
SMART and FAST: Test Generation for VLSI Scan-Design Circuits

M. Abramovici, J.J. Kulikowski, P.R. Menon, and D.T. Miller
AT&T Information Systems

This article describes new concepts and algorithms used to generate tests for VLSI scan-design circuits. The new algorithms include:

- 1. a low-cost fault-independent algorithm (SMART),*
 - 2. a fault-oriented algorithm (FAST), and*
 - 3. an algorithm for dynamic test set compaction.*
- The fault-oriented algorithm is guided by new controllability/observability cost functions whose objective is to minimize the amount of search done in test generation.*

With the increasing use of scan-design techniques, test generation for large combinational circuits has become a topic of great practical significance. Although scan-design techniques transform sequential circuits into combinational ones during testing, test generation for large scan-design circuits remains a very expensive computational process. For example, using an IBM 370/168, CPU times of 23 hours for a circuit with 48,000 gates,¹ and of 45 hours for a circuit with 32,000 gates² have been reported (estimated equivalent times for an IBM 3081K are 7.2 and 14 hours, respectively).

In this article, we present a multifront attack on the classical test generation problem for single stuck-at faults. Our approach is based on several new concepts and algorithms, integrated in a system that significantly advances the state of the art in test generation. The system has been used to generate tests for gate arrays and large boards. Our results show that even for large scan-design circuits, tests can be generated with reasonable runtimes. For example, generating tests for a board containing 75,000 gates took 1.6 hours of CPU time on an IBM 3081K.

Our algorithms have been implemented in a design aids system, LAMP2, used within AT&T. The tasks of the LAMP2 Test Generation (LTG) system include

- auditing a circuit for compliance with scan-design testability rules,
- partitioning a scan-design circuit and extracting combinational blocks for test generation,
- generating tests,
- postprocessing the results according to the target testing environment (device testing, in-circuit board testing, or board testing)

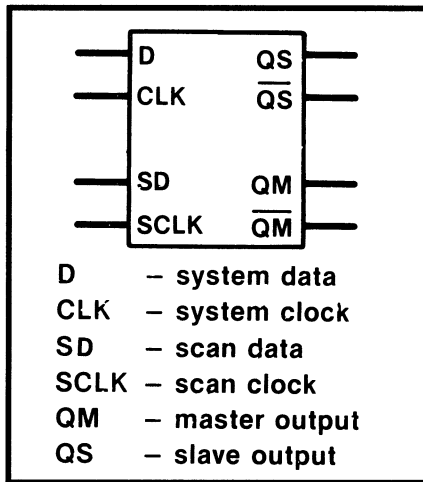


Figure 1. Basic SRL.

by a maintenance processor).

This article is organized as follows. First, we discuss types of circuits for which LTG is applicable and describe the preprocessing required to generate tests. We then outline LTG's two-phase structure, which couples a low-cost fault-independent algorithm (Phase 1) with a fault-oriented algorithm (Phase 2). Both phases closely interact with a fault simulator based on critical path tracing.³ Next we present the concepts on which this interaction is based, and describe the two phases in detail. Finally we present results and conclusions.

Circuits testable by LTG

LTG accepts a large class of scan designs, including both LSSD⁴ and Scan-Path⁵ circuits. Additional design flexibility is provided by a separate *test mode* that allows circuits to be automatically restructured for testing. Consequently, scan-design constraints need be satisfied only in test mode. This feature allows designers to use asynchronous loops that are broken during testing. Also, designers can combine in the same circuit scan-design and non-scan-design logic, which are separated in test mode. For example, clock generation logic (based on clock ANDing) can be included in the same chip with the logic driven by the clock.

The basic shift-register latch (SRL) used in designs accepted by LTG is a scanable dual-port master-slave flip-flop (see Figure 1). Because both the master and slave outputs can be used for system (normal mode) operation, the SRL generalizes both the LSSD SRL and the Scan-Path flip-flop. Although a circuit with master-

slave SRLs is not level-sensitive because of the potential race involved in the master-to-slave transfer, this problem is local and can be eliminated by careful design of the SRL. Compared to an LSSD double-latch design, a design using master-slave SRLs needs fewer independent system clocks (and hence has fewer clock skewing problems) and can operate at a higher speed. A more detailed discussion of the LTG scan-design rules is published elsewhere.⁶

Preprocessing

In addition to auditing the circuit for compliance with the LTG scan-design rules, the following operations are automatically performed before test generation:

- Removal of redundant gates and connections and identification of undetectable faults ("circuit clean").
- Model transformations needed for test generation.
- Circuit partitioning and extraction of combinational blocks for which tests will be generated.

Circuit cleaning. Gate array style designs consist of interconnections of predefined blocks. When blocks are only partially used, their unused inputs are tied to constant logic values within the chip, and their unused outputs are left unconnected. Signals with fixed values and signals without fanout can also result from techniques employed to reduce the number of gate array generic codes. One such technique employs different instances of the same code to perform slightly different functions under the control of "personality pins" tied to different constant logic values on the board. Another technique uses different subsets of output pins in different instances of the same code. The subcircuits feeding only lines without fanout become unnecessary and can be removed from the model. A consequence of having signals with fixed values is that many faults be-

come undetectable, yet they can be easily identified. These operations are performed by a "circuit clean" (CKTCLN) preprocessing task.

To identify undetectable faults, CKTCLN starts by simulating the circuit to propagate the fixed values as far as possible. For example, in the circuit of Figure 2, tying *A* to 0 results in *G* being set to 0. For every line *l* set to a binary value *v*, the fault *l-s-a-v* cannot be activated and therefore is undetectable. In Figure 2, *A-s-a-0* and *G-s-a-0* are undetectable. Now consider a gate with an input set to the controlling value of the gate (such as 0 for an AND), and let *l* be other input of the same gate. The effect of any fault on *l* or in the subcircuit feeding only *l* cannot propagate past *l*, and therefore is undetectable. In Figure 2, any fault on *B2*, *C*, and *F* is undetectable.

In addition, CKTCLN simplifies the model by performing a series of "local transformations," some of which are illustrated in Figure 3. The original configuration of Figure 3a (where *D* is a wired-AND) is commonly used to increase signal drive capability. First *B* and *C* are recognized as noninverting gates with single input and single fanout and are bypassed. Then the two connections between *A* and *D* are identified as redundant and one is removed. Next *D* is bypassed.

All transformations done by CKTCLN simplify the model without changing its function. The main benefit of CKTCLN is the speedup caused by eliminating undetectable faults that would otherwise consume a lot of time during test generation. In addition, the reduction in model size increases the size of the circuits that LTG can process and reduces its runtime.

Partitioning and model transformations. A scan-design circuit is partitioned into subcircuits that are separately tested. This partitioning is done during preprocessing and is based on system clocks. The partition determined by a system clock

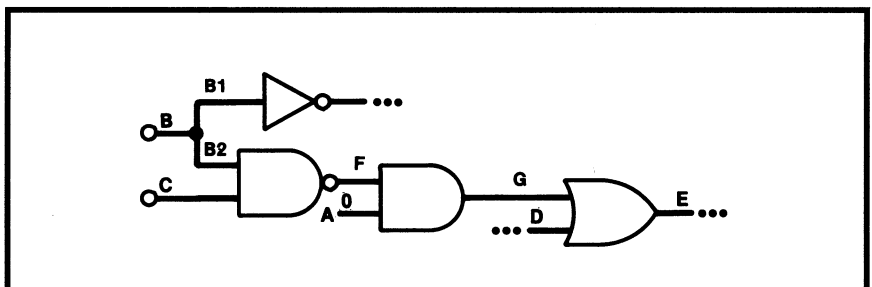


Figure 2. Fixed value causing undetectable faults.

