# COMBINATIONAL TEST GENERATION USING SATISFIABILITY

by

Paul R. Stephan, Robert K. Brayton, and
Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M92/112

2 October 1992

COVER PAGE

# COMBINATIONAL TEST GENERATION USING SATISFIABILITY

by

Paul R. Stephan, Robert K. Brayton, and
Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M92/112

2 October 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

# Combinational Test Generation using Satisfiability

Paul R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley

**Abstract**

We present a new algorithm for combinational test generation which improves on Larrabee's results by using more robust and simpler heuristics. In TEGUS, test generation using satisfiability, the characteristic function of all tests for a fault is constructed in conjunctive normal form (CNF). The CNF formula is solved by an algorithm for Boolean satisfiability (SAT) using simple but powerful new heuristics. With our implementation we have algorithmically generated a test for every fault in the ISCAS test generation benchmark networks without using random tests or fault simulation, demonstrating the robustness of the method. We demonstrate the efficiency of TEGUS by comparing it with results for 16 recently published structural algorithms. TEGUS combines the advantages of the elegant organization of Larrabee's formulation with the efficiency of structural algorithms such as the D-algorithm and PODEM.

## 1 Introduction

In 1966, Roth presented the D-algorithm [37] for combinational test generation which he proved complete, meaning that if a test for a fault exists, the D-algorithm will find it if allowed to run to completion. Every deterministic test generation algorithm developed since has the same worst case complexity. The difference between algorithms guaranteed to find all tests is only in the set of heuristics used to optimize the average case performance.

Structural search algorithms such as the D-algorithm [37] and PODEM [18] are currently accepted as being the most efficient, and consequently the most practical. Larrabee recently proposed a new algorithm[24] which generates an algebraic formula to describe the network, the fault, as well as search heuristics. A test is generated by solving the formula using an algorithm for Boolean satisfiability. Using this framework, Larrabee added several popular structural search heuristics to the formula and compared them for effectiveness.

When a test generation problem is expressed as an algebraic formula in conjunctive normal form (CNF), information about the network structure is lost. For example it is difficult to determine the number of network inputs or outputs by looking at the formula. However the information which is lost does not affect the overall complexity of test generation.

On the other hand, Larrabee [24] has shown that the algebraic formulation has several advantages over structural search techniques. The algorithm to generate a test is very simple because the CNF formula uniformly represents the entire test generation problem. Even structural test generation algorithms based on a single primitive such as NAND gates use at least five valued algebras, and distinguish between forward implications, backward implications, line justification, etc. In an algorithm for satisfiability, the single operation of assigning a Boolean value true or false subsumes all of this. However, the best results in [24] are still at least an order of magnitude slower than comparable structure-based test generation algorithms.

Many other NP-hard problems have been solved efficiently by applying suitable heuristics e.g. tautology, bin packing and unate covering. Thus instead of applying structural search heuristics by adding extra clauses to the formula as in [24], we have focused on finding SAT heuristics which solve the basic formula as efficiently as possible. Different heuristics represent tradeoffs between robustness, simplicity, and efficiency. For example, exhaustive search is 100%

robust and quite simple, but not very efficient. In this report we describe the basic heuristics of TEGUS, TEst Generation Using Satisfiability, which represents a particularly good balance among these three factors:

Robustness: Without fault simulation or random tests, TEGUS can algorithmically generate a test for every fault in the ISCAS '85 [6] and ISCAS '89 [5] test generation benchmarks.

Efficiency: TEGUS performance is comparable to the best published results for structural test generation algorithms; we compare TEGUS with several recently published results, using the ISCAS benchmark networks.

Simplicity: The uniformity of the CNF formula results in a naturally straightforward algorithm to generate a test. No testability measures or complicated backtracing heuristics are needed.

We have implemented TEGUS as two separate components: a generic package for solving SAT problems, and the package specific to its application in test generation. The heuristics developed for TEGUS can be applied to other problems which are easily formulated as satisfiability questions. For example, test generation for other fault models such as bridging faults or delay faults can be implemented by generating a suitable CNF formula and then applying the same SAT package to solve it. The results in this report show that this approach to test generation is practical as well as conceptually elegant.

We assume the reader is familiar with basic test generation concepts such as justification, implication, and D-frontier[30]. The remainder of the report is organized as follows. Section 2 gives an overview of TEGUS and some base results used to evaluate the new heuristics presented. Section 3 gives the problem formulation as derived from the method of Boolean differences. Section 4 describes the heuristics used by TEGUS to solve the resulting Boolean satisfiability problem. Section 5 summarizes the experimental results.

## 2 Overview

Algorithms for test generation are broadly categorized as either structural or algebraic. Structural algorithms analyze the gate network for information to guide the decisions made during test generation. For example, PODEM[18] only assigns values at the network primary inputs, which greatly improves the average case performance for networks with a lot of reconvergence. FAN[13] assigns values to signals in the network which have fanout greater than one, as well as at specially identified head lines, and computes unique sensitization values during the search. Heuristics such as these are based on analyzing the structure of the gate network.

To select and assign a value to an input or head line, additional heuristics using controllability and observability measures are used. For example, if an OR gate must have the value one, it is sufficient to set any one of its inputs to one. One way to choose an input is to estimate how easy it is to set each input to one, a controllability measure. Many different controllability and observability measures have been used [1, 11, 19, 20, 21, 33]. All these are based on the structure of the network.

In contrast, an algebraic algorithm translates the entire test generation problem into an algebraic formula. Then algebraic operations, such as factoring and elimination, are applied heuristically to simplify the formula; finally a test is generated by solving the formula. Early algebraic techniques [40] were not methodical enough to be automated. Arbitrary formulas can be simplified methodically using binary decision diagrams[7] but they are ineffective on some networks because the intermediate formulas grow too large, whereas the same networks are processed easily by structural algorithms. In general algebraic algorithms are not considered practical for combinational test generation.

A few techniques do not fit either of these categories. Results using logic programming, neural networks or other general problem solving systems are too inefficient to be practical, although this has been improving[41]. Recently, Larrabee proposed a new algorithm based on translating a test generation problem into a formula in conjunctive normal form, or product of sums[24, 25, 26]. Consider an AND gate with two inputs A and B, and output C. The logical relationship between these three values is described by the following characteristic equation.

```
read gate network
convert to INV-AND gates
model faults

do
    generate 64 random patterns
    fault simulate
while (number_caught != 0)

for each uncaught fault f
    extract CNF formula for f
    try to satisfy formula
    if (satisfied) then
        fault simulate
    else if (unsatisfiable) then
        mark redundant
    end if
end for
```

Figure 1: Overall algorithm for TEGUS

$$(C+\overline{A}+\overline{B})(\overline{C}+A)(\overline{C}+B) \tag{1}$$

Likewise the following characteristic equation expresses the condition that two values A and B must differ (e.g. the good and faulty value of a primary output).

$$(A+B)(\overline{A}+\overline{B}) \tag{2}$$

As described later in section 3, a single CNF equation describing the test generation problem for one fault can be constructed by combining the characteristic equations for the gates with those describing the conditions for detecting the fault. Then any assignment to the variables which makes the CNF equation true is a test for the fault.

In Larrabee's algorithm[24], no algebraic manipulations are used to simplify the formula before solving it, and some structural information is needed to solve the CNF formula efficiently. Thus it is not strictly an algebraic method for test generation. On the other hand, it is not a structural method because the search algorithm is executed over a formula instead of a gate network, and the formula describes more than just the logical function of the gate network. The heuristics described in [24, 25] analyze the formula to decide which variable to assign, and what value. Although the results are much better than for algebraic techniques, the performance is still over 10 times worse than existing structural algorithms.

We present new heuristics to solve the CNF formula which greatly improves the efficiency of Larrabee's formulation. The four most important differences are 1) to use the PODEM strategy of assigning values to primary inputs (but applied to the formula, not the gate network), 2) using a dynamic variable ordering instead of static ordering, 3) using a fast, greedy heuristic for choosing which variables to assign, and 4) iterating the computation of global implications for hard to detect or redundant faults. As in [24] we try several different variable orderings in succession, each derived using different heuristics.

It is rarely possible to prove that one heuristic is more efficient than another. Plausibility arguments are valuable for developing intuition but because they are necessarily based on small fragments of networks, they have a correspondingly small chance of being correct. Decisions based on local information, no matter how reasonable, can lead to conflicts. In general, the only reliable means for comparing the efficiency of different heuristics is through the performance of actual implementations on a suitable set of examples. However, in running an experiment to compare different heuristics

|  | Tested Faults |  | Untested Faults |  | Patterns |  | Mem | Time (sec) |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Network | RND | SAT | Red. | Ab. | RND | SAT | (Mb) | RND | CNF | SAT | SIM | Total |
| c432 | 359 | 14 | 40 | 0 | 69 | 12 | 0.2 | 0.1 | 0.3 | 0.2 | 0.0 | 0.5 |
| c499 | 790 | 52 | 8 | 0 | 70 | 36 | 0.2 | 0.1 | 0.3 | 0.1 | 0.0 | 0.4 |
| c880 | 667 | 36 | 0 | 0 | 92 | 17 | 0.4 | 0.1 | 0.1 | 0.1 | 0.0 | 0.3 |
| c1355 | 977 | 73 | 8 | 0 | 70 | 36 | 0.5 | 0.1 | 0.4 | 0.1 | 0.0 | 0.7 |
| c1908 | 938 | 155 | 9 | 0 | 93 | 73 | 0.5 | 0.2 | 0.6 | 0.2 | 0.0 | 1.1 |
| c2670 | 1 325 | 246 | 107 | 0 | 96 | 89 | 0.6 | 0.2 | 1.7 | 2.3 | 0.0 | 4.4 |
| c3540 | 1 986 | 70 | 121 | 0 | 197 | 45 | 0.9 | 0.6 | 3.1 | 0.3 | 0.0 | 4.1 |
| c5315 | 3 694 | 46 | 58 | 0 | 180 | 29 | 0.9 | 0.4 | 0.7 | 0.1 | 0.0 | 1.5 |
| c6288 | 4 801 | 0 | 34 | 0 | 43 | 0 | 1.8 | 2.8 | 2.7 | 0.0 | 0.0 | 5.8 |
| c7552 | 4 416 | 354 | 131 | 0 | 197 | 127 | 1.4 | 0.7 | 3.0 | 3.5 | 0.0 | 7.6 |
| s1494 | 1 093 | 115 | 12 | 0 | 145 | 52 | 0.4 | 0.1 | 0.1 | 0.0 | 0.0 | 0.3 |
| s5378 | 2 528 | 286 | 36 | 0 | 211 | 182 | 0.7 | 0.3 | 0.6 | 0.1 | 0.0 | 1.2 |
| s9234 | 3 151 | 1 197 | 327 | 0 | 238 | 362 | 1.2 | 1.0 | 4.8 | 2.6 | 0.2 | 9.1 |
| s13207 | 5 000 | 1 955 | 131 | 0 | 231 | 580 | 1.8 | 0.7 | 9.5 | 1.3 | 0.4 | 13. |
| s15850 | 7 208 | 795 | 337 | 0 | 334 | 362 | 2.4 | 1.7 | 11. | 1.2 | 0.2 | 15. |
| s35932 | 25 150 | 0 | 3 072 | 0 | 83 | 0 | 2.1 | 2.0 | 7.9 | 0.1 | 0.0 | 12. |
| s38417 | 17 955 | 3 004 | 154 | 0 | 543 | 1 069 | 2.9 | 5.7 | 12. | 3.6 | 1.6 | 25. |
| s38584 | 23 355 | 1 337 | 1 316 | 0 | 718 | 497 | 3.6 | 4.6 | 7.1 | 0.6 | 0.7 | 16. |
| TOTAL | 105 393 | 9 735 | 5 901 | 0 | 3 610 | 3 568 | 3.6 | 21.2 | 66.1 | 16.1 | 3.4 | 119. |

Table 1: TEGUS results for ISCAS benchmarks, using random patterns followed by deterministic test generation. Full scan is assumed for the eight sequential networks. About 55% of the time is spent extracting the CNF formulas. Memory usage and CPU times are for an IBM RS/6000 320.

there are many pitfalls, such as using an inappropriate set of examples, overoptimizing for one set of examples, or not properly accounting for differences in computer performance. In addition, if some programs are not publicly available, reimplementation of another person's algorithm often lacks sufficient motivation to make the code as efficient as possible. Inability to reproduce some published results may be because of this or inaccuracies elsewhere.

