1 or $\overline{D}$. A partially specified composite value $u/b$ or $b/u$ (where $b$ is binary) assigned to a PI is immediately transformed to $b/b$, because the PI cannot propagate fault effects. The benefit of the flexibility provided by

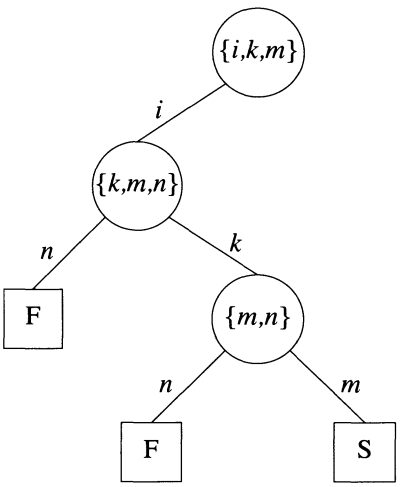
| Decisions | Implications | |
|---|---|---|
| | $a$=0 | Activate the fault |
| | $h$=1 | |
| | $b$=1 | Unique $D$-drive through $g$ |
| | $c$=1 | |
| | $g$=$D$ | |
| $d$=1 | | Propagate through $i$ |
| | $i$=$\bar{D}$ | |
| | $d'$=0 | |
| $j$=1 | | Propagate through $n$ |
| $k$=1 | | |
| $l$=1 | | |
| $m$=1 | | |
| | $n$=$D$ | |
| | $e'$=0 | |
| | $e$=1 | |
| | $k$=$\bar{D}$ | Contradiction |
| $e$=1 | | Propagate through $k$ |
| | $k$=$\bar{D}$ | |
| | $e'$=0 | |
| | $j$=1 | |
| $l$=1 | | Propagate through $n$ |
| $m$=1 | | |
| | $n$=$D$ | |
| | $f'$=0 | |
| | $f$=1 | |
| | $m$=$\bar{D}$ | Contradiction |
| $f$=1 | | Propagate through $m$ |
| | $m$=$\bar{D}$ | |
| | $f'$=0 | |
| | $l$=1 | |
| | $n$=$D$ | |

(b)

**Figure 6.24** (Continued)

the partially specified composite values is that it reduces the amount of search done for multiple path sensitization.

**Example 6.7:** Let us redo the problem from Example 6.6 using the 9-V algorithm. Figure 6.26(a) traces the value computation and Figure 6.26(b) shows the corresponding decision tree. Now the same test is generated without backtracking. □




(c)

**Figure 6.24** (Continued)

| | |
|---|---|
| $0/u$ | $\{0,\bar{D}\}$ |
| $1/u$ | $\{D,1\}$ |
| $u/0$ | $\{0,D\}$ |
| $u/1$ | $\{\bar{D},1\}$ |

**Figure 6.25** Partially specified composite values

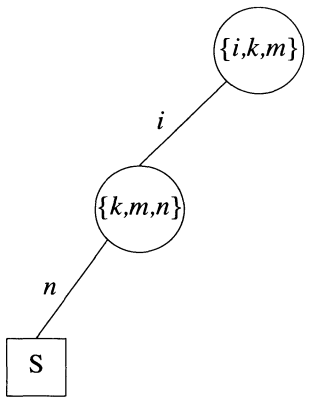
The main difference between the $D$-algorithm and 9-V algorithm can be summarized as follows. Whenever there are $k$ possible paths for error propagation, the $D$-algorithm may eventually try all the $2^k-1$ combinations of paths. The 9-V algorithm tries only one path at a time, but without precluding simultaneous error propagation on the other $k-1$ paths. This is made possible by the partially specified composite values that denote potential error propagation. Thus in a situation where the $D$-algorithm may enumerate up to $2^k-1$ combinations of paths, the 9-V algorithm will enumerate at most $k$ ways of error propagation.

### Single-Path Sensitization

Experience has shown that faults whose detection is possible only with multiple-path sensitization are rare in practical circuits (see, for example, the results presented in

| Decisions | Implications | |
|---|---|---|
| | $a$=0 | Activate the fault |
| | $h$=1 | |
| | $b$=1 | Unique $D$-drive through $g$ |
| | $c$=1 | |
| | $g$=$D$ | |
| | $i$=$u$/1 | |
| | $k$=$u$/1 | |
| | $m$=$u$/1 | |
| $d$=1 | | Propagate through $i$ |
| | $i$=$\overline{D}$ | |
| | $d'$=0 | |
| | $n$=1/$u$ | |
| $l$=$u$/1 | | Propagate through $n$ |
| $j$=$u$/1 | | |
| | $n$=$D$ | |
| | $f'$=$u$/0 | |
| | $f$=1 | |
| | $f'$=0 | |
| | $e'$=$u$/0 | |
| | $e$=1 | |
| | $e'$=0 | |
| | $k$=$\overline{D}$ | |
| | $m$=$\overline{D}$ | |

(a)




(b)

**Figure 6.26** Execution trace of the 9-V algorithm on the problem of Example 6.6

[Cha *et al.* 1978]). To reduce computation time, TG algorithms are often restricted to *single-path sensitization.*

To restrict the $D$-algorithm (Figure 6.23) to single-path sensitization, it should be modified as follows. After we select a gate from the *D-frontier* and propagate an error to its output, we consider only that $D$ or $\overline{D}$ for further propagation and ignore the other gates from the *D-frontier.* In this way we force the algorithm to propagate errors only on single paths.

**PODEM**

The goal of any (fault-oriented) TG algorithm is to find a test for a specified fault, that is, an input vector that detects the fault. Although the test belongs to the space of all input vectors, the search process of the algorithms discussed so far takes place in a different space. A decision in this search process consists of selecting either a gate

from the *D-frontier* or a way of justifying the value of a gate from the *J-frontier*. These decisions are eventually mapped into PI values, but the search process is an indirect one.

PODEM (Path-Oriented Decision Making) [Goel 1981] is a TG algorithm characterized by a *direct search* process, in which decisions consist only of PI assignments. We have seen that the problems of fault activation and error propagation lead to sets of line-justification problems. PODEM treats a value $v_k$ to be justified for line $k$ as an *objective* $(k,v_k)$ to be achieved via PI assignments [Snethen 1977]. A backtracing procedure (Figure 6.27) maps a desired objective into a PI assignment that is likely to contribute to achieving the objective. Let $(j, v_j)$ be the PI assignment returned by *Backtrace* $(k, v_k)$, and let $p$ be the inversion parity of the path followed from $k$ to $j$. All lines on this path have value $x$, and the value $v_j$ to be assigned and the objective value $v_k$ satisfy the relation $v_k = v_j \oplus p$. Note that no values are assigned during backtracing. Values are assigned only by simulating PI assignments.

```
Backtrace (k,v_k)
/* map objective into PI assignment */
begin
    v=v_k
    while k is a gate output
        begin
            i = inversion of k
            select an input (j) of k with value x
            v=v⊕i
            k=j
        end
    /* k is a PI */
    return (k,v)
end
```

**Figure 6.27** Backtracing of an objective

**Example 6.8:** Consider the circuit shown in Figure 6.28 and the objective $(f,1)$. Assume that *Backtrace*$(f,1)$ follows the path $(f,d,b)$ and returns $(b,1)$. Simulating the assignment $b$=1 does not achieve the objective $(f,1)$. Executing again *Backtrace*$(f,1)$ results in following the path $(f,d,c,a)$ and leads to $(a,0)$. Now simulating the assignment $a$=0 achieves $f$=1. □

Objectives are selected (see Figure 6.29) so that first the target fault is activated; then the resulting error is propagated towards a PO.

Figure 6.30 outlines the overall structure of PODEM. It uses the same five values — 0, 1, $x$, $D$, and $\overline{D}$ — as the $D$-algorithm. Initially all values are $x$. Non-$x$ values are generated only by simulating PI assignments. This 5-valued simulation is the task of the routine *Imply*, which also creates the initial $D$ or $\overline{D}$ when the fault is activated, and maintains the *D-frontier*. At every level of recursion, PODEM starts by analyzing

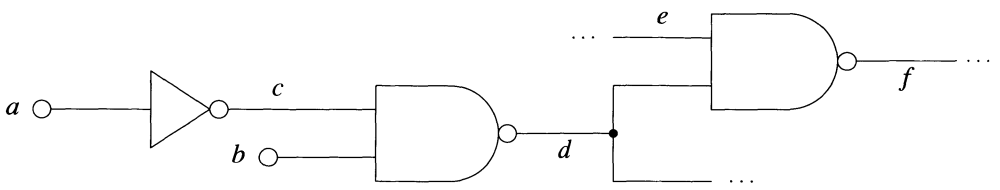



**Figure 6.28**

```
Objective()
begin
    /* the target fault is l s-a-v */
    if (the value of l is x) then return (l,v̄)
    select a gate (G) from the D-frontier
    select an input (j) of G with value x
    c = controlling value of G
    return (j,c̄)
end
```

**Figure 6.29** Selecting an objective

values previously established, with the goal of identifying a SUCCESS or a FAILURE state. SUCCESS is returned if a PO has a $D$ or $\overline{D}$ value, denoting that an error has been propagated to a PO. FAILURE is returned if the current values show that generating a test is no longer possible. This occurs when either of the following conditions applies:

- The target fault $l$ *s-a-v* cannot be activated, since line $l$ has value $v$.
- No error can be propagated to a PO, either because the *D-frontier* is empty or because the error propagation look-ahead shows that it will become empty.

If PODEM cannot immediately determine SUCCESS or FAILURE, it generates an objective $(k,v_k)$ that is mapped by backtracing into a PI assignment. The assignment $j=v_j$ is then simulated by *Imply* and a new level of recursion is entered. If this fails, PODEM backtracks by reversing the decision $j=v_j$ to $j=\overline{v}_j$. If this also fails, then $j$ is set to $x$ and PODEM returns FAILURE.

The selection of a gate from the *D-frontier* (done in *Objective*) and the selection of an unspecified gate input (done in *Objective* and in *Backtrace*) can be, in principle, arbitrary. Selection criteria that tend to increase the efficiency of the algorithm are discussed in Section 6.2.1.3.

```
PODEM()
begin
    if (error at PO) then return SUCCESS
    if (test not possible) then return FAILURE
    (k,v_k) = Objective()
    (j,v_j) = Backtrace(k,v_k)  /* j is a PI */
    Imply (j,v_j)
    if PODEM() = SUCCESS then return SUCCESS
    /* reverse decision */
    Imply (j,v̄_j)
    if PODEM() = SUCCESS then return SUCCESS
    Imply (j,x)
    return FAILURE
end
```

**Figure 6.30** PODEM

**Example 6.9:** Let us apply PODEM to the problem from Example 6.6. Figure 6.31(a) traces a possible execution of PODEM, showing the objectives, the PI assignments determined by backtracing objectives, the implications generated by simulating PI assignments, and the corresponding *D-frontier*. Note that the assignment $e$=0 causes the PO $n$ to have a binary value, which makes the $x$-path check fail; this shows that generating a test for the target fault is no longer possible and leads to backtracking by reversing the incorrect decision. Also note that PODEM handles multiple-path sensitization without any special processing. □

Since a decision in PODEM is choosing a value $v_j \in \{0,1\}$ to be assigned to the PI $j$, the decision tree of PODEM is a binary tree in which a node corresponds to a PI $j$ to be assigned and a branch emanating from the node $j$ is labeled with the selected value $v_j$. Figure 6.31(b) shows the decision tree for Example 6.9.

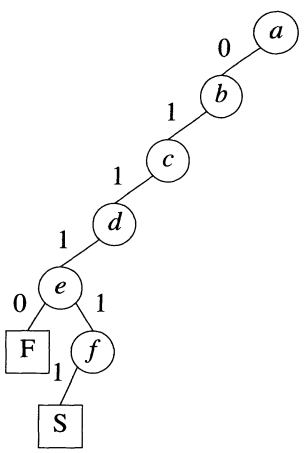
As illustrated by the structure of its decision tree, the PODEM search process is based on *direct implicit enumeration* of the possible input vectors. As in the *D*-algorithm, the search is exhaustive, such that FAILURE is eventually returned only if no test exists for the target fault (see Problem 6.12). (In practice, the amount of search is bounded by user-imposed limits.)

PODEM differs from the TG algorithms patterned after the schema given in Figure 6.10 in several aspects. In PODEM, values are computed only by forward implication of PI assignments. Consequently, the computed values are always self-consistent and all values are justified. Therefore, PODEM does not need

- consistency check, as conflicts can never occur;
- the *J-frontier*, since there are no values that require justification;
- backward implication, because values are propagated only forward.

| Objective | PI Assignment | Implications | *D-frontier* | |
|---|---|---|---|---|
| $a$=0 | $a$=0 | $h$=1 | $g$ | |
| $b$=1 | $b$=1 | | $g$ | |
| $c$=1 | $c$=1 | $g$=$D$ | $i,k,m$ | |
| $d$=1 | $d$=1 | $d'$=0<br>$i$=$\bar{D}$ | $k,m,n$ | |
| $k$=1 | $e$=0 | $e'$=1<br>$j$=0<br>$k$=1<br>$n$=1 | $m$ | $x$-path check fails |
| | $e$=1 | $e'$=0<br>$j$=1<br>$k$=$\bar{D}$<br>$n$=$x$ | $m,n$ | reversal |
| $l$=1 | $f$=1 | $f'$=0<br>$l$=1<br>$m$=$\bar{D}$<br>$n$=$D$ | | |

(a)




(b)

**Figure 6.31** Execution trace of PODEM on the problem of Example 6.6

Another important consequence of the direct search process is that it allows PODEM to use a simplified backtracking mechanism. Recall that backtracking involves restoring the state of computation to that existing before an incorrect decision. In the TG algorithms previously described, state saving and restoring is an explicit (and

time-consuming) process. In PODEM, since the state depends only on PI values and any decision deals only with the value of one PI, any change in state is easily computed by propagating a new PI value. Thus *backtracking is implicitly done by simulation* rather than by an explicit save/restore process.

Because of these advantages, PODEM is much simpler than other TG algorithms, and this simplicity is a key factor contributing to its success in practice. Experimental results presented in [Goel 1981] show that PODEM is generally faster than the *D*-algorithm. PODEM is the core of a TG system [Goel and Rosales 1981] successfully used to generate tests for large circuits.

**FAN**

The FAN (Fanout-Oriented TG) algorithm [Fujiwara and Shimono 1983] introduces two major extensions to the backtracing concept of PODEM:

- Rather than stopping at PIs, *backtracing in FAN may stop at internal lines.*
- Rather than trying to satisfy one objective, FAN uses a *multiple-backtrace* procedure that attempts to simultaneously satisfy a set of objectives.

The internal lines where FAN stops backtracing are defined as follows. A line that is reachable from (i.e., directly or indirectly fed by) at least one stem is said to be *bound.* A line that is not bound is said to be *free.* A *head line* is a free line that directly feeds a bound line. For example, in the circuit of Figure 6.32, *A, B, C, E, F, G, H,* and *J* are free lines, *K, L,* and *M* are bound lines, and *H* and *J* are head lines. Since the subcircuit feeding a head line *l* is fanout-free, a value of *l* can be justified without contradicting any other value previously assigned in the circuit. Thus backtracing can stop at *l*, and the problem of justifying the value of *l* can be postponed for the last stage of the TG algorithm. The following example illustrates how this technique may simplify the search process.

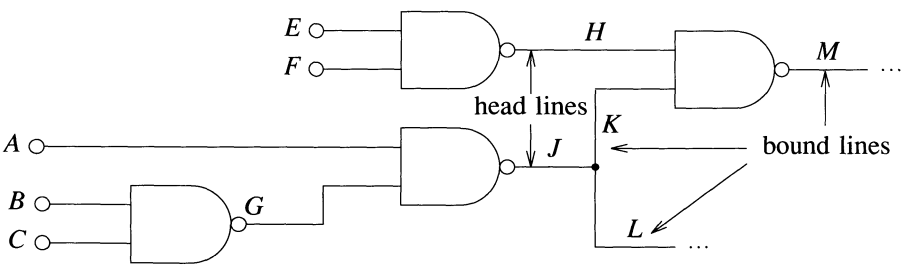



**Figure 6.32** Example of head lines

**Example 6.10:** For the circuit of Figure 6.32, suppose that at some point in the execution of PODEM, we want to set *J*=0. Moreover, we assume that with the PI assignments previously made, setting *J*=0 causes the *D-frontier* to become empty and hence leads to a FAILURE state. Figure 6.33(a) shows the portion of the PODEM decision tree corresponding to this failing search. Since *J* is a head line, backtracing in

FAN stops at $J$ and the assignment $J$=0 is tried *before* any attempt is made to justify $J$=0. This leads to the reduced decision tree in Figure 6.33(b). □

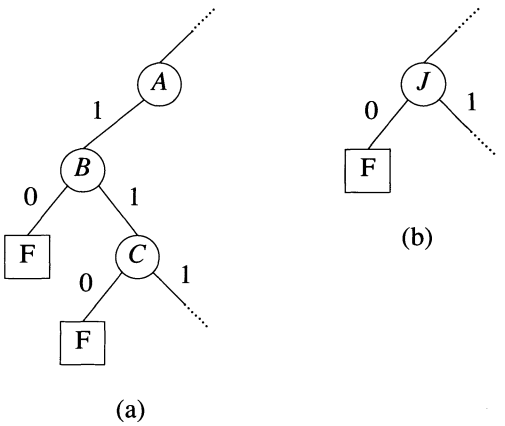



**Figure 6.33** Decision trees (a) For PODEM (backtracing to PIs) (b) For FAN (backtracing to head lines)

Recall that the fault-activation and the error-propagation problems map into a set of line-justification problems. In PODEM, each one of these problems becomes in turn an objective that is individually backtraced. But a PI assignment satisfying one objective may preclude achieving another one, and this leads to backtracking. To minimize this search, the multiple backtrace procedure of FAN (*Mbacktrace* in Figure 6.34) starts with a set of objectives (*Current_objectives*) and it determines an assignment $k$=$v_k$ that is likely either to contribute to achieving a subset of the original objectives or to show that some subset of the original objectives cannot be simultaneously achieved.

The latter situation may occur only when different objectives are backtraced to the same stem with conflicting values at its fanout branches (see Figure 6.35). To detect this, *Mbacktrace* stops backtracing when a stem is reached and keeps track of the number of times a 0-value and a 1-value have been requested on the stem.

*Mbacktrace* processes in turn every current objective until the set *Current_objectives* is exhausted. Objectives generated for head lines reached during this process are stored in the set *Head_objectives*. Similarly, the set *Stem_objectives* stores the stems reached by backtracing. After all current objectives have been traced, the highest-level stem from *Stem_objectives* is analyzed. Selecting the highest-level stem guarantees that all the objectives that could depend on this stem have been backtraced. If the stem $k$ has been reached with conflicting values (and if $k$ cannot propagate the effect of the target fault), then *Mbacktrace* returns the objective ($k$, $v_k$), where $v_k$ is the most requested value for $k$. Otherwise the backtracing is restarted from $k$. If no stems have been reached, then *Mbacktrace* returns an entry from the set *Head_objectives*.

```
Mbacktrace (Current_objectives)
begin
   repeat
      begin
         remove one entry (k,v_k) from Current_objectives
         if k is a head line
            then add (k,v_k) to Head_objectives
         else if k is a fanout branch then
                  begin
                     j = stem(k)
                     increment number of requests at j for v_k
                     add j to Stem_objectives
                  end
               else /* continue tracing */
                  begin
                     i = inversion of k
                     c = controlling value of k
                     if (v_k ⊕ i = c) then
                        begin
                           select an input (j) of k with value x
                           add (j,c) to Current_objectives
                        end
                     else
                        for every input (j) of k with value x
                           add (j,c̄) to Current_objectives
                  end
      end
   until Current_objectives = ∅
   if Stem_objectives ≠ ∅ then
     begin
        remove the highest-level stem (k) from Stem_objectives
        v_k = most requested value of k
        if (k has contradictory requirements and
            k is not reachable from target fault)
           then return (k,v_k)
        add (k,v_k) to Current_objectives
        return Mbacktrace (Current_objectives)
     end
   remove one objective (k,v_k) from Head_objectives
   return (k,v_k)
end
```

**Figure 6.34** Multiple backtrace

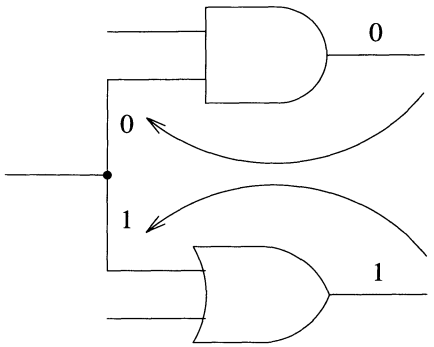



**Figure 6.35** Multiple backtrace generating conflicting values on a stem

**Example 6.11:** Consider the circuit in Figure 6.36(a). Figure 6.36(b) illustrates the execution of *Mbacktrace*, starting with (*I*,1) and (*J*,0) as current objectives. After all the current objectives are exhausted for the first time, *Stems_objectives* = {*A*,*E*}. Since the highest-level stem (*E*) has no conflicting requirements, backtracing is restarted from *E*. Next time when *Current_objectives* = ∅, stem *A* is selected for analysis. Because *A* has been reached with 1-value from *A1* and with 0-value from *A2*, *Mbacktrace* returns (*A*,*v*), where *v* can be either 0 or 1 (both values have been requested only once). □

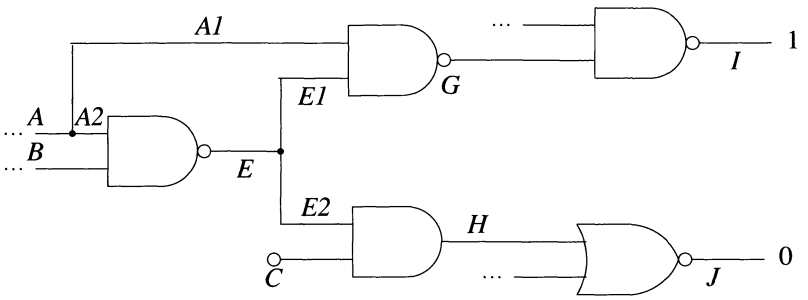
Figure 6.37 outlines a recursive version of FAN. Its implication process (denoted by *Imply_and_check*) is the same as the one described in Section 6.2.1.1. Unlike PODEM, implication propagates values both forward and backward; hence unjustified values may exist. FAN recognizes SUCCESS when an error has been propagated to a PO and all the bound lines have been justified. Then it justifies any assigned head line; recall that this justification process cannot cause conflicts.

If FAN cannot immediately identify a SUCCESS or a FAILURE state, it marks all the unjustified values of bound lines as current objectives, together with the values needed for the *D*-drive operation through one gate from the *D-frontier*. From these objectives, *Mbacktrace* determines an assignment for a stem or a head line to be tried next. The decision process is similar to the one used in PODEM. Experimental results presented in [Fujiwara and Shimono 1983] show that FAN is more efficient than PODEM. Its increased speed is primarily caused by a significant reduction in backtracking.

**Other Algorithms**

Next we will discuss other TG algorithms which extend the concept of head lines by identifying higher-level lines whose values can be justified without conflicts and using them to stop the backtracing process.

A *total-reconvergence line* used in the TOPS (*Top*ological *S*earch) algorithm [Kirkland and Mercer 1987] is the output *l* of a subcircuit *C* such that all paths between any line in *C* and any PO go through *l* (see Figure 6.38). In other words, cutting *l* would




(a)

| *Current_objectives* | *Processed entry* | *Stem_objectives* | *Head_objectives* |
|---|---|---|---|
| (*I*,1),(*J*,0) | (*I*,1) | | |
| (*J*,0),(*G*,0) | (*J*,0) | | |
| (*G*,0),(*H*,1) | (*G*,0) | | |
| (*H*,1),(*A1*,1),(*E1*,1) | (*H*,1) | | |
| (*A1*,1),(*E1*,1),(*E2*,1),(*C*,1) | (*A1*,1) | *A* | |
| (*E1*,1),(*E2*,1),(*C*,1) | (*E1*,1) | *A*,*E* | |
| (*E2*,1),(*C*,1) | (*E2*,1) | *A*,*E* | |
| (*C*,1) | (*C*,1) | *A*,*E* | *C* |
| ∅ | | *A* | *C* |
| (*E*,1) | (*E*,1) | *A* | *C* |
| (*A2*,0) | (*A2*,0) | *A* | *C* |
| ∅ | | *A* | *C* |

(b)

**Figure 6.36** Example of multiple backtrace

isolate *C* from the rest of the circuit. Clearly, a head line satisfies the definition of a total-reconvergence line; then *C* is a fanout-free subcircuit. However, the subcircuit *C* bounded by a total-reconvergence line may have fanout, but all such fanout must reconverge before *l*. Assuming that the function implemented by *C* is not a constant (0 or 1), any value assigned to *l* can be justified without conflicts with other values already assigned in the circuit. Thus the justification of a total-reconvergence line can be postponed in the same way as the justification of a head line. (Note that the justification of a total-reconvergence line may require backtracking).

Both the head lines and the total-reconvergence lines are found by a topological analysis of the circuit. A *backtrace-stop line* used in the FAST (*F*ault-oriented *A*lgorithm for *S*ensitized-path *T*esting) algorithm [Abramovici *et al.* 1986a] represents

```
FAN()
begin
   if Imply_and_check() = FAILURE then return FAILURE
   if (error at PO and all bound lines are justified) then
     begin
        justify all unjustified head lines
        return SUCCESS
     end
   if (error not at PO and D-frontier = ∅) then return FAILURE
   /* initialize objectives */
   add every unjustified bound line to Current_objectives
   select one gate (G) from the D-frontier
   c = controlling value of G
   for every input (j) of G with value x
      add (j,c̄) to Current_objectives
   /* multiple backtrace */
   (i,v_i) = Mbacktrace(Current_objectives)
   Assign(i,v_i)
   if FAN() = SUCCESS then return SUCCESS
   Assign(i,v̄_i)  /* reverse decision */
   if FAN() = SUCCESS then return SUCCESS
   Assign(i,x)
   return FAILURE
end
```

**Figure 6.37** FAN

another generalization of the head-line concept, based on an analysis that is both topological and functional. In FAST, a line $l$ is a backtrace-stop for value $v$, if the assignment $l = v$ can be justified without conflicts. For example, in Figure 6.39, $L$ is a backtrace-stop for 0, because $L = 0$ can be justified without assigning any line with reconvergent fanout (by $F = 0$ and $A = B = 1$). Note that $L$ is fed by reconvergent fanout, and it is not a total-reconvergence line. The identification of backtrace-stop lines will be explained in the next section.