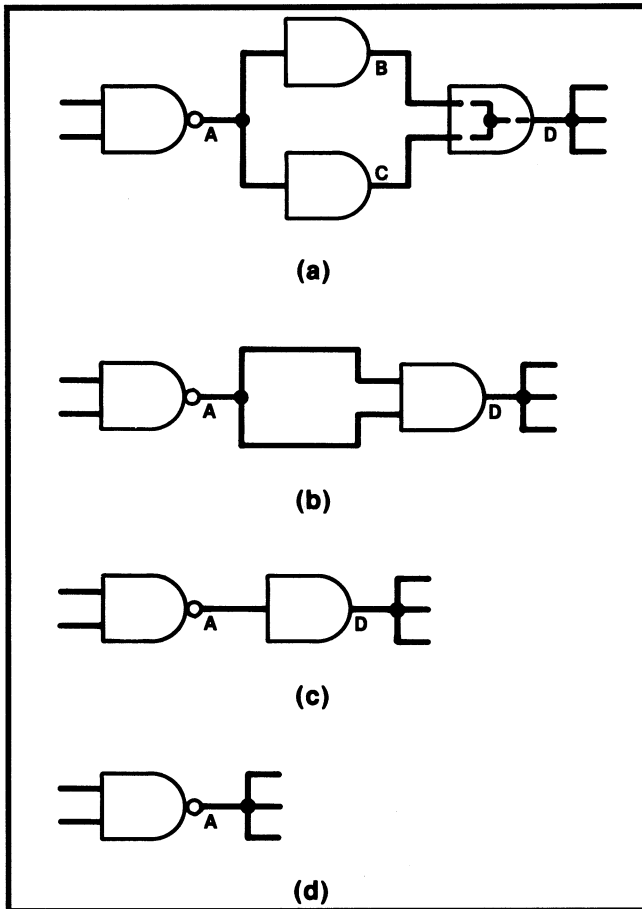


Figure 3. Sequence of model simplifications.

consists of all the SRLs controlled by that clock and all the cones* of their D and CLK inputs. Other system clocks feeding the same partition are set to their inactive value during testing. Testing one partition at a time avoids potential races that can occur when two or more system clocks are simultaneously activated.

Before tests are generated for a partition, its SRLs are replaced by combinational models. Figure 4 shows the model for the SRL in Figure 1. We do not consider faults internal to the SRL, as they will be detected by a separate “flush” test of the shift register.⁴ The model has a primary input (PI) QI to represent the SRL’s previous state and a primary output (PO) QO corresponding to its current state. The value of QI in a generated test must be applied to the scan-in PI and shifted in. Similarly, detecting a fault at QO implies that the value captured in the SRL must be shifted out to the scan-out PO for detection. Therefore, the model does not explic-

* A cone associated with a lead A consists of all logic feeding A , bounded by primary inputs and SRLs.

itly use the SRL’s scan data (SD) and scan clock ($SCLK$) leads. Note that QS and QM have the same value after scan-in.

This simple model correctly reflects the operation of the SRL during testing and lets the test generator work accurately without additional analysis. In contrast, the PODEM-X system¹ must analyze an SRL’s clock value to determine whether a fault effect on its data line is captured. Moreover, PODEM-X cannot generate tests for faults that affect clock propagation. With the model of Figure 4, a test generator will easily find out that to detect a fault whose effect propagates on the CLK line (including faults in the clock gating logic), QI and D must have opposite values (which means that the SRL must be preset with value \bar{D}).

We emphasize that our modeling technique allows uniform treatment of the logic feeding data and clock lines of SRLs. In other words, the test generator is given an ordinary combinational circuit where clocks need not be identified. In contrast, the technique described by Ogihara et al.⁷

introduces a special primitive for SRLs, requires identifying clock gating signals, and uses four additional logic values for clock propagation.

In LTG, clock identity is required only during pre- and postprocessing. The post-processor uses this knowledge to activate (that is, generate a pulse on) the clock PIs set to an active value during test generation.

Overall LTG structure

As is common in automatic test generation (ATG) systems, LTG couples two different strategies:

- *Phase 1*: a low-cost fault-independent method provides a set of tests for a large percentage of faults.
- *Phase 2*: a fault-oriented algorithm targets the remaining undetected faults.

Figure 5 outlines the structure of the first LTG phase. Tests are generated by SMART (Sensitizing Method for Algorithmic Random Testing), which replaces random test generation (RTG) used in

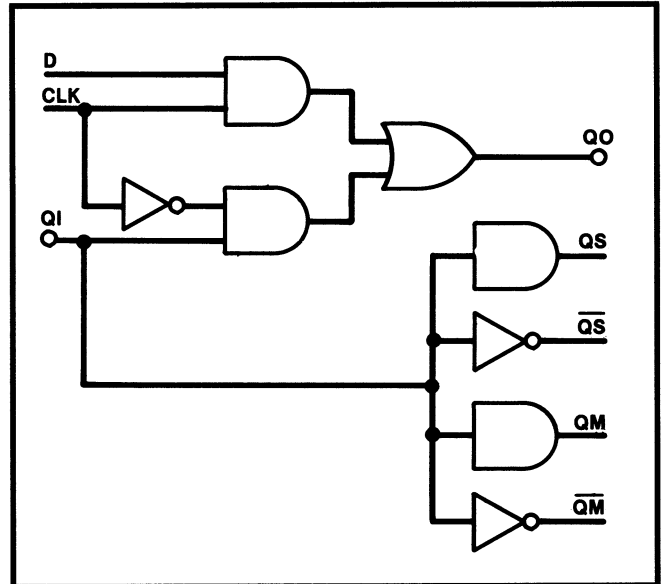


Figure 4. Combinational SRL model.

```

repeat
  {  $t = SMART()$ 
    if  $useful(t)$  then add  $t$  to the test set
  }
until  $endphase1()$ 

```

Figure 5. First phase of LTG.

```

while target_faults  $\neq \emptyset$ 
{
  remove a fault ( $f$ ) from target_faults
   $t = \text{FAST}(f)$ 
  if  $t \neq \emptyset$ 
  {
    Evaluate_and_improve( $t$ )
    add  $t$  to the test set
  }
}

```

Figure 6. Second phase of LTG.

most ATG systems. SMART uses fault simulation as an integral part of test generation. The function *useful* determines whether the test t generated by SMART should be added to the test set or discarded. The function *endphase1* decides when the switch to the second phase should occur.

Figure 6 outlines the structure of the second phase of LTG. The initial set of target faults is the set of faults undetected at the end of the first phase. Tests are generated by FAST (Fault-oriented Algorithm for Sensitized-path Testing).

A test vector produced by a fault-oriented algorithm is, in general, partially specified; that is, some PIs have unspecified (x) values. LTG uses a *dynamic compaction* technique (similar in concept to the method described by Goel and Rosales⁸) that tries to set PIs still unassigned, so that t will detect additional target faults. This is done by *Evaluate_and_improve*, which also includes fault simulation of t and updating the set of target faults.

Critical path tracing

Both phases of LTG use CRIPT (Critical Path Tracing) for fault simulation. CRIPT is a specialized technique for combinational circuits and is described in detail elsewhere.³ Here we summarize its distinguishing features:

- It deals with faults only implicitly and does not require fault enumeration.

- It deals directly only with detected faults rather than all simulated faults.
- It is based on a path-tracing algorithm that does not require computing values in faulty circuits.
- It is an approximate method, but the approximation seldom occurs and has practically no impact on test generation.

In addition, we will show how a close interaction with CRIPT lets both phases of LTG maximize the number of faults detected per test. 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 delimit 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 detectable faults that can cause l to take value 1(0). For example, consider the circuit and the test shown in Figure 7. (Critical paths are shown by heavy lines. A line l with value v in test t is critical if t detects the fault $l \text{ s-a-}\bar{v}$.) This test detects all the faults that can make $E=1$, $E2=1$, and $G=0$; hence E and $E2$ are 0-stops and G is a 1-stop (note that $E1$ is not a 0-stop). Stop lines are determined according to the following rules:

1. A PI l is a v -stop if it has had a critical value v (or $l \text{ s-a-}\bar{v}$ was identified as undetectable).
2. The output l of a gate with inversion i is a v -stop if it has had a critical value v (or $l \text{ s-a-}\bar{v}$ was identified as undetectable) and all the gate inputs are $(i \oplus v)$ -stops.
3. A fanout branch l is a v -stop if it has had a critical value v (or $l \text{ s-a-}\bar{v}$ was identified as undetectable) and its stem is a v -stop.

