Combinational Test Generation Using Satisfiability

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

Abstract — **We present a robust and efficient algorithm for combinational test generation using a reduction to satisfiability (SAT). The algorithm, TEGUS, has the following features. We choose a form for the test set characteristic equation which minimizes its size. The simplified equation is solved by an algorithm for SAT using simple, but powerful, greedy heuristics, ordering the variables using depth-first search and selecting a variable from the next unsatisfied clause at each branching point. For difficult faults the computation of global implications is iterated, which finds more implications than previous approaches and subsumes structural heuristics such as unique sensitization. Without random tests or fault simulation, TEGUS completes on every fault in the ISCAS networks, demonstrating its robustness, and is 11 times faster for those networks which have been completed by previous algorithms. Our publicly available implementation of TEGUS can be used as a base line for comparing test generation algorithms; we present comparisons with 45 recently published algorithms. TEGUS combines the advantages of the elegant organization of SAT-based algorithms, such as that by Svanæs, with the efficiency of structural algorithms, such as the D-algorithm.**

I Introduction

In 1966, Roth presented the D-algorithm [1] for combinational test generation which he proved complete,[1] meaning that if a test for a fault exists, the D-algorithm will find it if run to completion. All complete algorithms developed since have the same worst case complexity, differing only in the heuristics used to optimize average case performance [2]. Until a breakthrough in algorithm complexity is achieved, test generation algorithms must be evaluated using more detailed, empirical comparisons. Since practical test generation algorithms are incomplete, we call an algorithm more *robust* if it empirically completes on more faults using a suitable set of examples. For algorithms of comparable robustness, efficiency is important, especially execution time and memory usage. Finally, one algorithm may be much simpler than another.

While most practical algorithms like the D-algorithm are structural, operating directly on the gate network, the theoretical analysis of test generation is based on transformations to and from other difficult problems. The proof by Ibarra and Sahni [3] that combinational test generation is NP-complete uses a polynomial-time reduction from the logic problem *satisfiability* (SAT) to test generation. By Cook's results [4], this also defines a polynomial-time reduction from test generation to SAT, as well as to the hundreds of other NP-complete problems [2] such as traveling salesman, integer programming, bin packing, and instances of quadratic programming. Although test generation can be solved by algorithms using any of these reductions without changing its asymptotic complexity, a structural algorithm avoids the overhead of doing a reduction.

Most nonstructural algorithms use a reduction to SAT or its dual, referred to collectively as *SAT-based* algorithms, because the reduction is straightforward, SAT is a fundamental problem which has been widely studied, and the formulation as a set of characteristic functions directly fits the paradigm of logic programming. The overall approach of existing SAT-based algorithms [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17] is to translate the D-algorithm problem formulation into a characteristic equation (in product-of-sums form for SAT, or sum-of-products form for the dual), and then solve the equation using a branch-and-bound search.

For example, in an algorithm by Svanæs [5], the D-calculus logic function of each gate type is defined using clauses in PROLOG, the topology of the network is translated into clauses, a fault is specified as a D or $\overline{\mathrm{D}}$ on some net, and a test is generated by asking the PROLOG solver if it is possible for some output to have a value of D or $\overline{\mathrm{D}}$. This algorithm is simple and elegant, but inefficient. The other SAT-based algorithms improve or worsen the average case performance by adding various heuristics, but the fastest is still orders of magnitude slower than the best structural algorithms. Many are also not robust, requiring the use of random patterns to get good fault coverage.

We present a new SAT-based algorithm which is both robust and efficient. We choose a form for the characteristic equation to reduce its size, and use a new heuristic of ordering the equation clauses in DFS order. The simplified equation is solved using fast, greedy heuristics for dynamically choosing the next variable at each branch point. Finally, for difficult faults we iterate the computation of global implications, subsuming the various unique sensitization, path controller, and global implication heuristics used in other algorithms.

With these simple heuristics, our algorithm TEGUS (TEst Generation Using Satisfiability) is more robust than previous algorithms and is one of the most efficient. We compare TEGUS with other algorithms published in the past seven years, using the ISCAS benchmarks [19, 20]. We show that TEGUS completes on every fault in these networks without using fault simulation, and is 11 times faster for those which have been completed by previous algorithms. With fault simulation TEGUS performance is as good as the best structural algorithms (unfortunately robustness data is not available for these), and is orders of magnitude faster than previous nonstructural algorithms. The results show that this approach to test generation is practical as well as conceptually elegant.

[0]Manuscript submitted November 13, 1992; revised January 24, 1994, and August 31, 1994. This work was supported in part by the Semiconductor Research Corporation under Grant 92-DC-008 and by various grants from DEC, IBM, Intel, Motorola, AT&T, and BNR.

The authors are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, 94720.

IEEE MS Number D1447-R2.

[1]A test generation algorithm *completes* on (a problem instance defined by) a fault if it either generates a test for the fault or proves it redundant, *completes* on a network if it completes on every modeled fault in the network, and is *complete* if it completes on all networks.

II Simplified Equation

A SAT-based algorithm implements test generation by solving a characteristic equation for all tests for a fault. In an efficient algorithm, the time to construct these equations often exceeds the time needed to solve them, so in TEGUS the overhead is reduced by using a two step problem reduction, and by choosing a minimized form for the characteristic function of each gate.

As a one time overhead, the gate network is translated into a network of AND gates with inverted inputs such that every fault in the original network has an equivalent fault in the reduced network. Each characteristic equation is then constructed from the AND gate network. Since every gate has the same logic function, this step is now simpler and faster. The intermediate reduction also simplifies fault simulation, improving its performance.

The second improvement is to simplify the form of the SAT equation derived from the D-algorithm [1]. For each fault, the goal is to find an input vector which defines a *D-chain* from the fault site to a network output. Previous SAT-based algorithms derive each characteristic equation by a straightforward translation of the *singular cover* (used for all gates) and *primitive D-cubes* (used for gates on a potential D-chain). The resulting equations are larger than necessary because the D-algorithm characteristic functions contain redundant information which simplifies a structural algorithm but which generally has no benefits for a SAT-based algorithm.