To evaluate our heuristics, we compare an implementation of TEGUS against other published results using the ISCAS benchmark networks. The ISCAS '85 benchmarks are small, combinational networks which have been shown to be difficult for existing test generation algorithms[6]. The larger ISCAS '89 sequential networks[5] are tested assuming full scan of all latches. Only the eight largest ISCAS '89 benchmarks are reported since the smaller examples take negligible time and are not difficult for combinational test generation (as indicated by the results for network s1494).

The overall description of TEGUS is shown in Fig. 1. It is similar to existing systems, since most of our improvements are in the algorithm to solve the CNF formulas. The gate network is first converted into only AND gates with arbitrarily inverted inputs, similar to [42] which used only NAND gates. This simplifies the rest of the implementation such as the fault simulator, and the code to generate the CNF formulas. Note that this also increases the number of faults modeled for networks with XOR gates such as c432. The fault simulator uses a parallel pattern algorithm[44], with the fault effect propagation improved using radix sorting for events[34]. Pseudo random test patterns are optionally simulated to reduce the set of undetected faults, since this is more efficient for most networks. Then deterministic test generation is done on the remaining faults.

Most of the computation time is divided between fault simulation, extracting the CNF formulas, and satisfying the formulas. Table 1 summarizes the results of running TEGUS on the ISCAS benchmark networks. No attempt was made to minimize the test set, although the total number of patterns can be reduced from 7178 patterns to 4978 patterns using reverse fault simulation[39], which takes an additional 16 CPU seconds.

The first four columns shows the number of faults detected by random tests, detected by deterministic tests, proved redundant, and aborted. The next two columns show the number of test patterns generated during the random and deterministic phases respectively. Column Mem shows the memory usage in Mbytes. The last five columns show the

CPU times on an IBM RS/6000 320 with 32 Mbytes of memory. Column RND is the total time for the random test generation phase (primarily fault simulation time). Columns CNF and SAT show the times for extracting and solving the CNF characteristic formulas respectively. Column SIM is the time to fault simulate the test patterns generated deterministically.

The final column, Total, is the time for the entire run including reading the network, initializing the data structures, etc. For the 18 benchmark networks the total time is 122 seconds, and the maximum memory used is 3.6 Mbytes. About 55% of the time is spent extracting the CNF formulas, 20% in fault simulation, and 15% satisfying the formulas. In [24] about 75% of the time was used for satisfying the formula and only 9% in extracting the formulas. The improved heuristics in TEGUS have changed the balance so that extracting the formula is now the most time consuming step. As shown in section 5, the performance of TEGUS is comparable to the best published results for structural algorithms.

## 3 Problem Formulation

This section reviews Larrabee's formulation of test generation as a Boolean satisfiability problem. The formulation is based on expressing the characteristic equation of the method of Boolean differences [40] in CNF. Deriving the SAT formula using the method of Boolean differences as discussed below guarantees that the final algorithm using SAT is complete. The CNF formula can be constructed directly from the gate network, and is linear in the size of the network. The test generation problem then becomes the problem known as *satisfiability*[14]: is there an assignment to the variables which makes the formula true?

### 3.1 Boolean Differences

Let $F(x_1, \ldots, x_n)$ be a single output logic function, and $F_f(x_1, \ldots, x_n)$ be the corresponding function when a fault f is present. Then (3) is the characteristic function for all tests for fault f.

$$T_f = F_f \oplus F \tag{3}$$

In other words, for any general fault f, $T_f$ is true for any combination of inputs which will cause the output of the faulty network to differ from the value of the good network. Thus test generation for fault f consists of getting $T_f$ to evaluate to true.

For the stuck-at fault model, let i be a network element I/O with modeled stuck faults i/0 and i/1. Express i in terms of the network primary inputs, $i = G(x_1, \ldots, x_n)$, and then express F as $F(x_1, \ldots, x_n, G)$. Using (3) for fault i/0 and applying the definition of a stuck-at zero fault gives the following equation.

$$\begin{aligned} T_{i/0} &= F_{i/0}(x_1, \ldots, x_n, G_{i/0}) \oplus F(x_1, \ldots, x_n, G) && (4)\\ &= F(x_1, \ldots, x_n, 0) \oplus F(x_1, \ldots, x_n, G) && (5)\\ &= (F(x_1, \ldots, x_n, 0) \oplus F(x_1, \ldots, x_n, 1)) \cdot G(x_1, \ldots, x_n) && (6) \end{aligned}$$

To check this last step, note that when $G = 0, T_{i/0} = 0$, and when $G = 1$ then $T_{i/0} = F(x_1, \ldots, x_n, 0) \oplus F(x_1, \ldots, x_n, 1)$. $T_{i/0}$ is the characteristic function for all tests of fault i/0. Likewise (7) characterizes all tests for fault i/1.

$$T_{i/1} = (F(x_1, \ldots, x_n, 1) \oplus F(x_1, \ldots, x_n, 0)) \cdot \overline{G}(x_1, \ldots, x_n) \tag{7}$$

These two functions have in common the expression: (8) called the *Boolean difference* of F with respect to G.

$$\frac{\partial F}{\partial G} = F(x_1, \ldots, x_n, 0) \oplus F(x_1, \ldots, x_n, 1) \tag{8}$$

As outlined in [40], $T_f$ can be manipulated algebraically to find a test for fault f. Using the Boolean difference for test generation in this manner is called the method of Boolean differences [40].

## 3.2 Formulating the Test Problem in CNF

In the standard method of Boolean differences, the characteristic function $T_f$ is derived in a factored form with the same structure as the network being tested. To find a test, $T_f$ is simplified using algebraic operations as much as possible, and then finally some element of the on-set of $T_f$ is found. As noted by Larrabee [24], this procedure is tedious at best and far less efficient than existing structural search methods. However, Larrabee proposed expressing $T_f$ directly in a form more suitable for analysis – conjunctive normal form. Generating $T_f$ directly in CNF avoids the algebraic manipulations which make the method of Boolean differences impractical.

To derive $T_f$ in CNF, we again use characteristic functions. First the characteristic function of each primitive is expressed in CNF. For example, an AND gate $E = B \cdot C$ has the characteristic function (9).

$$(\overline{B} + \overline{C} + E)(B + \overline{E})(C + \overline{E}). \tag{9}$$

The characteristic function of a network is the conjunction of the characteristic functions of each primitive. A straightforward way to formulate $T_f$ in CNF is to consider the network representation of (3) and generate the corresponding characteristic equation. Recall that generating a test means finding an assignment for which $T_f = 1$. We can state this explicitly by adding the clause $(T_f)$ to the formula which can only be satisfied when $T_f$ is true. Now the test generation problem has become "Is there an assignment to the variables for which the formula evaluates to true?" But this is precisely the problem known as *satisfiability*[14], or SAT.

SATISFIABILITY
*Instance:* A set U of variables and a collection C of clauses over U.
*Question:* Is there a satisfying truth assignment for C?
*Cook's Theorem:* SATISFIABILITY is NP-complete.

SAT is extremely important in algorithm complexity analysis. To prove that some new problem X in NP is as hard as SAT, a reduction is found which converts any SAT problem into a corresponding X problem. This is the technique used to prove that test generation is NP-complete. Larrabee's problem formulation uses an inverse reduction to reduce a testing problem to a corresponding SAT problem.

## 3.3 Active Clauses

It is inefficient to satisfy (6) or (7) directly because the good and faulty network are only related at network inputs and outputs. The close correspondence between the two networks is only implicit in the SAT formula since the good and faulty network values are separate Boolean variables. Larrabee [24] showed how the SAT algorithm is speeded up significantly by making this information explicit with additional clauses. We present a modified derivation which uses fewer clauses than in [24].

For each signal S in the transitive fanout of the fault site, define a new variable $S_a$ called an *active variable* [24]. If $F_g$ is the value of F in the good network, and $F_f$ is the value of F in the faulty network, then the definition of $F_a$ using clauses is given by (10).

$$(\overline{F}_a + F_g + F_f)(\overline{F}_a + \overline{F}_g + \overline{F}_f) \tag{10}$$

In other words, if the fault effect is propagated through F, then $F_g$ and $F_f$ must have different values. With these definitions added to the formula, the following constraints can be added. State that the fault site F is active with the 1-clause $(F_a)$. Then for each signal G in the transitive fanout of the fault site, add the clause defined by (11) to state that if G is active then some fanout H of G must be active.

$$(\overline{G}_a + \sum_{H=\text{fanout 1 of G}}^{\text{fanout n of G}} H_a) \tag{11}$$

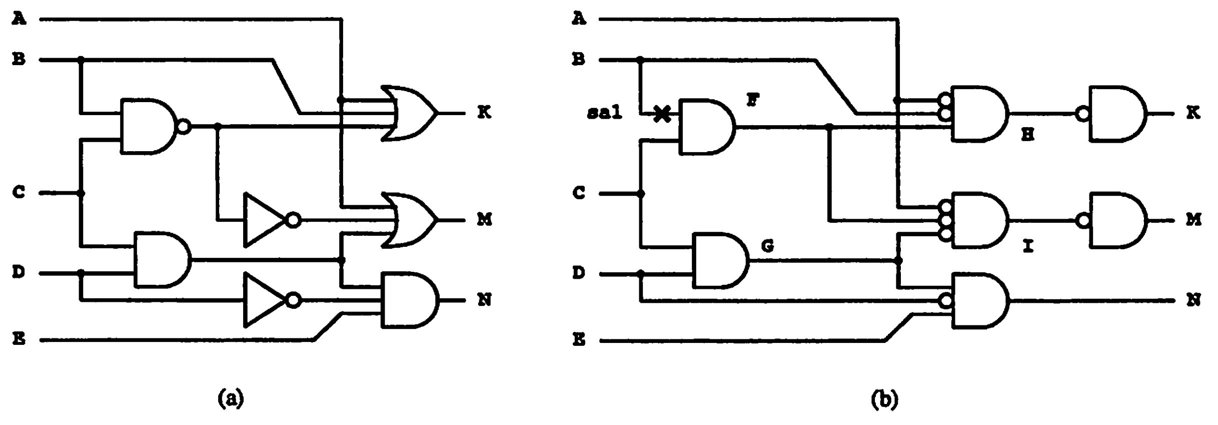


Figure 2: Example network for extracting a CNF formula for one fault. (a) Original gate network. (b) Converted to INV-AND format. Each gate is referred to by the name of its output net.

For example, if gate F fans out to gates H and I, then the clause $(\overline{F}_a + H_a + I_a)$ would be added to the formula. These additional clauses prune the search space by explicitly representing the close relationship between the good and faulty networks. Thus the SAT solver is not required to drive values all the way to network outputs before recognizing a contradiction. This representation of the active paths uses fewer clauses than in [24] since only one active variable is defined per gate, and gives a small savings in memory and CPU time.

Adding the clauses in (11) captures the heuristic that at all times there must still be some path to propagate the fault effect to an output. If a partial assignment blocks all paths from the fault site to an output, a conflict will occur for some clause from (11) possibly even before any logic values have been propagated to a network output. Thus an important heuristic in structural search methods is translated into clauses which implicitly guide the solution of the CNF formula.

## 3.4 Example

The network in Fig. 2(a) will be used to show an example CNF formula for combinational test generation of one fault. When the network is read in, it is converted into INV-AND format and all one-input gates are eliminated except at primary outputs, as illustrated in Fig. 2(b). Complex gates such as XOR gates are also decomposed directly into two levels of INV-AND gates.