### 6.2.1.3 Selection Criteria

The search process of any of the TG algorithms analyzed in this chapter involves decisions. A first type of decision is to select one of the several unsolved problems existing at a certain stage in the execution of the algorithm. A second type is to select one possible way to solve the selected problem. In this section we discuss *selection criteria* that are helpful in speeding up the search process. These selection criteria are based on the following principles:

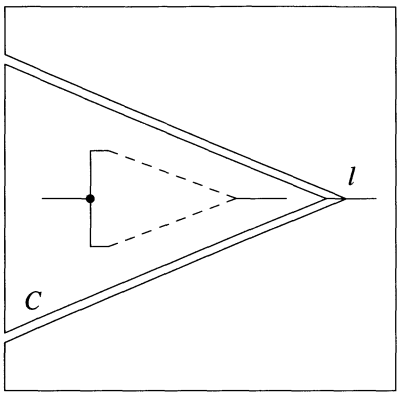



**Figure 6.38** Total-reconvergence line

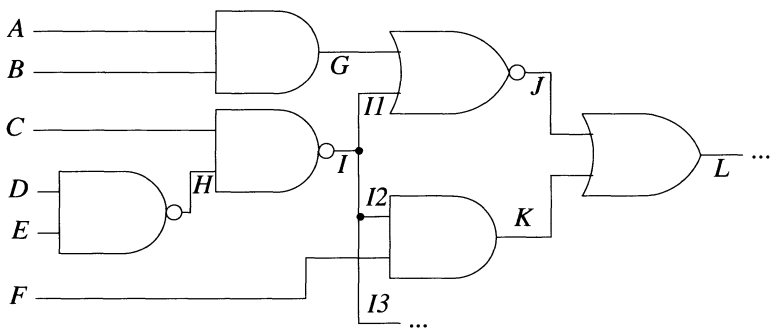



**Figure 6.39**

1. *Among different unsolved problems, first attack the most difficult one* (to avoid the useless time spent in solving the easier problems when a harder one cannot be solved).
2. *Among different solutions of a problem, first try the easiest one.*

Selection criteria differ mainly by the *cost functions* they use to measure "difficulty." Typically, cost functions are of two types:

- *controllability measures,* which indicate the relative difficulty of setting a line to a value;
- *observability measures,* which indicate the relative difficulty of propagating an error from a line to a PO.

Controllability measures can be used both to select the most difficult line-justification problem (say, setting gate $G$ to value $v$), and then to select among the unspecified inputs of $G$ the one that is the easiest to set to the controlling value of the gate. Observability measures can be used to select the gate from the *D-frontier* whose input error is the easiest to observe. Note that the cost functions should provide only *relative measures*. (Controllability and observability measures have also been used to compute "testability" measures, with the goal of estimating the difficulty of generating tests for specific faults or for the entire circuit [Agrawal and Mercer 1982]. Our goal here is different; namely we are looking for measures to guide the decision process of a TG algorithm.)

Controllability measures can also be used to guide the backtracing process of PODEM. Consider the selection of the gate input in the procedure *Backtrace* (Figure 6.27). While this selection can be arbitrary, according to the two principles stated above it should be based on the value $v$ needed at the gate input. If $v$ is the controlling (noncontrolling) value of the gate, then we select the input that is the easiest (most difficult) to set to $v$.

Of course, the savings realized by using cost functions should be greater than the effort required to compute them. In general, cost functions are *static*, i.e., they are computed by a preprocessing step and are not modified during TG.

**Distance-Based Cost Functions**

Any cost function should show that PIs are the easiest to control and POs are the easiest to observe. Taking a simplistic view, we can consider that the difficulty of controlling a line increases with its distance from PIs, and the difficulty of observing a line increases with its distance from POs. Thus we can measure the controllability of a line by its level, and its observability by its minimum distance from a PO. Although these measures are crude, it is encouraging to observe that using them still gives better results than not using any cost functions (i.e., taking random choices).

**Recursive Cost Functions**

The main drawback of the distance-based cost functions is that they do not take the logic into account. In this section we present more complex cost functions that do not suffer from this shortcoming [Rutman 1972, Breuer 1978, Goldstein 1979].

First we discuss controllability measures. For every signal $l$ we want to compute two cost functions, $C0(l)$ and $C1(l)$, to reflect, respectively, the relative difficulty of setting $l$ to 0 and 1. Consider an AND gate (see Figure 6.40) and assume that we know the $C0$ and $C1$ costs for every input. What can we say about the costs of the setting the output? To set the output $X$ to 0 it is enough to set any input to 0, so we can select the easiest one. Thus:

$$C0(X) = min\{C0(A), C0(B), C0(C)\} \tag{6.1}$$

To set $X$ to 1, we must simultaneously set all its inputs to 1. If $A$, $B$, and $C$ are independent (i.e., they do not depend on common PIs), then setting $X$ to 1 is the union of three disjoint problems, so we can add the three separate costs:

$$C1(X) = C1(A)+C1(B)+C1(C) \tag{6.2}$$

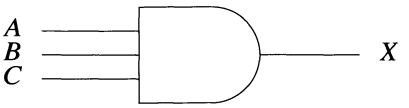



**Figure 6.40**

This formula may lead to less accurate results if applied when inputs of $X$ are not independent because of reconvergent fanout. In Figure 6.41(a), $B$ and $C$ are identical signals for which the cost of controlling them simultaneously should be the same as the cost of controlling each one of them. In Figure 6.41(b), $B$ and $C$ are complementary signals that can never be simultaneously set to the same value, so the correct value of $C1(X)$ should show that setting $X$=1 is impossible.

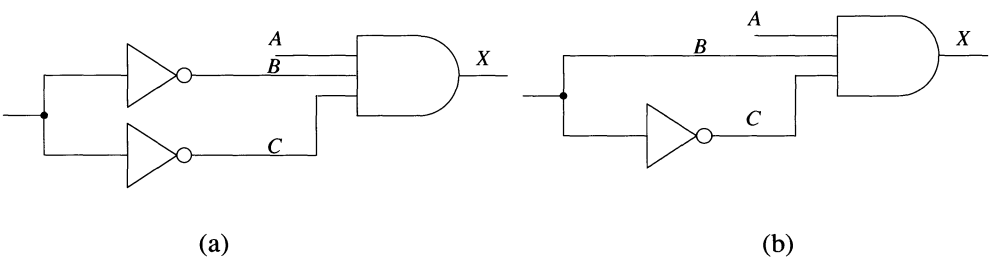



**Figure 6.41**

Nevertheless, to keep the computation of costs a simple process, we will use the simplifying assumption that costs of simultaneous line-setting problems are additive, and later we will try to compensate for potential inaccuracies by introducing *correction terms*.

Recursive formulas similar to (6.1) and (6.2) can be easily developed for other types of gates. The computation of controllability costs proceeds level by level. First $C0$ and $C1$ of every PI are set to 1. Then $C0$ and $C1$ are computed for every gate at level 1, then for every gate at level 2, and so on. In this way the costs of a line are computed only after the costs of its predecessors are known. Thus controllability costs are determined in one forward traversal of the circuit, and the algorithm is linear in the number of gates in the circuit.

Now let us discuss an observability measure $O(l)$ that reflects the relative difficulty of propagating an error from $l$ to a PO. Consider again the AND gate in Figure 6.40 and assume that we know $O(X)$. What can we say about the cost of observing the input $A$? To propagate an error from $A$ we must set both $B$ and $C$ to 1 (this propagates the error

from $A$ to $X$), and then we must propagate the error from $X$ to a PO. If these three problems are independent, then

$$O(A) = C1(B)+C1(C)+O(X) \tag{6.3}$$

Applying formula (6.3) when the problems of setting $B$ to 1, of setting $C$ to 1, and of propagating the error from $X$ to a PO are not independent leads to erroneous results (see Problem 6.14). Again, to keep the computation simple, we will use the simplifying assumption that problems are independent.

Now let us consider the problem of determining the observability of a stem $X$, knowing the observability of its fanout branches *X1*, *X2*, and *X3*. To propagate an error from $X$, we may choose any path starting at *X1*, *X2*, or *X3*. With the simplifying assumption that single-path propagation is possible, we can select the most observable path. Then

$$O(X) = min\{O(X1),O(X2),O(X3)\} \tag{6.4}$$

The computation of observability costs starts by setting the cost of every PO to 0. Then the circuit is traversed backward, applying formulas similar to (6.3) and (6.4) where appropriate. Now the cost of a line is computed only after the costs of all its successors are known. Note that computation of observability costs assumes that controllability costs are known. The algorithm is linear in the number of lines in the circuit.

**Fanout-Based Cost Functions**

We have seen that the existence of reconvergent fanout makes TG difficult. Consider the problem of justifying $X$=0 in the circuit in Figure 6.42(a). Clearly, we would like a selection criterion that would guide the TG algorithm to choose the solution $B$=0 rather than $A$=0, because $A$ has fanout and the side-effects of setting $A$ to 0 may cause a conflict. As Figure 6.42(b) shows, it is not enough to look only at the fanout count of the lines directly involved in the decision ($A$ and $B$).

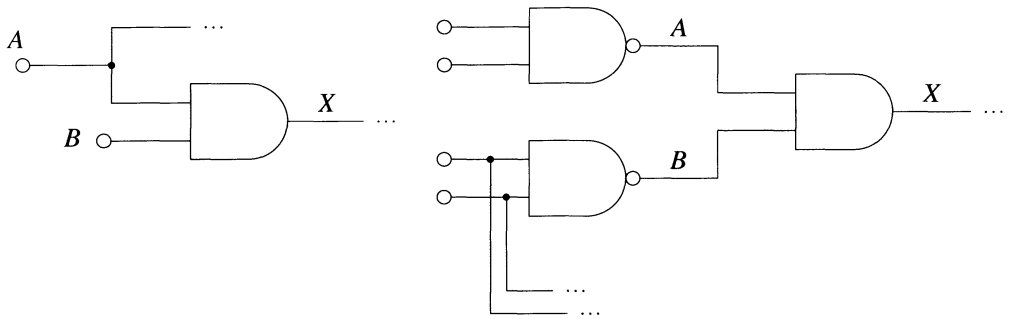
A fanout-based controllability measure *C(l)* reflects the relative potential for conflicts resulting from assigning a value to line $l$. The cost *C(l)* should depend both on the fanout count of $l$ and on the fanout count of the predecessors of $l$. Such a measure was first defined in [Putzolu and Roth 1971] as

$$C(l) = \sum_i C(i)+f_l-1 \tag{6.5}$$

where the summation is for all inputs of $l$, and $f_l$ is the fanout count of $l$. The cost of a PI $l$ is $f_l-1$. Note that *C(l)* = 0 iff $l$ is a free line (i.e., it is not reachable from a stem). Thus a 0 cost denotes a line that can be justified without conflicts. Applying (6.5) to the circuit in Figure 6.42(b), we obtain *C(A)* = 0, *C(B)* = 2, and *C(X)* = 2. Based on these values, a TG algorithm will select $A = 0$ to justify $X = 0$.

Formula (6.5) does not distinguish between setting a line to 0 and to 1. We obtain more accurate fanout-based measures if we use two controllability cost functions, *C0(l)* and *C1(l)*, to reflect the relative potential for conflict resulting from setting $l$ respectively to 0 and to 1 [Rutman 1972]. For an AND gate we have

$$C0(l) = \min_i\{C0(i)\}+f_l-1 \tag{6.6}$$




(a) (b)

**Figure 6.42**

and

$$C1(l) = \sum_i C1(i) + f_l - 1 \tag{6.7}$$

For a PI $l$, its $C0$ and $C1$ costs are set to $f_l - 1$. Formulas (6.6) and (6.7) also have the property that a 0 value indicates an assignment that can be done without conflicts. Thus these measures can identify the backtrace-stop lines used by FAST. Applying these measures to the circuit in Figure 6.42(a), we obtain

$$C0(A)=C1(A)=1,\ C0(B)=C1(B)=0,\ C0(X)=0,\ C1(X)=1.$$

Thus we correctly identify that $X$ can be set to 0 without conflicts, even if $X$ is fed by a stem.

Let us rewrite (6.1) and (6.2) in a more general form:

$$C0(l) = \min_i \{C0(i)\} \tag{6.1a}$$

$$C1(l) = \sum_i C1(i) \tag{6.2a}$$

and compare them with (6.6) and (6.7). We can observe that they are almost identical, except for the term $f_l - 1$. So we can consider (6.6) and (6.7) as being extensions of (6.1a) and (6.2a), with a *correction term* added to reflect the influence of fanout.

The measure presented in [Abramovici *et al.* 1986a] takes into account that only reconvergent fanout can cause conflicts and introduces correction terms that reflect the extent to which fanout is reconvergent.

Let us consider the circuit in Figure 6.43. The correction term for both $C0(A)$ and $C1(A)$ used in formulas (6.6) and (6.7) has value 1, since $A$ has a fanout count of 2. But if we analyze the effect of setting $A$ to 0 and to 1, we can see that $A$=0 has a much greater potential for conflicts than $A$=1. This is because $A$=0 results in $B$, $C$, $D$, and $E$ being set to binary values, while $A$=1 does not set any other gate output.

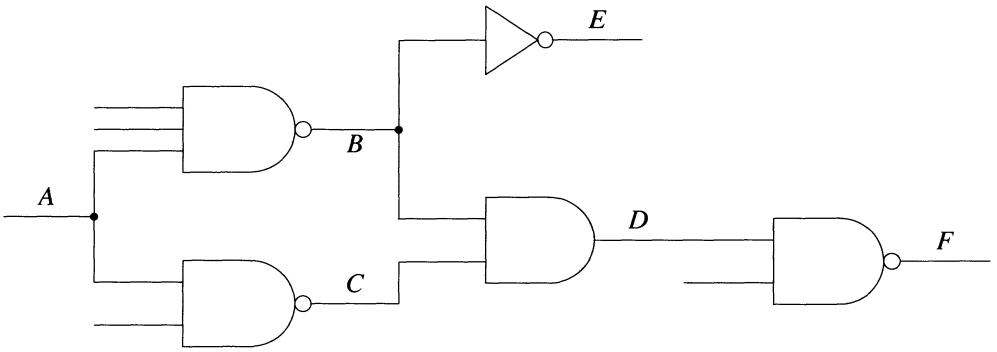



**Figure 6.43**

To account for this type of difference, Breuer[1978] has defined *side-effects cost functions*, $CS0(l)$ and $CS1(l)$, to reflect the relative potential for conflicts caused by setting $l$ to 0 and 1 respectively. These functions are computed by simulating the assignment $l=v$ ($v \in \{0,1\}$) in a circuit initialized to an all-$x$ state, then accounting for its effects as follows:

1. A gate whose output is set to a binary value increases the cost by 1.
2. A gate with $n$ inputs, whose output remains at $x$ but which has $m$ inputs set to a binary value, increases the cost by $m/n$.

For the circuit in Figure 6.43, the side-effects costs of setting $A$ are $CS0(A)=4\frac{1}{2}$ and $CS1(A)=\frac{1}{3}+\frac{1}{2}=\frac{5}{6}$.

Using the side-effects cost functions as correction terms we obtain (for an AND gate)

$$C0(l) = \min_i\{C0(i)\}+CS0(l) \tag{6.8}$$

and

$$C1(l) = \sum_i C1(i)+CS1(l) \tag{6.9}$$

**Concluding Remarks on Cost Functions**

The "best" controllability and observability measures could be obtained by actually solving the corresponding line-justification and error-propagation problems and measuring their computational effort. But such measures are useless, because their computation would be as expensive as the TG process they are meant to guide. Keeping the cost of computing the measures significantly lower than the TG cost requires that we accept less than totally accurate measures.

Suppose we compare two cost functions, $A$ and $B$, and we find that $A$ performs better than $B$ for some circuits. For other circuits, however, using $B$ leads to better results.

This type of "inconsistency" is inherent in the heuristic nature of cost functions. To compensate for this effect, Chandra and Patel [1989] suggest switching among cost functions during TG.

An approach that combines the computation of cost functions with a partial test generation is presented in [Ratiu *et al.* 1982]. Ivanov and Agarwal [1988] use *dynamic* cost functions, which are updated during test generation based on the already assigned values.

### 6.2.2 Fault-Independent ATG

The goal of the fault-oriented algorithms discussed in the previous section is to generate a test for a specified target fault. To generate a set of tests for a circuit, such an algorithm must be used with procedures for determining the initial set of faults of interest, for selecting a target fault, and for maintaining the set of remaining undetected faults. In this section we describe a fault-independent algorithm whose goal is to *derive a set of tests that detect a large set of SSFs without targeting individual faults.*

Recall that half of the SSFs along a path critical in a test $t$ are detected by $t$. Therefore it seems desirable to generate tests that produce long critical paths. Such a method of *critical-path TG* was first used in the LASAR (*L*ogic *A*utomated *S*timulus *A*nd *R*esponse) system [Thomas 1971]. A different critical-path TG algorithm is described in [Wang 1975].

The basic steps of a critical-path TG algorithm are

1. Select a PO and assign it a critical 0-value or 1-value (the value of a PO is always critical).
2. Recursively justify the PO value, trying to justify any critical value on a gate output by critical values on the gate inputs.

Figure 6.44 illustrates the difference between justifying a critical value and a noncritical value for an AND gate with three inputs. Critical values are shown in bold type (**0** and **1**). Unlike the primitive cubes used to justify a noncritical value, the input combinations used to justify a critical value — referred to as *critical cubes* — are always completely specified. This is because changing the value of a critical gate input should change the value of the gate output.

**Example 6.12:** Consider the fanout-free circuit in Figure 6.45(a). We start by assigning $G$=**0**. To justify $G$=**0**, we can choose ($E$,$F$)=**0**1 or ($E$,$F$)=1**0**. Suppose we select the first alternative. Then $E$=**0** is uniquely justified by ($A$,$B$)=**11**, and $F$=1 is uniquely justified by ($C$,$D$)=00. Now we have generated the test ($A$,$B$,$C$,$D$)=1100; Figure 6.45(b) shows the critical paths in this test.

Recall that our objective is to generate a set of tests for the circuit. To derive a new test with new critical lines, we return to the last gate where we had a choice of critical input combinations and select the next untried alternative. This means that now we justify $G$=**0** by ($E$,$F$)=1**0**. To justify $E$=1 we can arbitrarily select either $A$=0 or $B$=0. Exploring both alternatives to justify $F$=**0** results in generating two new tests, shown in Figures 6.45(c) and (d).

| A | B | C | Z |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 0 | *x* | *x* | 0 |
| *x* | 0 | *x* | 0 |
| *x* | *x* | 0 | 0 |

(a)

| A | B | C | Z |
|---|---|---|---|
| **1** | **1** | **1** | **1** |
| **0** | 1 | 1 | **0** |
| 1 | **0** | 1 | **0** |
| 1 | 1 | **0** | **0** |

(b)

**Figure 6.44** Line justification for an AND gate (a) By primitive cubes (b) By critical cubes

Now we have exhausted all the alternative ways to justify $G$=**0**. Two additional tests can be obtained by starting with $G$=**1** (see Figures 6.45(e) and (f)). The five generated tests form a complete test set for SSFs. □

Figure 6.46 outlines a recursive procedure *CPTGFF* for critical-path TG for fanout-free circuits. The set *Critical* contains the critical lines to be justified and their values. To generate a complete set of tests for a fanout-free circuit whose PO is $Z$, we use

add (Z,0) to *Critical*
*CPTGFF()*
add (Z,1) to *Critical*
*CPTGFF()*

(see Problem 6.18).

*CPTGFF* justifies in turn every entry in the set *Critical*. New levels of recursion are entered when there are several alternative ways to justify a critical value. Noncritical values are justified (without decisions) using the procedure *Justify* presented in Figure 6.4. A new test is recorded when all lines have been justified.

The decision process and its associated backtracking mechanism allow the systematic generation of all possible critical paths for a given PO value. As we did for the fault-oriented TG algorithms, we use a decision tree to trace the execution of the algorithm. A decision node (shown as a circle) corresponds to a critical line-justification problem for which several alternatives are available. Each alternative decision is denoted by a branch leaving the decision node. A terminal node (shown as a square) is reached when all lines have been justified. The number of terminal nodes is the number of tests being generated. Figure 6.47 gives the decision trees for Example 6.12.

The algorithm can be easily extended to multiple-output circuits that do not have reconvergent fanout (that is, every fanout branch of a stem leads to a different PO). Here, every cone of a PO is a fanout-free circuit that can be processed in turn as before. However, one problem appears because of logic common to two (or more) PO cones. Consider a line $l$ that feeds two POs, $X$ and $Y$. Suppose that we first generate all critical paths with $X$=**0** and with $X$=**1**. When we repeat the procedure for $Y$, at some point we will have to justify $l$=**0** or $l$=**1**. But all the possible ways to do this

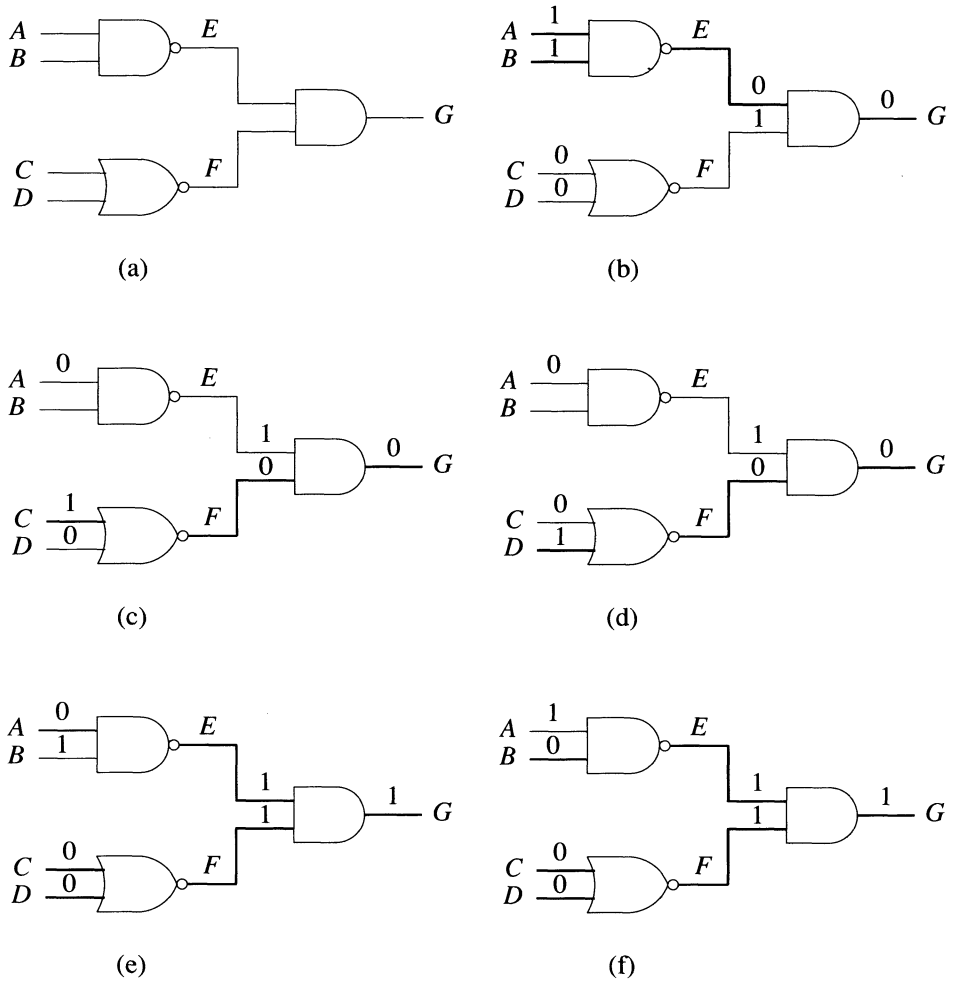



**Figure 6.45** Example of critical-path TG

have been explored in the cone of $X$, and repeating them may generate many unnecessary tests that do not detect any faults not detected by previously generated tests. This can easily be avoided by not trying to make critical a line $l$ that has already been marked as critical in a previous test (see Problem 6.19).

In attempting to extend critical-path TG to circuits with reconvergent fanout, we encounter problems caused by

- conflicts,
- self-masking,

```
CPTGFF()
begin
   while (Critical ≠ ∅)
      begin
         remove one entry (l,val) from Critical
         set l to val
         mark l as critical
         if l is a gate output then
            begin
               c = controlling value of l
               i = inversion of l
               inval = val ⊕ i
               if (inval = c̄)
                  then for every input j of l
                     add (j,c̄) to Critical
                  else
                     begin
                        for every input j of l
                           begin
                              add (j,c) to Critical
                              for every input k of l other than j
                                 Justify (k,c̄)
                              CPTGFF()
                           end
                        return
                     end
            end
      end
   /* Critical = ∅ */
   record new test
   return
end
```

**Figure 6.46** Critical-path TG for fanout-free circuits

- multiple-path sensitization,
- overlap among PO cones.

In the remainder of this section we discuss these problems, but because of the level of detail involved, we will not present a complete critical-path TG algorithm.

Conflicts appear from attempts to solve a set of simultaneous line-justification problems. As in fault-oriented TG algorithms, conflicts are dealt with by backtracking. A new aspect of this problem is that some conflicts may be caused by trying to justify a critical value by critical cubes. Compared with primitive cubes, critical cubes specify

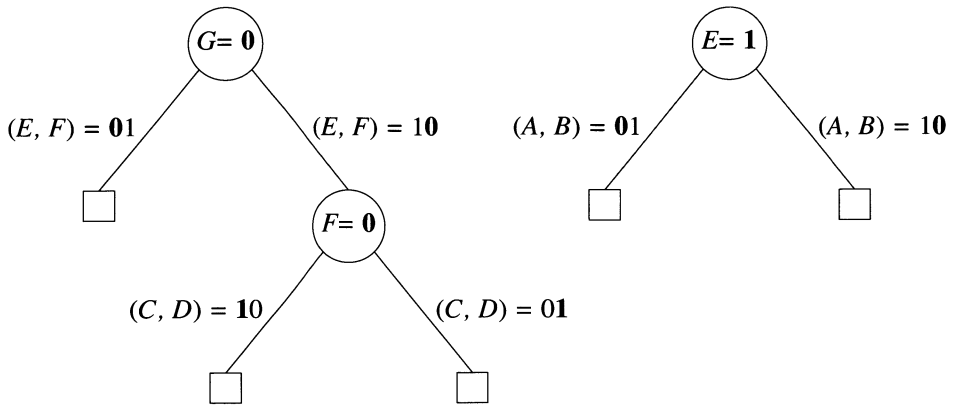



**Figure 6.47** Decision trees for Example 6.12

more values, which may lead to more conflicts. Therefore when all attempts to justify a critical line by critical cubes have failed, we should try to justify it by primitive cubes. Here, in addition to recovery from incorrect decisions, backtracking also achieves a systematic exploration of critical paths.

We have seen that reconvergent fanout may lead to self-masking, which occurs when a stem is not critical in a test $t$, although one or more of its fanout branches are critical in $t$. Therefore, when a fanout branch is made critical, this does not imply that its stem is also critical. One solution is to determine the criticality of the stem by an analysis similar to that done by the critical-path tracing method described in Chapter 5. Another solution, relying on the seldomness of self-masking, is to continue the TG process by assuming that the stem is critical. Because this strategy ignores the possibility of self-masking, it must simulate every generated test to determine the faults it detects.

Tests generated by critical-path methods may fail to detect faults that can be detected only by multiple-path sensitization. For example, for the circuit in Figure 6.48, a critical-path TG algorithm will not produce any test in which both $P$ and $Q$ have value 1. But this is the only way to detect the fault $B$ $s$-$a$-0. In practice, however, this situation seldom occurs, and even when it does, the resulting loss of fault coverage is usually insignificant.

Because of overlap among the PO cones, a critical-path TG algorithm may repeatedly encounter the same problem of justifying a critical value $v$ for a line $l$. To avoid generating unnecessary tests, we can adopt the strategy of not trying to make line $l$ have a critical value $v$ if this has been done in a previously generated test. Using this strategy in circuits with reconvergent fanout may preclude the detection of some faults, but this problem is insignificant in practical circuits.