To simplify the equation, we start from the high level description of the D-algorithm rather than from the cube covers. Let $G(\bar{x})$ denote the logic function for a gate G with inputs $\bar{x} = \{x_1, \ldots, x_n\}$ and output G, let G_d be a binary variable which implies gate G is in a D-chain to a network output, and let X_f and X_g denote the values of a net X with and without the fault, respectively. For a gate G with fanout gates $H_i, i = 1 \ldots k$, the D-algorithm characteristic function can be expressed as

$$G_g = G(\bar{x}_g) \;\wedge\; G_f = G(\bar{x}_f) \;\wedge\; G_d \rightarrow (G_g \neq G_f) \;\wedge\; G_d \rightarrow (H_{1,d} \vee \ldots \vee H_{k,d}). \quad (1)$$

Roth [1] derived alternative ways to express the D-chain condition; this choice is one of the main differences between the various SAT-based algorithms. The form chosen here is most efficient with the greedy branch-and-bound heuristics in the next section because it includes the forward D-chain implications. If G is not on any path from the fault to an output, (1) can be simplified to $G_g = G(\bar{x}_g) \wedge (G_f = G_g)$.

Minimizing each component function reduces the overall characteristic equation length. For example, in Fig. 1, assume that all inputs of gate J are on potential D-chains. With Larrabee's reduction [14], the 3SAT characteristic function for J has 67 literals [21]. The reduction in [17] uses a slightly different expression of the D-chain condition, replacing the second implication in (1) with logical equivalence (adding these backward D-chain implications seems to have no benefits because the fault site is known a priori to be on any final D-chain). The corresponding 3SAT function has 51 literals [21]. The simplified characteristic function used in TEGUS is found by substituting the gate logic function $J = \overline{H}\,\overline{I}\,E$ into (1) and expressing it in minimum product-of-sums form. The resulting function is

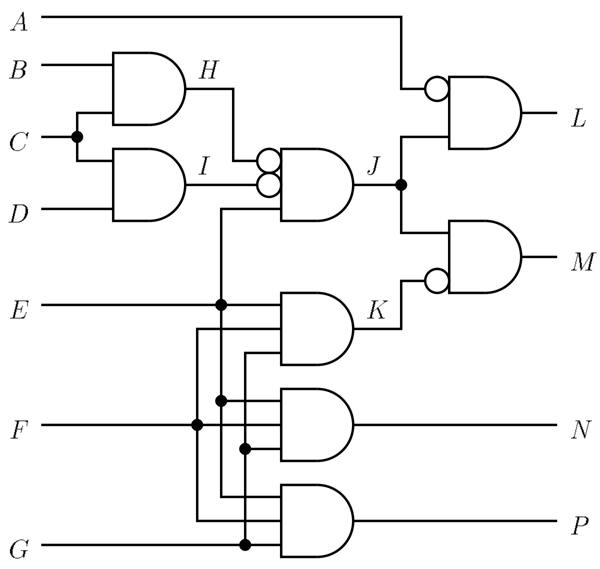

Figure 1: Example network fragment.

$$(J_g + H_g + I_g + \overline{E}_g)(\overline{H}_g + \overline{J}_g)(\overline{I}_g + \overline{J}_g)(E_g + \overline{J}_g)$$
$$(J_f + H_f + I_f + \overline{E}_f)(\overline{H}_f + \overline{J}_f)(\overline{I}_f + \overline{J}_f)(E_f + \overline{J}_f)$$
$$(\overline{J}_d + J_g + J_f)(\overline{J}_d + \overline{J}_g + \overline{J}_f)(\overline{J}_d + L_d + M_d),$$

with 11 clauses and 29 literals. The savings over earlier SAT-based algorithms are comparable. The equation can be further simplified by minimizing the functions for several gates simultaneously (e.g., the gates in a fanout-free region) but this makes the algorithm more complicated for a smaller additional improvement.

The TEGUS reduction eliminates redundancy found in other reductions. For example, the reduction in [17] represents the constraint $(E_g + \overline{J}_g)$ twice, as $J \rightarrow E$ and also as $\overline{E} \rightarrow \overline{J}$. Although the search algorithms can be modified to partially compensate for this redundancy (called *duality* in [17]), it is more straightforward not to add the redundancy in the first place.

Because TEGUS uses an intermediate reduction, it only needs the minimum form for an AND gate, but characteristic functions for other gates can be similarly derived. Since the equations can be long (e.g., for c6288, several equations have over 50 000 literals in the TEGUS minimized form), minimizing each function improves the performance considerably, as shown in Section V.

III Greedy Search

As a search problem, test generation is characterized by the variable orderings used for branching and what processing to do at each branch point. Although such search heuristics do not affect the completeness or asymptotic complexity of an algorithm, they can drastically affect the average case performance.

Most test generation algorithms use analysis of a network (or equation) to determine a variable ordering, such as the backtracing heuristics in a structural algorithm. The orderings used by early SAT-based algorithms depend primarily on the underlying solvers, although they can be influenced by ordering the problem description [6] or the use of other mechanisms [13]. Recent SAT-based algorithms [14, 15, 22, 16, 23, 17] analyze the 2-clauses of an equation, satisfying the easy part of the equation and then checking if the assignment also satisfies the entire equation. This heuristic emphasizes parts of the network where many inverters or buffers are used, giving little information relevant to a good variable ordering.

The recent SAT-based algorithms also use a static variable ordering, fixing a global ordering for the entire search regardless of the current partial assignment. This leads to unnecessary assignments and backtracking. For example, setting one input of an n-input AND gate to zero fixes the output to zero but does not force any value for the other inputs. A static variable ordering may assign values to these inputs even if no test exists when the gate output is zero, potentially wasting 2^{n-1} backtracks. Experiments confirm the disadvantages of static variable orderings [24]; they typically abort on an order of magnitude more faults using the same backtrack limits and are also an order of magnitude slower. Good fault coverage is achieved only with random patterns and very high backtrack limits.

Unlike previous algorithms, TEGUS uses a simple greedy variable selection. Since the equation is satisfied exactly when every clause is satisfied, the most obvious step which increases the number of satisfied clauses is to find the first unsatisfied clause in the equation and assign the first variable which can satisfy it. Clauses with three or more literals are in some sense the difficult part of the equation, so only these are considered during variable selection. The result is a fast, dynamic variable ordering which ignores 2-clauses, essentially the direct opposite of previous approaches.

The efficiency of such a heuristic depends strongly on the ordering of the clauses within the equation. In TEGUS the characteristic functions for the gates are added in depth-first search (DFS) order starting from primary inputs, with the effect that for each cone of logic, the clauses for gates driven by primary inputs occur first. This ordering has three advantages over those used in other SAT-based algorithms. First, with this ordering, the greedy variable selection mimics the PODEM [25] heuristic of branching only on primary input variables and deriving all other assignments from implications. This heuristic avoids many unnecessary conflicts in networks with reconvergence, but is not used in any other SAT-based algorithms.