Now consider the stuck-at-one fault marked in Fig. 2(b). First the network is traced using a forward depth-first search (DFS) to see which outputs can be reached by the fault effect. In this example, the fault effect can reach outputs K and M, but not output N. Then a backward DFS is done from these outputs to generate clauses describing all the gates which are part of the test generation problem for the fault. In this small example the DFS reaches all gates except for N. Now for each n-input gate, its characteristic equation is extracted in the form of n 2-clauses (clauses with 2 literals) and one (n+1)-clause.

- Good Network, 42 literals:
  $(B_g + \overline{F}_g)(C_g + \overline{F}_g)(F_g + \overline{B}_g + \overline{C}_g)$
  $(C_g + \overline{G}_g)(D_g + \overline{G}_g)(G_g + \overline{C}_g + \overline{D}_g)$
  $(\overline{A}_g + \overline{H}_g)(\overline{B}_g + \overline{H}_g)(F_g + \overline{H}_g)(H_g + A_g + B_g + \overline{F}_g)$
  $(\overline{A}_g + \overline{I}_g)(\overline{F}_g + \overline{I}_g)(\overline{G}_g + \overline{I}_g)(I_g + A_g + F_g + G_g)$
  $(K_g + H_g)(\overline{K}_g + \overline{H}_g)(M_g + I_g)(\overline{M}_g + \overline{I}_g)$

Next, clauses are generated to describe the gates in the transitive fanout of the fault site, since the values of these gates may differ between the good and faulty networks. Since the fault is on the input of gate F, the transitive fanout of the fault site consists of gates F, H, I, K and M. The description of a gate in the faulty network uses the good variables for signals which cannot be affected by the fault. For example, the description of gate H uses $A_g$, $B_g$, $F_f$ and $H_f$.

- Faulty Network, 35 literals:
$(B_f + \overline{F}_f)(C_g + \overline{F}_f)(F_f + \overline{B}_f + \overline{C}_g)$
$(\overline{A}_g + \overline{H}_f)(\overline{B}_g + \overline{H}_f)(F_f + \overline{H}_f)(H_f + A_g + B_g + \overline{F}_f)$
$(\overline{A}_g + \overline{I}_f)(\overline{F}_f + \overline{I}_f)(\overline{G}_g + \overline{I}_f)(I_f + A_g + F_f + G_g)$
$(K_f + H_f)(\overline{K}_f + \overline{H}_f)(M_f + I_f)(\overline{M}_f + \overline{I}_f)$

Active variables are defined for each gate in the transitive fanout of the fault site. If the gate is active, then its good and faulty values must differ. The first ten clauses define the active variables. Then for each gate, if the gate is active, one of its fanout gates must be active. In this example, this results in the last three clauses.

- Active Clauses, 37 literals:
$(\overline{F}_a + F_g + F_f)(\overline{F}_a + \overline{F}_g + \overline{F}_f)(\overline{H}_a + H_g + H_f)(\overline{H}_a + \overline{H}_g + \overline{H}_f)$
$(\overline{I}_a + I_g + I_f)(\overline{I}_a + \overline{I}_g + \overline{I}_f)(\overline{K}_a + K_g + K_f)(\overline{K}_a + \overline{K}_g + \overline{K}_f)$
$(\overline{M}_a + M_g + M_f)(\overline{M}_a + \overline{M}_g + \overline{M}_f)$
$(\overline{F}_a + H_a + I_a)(\overline{H}_a + K_a)(\overline{I}_a + M_a)$

Since the fault is a stuck-at-one, clauses are added to say that the fault site must have value zero in the good network and one in the faulty network, and that the gate with the fault is active. Finally, the objective is added that at least one of the reachable outputs must be active. This results in the following clauses.

- Fault Site and objective, 5 literals:
$(\overline{B}_g)(B_f)(F_a)\ (K_a + M_a)$

To describe this one fault from a six gate network takes 50 clauses containing 119 literals. This CNF formula is satisfiable by any assignment with $A = 0, B = 0, C = 1$, which detects the fault at output K. Using satisfiability, the entire CNF formula is extracted before a search is made for a test (this is a severe limitation to applying this technique directly to sequential test generation).

The overhead of generating a new formula for each fault does decrease the efficiency of TEGUS. Even for relatively small networks, the CNF formula can be many thousands of literals. Several of the CNF formulas for network c6288 have over 50 000 literals. As shown in Table 1, about half the total time is spent extracting the formulas.

Some of the time lost in extracting the formula is regained because the uniform CNF formula can be solved by a much simpler branch and bound algorithm. Another interesting benefit of extracting the formula is that gates which are not relevant to detecting a fault are not represented in the formula. In the previous example, gate N was not added to the formula because it cannot be used either to activate or propagate the fault on gate F. However, a structural test generation algorithm would propagate values to gate N when gate G changes. This benefit was canceled out in [24] by the inefficient heuristics used to solve the formula. Later, a structural algorithm used the same DFS tracing to create a reduced gate network for test generation[29], but with much less benefit since it does not simplify the branch and bound algorithm as it does in TEGUS.

## 4 Checking Satisfiability

After a CNF formula has been extracted for a fault, the next step is to find a satisfying assignment. If such a satisfying assignment exists, the set of values it assigns to the network primary inputs is a test for the fault. Finding an assignment is an NP-complete problem[14]. If no satisfying assignment exists, the fault is redundant. Proving this is a co-NP-complete problem[14].

This section describes the basic branch and bound algorithm used to search for a satisfying assignment or prove that none exists. The heuristics used to guide the branching are called a search strategy. Several greedy search strategies are proposed and compared for effectiveness and efficiency. As in structural algorithms [1, 11, 21, 31, 33], the best results are achieved by trying several strategies in succession.

For redundant faults, switching strategies is not effective. To identify hard redundant faults, we use two types of global implications first introduced for structural test generation in [38]. We show how more of these global implications can be generated by iterating the basic procedure, and how in TEGUS they subsume many of the other important heuristics used in structural test generation such as the X-path check in PODEM, and unique sensitization in FAN.

Based on the experiments described in this section, we identify four greedy strategies which form the core of the TEGUS heuristics. These are tried in succession with low backtrack limits to detect nearly all testable faults and the easy redundant faults. For the few remaining faults, the formula is analyzed for global implications and the same greedy strategies are retried with a higher backtrack limit to prove the difficult redundant faults are indeed redundant.

## 4.1 Branch and Bound

Branch and bound is a standard technique for the exact solution of NP-complete problems. In SAT, branching consists of selecting an unassigned variable, setting it either true or false, and reversing the assignment (backtracking) if the first choice does not lead to any solution.

The search is bounded in three ways. First, if an unsatisfied clause has no unassigned literals, then the current partial assignment is contradictory and the search can be bounded. Second, if an unsatisfied clause has only one unassigned literal, that literal can be immediately asserted. This avoids branching on the corresponding variable. Third, if the previous step tries to assert A but $\overline{A}$ has already been asserted, the assignment is contradictory and the search can be bounded. Pseudo code for the bound procedure is shown in Fig. 3.

In spite of bounding the search in this way, we still may not realize immediately that a partial assignment cannot be extended to satisfy the formula. This is what leads to the worst case performance. For example, given a partial assignment which uniformly requires 4 additional assignments to detect a contradiction, a total of $2^4$ partial assignments will be generated where ideally none would be required. Each of the heuristics described in this section can be seen as an to attempt minimize this explosion of the search space.

Like other test generation algorithms, TEGUS attempts to minimize the number of backtracks needed to generate a test. On the other hand, since the real objective is to minimize computation time, it is misleading to focus only on minimizing the number of backtracks. Using the previous example, a heuristic is not effective if it reduces the additional assignments from 4 to 0 but requires twice as much computation time as just doing the 16 backtracks.

## 4.2 Search Strategies

A set of heuristics used to guide the branch and bound search is called a search strategy. The heuristics for branch and bound can be categorized by three parameters (this is true for structural algorithms as well):

- Variable order for branching.
- What processing to do at each branch point (dynamic processing).
- How long to search before giving up (e.g. backtrack limit).

Variable order can be classified as static or dynamic. A static ordering fixes a single global order for all the variables and uses this for the entire search. The problem with a static ordering is that it generally does many unnecessary variable assignments. As explained previously, this exacerbates the worst case behavior of the search. For example setting one input of an AND gate to zero fixes the output to zero but does not force any value for the other inputs. Since these other inputs are unassigned, a fixed variable ordering may assign values to the other inputs, while the

```
branch (strategy)        // recursive binary branch and bound
    v = find_next_variable (strategy)
    if (none) return true

    sp = save_stacks ()
    if (bound(v,false) and branch(strategy)) return true

    if (++num_backtrack > backtrack_limit)
        gave_up = true
        return true
    end if
    undo_assign (sp)

    return (bound(-v,false) and branch(strategy))
end


bound (lit,add_nli)      // assign lit and bound the search on a contradiction
    if (opp(lit) == true) return false

    for each lit2 implied by lit
        if (bound(lit2,add_nli) == false) return false
    end for

    for each clause c containing lit
        mark c satisfied
    end for

    for each unsatisfied clause c containing -lit
        if c has no free literals return false
        if c has 1 free literal lit2
            if (bound(lit2,add_nli) == false) return false;
            if (add_nli) add implication (-lit2 => -lit)
        end if
    end for

    return true
end


find_next_var (strategy)    // select next variable for branching
    if (strategy == G1)
        return first free literal in first unsatisfied clause of subformula
    if (strategy == G2)
        return last free literal in first unsatisfied clause of subformula
    if (strategy == G3)
        return most frequent free literal in unsatisfied clauses of subformula
    if (strategy == G4)
        return first free literal in last unsatisfied clause of subformula
    if (strategy == G5)
        return last free literal in last unsatisfied clause of subformula
end
```

Figure 3: Algorithm for branch and bound on CNF formula.

| Network | LSAT | G1 | G2 | G3 | G4 | G5 | Comb | NLI | ABC | G1R | G2R | G3R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| c432 | 82 | 5 | 3 | 3 | 184 | 25 | 3 | 2 | 1 | 126 | 103 | 3 |
| c499 | 4 | 0 | 16 | 473 | 7 | 42 | 0 | 0 | 0 | 529 | 70 | 435 |
| c880 | 3 | 8 | 0 | 0 | 28 | 14 | 0 | 0 | 0 | 27 | 33 | 0 |
| c1355 | 3 | 0 | 40 | 0 | 0 | 73 | 0 | 0 | 0 | 630 | 41 | 115 |
| c1908 | 342 | 8 | 2 | 26 | 21 | 27 | 2 | 0 | 0 | 605 | 372 | 84 |
| c2670 | 19 | 29 | 23 | 93 | 111 | 81 | 23 | 5 | 7 | 1 095 | 702 | 81 |
| c3540 | 168 | 36 | 26 | 116 | 450 | 1 200 | 5 | 0 | 1 | 1 292 | 1 348 | 180 |
| c5315 | 400 | 43 | 28 | 277 | 626 | 318 | 0 | 0 | 0 | 805 | 344 | 452 |
| c6288 | 4 962 | 9 | 81 | 2 078 | 1 388 | 1 236 | 0 | 0 | 0 | 4 698 | 4 509 | 2 904 |
| c7552 | 842 | 233 | 70 | 418 | 312 | 472 | 28 | 0 | 0 | 1 409 | 1 563 | 621 |
| s1494 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 0 | 8 | 1 | 0 |
| s5378 | 0 | 0 | 0 | 43 | 26 | 13 | 0 | 0 | 0 | 119 | 25 | 6 |
| s9234 | 134 | 101 | 83 | 371 | 223 | 417 | 59 | 18 | 7 | 695 | 233 | 319 |
| s13207 | 5 | 2 | 26 | 131 | 328 | 559 | 0 | 0 | 0 | 177 | 37 | 0 |
| s15850 | 6 | 1 | 6 | 13 | 647 | 224 | 0 | 0 | 0 | 905 | 173 | 1 |
| s35932 | 51 | 0 | 0 | 0 | 293 | 4 | 0 | 0 | 0 | 1 | 6 | 4 |
| s38417 | 35 | 116 | 120 | 28 | 2 560 | 1 521 | 4 | 0 | 0 | 2 907 | 1 931 | 60 |
| s38584 | 0 | 7 | 0 | 43 | 523 | 546 | 0 | 0 | 0 | 991 | 108 | 87 |
| Total | 7 056 | 598 | 524 | 3 982 | 7 727 | 6 772 | 124 | 25 | 16 | 17 019 | 11 599 | 9 786 |
| Time | 4.2h | 0.4h | 0.4h | 3.0h | 0.5h | 0.5h | 0.5h | 14.5h | 14.1h | 0.5h | 0.4h | 2.7h |

Table 2: Aborted faults for 12 different strategies, backtrack limit 50, no fault simulation. For comparison, there are 121 029 total faults. The last line shows the total time for each strategy in CPU hours.

actual conflict may be because the gate has an output value of zero. If the gate has four inputs, then at least $2^3$ partial assignments are wasted effort.