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 not a c -stop line.

For the example of Figure 8, if we assume that C is not a 0-stop, then G is a restart gate. No special effort is required to determine the restart gates, as they are a subset of the gates where CRIPT terminates its backtracing. In Figure 8, to make the same test detect additional new faults, we should set $E=D=1$; then C becomes a critical line and the same test is made to detect $C \text{ s-a-}1$ and potentially other faults that make $C=1$. These faults are determined by restarting CRIPT from the restart gates.

We emphasize that finding restart gates and stop lines would be much more difficult with conventional fault-simulation techniques. CRIPT's capability to do a partial fault simulation beginning from the restart gates lets test vectors be *incrementally generated and evaluated*.

The original CRIPT algorithm³ dealt only with binary values. For LTG we extended CRIPT to process partially specified vectors.

LTG Phase 1

Although generating a (pseudo-)random vector is straightforward, RTG suffers from the following problems:

1. The number of new faults detected per test is initially large, but decreases rapidly. Hence the number of vectors that do not detect any new faults increases accordingly and the time spent in their evaluation by fault simulation is wasted.
2. "Random-pattern resistant" faults are detected only after many random vectors are generated and evaluated, even if tests for some of these faults can be easily produced by a deterministic algorithm.
3. Special care is needed in handling PIs that correspond to system clocks in a scan-design circuit, because if we let such a PI be assigned 0 and 1 values with equal probabilities, almost half of the generated

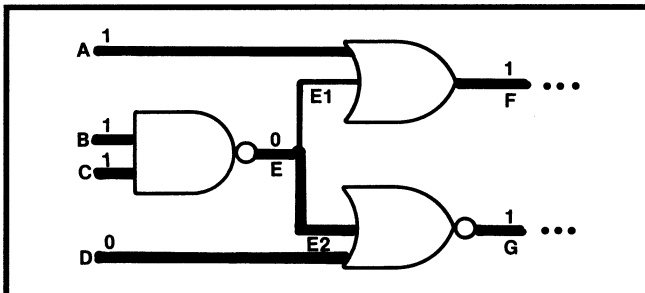


Figure 7. Example of stop lines.

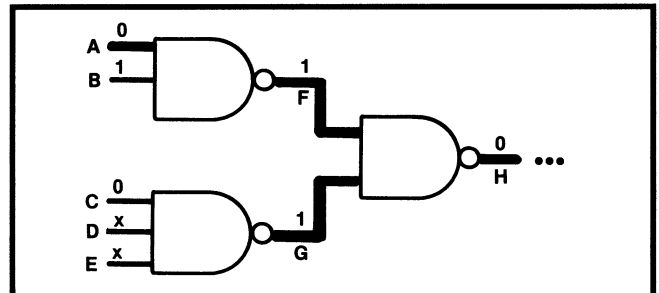


Figure 8. Example of restart gate.

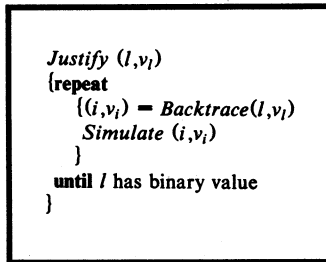


Figure 9. Line justification.

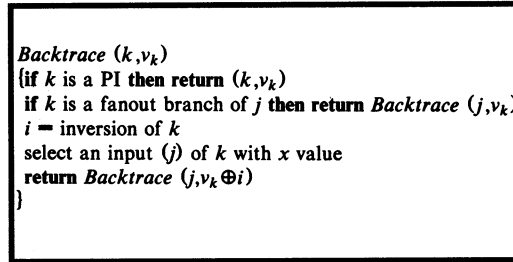


Figure 10. Backtracing an objective.

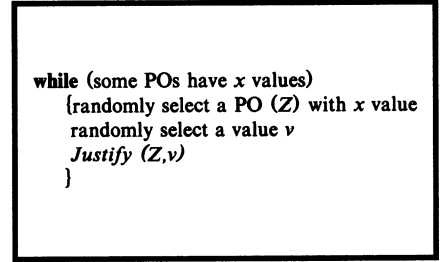


Figure 11. First stage of RAPS.

tests (that is, the ones in which the clock is inactive) will not detect any new faults.

Random Path Sensitization (RAPS)⁹ is an alternative to RTG that combines features of deterministic test generation with those of RTG to achieve higher fault coverage and smaller test sets with lower computational effort. SMART extends these features of RAPS.

RAPS' goal is to create in every test a set of random critical paths between PIs and POs. Central to RAPS is the concept of *objective*, that is, a desired value v_l for line l . *Justify*, shown in Figure 9, attempts to achieve an objective (l, v_l) by repeatedly finding a PI assignment (of value v_i to PI i) that is likely to contribute to setting l to v_l . Internal line values are generated only by simulating PI assignments. Mapping an objective into a PI assignment is recursively done by *Backtrace* (see Figure 10). In RAPS, *Backtrace* randomly selects a gate input with value x to follow.

RAPS generates a vector in two stages. In the first stage, outlined in Figure 11, it generates a partial vector to justify a set of random PO values. Usually all the PO values are justified by assigning only some PIs. The second stage of RAPS assigns the PIs left with x values such that sensitized paths are generated. RAPS does this by selecting gates that satisfy the following conditions (see Figure 12):

1. Their output value is justified by an input having the controlling value of the gate.
2. At least one gate input has value x .

Then RAPS tries to justify noncontrolling values on the gate inputs with value x (such as b in Figure 12).

Figure 13 illustrates one problem encountered in the first stage of RAPS. $A=0$ was selected to justify the value of $PO1$, then $B=0$ to justify the value of $PO2$. But the latter selection precludes propagation of any fault effects to $PO1$.

Another problem is illustrated in Figure 14. Because of the random selection process of RAPS, half the tests in which $Z=1$

will have $B=0$. But only one test with $B=0$ (and $C=1$) is useful, since none of the subsequent tests with $B=0$ will detect any new faults on the PO Z .

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 to value 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 15 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 setting $b=1$ and $c=1$ does not detect any additional faults because d is not critical. Not only is setting $b=1$ and $c=1$ a waste of time, but it will also have the detrimental effect of precluding the detection of additional faults through gate h (because we have to set $e=0$).

SMART. Like RAPS, SMART starts by justifying a randomly chosen value for a randomly selected PO Z . But unlike RAPS, SMART tries to create critical paths leading to Z before processing other POs. Thus for the example shown in Figure 13, after justifying $PO1=0$ by $A=0$, SMART will try to set $B=1$ before processing $PO2$. Then $PO2=0$ will be justified by $C=0$.

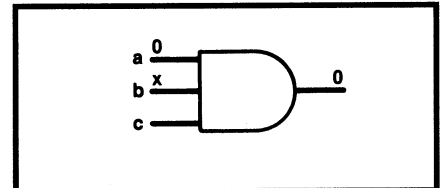


Figure 12.

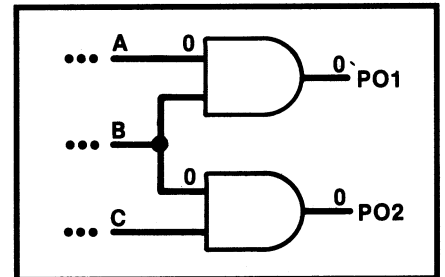


Figure 13. Effect of multiple output justification on path sensitization.