Second, the DFS ordering improves on the PODEM heuristic by grouping together the clauses for gates whose output values converge within a small number of logic levels. This usually allows conflicts to be detected more quickly than by assigning values to gates which are logically distant (i.e., whose values converge only after many levels of logic or not at all). For example, in Fig. 1, after input G is assigned, a conflict is more likely to be detected by next assigning F instead of A since inputs G and F converge immediately while G and A do not converge at all. In a large network, this heuristic avoids many useless backtracks and is another reason why other algorithms (particularly the SAT-based algorithms) are not as robust, even with much higher search time limits.

The SAT solver must continue its 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 equation has indeed been solved. Thus, for nonstructural algorithms, the TEGUS DFS ordering has a third advantage of avoiding conflicts after a test has been successfully generated. If primary inputs are not assigned first, the search could have to backtrack and possibly even abort the search after a test has already been found. This is another reason why other SAT-based algorithms are less robust.

Backtrack Limit	Aborted Faults Per Strategy G1	G2	G3	G4	Combined abt	cpu
15	1 476	1 549	9 167	8 404	302	440.
150	639	574	6 235	6 535	202	510.
1500	559	429	4 104	5 286	174	620.

Table 1: Results for 18 ISCAS networks, no fault simulation.

For example, in Fig. 1 consider the fault G stuck-at-1. Assume that input C is driven by some additional logic and that variables E_g, F_g, and G_g have already been assigned 1, 1, and 0, respectively. This partial assignment is a test, but clause $(J_g + H_g + I_g + \overline{E}_g)$ describing gate J (see Section II) has not been satisfied yet. If the next decision is to set J_g to 0, i.e., the AND gate to a 0, the SAT solver may have to backtrack in order to justify a value of 0 for variable J_g. With the DFS ordering used in TEGUS, such backtracks never occur, improving the average case performance.

Other test generation algorithms have to do analysis during the search to avoid these problems which the TEGUS greedy ordering avoids naturally. One analysis heuristic dynamically marks a gate as *useless* when all its fanouts already have a justified value or are marked as useless [26, 23, 27]. By not branching on useless signals, this structural heuristic avoids backtracks after a test has been found, and can also be used to dynamically identify unique sensitization points [26]. However this analysis gives gives no information on which useful variable to choose next for branching; algorithms with no additional backtracing heuristics, such as [23], may still frequently branch on variables which are logically distant. Another heuristic, *dependency-directed backtracking* [27], avoids some of the wasted backtracks from branching on logically distant variables. When a contradiction is found during the search, the analysis checks if the search can immediate backtrack several decisions rather than only to the most recent reversible decision.

All such analysis heuristics are a tradeoff between the time needed to perform the analysis and the time saved by reducing the number of backtracks. Most of them also depend on information about the network structure which is not readily available from the SAT reduction. Fortunately, the greedy DFS variable ordering in TEGUS naturally avoids many bad decisions without doing any analysis whatsoever, and, as shown in Section V, is more robust and efficient than existing algorithms which use such analysis heuristics.

The results in Table 1 demonstrate the effectiveness of a fast greedy ordering, using the 18 ISCAS networks listed in Table 3. Column `G1` shows the number of aborted faults using the greedy strategy with DFS clause ordering for three different backtrack limits. With a limit of 150, this strategy completes on over 99.5% of the faults including over 96% of the redundant faults. As shown in Section 5.1, this single greedy strategy is more robust and efficient than many recent structural algorithms. We also tried to find an optimal DFS ordering using the distance from each gate to any primary input and the logic cone sizes, but found no ordering which was significantly better (or worse) than that defined by the network description. To emphasize the importance of the DFS ordering, when the formula is constructed in reverse order from outputs to inputs, G1 does just as poorly as a static ordering [24].

To improve the search efficiency, the characteristic equation is divided into two parts, the set of clauses for gates driven by at least one primary input, called the *subformula*, and the remaining clauses. During branching, only the subformula needs to be searched for unsatisfied clauses, since if the subformula is satisfied and all implications are followed without conflicts, the remaining clauses must also be satisfiable.

The form of strategy G1 suggests three complementary strategies, a heuristic which has been effective in other algorithms (e.g., [28, 14]), Greedy strategy G2 varies G1 by selecting the last free literal in the first unsatisfied clause of the subformula. Greedy strategies G3 and G4 select the first or last free literal in the last unsatisfied clause of the subformula. Choosing between the first and last free literal effectively changes the value assigned to a gate. For example, in Fig. 1, if the clauses for gate P are first in the subformula, one choice sets the output of gate P to 1, the other sets an input of gate P to 0, forcing the output to 0. Similarly, for G3 and G4, choosing between the first and last unsatisfied clause starts the search in different regions of the network, such as gate L versus gate P in the example.

Individually, G3 and G4 do worse than G1 and G2 because clauses at the end of a subformula often do not belong to the same cone of logic. Rather, they are whatever clauses were left over after the earlier cones were searched, and do not converge as directly, if at all. The last two columns of Table 1 show the number of aborted faults and total CPU time in seconds when all four strategies are applied in succession.

IV Iterated Global Implications

To complete on faults for which straightforward branch-and-bound aborts, most of which are redundant, the Socrates algorithm [29] introduced a procedure for computing global implications using the tautologies

$$(A \rightarrow B) \wedge (A \rightarrow \overline{B}) \quad \Rightarrow \quad \overline{A}, \tag{2}$$

$$(A \rightarrow B) \quad \Rightarrow \quad (\overline{B} \rightarrow \overline{A}). \tag{3}$$

Larrabee's algorithm [14] contains the following improved global implications procedure which, because it is SAT-based, is also simpler. For each free literal A, temporarily set A to 1 and call the bounding procedure of the SAT branch-and-bound solver. If a contradiction occurs, (2) is applied to infer $\overline{A}$. If an unsatisfied clause which initially had at least three free literals ends up with only a single free literal B, (3) is applied to infer $\overline{B} \rightarrow \overline{A}$. Thus this powerful heuristic is applied using only a trivial extension of the basic branch-and-bound.

We have improved the computation of global implications after observing that the results of this procedure can depend on what order the variables are processed. This ordering dependency is illustrated by the following example.