Compared to fault-oriented TG, critical-path TG offers the following advantages:

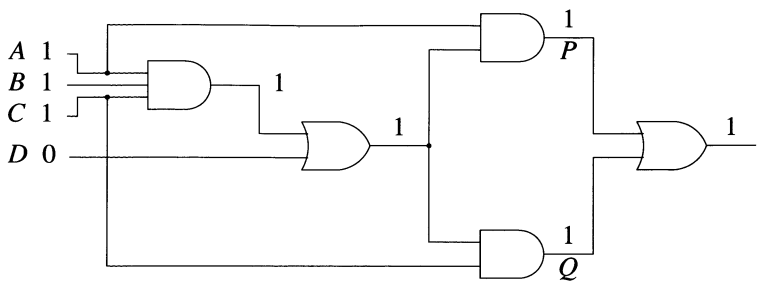



**Figure 6.48**

- The SSFs detected by the generated tests are immediately known without using a separate fault simulation step, as is required with a fault-oriented algorithm.
- A new test is generated by modifying the critical paths obtained in the previous test. This may avoid much duplicated effort inherent in a fault-oriented algorithm. For example, suppose that a critical-path algorithm has created the critical path shown in Figure 6.49, in which $A$=**0**. The next test that makes $B$=**0** is immediately obtained by switching ($A$,$B$)=**0**1 to ($A$,$B$)=1**0**. By contrast, a fault-oriented TG algorithm treats the generation of tests for the targets $A$ *s-a*-1 and $B$ *s-a*-1 as unrelated problems, and hence it would duplicate the effort needed to propagate the error from $C$ to $Z$.

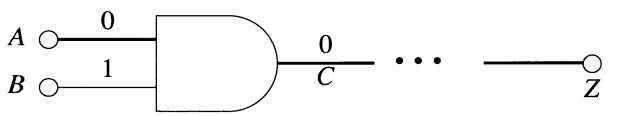



**Figure 6.49**

Avoiding the duplication of effort caused by repeated error propagation along the same paths is also one of the goals of the *subscripted D-algorithm* [Airapetian and McDonald 1979, Benmehrez and McDonald 1983], which tries to *generate simultaneously a family of tests* to detect SSFs associated with the same gate by using *flexible signals*. The output of an $n$-input target gate is assigned the flexible signal $D_0$, while its inputs are assigned the flexible signals $D_i$, $1 \le i \le n$ (see Figure 6.50(a)). The $D_0$ is propagated to a PO in the same way as a $D$ in the $D$-algorithm, and the $D_i$ signals are propagated towards PIs. A $D_i$ on the output of a gate $G$ is propagated by setting all the currently unassigned inputs of $G$ to value $D_i$ or $\overline{D}_i$, depending on the inversion of $G$. In general, conflicts among the $D_i$ values preclude the simultaneous propagation of all of them. But usually a subset of the $D_i$s can be propagated. For

example, in the circuit of Figure 6.50(b), the algorithm starts by trying to propagate $D_1$, $D_2$, and $D_3$ from the inputs of gate $W$. (Because $W$ is a PO, the $D_0$ value has already reached a PO.) But the reconvergent fanout of $Q$ and $R$ causes conflicts that are resolved by replacing $D_2$ by a 0. (More details on the propagation of $D_i$s and on the conflict resolution can be found in [Benmehrez and McDonald 1983].) Then only $D_1$ and $D_3$ are propagated. The vector $\bar{D}_1 1 1 \bar{D}_3$ represents a family of tests where $D_1$ and $D_3$ can be independently replaced by 0 or 1 values. Setting $P$, $S$ in turn to 01, 10, and 00, we obtain three (out of the four) critical cubes of gate $W$: **1**00, 00**1**, and **000**.

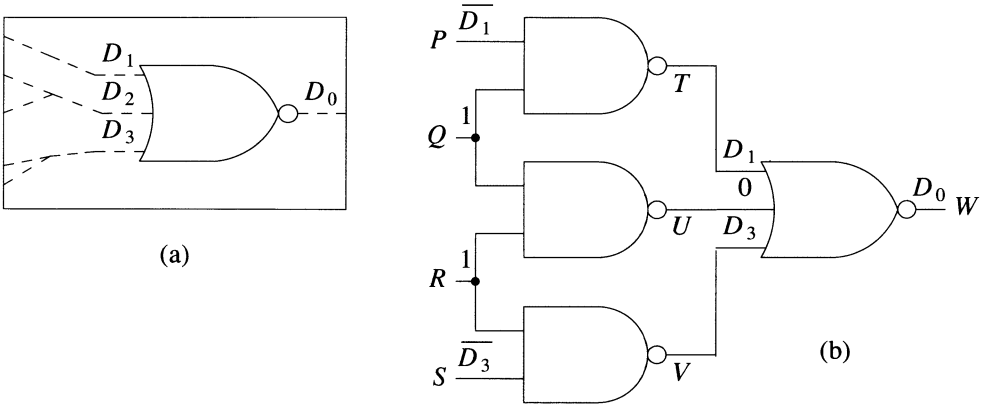



**Figure 6.50** The subscripted $D$-algorithm (a) Flexible signals (b) Generating a family of tests

Unlike a fault-oriented algorithm, a fault-independent algorithm cannot identify undetectable faults.

### 6.2.3 Random Test Generation

RTG is not a truly random process of selecting input vectors, but a *pseudorandom* one. This means that vectors are generated by a deterministic algorithm such that their statistical properties are similar to those of a randomly selected set.

The main advantage of RTG is the ease of generating vectors. Its main disadvantage is that a randomly generated test set that detects a set of faults is much larger (usually 10 times or more) than a deterministically generated test set for the same set of faults. This raises the problem of how to determine the quality of a test set obtained by RTG, because conventional methods based on fault simulation may become too costly. In this section we analyze statistical methods that estimate the quality of a test set based on probabilities of detecting faults with random vectors. A related problem is to determine the number of randomly generated vectors needed to achieve a given test quality.

Initially we assume that the *input vectors are uniformly distributed*, that is, each one of the $2^n$ possible input vectors of a circuit with $n$ PIs is equally likely to be generated.

This means that every PI has equal probability of having a 0-value or a 1-value. We also assume that the *input vectors are independently generated.* Then the same vector may appear more than once in the generated sequence. However, most pseudorandom vector generators work such that an already generated vector is not repeated. This mechanism leads to test sets that are smaller than those generated under the assumption of independent vectors [Wagner *et al.* 1987].

#### 6.2.3.1 The Quality of a Random Test

The quality of a random test set should express the *level of confidence* we have in interpreting the results of its application. Clearly, if at least one applied test fails, then the circuit under test is faulty. But if all the tests pass, how confident can we be that the circuit is indeed fault-free? (By fault-free we mean that none of the detectable SSFs is present.) This level of confidence can be measured by the probability that the applied tests detect every detectable SSF. Thus for a test sequence of length $N$, we define its *testing quality* $t_N$ as the probability that all detectable SSFs are detected by applying $N$ random vectors. In other words, $t_N$ is the probability that $N$ random vectors will contain a complete test set for SSFs.

A different way of measuring the level of confidence in the results of random testing is to consider the faults individually. Namely, if all the $N$ applied tests pass, how confident are we that the circuit does not contain a fault $f$? This can be measured by the probability $d_N^f$ that $f$ is detected (at least once) by applying $N$ random vectors; $d_N^f$ is called the *N-step detection probability of f.* The *detection quality* $d_N$ of a test sequence of length $N$ is the lowest $N$-step detection probability among the SSFs in the circuit:

$$d_N = \min_f d_N^f \tag{6.10}$$

The difference between the testing quality $t_N$ and the detection quality $d_N$ of a test sequence of length $N$ is that $t_N$ is the probability of detecting every fault, while $d_N$ is the probability of detecting the fault that is most difficult to detect [David and Blanchet 1976]. Note that $t_N < d_N$.

#### 6.2.3.2 The Length of a Random Test

Now we discuss the problem of determining the length $N$ required to achieve a level of confidence of at least $c$. Usually $c$ relates to the detection quality, so $N$ is chosen to satisfy

$$d_N \geq c \tag{6.11}$$

This is justified since a test sequence long enough to detect the most difficult fault with probability $c$ will detect any other fault $f$ with a probability $d_N^f \geq c$.

The *N-step escape probability* $e_N^f$ of a fault $f$ is the probability that $f$ remains undetected after applying $N$ random vectors. Clearly:

$$d_N^f + e_N^f = 1 \tag{6.12}$$

Let $T_f$ be the set of all tests that detect $f$ in a circuit with $n$ PIs. Then the probability $d_f$ that a random vector detects $f$ is

$$d_f = |T_f| / 2^n \tag{6.13}$$

This is $d_1^f$, the one-step detection probability of $f$; we will refer to it as the *detection probability* of $f$.

The probability of $f$ remaining undetected after one vector is $e_1^f = 1-d_f$. Because the input vectors are independent, the $N$-step escape probability of $f$ is

$$e_N^f = (1-d_f)^N \tag{6.14}$$

Let $d_{\min}$ be the lowest detection probability among the SSFs in the circuit. Then the required $N$ to achieve a detection quality of at least $c$ is given by

$$1 - (1-d_{\min})^N \geq c \tag{6.15}$$

The smallest value of $N$ that satisfies this inequality is

$$N_d = \left\lceil \frac{ln(1-c)}{ln(1-d_{\min})} \right\rceil \tag{6.16}$$

(For values of $d_{\min} << 1$, $ln(1-d_{\min})$ can be approximated by $-d_{\min}$.)

This computation of $N$ assumes that the detection probability $d_{\min}$ of the most difficult fault is known. Methods of determining detection probability of faults are discussed in the next section.

If the desired level of confidence $c$ relates to the testing quality (rather than to the detection quality), then $N$ is chosen to satisfy

$$t_N \geq c \tag{6.17}$$

Savir and Bardell [1984] have derived the following formula to estimate an upper bound on the smallest value of $N$ that achieves a testing quality of at least $c$:

$$N_t \simeq \left\lceil \frac{ln(1-c) - ln(k)}{ln(1-d_{\min})} \right\rceil \tag{6.18}$$

where $k$ is the number of faults whose detection probability is in the range $[d_{\min}, 2d_{\min}]$. Faults whose detection probability is $2d_{\min}$ or greater do not significantly affect the required test length.

**Example 6.13:** Figure 6.51 tabulates several values of $N_d$ and $N_t$ for different values of $c$ and $k$ for a circuit in which $d_{\min} = 0.01$*. For example, to detect any fault with a probability of at least 0.95 we need to apply at least $N_d = 300$ random vectors. If the circuit has only $k$=2 faults that are difficult to detect (i.e., their detection probability is in the range [0.01, 0.02]), we need to apply at least $N_t = 369$ vectors to have a probability of at least 0.95 of detecting every fault.

For $d_{\min} = 0.001$, all the $N_d$ and $N_t$ values in Figure 6.51 are increased by a factor of 10. □

* This value should be understood as an approximate value rather than an exact one (see Problem 6.20).

| $c$ | $N_d$ | $k$ | $N_t$ |
|---|---|---|---|
| 0.95 | 300 | 2 | 369 |
| | | 10 | 530 |
| 0.98 | 392 | 2 | 461 |
| | | 10 | 622 |

**Figure 6.51**

### 6.2.3.3 Determining Detection Probabilities

We have seen that to estimate the length of a random test sequence needed to achieve a specified detection quality, we need to know $d_{\min}$, the detection probability of the most difficult fault. In addition, to estimate the length needed to achieve a specified testing quality, we need to know the number $k$ of faults whose detection probability is close to $d_{\min}$. In this section we will discuss the problem of determining the detection probability of faults.

#### A Lower Bound on $d_{min}$

The following result [David and Blanchet 1976, Schedletsky 1977] provides an easy-to-derive lower bound on $d_{\min}$.

**Lemma 6.1:** In a multiple-output combinational circuit, let $n_{\max}$ be the largest number of PIs feeding a PO. Then

$$d_{\min} \geq 1/2^{n_{\max}}.$$

**Proof:** Let $f$ be a detectable SSF in a circuit with $n$ PIs. There exists at least one test that detects $f$ such that some PO, say $Z_k$, is in error. Let $n_k$ be the number of PIs feeding $Z_k$. Because the values of the other $n-n_k$ PIs are irrelevant for the detection of $f$ at $Z_k$, there exist at least $2^{n-n_k}$ vectors that detect $f$. Then the detection probability of $f$ is at least $2^{n-n_k}/2^n = 1/2^{n_k}$. The lower bound is obtained for the PO cone with the most PIs. □

This lower bound, however, is usually too conservative, since by using it for the value of $d_{\min}$ in formula (6.16), we may often obtain $N_d > 2^n$, which means that we would need more vectors for random testing than for exhaustive testing [Schedletsky 1977].

#### Detection Probabilities Based on Signal Probabilities

The *signal probability* of a line $l$, $p_l$, is defined as the probability that $l$ is set to 1 by a random vector:

$$p_l = Pr(l=1) \tag{6.19}$$

An obvious relation exists between the signal probability of a PO and the detection probabilities of its stuck faults. Namely, the detection probability of a *s-a-c* fault on a PO $Z$ is $p_Z$ for $c$=0 and $1-p_Z$ for $c$=1. We will illustrate the general relation between signal probabilities and detection probabilities by reference to Figure 6.52 [Savir *et al.*

1984]. Suppose that in a single-output circuit there exists only one path between line $l$ and the PO. Let $f$ be the fault $l$ $s$-$a$-0. A vector that detects $f$ must set $l$=1 and also set all the values needed to sensitize the path ($A$=1, $B$=0, etc). To account for this set of conditions we introduce an auxiliary AND gate $G$, such that $G$=1 iff all the conditions required for detecting $f$ are true. If $x$=$v$ is a required condition, then $x$ is directly connected to $G$ if $v$=1, or via an inverter if $v$=0. Thus the detection probability of $f$ is given by the signal probability of $G$ in the modified circuit:

$$d_f = p_G \tag{6.20}$$

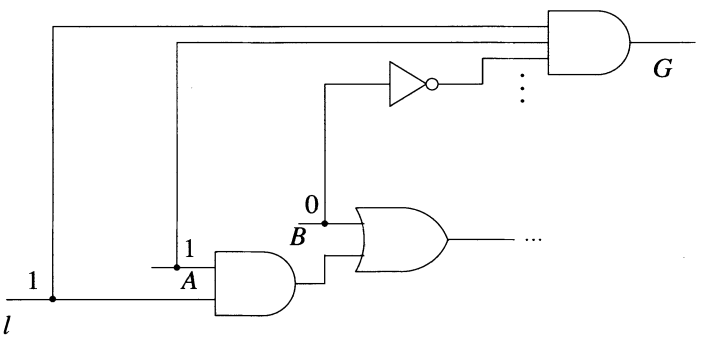



**Figure 6.52**

In general, there may be several paths along which an error caused by a fault $f$ can propagate to a PO. Then the probability of detecting $f$ propagating an error along only one of these paths is clearly a lower bound on the probability of detecting $f$. Denoting by $G_k$ the auxiliary gate associated with the $k$-th propagation path, we have

$$d_f \geq p_{G_k} \tag{6.21}$$

Thus signal probabilities can be used to compute detection probabilities or lower bounds on the detection probabilities. Now we turn to the problem of determining signal probabilities. Since signal probabilities of PIs are known (until now we have assumed that for every PI $i$, $p_i = 1/2$), let us consider the problem of computing $p_Z$ for a gate output $Z$, knowing $p_X$ of every input $X$ of $Z$ [Parker and McCluskey 1975].

**Lemma 6.2**: For an inverter with output $Z$ and input $X$

$$p_Z = 1 - p_X \tag{6.22}$$

**Proof**

$$p_Z = Pr(Z=1) = Pr(X=0) = 1 - p_X \qquad \square$$

**Lemma 6.3**: For an AND gate with output $Z$ and inputs $X$ and $Y$, if $X$ and $Y$ do not depend on common PIs then:

$$p_Z = p_X p_Y \tag{6.23}$$

**Proof**

$$p_Z = Pr(Z=1) = Pr(X=1 \cap Y=1)$$

Because $X$ and $Y$ do not depend on common PIs, the events $X$=1 and $Y$=1 are independent. Thus

$$p_Z = Pr(X=1)Pr(Y=1) = p_X p_Y \quad \square$$

**Lemma 6.4**: For an OR gate with output $Z$ and inputs $X$ and $Y$, if $X$ and $Y$ do not depend on common PIs, then

$$p_Z = p_X + p_Y - p_X p_Y \tag{6.24}$$

**Proof**

$$p_Z = 1 - Pr(Z=0) = 1 - Pr(X=0 \cap Y=0)$$

Since the events $X$=0 and $Y$=0 are independent

$$p_Z = 1 - Pr(X=0)Pr(Y=0) = 1-(1-p_X)(1-p_Y) = p_X + p_Y - p_X p_Y \quad \square$$

Formulas (6.23) and (6.24) can be generalized for gates with more than two inputs. A NAND (NOR) gate can be treated as an AND (OR) gate followed by an inverter. Then we can compute signal probabilities in any circuit in which inputs of the same gate do not depend on common PIs; these are fanout-free circuits and circuits without reconvergent fanout. The time needed to compute all signal probabilities in such a circuit grows linearly with the number of gates.

Now let us consider a circuit that has only one stem, say $A$, with reconvergent fanout. Let $Z$ be a signal that depends on $A$. Using conditional probabilities, we can express $p_Z$ as

$$Pr(Z=1) = Pr(Z=1 \mid A=0)Pr(A=0) + Pr(Z=1 \mid A=1)Pr(A=1)$$

or

$$p_Z = Pr(Z=1 \mid A=0)(1-p_A) + Pr(Z=1 \mid A=1)p_A \tag{6.25}$$

We can interpret this formula as follows. Let us create two circuits, $N^0$ and $N^1$, obtained from the original circuit $N$ by permanently setting $A$=0 and $A$=1 respectively. Then $Pr(Z=1 \mid A=0)$ is simply $p_Z$ computed in $N^0$, and $Pr(Z=1 \mid A=1)$ is $p_Z$ computed in $N^1$. The result of this conceptual splitting is that both $N^0$ and $N^1$ are free of reconvergent fanout so we can apply the formulas (6.23) and (6.24).

**Example 6.14:** Let us compute signal probabilities for the circuit in Figure 6.53(a). The only gates whose inputs depend on common PIs are $Z_1$ and $Z_2$; the computation for all the other gates is straightforward and their values are shown in Figure 6.53(a). Figures 6.53(b) and (c) show the values obtained for $A$=0 and $A$=1 respectively (the computation is done only for the lines affected by $A$). Finally we compute $p_{Z_1}$ and $p_{Z_2}$ using formula (6.25)

$$p_{Z_1} = \frac{1}{2}\cdot\frac{1}{4} + \frac{5}{8}\cdot\frac{3}{4} = \frac{19}{32}$$

$$p_{Z_2} = \frac{1}{2}\cdot\frac{1}{4} + \frac{11}{16}\cdot\frac{3}{4} = \frac{41}{64}$$

□

Note that for every line affected by the stem $A$ we compute two signal probabilities, one for $A$=0 and one for $A$=1. If we generalize this approach for a circuit that has $k$ stems with reconvergent fanout, for every line we may have to compute and store up to $2^k$ signal probabilities. This exponential growth renders this type of approach impractical for large circuits. Different methods for computing exact signal probabilities [Parker and McCluskey 1975, Seth *et al.* 1985] suffer from the same problem. An approximate method is described in [Krishnamurthy and Tollis 1989].

The *cutting algorithm* [Savir *et al.* 1984] reduces the complexity of computation by computing a range $[p_l^L, p_l^U]$ rather than an exact value for the signal probability $p_l$, such that $p_l \in [p_l^L, p_l^U]$. Its basic mechanism, illustrated in Figure 6.54, consists of cutting $k-1$ fanout branches of a stem having $k$ fanout branches; the cut fanout branches become PIs with unknown signal probabilities and are assigned the range [0,1]. Only reconvergent fanout branches should be cut. The resulting circuit is free of reconvergent fanout, so signal probabilities are easy to compute. Whenever ranges are involved, the computations are done separately for the two bounds.

**Example 6.15:** Figure 6.55 shows the signal probability ranges computed by the cutting algorithm for the circuit in Figure 6.53(a). Note that the fanout branch that remains connected to the stem inherits its signal probability. We can check that the exact values of $p_{Z_1}$ and $p_{Z_2}$ determined in Example 6.14 do fall within the ranges computed for $p_{Z_1}$ and $p_{Z_2}$ by the cutting algorithm. □

Various extensions of the cutting algorithm that compute tighter bounds on signal probabilities are described in [Savir *et al.* 1984].

### Finding the Difficult Faults

Usually the maximal length of a test sequence $N_{\max}$ is limited by the testing environment factors such as the capabilities of the tester or the maximum time allocated for a testing experiment. Given $N_{\max}$ and a desired detection quality $c$, from (6.15) we can derive the required lower bound $d_L$ on the detection probability of every fault in the circuit. In other words, if $d_f \geq d_L$ for every fault $f$, then testing the circuit with $N_{\max}$ vectors will achieve a detection quality of at least $c$. Then, given a circuit, we want to determine whether it contains "difficult" faults whose detection probability is lower than $d_L$.

The following result shows that the difficult faults, if any, are among the checkpoint faults of the circuit.

**Lemma 6.5:** The fault with the smallest detection probability in a circuit is one of its checkpoint faults.

**Proof:** For any fault $g$ on a line that is not a checkpoint (fanout branch or PI), we can find at least one checkpoint fault $f$ such that $g$ dominates $f$. Thus the number of tests that detect $f$ is smaller than or equal to the number of tests that detect $g$. Hence $p_f \leq p_g$.
□

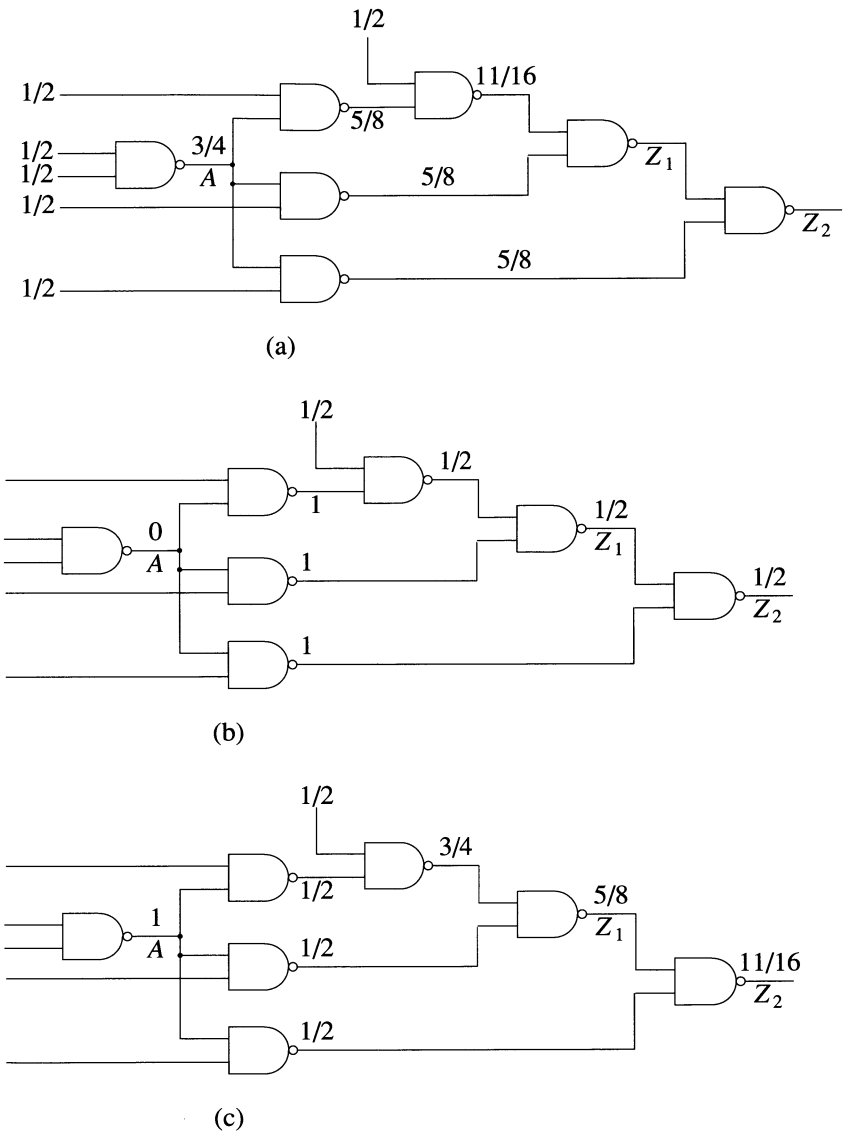



**Figure 6.53**

Figure 6.56 presents a procedure for determining the difficult faults [Savir *et al.* 1984]. Note that only single propagation paths are analyzed. A fault that can be detected only by multiple-path sensitization will be marked as difficult (see Problem 6.27). Redundant faults will also be among the difficult ones (see Problem 6.28).

Once the difficult faults of a circuit are identified, the circuit can be modified so that their detection probabilities become acceptable [Eichelberger and Lindbloom 1983].

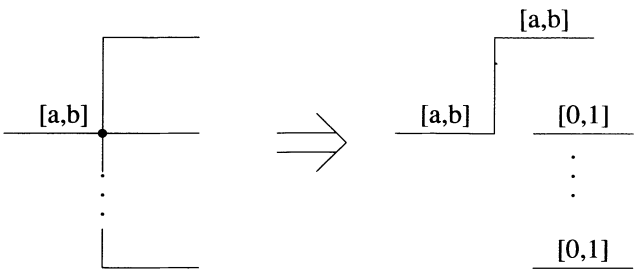



**Figure 6.54** Basic transformation in the cutting algorithm

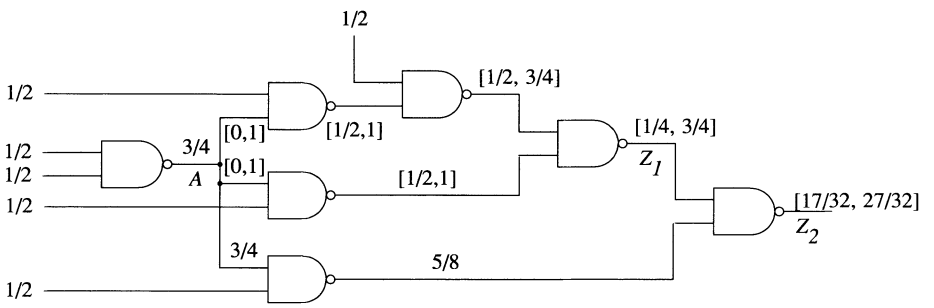



**Figure 6.55** Signal probabilities computed by the cutting algorithm

### 6.2.3.4 RTG with Nonuniform Distributions

Until now we have assumed that input vectors are uniformly distributed; hence the signal probability of every PI $i$ is $p_i$=0.5. This uniform distribution, however, is not necessarily optimal; that is, different $p_i$ values may lead to shorter test sequences.

For example, Agrawal and Agrawal [1976] showed that for fanout-free circuits composed of NAND gates with the same number ($n$) of inputs, the optimal value $p_{opt}$ of $p_i$ is given by

$$p_i = 1-p_i^n \tag{6.26}$$

For $n$=2, $p_{opt}$=0.617. For such a circuit with 10 levels, one would need about 8000 random vectors to achieve a detection quality of 0.99 using $p_i$=0.5, while using $p_{opt}$ the same detection quality can be achieved with about 700 vectors.