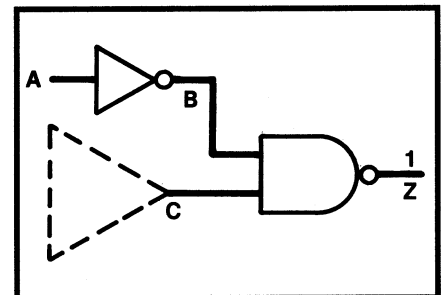


Figure 14. Inefficiency of purely random justification.

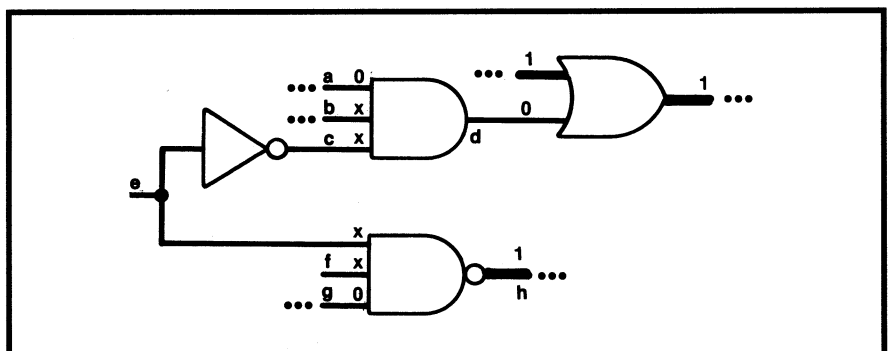


Figure 15. Which gate to sensitize: d or h ?

```

SMART()
{while (useful POs have  $x$  values)
  {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  $Z$ 
   Selective_justify ( $Z, v$ )
   CRIPT()
   while ( $restart\_gates \neq \emptyset$ )
     {remove a gate ( $G$ ) from  $restart\_gates$ 
       $c$  = controlling value of  $G$ 
      for every input ( $j$ ) of  $G$  with  $x$  value
        Justify ( $j, \bar{c}$ )
        CRIPT()
      }
    }
  }
return vector of PI values
}

```

Figure 16. SMART.

Using stop lines avoids the problem illustrated in Figure 14. After the first test that sets $B=0$ and $C=1$, B becomes a 0-stop and should not be used anymore to justify $Z=0$. SMART uses a *Selective_backtrace* procedure that differs from *Backtrace* in the following way. Whenever an objective (k, v_k) for a gate output is mapped into an objective (j, v_j) for a gate input, *Selective_backtrace* gives preference to inputs that are not v_j -stops. In addition, SMART will not select a value v as an objective for a PO that is a v -stop.

After the selected PO is driven to a binary value, SMART uses CRIPT to do a fault simulation of the partially generated vector. Then it selects a restart gate and tries to justify noncontrolling values on its inputs with value x . Thus in Figure 15 SMART will ignore d , which is not a restart gate, and will select gate h and try to set $e=1$ and $f=1$.

SMART is outlined in Figure 16. A line l that is not 0(1)-stop is said to be 0(1)-*useful*, since there are some new faults that can be detected when l has value 0(1). A line that is 0- or 1-useful is said to be *useful*. SMART selects a PO objective (Z, v) such that Z is v -useful. *Selective_justify* is similar to *Justify* shown in Figure 9, except that it uses *Selective_backtrace* instead of *Backtrace*. The partial vector generated by *Selective_justify* is fault simulated by CRIPT. CRIPT backtraces critical paths from 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. This close interaction with CRIPT greatly contributes to the efficiency of SMART.

When is a test useful? In most ATG systems using the two-phase approach, a test generated during the first phase is considered useful if it detects new faults. But let's consider a test t that only detects a new fault on a PO. Because the same fault will also be detected by many other tests, adding t to the test set will unnecessarily increase its size.

In general, the detection of a fault f also results in detecting many other faults along the sensitized path that propagates the error originating at f to POs. Consider a still-undetected fault l s - a - v with the property that every other detectable fault that can cause l to take value v has been detected by the tests generated so far. In other words, l s - a - v is the lowest level (that is, the closest to PIs) undetected fault whose effect can set l to value v . Let's refer to such a fault as a *boundary fault*.

LTG considers a test useful if it detects at least one boundary fault. If a test detects only nonboundary faults, most of them (usually all of them) will be detected anyway by tests that detect boundary faults. Any boundary fault not detected during Phase 1 will become a target for the second phase. Of course, this strategy involves the risk that some nonboundary faults detect-

ed by a discarded test will not be detected by tests that detect boundary faults. Nevertheless, our results show that this concept of usefulness leads to a reduction in the number of generated tests without increasing the computation time.

Before any test is generated, the set of boundary faults is the set of PI faults. It is easy to show that any new stop line created during Phase 1 of LTG results from detecting a boundary fault. Thus in practice, *a test is considered useful if it creates new stop lines*.

When to switch to Phase 2. The number of new faults detected per test in Phase 1 generally decreases as the number of tests increases; hence at some time it is necessary to switch to Phase 2. Unlike other ATG systems which quit Phase 1 only after it has "run out of steam," the objective in LTG is to maximize the combined effectiveness of Phase 1 and Phase 2. The switch occurs when LTG determines that Phase 2 will be more effective than Phase 1, even if at that time Phase 1 may still detect a relatively large number of new faults. The function *endphase1* analyzes the last n generated tests (whether or not they were useful) by computing the average number of new faults detected per test (the number of new faults detected by a rejected test is set to 0). If this average becomes less than the number of useful POs, then *endphase1* returns *true*, signaling the switch to the second phase. In other words, the switch occurs when Phase 1 detects, on the average, less than one new fault per useful PO. Most of the remaining faults are either undetectable or difficult to detect by a fault-independent algorithm.

Increasing n prolongs Phase 1 because more of the early tests (that detected more faults) are allowed to contribute to the average. On the basis of our experience with several large circuits, n was fixed at 30.

LTG Phase 2

In Phase 2 (see Figure 6) LTG tries to generate tests for the faults not detected in Phase 1.

Target fault selection. Among the potential new target faults (that is, still-undetected faults that have not been targets before), LTG selects the one located at the lowest possible level in the circuit. In this way, successful test generation for the selected target fault f creates a long sensitized path between the site of f and a PO, increasing the chance of detecting other

still-undetected faults along the same path. This strategy is commonly used in ATG systems and tends to increase the number of new faults detected per test.

Unlike some other ATG systems,¹⁰ LTG does not limit the target faults to the checkpoint faults of the circuit (checkpoints consist of PIs and fanout branches). Recently, it has been shown that checkpoint faults do not always provide a sufficient set of target faults.¹¹ That LTG deals with the entire (uncollapsed) set of stuck faults does not impact its efficiency, because CRIPT is an implicit fault-simulation technique not affected by the size of the simulated set of faults. The only situation in which LTG does some fault collapsing is after FAST fails to generate a test for a target fault f . Then LTG identifies the faults equivalent to or dominated by f so that none of these faults will become a target for FAST. For example, for the circuit in Figure 17, let's assume that FAST has determined that the fault C s - a -0 is undetectable. Then LTG will mark as undetectable the s - a -1 faults on the lines E, F, D, A , and B .

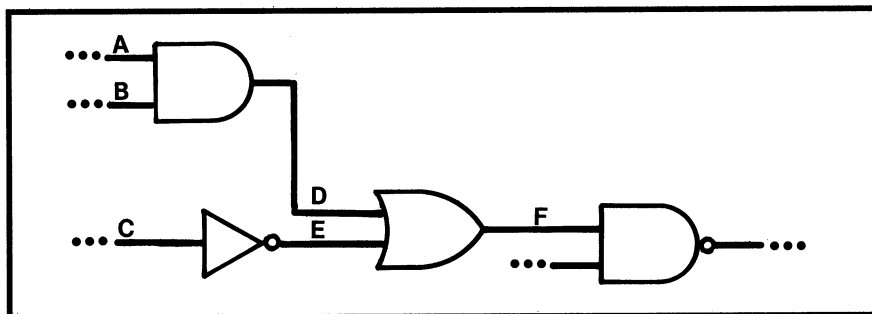


Figure 17.