The results in [24] use static variable ordering. Three strategies are used which sort the variables based on the number of implications they have. These strategies are described further in [26]. Column LSAT of Table 2 shows our results of using these static variable ordering strategies without fault simulation. This static ordering is not effective or efficient, and is more complicated than the orderings presented below. Other algorithms using a static variable ordering give equally poor results[9].

A dynamic ordering chooses which variable to branch on at each branch point of the search. Generally, the strategies used in structural test generation algorithms use dynamic variable ordering. With this flexibility, the problems of a static ordering can be avoided, but it can be equally inefficient to use a complex strategy for choosing each variable. To try to improve the performance of satisfying the CNF formula, we experimented with several greedy strategies.

## 4.3 Greedy Variable Selection

One possible greedy strategy is to pick any remaining unsatisfied clause in the formula and satisfy it by assigning one of its literals to be true. This is a dynamic variable ordering which is fast to compute, but its effectiveness depends on the order that the clauses are searched. The intuition for a good clause ordering is found by considering the variable orderings used in effective structural algorithms. Note that the clause ordering is a heuristic to improve the average performance, and is not necessary to make the algorithm complete.

The most important improvement of PODEM over the D-algorithm was to do the implicit search by only assigning primary inputs. All other gate values are then defined by implications. PODEM is generally faster than the D-algorithm because this avoids a large number of unnecessary conflicts, especially in networks with a lot of reconvergence. Thus it would be best to order the clauses so that those with literals related to primary inputs were searched first. In the example from section 3.4, consider the following clauses from the good network.

- Subformula, 25 literals:
  $(F_g + \overline{B}_g + \overline{C}_g)(G_g + \overline{C}_g + \overline{D}_g)(H_g + A_g + B_g + \overline{F}_g)(I_g + A_g + F_g + G_g)$
  $(F_f + \overline{B}_f + \overline{C}_g)(H_f + A_g + B_g + \overline{F}_f)(I_f + A_g + F_f + G_g)$

These are all the (n+1)-clauses (see section 3.4) which contain at least one literal from a primary input. If these clauses are searched first using a greedy strategy, values are assigned to primary inputs rather than to arbitrary network variables. For example, to satisfy the first clause, a greedy strategy could assign $F_g$ to true. Because of clauses $(B_g + \overline{F}_g)(C_g + \overline{F}_g)$ in the formula, this would directly force inputs B and C to one. On the other hand, it could choose to satisfy the formula by assigning $\overline{B}_g$ to true, which means input B is assigned to zero. When the CNF formula for a fault is extracted, these clauses containing a literal for a primary input are specially designated as a subformula, since there is no direct way to identify these special clauses by looking only at the final formula.

This raises a second problem which is how to order the clauses of the subformula, i.e. how to order the primary inputs. From the analysis at the beginning of the section, if a variable assignment is going to lead to a conflict, we want to detect this as soon as possible. In Fig. 2, after input A is assigned, a conflict is more likely to be detected by assigning B next instead of E since inputs A and B converge immediately while A and E do not converge at all. In a large network, this effect can lead to many wasted backtracks and it is one of the reasons the backtrack limits in [24] needed to be so high. Thus we order the clauses of the subformula using a depth-first search from the reachable primary outputs. In a structural algorithm, this heuristic is embedded in the backtracing heuristics used to select a primary input, and in the ordering of objectives.

The ordering derived from the DFS depends on how the fanins of each gate are ordered. We experimented with several different orderings based on traditional testability measures but did not find an ordering which was consistently superior. Thus we simply use the gate fanin ordering given in the input file, and the DFS searches the fanins for each gate in first to last order. Starting with output K, then M in Fig. 2, the DFS ordering of the inputs is A, B, C, D.

In addition to avoiding conflicts while searching for a test, these clause orderings have a second important advantage. The SAT solver continues the branch and bound search until every clause is satisfied, even if the current partial assignment happens to be a test. This is the only way the SAT solver can guarantee that the formula has indeed been satisfied. If primary inputs are not assigned first, the search could encounter conflicts and have to backtrack. This can cause the SAT solver to abort even though a test has been found. For example, in Fig. 2 assume that input D is actually driven by some complicated logic, and that variables $A_g, B_g$ and $C_g$ have already been assigned false, false and true respectively. This partial assignment is a test, but clause $(G_g + \overline{C}_g + \overline{D}_g)$ has not been satisfied yet. If the next decision is to set $G_g$ to true (the AND gate to a one), the solver may have to backtrack in order to justify a value of true for variable $D_g$. This is another reason why such high backtrack limits are needed in [24]. On the other hand, if assignments are only made to primary inputs by first satisfying the subformula, all these needless backtracks are avoided.

Although a generic SAT solver cannot know a test has been generated until the entire formula has been solved, it is possible to modify the SAT solver to monitor the active variables to determine when a test has been generated and terminate the branch and bound early. This could be important for dynamic test compaction, but would probably decrease the performance of the SAT solver.

## 4.4 Comparison of Strategies

This section presents experimental results comparing the heuristics proposed in [24], several variations on the greedy strategy just described, and three strategies which the previous analysis predicts will not work well. In order to compare only the heuristics for deterministic test generation, no fault simulation is used for the results in Table 2.

The algorithm in [24] uses three static variable orderings tried in succession. These heuristics are described in more detail in [26]. Column LSAT of Table 2 shows the results of applying all three static orderings, each with a backtrack limit of 50 (the results in [24] use a backtrack limit equal to the number of variables in the formula). After 4.2 hours of CPU time, the algorithm still aborted on over 7 000 faults. By increasing the backtrack limits, and by simulating random patterns to detect most of the faults, it is possible to get good coverage on the benchmark networks using the static variable orderings. However, as shown in [24], it is much less efficient than existing structural algorithms.

Next we consider several greedy strategies. Strategy G1 (greedy 1), searches the subformula in order for the first unsatisfied clause, and then selects the first unassigned literal in this clause. It branches on the corresponding variable by first trying the assignment which satisfies the clause. Column `G1` of Table 2 shows the number of aborted faults using this strategy with a backtrack limit of 50 and no fault simulation. It aborts on many fewer faults and is nearly an order of magnitude faster than the static variable ordering LSAT. Greedy strategy G2 selects the *last* unassigned literal in the first unsatisfied clause of the subformula. As shown in column `G2` of Table 2, this does approximately as well as G1.

At each branching point, strategy G3 selects the literal which satisfies the most clauses which are still unsatisfied in the entire formula (not just the subformula). Ties are broken by picking the first such literal using the given clause ordering. Column `G3` shows this strategy does much worse than G1 and G2, as well as being slower. Satisfying as many clauses as possible at each step tends to favor the variables associated with high fanout or high fanin gates. Branching early on these values does not seem to be effective, probably for the same reasons a static variable ordering is not.

Strategies G4 and G5 are similar to G1 and G2 respectively, except they search the subclauses in reverse order. Overall they do worse than G1 and G2. Since clauses in the subformula are ordered by DFS as previously described, the inputs at the end of the subformula usually do not all belong to the same cones of logic. Rather, they are whatever inputs were left over after the earlier cones were searched. Thus the inputs at the end of the subformula in general do not converge as directly if at all, and many backtracks are wasted changing assignments which are not the real cause of the contradiction.

To emphasize the importance of assigning input variables early, we ran three experiments which reversed the clause ordering, i.e. put clauses for network outputs first and clauses for inputs at the end, and then applied the greedy strategies to all the clauses in the formula. Columns `G1R`, `G2R` and `G3R` show the results of applying G1, G2 and G3 respectively to the reversed formula. In all three cases, this ordering makes the strategies much less effective. Even G3 is affected since it uses the clause ordering to break ties. For a structural algorithm, such a strategy corresponds to starting by assigning values to the network outputs. These experiments and others not shown in the table indicate that dynamic variable orderings which assign values to inputs in topological order are both more effective and efficient than static orderings or strategies which branch on internal network variables.

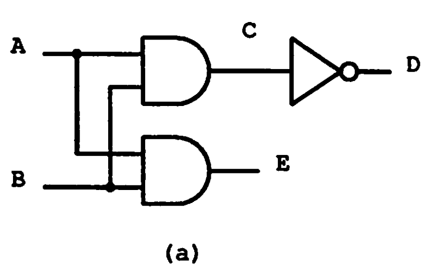
## 4.5 Orthogonal Strategies

Although the results in Table 2 for G1 and G2 are not bad (they abort on less than 0.5% of the faults without fault simulation in very reasonable time), current demands of test generation are for essentially no aborted faults. Increasing the backtrack limit for any of these strategies increases the coverage only very slowly and for enormous increase in computation time. But we can exploit the fact that each strategy may succeed on different faults by trying several strategies in succession.

One of the first suggestions to combine several different strategies is in [3]. Similar approaches are described in [11, 31, 33], and almost all algorithms have at least two or three different "phases" which use different search strategies. Of these, the work in [31] is particularly interesting since it develops an analytical expression to estimate when two strategies will be more efficient than one.

Min and Rogers defined the orthogonality of two search strategies [31] to measure how well two strategies can complement each other by detecting different sets of faults. Let S be the total set of faults in the networks used for the estimate. Let $S_A$ be the set of faults detected by strategy A (using some fixed number of backtracks) and likewise $S_B$ for strategy B. Then the orthogonality of A with respect to B is given by (12).

$$O_{A,B}(S) = \frac{|S_A - S_B|}{|S_A|} \tag{12}$$



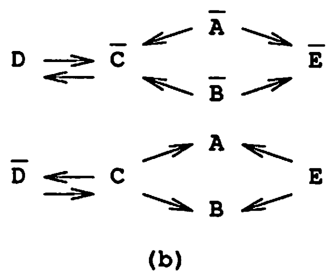


Figure 4: Use of implication graph. (a) Example network. (b) Corresponding implication graph.

Orthogonality as low as 0.026 can be enough to make two search strategies more cost effective than one[31]. Unfortunately the value of $O_{A,B}(S)$ decreases rapidly when the number of easy faults in the network increases regardless of how effective the strategies are on hard faults. This makes it difficult to compare different combinations of strategies using orthogonality. However, the concept can be used to explain the improved results from combining several greedy strategies.

Since G1 and G2 both give excellent results, a first combination is to try G1 followed by G2, each with a backtrack limit of 50. This reduces the number of aborted faults to 253, with negligible increase in CPU time. Since the other three greedy strategies each aborted on many thousand faults, it might seem that they would be of very little use. However, trying G4 or G5 after G1 and G2 further reduces the number of aborted faults. Even though they are not effective by themselves, G4 and G5 are useful because they are somewhat orthogonal to G1 and G2. On the other hand, G3 did not reduce the number of aborted faults significantly. Column Comb of Table 2 shows the number of aborted faults when strategies G1, G2, G4 and G5 are tried in succession. The number of aborted faults is reduced with a small decrease in performance.