Similar experimental results, showing that nonuniform distributions lead to shorter random test sequences, have also been obtained for general circuits. An even better approach is to allow different PIs to have different $p_i$ values. Such random vectors are

```
for every checkpoint fault f
  begin
    repeat
      begin
        select an untried propagation path P for f
        introduce an auxiliary AND gate G such that
            p_G is the detection probability of f along P
        compute p_G^L, the lower bound of p_G
      end
    until p_G^L ≥ d_L or all propagation paths have been analyzed
    if p_G^L < d_L then mark f as difficult
  end
```

**Figure 6.56** Procedure for finding the difficult faults

said to be *weighted* or *biased*. Methods for computing the $p_i$ values are presented in [Wunderlich 1987, Waicukauski *et al.* 1989].

*Adaptive RTG* methods dynamically modify the $p_i$ values trying to reach "optimal" values. This requires an adaptation mechanism based on monitoring the TG process to determine "successful" random vectors. The method used in [Parker 1976] grades the random vectors by the number of faults they detect. Then the frequency of 1-values on PIs is measured in the set of tests that detected the most faults, and the $p_i$ values are set according to these frequencies. A different method, based on monitoring the rate of variation of the activity count generated by changing PI values, is presented in [Schnurmann *et al.* 1975]. Lisanke *et al.* [1987] describe an adaptive method in which the $p_i$ values are changed with the goal of minimizing a testability cost function.

## 6.2.4 Combined Deterministic/Random TG

Because RTG is oblivious to the internal model of the circuit, faults that have small detection probabilities require long sequences for their detection. But some of these faults may be easy to detect using a deterministic algorithm. As a simple example, consider an AND gate with 10 inputs. The detection probability of any input fault is 1/1024. However, generating tests for these faults is a simple task for any deterministic TG algorithm. In this section we analyze TG methods that combine features of deterministic critical-path TG algorithms with those of RTG.

### RAPS

Random Path Sensitization (RAPS) [Goel 1978] attempts to create random critical paths between PIs and POs (see Figure 6.57). Initially all values are $x$. RAPS starts by randomly selecting a PO $Z$ and a binary value $v$. Then the objective $(Z,v)$ is mapped into a PI assignment $(i,v_i)$ by a *random backtrace* procedure *Rbacktrace*. *Rbacktrace* is similar to the *Backtrace* procedure used by PODEM (Figure 6.27), except that the selection of a gate input is random. (In PODEM this selection is guided by controllability costs). The assignment $i=v_i$ is then simulated (using

3-valued simulation), and the process is repeated until the value of $Z$ becomes binary. After all the POs have binary values, a second phase of RAPS assigns those PIs with $x$ values (if any), such that the likelihood of creating critical paths increases (see Problem 6.29).

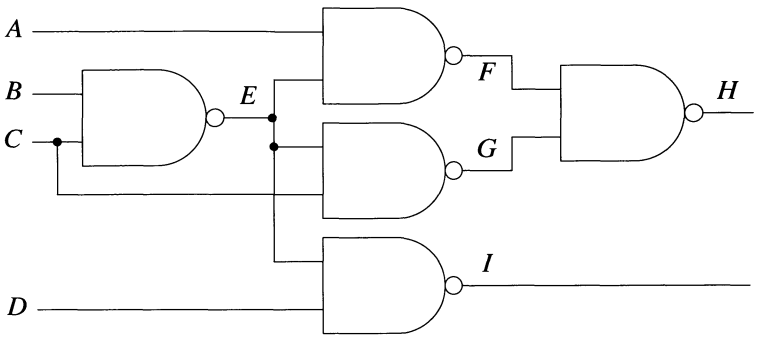
```
repeat
   begin
      randomly select a PO (Z) with x value
      randomly select a value v
      repeat
         begin
            (i,v_i) = Rbacktrace (Z,v)
            Simulate (i,v_i)
         end
      until Z has binary value
   end
until all POs have binary values
while some PIs have x values
   begin
      G = highest level gate with x input values
      c = controlling value of G
      randomly select an input (j) of G with x value
      repeat
         begin
            (i,v_i) = Rbacktrace (j,c̄)
            Simulate (i,v_i)
         end
      until j has binary value
   end
```

**Figure 6.57** RAPS

The entire procedure is repeated to generate as many tests as needed. A different set of critical paths is likely to be generated in every test. Figure 6.58(b) traces a possible execution of RAPS for the circuit in Figure 6.58(a). In general, test sets generated by RAPS are smaller and achieve a higher fault coverage than those obtained by RTG.

Figure 6.59 illustrates one problem that may reduce the efficiency of RAPS. First $A = 0$ has been selected to justify $PO1 = 0$, then $B = 0$ to justify $PO2 = 0$. But the latter selection precludes propagation of any fault effects to $PO1$.

Another problem is shown in Figure 6.60. Because of the random selection process of RAPS, half of the tests in which $Z = 1$ will have $B = 0$. But only one test with $B = 0$ (and $C = 1$) is useful, because none of the subsequent tests with $B = 0$ will detect any new faults on the PO $Z$.




(a)

| | Objective | Path backtraced | PI assignment | Simulation |
|---|---|---|---|---|
| $t_1$ | $H = 0$ | $H, F, A$<br>$H, G, C$ | $A = 0$<br>$C = 0$ | $F = 1$<br>$E = 1, G = 1, H = 0$ |
| | $I = 1$ | $I, D$ | $D = 0$ | $I = 1$ |
| | $B = 1$ | | $B = 1$ | |
| $t_2$ | $H = 0$ | $H, G, E, B$<br>$H, F, E, C$ | $B = 1$<br>$C = 1$ | $E = 0, F = 1, G = 1, H = 0, I = 1$ |
| | $D = 1$ | | $D = 1$ | |
| | $A = 1$ | | $A = 1$ | |
| $t_3$ | $I = 0$ | $I, E, C$<br>$I, D$ | $C = 0$<br>$D = 1$ | $E = 1, G = 1$<br>$I = 1$ |
| | $H = 1$ | $H, F, A$ | $A = 1$ | $F = 0, H = 1$ |
| | $B = 1$ | | $B = 1$ | |

(b)

**Figure 6.58** Example of random path sensitization

A similar problem occurs when all faults that can be detected at a PO $Z$ when $Z = v$ have been detected by the already generated tests. In half of the subsequent tests, RAPS will continue trying to set $Z = v$, which, in addition to being useless, may also prevent detection of faults at other POs that depend on PIs assigned to satisfy $Z = v$.

Figure 6.61 illustrates a problem in the second stage of RAPS. Because the value of $d$ is justified by $a = 0$, while $b = c = x$, RAPS will try to justify 1's for $b$ and $c$. But

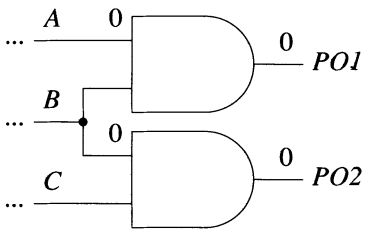



**Figure 6.59**

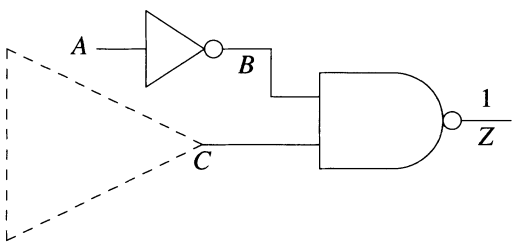



**Figure 6.60**

setting $b = c = 1$ is useless, because no additional faults can be detected through $d$; in addition, $c = 1$ requires $e = 0$, which will preclude detecting any other faults through $h$.

### SMART

SMART (*S*ensitizing *M*ethod for *A*lgorithmic *R*andom *T*esting) is another combined deterministic/random TG method [Abramovici *et al.* 1986a], which corrects the problems encountered by RAPS. SMART generates a vector incrementally and relies on a close interaction with fault simulation. The partial vector generated at every step is simulated using critical-path tracing (CRIPT), whose results guide the test generation process. This interaction is based on two byproducts of CRIPT: *stop lines* and *restart gates*. Stop lines delimit areas of the circuit where additional fault coverage cannot be obtained, while restart gates point to areas where new faults are likely to be detected with little effort.

A line $l$ is a 0(1)-*stop line* for a test set $T$, if $T$ detects all the faults that can cause $l$ to take value 1(0). For example, consider the circuit and the test shown in Figure 6.62. This test detects all the faults that can set $E = 1$, $E2 = 1$ and $G = 0$; hence $E$ and $E2$ are 0-stops and $G$ is a 1-stop. A line $l$ that is not a 0(1)-stop is said to be 0(1)-*useful*, because there are some new faults that could be detected when $l$ has value 0(1). In

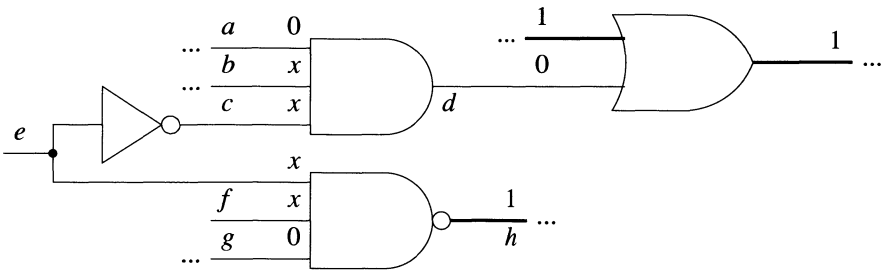



**Figure 6.61**

Figure 6.62, *E1* is 0-useful, because the still undetected fault *E1* *s-a*-1 requires *E1* = 0 for detection.

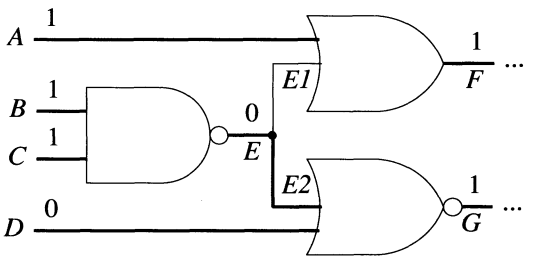



**Figure 6.62** Example of stop lines

A line $l$ becomes a $v$-stop for a test set $T$ if $l$ has had a critical value $v$ and

1. If $l$ is a fanout branch, its stem is a $v$-stop.
2. If $l$ is the output of a gate with inversion $i$, all the gate inputs are $(v \oplus i)$-stops.

A PI becomes a $v$-stop when it has had a critical value $v$. While TG progresses and the set $T$ grows, more and more lines become stops and they advance from PIs toward POs. CRIPT uses stop lines to avoid tracing paths in any area bounded by a $v$-stop line $l$ in any test in which $l$ has value $v$.

A *restart gate* (in a test $t$) must satisfy the following conditions:

1. Its output is critical (in $t$) but none of its inputs is critical.
2. Exactly one input has the controlling value $c$, and this input is $c$-useful.

For the example in Figure 6.63, if we assume that $C$ is 0-useful, then $G$ is a restart gate. To make the same test detect additional faults we should set $E = D = 1$; then $C$

becomes critical and the faults that make $C = 1$ are likely to be detected along the existing critical path from $G$. No special effort is required to identify restart gates, since they are a subset of the gates where CRIPT terminates its backtracing. (The CRIPT algorithm described in Chapter 5 handles only binary values; for this application it needs to be extended to process partially specified vectors).

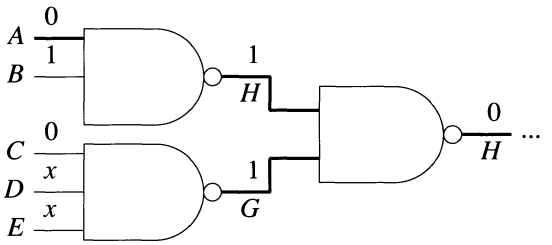



**Figure 6.63** Example of restart gate

SMART (see Figure 6.64) starts by randomly selecting a PO objective $(Z,v)$ such that $Z$ is $v$-useful. Then the objective $(Z,v)$ is mapped into a PI assignment $(i, v_i)$ by the procedure *Useful_rbacktrace*. When a value $v_j$ is needed for a gate input, *Useful_rbacktrace* gives preference to those inputs that are $v_j$-useful and have value $x$. The partial vector generated so far is fault simulated by CRIPT, which backtraces critical paths from the POs that have been set to binary values. CRIPT also determines stop lines and restart gates. Then SMART repeatedly picks a restart gate, tries to justify noncontrolling values on its inputs with value $x$, and reruns CRIPT. Now CRIPT restarts its backtracing from the new POs that have binary values (if any) and from the restart gates.

Unlike RAPS, SMART tries to create critical paths leading to the chosen PO $Z$ before processing other POs. Thus in the example shown in Figure 6.59, after justifying $PO1 = 0$ by $A = 0$, the gate $PO1$ is a restart gate; hence SMART will try to justify $B = 1$ before processing $PO2$. Then $PO2 = 0$ will be justified by $C = 0$.

Using stop lines avoids the problem illustrated in Figure 6.60. After the first test that sets $B = 0$ and $C = 1$, $B$ becomes a 0-stop and will no longer be used to justify $Z = 1$.

In the example in Figure 6.61, keeping track of restart gates allows SMART to select $h$ (and ignore $d$, which is not a restart gate), and continue by trying to set $a = f = 1$.

Experimental results [Abramovici *et al.* 1986a] show that, compared with RAPS, SMART achieves higher fault coverage with smaller test sets and requires less CPU time.

## 6.2.5 ATG Systems

### Requirements

In general, a TG algorithm is only a component of an *ATG system*, whose overall goal is to generate a test set for the SSFs in a circuit. Ideally, we would like to have

```
while (useful POs have x values)
  begin
    randomly select a useful PO (Z) with x value
    if (Z is both 0- and 1-useful)
      then randomly select a value v
    else v = the useful value of Z
    repeat
      begin
        (i, v_i) = Useful_rbacktrace (Z, v)
        Simulate (i, v_i)
      end
    until Z has binary value
    CRIPT()
    while (restart_gates ≠ ∅)
      begin
        remove a gate (G) from restart_gates
        c = controlling value of G
        for every input (j) of G with value x
          repeat
            begin
              (i, v_i) = Rbacktrace (j, c̄)
              Simulate (i, v_i)
            end
          until j has binary value
        CRIPT()
      end
  end
```

**Figure 6.64** SMART

- the fault coverage of the generated tests as high as possible,
- the cost of generating tests (i.e., the CPU time) as low as possible,
- the number of generated tests as small as possible.

These requirements follow from economic considerations. The fault coverage directly affects the quality of the shipped products. The number of generated tests influences the cost of applying the tests, because a larger test set results in a long test-application time. Unlike the cost of generating tests, which is a one-time expense, the test time is an expense incurred every time the circuit is tested. Of course, the above ideal requirements are conflicting, and in practice the desired minimal fault coverage, the maximal TG cost allowed, and the maximal number of vectors allowed are controlled by user-specified limits.

The desired fault coverage should be specified with respect to the detectable faults, because otherwise it may be impossible to achieve it. For example, in a circuit where

4 percent of the faults are redundant, a goal of 97 percent absolute fault coverage would be unreachable. The fault coverage for detectable faults is computed by

$$\frac{\textit{number of detected faults}}{\textit{total number of faults} - \textit{number of undetectable faults}}$$

that is, the faults proven redundant are excluded from the fault universe. Thus it is important that the ATG algorithm can identify redundant faults.

Another question to consider is whether the ATG system should collapse faults and, if so, what fault-collapsing technique it should use. Fault collapsing is needed for systems relying on a fault simulator whose run-time grows with the number of simulated faults. Fault collapsing can be based on equivalence and/or dominance relations. The latter can be used only if the ATG system does not produce fault-location data. Heap and Rogers [1989] have shown that computing the fault coverage based on a collapsed set of faults is less accurate than computing based on the original (uncollapsed) set; they presented a method to compute the correct fault coverage from the results obtained from simulating the collapsed set. In Chapter 4 we have shown that detecting all checkpoint faults results in detecting all SSFs in the circuit. Some ATG systems start only with the checkpoint faults (which can be further collapsed using equivalence and dominance relations). Abramovici *et al.* [1986b] have shown that this set may not be sufficient in redundant circuits, where detecting all the detectable checkpoint faults does not guarantee the detection of all the detectable faults. They presented a procedure that determines additional faults to be consideredso that none of the potentially detectable faults is overlooked. Note that fault collapsing is not needed in a system using critical-path tracing, which deals with faults only implicitly and does not require fault enumeration.

**Structure**

Most ATG systems have a 2-phase structure (Figure 6.65), which combines two different methods. The first phase uses a low-cost, fault-independent procedure, providing an initial test set which detects a large percentage of faults (typically, between 50 and 80 percent). In the second phase, a fault-oriented algorithm tries to generate tests for the faults left undetected after the first phase [Breuer 1971, Agrawal and Agrawal 1972].

Figure 6.66 outlines the structure of the first phase of an ATG system. *Generate_test* can be a pseudorandom generation routine or a combined deterministic/random procedure (RAPS or SMART). The generated test $t$ is fault simulated and the function *value* determines a figure of merit usually based on the number of new (i.e., previously undetected) faults detected by $t$. Depending on its value, which is analyzed by the function *acceptable*, $t$ is added to the test set or rejected. A test that does not detect any new fault is always rejected. A characteristic feature of a sequence of vectors generated in the first phase is that the number of new faults detected per test is initially large but decreases rapidly. Hence the number of rejected vectors will increase accordingly. The role of the function *endphase1* is to decide when the first phase becomes inefficient and the switch to the use of a deterministic fault-oriented TG algorithm should occur.

Although RTG produces a vector faster than a combined deterministic/random method, a vector generated by RAPS or SMART detects more new faults than a pseudorandom

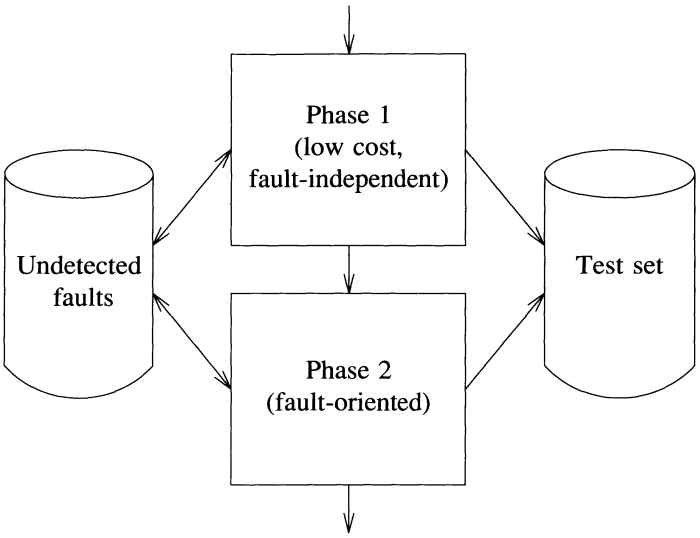



**Figure 6.65** Two-phase ATG system

vector. To reach the same fault coverage as RAPS or SMART, RTG needs more vectors, many of which will be rejected. If every generated vector is individually fault simulated, the total computational effort (test generation and test evaluation) is greater when using RTG. Groups of vectors generated by RTG or RAPS can be evaluated in parallel using a fault simulator based on parallel-pattern evaluation, thus significantly reducing the CPU time spent in fault simulation. SMART cannot use parallel-pattern evaluation, because the results obtained in simulating one vector are used in guiding the generation of the next one. However, SMART achieves higher fault coverage with fewer vectors than RTG or RAPS.

Many techniques can be used to implement the functions *value, acceptable* and *endphase1*. The simplest techniques define the value of a vector as the number of new faults it detects, accept any test that detects more than $k$ new faults, and stop the first phase when more than $b$ consecutive vectors have been rejected ($k$ and $b$ are user-specified limits). More sophisticated techniques are described in [Goel and Rosales 1981, Abramovici *et al.* 1986a].

Figure 6.67 outlines the structure of the second phase of an ATG system. The initial set of target faults is the set of faults undetected at the end of the first phase. Since some faults may cause the TG algorithm to do much search, the computational effort allocated to a fault is controlled by a user-specified limit; this can be a direct limit on the CPU time allowed to generate a test or an indirect one on the amount of backtracking performed.

ATG algorithms spend the most time when dealing with undetectable faults, because to prove that a fault is undetectable the algorithm must fail to generate a test by

```
repeat
   begin
      Generate_test(t)
      fault simulate t
      v = value(t)
      if acceptable(v) then add t to the test set
   end
until endphase1()
```

**Figure 6.66** First phase of an ATG system

```
repeat
   begin
      select a new target fault f
      try to generate a test (t) for f
      if successful then
         begin
            add t to the test set
            fault simulate t
            discard the faults detected by t
         end
   end
until endphase2()
```

**Figure 6.67** Second phase of an ATG system

exhaustive search. Thus identifying redundant faults without exhaustive search can significantly speed up TG. Two types of redundancy identification techniques have been developed. *Static techniques* [Menon and Harihara 1989] work as a preprocessing step to TG, while *dynamic techniques* [Abramovici *et al.* 1989] work during TG.

A good way of selecting a new target fault $f$ (i.e., a still undetected fault that has not been a target before) is to choose one that is located at the lowest possible level in the circuit. In this way, if the test generation for $f$ is successful, the algorithm creates a long path sensitized to $f$, so the chances of detecting more still undetected faults along the same path are increased. Detecting more new faults per test results in shorter test sequences.

A different method is to select the new target fault from a precomputed set of *critical faults* [Krishnamurthy and Sheng 1985]. The critical faults are a maximal set of faults

with the property that there is no test that can detect two faults from this set simultaneously.

Of course, the second phase stops when the set of target faults is exhausted. In addition, both the second and the first phase may also be stopped when the generated sequence becomes too long, or when a desired fault coverage is achieved, or when too much CPU time has been spent; these criteria are controlled by user-imposed limits.

**Test Set Compaction**

Test vectors produced by a fault-oriented TG algorithm are, in general, partially specified. Two tests are *compatible* if they do not specify opposite values for any PI. Two compatible tests $t_i$ and $t_j$ can be combined into one test $t_{ij}=t_i \cap t_j$ using the intersection operator defined in Figure 2.3. (The output vectors corresponding to $t_i$ and $t_j$ are compatible as well — see Problem 6.30.) Clearly, the set of faults detected by $t_{ij}$ is the union of the sets of faults detected by $t_i$ and $t_j$. Thus we can replace $t_i$ and $t_j$ by $t_{ij}$ without any loss of fault coverage. This technique can be iterated to reduce the size of a test set generated in the second phase of an ATG system; this process is referred to as *static compaction.*

In general, the test set obtained by static compaction depends on the order in which vectors are processed. For example, let us consider the following test set:

$$\begin{aligned} t_1 &= 01x \\ t_2 &= 0x1 \\ t_3 &= 0x0 \\ t_4 &= x01 \end{aligned}$$

If we first combine $t_1$ and $t_2$, we obtain the set

$$\begin{aligned} t_{12} &= 011 \\ t_3 &= 0x0 \\ t_4 &= x01 \end{aligned}$$

which cannot be further compacted. But if we start by combining $t_1$ and $t_3$, then $t_2$ and $t_4$ can also be combined, producing the smaller test set

$$\begin{aligned} t_{13} &= 010 \\ t_{24} &= 001. \end{aligned}$$

However, the use of an optimal static compaction algorithm (that always yields a minimal test set) is too expensive. The static compaction algorithms used in practice are based on heuristic techniques.

Static compaction is a postprocessing operation done after all vectors have been generated. By contrast, in *dynamic compaction* every partially specified vector is processed immediately after its generation, by trying to assign PIs with $x$ values so that it will detect additional new faults [Goel and Rosales 1979]. Figure 6.68 outlines such a dynamic compaction technique that is activated after a test $t$ has been successfully generated for the selected target fault (see Figure 6.67). The function *promising* decides whether $t$ is a good candidate for dynamic compaction; for example, if the percentage of PIs with $x$ values is less than a user-specified threshold, $t$ is not processed for compaction. While $t$ is "promising," we select a secondary target fault $g$ and try to extend $t$ (by changing some of the PIs with $x$ values) such that it will detect

$g$. This is done by attempting to generate a test for $g$ without changing the already assigned PI values.

```
while promising(t)
   begin
      select a secondary target fault g
      try to extend t to detect g
   end
```

**Figure 6.68** Dynamic compaction

Experimental results have shown that dynamic compaction produces smaller test sets than static compaction with less computational effort. Note that dynamic compaction cannot be used with a fault-independent TG algorithm.

The most important problem in dynamic compaction is the selection of secondary target faults. Goel and Rosales [1980] compare an arbitrary selection with a technique based on analyzing the values determined by the partially specified test $t$ with the goal of selecting a fault for which TG is more likely to succeed. Their results show that the effort required by this additional analysis pays off both in terms of test length and the overall CPU time.

A better dynamic compaction method is presented in [Abramovici *et al.* 1986a]. It relies on a close interaction with fault simulation by critical-path tracing, whose results are used to select secondary target faults already activated by the partially generated vector and/or whose effects can be easily propagated to a PO. When simulating a partially specified vector, critical-path tracing also identifies restart gates, which point to areas where additional faults are likely to be easily detected. In the example in Figure 6.61, if $g$ is 0-useful, $h$ is a restart gate. A good secondary target fault is likely to be found in the area bounded by $g$.

## 6.2.6 Other TG Methods

### Algebraic Methods

The TG methods discussed in this chapter can be characterized as *topological*, as they are based on a structural model of a circuit. A different type of methods, referred to as *algebraic*, use Boolean equations to represent the circuit. (A review of algebraic methods appears in [Breuer and Friedman 1976]). Algebraic methods are impractical for large circuits.

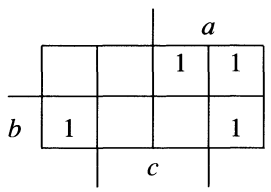
### Extensions for Tristate Logic

Extensions of the conventional test generation algorithms to handle circuits with tristate busses are presented in [Breuer 1983, Itazaki and Kinoshita 1986]. An additional problem for test generation in this type of circuits is to avoid conflicts caused by the simultaneous enabling of two or more bus drivers.

### TG for Module-Level Circuits

Until now we have considered only gate-level circuits. The previously discussed TG algorithms can be extended to process more complex components, referred to as *modules* (for example, exclusive-ORs, multiplexers, etc.). The internal gate-level structure of a module is ignored. While for gates we had simple and uniform procedures for line justification and error propagation, for every type of module we need to precompute and store all the possible solutions to the problems of justifying an output value and of propagating errors from inputs to outputs. For some modules, a set of solutions may also be incrementally generated by special procedures, that, on demand, provide the next untried solution from the set.

**Example 6.16:** Consider a module that implements the function $Z=a\bar{b}+b\bar{c}$, whose Karnaugh map is shown in Figure 6.69(a). The solutions to the problems of justifying $Z=1$ and $Z=0$ are given by the primitive cubes of $Z$, shown in Figure 6.69(b). The primitive cubes with $Z=1(0)$ correspond to the prime implicants of $Z(\bar{Z})$.