Example 1: In Fig. 1 with fault G stuck-at-1, if variable C_g is processed first, trying C_g as both 1 and 0 results in neither a conflict nor any global implications. Later, trying $E_g = 0$ forces K_f, K_d, N_f, N_d, P_f, and P_d to 0, causing a contradiction because clause $(K_d+N_d+P_d)$ is unsatisfied, i.e., all propagation paths are blocked. Thus by (2), $E_g = 1$. But now it is possible to find a global implication for C_g, namely $\overline{C}_g \rightarrow J_g$, so the global implication $\overline{J}_g \rightarrow C_g$ can be deduced. To find this implication, C_g must be processed after E_g, but previous algorithms have no means for ensuring this order will occur.

More complex dependencies can occur such that no ordering will find all the global implications by asserting each literal only once. Therefore we iterate through the list of free literals, computing global implications until one full iteration produces no new implications. More implications are found than in previous algorithms (such as [29, 14]), and the search space is reduced enough that TEGUS only computes global implications statically before the branch-and-bound search is started, not dynamically at every branch point of the search. Some benefits of dynamically computing an incomplete set of global implications [29, 30, 16, 31, 32, 17] are a result of this ordering dependence. For extremely difficult redundant faults, the iterated computation can also be applied dynamically during the search (although this is not necessary for any faults in the ISCAS networks).

The iterated procedure of TEGUS finds more global implications than previous algorithms. Socrates [29] only computes global implications for a subset of signals in the good network, does not handle all ordering dependencies, and uses too strict of a criterion for applying (3) as shown below in Example 2. Larrabee's procedure [14] computes global implications for all variables and uses a more general criterion for applying the tautologies, but it does not handle ordering dependencies.

The algorithms of Kunz *et al.* [32] and Silva *et al.* [27] have applied Larrabee's criterion in a structural algorithm, but they also do not handle ordering dependencies and, like Socrates, do not compute global implications for all variables. For example, they do not find the implication $E_g = 1$ in Example 1 because this global implication can only be derived using information about the faulty network and potential D-chains.

Finally, the transitive closure algorithms [15, 16, 23, 17] are even more limited because they only compute global implications using 2-clauses [21]. For example, these procedures cannot find the implications $E_g = 1$, $\overline{C}_g \rightarrow J_g$ in Example 1 because these implications are only found by considering the 3-clause $(K_d + N_d + P_d)$, i.e., one of these gates must be on a D-chain. For this reason, contrary to the claims in these papers, the dominator, unique sensitization, and global implication heuristics of earlier algorithms (e.g., [29, 14]) are not implicit in transitive closure.

The TEGUS global implications procedure can also be used with the exhaustive method proposed by Kunz and Pradhan [33] for dynamically computing global implications, since the two heuristics are independent. Where existing global implication

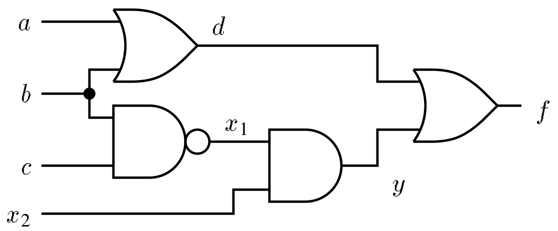

Figure 2: Network for Example 2.

procedures (e.g., [29, 14, 24]) call a standard bounding procedure to detect conflicts, they propose calling a secondary full branch-and-bound procedure (a.k.a. *recursive learning* procedure). If the secondary branch-and-bound is called with no backtrack limit (a.k.a. maximum recursion depth r_{max}), all necessary assignments are guaranteed to be found and the primary branch-and-bound will never backtrack. Since computing all necessary assignments is a co-NP-complete problem [2], backtrack limits are used in [33] for both search procedures. The TEGUS implications procedure could be used to improve the performance of the primary search, the secondary search, or both.

Iterating the global implications computation subsumes the various unique sensitization, path controller assignments, and dominator conditions used in other test generation algorithms, which is not true if each literal is only processed once [24][21]. Example 1 illustrates how TEGUS subsumes the improved unique sensitization procedure in Socrates [34]. Case analysis of the other global implications heuristics which have been proposed for structural and SAT-based algorithms is lengthly but straightforward. Similarly, if the global implications procedure of TEGUS is applied dynamically during the search, it subsumes heuristics such as dynamic unique sensitization. In TEGUS, the global implications procedure determines, for the current partial assignment, every single literal X such that assigning $\overline{X}$ causes a contradiction.

Three types of implications which have been recently proposed are not computed in TEGUS because they are already handled by the conventional implication procedure used during branch-and-bound search. The first type, *simple equivalence*, is based on identifying cycles of implications [14, 16, 15, 17]. All variables in such a cycle must always have the same value. This analysis indirectly identifies trivial cases of equivalence such as buffers and inverters, but does not detect more general equivalences such as the equivalence of variables K_g, N_g, and P_g in Fig. 1. Identifying variables which are forced to be equivalent or opposite via direct implications does not reduce the number of backtracks because when one variable is assigned, the others are immediately forced to the appropriate values.

The second type, *exclusion* [17], is just another name for direct implication. For example, in Fig. 1, since $\overline{E}_g \rightarrow \overline{J}_g$, and $\overline{J}_g \rightarrow \overline{L}_g$, the direct consequence, or *exclusion*, $\overline{E}_g \rightarrow \overline{L}_g$ can be inferred. Such implications are implicitly derived during the search, so computing them explicitly has no benefits.

The third type, *clause reduction* implications [17], are implications derived when a partial assignment results in a clause having only two free literals. In other words, this means taking

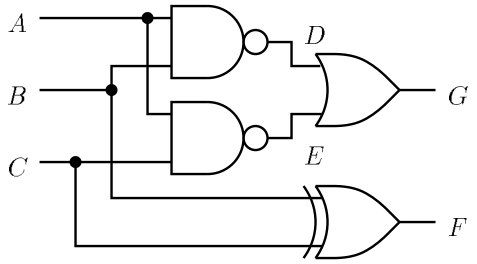

Figure 3: Network for Example 3.

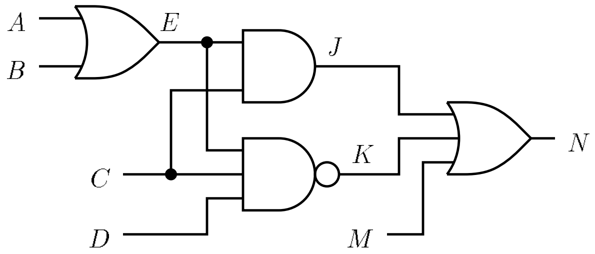

Figure 4: Network for Example 4.