The relatively high orthogonality of G1 and G2 can be explained by looking at how the clauses are formed and ordered. Consider the effect of G1 and G2 on the clauses for an AND gate in (9). If the 3-clause $(\overline{B} + \overline{C} + E)$ is not satisfied, strategy G1 will try to assign $\overline{B}$ which sets the output of the AND gate to zero. On the other hand, strategy G2 will try to assign E which sets the output of the gate to one. The orthogonality of G4 and G5 with respect to G1 and G2 is high because they start by assigning values to different primary inputs.

## 4.6 Solving Hard Formulas: Global Implications

A combination of greedy strategies still does not solve every CNF formula in the benchmark networks; strategy Comb still aborts on 124 faults. Almost all of them are redundant faults, so switching to yet another strategy or increasing the backtrack limit is not cost effective. For these remaining hard faults, additional information is needed to solve the CNF formula. Additional data used to improve the effectiveness of many test generation algorithms are called global implications because they are derived by analyzing the overall network structure (or formula structure in our case). The FAN algorithm introduced the use of dominators[13] to identify unique sensitization values as one type of global implication. Several additional global implications were introduced in [38]. This section describes the two global implications we have found most effective in TEGUS: nonlocal implications, and assignments by contradiction.

### Implication Graph

Each 2-clause in the CNF formula is equivalent to two implications.

$$(A \vee B) \Leftrightarrow (\overline{A} \rightarrow B) \wedge (\overline{B} \rightarrow A) \tag{13}$$

An implication graph is formed by associating a vertex with each literal in the formula, and letting the implications from 2-clauses define the edges. For example, the characteristic equation for the network in Fig. 4(a) has six 2-clauses.

$$(C+\overline{A}+\overline{B})(\overline{C}+A)(\overline{C}+B)(E+\overline{A}+\overline{B})(\overline{E}+A)(\overline{E}+B)(C+D)(\overline{C}+\overline{D}) \quad (14)$$

The corresponding implication graph is shown in Fig. 4(b). It is possible to analyze this graph to extract additional information about the formula. Strongly connected components (SCCs) of the graph indicate equivalent variables[24]. In Fig. 4(b), the two SCCs show that $D = \overline{C}$. This corresponds to recognizing buffers and inverters in the original network. This analysis cannot detect more interesting cases, such as determining that the outputs of the two AND gates are equivalent. Thus there is no real benefit from performing this check.

Another use of the implication graph is to find a literal X such that there is a path from X to $\overline{X}$ in the graph [8, 24], i.e. $X \Rightarrow \overline{X}$. In this case, $\overline{X}$ can be assigned true immediately without branching. Unfortunately, this condition does not usually occur until after several variables have been branched on, dynamically creating some additional 2-clauses. In the previous example, if A is assigned true, then the clause $(C+\overline{A}+\overline{B})$ is reduced to $(C+\overline{B})$, and the corresponding implications $\overline{C} \Rightarrow \overline{B}$ and $B \Rightarrow C$ can be added to the graph. In a more complicated example, adding such additional implications can create a path from X to $\overline{X}$ for some literal X. The major drawback is that at each branching point of the search, the implication graph must be temporarily updated using any new reduced clauses, and an $O(n^2)$ processing step must be used to detect contradictions in the graph. Any assignments found by analyzing the implication graph are dependent on the previous assignments, so if the search backtracks, the temporary implications must be removed from the graph, and any information derived from the graph must be discarded. This leads to the poor performance reported in [8].

We tried analyzing the implication graph for the more general condition $(X \rightarrow A) \wedge (X \rightarrow \overline{A}) \Rightarrow \overline{X}$, and tried to reduce the overhead using heuristics such as only analyzing the implication graph on every other branching point. This did not appreciably improve the effectiveness or the performance. Although other nonstructural algorithms [8, 24] place great emphasis on information derived from the implication graph, we found no benefits from using this costly procedure.

**Nonlocal Implications**

The two implications derived from each 2-clause can be considered local implications since in the original network they relate the input and output values of a single primitive, or the stem and branch values of a fanout point. While this locality makes it easy to construct the formula, more global information is needed to solve difficult CNF formulas, particularly for proving there are no solutions (i.e. a redundant fault). One form of global information is derived using (15). This rule was applied to only the good network values for structural test generation in [38], and applied to the entire CNF formula in [24]. For each literal A in the formula, assert A and check for condition (15).

$$(A \rightarrow B) \Rightarrow (\overline{B} \rightarrow \overline{A}) \quad (15)$$

Chains of local implications satisfy (15), but because they were derived from 2-clauses originally, the consequent implication is already represented in the implication graph. However if assigning A reduces some (3+)-clause to a 1-clause (F), i.e. $(A \Rightarrow F)$, then the implication $(\overline{F} \Rightarrow \overline{A})$ is likely to be new information. For example, in (16), the chain of implications from asserting A is listed in (17).

$$(\overline{A}+B)(\overline{B}+\overline{X})(\overline{B}+\overline{Y})(X+Y+\overline{F}) \quad (16)$$

$$A \Rightarrow B,\ \ B \Rightarrow \overline{X},\ \ \overline{X} \Rightarrow (Y+\overline{F}),\ \ B \Rightarrow \overline{Y},\ \ \overline{Y} \Rightarrow (\overline{F}) \quad (17)$$

In this example, literal A has 4 implication: $A \Rightarrow B$, $A \Rightarrow \overline{X}$, $A \Rightarrow \overline{Y}$, and $A \Rightarrow \overline{F}$. The first three implications resulted from chains of local implications and are ignored. However the conclusion $\overline{F}$ of the last implication came from the 3-clause $(X+Y+\overline{F})$, and so $F \Rightarrow \overline{A}$ is added to the formula.

```
branch_and_bound ()          // top level

    for i from 1 to 2

        if i == 2 then
            if (find_global_impl() == false) return false
        endif

        foreach strategy s in {G1,G2,G4,G5}
            gave_up = false
            result = branch(s)
            if (gave_up == false) return result
        end for

    end for

    return gave_up       // all strategies gave up
end


find_global_impl ()          // Find nonlocal impl, and assignments by contradiction
    foreach literal lit
        sp = save_stacks ()
        if (bound(lit,true) == false)
            if (bound(-lit,false) == false) return false
        else
            undo_assign (sp)
        end if
    end for
end
```

Figure 5: Trying several strategies in succession on the same formula.

Column NLI in Table 2 shows the results when nonlocal implications are computed for each formula before trying the strategies G1, G2, G4 and G5 in succession. The number of aborted faults is reduced by a factor of five, but the CPU time is increased 30 times. Since this is so time consuming, we first try the four greedy strategies without nonlocal implications. If all these strategies abort, then nonlocal implications are computed for all variables and the greedy strategies are retried, as shown in Fig. 5. With this approach, the total time is only slightly longer than that for strategy Comb. The backtrack limits used in [24] were much higher (up to 10 000) so nonlocal implications were not as useful as they are in TEGUS.

Nonlocal implications are more difficult to find in a structural algorithm because there are several types of implications: forward and backward implications, good and faulty network implications, and implications including the different fault effect paths (captured in TEGUS by the active variables). To make nonlocal implications as powerful as in TEGUS, a structural algorithm would have to handle all of these different cases. Thus the uniformity of the CNF representation is a tremendous advantage in the simplicity of the algorithm and the power of the heuristics.

## Assignments by Contradiction

Another type of global information is unique variable assignments found by contradiction. For each literal A in the formula, assert A and check for the implication of (18).

$$(A \rightarrow \text{false}) \Rightarrow \overline{A} \qquad (18)$$

To detect such an implication for literal A, assert A and apply the bounding steps described earlier. If this leads to a contradiction, then $\overline{A}$ is implied. Unlike the contradictions found using the implication graph, this also makes use of clauses with more than 2 literals, and consequently will find more assignments.

Column ABC of Table 2 shows the number of aborted faults when assignments by contradiction are applied in the same manner as described for nonlocal implications. Socrates applied (18) to structural test generation[38] but, as with nonlocal implications, it is more powerful when applied to the CNF formula since it is automatically applied to all literals in the formula.

**Iterating the Global Implications Computation**

In Socrates[38], global implications are searched for in two different phases. Initially the network is preprocessed to compute nonlocal implications for the good network. These implications are used in search phase DYN1. After this phase, any remaining faults are retried by phase DYN2 which computes global implications dynamically at each branching point of the search. Phase DYN2 is only needed for a few very difficult faults in the ISCAS networks. Similar results are reported in [29].

We have improved the global implication procedure in [38] after observing that the results of the implication search can depend on what order the variables are processed. As an example, consider the network in Fig. 2, for the fault stuck-at 1 on the first input of gate F (recall that in TEGUS, global implications for a formula are specific to one fault). Assume input C is processed first. Assigning $C_g$ to either true or false results in neither a conflict nor any nonlocal implications. Later input A is processed. When $A_g$ is true, variables $I_g, I_f, H_g$ and $H_f$ are all forced to false which forces $H_a$ and $F_a$ both to false. Then to satisfy clause $(\overline{F}_a + H_a + I_a)$, it must be that $F_a$ is false, which contradicts the objective clause $(F_a)$. Thus, by contradiction, $A_g$ must be assigned the value false. But now it is possible to find a nonlocal implication for C, namely that setting $C_g$ false forces $I_g$ to true, so the nonlocal implication $\overline{I}_g \Rightarrow C_g$ can be deduced. To find this nonlocal implication in one pass, the variable $A_g$ must be processed before $C_g$.

More complex ordering dependencies can be demonstrated, including interdependencies such that no ordering will find all the global implications on a single pass. Thus we iterate the processing until some pass results in no new implications. For the ISCAS benchmark networks, this usually terminates after 3 or 4 iterations with a contradiction, proving the fault is redundant.

It is likely that the benefits of dynamic processing shown in [38] were actually a result of this ordering dependence. Continuing the same example, assume that in the static phase C was processed before A, and that the basic strategies aborted on the fault. When the dynamic processing phase is invoked, if some variable besides C is selected as the first variable to branch on, then the dynamic processing at this first branch point will now detect the nonlocal implication $\overline{I} \Rightarrow C$. Although the implication was not detected as soon as possible, with luck it may still be soon enough to avoid some backtracking. There may be some examples where dynamic processing is helpful, but so far we have not found any cases which are not handled by iterating the static processing.

## 4.7 Comparison with Unique Sensitization

In this section we compare the various unique sensitization conditions used in structural test generation algorithms with the global implications computed by TEGUS. When both nonlocal implications and assignments are iterated to completion on a formula containing the active variables, the resulting global implications subsume the unique sensitization conditions used by structural algorithms.

PODEM used a very simple condition called an X-check: it makes sure there is at least one path from an active gate to a primary output, such that all the gates on this path have not been assigned a value yet. If there is no such path, this implies the fault effect cannot possibly be propagated to an output and the search can backtrack.

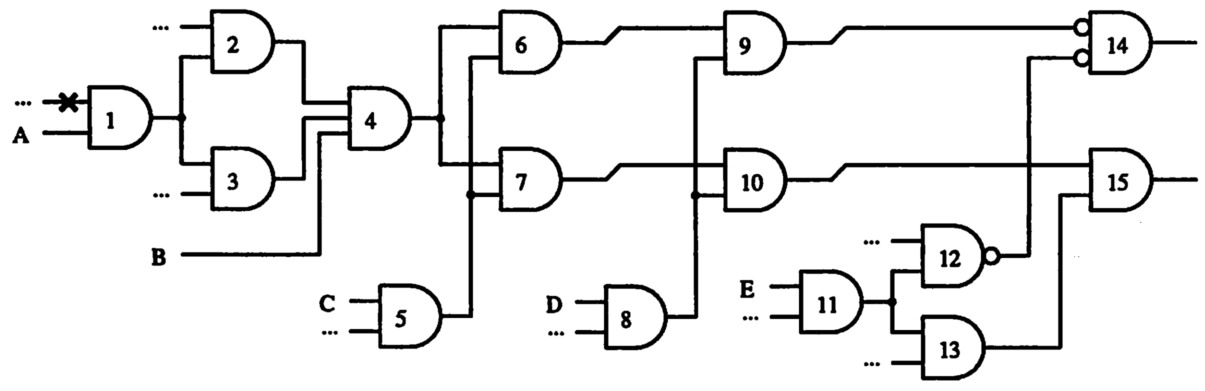


Figure 6: Example of configurations that different structural algorithms search for to determine unique sensitization values. TEGUS finds all of these using assignments by contradiction combined with active clauses.