(a)

| a | b | c | Z |
|---|---|---|---|
| 1 | x | 0 | 1 |
| 1 | 0 | x | 1 |
| x | 1 | 0 | 1 |
| 0 | x | 1 | 0 |
| 0 | 0 | x | 0 |
| x | 1 | 1 | 0 |

(b)

| a | b | c | Z |
|---|---|---|---|
| D | 0 | x | D |
| 0 | D | 0 | D |
| 1 | D | 1 | $\bar{D}$ |
| x | 1 | D | $\bar{D}$ |

(c)

**Figure 6.69** (a) Karnaugh map (b) Primitive cubes (c) Propagation $D$-cubes

Now let us assume that $a=D$ and we want to determine the minimal input conditions that will propagate this error to $Z$. In other words, we look for values of $b$ and $c$ such that $Z$ changes value when $a$ changes from 1 to 0. This means finding pairs of cubes of $Z$ of the form $1v_bv_c \mid v_z$ and $0v_b'v_c' \mid \bar{v}_z$, where the values of $b$ and $c$ are compatible. From such a pair we derive a cube of the form $D(v_b \cap v_b')(v_c \cap v_c') \mid D'$, where $D'=D(\bar{D})$ if $v_z=1(0)$. For example, from the primitive cubes $10x \mid 1$ and $00x \mid 0$ we construct the cube $D0x \mid D$; this tells us that setting $b=0$ propagates a $D$ from $a$ to $Z$. Such a cube is called a *propagation D-cube* [Roth 1980]. Figure 6.69(c) lists the propagation $D$-cubes dealing with the propagation of one error. (The propagation $D$-cube $D00 \mid D$, obtained by combining $1x0 \mid 1$ and $00x \mid 0$, is covered by $D0x \mid D$ and does not have to be retained). Propagation $D$-cubes for multiple errors occurring on the inputs of $Z$ can be similarly derived. □

Clearly, for every propagation $D$-cube there exists a corresponding one in which all the $D$ components are complemented. For example, if $x1D \mid \bar{D}$ is a propagation $D$-cube, then so is the cube $x1\bar{D} \mid D$.

A TG algorithm for gate-level circuits uses built-in procedures for line justification and error propagation. In a TG algorithm for module-level circuits, whenever an output line of a module has to be justified (or errors have to be propagated through a module), the table of primitive cubes (or propagation $D$-cubes) associated with the type of the module is searched to find a cube that is compatible with the current line values. The intersection operator defined in Figure 2.3 is now extended to composite logic values as shown in Figure 6.70.

| $\cap$ | 0 | 1 | $D$ | $\overline{D}$ | $x$ |
|---|---|---|---|---|---|
| 0 | 0 | $\varnothing$ | $\varnothing$ | $\varnothing$ | 0 |
| 1 | $\varnothing$ | 1 | $\varnothing$ | $\varnothing$ | 1 |
| $D$ | $\varnothing$ | $\varnothing$ | $D$ | $\varnothing$ | $D$ |
| $\overline{D}$ | $\varnothing$ | $\varnothing$ | $\varnothing$ | $\overline{D}$ | $\overline{D}$ |
| $x$ | 0 | 1 | $D$ | $\overline{D}$ | $x$ |

**Figure 6.70** 5-valued intersection operator

The following example illustrates module-level TG.

**Example 6.17:** The circuit shown in Figure 6.71 uses two modules of the type described in Figure 6.69. Assume that the target fault is $P$ *s-a-1*, which is activated by $P$=0. To propagate the error $P = \overline{D}$ through *M1*, we intersect the current values of *M1* ($QPRS = x\overline{D}x|x$) with the propagation $D$-cubes given in Figure 6.69(c), and with those obtained by complementing all $D$s and $\overline{D}$s. The first consistent propagation $D$-cube is $0\overline{D}0|\overline{D}$. Thus we set $Q = R = 0$ to obtain $S = \overline{D}$. To check the effect of these values on *M2*, we intersect its current values ($RQOT = 00x|x$) with the primitive cubes given in Figure 6.69(b). A consistent intersection is obtained with the second cube with $Z = 0$. Hence $T = 0$, which precludes error propagation through gate $V$. Then we backtrack, looking for another propagation $D$-cube for *M1*. The cube $1\overline{D}1|D$ is consistent with the current values of *M1*. The values $R$=$Q$=1 do not determine a value for $T$. To propagate $S$=$D$, we need $T$=1. Then we intersect the values of *M2* ($RQOT = 11x|1$) with the primitive cubes with $Z$=1. The first cube is consistent and implies $O = 0$. The generated vector is $PQRO = 0110$. □

Compared with its gate-level model, a module-level model of a circuit has a simpler structure. This results from having fewer components and lines, and also from eliminating the internal fanout structure of modules. (A circuit using exclusive-OR modules offers a good example of this simplification.) Another advantage of a module-level model occurs in circuits containing many instances of the same module type, because the preprocessing required to determine the primitive cubes and the propagation $D$-cubes is done only once for all the modules of the same type. While a simpler circuit structure helps the TG algorithm, the components are now more complex and their individual processing is more complicated. Because of this trade-off, the question of whether TG at the module level is more efficient than TG at the gate level does not have a general answer [Chandra and Patel 1987].

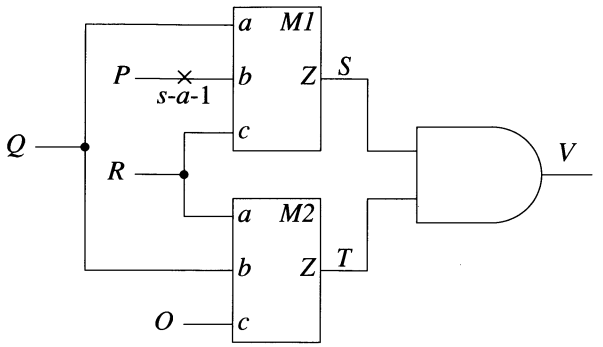



**Figure 6.71** Module-level circuit

In general, modules are assumed to be free of internal faults, and TG at the module level is limited to the pin-fault model. This limitation can be overcome in two ways. The first approach is to use a hierarchical mixed-level model, in which one module at a time is replaced by its gate-level model. This allows generating tests for faults internal to the expanded module. The second approach is to preprocess every type of module by generating tests for its internal faults. To apply a precomputed test to a module $M$ embedded in a larger circuit, it is necessary to justify the values required at the inputs of $M$ using the modules feeding $M$, and at the same time, to propagate the error(s) from the output(s) of $M$ to POs using the modules fed by $M$. A design for testability technique that allows precomputed tests to be easily applied to embedded modules is discussed in [Batni and Kime 1976]. Other approaches to module-level TG are described in [Somenzi *et al.* 1985, Murray and Hayes 1988, Renous *et al.* 1989].

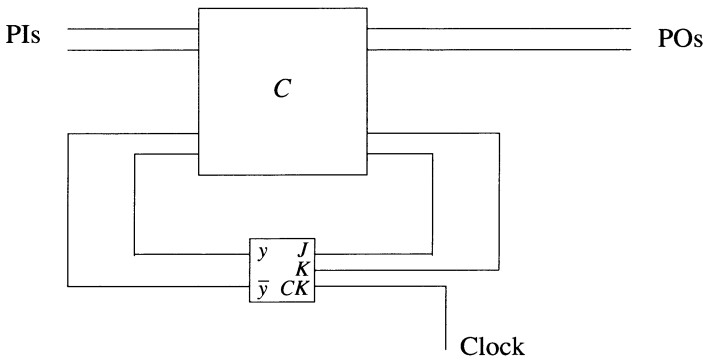
# 6.3 ATG for SSFs in Sequential Circuits

## 6.3.1 TG Using Iterative Array Models

In this section we show how the TG methods for combinational circuits can be extended to sequential circuits. First we restrict ourselves to synchronous sequential circuits whose components are gates and clocked flip-flops. The extension is based on the modeling technique introduced in Section 2.1.2, which transforms a synchronous sequential circuit into an iterative combinational array (see Figure 2.9). One cell of this array is called a *time frame*. In this transformation a F/F is modeled as a combinational element having an additional input $q$ to represent its current state and an additional output $q^+$ to represent its next state, which becomes the current state in the next time frame. An input vector of the iterative combinational array represents an input sequence for the sequential circuit. Initially we assume that all F/Fs are directly driven by the same clock line which is considered to be fault-free. With these assumptions we can treat the clock line as an implicit line in our model.

Figure 6.72(b) shows the structure of one time frame of an iterative array model corresponding to the circuit in Figure 6.72(a) that uses *JK* F/Fs. The gate-level model for the *JK* F/F implements the equations

$$q^+ = J\bar{q} + \bar{K}q$$
$$y = q$$
$$\bar{y} = \bar{q}$$




(a)

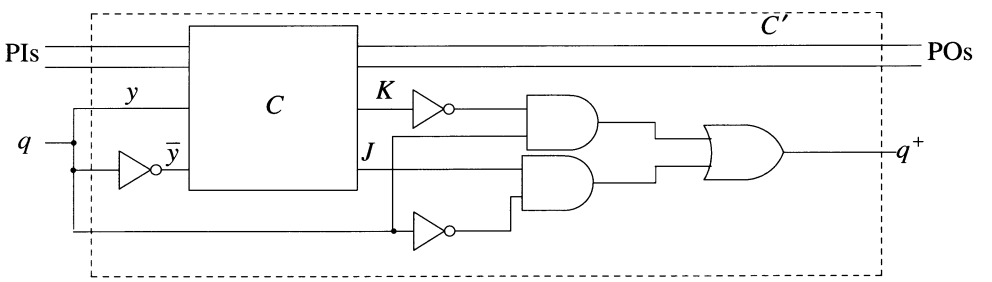



**Figure 6.72** Sequential circuit with *JK* F/F (a) General model (b) Model for one time frame

If we are not interested in faults internal to F/Fs, we can model the circuit that realizes $q^+$ as a module with three inputs ($J$, $K$, and $q$).

Because all time frames have an identical structure, there is no need to actually construct the complete model of the iterative array. Thus the TG algorithms can use the same structural model for every time frame; however, signal values in different time frames should be separately maintained.

Since the circuit $C'$, obtained by adding the F/F models to the original combinational circuit $C$, is also a combinational circuit, we can apply any TG algorithm discussed in Section 6.2. First let us consider a fault-oriented algorithm. A generated test vector $t$ for $C'$ may specify values both for PIs and for $q$ variables; the latter must be justified in the previous time frame. Similarly, $t$ may not propagate an error to a PO but to a $q^+$ variable. Then the error must be propagated into the next time frame. Thus the search process may span several time frames, going both backward and forward in time. A major difficulty arises from the lack of any *a priori* knowledge about the number of time frames needed in either direction.

When dealing with multiple time frames, the target fault is present in every time frame. In other words, the original single fault is equivalent to a multiple fault in the iterative array model. Consequently, an error value ($D$ or $\overline{D}$) may propagate onto the faulty line itself (this cannot happen in a combinational circuit with a single fault). If the faulty line is *s-a-c*, and the value propagating onto it is $v/v_f$, the resulting composite value is $v/c$ (see Figure 6.73). Note that when $v$=$c$, the faulty line stops the propagation of fault effects generated in a previous time frame.

| Value propagated onto line *l* | Fault of line *l* | Resulting value of line *l* |
|---|---|---|
| $D$ | *s-a-0* | $D$ |
| $D$ | *s-a-1* | 1 |
| $\overline{D}$ | *s-a-0* | 0 |
| $\overline{D}$ | *s-a-1* | $\overline{D}$ |

**Figure 6.73** Result of a fault effect propagating to a faulty line

When generating a test sequence for a fault it is never necessary to enter the same state twice. Moreover, allowing a previously encountered state to be entered again will cause the program to go into an infinite loop. To avoid this, any TG algorithm for sequential circuits includes some mechanism to monitor the sequence of states and to initiate backtracking when states are repeated.

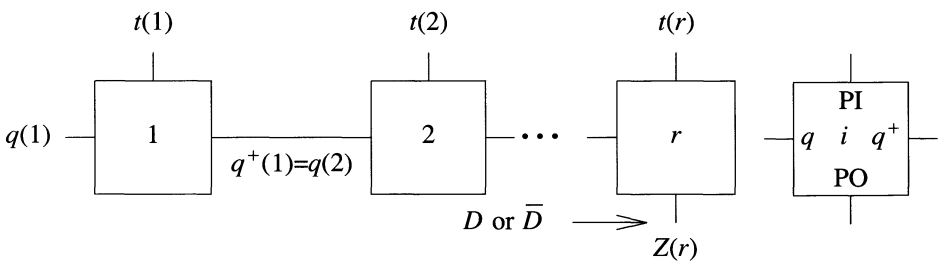
### TG from a Known Initial State

The most general form of a TG algorithm for sequential circuits should assume that the initial state in the fault-free and in the faulty circuit is unknown, because F/F states are arbitrary when power is first applied to a circuit. First we outline a less general algorithm that assumes that the initial state vector $q(1)$ is known both in the fault-free and the faulty circuit (see Figure 6.74). Here the search process goes only forward in time. Since we desire to generate a test sequence as short as possible, we will try a sequence of length $r$ only if we fail to generate a sequence of length $r$–1. Thus when we consider an iterative array of $r$ time frames, we ignore the POs of the first $r$–1 frames. We also ignore the $q^+$ outputs of the last frame. Once the iterative array model is built, we can use any of the fault-oriented TG algorithms presented in

Section 6.2.1 (some minor modification is required to account for the existence of the faulty line in every frame).

```
r=1
repeat
   begin
      build model with r time frames
      ignore the POs in the first r−1 frames
      ignore the q+ outputs in the last frame
      q(1) = given initial state
      if (test generation is successful) then return SUCCESS
      /* no solution with r frames */
      r = r + 1
   end
until r = f_max
return FAILURE
```

(a)




(b)

**Figure 6.74** TG from a known initial state (a) General procedure (b) Iterative array model

An important question is when to stop increasing the number of time frames. In other words, we would like to know the maximal length of a possible test sequence for a fault. For a circuit with $n$ F/Fs, for every one of the $2^n$ states of the fault-free circuit, the faulty circuit may be in one of its $2^n$ states. Hence no test sequence without repeated states can be longer then $4^n$. This bound, however, is too large to be useful in practice, so the number of time frames is bounded by a (much smaller) user-specified limit $f_{max}$.

While conceptually correct, the algorithm outlined in Figure 6.74 is inefficient, because when the array with $r$ frames is created, all the computations previously done in the first $r-1$ frames are ignored. To avoid this waste, we save the partial sequences that can be extended in future frames. Namely, while searching for a sequence $S_r$ of length $r$, we save every such sequence that propagates errors to the $q^+$ variables of the

$r$-th frame (and the resulting state vector $q^+(r)$) in a set $SOL_r$ of partial solutions. If no sequence $S_r$ propagates an error to a PO, the search for a sequence $S_{r+1}$ starts from one of the saved states $q^+(r)$.

**Example 6.18:** Consider the sequential circuit in Figure 6.75(a) and the fault $a$ $s$-$a$-1. We will derive a test sequence to detect this fault, assuming the initial state is $q_1$=$q_2$=0. Figure 6.75(b) shows the model of one time frame.

*Time frame 1:* We apply $q_1$=$q_2$=0, which creates a $\bar{D}$ at the location of the fault. The vector $I$(1)=1 propagates the error to $q_2^+$. Because the fault effect does not reach $Z$, we save the sequence $S_1$ = (1) and the state $q^+(1) = (0,\bar{D})$ in the set $SOL_1$.

*Time frame 2:* We apply $(0,\bar{D})$ to the $q$ lines of frame 2. The *D-frontier* is $\{G_1,G_3,G_4\}$. Selecting $G_1$ or $G_4$ for error propagation results in $I$(2)=1 and the next state $q^+(2)=(\bar{D},\bar{D})$. Trying to propagate the error through $G_3$ results in $I$(2)=0 and the next state $q^+(2)=(0,D)$.

*Time Frame 3:* Now the set $SOL_2$ contains two partial solutions: (1) the sequence (1,1) leading to the state $(\bar{D},\bar{D})$ and (2) the sequence (1,0) leading to the state $(0,D)$. Trying the first solution, we apply $(\bar{D},\bar{D})$ to the $q$ lines of frame 3, which results in the *D-frontier* $\{Z,G_1,G_2,G_3,G_4\}$. Selecting $Z$ for error propagation we obtain $I$(3)=1 and $Z$=$D$. The resulting test sequence is $I$=(1,1,1). □

With the given initial state, it may not always be possible to activate the fault in the first time frame; then a sequence is needed for activation.

**Generation of Self-Initializing Test Sequences**

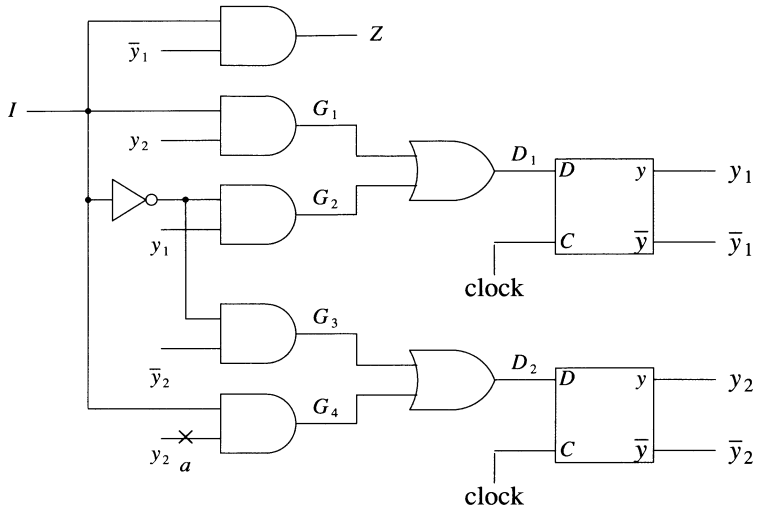
The algorithm outlined in Figure 6.76(a) handles the general case when the initial state is unknown. The fault is activated in one time frame (temporarily labeled 1), and the resulting error is propagated to a PO going forward in time using $r \geq 1$ frames (see Figure 6.76(b)). If some $q$ values of frame 1 are binary, these are justified going backward in time using $p$ time frames temporarily labeled 0, −1, ..., −($p$−1). This process succeeds when the $q$ values in the first time frame are all $x$. Such a test sequence is said to be *self-initializing*. (After $p$ is known, the time frames can be correctly labeled 1,...,$p$,$p$+1,...,$p$+$r$.)

We will try a sequence of length $p + r + 1$ only when we fail to generate a sequence of length $p + r$. To reuse previous computations, the algorithm saves all partial solutions of the form ($q$(−($p$−1); $S_{p+r}$; $q^+(r)$), where $S_{p+r}$ is a sequence obtained with $p + r$ frames that propagates errors to $q^+(r)$.

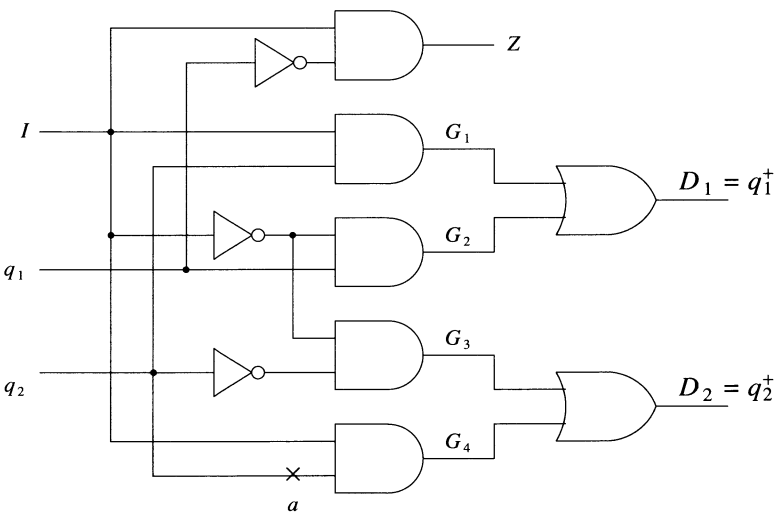
**Example 6.19:** Let us derive a self-initializing test sequence for the fault $Z$ $s$-$a$-0 in the circuit of Figure 6.77(a). Figure 6.77(b) shows the structure of one time frame, using a module to realize the function $q^+=J\bar{q}+q\bar{K}$. The primitive cubes of this module are given in Figure 6.69(b) (substitute $J$,$q$,$K$,$q^+$ for $a$,$b$,$c$,$Z$).

*Time frame 1:* The only test that detects $Z$ $s$-$a$-0 is ($I$,$q_1$)=10. Since $Z$ has value $D$, an error is propagated to the PO using $r$=1 frames. Since $q_1(1)\neq x$, we need $p\geq 1$.

*Time frame* 0: To justify $q_1^+(0)$=0, we first try $J_1$=0 and $K_1$=1 (see the decision tree in Figure 6.78(a)). Both solutions justifying $J_1$=0 involve assignments to $q_2$. The other two solutions justifying $q_1^+$ involve assignments to $q_1$. Hence there is no solution with




(a)




(b)

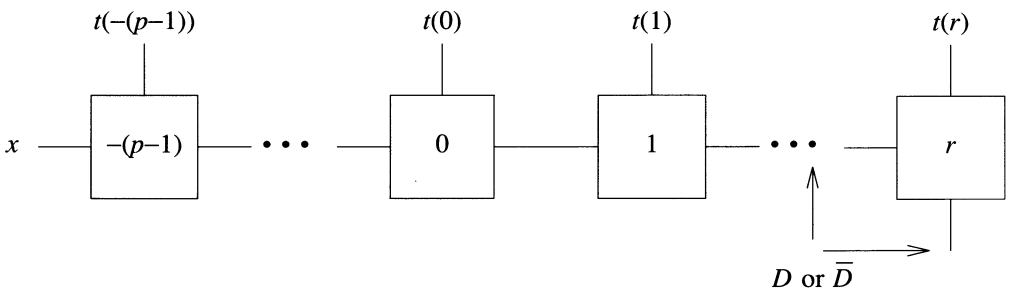
**Figure 6.75**

$p$=1. Next we try $p$=2 and we return to the first solution for $J_1$=0, namely $I$=0 and $q_2$=1.

*Time frame* −1: To justify $q_2^+$=1, we first try $J_2$=1 and $K_2$=0, which are both satisfied by setting $I$=0. Since all lines are justified while $q_1$ and $q_2$ of this time frame are both

```
r = 1
p = 0
repeat
   begin
      build model with p + r time frames
      ignore the POs in the first p + r − 1 frames
      ignore the q+ outputs in the last frame
      if (test generation is successful and every q input in the first frame has
         value x) then return SUCCESS
      increment r or p
   end
until (r+p=f_max)
return FAILURE
```

(a)



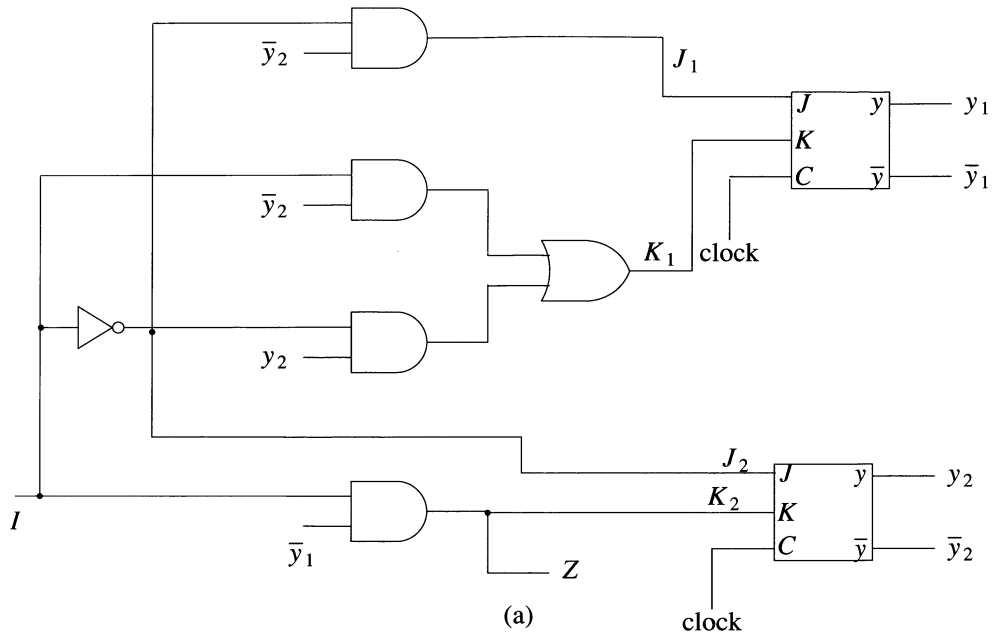

(b)

**Figure 6.76** Deriving a self-initializing test sequence (a) General procedure (b) Iterative array model

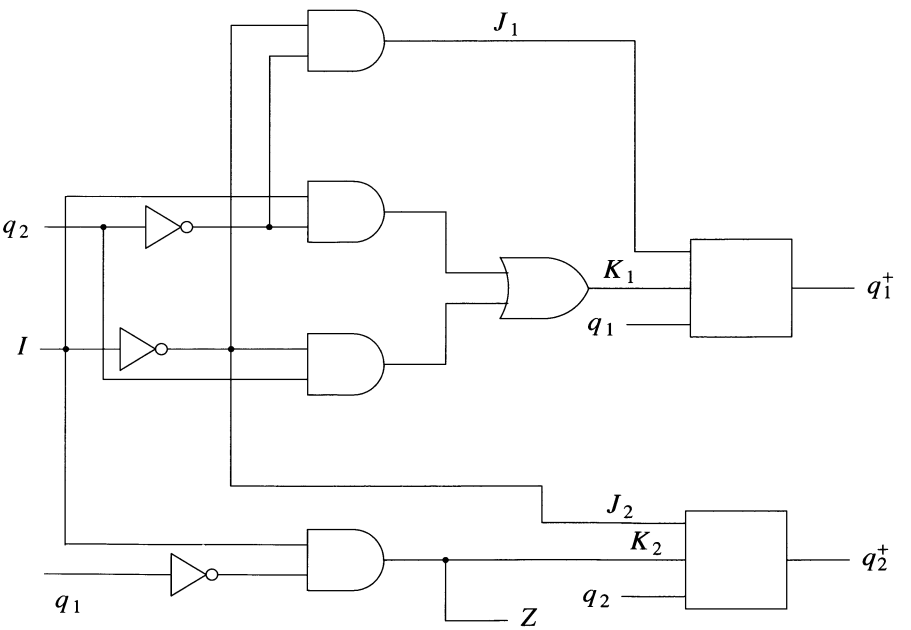
$x$, we have obtained the self-initializing test sequence $I$=(0,0,1) with $p$=2 and $r$=1. Figure 6.78(b) shows the corresponding iterative array. □

### Other Algorithms

A major complication in the algorithm outlined in Figure 6.76 results from the need of going both forward and backward in time. One approach that avoids this problem is the EBT (Extended Backtrace) method [Marlett 1978], whose search process goes only backward in time. EBT first selects a path from the fault site to a PO (this path may involve several time frames), then sensitizes the path starting from the PO. After the error propagation succeeds, the fault is activated by justifying the needed value at the fault site. All line-justification problems that cannot be solved in the current time frame are continued in the previous frame. Thus all vectors are generated in the order



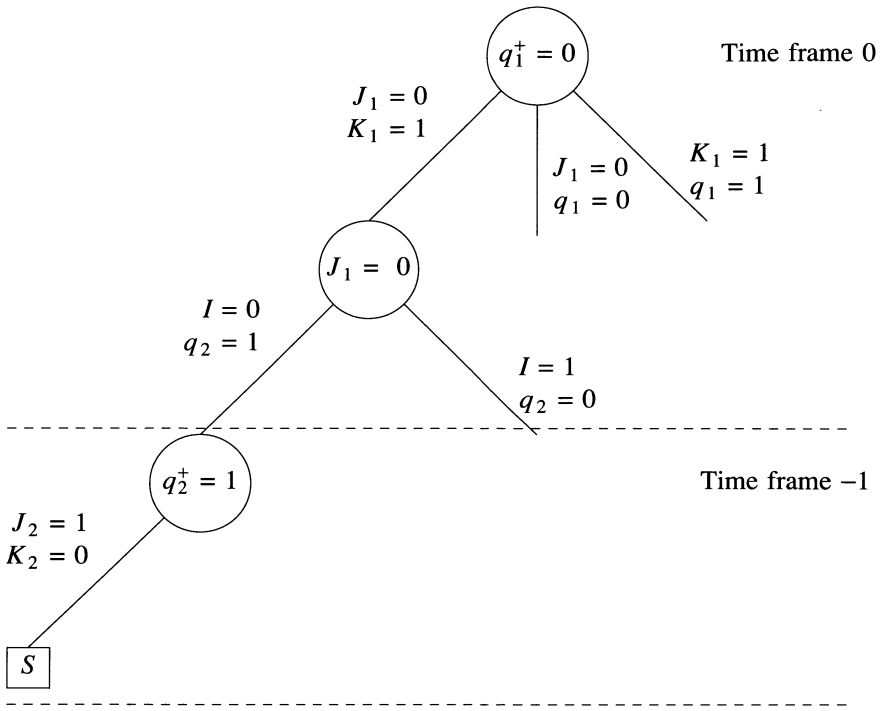

(a)



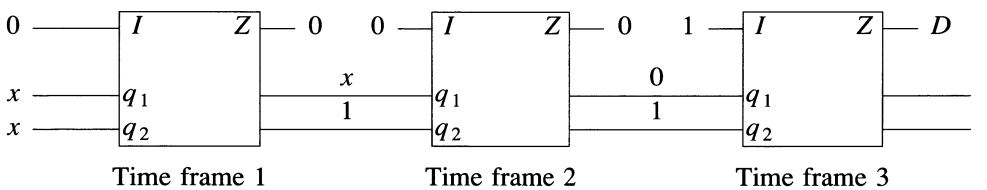

(b)

**Figure 6.77**

opposite to that of application. The same principle is used in the algorithms described in [Mallela and Wu 1985] and [Cheng and Chakraborty 1989].