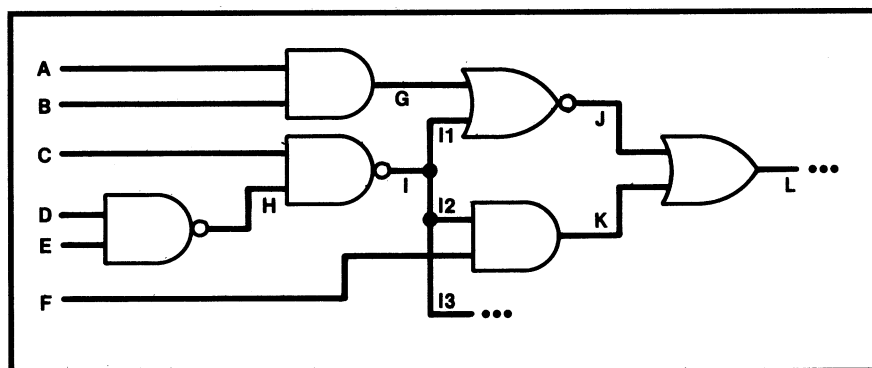


Figure 18.

Where to stop backtracing. FAST combines and extends features of PODEM¹² and FAN.¹³ PODEM is directed by objectives. To generate a test for the fault l s - a - v , a first objective is to activate the fault, that is, to set l to value v . After this is achieved, objectives are chosen to propagate an error (D or \bar{D}) to a PO. An objective is mapped into a PI assignment using *Backtrace* shown in Figure 10, with the difference that the selection of a gate input is no longer random (as in RAPS or SMART), but is guided by controllability cost functions.

FAN extended the backtracing concept of PODEM by allowing the backtracing to stop at certain internal lines rather than always go all the way to PIs. The lines where FAN stops backtracing are called *head* lines and are defined as follows. A line is said to be *free* if all its predecessors have only one fanout. A head line is a free line that directly feeds a nonfree line. For example, in Figure 18 the free lines are A, B, C, D, E, F, G, H , and I . Among these, F, G , and I are head lines. The reason for stopping the backtracing at a head line is that any value of a head line can be justified without conflicts with other values already assigned. Hence any value desired for a head line is immediately assigned without justifying it by PI assignments. The assigned head lines are justified only after FAN propagates an error to a PO.

Since an assignment of a head line is tried before its value is justified, in the case when FAN fails to propagate an error to a PO, backtracing is immediately done by complementing the value of the head line, and the time spent (by PODEM) in justification is saved. This leads to a significant reduction in the amount of backtracking.

We have extended the concept of head lines so that *we stop backtracing at any line that we know can be assigned without conflicts*. We couple the identification of these lines, referred to as *backtrace-stop lines*, with the computation of cost functions to guide the algorithm.

Guiding a test generation algorithm. The execution of a fault-oriented algorithm can be viewed as a search process. One type of decision in this process is the selection of the next problem to work on among the unsolved problems existing at a certain stage; an example is selecting one of the inputs of an AND gate to be set to 1 to justify a 1 value on its output. Since all of these problems must be solved, we would like to *attack the most difficult problem first* so as not to waste time solving easier problems when a harder one cannot be solved. Another type of decision

is to select one of the possible solutions to a problem; for example, selecting one of the inputs of an AND gate to be set to 0 to justify a 0 value on its output. Here we would like to *try the easiest solution first*. To guide the search, we need cost functions that measure "difficulty." Typically, cost functions are of two types: *controllability costs* that indicate the relative difficulty of setting a line to a value, and *observability costs* that indicate the relative difficulty of propagating an error from a line to a PO.

The worst-case behavior of a test generation algorithm is characterized by many wrong decisions from which the algorithm must recover using backtracking. A decision is incorrect if it leads to conflicts (inconsistencies). On the other hand, the best case occurs when the execution completes without backtracking. Hence the objective of cost functions should be to minimize the amount of search by *guiding the algorithm toward decisions less likely to cause conflicts*. The potential for conflicts relates directly to the fanout structure of the circuit and not to its size. For example, no conflicts can occur in any fanout-free circuit or in any circuit without reconvergent fanout. Note that commonly used control-

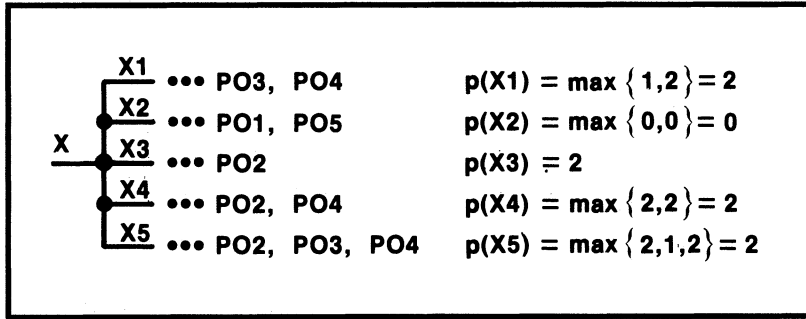


Figure 19. Penalties based on reconvergent fanout.

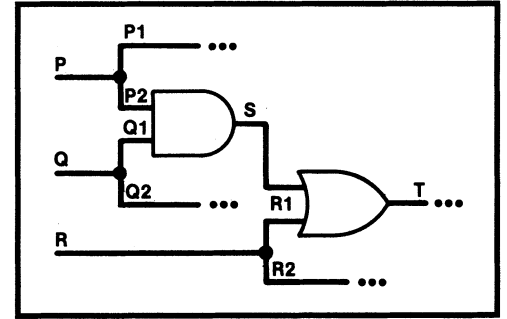


Figure 20.

lability and observability measures, such as SCOAP,¹⁴ or the measures used in random testing,¹⁵ do not reflect the potential for conflicts and therefore are not applicable for our objective.

First let's consider controllability costs. A cost of 0 should denote an assignment that cannot lead to conflicts. This will let us identify backtrace-stop lines among the lines with 0 controllability cost. We now introduce several controllability cost functions with increasing accuracy. A unique feature of these cost functions is that the cost of a fanout branch is different from the cost of its stem.

The first measure is a variant of the "fanin-weight" cost proposed by Putzolu and Roth.¹⁶ The controllability cost $C(l)$ of a line l reflects the relative potential for conflicts that result from trying to justify a value for l . $C(l)$ is computed as follows:

- if l is a PI, $C(l)=0$
- if l is a gate output

$$C(l) = \sum_i C(i)$$

where the summation is over all the gate inputs i .

- If l is a fanout branch of a stem i whose fanout count is f_i

$$C(l) = C(i) + f_i - 1$$

It is easy to see that $C(l)=0$ if and only if l is a free line. Thus l is a head line if $C(l)=0$ and l feeds a line k with $C(k) \neq 0$; therefore this measure can be used to identify head lines.

Unlike the first measure, which is strictly topological, the second measure also takes the logic into account. It uses two controllability cost functions, $C0(l)$ and $C1(l)$, to reflect the relative potential for conflicts that result from trying to justify, respectively, a 0 and a 1 value for l . These cost functions are derived from those used by Rutman¹⁷ and are computed as follows:

- if l is a PI, $C0(l)=C1(l)=0$
 - if l is the output of an AND gate
- $$C0(l) = \min_i \{C0(i)\}$$
- $$C1(l) = \sum_i C1(i)$$

where the minimization and summation are over all the gate inputs i . The computation for the other gate types is similar.