FAN[13] and TOPS[23] analyze the network topology to identify dominator gates for each fault. Given gates G1 and G2, G2 is a dominator of G1 if all paths from G1 to a primary output pass through G2. In Fig. 6, for the input fault on gate 1 there are two dominator gates: 1 and 4. Because the fault effect must pass through these two gates, any inputs which cannot be reached from the fault site must have a noncontrolling value. Thus the FAN algorithm would immediately assign signals A and B to the value 1. During test generation, whenever the fault effect is only being propagated by a single gate, the dominators of that gate are checked for possible unique value assignments.

Socrates[39] finds additional unique assignments by checking if a dominator gate fans out to multiple gates which are in turn controlled by another single gate. In Fig. 6, dominator gate 4 fans out to gates 6 and 7, a *path dominator set*, which are both controlled by gate 5, the *path controller*. If the path controller cannot be reached from the fault site, then it also must be set to a noncontrolling value. In this example, Socrates would immediately set signal C to 1. FAN would not identify this since neither 6 nor 7 are dominators.

A further improvement made in [22] is to identify additional path dominator sets which are not direct fanouts of a dominator. Gates 9 and 10 form such a set, and consequently signal D must also be set to 1. Socrates would not find this because gates 9 and 10 do not have a common fanin which is on a fault path.

Yet a further extension is described in [29], where the path controller gate itself may fan out before reaching the fault paths. Gate 11 is a path dominator which fans out through gates 12 and 13 before reaching 14 and 15 which together dominate all paths from the fault site. If gates 12 and 13 are just inverters or buffers, the algorithm in [29] would determine that signal E must be set to 1. If these gates have other inputs, the unique assignment is not detected. The algorithm in [22] would not identify gates 14 and 15 as a path controller set because they have no common fanin gate.

In TEGUS, the assignments by contradiction described previously will determine every single literal X such that assigning $\overline{X}$ causes a contradiction. When active clauses are added to the CNF formula as described in section 3, these global implications subsume all possible dominator and path controller assignments, including the PODEM X-path check. To see this, consider the result of assigning $E_g$ false (since the global implications processing is applied to *every* literal in the CNF formula). This forces $12_g, 12_f$ to true, and $13_g, 13_f, 14_g, 14_f, 15_g$ and $15_f$ to false. The active variable for gate 14 is defined as follows (see section 3.3).

$$(\overline{14}_a + 14_g + 14_f)(\overline{14}_a + \overline{14}_g + \overline{14}_f)) \tag{19}$$

Since both $14_g$ and $14_f$ are false, the only way the first clause can be satisfied is for $14_a$ to be false, i.e. the fault effect does not propagate through gate 14, and likewise for gate 15. Now consider the following active path clauses generated by (11).

$$(\overline{4}_a + 6_a + 7_a)(\overline{6}_a + 9_a)(\overline{9}_a + 14_a) \tag{20}$$

Since $14_a$ is false, the third clause forces $9_a$ to false, and then the second clause forces $6_a$ to false. Similarly $10_a$ and $7_a$ will be forced to false because $15_a$ is false. Now by the first clause, $4_a$ is forced to false, and by similar arguments, $2_a$, $3_a$ and then $1_a$ are all forced to false. But this is a contradiction since one of the goal clauses states that the faulty gate must be active, i.e. $(1_a)$. A similar contradiction occurs if a partial assignment would eliminate the D-frontier, thus giving the same results as the PODEM X-path check.

This entire chain of implications is carried out by procedure `bound` of Fig. 3, and since a contradiction is a result, $E_g$ can be immediately assigned true. It is straightforward to show that this will also find the unique assignments for $A_g$, $B_g$, $C_g$ and $D_g$. In TEGUS, these are all determined using the simple procedure `find_global_impl` of Fig. 5, without any topological analysis of the network. Some of the dominator and path controller information may be derived more than once as different faults are processed, but since these global implications are only needed for a few faults, the overall performance is hardly affected.

### 4.8 Combined Heuristics

Pseudo code for the branch and bound algorithm to solve a CNF formula is listed in Fig. 3. At each level of recursion, some unassigned variable is selected and branched on. Function `bound` assigns a literal to be true and then bounds the search by following up with all direct implications of this assignment. If any contradiction is found, the branching is stopped and the most recent decision is reversed.

Fig. 5 describes how four basic greedy strategies are tried in succession. Given the CNF formula, the clauses of the subformula are put in DFS order and the four basic greedy strategies are tried in succession, each with a backtrack limit of 15. If all four strategies abort, global implications are computed for all literals in the formula and the four strategies are retried with a higher backtrack limit of 500. This is preferred to computing the global implications immediately for all formulas because, as shown in Table 2, most formulas can be solved without this very expensive procedure. In a practical test generation algorithm, a good balance between robustness and efficiency is crucial.

Since the four basic strategies might each be applied up to two times, TEGUS can be considered to have in effect 8 different search strategies. Because of the uniformity of the CNF formula, it is very straightforward to implement so many different strategies. One weakness of the current heuristics is that the backtrack limits are not adjusted for changes in problem size. Further experiments are needed to determine a good heuristic for scaling the backtrack limits. A constant limit works well for the ISCAS benchmark networks because the larger networks primarily get wider (more cones of logic) but not much deeper (more levels of logic). For example, the average CNF formula size for c2670 is 3 700 literals, while for s35932 it is only 630 literals, even though the latter has 16 times more gates. This is the basis for the misleading observation that "ATPG CPU time increases linearly with gate count" [43]. A more appropriate measure would be time versus the size of the subnetwork formed by the cones of logic containing the fault, which is what is measured by the CNF formula size.

## 5 Results

In this section, we compare the experimental results from our implementation of TEGUS to other results in the literature. Since the heuristics are a tradeoff of robustness, simplicity, and efficiency, we will consider each of these separately. From the results on the ISCAS benchmark networks, TEGUS is a good balance of these three factors.

For a reasonable evaluation of a new test generation algorithm, it is necessary to consider both the total time and number of aborted faults, each with and without fault simulation. The execution times should be the total run time, including all preprocessing. The results should be compared to the best known results, not to an attempted reimplementation of another algorithm. For comparison, the times should be normalized based on direct execution (not extrapolated from MIPS or other performance measures), and absolute times should be reported as well to allow future direct comparisons. Until a fundamental improvement in algorithm complexity is achieved, such experimental comparisons are crucial for comparing different algorithms. Unfortunately very few published results meet these conditions.

| Network | Time (sec) CNF | SAT | Total | Aborted Faults | Network | Time (sec) CNF | SAT | Total | Aborted Faults |
|---|---|---|---|---|---|---|---|---|---|
| c432 | 2. | 1. | 3. | 0 | c7552 | 62. | 23. | 86. | 0 |
| c499 | 8. | 3. | 11. | 0 | s1494 | 2. | 1. | 3. | 0 |
| c880 | 2. | 1. | 3. | 0 | s5378 | 7. | 2. | 9. | 0 |
| c1355 | 14. | 5. | 19. | 0 | s9234 | 32. | 15. | 48. | 0 |
| c1908 | 11. | 3. | 14. | 0 | s13207 | 46. | 8. | 56. | 0 |
| c2670 | 16. | 6. | 22. | 0 | s15850 | 100. | 25. | 125. | 0 |
| c3540 | 47. | 21. | 68. | 0 | s35932 | 48. | 11. | 61. | 0 |
| c5315 | 32. | 9. | 42. | 0 | s38417 | 145. | 46. | 193. | 0 |
| c6288 | 274. | 84. | 360. | 0 | s38584 | 77. | 18. | 98. | 0 |
|  |  |  |  |  | TOTAL | 930. | 280. | 1 220. | 0 |

Table 3: Applying TEGUS without fault simulation demonstrates its robustness. Extracting the formulas takes over 3 times as long as solving them, since no easy faults were dropped by random tests.

The final test set size is one factor we are not considering in this report. Techniques for test compaction can be applied to reduce the test set, but with some penalty in performance, and usually requiring a more complicated algorithm. As one extreme example, in [35] a test set of 13 patterns is generated for network s35932. This is six times smaller than the test set in Table 1, but it takes about 400 times more CPU time to generate this smaller test set. Because this tradeoff varies so widely, and since many compaction techniques are independent of the test generation heuristics, we will not compare the number of patterns generated by TEGUS except to note that it is about the same as other published results using no special compaction techniques (see Table 1).

## 5.1 Robustness

Algorithms such as the D-algorithm and PODEM are complete, meaning that given enough time, a test will be generated for each non-redundant fault. However, practical implementations of these algorithms add limits to prevent exorbitant amounts of time being spent on difficult faults. Commonly, a limit is placed on the number of backtracks allowed for each fault, i.e. the number of times the branching procedure is allowed to change a decision. Another approach is to impose a time limit for each fault (this can make it difficult to reproduce results on the same machine as system loads vary, much less across different machines). If such a limit is exceeded for a fault, the fault is *aborted*, i.e. neither detected nor proved redundant.

Decreasing the number of aborted faults, even by a small amount, is expensive. If not aborted, the time used for just one fault exhibiting the exponential worst case can dominate the total test generation time. Thus the first consideration in comparing two test generation algorithms is how many faults they abort on. It is generally not possible to prove the heuristics of one test generation algorithm will abort less frequently than another, so if one algorithm empirically aborts on fewer faults than another, we will call the former more *robust*.

Ideally, robustness measures how well an algorithm will do on examples not yet seen, as well as on the benchmark suite. When fault simulation is used (with or without random patterns), many algorithms have no aborted faults for the ISCAS '85 benchmarks, and several algorithms have none for the ISCAS '89 benchmarks. This is an unreliable evaluation of robustness because when fault simulation is used, tests are deterministically generated only for 5-10% of the faults. Since most of the ISCAS benchmarks have fewer than 4000 gates (not counting one-input gates), this does not provide enough data for a valid comparison. The smaller set of faults is also not easily reproduced, which can affect the results since different algorithms have trouble with different sets of faults, as shown in section 4.5. Thus algorithm robustness should be evaluated without using fault simulation.

Table 3 shows that without fault simulation or random patterns, TEGUS either detects or proves redundant every fault in the ISCAS benchmarks. Fig. 7 shows the corresponding distribution of backtracks. To be consistent with other reported results, this shows only the number of backtracks for the strategy which was successful. For example, if G1 and G2 both aborted on a formula after 25 backtracks and then G4 satisfied it with 3 backtracks, this is counted as 3

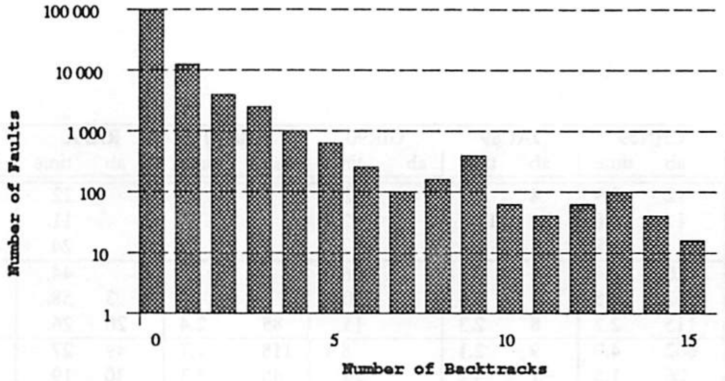


Figure 7: Distribution of backtracks for results of Table 3. Note the logarithmic scale.

backtracks instead of 53. Since nearly 98% of the faults took three or less backtracks, it is apparent that if a strategy is going to be successful it is usually successful quite early.