(a)




(b)

**Figure 6.78** Example 6.19 (a) Decision tree (b) Iterative array

Critical-path TG algorithms for combinational circuits have also been extended to synchronous sequential circuits using the iterative array model [Thomas 1971, Breuer and Friedman 1976].

The method described in [Ma *et al.* 1988] relies on the existence of a *reset state* and assumes that every test sequence starts from the reset state. Like the algorithm given in Figure 6.76(a), it begins by activating the fault in time frame 1 and propagating the

error forward to a PO, using several time frames if necessary. The state required in frame 1, *q(1)*, is justified by a sequence that brings the circuit from the reset state to *q(1)*. To help find this sequence, a preprocessing step determines a *partial state-transition graph* by systematically visiting states reachable from the reset state. Thus transfer sequences from the reset state to many states are precomputed. If the state *q(1)* is not in the partial state diagram, then a state-justification algorithm searches for a transfer sequence from the reset state to *q(1)*. This search process systematically visits the states from which *q(1)* can be reached, and stops when any state included in the partial state-transition graph is encountered.

**Logic Values**

Muth [1976] has shown that an extension of the *D*-algorithm restricted to the values $\{0,1,D,\overline{D},x\}$ may fail to generate a test sequence for a detectable fault, while an algorithm using the complete set of nine composite values would be successful.

With the nine composite values, a line in both the fault-free and the faulty circuit may assume one of the three values $\{0,1,u\}$. Because of the limitations of the 3-valued logic (discussed in Chapter 3), it is possible that a TG algorithm using the 9-valued logic may not be able to derive a self-initializing test, even when one does exist. (A procedure guaranteed to produce an initializing sequence, when one exists, requires the use of multiple unknown values.)

We have seen that for a self-initializing test sequence, all the $q$ variables of the first time frame have value $x$. Some systems use a different value $U$ to denote an initial unknown state. The difference between $U$ and $x$ is that $x \cap a = a$, while $U \cap a = \emptyset$; in other words, an $x$ can be changed during computation, but an $U$ cannot.

One way to avoid the complex problem of initialization is to design sequential circuits that can be easily initialized. The simplest technique is to provide each F/F with a common reset (or preset) line. Then one vector can initialize the fault-free circuit and most of the faulty circuits. Such design techniques to simplify testability will be considered in greater detail in a separate chapter.

**Reusing Previously Solved Problems**

An additional difficulty in TG for sequential circuits is the existence of "impossible" states, that is, states never used in the fault-free operation. For example, a 5-state counter implemented with three F/Fs may have three states that cannot be entered. Trying to justify one of these invalid states during TG is bound eventually to fail and may consume a significant amount of time. To help alleviate this problem, a table of invalid and unjustifiable states can be constructed. Initially, the user may make entries in this table. Then the system itself will enter the states whose justification failed or could not be completed within a user-specified time limit. Before attempting to justify a state **q**, the TG system will compare it with the entries in the invalid-states table; backtracking is initiated if **q** is covered by an invalid state. For example, if $0xx1$ could not be justified (within the allowed limits), then it is useless to try to justify $01x1$, which is a more stringent restriction on the state space.

This concept of keeping track of previous state-justification problems can be extended to successfully solved problems [Rutman 1972]. For example, suppose that we want to justify the state $\mathbf{q}=0x1x$, and that the state $011x$ has been previously justified. Clearly,

we can again apply the sequence used to justify a state covered by **q**. Hence we construct a table of solved state-justification problems and their corresponding solutions. Whenever we wish to justify a state **q**, we search this table to see if **q** (or a state covered by **q**) has been justified before. If so, the stored justification sequence is reused.

**Cost Functions**

The controllability and observability cost functions discussed in Section 6.2.1.3 have been extended to sequential circuits [Rutman 1972, Breuer 1978, Goldstein 1979]. The computation of controllability costs starts by assigning $C0(l)=C1(l)=\infty$ for every line $l$ (in practice, $\infty$ is represented by a large integer). Next the $C0(i)$ and $C1(i)$ values of every PI $i$ are changed to $f_i-1$, where $f_i$ is the fanout count of $i$ (using fanout-based cost functions). Then *cost changes* are further propagated in the same way logic events are propagated in simulation. To compute the new cost of the output of a gate in response to cost changes on gate inputs, we use formulas similar to (6.6) and (6.7). Note that for an AND gate $l$, $C1(l)$ will keep the value $\infty$ until every input of the gate $l$ has a finite $C1$ value. F/Fs are processed based on their equations. The controllability costs of the state variables and of the lines affected by them may change several times during this process, until all costs reach their stable values. The convergence of the cost computation is guaranteed because when a cost changes, its value always decreases. Usually the costs reach their stable values in only a few iterations.

After the controllability costs are determined, observability costs are computed by a similar iterative process. We start by setting $O(l)=\infty$ for every line $l$ and changing the observability cost of every PO to 0. Now changes in the observability costs are propagated going backward through the circuit, applying formulas similar to (6.3) and (6.4).

**Selection of Target Faults**

The structure of a TG system for sequential circuits using a fault-oriented TG algorithm is similar to that outlined in Figure 6.67, except that a test sequence (rather than one vector) is generated to detect the target fault. A self-initializing sequence is necessary only for the first target fault. For any other target fault, TG starts from the state existing at the end of the previously generated sequence.

Every generated sequence is simulated to determine all the faults it detects. An important byproduct of this fault simulation is that it also determines the undetected faults whose effects have been propagated to state variables. This information is used by selecting one such fault as the next target; this "opportunistic" selection saves the effort involved in activating the fault and propagating an error to a state variable.

**Clock Logic and Multiple Clocks**

Until now we have assumed that all F/Fs are directly driven by the same fault-free clock line. In practice, however, synchronous sequential circuits may have multiple clock lines and clocks may propagate through logic on their way to F/Fs. Figure 6.79(a) shows the general structure of such a circuit and Figure 6.79(b) gives an example of clock logic. (We assume that clock PIs do not feed data inputs of F/Fs.)

To handle circuits with this structure, we use F/F models where the clock line appears explicitly. Figure 6.80 illustrates the model for a $D$ F/F that implements the equation

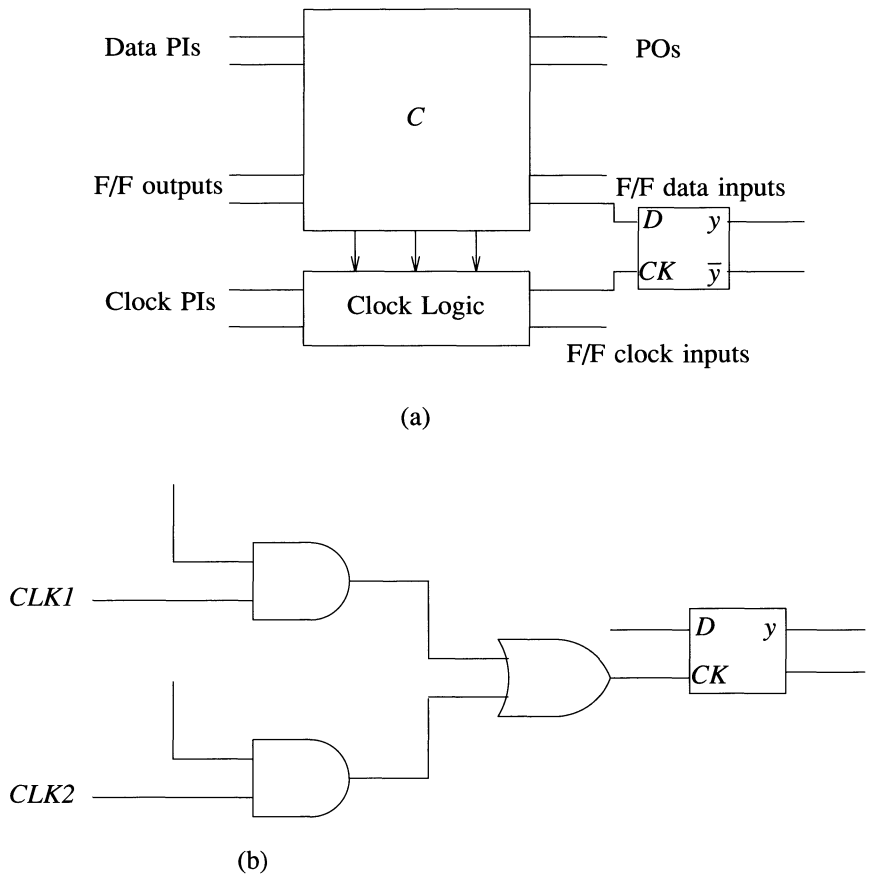



**Figure 6.79** (a) Synchronous circuit with clock logic (a) General structure (b) Example of clock logic

$q^+=CD+\bar{C}q$. Note that in this model the F/F maintains its current state if its clock is not active.

This modeling technique allows the clock logic to be treated as a part of the combinational circuit. A 1(0)-value specified for a clock PI $i$ in a vector of the generated test sequence means that a clock pulse should (should not) be applied to $i$ in that vector. Although the values of the data PIs and of the clock PIs are specified by the same vector, the tester must delay the clock pulses to ensure that the F/F clock inputs change only after the F/F data inputs have settled to their new values.

For a F/F for which clocking is the only means to change its state, certain faults in the clock logic preclude clock propagation and make the F/F keep its initial state. A conventional ATG procedure cannot detect such a fault, because its fault effects are of the form $1/u$ or $0/u$ (never $D$ or $\bar{D}$), which, when propagated to a PO, indicate only a

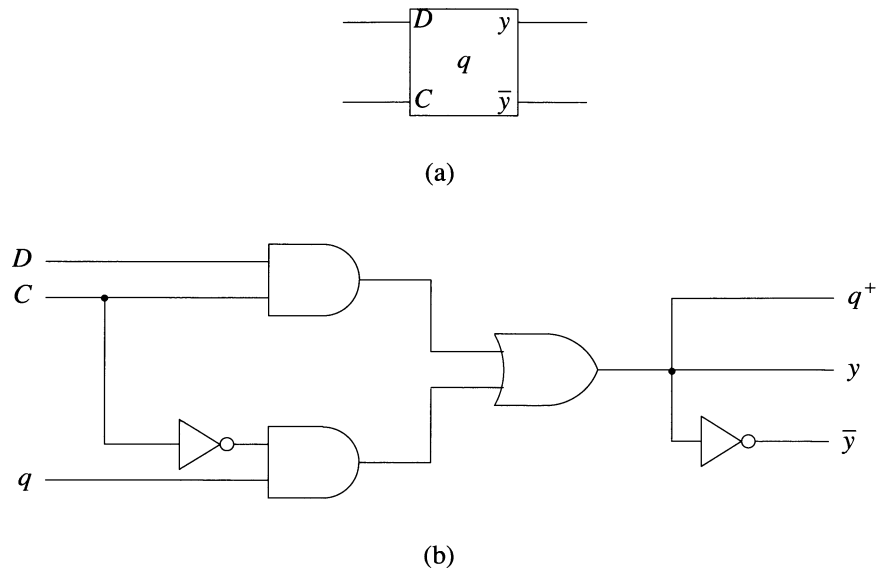



**Figure 6.80** (a) *D* flip-flop (b) Model with explicit clock line

potential detection. The method described in [Ogihara *et al.* 1988] takes advantage of the F/F maintaining its initial state, say $q_i$, and tries to propagate both a $1/q_i$ and a $0/q_i$ fault effect to a PO (they may propagate at different times). Then one of these two fault effects is guaranteed to indicate a definite error.

**Complexity Issues**

TG for sequential circuits is much more complex than for combinational circuits, mainly because both line justification and error propagation involve multiple time frames. In the worst case the number of time frames needed is an exponential function of the number of F/Fs.

An important structural property of a sequential circuit is the presence of *cycles*, where a cycle is a loop involving F/Fs [Miczo 1983]. TG for a sequential circuit without cycles is not much more difficult than for a comparable combinational circuit. The complexity of TG for a sequential circuit with cycles is directly related to its cyclic structure, which can be characterized by

- the number of cycles;
- the number of F/Fs per cycle;
- the number of cycles a F/F is involved in (reflecting the degree of interaction between cycles) [Lioy *et al.* 1989].

We cannot expect TG for large sequential circuits with complex cyclic structures to be practically feasible. Practical solutions rely on design for testability techniques.

Several such techniques (to be discussed in Chapter 9) transform a sequential circuit into a combinational one during testing. Cheng and Agrawal [1989] describe a technique whose goal is to simplify the cyclic structure of a sequential circuit.

**Asynchronous Circuits**

TG for asynchronous circuits is considerably more difficult than for synchronous circuits. First, asynchronous circuits often contain races and are susceptible to improper operation caused by hazards. Second, to obtain an iterative array model of the circuit, all feedback lines must be identified; this is a complex computational task. Finally, correct circuit operation often depends on delays intentionally placed in the circuit, but none of the TG algorithms previously discussed takes delays into account.

Note that an asynchronous circuit may go through a sequence of states in response to an input change. We assume that the PO values are measured only after a stable state is reached and that PIs are not allowed to change until the circuit has stabilized. Based on these assumptions, we can use the modeling technique illustrated in Figure 6.81 [Breuer 1974]. Here the time scale is divided into *frames* and *phases*. A time frame $i$ corresponds to the application of one input vector $x(i)$. (Unlike synchronous circuits, two consecutive vectors must differ in at least one variable). Every frame $i$ is composed of one or more phases to reflect the sequence of changes of the state variables. All the phases of the same frame receive the same vector $x(i)$. PO values are observed only after stability has been reached ($y$=$Y$) in the last phase of a frame. One problem is that the number of phases in a frame is not known *a priori*.

The TG procedures for synchronous circuits can be extended to handle the model in Figure 6.81 and can be further modified so that the generated test will be free of hazards [Breuer and Harrison 1974]. In practice, however, ATG for large asynchronous circuits is not feasible.

### 6.3.2 Simulation-Based TG

The principle of simulation-based TG [Seshu 1965] is to explore the space of input vectors as follows:

1. Generate and (fault) simulate trial vectors.
2. Based on the simulation results, evaluate trial vectors according to some cost function.
3. Select the "best" trial vector and add it to the test sequence.

Initially we can start with an arbitrary (or user-specified) vector. Trial vectors are usually selected among the "neighbors" of the current input vector $t$, i.e., vectors different from $t$ in one bit. The selection process is random, so this method is in part similar to RTG. The cost function depends on the objective of the current phase of the TG process [Agrawal *et al.* 1988]:

1. *Initialization:* Here the objective is to set all F/Fs to binary values. The cost function is the number of F/Fs in an unknown state. (This phase does not require fault simulation.)
2. *Test Generation for Groups of Faults:* Here the objective is to detect as many faults as possible, hence all undetected faults are simulated. For every activated

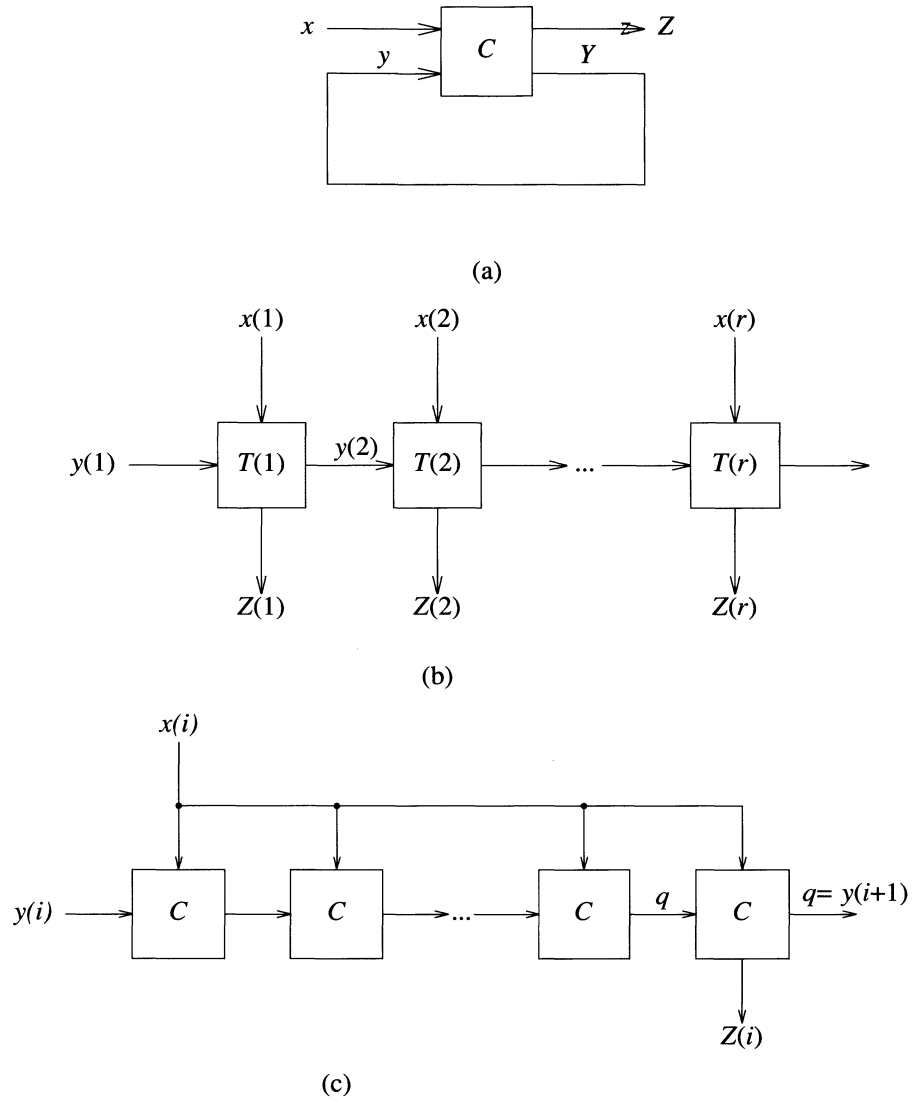



**Figure 6.81** (a) Asynchronous circuit (b) Iterative array model (c) Structure of time frame *T(i)*

fault $i$ we compute its cost $c_i$ as the shortest distance between the gates on its *D-frontier* and POs. The distance is the weighted number of elements on a path, with a F/F having a heavier weight than a gate. The cost $c_i$ represents a measure of how close is fault $i$ to being detected. The cost function is $\sum c_i$, where the summation is taken over a subset of faults containing the faults having the lowest costs.

3. *Test Generation for Individual Faults:* Here we target one of the remaining undetected faults and the objective is to generate a test sequence to detect it. The cost function is a *dynamic testability measure*, designed to measure the additional effort required to detect the fault, starting from the current state.

In all three phases, a "good" trial vector should reduce the cost. During the initialization phase, reducing the cost means initializing more F/Fs. When the cost becomes 0, all F/Fs have binary values. In the other two phases, reducing the cost means bringing a group of faults, or an individual fault, closer to detection. In principle, we should evaluate all possible trial vectors and select the one leading to the minimal cost. (In a circuit with $n$ PIs, $n$ trial vectors can be obtained by 1-bit changes to the current vector.) In practice, we can apply a "greedy" strategy, which accepts the first vector that reduces the cost. Rejecting a trial vector that does not reduce the cost means restoring the state of the problem (values, fault lists, etc.) to the one created by the last accepted vector; this process is similar to backtracking. At some point, it may happen that no trial vector can further reduce the cost, which shows that the search process has reached a local minimum. In this case a change of strategy is needed, such as (1) accept one vector that increases the cost, (2) generate a new vector that differs from the current one in more than one bit, (3) go to the next phase.

An important advantage of simulation-based TG is that it can use the same delay model as its underlying fault simulator. It can also handle asynchronous circuits. Because its main component is a conventional fault simulator, simulation-based TG is easier to implement than methods using the iterative array model.

SOFTG (*S*imulator-*O*riented *F*ault *T*est *G*enerator) is another simulation-based TG method [Snethen 1977]. SOFTG also generates a test sequence in which each vector differs from its predecessor in one bit. Rather than randomly selecting the bit to change, SOFTG determines it by a backtrace procedure similar to the one used in PODEM.

### 6.3.3 TG Using RTL Models

The TG methods described in the previous section work with a circuit model composed of gates and F/Fs. By contrast, a designer or a test engineer usually views the circuit as an interconnection of higher-level components, such as counters, registers, and multiplexers. Knowing how these components operate allows simple and compact solutions for line-justification and error-propagation problems. For example, justifying a 1 in the second bit of a counter currently in the all-0 state can be done by executing two increment operations. But a program working with a low-level structural model (in which the counter is an interconnection of gates and F/Fs) cannot use this type of functional knowledge.

In this section we present *TG methods that use RTL models* for components (or for the entire circuit). They generate tests for SSFs and can be extended to handle stuck RTL variables. (TG for functional faults is the topic of Chapter 8.)

The main motivation for RTL models is to speed up TG for sequential circuits and to increase the size of the circuits that can be efficiently processed. A second motivation is the use of "off-the-shelf" commercial components for which gate-level models are not available, but whose RTL descriptions are known or can be derived from their functional specifications.

### 6.3.3.1 Extensions of the *D*-Algorithm

To extend the *D*-algorithm to circuits containing RTL components, we keep its overall structure, but we extend its basic operations — line-justification, error-propagation, and implication. We will consider devices of moderate complexity, such as shift registers and counters [Breuer and Friedman 1980]. The function of such a device is described using the following primitive RTL operators:

- *Clear* (reset),
- parallel *Load*,
- *Hold* (do nothing),
- shift *Left*,
- shift *Right*,
- increment (count *Up*),
- decrement (count *Down*).

Figure 6.82(b) describes the operation of the bidirectional shift register shown in Figure 6.82(a), where **Y** and **y** denote the next and the current state of the register, $Clock = \uparrow$ represents a 0 to 1 transition of *Clock*, and $Clock \neq \uparrow$ means that the consecutive values of *Clock* are 00 or 11 or 10. The outputs **Q** always follow the state variables **y**.

The primitive RTL operators are implemented by built-in *generic algorithms*, which are applicable to any $n$-bit register. Different types of components are characterized by different sets of conditions that activate the operators; for example negative versus positive clock, or synchronous versus asynchronous clear. For every type of device we use a *translation table* similar to the one in Figure 6.82(b) to provide the mapping between operators and the values of the control lines.

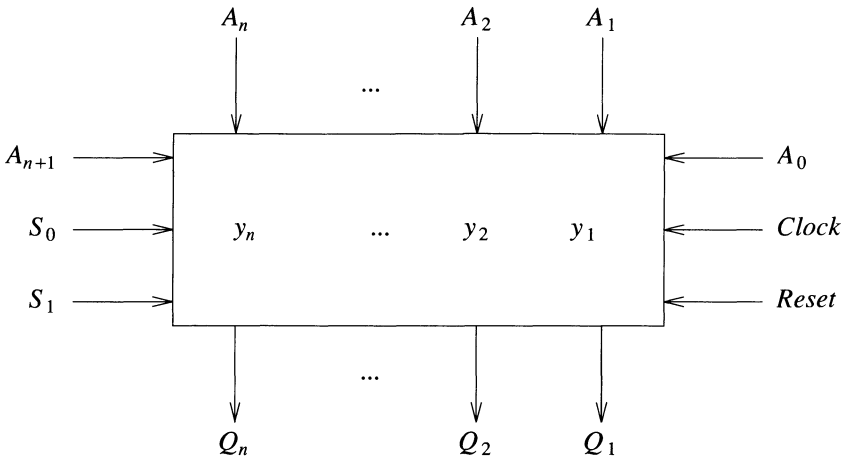
**Implication**

A first type of implication is to compute **Y** given **y**, **A** and the values of the control lines. For this we first determine what operation will be executed. When some control lines have $x$ values, the device may execute one of a set of *potential operations*. For example, if $(Reset, Clock, S_0, S_1) = 1 \uparrow 1\ x$, the shift register may execute either *Right* or *Load*. Even if we do not know the operation executed, some of the resulting values can be implied by the following *union algorithm*, obtained by taking the common results of *Right* and *Load*:

$$\begin{array}{ll} \textbf{if } A_i = y_{i+1} \textbf{ then } Y_i = A_i & \\ \textbf{else } Y_i = x & i = 1, \ldots, n-1 \\ \textbf{if } A_n = A_{n+1} \textbf{ then } Y_n = A_n & \\ \textbf{else } Y_n = x & \end{array}$$

There are 20 union algorithms for the shift register of Figure 6.82 [Breuer and Friedman 1980].