- If l is a fanout branch of a stem i whose fanout count is f_i
- $$C0(l) = C0(i) + f_i - 1$$
- $$C1(l) = C1(i) + f_i - 1$$

To illustrate the difference between these two measures, consider again the circuit of Figure 18. Using the first measure, $C(l)=0$ for every free line l , $C(I1)=C(I2)=C(I3)=2$, $C(J)=C(K)=2$, $C(L)=4$. Using the second measure, $C0(l)=C1(l)=0$ for every free line l , $C0(k)=C1(k)=2$ for $k \in \{I1, I2, I3\}$, $C0(J)=0$, $C1(J)=2$, $C0(K)=0$, $C1(K)=2$, $C0(L)=0$, $C1(L)=2$. Thus while the first measure identifies F , G , and I as head lines, the second one identifies L as a backtrace-stop line for value 0, and F , G , and I as backtrace-stop lines for value 1. (It can be easily verified that $L=0$ can be justified without assigning any line with fanout.)

The first two measures use $f_i - 1$ as a "penalty term" added to the cost of all the f_i fanout branches of a stem. The third measure penalizes only reconvergent fanout branches because only reconvergent fanout can cause conflicts. The penalty $p(l)$ for a fanout branch l is the maximum number of other fanout branches of the same stem that reconverge with l . This is illustrated in Figure 19, where for every fanout branch we indicate the POs it feeds (this information is readily available from the data structures used by CRIPT) and its penalty term. Fanout branches feeding the same PO are reconvergent. The costs $C0(l)$ and $C1(l)$ of a line l are computed in the same way as before except that when l

is a fanout branch, $p(l)$ replaces the term $(f_i - 1)$.

Figure 20 illustrates the advantages of the more accurate third measure. Using the second measure, $C0(S)=1$, $C1(S)=2$, $C0(T)=3$, $C1(T)=1$. Using the third measure and assuming that P and Q do not have reconvergent fanout, we obtain $C0(S)=C1(S)=0$, $C0(T)=1$, $C1(T)=0$. A first advantage is that now T is a backtrace-stop line for value 1. The second advantage is that now $T=1$ will be justified by $S=1$, which cannot lead to conflicts, while with the second measure we would have selected $R1=1$ (because $C1(R1) < C1(S)$), which may lead to conflicts by assigning a line (R) with reconvergent fanout. These advantages reduce the amount of backtracking done by FAST.

Both FAST and dynamic compaction are also guided by observability costs. The observability cost of a line l , $O(l)$, reflects the relative potential for conflicts involved in observing the value of l at a PO and is computed by rules similar to those used by Rutman:¹⁷

- if l is a PO, $O(l)=0$
- if l is an input of an AND gate with output k

$$O(l) = O(k) + C1(k) - C1(l)$$

where $C1(k) - C1(l)$ is the cost of setting all inputs of k except l to the non-controlling value 1. The computation for the other gate types is similar.

- If l is a stem

$$O(l) = \min_i \{O(i)\}$$

where i ranges over the fanout branches of l .

While similar rules are used in most other ATG systems, the observability costs computed by LTG provide better guidance because they are based on controllability costs that more accurately re-

flect the potential for conflicts. This better guidance results in less backtracking in FAST and also increases the efficiency of dynamic compaction.

FAST. Figure 21 presents a high-level recursive outline of FAST. At every level of recursion, FAST starts by analyzing the values inherited from the previous levels. All binary values are justified by assignments to backtrace-stop lines. FAST returns successfully when an error has been propagated to a PO; then the only remaining work is to justify the values of the assigned backtrace-stop lines (this is always possible without conflicts and hence cannot cause backtracking). FAST recognizes that the generation of a test cannot succeed when the target fault f s - a - v cannot be activated (because f has value v) or when an error (D or \bar{D}) cannot be propagated to a PO; then it returns indicating failure.

If the existing values do not indicate an immediate success or failure, then FAST finds a next objective to work on. The first objective, associated with the target fault f s - a - v , is (f, v) . After the target fault has been activated, *Objective* selects a gate G having a D or \bar{D} on an input and x on the output; then the next objective is (i, \bar{c}) , where i is an input of G with value x , and c is the controlling value of G . The selection of G is such that the most observable error is selected for propagation; among the inputs of G with x value, i is the most difficult to set to value \bar{c} .

The selected objective is mapped (by *Limited_backtrace*) into an assignment (i, v_i) , where i is a backtrace-stop line for value v_i . Then the assignment of i is simulated (using 5-valued simulation) and the entire process is recursively repeated. If the next level of recursion fails to generate a test, FAST tries to reverse the assignment of i to \bar{v}_i . This differs from the corresponding process in FAN because, in general, a line that is a backtrace-stop for value v is not also a backtrace-stop for \bar{v} , as is the case with head lines. The way the assignment $i = v_i$ is reversed depends on whether the value \bar{v}_i can also be justified without conflicts. If the cost of setting i to \bar{v}_i is 0, the assignment $i = \bar{v}_i$ is simulated and tried. If this fails too, FAST sets i to x and returns indicating failure. If the cost of setting i to \bar{v}_i is nonzero, then (i, v_i) becomes an objective and FAST continues by trying to justify this value.

FAST is an exhaustive algorithm which, if run to completion, either generates a test for a fault or proves that the fault is unde-

```

FAST(f)
{if (error at PO) then
  {justify all assigned backtrace-stop lines
   return vector of PI values
  }
if (test not possible) then return  $\emptyset$ 
 $(k, v_k) = \text{Objective}(f)$ 
repeat
   $(i, v_i) = \text{Limited\_backtrace}(k, v_k)$ 
  Simulate  $(i, v_i)$ 
   $t = \text{FAST}(f)$ 
  if  $t \neq \emptyset$  then return  $t$ 
  if (cost of setting  $i$  to  $\bar{v}_i$  is 0) then
    {Simulate  $(i, \bar{v}_i)$ 
      $t = \text{FAST}(f)$ 
     if  $t \neq \emptyset$  then return  $t$ 
     Simulate  $(i, x)$ 
     return  $\emptyset$ 
    }
  else  $(k, v_k) = (i, \bar{v}_i)$ 
}

```

Figure 21. FAST.

```

Evaluate_and_improve(t)
{CRIPT(i)
while there exist untried restart gates
  {select a restart gate  $G$ 
    $j =$  the input of  $G$  with controlling value
    $g = \text{Target\_b}(j)$ 
   if  $g \neq \emptyset$  then
      $t = \text{FAST}(g)$ 
     if  $t \neq \emptyset$  then Evaluate_and_improve(t)
   }
for every useful PO  $Z$  with  $x$  value
   $g = \text{Target\_x}(Z)$ 
  if  $g \neq \emptyset$  then
     $t = \text{FAST}(g)$ 
    if  $t \neq \emptyset$  then Evaluate_and_improve(t)
  }
}

```

Figure 22. Dynamic compaction.

tectable. Because the guidance provided by our cost functions attempts to avoid conflicts, tests for most detectable faults are generated with very little or no backtracking. FAST uses a backtracking limit to control the amount of search done for undetectable or hard-to-detect faults.

Dynamic compaction. Most ATG systems attempt to reduce the size of the generated test set by *static compaction*, which consists of iteratively searching for pairs of compatible vectors t_i and t_j and replacing them by one vector $t_i \cap t_j$ (two vectors are compatible if they do not specify complementary values in any position; $t_i \cap t_j$ has all the binary values of both t_i and t_j). Static compaction is a postprocessing operation done after all vectors have been generated. By contrast, *dynamic compaction* processes every partially specified vector immediately after generation. The principle of dynamic compaction was introduced by Goel and Rosales.⁸ Their method produces smaller sets of tests with less computational effort than static compaction. It consists of repeatedly selecting a secondary target fault and trying to generate a test for it without changing the already assigned PIs. The most important issue in dynamic compaction is the selection of secondary target faults.¹⁸

Our dynamic compaction technique, represented in Figure 6 by *Evaluate_and_improve*, is first activated after a test has been generated for a primary target fault (denoted by f in Figure 6). On the basis of a close interaction with CRIPT, it tries to take advantage of the partial vector al-

ready generated by searching for faults that are activated and/or whose effects can be easily propagated to a PO. Clearly, the restart gates are very useful in searching for such faults. For example, consider the restart gate G in Figure 8. Because C is not a 0-stop, there exists at least one detectable (but not yet detected) fault that can make $C=1$. Moreover, once the effect of such a fault is propagated to G , it most likely will be further propagated to a PO, because G is critical.