To evaluate the robustness of TEGUS, Table 4 compares it with seven other recent algorithms. No fault simulation was used for any of the results in this table. A reminder of the central idea for each algorithm follows; the references should be consulted for details. Algorithm `CHE88`[12] uses the D-algorithm with a modified 9-valued algebra. Algorithm `CHN89`[11] uses five different search strategies in succession, based on different testability measures such as SCOAP, CAMELOT, and random. Algorithm `JAC89`[22] uses an improved unique sensitization procedure in a 9-valued algebra. Algorithm `GIR90`[16] extends the concept of a D-frontier to an E-frontier of 0 and 1 values as well as D and N. These frontiers are stored in a hash table to reuse search state information across different faults. For `GIR91`[17] this algorithm was augmented with an improved backtrace procedure. Algorithm `ABR90`[1] uses a new testability measure GLOBAL which is an improvement on FAST and SCOAP. Finally, algorithm `RAJ90`[36] uses a 16-valued algebra to identify necessary assignments during the search.

Only two algorithms detect or prove redundant every fault in the ISCAS benchmark networks: TEGUS, and the EST algorithm by Giraldi and Bushnell[16, 17]. We reiterate that because of backtrack limits, all the algorithms in Table 4 are incomplete, so it should not be inferred that TEGUS or EST will never abort on any fault. Of the remaining five algorithms, four aborted on more faults than even the greedy strategies G1 or G2 from Table 2. Any more complicated algorithm which is not as robust as a single greedy strategy is of questionable value.

## 5.2 Efficiency

Assuming that two algorithms are equally robust, the second question is how they compare in efficiency. Tables 1 and 3 show the performance of TEGUS with and without random patterns respectively. To compare our results with other algorithms, for each algorithm A we performed the following experiment.

- Port the code for TEGUS to the same model of computer used for the published results of algorithm A.
- Run TEGUS on the ISCAS benchmark examples using the same options reported for algorithm A (e.g. with/without fault collapsing, with/without random patterns, with/without reverse pattern simulation).
- Normalize the reported times for algorithm A to those obtained for TEGUS, taking into account whether the reported times for A included preprocessing or fault simulation time.

In order to guarantee identical results for TEGUS across different machines, we used the portable pseudo random number generator described by Park and Miller[32]. The variety of DEC, Sun, Apollo, Amdahl and IBM computers used all supported 32 bit integers. Consequently even the results for random test generation are identical for TEGUS across all machines. In all cases, the native C compiler was used with full optimizations enabled. Variations caused

| | CHE88 | | CHN89 | | JAC89 | | GIR90 | | ABR90 | | RAJ90 | | GIR91 | | TEGUS | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Network | ab | time | ab | time | ab | time | ab | time | ab | time | ab | time | ab | time | ab | time |
| c432 | 7 | 1.3 | 12 | 2.9 | 4 | 2.1 | | 42. | | - | | 22. | | 12. | | 1.0 |
| c499 | | 0.5 | 17 | 6.8 | 20 | 11. | | 2.5 | | - | | 11. | | 13. | | 1.0 |
| c880 | | 0.9 | | 1.8 | | 2.6 | | 7.7 | | - | | 24. | | 13. | | 1.0 |
| c1355 | 128 | 2.4 | 26 | 2.4 | 48 | 9.1 | | 6.2 | | - | | 44. | | 7.1 | | 1.0 |
| c1908 | 82 | 2.7 | 252 | 4.5 | | 4.5 | | 150. | | - | 3 | 58. | | 18. | | 1.0 |
| c2670 | 43 | 1.4 | 115 | 2.2 | 8 | 2.3 | | 15. | 85 | 2.4 | 20 | 26. | | 9.7 | | 1.0 |
| c3540 | 62 | 1.2 | 662 | 4.0 | 9 | 2.1 | | 8.4 | 118 | 2.7 | 49 | 27. | | 10. | | 1.0 |
| c5315 | | 1.6 | 26 | 1.5 | 1 | 4.2 | | 23. | 45 | 3.3 | 30 | 19. | | 15. | | 1.0 |
| c6288 | 231 | 1.2 | 4 | 1.5 | 514 | 8.1 | | 10. | 4 | 0.9 | 1127 | 21. | | 11. | | 1.0 |
| c7552 | 245 | 2.9 | 94 | 1.9 | 89 | 5.3 | | 23. | 156 | 2.8 | 39 | 29. | | 15. | | 1.0 |
| s1494 | | - | | - | | - | | - | | - | | - | | - | | 1.0 |
| s5378 | | - | | - | | - | | - | | - | | - | | - | | 1.0 |
| s9234 | | - | | - | | - | | - | | - | | - | | - | | 1.0 |
| s13207 | | - | | - | | - | | - | | - | | - | | - | | 1.0 |
| s15850 | | - | | - | | - | | - | | - | | - | | - | | 1.0 |
| s35932 | | - | | - | | - | | - | | 310. | | - | | - | | 1.0 |
| s38417 | | - | | - | | - | | - | | - | | - | | - | | 1.0 |
| s38584 | | - | | - | | - | | - | | - | | - | | - | | 1.0 |
| Total | 798 | 1.6 | 1208 | 2.2 | 693 | 6.6 | 0 | 16. | 408 | 30. | 1268 | 24. | 0 | 12. | 0 | 1.0 |

Table 4: Results without using any fault simulation (dash indicates not available). For each algorithm, column `ab` is the number of aborted faults, column `time` is the execution time normalized to TEGUS. Backtrack limits range from 25 for TEGUS to $2 \times 10^6$ for `GIR90`.

| Network | SIM89 | SCH89 | MIN89 | LAR89 | JAC89 | SCH90 | WAI90 | MAH90 | CHK91 | GIR91 | MAT92 | TEGUS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| c432 | 15. | 0.7 | 4.0 | 9.0 | 3.2 | 1.0 | 0.6 | 1.1 | 2900. | 14. | 0.5 | 1.0 |
| c499 | 49. | 1.1 | 3.9 | 11. | 6.3 | 1.7 | 0.4 | 2.9 | - | 11. | 0.4 | 1.0 |
| c880 | 60. | 1.6 | 7.3 | 65. | 5.4 | 2.4 | 0.8 | 0.6 | - | 25. | 0.7 | 1.0 |
| c1355 | 51. | 1.8 | 9.9 | 17. | 4.9 | 2.7 | 0.9 | 5.1 | - | 20. | 1.1 | 1.0 |
| c1908 | 76. | 2.9 | 14. | 52. | 5.9 | 3.2 | 1.0 | 2.9 | - | 44. | 0.9 | 1.0 |
| c2670 | 20. | 1.0 | 12. | 37. | 4.9 | 1.4 | 0.5 | 3.7 | - | 12. | 0.3 | 1.0 |
| c3540 | - | 1.2 | 5.4 | 37. | 3.6 | 4.0 | 0.6 | 1.4 | - | 18. | 0.7 | 1.0 |
| c5315 | - | 2.1 | 16. | 32. | 13. | 3.9 | 1.0 | 1.9 | - | 59. | 1.1 | 1.0 |
| c6288 | - | 1.3 | - | 15. | 1.8 | 1.4 | 0.5 | 2.8 | - | 13. | 0.5 | 1.0 |
| c7552 | - | 2.5 | 40. | 36. | 12. | 7.6 | 0.7 | 6.7 | - | 38. | 0.5 | 1.0 |
| s1494 | - | - | - | - | - | 12. | 3.5 | - | - | - | 1.7 | 1.0 |
| s5378 | - | - | - | - | - | 11. | 2.5 | - | - | - | 1.5 | 1.0 |
| s9234 | - | - | - | - | - | 65. | 1.1 | - | - | - | 1.0 | 1.0 |
| s13207 | - | - | - | - | - | 45. | 0.9 | 6.5 | - | - | 0.8 | 1.0 |
| s15850 | - | - | - | - | - | 17. | 0.8 | 5.0 | - | - | 0.7 | 1.0 |
| s35932 | - | - | - | - | - | 38. | 1.0 | 6.4 | - | - | 1.1 | 1.0 |
| s38417 | - | - | - | - | - | 13. | 1.0 | 17. | - | - | 0.9 | 1.0 |
| s38584 | - | - | - | - | - | 120. | 1.6 | 15. | - | - | 1.3 | 1.0 |
| Total | 30. | 1.7 | 22. | 31. | 6.8 | 36. | 1.0 | 9.4 | 2900. | 25. | 0.9 | 1.0 |

Table 5: Execution time when random patterns are included, normalized to TEGUS (dash indicates not available). Each algorithm uses slightly different limits for terminating the random pattern phase, and for aborting the deterministic test generation for a fault.

by changes in the compilers and operating systems since the original results were reported could not be avoided since these details are rarely available. Also, since fault simulation was used for Table 5, deterministic test generation was not applied to the same set of faults in all algorithms. The total times for TEGUS on the different machines used in these experiments varied by a factor of 17, indicating how meaningless it is to compare the unnormalized CPU times.

Tables 4 and 5 show the results of these experiments. The algorithms in Table 4 and JAC89, GIR91 of Table 5 were mentioned previously. Algorithm SIM89[41] uses the general problem solving mechanism built into Prolog, augmented with special guidance procedures called demons. It is not clear if any redundant faults were actually proved redundant, nor why results are not provided for the larger networks. Algorithm SCH89[38] is the version of Socrates which includes nonlocal implications, both static and dynamic. The reported times do not include preprocessing, which can be significant in structural algorithms. Algorithm MIN89[31] uses a combination of two search strategies, one of which tries to justify values first and then propagate the fault effect, the other tries propagation first. With a backtrack limit of 100, there are still 16 aborted faults which is close enough to be comparable. Algorithm LAR89[24] is Larrabee's SAT-based algorithm. Column SCH90 shows the results from running the program Socrates 4.0[15]. This comparison uses the total run time including preprocessing. Algorithm WAI90[43] is an improved implementation of several existing heuristics, primarily those from Socrates[38]. Algorithm MAH90[28] revisits network partitioning to improve performance, as described earlier in [4]. Algorithm CHK91[8] applies some of the implication graph computations from [24] to a structural algorithm. It is understandable why results are only available for the smallest benchmark network. Algorithm MAT92[29] improves the unique sensitization procedure from [22], and adds the subnetwork extraction technique from [24] to improve performance.

Although many of these report improved performance over other algorithms, it is not clear on what data these claims are based. Some do not use the ISCAS benchmark networks [9, 10]. Other reports either make no experimental comparison [8, 24, 29, 36, 38, 39, 41, 43], or compare against other heuristics of their own implementation [1, 2, 13, 16, 17, 18, 23, 25, 28, 31]. Neither of these approaches give a meaningful comparison. A noteworthy exception is Cheng[12] who reported absolute run times, permitting future comparisons with his results, and also ran the new algorithm on the same computer as in [11] for accurate normalized comparison with previous work.

From these tables, TEGUS performance is as good as the best results published for structural algorithms. It is important to note that the same strategies and backtrack limits were used for TEGUS in both tables. Since some algorithms were not run on the complete set of networks, the total times must be compared with care. TEGUS is 12 times faster than the one algorithm which has been shown to be as robust. Unfortunately, for the two algorithms WAI90 and MAT92 which are as efficient as TEGUS, there are no results without using fault simulation to compare their robustness.

The dynamic ordering of primary input variables used by TEGUS is the main reason it is faster than the heuristics used by Larrabee. The improved global implications procedure also makes it more effective. Overall the performance of TEGUS is very close to the best published results for structural algorithms. Although CPU time is the primary concern, the memory requirements are also quite modest as shown in Table 1. Thus even with the overhead of generating the CNF formulas, TEGUS is practical and competes with the best structural methods for combinational test generation.