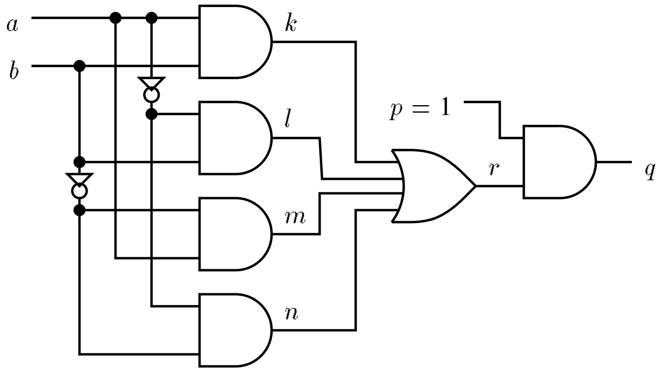

Figure 5: Network for Example 5.

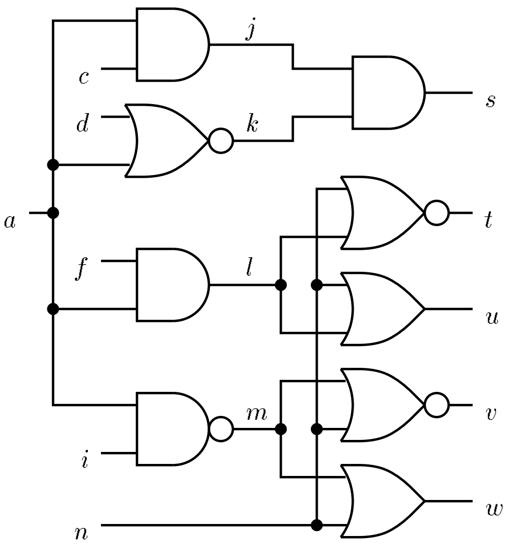

Figure 6: Network for Example 6.

into account signals which already have an assignment when doing implication. For example, in Fig. 1, when input B is assigned the value 1, the clause $(H_g + \overline{B}_g + \overline{C}_g)$ describing gate H has only two remaining free literals. Consequently, this clause can be reduced to $\overline{H}_g \rightarrow \overline{C}_g$, $C_g \rightarrow H_g$, or both. Despite the claim that this is a novel feature [17], all existing algorithms make use of the information that $C_g = H_g$ when $B = 1$.

As observed by Schulz *et al.* [34], there is obviously no benefit from explicitly deriving such implications that can be performed by any conventional implication procedure. Computing simple equivalences, exclusions, or clause reduction implications generally reduces efficiency without improving robustness. Consequently, in contrast with previous SAT-based algorithms, TEGUS does no special processing of direct implications or 2-clauses.

The simplicity of computing global implications in a SAT-based algorithm belies its power and generality. The benefits of even Larrabee's procedure are often underestimated, as illustrated by the following examples. The improved computation of TEGUS is not required for these small examples (because they do not have any ordering dependencies), but since TEGUS uses an extension of the procedure in [14], the examples also illustrate how TEGUS subsumes many previous procedures for computing global implications.

Example 2: The network in Fig. 2 is an example from [32] used to claim an improved global implication procedure. We show how Larrabee's procedure, and consequently TEGUS, also succeeds on this example (although it is true that Socrates cannot find the following global implication). As part of the global implications procedure, variable f_g is assigned the value 0 and the bounding procedure is called. Clauses $(\overline{d}_g + f_g)(\overline{y}_g + f_g)$ for gate f force $d_g = 0$ and $y_g = 0$, since the literal f_g cannot satisfy these clauses. Subsequently, clause $(\overline{b}_g + d_g)$ forces $b_g = 0$, and clause $(b_g + x_{1,g})$ for gate x_1 forces $x_1 = 1$. But now clause $(y_g + \overline{x}_{1,g} + \overline{x}_{2,g})$ can only be satisfied by $x_{2,g} = 0$. Since this clause initially had more than two free literals, (3) is applied to infer $x_{2,g} \rightarrow f_g$, contradicting the claim that this implication is not found by earlier procedures. The learning criterion proposed in [32] is a partial application of Larrabee's criterion in a structural algorithm (partial, because it is not applied to all variables).

Example 3: The examples used in [17] to claim an improved global implications procedure are likewise incorrect. Consider the network in Fig. 3 from [17] with the partial assignment $F_g = 0$, $G_g = 1$. As part of executing Larrabee's global implications procedure, variable D_g is assigned 0 and the bounding procedure is called. Setting D_g to 0 forces $A_g = 1$, $B_g = 1$, and $E_g = 1$. Clause $(\overline{A}_g + \overline{C}_g + \overline{E}_g)$ describing gate E now forces $C_g = 0$ since $\overline{C}_g$ is the only free literal. This results in a contradiction because clause $(\overline{B}_g + C_g + F_g)$ describing the XOR gate cannot be satisfied. Thus by (2), $D_g = 1$, contradicting the claim in [17] that this implication is not found. The other examples and related claims in [17] are similarly shown to be in error. Creating explicit 2-clauses from clauses with only two free literals is unnecessary because, unlike [17], all unsatisfied clauses are processed directly by the bounding procedure.

Example 4: During the computation of global implications for the network in Fig. 4, variable N_g is assigned the value 0 and the bounding procedure is called. This implies, through the 2-clauses in the equation, that J_g, K_g, and M_g must all be 0. Subsequently, $K_g = 0$ implies that E_g, C_g, and D_g must all be 1. But now clause $(J_g + \overline{E}_g + \overline{C}_g)$ describing gate J is unsatisfiable, and by (2), $N_g = 1$. Thus fault N stuck-at-1 is undetectable, contradicting the claim in [35] that this cannot be determined by any previous methods.

Example 5: The network in Fig. 5 is used in [33] to illustrate an improved global implications procedure. With the partial assignment $p_g = 1$, consider the global implications for $\overline{q}_g$. From the clauses for gates q and r, the bounding procedure directly assigns $r_g = k_g = l_g = m_g = n_g = 0$. The clauses for gate k now imply that $a_g = b_g = 0$. But now clause $(n_g + \overline{a}_g + \overline{b}_g)$ cannot be satisfied and by (2), $q_g = 1$, contradicting the claim in [33] that previous algorithms cannot find this implication.