Evaluate_and_improve is outlined in Figure 22. (Many details are omitted for sake of brevity.) After evaluating test t by CRIPT, it repeatedly selects a restart gate G and searches for an already activated fault whose effect can be propagated to the input of G having the controlling value of the gate. This search is done by *Target_b*, where “ b ” denotes that the search takes place in an area where signals have binary values. When such a fault g is found, it becomes a (secondary) target for FAST. If FAST succeeds in extending t to detect g , dynamic compaction continues recursively. After the restart gates are exhausted, *Evaluate_and_improve* looks for secondary target faults in the cones of useful POs having x values. This search is done by *Target_x*.

Target_b proceeds along a path of useful lines with binary values, trying to find an already activated undetected fault whose effect can be propagated along the path. Figure 23 illustrates the different situations analyzed by *Target_b* when an AND gate (whose output value is useful) is encountered on the path. The possible out-

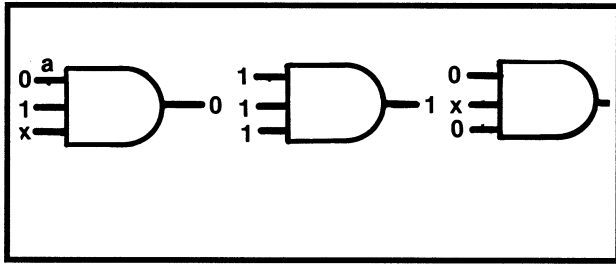


Figure 23. Situations analyzed by *Target_b*.

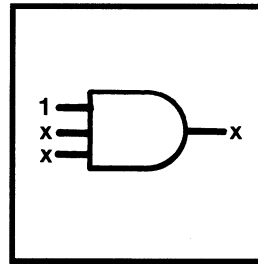


Figure 24. Gate analyzed by *Target_x*.

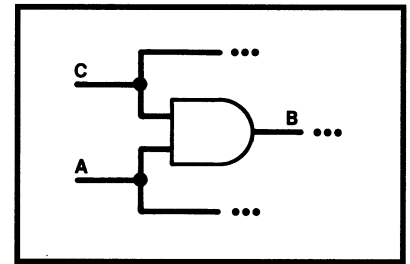


Figure 25.

comes are to continue the search from a gate input or to stop the search at the gate output. In Figure 23a, if *a* is 0-useful, the search continues from *a*. In Figure 23b, the 1-useful gate inputs are explored in increasing order of their observability costs; in this way the search is directed toward faults that are easier to observe. In Figure 23c, the search stops at the gate output. Whenever the search stops, *Target_b* returns the lowest level already activated undetected fault found along the explored path (if any).

Figure 24 illustrates the general situation encountered by *Target_x* when analyzing an AND gate whose output is useful. Priority is given to the 1-useful gate

inputs with 1 value (if any) because they may lead to already activated faults. In such a case the search is continued by *Target_b*. Otherwise the useful gate inputs with value *x* are explored (in increasing order of their observability costs).

Because tests for secondary target faults are generated under the restrictions imposed by the already assigned PI values, a failure to generate a test does not imply that the target fault is undetectable, as is the case with a primary target fault. Therefore, a low backtracking limit is used for secondary target faults.

Another difference between test generation for primary and secondary target

faults is the cost function used to guide the algorithm. For secondary target faults we cannot use the third controllability cost measure whose penalty term is computed on the basis of the amount of reconvergent fanout, because when generating tests for secondary targets, nonreconvergent fanout may also cause conflicts. This is illustrated in Figure 25. Since the PI *A* has no reconvergent fanout, the *CO* cost of *B* (using the third measure) is 0. But if *A* has been assigned value 1 to detect a previous target, stopping the backtrace at *B* for value 0 would be incorrect. Our solution is to use the third cost measure only for a primary target fault and the second cost measure for any secondary target fault.

Table 1.
LTG results on benchmark circuits.

Circuit	gates	PIs	POs	faults	tests	CPU time	% coverage
C880	383	60	26	1760	39	4	100
C1355	546	41	32	2710	93	25	99.7
C1908	880	33	25	3816	151	39	99.76
C2670	1193	233	140	5340	78	38	97.03
C3540	1669	50	22	7080	178	136	97.88
C5315	2307	178	123	10740	97	63	99.78
C6288	2416	32	32	12576	40	225	99.96
C7552	3512	207	108	15104	143	165	98.72

Table 2.
Characteristics of sample circuits.

Circuit	gates	PIs	POs	faults
A	2954	182	74	10708
B	6817	486	200	23786
C	8210	601	190	29366

Table 3.
RAPS versus SMART (run to completion).

Circuit	Algorithm	tests	CPU time	% coverage
A	RAPS	35	52	81
	SMART	32	50	89.8
B	RAPS	39	114	82.8
	SMART	40	123	85
C	RAPS	39	144	64.6
	SMART	38	155	76.9

Results and conclusions

Table 1 shows the characteristics of a set of benchmark circuits¹⁹ and the results obtained by LTG. The number of faults refers to the uncollapsed set of faults. All entries for "CPU time" are in IBM 3081K seconds. Comparing these results with other published results using the same benchmarks,²⁰ we can observe that in most cases LTG achieved the same or higher fault coverage with a much smaller test set. LTG was several times faster than other ATG systems using an IBM 3081.

The benchmark circuits, however, are not representative for VLSI circuits. We will consider a scan-design board containing approximately 75,000 gates, and three partitions extracted from this board. Table 2 summarizes the characteristics of these three circuits.

For comparison, we have implemented RAPS and PODEM in LTG. All comparisons refer to these implementations.

Table 3 compares RAPS and SMART runs that were allowed to complete Phase 1, that is, they worked until LTG determined that Phase 1 was no longer effective. SMART always achieved higher fault coverage. Table 4 compares RAPS and SMART runs that were allowed to proceed until a specified fault coverage was achieved. We can conclude that SMART is more efficient than RAPS. We should mention that both RAPS and SMART are much better than RTG. For example, RTG on circuit C achieved only 68.4 percent fault coverage, generating 145 tests in 884 seconds.

Table 5 compares PODEM and FAST. Both algorithms were run for all target faults (that is, without preceding them by a fault-independent algorithm), without dynamic compaction, with the same backtracking limit, and with the same cost functions. (Our cost functions are better than those used in the original implementation of PODEM.) The results show that FAST is a more efficient algorithm. FAST was also about 10 times faster than an implementation of the *D*-algorithm currently in use within AT&T.

Table 6 shows that the results obtained by the combined execution of SMART and FAST are better than those obtained by RAPS and PODEM. In the runs marked with (1), the unspecified PIs were assigned random binary values; the runs marked with (2) were done with dynamic compaction.

The following results were obtained by LTG on the board from which the parti-

tions A, B, and C (Table 2) were extracted. The board contains about 75,000 gates and 1500 SRLs. In about 15 minutes of IBM 3081K CPU time, SMART achieved 86 percent fault coverage. The entire LTG run took one hour and 35 minutes and gen-

erated about 3000 tests with 94 percent fault coverage.

From our results we conclude that LTG has significantly advanced the

Table 4.
RAPS versus SMART (for specified fault coverage).


Circuit	Algorithm	tests	CPU time	% coverage
A	RAPS	24	43	70
	SMART	7	38	
B	RAPS	17	92	
	SMART	19	99	
C	RAPS	55	185	80
	SMART	19	136	
A	RAPS	34	50	
	SMART	12	37	
B	RAPS	32	110	
	SMART	26	104	
C	RAPS	159	426	
	SMART	57	196	

Table 5.
PODEM versus FAST.