## 5.3 Simplicity

We claim as well an advantage of simplicity for TEGUS. Simplicity of an algorithm is very subjective, but it is important to consider since it balances the tradeoffs of efficiency and robustness. Many of the algorithms in Tables 4 and 5 can probably be made more efficient or robust, but usually requiring a more complicated implementation. For example, the authors of algorithm GIR91[17] claim that their algorithm is 5.81 times faster than Socrates[38]. However, as Table 5 makes evident, they do not actually compare with the published results for Socrates, but against another algorithm they implemented which they believe is similar to Socrates. When GIR91 is compared directly to SCH89 taking into account the difference in performance of the two computers used, Socrates is shown to be 13.5 times faster. If it is even possible to improve the performance of algorithm GIR91 by nearly two orders of magnitude (the factor needed to make GIR91 5.8 times faster than SCH89), it will require a carefully optimized, and probably more complicated, implementation. Similar comments apply to the other algorithms, which range from 1.1 to 3200 times slower than the fastest algorithm, MAT92. Thus the simplicity of an algorithm is an important factor.

The only operation in TEGUS is that of assigning a value to a binary variable, so the branch and bound algorithm is much simpler than for structural algorithms, as shown in Fig. 3. By using separate binary variables for the good and faulty values of a gate, the algorithm is equivalent to the 9-valued algebras used in structural algorithms. The greedy heuristics are generally simpler than the testability measures and backtrace procedures used with other algorithms. The computation of global implications is also much simpler using the CNF formula, and subsumes all the unique sensitization conditions used in structural algorithms.

Our implementation of TEGUS is about 3 000 lines of C code. Of this, about 800 lines implement the general SAT package for constructing and solving a CNF formula. Much of this is for efficient memory management, since a different CNF formula is extracted for each fault and this must be carefully optimized (consequently making it more complex). About 300 lines are for the code which specifically extracts a formula for a stuck-at fault. The fault simulator is about 80 lines. The rest of the code is for program control and I/O. TEGUS can be applied to other fault models by changing the fault extraction code. Given a characteristic equation for the fault and the network elements, this is relatively straightforward, which is an advantage of using the generic SAT problem formulation. In contrast, consider the extensive changes which must be made to a conventional algorithm when the fault model or network elements are changed, such as the switch-level test generation algorithm in [27].

TEGUS does not require many of the heuristics added to structural algorithms. It does not use testability measures, multiple backtracing, special processing of fanout-free regions, head lines, dominators, or dynamic global implications. Some of these may improve the performance of TEGUS a small amount, but the greatest gain would be to speed up the formula extraction. Using a cache or some other heuristic to reuse parts of the formula might save some time, but would increase the memory usage, and make the implementation more complicated. Overall, the current implementation is a good balance of robustness, efficiency and simplicity.

## 6 Conclusions

We have described TEGUS, an algorithm for combinational test generation based on satisfiability. Applying branch and bound to the CNF formula results in a simple, elegant framework for test generation which makes it easy to experiment with different heuristics. This is to be compared with other algorithms which must use multiple backtracing, testability measures, special cases for different gates, 5-, 9- or 16-valued algebras, etc. Using our heuristics, this approach is also competitive with the best structural algorithms.

We have identified several greedy search strategies using dynamic variable ordering that, when used in concert, solve the CNF formulas derived for combinational test generation much more efficiently than Larrabee's static heuristics. Formulas for hard faults are solved efficiently using an improved static global implications procedure. This can also be applied dynamically although we have not found examples where this is helpful.

Based on the results in this report, TEGUS is an excellent balance of robustness, simplicity and efficiency. It is also robust; as shown in Table 4, TEGUS either detects or proves redundant every fault in the ISCAS benchmark networks *without using fault simulation*, and furthermore does so more efficiently than previously published results. Doing this comparison without fault simulation is essential to determine the real strength of the deterministic algorithm, and it is unfortunate that most recent algorithms have been published without this information.

From Table 5, overall TEGUS performance is comparable to the best published results for structural algorithms in spite of the overhead of extracting a new CNF formula for each fault. Since all existing algorithms for deterministic test generation have the same worst case time complexity, accurate comparisons of average case performance must be measured using a real implementation. A reliable comparison must take into account differences in computer performance and should be based on a direct comparison with existing results, not on extrapolations from a related algorithm implemented by the experimenters.

The TEGUS algorithm would be improved by finding good heuristics to scale the backtrack limits based on formula size. Since over half the time is spent extracting a new CNF formula for each fault, finding more efficient methods for doing this would greatly benefit TEGUS.

## Acknowledgements

We thank D. Bultman, A. Ghosh, S. Jarriel, R. Neher, J. Patterson and S. Stephan for their generous assistance in collecting the data used for Tables 4 and 5. We thank Tracy Larrabee for several enlightening discussions on her new formulation of the testing problem. This work was supported by the Semiconductor Research Corporation under grant 92-DC-008, and by various grants from DEC, IBM, Intel, Motorola, AT&T and BNR.

## References

[1] M. Abramovici, D. T. Miller, and R. Henning. Global cost functions for test generation. In *Proc. Int'l Test Conf.*, pages 35–43, 1990.

[2] M. Abramovici, D. T. Miller, and R. K. Roy. Dynamic redundancy identification in automatic test generation. In *Proc. Int'l Conf. Computer-Aided Design*, pages 466–469, 1989.

[3] V. D. Agrawal, S. C. Seth, and C. C. Chuang. Probabilistically guided test generation. In *Proc. Int'l Symp. Circuits and Systems*, pages 687–690, 1985.

[4] P. S. Bottorff, R. E. France, N. H. Garges, and E. J. Orosz. Test generation for large logic networks. In *Proc. 14th Design Automat. Conf.*, pages 479–485, June 1977.

[5] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *Proc. Int'l Symp. Circuits and Systems*, pages 1929–1934, May 1989.

[6] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN. In *Proc. Int'l Symp. Circuits and Systems*, pages 663–698, June 1985.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, 1986.

[8] S. T. Chakradhar and V. D. Agrawal. A transitive closure based algorithm for test generation. In *Proc. 28th Design Automat. Conf.*, pages 353–358, 1991.

[9] S. T. Chakradhar, V. D. Agrawal, and M. L. Bushnell. Automatic test generation using quadratic 0-1 programming. In *Proc. 27th Design Automat. Conf.*, pages 654–659, 1990.

[10] S. T. Chakradhar, M. L. Bushnell, and V. D. Agrawal. Automatic test generation using neural networks. In *Proc. Int'l Conf. Computer-Aided Design*, pages 416–419, 1988.

[11] S. J. Chandra and J. H. Patel. Experimental evaluation of testability measures for test generation. *IEEE Trans. Computer-Aided Design*, 8(1):93–97, Jan. 1989.

[12] W.-T. Cheng. Split circuit model for test generation. In *Proc. 25th Design Automat. Conf.*, pages 96–101, 1988.

[13] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Trans. Comput.*, C-32(12):1137–1144, Dec. 1983.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[15] A. Ghosh. personal communication, June 1992.

[16] J. Giraldi and M. L. Bushnell. EST: The new frontier in automatic test pattern generation. In *Proc. 27th Design Automat. Conf.*, pages 667–672, June 1990.

[17] J. Giraldi and M. L. Bushnell. Search state equivalence for redundancy identification and test generation. In *Proc. Int'l Test Conf.*, pages 184–193, 1991.

[18] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Trans. Comput.*, C-30(3):215–222, Mar. 1981.

[19] L. H. Goldstein. Controllability/observability analysis for digital circuits. *IEEE Trans. Circuits and Systems*, CAS-26:685–693, Sept. 1979.

[20] A. Ivanov and V. K. Agarwal. Dynamic testability measures for ATPG. *IEEE Trans. Computer-Aided Design*, 7(5):598–608, May 1988.

[21] A. Ivanov and V. K. Agrawal. Testability measures – what do they do for ATPG? In *Proc. Int'l Test Conf.*, pages 129–138, 1986.

[22] R. Jacoby, P. Moceyunas, H. Cho, and G. Hachtel. New ATPG techniques for logic optimization. In *Proc. Int'l Conf. Computer-Aided Design*, pages 548–551, Nov. 1989.

[23] T. Kirkland and M. R. Mercer. A topological search algorithm for ATPG. In *Proc. 24th Design Automat. Conf.*, pages 502–508, June 1987.

[24] T. Larrabee. Efficient generation of test patterns using boolean difference. In *Proc. Int'l Test Conf.*, pages 795–801, 1989.

[25] T. Larrabee. A framework for evaluating test pattern generation strategies. In *Proc. Int'l Conf. on Computer Design*, pages 44–47, 1989.

[26] T. Larrabee. *Efficient Generation of Test Patterns Using Boolean Satisfiability*. PhD thesis, Stanford University, Feb. 1990.

[27] K. J. Lee, C. A. Njinda, and M. A. Breuer. SWiTEST: A switch level test generation system for CMOS combinational circuits. In *Proc. 29th Design Automat. Conf.*, pages 26–29, June 1992.

[28] U. Mahlstedt, T. Grüning, C. Özcan, and W. Daehn. CONTEST: A fast ATPG tool for very large combinational circuits. In *Proc. Int'l Conf. Computer-Aided Design*, pages 222–225, Nov. 1990.

[29] Y. Matsunaga and M. Fujita. A fast test pattern generation for large scale circuits. In *Proc. Synth. and Simulation Meeting and Int'l Interchange*, pages 263–271, Apr. 1992.

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

[31] H. B. Min and W. A. Rogers. Search strategy switching: An alternative to increased backtracking. In *Proc. Int'l Test Conf.*, pages 803–811, Aug. 1989.

[32] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Commun. of the ACM*, 31(10):1192–1201, Oct. 1988.

[33] S. T. Patel and J. H. Patel. Effectiveness of heuristic measures for automatic test pattern generation. In *Proc. 23th Design Automat. Conf.*, pages 547–552, 1986.

[34] N. D. Phillips and J. G. Teller. Efficient event manipulation: The key to large scale simulation. In *Proc. Int'l Test Conf.*, pages 266–273, 1978.

[35] I. Pomeranz, L. N. Reddy, and S. M. Reddy. COMPACTEST: A method to generate compact test sets for combinational circuits. In *Proc. Int'l Test Conf.*, pages 194–203, 1991.

[36] J. Rajski and H. Cox. A method to calculate necessary assignments in algorithmic test pattern generation. In *Proc. Int'l Test Conf.*, pages 25–34, 1990.

[37] J. P. Roth. Diagnosis of automata failures: A calculus and a method. *IBM Journal Res. and Dev.*, 10:278–291, July 1966.

[38] M. Schulz and E. Auth. Improved deterministic test pattern generation with applications to redundancy identification. *IEEE Trans. Computer-Aided Design*, 8(7):811–816, July 1989.

[39] M. H. Schulz, E. Trischler, and T. M. Sarfert. SOCRATES: A highly efficient automatic test pattern generation system. *IEEE Trans. Computer-Aided Design*, 7(1):126–137, Jan. 1988.

[40] F. F. Sellers, Jr., M. Y. Hsiao, and L. W. Bearnson. Analyzing errors with the boolean difference. *IEEE Trans. Comput.*, C-17(7):676–683, July 1968.

[41] H. Simonis. Test generation using the constraint logic programming language CHIP. In G. Levi and M. Martelli, editors, *Proc. 6th Int'l Conf. on Logic Programming*, pages 101–112. MIT Press, June 1989.

[42] J. J. Thomas. Automated diagnostic test programs for digital networks. *Computer Design*, pages 63–67, Aug. 1971.

[43] J. Waicukauski, P. Shupe, D. Giramma, and A. Matin. ATPG for ultra-large structured designs. In *Proc. Int'l Test Conf.*, pages 44–51, Aug. 1990.

[44] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, and T. McCarthy. Fault simulation for structured VLSI. *VLSI Systems Design*, 6(12):20–32, Dec. 1985.