Example 6: Consider the network in Fig. 6, from [33], for the fault a stuck-at-0. The assignment $a_f = 0$, $a_g = a_d = 1$ directly implies $j_f = s_f = 0$ and $k_g = s_g = 0$. The clause $(\overline{s}_d + s_g + s_f)$, describing the D-chain condition for gate s, implies $s_d = 0$, which in turn implies the fixed assignments $j_d = k_d = 0$. Subsequently, the global implications procedure is invoked for $n_g = 1$. The bounding procedure determines that this directly implies $w_g = w_f = 1$, etc., which, similarly to gate s, implies $w_d = v_d = u_d = t_d = 0$. Now clauses $(\overline{l}_d + t_d + u_d)(\overline{m}_d + v_d + w_d)$ can only be satisfied by the assignment $l_d = m_d = 0$, resulting in a contradiction for clause $(\overline{a}_d + j_d + k_d + l_d + m_d)$ and by (2), $n_g = 0$. This unique assignment is efficiently determined using only the bounding procedure of Larrabee's algorithm (or TEGUS), and does not require the exhaustive secondary branch-and-bound procedures used in [33].

The uniformity of the SAT representation is an advantage for computing global implications. Structural algorithms [34, 29, 30, 31, 32, 33] have to handle special cases for implications in forward versus backward directions, for implications of different gate types, for implications in both the good and faulty network, and for implications involving the fault propagation path. Having to check all these special cases can degrade overall performance, and is complicated enough that often global implications are not computed at all (e.g., [36, 37, 35, 38]) even though they are known to significantly improve robustness.

The SAT-based algorithms in [15, 16, 23, 17] are also much more complicated, computing global implications by constructing an implication graph from an energy function for a network, finding the strongly connected components with a special algorithm for sparse graphs with duality, constructing a second graph of the strongly connected components, computing its transitive closure, and searching it for edges corresponding to the hypothesis of (2). This costly procedure is performed at every branch point even for easy faults, degrading the overall performance as shown in the next section.

V EXPERIMENTAL RESULTS

To summarize the overall algorithm TEGUS, for each fault a characteristic equation is constructed in DFS order and four greedy strategies are tried, each with a low backtrack limit. If all fail, the computation of global implications is iterated and

Mode	Tested Faults SIM	SAT	Untested Faults RED	ABT	Memory (MB)	CPU Time (sec) SIM	EQN	SAT	Total
no fault simulation	-	133 995	7 250	0	13.5	-	343.	131.	477.
fault simulation	119 721	14 274	7 250	0	11.5	9.1	51.6	19.3	83.4
fault sim. & random tests	130 235	3 760	7 250	0	6.8	8.7	20.0	14.0	45.9
redundancy removal	126 274	2 827	0	0	6.9	17.1	18.9	5.6	46.3

Table 2: TEGUS totals for 18 ISCAS networks under four modes of operation.

the greedy strategies are retried with a higher backtrack limit. For the following experiments, we used an implementation with limits of 15 and 150 respectively.[2]

To evaluate TEGUS, we compare it with other published results using the ISCAS benchmark networks [19, 20]. The eight largest ISCAS89 sequential networks are tested assuming full scan.[3] Although these networks have many gates, the logic cones are quite shallow. For example, the average length of the characteristic equations for `c2670` is 3700 literals, while for `s35932`, it is only 630, even though the latter has 16 times more gates. The observation that test generation time increases linearly with network gate count [17, 39] is a result of this characteristic of the ISCAS networks and is not true in general; search time versus equation size is a more appropriate measure of asymptotic complexity.

Table 2 summarizes the data for TEGUS on the ISCAS85 and eight largest ISCAS89 networks using a DEC 7000/610 AXP, showing that TEGUS is a good balance of robustness, efficiency, and simplicity. The four rows correspond to four of the different modes for running TEGUS: without fault simulation, with fault simulation, with fault simulation and pseudo random patterns, and with redundancy removal. These are the modes of operation most commonly found in the literature. The first four columns show the number of faults detected by simulation, detected by the SAT algorithm, proved redundant, and aborted. Column `Memory` is the peak memory usage in megabytes as measured using our implementation, not estimated using assumptions about the sizes of data structures. Column `SIM` is the total time for fault simulation, column `EQN`, for extracting the characteristic equations, and column `SAT`, for solving the equations, all in seconds. Total time measures the complete program execution including reading the network, initialization, etc. The fast, greedy heuristics of TEGUS make creating the equations the most time consuming step.

5.1 ROBUSTNESS

Practical test generation algorithms are incomplete, aborting the search for a test after some limit has been reached to avoid spending an exorbitant amount of time on a fault. It is generally not possible to prove one algorithm will complete on more faults than another, so if one empirically completes on more faults, we call it more robust.

Ideally, robustness measures how well an algorithm will do on

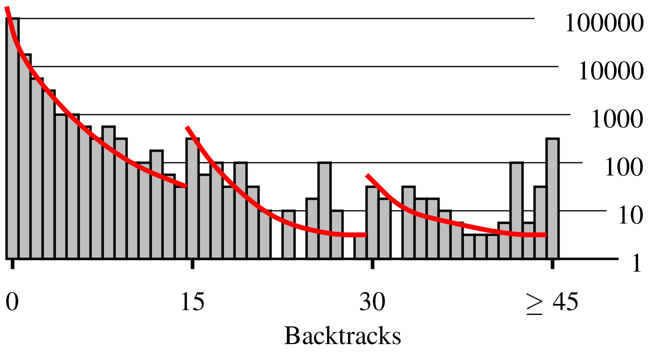

Figure 7: Distribution without fault simulation.

examples not yet seen, as well as on the benchmark suite. When fault simulation is used, several algorithms have no aborted faults for the ISCAS networks. This is very little data for comparing robustness since most of the ISCAS networks have fewer than 4000 gates (ignoring one-input gates), less than 4% of the testable faults are actually targeted by the deterministic part of the algorithm, and this small set is not easily reproduced (there is an element of randomness in which faults are targeted). We propose that the robustness of a test generation algorithm should be evaluated without using fault simulation since then more faults are targeted, giving a larger sample, and the sample set is unique.[4]

As shown in Table 2, without fault simulation TEGUS completes on every fault in the ISCAS networks. Fig. 7 shows the corresponding distribution of backtracks. Over 95% of the faults were completed in three or less backtracks. The shaded curves highlight the evidence that the greedy strategies are complementary, making them more efficient than a single strategy.

To evaluate the robustness of TEGUS, Table 3 compares the number of aborted faults on the ISCAS networks with fifteen recent algorithms. In this table, deterministic test generation (DTG) is applied to every modeled fault in each network. For each algorithm, column `abt` is the number of aborted faults, and column `cpu` is the execution time normalized to TEGUS (explained further in the next section). Backtrack limits range from 15 for TEGUS to 2×10^6 for `GIR90`.