A second type of implication is to determine what operation could have been executed, given the values of **Y**. For example, if the values $A_i$ are compatible with the values $Y_i$,




(a)

| Control lines | | | | Operation | Algorithm | |
|---|---|---|---|---|---|---|
| *Reset* | *Clock* | $S_0$ | $S_1$ | | | |
| 0 | $x$ | $x$ | $x$ | *Clear* | $Y_i = 0$ | $i = 1, ..., n$ |
| 1 | $\neq\uparrow$ | $x$ | $x$ | *Hold* | $Y_i = y_i$ | $i = 1, ..., n$ |
| 1 | $\uparrow$ | 0 | 0 | *Hold* | | |
| 1 | $\uparrow$ | 0 | 1 | *Left* | $Y_i = y_{i-1}$<br>$Y_1 = A_0$ | $i = 2, ..., n$ |
| 1 | $\uparrow$ | 1 | 0 | *Right* | $Y_i = y_{i+1}$<br>$Y_n = A_{n+1}$ | $i = 1, ..., n-1$ |
| 1 | $\uparrow$ | 1 | 1 | *Load* | $Y_i = A_i$ | $i = 1, ..., n$ |

(b)

**Figure 6.82** Shift register (a) Module-level view (b) Generic algorithms

i.e., $Y_i \cap A_i \neq \varnothing$ for $i = 1, ..., n$, then *Load* is a potential operation. Figure 6.83 summarizes these implications for a shift register.

A third type of implication is to determine the values of **A** or **y**, knowing **Y** and the operation executed. This is done using the concept of *inverse operations*. That is, if $\mathbf{Y} = g(\mathbf{y})$, then $\mathbf{y} = g^{-1}(\mathbf{Y})$. Figure 6.84 presents the inverse operations for a shift register.

Note that $Clear^{-1}$ is undefined, because the previous values $y_i$ cannot be determined once the register has been cleared.

| Conditions | | Potential operation |
|---|---|---|
| $Y_i \cap y_i \neq \varnothing$ | $i = 1, ..., n$ | *Hold* |
| $Y_i \cap A_i \neq \varnothing$ | $i = 1, ..., n$ | *Load* |
| $Y_i \cap y_{i-1} \neq \varnothing$<br>$Y_1 \cap A_0 \neq \varnothing$ | $i = 2, ..., n$ | *Left* |
| $Y_i \cap y_{i+1} \neq \varnothing$<br>$Y_n \cap A_{n+1} \neq \varnothing$ | $i = 1, ..., n-1$ | *Right* |
| $Y_i \neq 1$ | $i = 1, ..., n$ | *Clear* |

**Figure 6.83** Potential operations for a shift register

| Inverse Operation | Algorithm | |
|---|---|---|
| $Hold^{-1}$ | $y_i = Y_i$ | $i = 1, ..., n$ |
| $Left^{-1}$ | $y_i = Y_{i+1}$<br>$A_0 = Y_1$ | $i = 1, ..., n-1$ |
| $Right^{-1}$ | $y_i = Y_{i-1}$<br>$A_{n+1} = Y_n$ | $i = 2, ..., n$ |
| $Load^{-1}$ | $A_i = Y_i$ | $i = 1, ..., n$ |

**Figure 6.84** Inverse operations for a shift register

**Example 6.20:** For the shift register of Figure 6.82, assume that $n = 4$, (*Reset*, *Clock*, $S_0$, $S_1$) = $1 \uparrow x\ 0$, $\mathbf{Y} = (Y_4, Y_3, Y_2, Y_1) = x\ x\ 0\ x$, $\mathbf{y} = (y_4, y_3, y_2, y_1) = x\ 1\ x\ 0$ and $\mathbf{A} = (A_5, A_4, A_3, A_2, A_1, A_0) = x\ x\ x\ x\ x\ x$. From the values of the control lines and Figure 6.82(b), we determine that the potential operations are *Right* and *Hold*. Checking their conditions given in Figure 6.83, we find that *Hold* is possible but *Right* is not (since $Y_2 \cap y_3 = \varnothing$). Hence we conclude that the operation is *Hold*. This implies $S_0 = 0$, $Y_3 = y_3 = 1$ and $Y_1 = y_1 = 0$. Using the inverse operator $Hold^{-1}$, we determine that $y_2 = Y_2 = 0$. □

### Line Justification

For line justification, we first determine the set of potential operations the device can execute; among these we then select one that can produce the desired result.

**Example 6.21:** Assume we wish to justify $\mathbf{Y} = 0110$ and that currently $\mathbf{y} = x\ 0\ x\ x$, $\mathbf{A} = x\ x\ x\ x\ x\ x$, and (*Reset*, *Clock*, $S_0$, $S_1$) = $1 \uparrow x\ x$. From the values of the control lines we determine that the potential operations are {*Hold*, *Left*, *Right*, *Load*}. *Hold* is not possible because $Y_3 \cap y_3 = \varnothing$. *Right* is not possible since $Y_2 \cap y_3 = \varnothing$. Thus we have two solutions:

1. Execute *Left*. For this we set $(S_0, S_1) = 01$. By applying $Left^{-1}$ to $\mathbf{Y} = 0110$, we determine that $(y_2, y_1) = 11$ and $A_0 = 0$.
2. Execute *Load*. For this we set $(S_0, S_1) = 11$. By applying $Load^{-1}$ to $\mathbf{Y} = 0110$, we determine that $(A_4, A_3, A_2, A_1) = 0110$.

Before selecting one of these two solutions, we can imply $S_1 = 1$, which is common to both of them. □

**Error Propagation**

We consider only single error-propagation problems, i.e., we have a $D$ or $\overline{D}$ affecting one input of the shift register. The simple case occurs when the error affects a data line $A_i$. Then to propagate the error through the shift register we execute

1. *Left* if $i = 0$;
2. *Load* if $1 \leq i \leq n$;
3. *Right* if $i = n + 1$.

To propagate an error that affects a control line, we have to make the shift register in the faulty circuit $f$ execute an operation $O_f$ different from the operation $O$ done by the shift register in the good circuit, and to ensure that the results of $O$ and $O_f$ are different. Figure 6.85 gives the conditions needed to propagate a $D$ on a control line and the resulting *composite operation* $O/O_f$. The conditions needed to propagate a $\overline{D}$ are the same, but the resulting composite operation is reversed (that is, if $D$ produced $O_1/O_2$, $\overline{D}$ will produce $O_2/O_1$). The notation $Clock = \uparrow/\neq\uparrow$ means that *Clock* has a 0-to-1 transition in the good circuit but not in the faulty circuit (i.e., the consecutive composite values of *Clock* can be $0D$ or $\overline{D}1$ or $\overline{D}D$).

| *Reset* | *Clock* | $S_0$ | $S_1$ | $O/O_f$ |
|---|---|---|---|---|
| $D$ | $x$ | $x$ | $x$ | *{Hold,Left,Right,Load}/Clear* |
| 1 | $\uparrow$ | $D$ | 0 | *Right/Hold* |
| 1 | $\uparrow$ | $D$ | 1 | *Load/Left* |
| 1 | $\uparrow$ | 0 | $D$ | *Left/Hold* |
| 1 | $\uparrow$ | 1 | $D$ | *Load/Right* |
| 1 | $\uparrow/\neq\uparrow$ | 1 | $x$ | *{Load,Right}/Hold* |
| 1 | $\uparrow/\neq\uparrow$ | $x$ | 1 | *{Load,Left}/Hold* |

**Figure 6.85** Propagation of errors on control lines of a shift register

**Example 6.22:** Assume we want to propagate a $D$ from $S_1$ and that currently $\mathbf{y} = 0000$ and $\mathbf{A} = 010000$. From Figure 6.85 we have two potential solutions:

1) $(Reset, Clock, S_0) = 1 \uparrow 0$ with the composite operation *Left/Hold*. This is unacceptable, since the result is $\mathbf{Y} = 0000$.

2) (*Reset*, *Clock*, $S_0$) = 1 ↑ 1 with the composite operation *Load/Right*. This produces **Y** = *D*000. □

An ATG system that successfully implements the concepts introduced in this section is described in [Shteingart *et al.* 1985]. Methods for error propagation through language constructs used in RTLs are presented in [Levendel and Menon 1982, Norrod 1989].

### 6.3.3.2 Heuristic State-Space Search

SCIRTSS (*S*equential *C*ircuit *T*est *S*earch System) [Hill and Huey 1977] is an ATG system that generates tests for synchronous circuits for which both a low-level model (gate and F/F) and an RTL-level model are available. Based on the low-level model, SCIRTSS employs a TG algorithm for combinational circuits to generate a test for a target SSF in one time frame. Then the search for finding sequences to propagate an error to a PO and to bring the circuit into the needed state is done using the RTL model.

SCIRTSS assumes an RTL model that provides a clear separation between the data and the control parts of the circuit, as shown in Figure 6.86. This separation allows the control inputs to be processed differently from the data inputs.

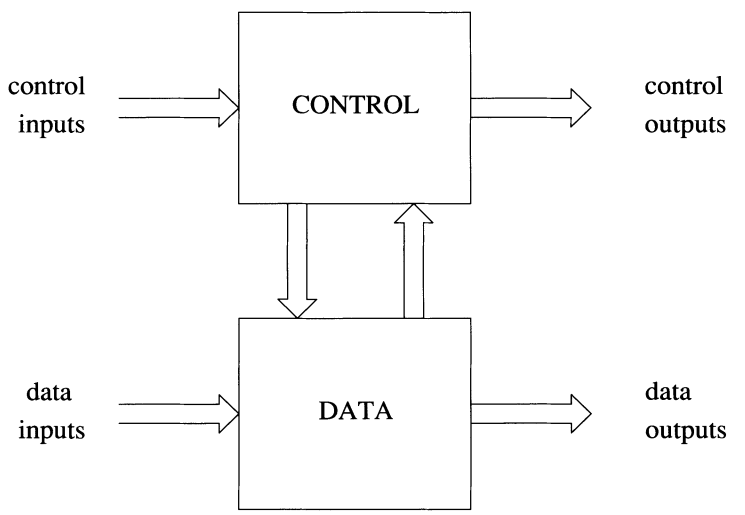



**Figure 6.86** Model with separation between data and control

We discuss the mechanism employed by SCIRTSS in searching for an input sequence to propagate an error stored in a state variable to a PO. From the current state (of both the good and the faulty circuit), SCIRTSS simulates the RTL description to compute the next states for a set of input vectors. All possible combinations of control inputs are applied, while data inputs are either user-specified or randomly generated. This strategy assumes that, compared to the data inputs, the control inputs have a stronger influence in determining the next state of the circuit, and their number is much smaller.

A state for which the set of next states has been computed is said to be *expanded.* Based on the results of RTL simulation, SCIRTSS determines whether the desired goal (propagating an error to a PO) has been achieved in one of the newly generated states. If not, one unexpanded state is selected and the expansion process repeats.

Figure 6.87 illustrates the first two steps of such a search process. The expanded states are 0 and 2. The selection of an unexpanded state is based on *heuristic functions* whose purpose is to guide the search on the path most likely to succeed. This type of search is patterned after techniques commonly used in artificial intelligence. The heuristic value $H_n$ associated with a state $n$ is given by the expression

$$H_n = G_n + w\ F_n$$

where

- $G_n$ is the length of the sequence leading to $n$ (in Figure 6.87, $G_1 = G_2 = \cdots = G_k = 1$, $G_{k+1} = G_{k+2} = 2$).
- $F_n$ is the main heuristic function that will be further discussed.
- $w$ is a weight that determines the extent to which the search is to be directed by $F_n$.

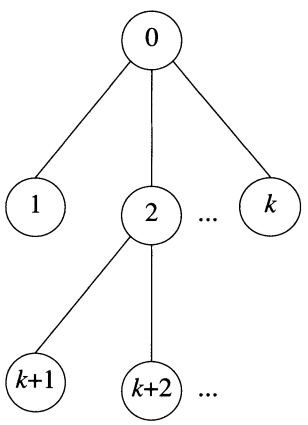



**Figure 6.87** SCIRTSS search process

The unexpanded state with the smallest $H_n$ is selected for expansion. Note that for $w = 0$ the search degenerates to breadth-first search. In general, $F_n$ is a linear combination of different heuristics

$$F_n = \sum_i w_i\ F_{ni}$$

where $w_i$ is the weight of the heuristic $F_{ni}$.

A useful heuristic in error-propagation problems is the *fault proliferation function*, expressed as

$$1 - \frac{\textit{the number of } F/Fs \textit{ with error values in state } n}{\textit{total number of } F/Fs}$$

Other heuristic functions [Huey 1979] are based on

- distance to a goal node;
- probabilities of reaching different states.

After generating a test sequence for a fault, SCIRTSS fault simulates the generated sequence on the low-level model. Then it discards all the detected faults. The results of the fault simulation are also used to select opportunistically the next target fault among the faults whose effects are propagated to state variables.

SCIRTSS has been successfully applied to generate tests for moderately complex sequential circuits. Its distinctive feature is the use of an RTL model as a basis for heuristic search procedures. Its performance is strongly influenced by the sophistication of the user, who has to provide heuristic weights and some data vectors.

### 6.3.4 Random Test Generation

RTG for sequential circuits is more complicated than for combinational circuits because of the following problems:

- Random sequences may fail to properly initialize a circuit that requires a specific initialization sequence.
- Some control inputs, such as clock and reset lines, have much more influence on the behavior of the circuit than other inputs. Allowing these inputs to change randomly as the other ones do, may preclude generating any useful sequences.
- The evaluation of a vector cannot be based only on the number of new faults it detects. A vector that does not detect any new faults, but brings the circuit into a state from which new faults are likely to be detected, should be considered useful. Also a vector that may cause races or oscillations should be rejected.

Because of the first two problems, a *semirandom* process is often used, in which pseudorandom stimuli are combined with user-specified stimuli [Schuler *et al.* 1975]. For example, the user may provide initialization sequences and deterministic stimuli for certain inputs (e.g., clock lines). The user may also specify inputs (such as reset lines) that should change infrequently and define their relative rate of change. In addition, the user is allowed to define the signal probabilities of certain inputs.

Having nonuniform signal probabilities is especially useful for control inputs. An adaptive procedure that dynamically changes the signal probabilities of control inputs is described in [Timoc *et al.* 1983].

To determine the next vector to be appended to the random sequence, the RTG method described in [Breuer 1971] generates and evaluates $Q$ candidate random vectors. Each vector $t$ is evaluated according to the function

$$v(t) = a\ v_1 + b(v_2 - v_3)$$

where

- $v_1$ is the number of new faults detected by $t$.
- $v_2$ is the number of new faults whose effects are propagated to state variables by $t$ (but are not detected by $t$).
- $v_3$ is the number of faults whose effects were propagated to state variables before the application of $t$, but were "lost" by $t$ (and have not yet been detected).
- $a$ and $b$ are weight factors.

Thus the value $v(t)$ of a vector $t$ is also influenced by its potential for future detections. At every step the vector with the largest value is selected (this represents a "local" optimization technique).

The more candidate vectors are evaluated, the greater is the likelihood of selecting a better one. But then more time is spent in evaluating more candidates. This trade-off can be controlled by an adaptive method that monitors the "profit" obtained by increasing $Q$ and increments $Q$ as long as the profit is increasing [Breuer 1971].

## 6.4 Concluding Remarks

Efficient TG for large circuits is an important practical problem. Some of the new research directions in this area involve hardware support and Artificial Intelligence (AI) techniques.

*Hardware support for TG* can be achieved by a special-purpose architecture using pipelining [Abramovici and Menon 1983] or by a general-purpose multiprocessing architecture [Motohara *et al.* 1986, Chandra and Patel 1988, Patil and Banerjee 1989]. The latter can take advantage of the available parallelism in several ways:

- *concurrent fault processing*, where each processor executes the same test generation job for a subset of faults;
- *concurrent decision processing*, where different processors simultaneously explore different branches from the same node in a decision tree;
- *concurrent guidance processing*, where different processors target the same fault using different cost functions for guidance.

*AI techniques* are applied to TG in an attempt to emulate features of the reasoning process of expert test engineers when generating tests. In contrast with an ATG algorithm, which is oblivious to the functionality of the circuit and looks only at a narrow portion of its structure at a time, a test engineer relies heavily on high-level knowledge about the intended behavior of the circuit and of its components, and uses a global view of its hierarchical structure. Another difference is the ability of the test engineer to recognize a family of similar problems and to take advantage of their similarity by providing a common solution mechanism; for example, the same control sequence can be repeated with different data patterns to set a register to different values. In contrast, an ATG algorithm would repeatedly regenerate the control sequence for every required data pattern.

HITEST [Robinson 1983, Bending 1984] is a knowledge-based system that combines algorithmic procedures with user-provided knowledge. A combinational test generator (based on PODEM) is used to generate a test in one time frame. For state-justification

and error-propagation problems, HITEST relies on user-provided solutions. Such solutions are often obvious for engineers but may be difficult to derive by a conventional test generator. For example, the knowledge specifying how to bring an up-down counter into a desired state would (1) indicate values needed for passive inputs (such as ENABLE, RESET), (2) determine, by comparing the desired and the current states, the number of clock pulses to be applied and the necessary setting for the UP/DOWN control input. The knowledge is stored and maintained using AI techniques.

Unlike an ATG algorithm, which explores the space of possible operations of a circuit, a test engineer searches the more restricted space of intended operations, i.e., the operations the circuit was designed to perform. Similarly, the system described by Shirley *et al.* [1987] uses knowledge about the intended behavior of the circuit to reduce the search domain. Rather than being provided by the user, the knowledge is acquired by analyzing traces obtained during a symbolic simulation of the circuit. In addition to the constant values propagated by a conventional simulator, a symbolic simulator also propagates variables, and its results are expressions describing the behavior of the circuit for all the possible data represented by variables. For example, if $X$ and $Y$ are variables assigned to the $A$ and $B$ inputs of an adder, its output will be set to $X+Y$, and if $X=0$, the output will be $Y$. During test generation, the desired objectives are matched with the simulation traces. If the objective for the adder is to propagate an error from the $B$ input, the solution $A=0$ will be found by matching with the simulation traces, which also recorded the events that had set $A=0$; these events are retrieved and reused as part of the generated test.

Other applications of AI techniques to TG problems are described in [Krishnamurthy 1987, Singh 1987].

TG techniques have other applications beyond TG. Sometimes one has two different combinational circuits supposed to implement the same function. For example, *C1* is a manual design and *C2* is a circuit automatically synthesized from the same specifications. Or *C1* is an existing circuit and *C2* is a redesign of *C1* using a different technology. Of course, we can simulate *C1* and *C2* with the same stimuli and compare the results, but a simulation-based process usually cannot provide a complete check. *Logic verification* can automatically compare the two circuits and either prove that they implement the same function or generate vectors that cause different responses. The basic mechanism [Roth 1977] is to combine *C1* and *C2* to create a composite circuit *M* (see Figure 6.88), then to use a line-justification algorithm to try to set each PO of *M* in turn to 1. If the line justification succeeds, then the generated vector sets two corresponding POs in *C1* and *C2* to different values; otherwise no such vector exists and the two POs have the same function.

Abadir *et al.* [1988] have shown that a complete test set for SSFs in a combinational circuit is also useful in *detecting* many typical *design errors*, such as missing or extra inverters, incorrect gate types, interchanged wires, and missing or extra wires.

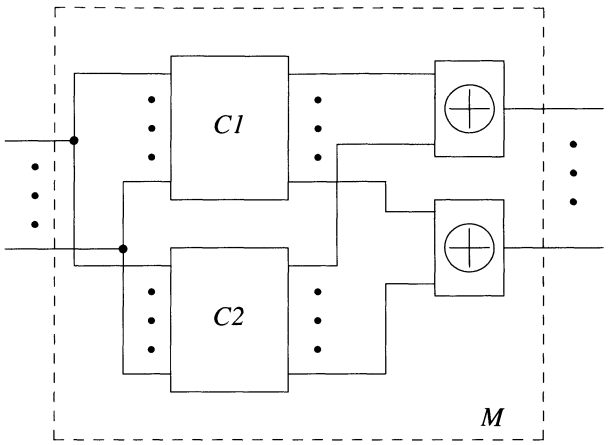



**Figure 6.88** Composite circuit for proving equivalence between *C1* and *C2*

# REFERENCES

[Abadir *et al.* 1988] M. S. Abadir, J. Ferguson, and T. E. Kirkland, "Logic Design Verification via Test Generation," *IEEE Trans. on Computer-Aided Design*, Vol. 7, No. 1, pp. 138-148, January, 1988.

[Abramovici and Menon 1983] M. Abramovici and P. R. Menon, "A Machine for Design Verification and Testing Problems," *Proc. Intn'l. Conf. on Computer-Aided Design*, pp. 27-29, September, 1983.

[Abramovici *et al.* 1986a] 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.

[Abramovici *et al.* 1986b] M. Abramovici, P. R. Menon, and D. T. Miller, "Checkpoint Faults Are Not Sufficient Target Faults for Test Generation," *IEEE Trans. on Computers*, Vol. C-35, No. 8, pp. 769-771, August, 1986.

[Abramovici and Miller 1989] M. Abramovici and D. T. Miller, "Are Random Vectors Useful in Test Generation?," *Proc. 1st European Test Conf.*, pp. 22-25, April, 1989.

[Abramovici *et al.* 1989] M. Abramovici, D. T. Miller, and R. K. Roy, "Dynamic Redundancy Identification in Automatic Test Generation," *Proc. Intn'l. Conf. on Computer-Aided Design*, November, 1989 (to appear).

[Agrawal and Agrawal 1972] V. D. Agrawal and P. Agrawal, "An Automatic Test Generation System for ILLIAC IV Logic Boards," *IEEE Trans. on Computers*, Vol. C-21, No. 9, pp. 1015-1017, September, 1972.

[Agrawal and Agrawal 1976] P. Agrawal and V. D. Agrawal, "On Monte Carlo Testing of Logic Tree Networks," *IEEE Trans. on Computers*, Vol. C-25, No. 6, pp. 664-667, June, 1976.

[Agrawal and Mercer 1982] V. D. Agrawal and M. R. Mercer, "Testability Measures — What Do They Tell Us?," *Digest of Papers 1982 Intn'l. Test Conf.*, pp. 391-396, November, 1982.

[Agrawal *et al.* 1988] V. D. Agrawal, H. Farhat, and S. Seth, "Test Generation by Fault Sampling," *Proc. Intn'l. Conf. on Computer Design*, pp. 58-61, October, 1988.

[Agrawal *et al.* 1989] V. D. Agrawal, K. Cheng, and P. Agrawal, "A Directed Search Method for Test Generation Using a Concurrent Simulator," *IEEE Trans. on Computer-Aided Design*, Vol. 8, No. 2, pp. 131-138, February, 1989.

[Airapetian and McDonald 1979] A. N. Airapetian and J. F. McDonald, "Improved Test Set Generation Algorithm for Combinational Logic Control," *Digest of Papers 9th Annual Intn'l. Symp. on Fault-Tolerant Computing*, pp. 133-136, June, 1979.

[Akers 1976] S. B. Akers, "A Logic System for Fault Test Generation," *IEEE Trans. on Computers*, Vol. C-25, No. 6, pp. 620-630, June, 1976.

[Batni and Kime 1976] R. P. Batni and C. R. Kime, "A Module-Level Testing Approach for Combinational Networks," *IEEE Trans. on Computers*, Vol. C-25, No. 6, pp. 594-604, June, 1976.

[Bellon *et al.* 1983] C. Bellon, C. Robach, and G. Saucier, "An Intelligent Assistant for Test Program Generation: The SUPERCAT System," *Proc. Intn'l. Conf. on Computer-Aided Design*, pp. 32-33, September, 1983.

[Bending 1984] M. T. Bending, "Hitest: A Knowledge-Based Test Generation System," *IEEE Design & Test of Computers*, Vol. 1, No. 2, pp. 83-92, May, 1984.

[Benmehrez and McDonald 1983] C. Benmehrez and J. F. McDonald, "Measured Performance of a Programmed Implementation of the Subscripted D-Algorithm," *Proc. 20th Design Automation Conf.*, pp. 308-315, June, 1983.

[Bhattacharya and Hayes 1985] D. Bhattacharya and J. P. Hayes, "High-Level Test Generation Using Bus Faults," *Digest of Papers 15th Annual Intn'l. Symp. on Fault-Tolerant Computing,* pp. 65-70, June, 1985.

[Breuer 1983] M. A. Breuer, "Test Generation Models for Busses and Tri-State Drivers," *Proc. IEEE ATPG Workshop,* pp. 53-58, March, 1983.

[Breuer 1971] M. A. Breuer, "A Random and an Algorithmic Technique for Fault Detection Test Generation for Sequential Circuits," *IEEE Trans. on Computers,* Vol. C-20, No. 11, pp. 1364-1370, November, 1971.

[Breuer 1974] M. A. Breuer, "The Effects of Races, Delays, and Delay Faults on Test Generation," *IEEE Trans. on Computers*, Vol. C-23, No. 10, pp. 1078-1092, October, 1974.

[Breuer 1978] M. A. Breuer, "New Concepts in Automated Testing of Digital Circuits," *Proc. EEC Symp. on CAD of Digital Electronic Circuits and Systems*, pp. 69-92, North-Holland Publishing Co., November, 1978.

[Breuer and Friedman 1976] M. A. Breuer and A. D. Friedman, *Diagnosis & Reliable Design of Digital Systems*, Computer Science Press, Rockville, Maryland, 1976.

[Breuer and Friedman 1980] M. A. Breuer and A. D. Friedman, "Functional Level Primitives in Test Generation," *IEEE Trans. on Computers*, Vol. C-29, No. 3, pp. 223-235, March, 1980.

[Breuer and Harrison 1974] M. A. Breuer and L. M. Harrison, "Procedures for Eliminating Static and Dynamic Hazards in Test Generation," *IEEE Trans. on Computers*, Vol. C-23, No. 10, pp. 1069-1078, October, 1974.

[Cha *et al.* 1978] C. W. Cha, W. E. Donath, and F. Ozguner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits," *IEEE Trans. on Computers*, Vol. C-27, No. 3, pp. 193-200, March, 1978.

[Chandra and Patel 1987] S. J. Chandra and J. H. Patel, "A Hierarchical Approach to Test Vector Generation," *Proc. 24th Design Automation Conf.*, pp. 495-501, June, 1987.

[Chandra and Patel 1988] S. J. Chandra and J. H. Patel, "Test Generation in a Parallel Processing Environment," *Proc. Intn'l. Conf. on Computer Design,* pp. 11-14, October, 1988.

[Chandra and Patel 1989] S. J. Chandra and J. H. Patel, "Experimental Evaluation of Testability Measures for Test Generation," *IEEE Trans. on Computer-Aided Design*, Vol. 8, No. 1, pp. 93-98, January, 1989.

[Cheng and Agrawal 1989] K.-T. Cheng and V. D. Agrawal, "An Economical Scan Design for Sequential Logic Test Generation," *Digest of Papers 19th Intn'l. Symp. on Fault-Tolerant Computing*, pp. 28-35, June, 1989.

[Cheng and Chakraborty 1989] W.-T. Cheng and T. Chakraborty, "Gentest — An Automatic Test-Generation System for Sequential Circuits," *Computer*, Vol. 22, No. 4, pp. 43-49, April, 1989.

[David and Blanchet 1976] R. David and G. Blanchet, "About Random Fault Detection of Combinational Networks," *IEEE Trans. on Computers*, Vol. C-25, No. 6, pp. 659-664, June, 1976.

[Eichelberger and Lindbloom 1983] E. E. Eichelberger and E. Lindbloom, "Random-Pattern Coverage Enhancement and Diagnosis for LSSD Logic Self-Test," *IBM Journal of Research and Development*, Vol. 27, No. 3, pp. 265-272, March, 1983

[Fujiwara and Shimono 1983] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Trans. on Computers*, Vol. C-32, No. 12, pp. 1137-1144, December, 1983.