Circuit	Algorithm	CPU time	backtracks	% coverage
A	PODEM	440	2361	99.31
	FAST	264	2260	99.31
B	PODEM	1324	4948	98.71
	FAST	882	4703	98.82
C	PODEM	2240	19681	97.79
	FAST	1311	13053	98.63

Table 6.
RAPS and PODEM versus SMART and FAST.

Circuit	Algorithm	tests	CPU time	% coverage
A	RAPS + PODEM (1)	222	361	99.31
	SMART + FAST (1)	119	129	99.31
	SMART + FAST (2)	103	156	99.31
B	RAPS + PODEM (1)	324	825	98.71
	SMART + FAST (1)	259	393	98.82
	SMART + FAST (2)	172	350	98.82
C	RAPS + PODEM (1)	518	2250	97.79
	SMART + FAST (1)	273	897	98.63
	SMART + FAST (2)	132	782	98.63

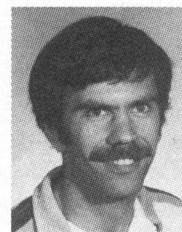
state of the art in test generation. A major contributor to the efficiency of LTG is the concept of incremental test generation and evaluation based on a close interaction with CRIPT. This is the key concept used in both our new low-cost fault-independent algorithm and our new dynamic test set compaction method. Another important new concept in LTG is the use of controllability/observability cost functions that estimate conflict potential. These functions guide our new fault-oriented algorithm toward decisions less likely to cause conflicts, and they also identify lines whose justification is conflict-free. 

Acknowledgments

We gratefully acknowledge the contributions of R.H. Hellman, H.W. Pribble, G.H. Simmons, and D.R. Sloan to the formulation of the scan-design rules; R.Bencivenga*, D.C. Chen*, J. Dussault*, S.J. Gelman*, and D.R. Gross* to the design and implementation of the testability audits; R.W. Allen*, G.J. Werner, and J.D. Yeager* to postprocessing; and K.A. Heitke to the development of the tools for handling large circuits. The support provided by R.G. Taylor and R.E. Tulloss* is greatly appreciated. We are also grateful to the referee who pointed out an error in our earlier version of CKTCLN.

References

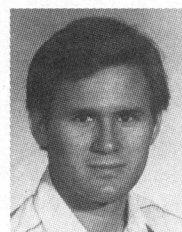
1. P. Goel and B.C. Rosales, "PODEM-X: An Automatic Test Generation System for VLSI Logic Structures," *Proc. 18th Design Automation Conf.*, June 1981, pp. 260-268.
2. C. Radke, "Experiences in VLSI Testing," *IEEE Design and Test of Computers*, Vol. 3, No. 1, Feb. 1986, pp. 83-85.
3. M. Abramovici, P.R. Menon and D.T. Miller, "Critical Path Tracing: An Alternative to Fault Simulation," *IEEE Design and Test of Computers*, Vol. 1, No. 1, Feb. 1984, pp. 83-93.
4. E.B. Eichelberger and T.W. Williams, "A Logic Design Structure for LSI Testability," *Proc. 14th Design Automation Conf.*, June 1977, pp. 462-468.
5. S. Funatsu, N. Wakatsuki, and T. Arima, "Test Generation Systems in Japan," *Proc. 12th Design Automation Conf.*, June 1975, pp. 114-122.
6. M. Abramovici et al., "Test Generation in LAMP2: System Overview," *Proc. 1985 Int'l Test Conf.*, Nov. 1985, pp. 45-48.
7. T. Ogihara et al., "Test Generation for Scan Design Circuits with Tri-State Modules and Bidirectional Terminals," *Proc. 20th Design Automation Conf.*, June 1983, pp. 71-78.
8. P. Goel and B.C. Rosales, "Test Generation and Dynamic Compaction of Tests," *Proc. 1979 Test Conf.*, Oct. 1979, pp. 189-192.
9. P. Goel, "RAPS Test Pattern Generator," *IBM Technical Disclosure Bulletin*, Vol. 21, No. 7, Dec. 1978, pp. 2787-2791.
10. A. Yamada et al., "Automatic System Level Test Generation and Fault Location for Large Digital Systems," *Proc. 15th Design Automation Conf.*, June 1978, pp. 347-352.
11. M. Abramovici, P.R. Menon, and D.T. Miller, "Checkpoint Faults are not Sufficient Target Faults for Test Generation," *IEEE Trans. Computers*, Vol. C-35, No. 8, Aug. 1986, pp. 769-771.
12. P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. Computers*, Vol. C-30, No. 3, Mar. 1981, pp. 215-222.
13. H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Trans. Computers*, Vol. C-32, No. 12, Dec. 1983, pp. 1137-1144.
14. L.H. Goldstein, "Controllability/Observability Analysis of Digital Circuits," *IEEE Trans. Circuits and Systems*, Vol. CAS-26, No. 9, Sept. 1979, pp. 685-693.
15. J. Savir, "Good Controllability and Observability Do Not Guarantee Good Testability," *IEEE Trans. Computers*, Vol. C-32, No. 12, Dec. 1983, pp. 1198-1200.
16. G.R. Putzolu and J.P. Roth, "A Heuristic Algorithm for the Testing of Asynchronous Circuits," *IEEE Trans. Computers*, Vol. C-20, No. 6, June 1971, pp. 639-647.
17. R.A. Rutman, "Fault Detection Test Generation for Sequential Logic by Heuristic Tree Search," *IEEE Computer Group Repository*, paper no. R-72-187, 1972.
18. P. Goel and B.C. Rosales, "Dynamic Test Compaction with Fault Selection Using Sensitizable Path Tracing," *IBM Technical Disclosure Bulletin*, Vol. 23, No. 5, Oct. 1980, pp. 1954-1957.
19. F. Brglez, P. Pownall, and P. Hum, "Accelerated ATPG and Fault Grading via Testability Analysis," *Proc. 1985 IEEE Int'l Symp. Circuits and Systems*, June 1985.
20. Special session on "Recent Algorithms for Gate-Level ATPG with Fault Simulation and Their Performance Assessment," *Proc. 1985 IEEE Int'l Symp. Circuits and Systems*, June 1985.



James J. Kulikowski is a member of technical staff at AT&T Information Systems, where he is involved in the development of software tools to aid in circuit design. He was with AT&T Bell Laboratories from 1977 to 1985. He received a BS in computer science from the University of Connecticut in 1977 and an MS in computer science from Northwestern University in 1980.



Premachandran R. Menon is a member of technical staff at AT&T Information Systems. He is also an editor of *IEEE Transactions on Computers*. Since joining AT&T Bell Laboratories in 1963, he has been involved in research and development in switching theory, testing, and simulation. He is a coauthor of *Fault Detection in Digital Circuits*, Prentice-Hall, 1962, and *Theory and Design of Switching Circuits*, Computer Science Press, 1975. He is a senior member of IEEE and a recipient of AT&T Bell Laboratories' Distinguished Technical Staff Award. He received a BSc from Banaras Hindu University, India, and a PhD from the University of Washington, both in electrical engineering.



David T. Miller is a member of technical staff at AT&T Information Systems, where he designs software for CAD tools. He joined AT&T Bell Laboratories in 1982. His research interests include automatic test generation and compiler design. He received his BS in computer science from the University of Nebraska at Lincoln in 1982 and his MS in computer science from the Illinois Institute of Technology in 1983.

The authors' address is AT&T Information Systems, 1100 E. Warrenville Rd., Naperville, IL 60566.



Miron Abramovici is a member of technical staff at AT&T Information Systems. Before joining AT&T Bell Labs in 1980, he was a research assistant and a part-time lecturer with the Department of Electrical Engineering at the University of Southern California. From 1970 to 1976 he was with the Department of Applied Mathematics of the Weizmann Institute of Science, Rehovot, Israel. He received an EE degree in applied electronics from the Bucharest Polytechnic in 1969 and a PhD in electrical engineering (computers) from the University of Southern California in 1980.

*AT&T Technologies Engineering Research Center.