With or without fault simulation or fault collapsing,[5] TEGUS

[2] For the ISCAS networks, a second limit of 15 is sufficient, but the higher limit may be useful for more realistic networks.

[3] The smaller ISCAS89 networks are not difficult for combinational test generation; TEGUS completes on all 22 of them in a total of two seconds.

[4] If fault collapsing is used, we assume that the standard techniques of simple fault equivalence and fault dominance are applied.

[5] Simple fault collapsing is fairly standard and does not mask nonrobustness like fault simulation does. The only published results we have found without fault simulation or fault collapsing are those of Chandra and Patel, column

completes on every fault in all the ISCAS networks. The DYTEST and EST algorithms, columns MAO90[41], GIR90[42], and GIR91[43], complete on the ten ISCAS85 networks, but the fastest of them is 11 times slower than TEGUS. For these algorithms, no data are presented for the ISCAS89 networks, which is unfortunate since the ISCAS85 and ISCAS89 networks have different characteristics and heuristics which work well on one set may not on the other. One of the algorithms by Kunz and Pradhan [33] completes on all faults, but is about 110 times slower than TEGUS. Of the remaining algorithms, seven aborted on more faults than greedy strategy G1 alone with a backtrack limit of 15 (when compared using the available data) as well as being slower.

One application of a robust test generator is for redundancy removal. TEGUS uses the straightforward method described by Bryan *et al.* [44], removing one redundancy at a time and iterating until no untestable fault remains. Table 2 shows the performance on the ISCAS networks, and Table 5 compares TEGUS with several other redundancy removal algorithms. For ABR92[45], it is not clear if all redundancies are successfully removed, and algorithm MEN94[35] fails to remove 298 redundant faults.[6]

5.2 Efficiency

When two test generation algorithms are equally robust, the second question is how they compare in efficiency. 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 algorithms is through the performance of actual implementations on a suitable set of examples.

Such experiments are prone to many pitfalls. Choosing an inappropriate set of examples or over-optimizing for the examples distorts results. Shortcomings may be masked by focusing on only a few isolated faults or by changing the algorithm parameters for different networks. Differences in computer performance are often unaccounted for, and preprocessing time is sometimes omitted even when it is a significant fraction of the total time. Indirect comparisons of efficiency using the number of backtracks or performance measures such as MIPS can be highly inaccurate. When a multiprocessor is used to run an algorithm [46], speedup factors may actually be a result of omitting the time for sequential portions of the algorithm, or of comparing with an inferior algorithm running on a uniprocessor, rather than a result of successful parallelization (in extreme cases, this can even give the illusion of superlinear speedup). If the programs for previous algorithms are not publicly available, there is little motivation to reimplement them as efficiently as possible, and it is incorrect to assume that if a heuristic improves an inefficient reimplementation, it will also improve the original.

CHN89[40] in Table 3.

[6]In comparison, the single greedy strategy G1 with a backtrack limit of 15 fails to complete on only 105 redundant faults for the same seven networks.

Finally, most algorithms are incompletely specified,[7] leading to different results when they are reimplemented or even just run on a different computer. These variations make it difficult to tell if an algorithm has even been reimplemented correctly, not to mention efficiently.

The preferred way to avoid most of these pitfalls is to do a direct comparison, running two implementations side by side on the same examples. However, since the programs used for most published experiments are not publicly available, we have had to use the next best approach. To compare TEGUS with other algorithms, for each algorithm A we performed the following experiment.

1. Port TEGUS to the model of computer reported in the published results for A, including any stated constraints on available memory, operating system, etc.
2. Run TEGUS on the ISCAS networks with the same options reported for A: with or without fault collapsing, with or without reverse order pattern simulation, with or without redundancy removal, with or without random tests, etc.
3. Normalize the reported times for A to the corresponding times obtained for TEGUS.

In all cases, the native C compiler was used with optimization enabled. To ensure consistent results across the variety of Amdahl, Apollo, DEC, HP, IBM, and Sun computers used in these experiments, the values used for pseudo random patterns are generated directly [48] rather than calling a system random number generator. Thus even the pseudo random patterns which are generated can be reproduced identically on different computer models. These experiments give a reasonably accurate comparison of efficiency, avoiding the aforementioned pitfalls which are within our control.

Tables 3, 4, and 5 show the results of these experiments. Tables 3 and 5 were described earlier. Table 4 compares TEGUS with reported results where deterministic test generation is applied only to a subset of faults. Most often fault simulation is used with or without random patterns (corresponding to the middle two rows of Table 2). The same strategies and backtrack limits were used with TEGUS for all networks in the three tables.

The last row of each table shows the normalized total time *for the available data*. For example, in the first column of Table 4, 21 is the sum of the six times reported in [13] divided by the sum for these same six networks using TEGUS. For algorithms run on different networks or with different options, these total times must be compared with care. Also, deterministic test generation is not applied to the same set of faults in Tables 4 and 5, and each algorithm uses different limits for terminating the random pattern phase (if there is one), and for aborting a search.

Although improvements are claimed for all of these algorithms, it is not clear on what data these claims are based. Some do not use the ISCAS benchmark networks [50, 15, 53, 69]

[7]For example, when a variable ordering is based on the number of implications, often hundreds of variables have the same priority. In this case, the ordering depends strongly on how ties are broken, which is implementation-dependent. The ordering dependency in Section IV and use of random values are two other examples.

	1988				1989				1990											
	CHE [49]		CHK [50]		CHN [40]		JAC [30]		PAT [47][a]		GIR [42]		ABR [51]		MAO [41]		RAJ [52]		CHK [53]	
Network	abt	cpu	abt	cpu	abt	cpu	abt	cpu	abt	cpu	abt	cpu	abt	cpu	abt	cpu	abt	cpu	abt	cpu
C432	7	1.4		-	12	3.8	4	2.1	5	0.9		43		-		3		23		-
C499		0.6		-	17	8.8	20	14.	8	0.8		3		-		2		15		-
C880		1.0		-		2.3		3.0		1.3		9		-		6		27		-
C1355	128	2.7		-	26	3.0	48	10.	14	1.7		7		-		6		48		-
C1908	82	1.9		-	252	5.8		3.2	10	1.1		100		-		7	3	41		-
C2670	43	0.8		-	115	2.2	8	1.3	203	0.9		9	85	1		13	20	15		-
C3540	62	0.9		-	662	4.6	9	1.6	289	1.5		6	118	2		9	49	20		-
C5315		1.4		-	26	2.1	1	3.7	70	1.4		20	45	3		13	30	17		-
C6288	231	1.2		-	4	1.6	514	7.7	24	0.9		9	4	1		8	1127	20		-
C7552	245	2.4		-	94	2.5	89	4.4	434	2.1		20	156	2		33	39	24		-
S1494		-		-		-		-		-		-		-		-		-		-
S5378		-		-		-		-		-		-		-		-		-		-
S9234		-		-		-		-		-		-		-		-		-		-
S13207		-		-		-		-		-		-		-		-		-		-
S15850		-		-		-		-		-		-		-		-		-		-
S35932		-		-		-		-		-		-		300		-		-		-
S38417		-		-		-		-		-		-		-		-		-		-
S38584		-		-		-		-		-		-		-		-		-		-
Total	798	1.4	-	-	1208	2.6	693	5.8	1057	1.2	**0**	**14**	408	27	**0**	**12**	1268	21	-	-