[Goel 1978] P. Goel, "RAPS Test Pattern Generator," *IBM Technical Disclosure Bulletin*, Vol. 21, No. 7, pp. 2787-2791, December, 1978.

[Goel 1981] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. on Computers*, Vol C-30, No. 3, pp. 215-222, March, 1981.

[Goel and Rosales 1979] P. Goel and B. C. Rosales, "Test Generation & Dynamic Compaction of Tests," *Digest of Papers 1979 Test Conf.*, pp. 189-192, October, 1979.

[Goel and Rosales 1980] P. Goel and B. C. Rosales, "Dynamic Test Compaction with Fault Selection Using Sensitizable Path Tracing," *IBM Technical Disclosure Bulletin*, Vol. 23, No. 5, pp. 1954-1957, October, 1980.

[Goel and Rosales 1981] P. Goel and B. C. Rosales, "PODEM-X: An Automatic Test Generation System for VLSI Logic Structures," *Proc. 18th Design Automation Conf.*, pp. 260-268, June, 1981.

[Goldstein 1979] L. H. Goldstein, "Controllability/Observability Analysis of Digital Circuits," *IEEE Trans. on Circuits and Systems*, Vol. CAS-26, No. 9, pp. 685-693, September, 1979.

[Heap and Rogers 1989] M. A. Heap and W. A. Rogers, "Generating Single-Stuck-Fault Coverage from a Collapsed-Fault Set," *Computer*, Vol. 22, No. 4, pp. 51-57, April, 1989

[Hill and Huey 1977] F. J. Hill and B. Huey, "SCIRTSS: A Search System for Sequential Circuit Test Sequences," *IEEE Trans. on Computers*, Vol. C-26, No. 5, pp. 490-502, May, 1977.

[Huey 1979] B. M. Huey, "Heuristic Weighting Functions for Guiding Test Generation Searches," *Journal of Design Automation & Fault-Tolerant Computing*, Vol. 3, pp. 21-39, January, 1979.

[Itazaki and Kinoshita 1986] N. Itazaki and K. Kinoshita, "Test Pattern Generation for Circuits with Three-state Modules by Improved Z-algorithm," *Proc. Intn'l. Test Conf.*, pp. 105-108, September, 1986.

[Ivanov and Agarwal 1988] A. Ivanov and V. K. Agarwal, "Dynamic Testability Measures for ATPG," *IEEE Trans. on Computer-Aided Design*, Vol. 7, No. 5, pp. 598-608, May, 1988.

[Kirkland and Mercer 1987] T. Kirkland and M. R. Mercer, "A Topological Search Algorithm for ATPG," *Proc. 24th Design Automation Conf.*, pp. 502-508, June, 1987.

[Krishnamurthy and Sheng 1985] B. Krishnamurthy and R. L. Sheng, "A New Approach to the Use of Testability Analysis in Test Generation," *Proc. Intn'l. Test Conf.*, pp. 769-778, November, 1985.

[Krishnamurthy 1987] B. Krishnamurthy, "Hierarchical Test Generation: Can AI Help?," *Proc. Intn'l. Test Conf.*, pp. 694-700, September, 1987.

[Krishnamurthy and Tollis 1989] B. Krishnamurthy and I. G. Tollis, "Improved Techniques for Estimating Signal Probabilities," *IEEE Trans. on Computers*, Vol. 38, No. 7, pp. 1041-1045, July, 1989.

[Levendel and Menon 1982] Y. H. Levendel and P. R. Menon, "Test Generation Algorithms for Computer Hardware Description Languages," *IEEE Trans. on Computers*, Vol. C-31, No. 7, pp. 577-588, July, 1982.

[Lioy 1988] A. Lioy, "Adaptive Backtrace and Dynamic Partitioning Enhance a ATPG," *Proc. Intn'l. Conf. on Computer Design*, pp. 62-65, October, 1988.

[Lioy *et al.* 1989] A. Lioy, P. L. Montessoro, and S. Gai, "A Complexity Analysis of Sequential ATPG," *Proc. Intn'l. Symp. on Circuits and Systems*, pp. 1946-1949, May, 1989.

[Lisanke *et al.* 1987] R. Lisanke, F. Brglez, A. J. de Geus, and D. Gregory, "Testability-Driven Random Test-Pattern Generation," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-6, No. 6, November, 1987.

[Ma *et al.* 1988] H.-K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli, "Test Generation for Sequential Circuits," *IEEE Trans. on Computer-Aided Design*, Vol. 7, No. 10, pp. 1081-1093, October, 1988.

[Mallela and Wu 1985] S. Mallela and S. Wu, "A Sequential Circuit Test Generation System," *Proc. Intn'l. Test Conf.*, pp. 57-61, November, 1985.

[Marlett 1978] R. A. Marlett, "EBT: A Comprehensive Test Generation Technique for Highly Sequential Circuits," *Proc. 15th Design Automation Conf.*, pp. 335-339, June, 1978.

[Menon and Harihara 1989] P. R. Menon and M. R. Harihara, "Identification of Undetectable Faults in Combination Circuits," *Proc. Intn'l. Conf. on Computer Design*, October, 1989 (to appear).

[Miczo 1983] A. Miczo, "The Sequential ATPG: A Theoretical Limit," *Proc. Intn'l. Test Conf.*, pp. 143-147, October, 1983.

[Motohara *et al.* 1986] A. Motohara, K. Nishimura, H. Fujiwara, and I. Shirakawa, "A Parallel Scheme for Test-Pattern Generation," *Proc. Intn'l. Conf. on Computer-Aided Design*, pp. 156-159, November, 1986.

[Murray and Hayes 1988] B. T. Murray and J. P. Hayes, "Hierarchical Test Generation Using Precomputed Tests for Modules," *Proc. Intn'l. Test Conf.*, pp. 221-229, September, 1988.

[Muth 1976] P. Muth, "A Nine-Valued Circuit Model for Test Generation," *IEEE Trans. on Computers*, Vol. C-25, No. 6, pp. 630-636, June, 1976.

[Norrod 1989] F. E. Norrod, "An Automatic Test Generation Algorithm for Hardware Description Languages," *Proc. 26th Design Automation Conf.*, pp. 429-434, June, 1989.

[Ogihara *et al.* 1988] T. Ogihara, S. Saruyama, and S. Murai, "Test Generation for Sequential Circuits Using Individual Initial Value Propagation," *Proc. Intn'l. Conf. on Computer-Aided Design*, pp. 424-427, November, 1988.

[Parker 1976] K. P. Parker, "Adaptive Random Test Generation," *Journal of Design Automation and Fault-Tolerant Computing*, Vol. 1, No. 1, pp. 62-83, October, 1976.

[Parker and McCluskey 1975] K. P. Parker and E. J. McCluskey, "Probabilistic Treatment of General Combinational Networks," *IEEE Trans. on Computers*, Vol. C-24, No. 6, pp. 668-670, June, 1975.

[Patil and Banerjee 1989] S. Patil and P. Banerjee, "A Parallel Branch and Bound Algorithm for Test Generation," *Proc. 26th Design Automation Conf.*, pp. 339-344, June, 1989.

[Putzolu and Roth 1971] G. R. Putzolu and T. P. Roth, "A Heuristic Algorithm for the Testing of Asynchronous Circuits," *IEEE Trans. on Computers*, Vol. C-20, No. 6, pp. 639-647, June, 1971.

[Ratiu *et al.* 1982] I. M. Ratiu, A. Sangiovanni-Vincentelli, and D. O. Pederson "VICTOR: A Fast VLSI Testability Analysis Program," *Digest of Papers 1982 Intn'l. Test Conf.*, pp. 397-401, November, 1982.

[Renous *et al.*] R. Renous, G. M. Silberman, and I. Spillinger, "Whistle — A Workbench for Test Development of Library-Based Designs," *Computer,* Vol. 22, No. 4, pp. 27-41, April, 1989.

[Robinson 1983] G. D. Robinson, "HITEST-Intelligent Test Generation," *Proc. Intn'l. Test Conf.*, pp. 311-323, October, 1983.

[Roth 1966] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, Vol. 10, No. 4, pp. 278-291, July, 1966.

[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 Electronic Computers*, Vol. EC-16, No. 10, pp. 567-579, October, 1967.

[Roth 1977] J. P. Roth, "Hardware Verification," *IEEE Trans. on Computers*, Vol. C-26, No. 12, pp. 1292-1294, December, 1977.

[Roth 1980] J. P. Roth, *Computer Logic, Testing and Verification*, Computer Science Press, Rockville, Maryland, 1980.

[Rutman 1972] R. A. Rutman, "Fault Detection Test Generation for Sequential Logic by Heuristic Tree Search," *IEEE Computer Group Repository*, Paper No. R-72-187, 1972.

[Savir and Bardell 1984] J. Savir and P. H. Bardell, "On Random Pattern Test Length," *IEEE Trans. on Computers*, Vol. C-33, No. 6, pp. 467-474, June, 1984.

[Savir *et al.* 1984] J. Savir, G. S. Ditlow, and P. H. Bardell, "Random Pattern Testability," *IEEE Trans. on Computers*, Vol. C-33, No. 1, pp. 79-90, January, 1984.

[Schneider 1967] R. R. Schneider, "On the Necessity to Examine D-Chains in Diagnostic Test Generation," *IBM Journal of Research and Development*, Vol. 11, No. 1, p. 114, January, 1967.

[Schnurmann *et al.* 1975] H. D. Schnurmann, E. Lindbloom, and R. G. Carpenter, "The Weighted Random Test-Pattern Generator," *IEEE Trans. on Computers*, Vol. C-24, No. 7, pp. 695-700, July, 1975.

[Schuler *et al.* 1975] D. M. Schuler, E. G. Ulrich, T. E. Baker, and S. P. Bryant, "Random Test Generation Using Concurrent Fault Simulation," *Proc. 12th Design Automation Conf.*, pp. 261-267, June, 1975.

[Schulz *et al.* 1988] M. H. Schulz, E. Trischler, and T. M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Trans. on Computer-Aided Design*, Vol. 7, No. 1, pp. 126-137, January, 1988.

[Schulz and Auth 1989] M. Schulz and E. Auth, "Improved Deterministic Test Pattern Generation with Applications to Redundancy Identification," *IEEE Trans. on Computer-Aided Design*, Vol. 8, No. 7, pp. 811-816, July, 1989.

[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 *et al.* 1985] S. C. Seth, L. Pan, and V. D. Agrawal, "PREDICT — Probabilistic Estimation of Digital Circuit Testability," *Digest of Papers 15th Annual Intn'l. Symp. on Fault-Tolerant Computing*, pp. 220-225, June, 1985.

[Shedletsky 1977] J. J. Shedletsky, "Random Testing: Practicality vs. Verified Effectiveness," *Proc. 7th Annual Intn'l. Conf. on Fault-Tolerant Computing*, pp. 175-179, June, 1977.

[Shirley *et al.* 1987] M. Shirley, P. Wu, R. Davis, and G. Robinson, "A Synergistic Combination of Test Generation and Design for Testability," *Proc. Intn'l. Test Conf.*, pp. 701-711, September, 1987.

[Shteingart *et al.* 1985] S. Shteingart, A. W. Nagle, and J. Grason, "RTG: Automatic Register Level Test Generator," *Proc. 22nd Design Automation Conf.*, pp. 803-807, June, 1985.

[Silberman and Spillinger 1988] G. M. Silberman and I. Spillinger, "G-RIDDLE: A Formal Analysis of Logic Designs Conducive to the Acceleration of Backtracking," *Proc. Intn'l. Test Conf.*, pp. 764-772, September, 1988.

[Singh 1987] N. Singh, *An Artificial Intelligence Approach to Test Generation*, Kluwer Academic Publishers, Norwell, Massachusetts, 1987.

[Snethen 1977] T. J. Snethen, "Simulator-Oriented Fault Test Generator," *Proc. 14th Design Automation Conf.*, pp. 88-93, June, 1977.

[Somenzi *et al.* 1985] F. Somenzi, S. Gai, M. Mezzalama, and P. Prinetto, "Testing Strategy and Technique for Macro-Based Circuits," *IEEE Trans. on Computers*, Vol. C-34, No. 1, pp. 85-90, January, 1985.

[Thomas 1971] J. J. Thomas, "Automated Diagnostic Test Programs for Digital Networks," *Computer Design*, pp. 63-67, August, 1971.

[Timoc *et al.* 1983] C. Timoc, F. Stott, K. Wickman and L. Hess, "Adaptive Probabilistic Testing of a Microprocessor," *Proc. Intn'l. Conf. on Computer-Aided Design*, pp. 71-72, September, 1983.

[Wagner *et al.* 1987] K. D. Wagner, C. K. Chin, and E. J. McCluskey, "Pseudorandom Testing," *IEEE Trans. on Computers*, Vol. C-36, No. 3, pp. 332-343, March, 1987.

[Waicukauski *et al.* 1989] J. A. Waicukauski, E. Lindbloom, E. B. Eichelberger, and O. P. Forlenza, "WRP: A Method for Generating Weighted Random Test Patterns," *IBM Journal of Research and Development*, Vol. 33, No. 2, pp. 149-161, March, 1989.

[Wang 1975] D. T. Wang, "An Algorithm for the Generation of Test Sets for Combinational Logic Networks," *IEEE Trans. on Computers*, Vol. C-24, No. 7, pp. 742-746, July, 1975.

[Wang and Wei 1986] J.-C. Wang and D.-Z. Wei, "A New Testability Measure for Digital Circuits," *Proc. Intn'l. Test Conf.*, pp. 506-512, September, 1986.

[Wunderlich 1987] H.-J. Wunderlich, "On Computing Optimized Input Probabilities for Random Tests," *Proc. 24th Design Automation Conf.*, pp. 392-398, June, 1987.

# PROBLEMS

**6.1** For the circuit of Figure 6.89, generate a test for the fault $g$ *s-a*-1. Determine all the other faults detected by this test.

**6.2** Modify the TG algorithm for fanout-free circuits (given in Figures 6.3, 6.4, and 6.5) so that it tries to generate a test that will detect as many faults as possible in addition to the target fault.

**6.3** Use only implications to show that the fault $f$ *s-a*-0 in the circuit of Figure 6.12(a) is undetectable.

**6.4** Construct the truth table of an XOR function of two inputs using the five logic values 0, 1, $x$, $D$, and $\overline{D}$.

**6.5** Can a gate on the *D-frontier* have both a $D$ and a $\overline{D}$ among its input values?

**6.6** For the circuit of Figure 6.90, perform all possible implications starting from the given values.

**6.7** For the circuit of Figure 6.91, perform all possible implications starting from the given values.

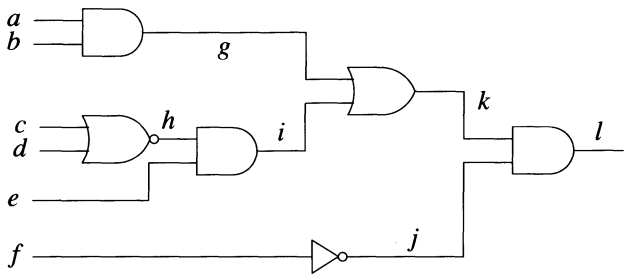



**Figure 6.89**

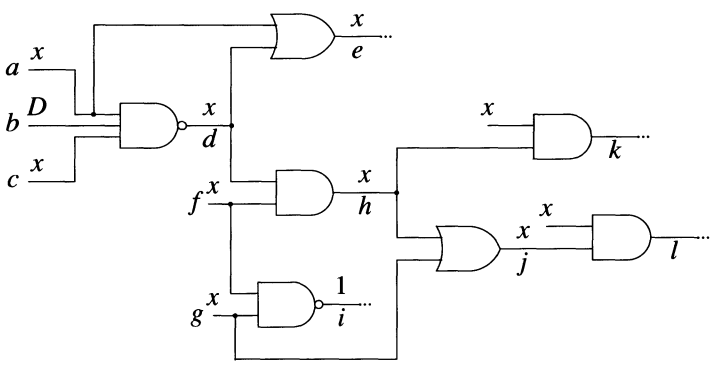



**Figure 6.90**

**6.8** Consider the circuit and the values shown in Figure 6.92. Let us assume that trying to justify $d$=0 via $a$=0 has lead to an inconsistency. Perform all the implications resulting from reversing the incorrect decision.

**6.9** Define the *D-frontier* for the 9-V algorithm.

**6.10** Using the five logic values of the $D$-algorithm, the consistency check works as follows. Let $v'$ be the value to be assigned to a line and $v$ be its current value. Then $v'$ and $v$ are consistent if $v$=$x$ or $v$=$v'$. Formulate a consistency check for the 9-V algorithm.

**6.11** Outline a TG algorithm based on single-path sensitization. Apply the algorithm to the problem from Example 6.6.

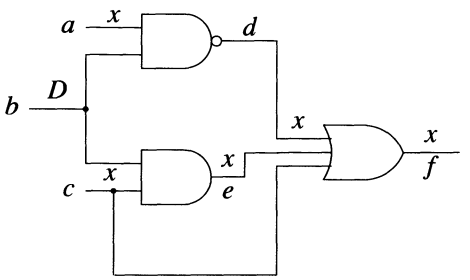



**Figure 6.91**

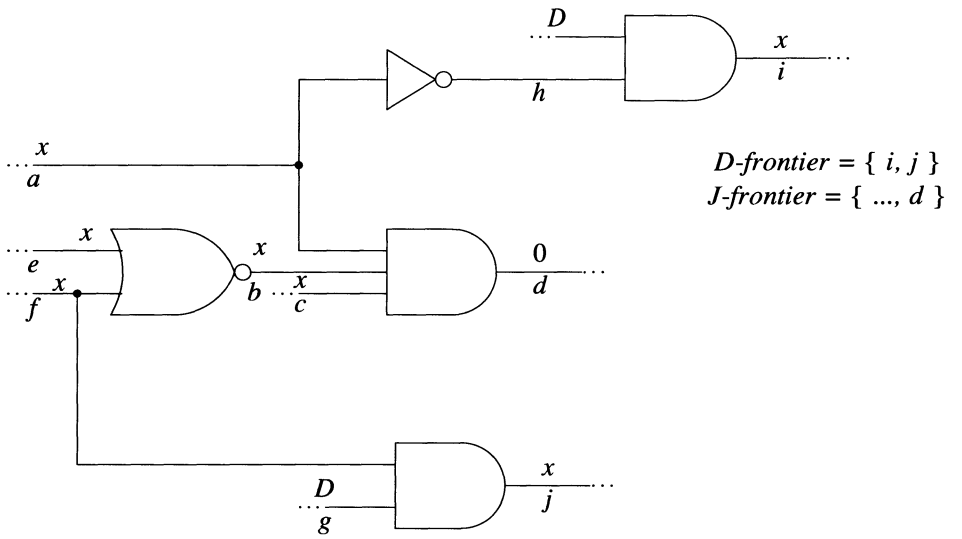



**Figure 6.92**

**6.12** Apply PODEM to the problem from Example 6.4. The *D*-algorithm can determine that a fault is undetectable without backtracking (i.e., only by implications). Is the same true for PODEM?

**6.13** Apply FAN to the problem from Example 6.4.

**6.14** Discuss the applicability of formula (6.3) to the circuit in Figure 6.93.

**6.15** Extend formulas (6.1) ... (6.7) for different gate types.

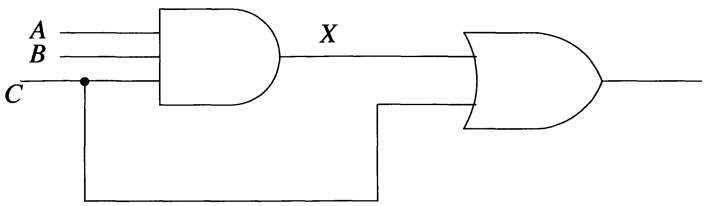



**Figure 6.93**

**6.16** Show that in a fanout-free circuit, if we start by setting *C0* and *C1* of every PI to 1, and then recursively compute controllability costs, *C0(l)* and *C1(l)* represent the minimum number of PIs we have to assign binary values to set $l$ to 0 and 1, respectively.

**6.17** Compute controllability and observability costs for every line in the circuit in Figure 6.7.

a. Use the (generalized) formulas (6.1) to (6.4).

b. Repeat, by introducing fanout-based correction terms in controllability formulas.

c. Use the costs computed in a. to guide PODEM in generating a test for the fault $G_1$ *s-a-*1.

d. Repeat, using the costs computed in b.

**6.18** Prove that the test set generated by using the procedure *CPTGFF* (Figure 6.46) for a fanout-free circuit is complete for SSFs.

**6.19**

a. Modify the procedure *CPTGFF* (Figure 6.46) for use in multiple-output circuits without reconvergent fanout.

b. Apply critical-path TG for the circuit in Figure 6.94.

**6.20** Can the detection probability of a fault in a combinational circuit be 1/100?

**6.21** Show that the expected fault coverage of a random test sequence of length $N$ is greater than

a. its testing quality $t_N$;

b. its detection quality $d_N$.

**6.22**

a. Compute signal probabilities in the circuit in Figure 6.94.

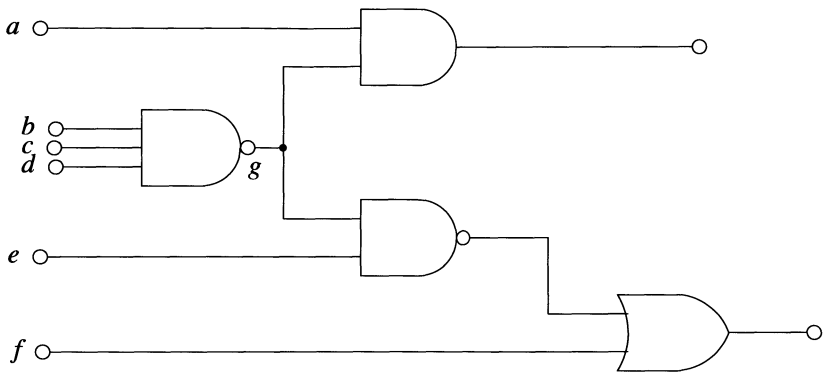



**Figure 6.94**

b. Compute the detection probability for the fault $g$ $s$-$a$-0.

**6.23** Consider a fanout-free circuit with $L$ levels composed of NAND gates. Each gate has $n$ inputs. Let $p_0$ be the signal probability of every PI. Because of symmetry, all signals at level $l$ have the same signal probability $p_l$.

a. Show that

$$p_l = 1-(p_{l-1})^n$$

b. Show that the smallest detection probability of a fault is

$$d_{\min} = r\prod_{k=0}^{L-1}(p_k)^{n-1}$$

$$\text{where } r = min\{p_0, 1-p_0\}$$

c. For $L$=2 and $n$=2, determine the value $p_0$ that maximizes $d_{\min}$.

**6.24** Use formula (6.25) to compute $p_z$ for

a. $Z=A.A$
b. $Z=A.\overline{A}$
c. $Z=A+A$
d. $Z=A+\overline{A}$

**6.25** Let $Z$ be the output of an OR gate whose inputs, $X$ and $Y$, are such that $X.Y=0$ (i.e., are never simultaneously 1). Show that

$$p_Z = p_A+p_B$$

**6.26** Let $Z$ be the output of a combinational circuit realizing the function $f$. Let us express $f$ as a sum of minterms, i.e., $f = \sum_{i=1}^{k} m_i$.

a. If $p_i = P_r(m_i = 1)$, show that

$$p_Z = \sum_{i=1}^{k} p_i$$

b. For the circuit in Figure 6.95, compute $p_Z$ as a function of $p_A$,$p_B$, and $p_C$.

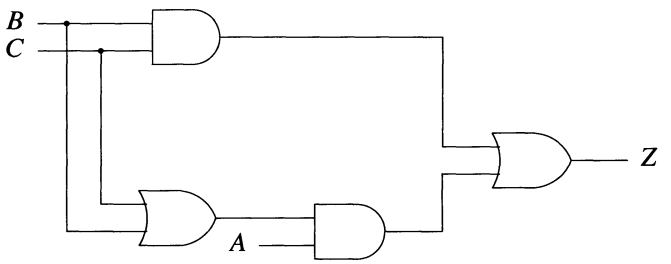



**Figure 6.95**

c. Let $p_A = p_B = p_C = q$. Plot $p_Z$ as a function of $q$.

**6.27** Determine lower bounds on the detection probability of every checkpoint fault for the circuit in Figure 6.48.

**6.28** Determine the lower bound on the detection probability of the fault $Y$ $s$-$a$-0 in the circuit in Figure 6.96.

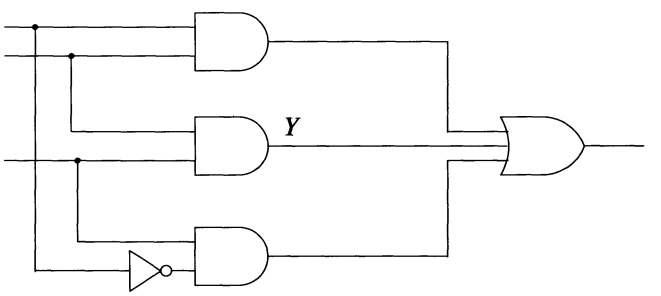



**Figure 6.96**

**6.29**

a. Show that the gate $G$ selected in the second phase of the procedure RAPS (Figure 6.57) has at least one input with controlling value.

b. When $G$ has more than one input with controlling value, do we gain anything by trying to justify noncontrolling values on the other inputs?

**6.30** Show that if two input vectors (of the same circuit) are compatible, then their corresponding output vectors are also compatible.

**6.31** Determine the primitive cubes and the propagation $D$-cubes for an exclusive-OR module with two inputs.

**6.32** Derive a test sequence for the fault $Z$ $s$-$a$-0 in the circuit of Figure 6.75.

a. Assume an initial state $q_1$=$q_2$=1.

b. Assume an unknown initial state.

**6.33** Derive a self-initializing test sequence for the fault $J_1$ $s$-$a$-1 in the circuit of Figure 6.77.

**6.34** Try to use the TG procedure given in Figure 6.76 to generate a self-initializing test sequence for the fault $R$ $s$-$a$-1 in the circuit of Figure 6.97. Explain why the procedure fails.

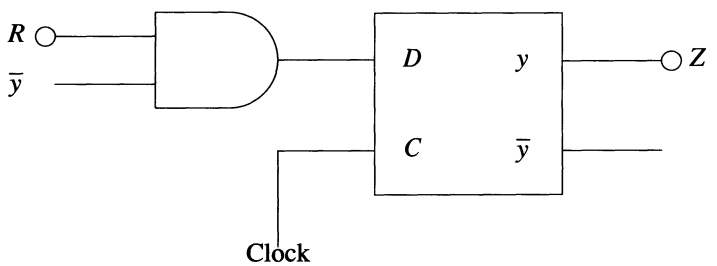



**Figure 6.97**

**6.35** For the shift register of Figure 6.82, determine the union algorithm executed for (*Reset*, *Clock*, $S_0$, $S_1$) = 1 ↑ $x$ 0.

**6.36** For the shift register of Figure 6.82, assume that $n$ = 4, (*Reset*, *Clock*, $S_0$, $S_1$) = 1 ↑ $x$ 1, **y** = 0 1 $x$ $x$ and **A** = $x$ $x$ $x$ $x$ $x$ $x$. Determine how to justify **Y** = 1 1 $x$ 0.