	1991				1992		1993		1994							
	STA [54][b]		GIR [43]		**TEG** [24]		TER [55]		COX [56]		LEE [38][c]		SIL [27]		KNZ [33][d]	
Network	abt	cpu	abt	cpu	abt	cpu	abt	cpu	abt	cpu	abt	cpu	abt	cpu	abt	cpu
C432		30		12		**1.0**	3	22		6.3		-		6		6
C499		580		17		**1.0**		1		2.8		-		11		5
C880		200		15		**1.0**		3		6.4		2		12		12
C1355		820		8		**1.0**		3		9.9		2		23		17
C1908		1300		13		**1.0**		7		8.7	22	6		11		10
C2670	2197	-		6		**1.0**	11	18	32	4.3	117	8		12		14
C3540	2931	-		8		**1.0**		3	2	5.9	323	28		6		7
C5315		120		13		**1.0**		3	2	5.1		-		16		21
C6288	5840	-		10		**1.0**	42	27	71	10.		-	2	11		8
C7552	6320	-		13		**1.0**	5	20	28	7.7	269	12		22		23
S1494		25		-		**1.0**		6		-		-		-		-
S5378		72		-		**1.0**		6		-		-		-		31
S9234	5816	-		-		**1.0**		4		-		-		-		28
S13207		39		-		**1.0**		10		-		-		-		62
S15850		47		-		**1.0**		5		-		-		-		36
S35932		260		-		**1.0**		71		-		-		-		620
S38417	25319	-		-		**1.0**		20		-		-		-		210
S38584		280		-		**1.0**		49		-		-		-		460
Total	48423	180	**0**	**11**	**0**	**1.0**	61	21	135	8.5	731	15	2	13	0	110

[a] Results for uniprocessor with SCOAP heuristics.
[b] Exceeded an unspecified memory limit on six networks after an unreported amount of time.
[c] Results for stuck-at faults only with a backtrack limit of 1000.
[d] Results for two phase ATPG with secondary branch-and-bound.

Table 3: Relative CPU times (normalized to TEGUS) and aborted faults, DTG applied to all modeled faults (- indicates not available).

	1989					1990				1991				1992		
	SIM	SCH	MIN	LAR	JAC	SCH	CHK	WAI	MAH	JAI	KND	CHK	GIR	ABR	MAT	WUD
Network	[13]	[29][a]	[28][b]	[14]	[30]	[29][c]	[15]	[39]	[57]	[58][d]	[59][e]	[16][f]	[43]	[45]	[31]	[60][g]
C432	25	1.1	7	19	5.0	2	-	0.8	2	23	99	4900	21	-	0.7	14
C499	110	2.0	9	27	12.	3	-	0.6	5	46	920	-	20	-	0.6	27
C880	97	2.4	12	120	7.9	3	-	1.0	1	29	7	-	32	-	0.8	22
C1355	100	3.0	20	40	8.1	4	-	1.3	8	96	740	-	33	-	1.7	47
C1908	140	3.9	25	98	8.0	4	-	1.2	4	70	320	-	58	-	1.1	320
C2670	11	0.5	6	25	2.8	1	-	0.3	2	14	140	-	6	3	0.2	9
C3540	-	1.3	6	57	4.0	4	-	0.6	2	51	32	-	19	-	0.7	93
C5315	-	3.0	31	64	19.	5	-	1.4	3	200	14	-	82	37	1.4	100
C6288	-	1.3	-	21	1.8	1	-	0.4	3	65	26	-	13	-	0.5	9
C7552	-	1.9	30	35	9.5	5	-	0.5	5	67	280	-	26	-	0.4	21
S1494	-	-	-	-	-	16	-	4.3	-	-	-	-	-	-	2.3	-
S5378	-	-	-	-	-	13	-	2.6	-	-	-	-	-	-	1.6	-
S9234	-	-	-	-	-	32	-	0.5	-	-	-	-	-	-	0.5	-
S13207	-	-	-	-	-	54	-	1.1	8	-	-	-	-	-	0.9	-
S15850	-	-	-	-	-	20	-	0.9	6	-	-	-	-	-	0.7	-
S35932	-	-	-	-	-	32	-	0.8	5	-	-	-	-	740	1.0	-
S38417	-	-	-	-	-	14	-	1.1	19	-	-	-	-	12	1.1	-
S38584	-	-	-	-	-	140	-	1.9	17	-	-	-	-	-	1.6	-
Total	21	1.4	18	35	5.8	33	-	**0.9**	9	54	170	4900	20	230	**0.9**	34

	1992			1993							1994		
	KND	KNZ	**TEG**	SRI	KNZ	POM	CHK	KLE	KON	TER	LIW	KNZ	KNZ	KNZ
Network	[61][h]	[36]	[24]	[62][i]	[32][j]	[37][k]	[17][l]	[63][m]	[64]	[55]	[65]	[33][n]	[33][o]	[33][p]
C432	1	59	**1.0**	26	170	-	2	12	12	6	3	120	2	11
C499	2	160	**1.0**	1000	-	-	8	18	26	4	20	4	4	75
C880	2	24	**1.0**	-	-	3	14	8	14	6	5	-	-	38
C1355	4	820	**1.0**	1200	680	7	16	51	46	13	29	4	4	120
C1908	6	180	**1.0**	210	5100	5	17	55	55	16	21	94	4	64
C2670	1	230	**1.0**	-	-	1	11	10	9	14	2	68	37	34
C3540	2	25	**1.0**	390	-	5	7	63	57	5	7	390	2	41
C5315	4	89	**1.0**	180	62000	11	31	78	90	14	43	10	10	210
C6288	4	4	**1.0**	-	-	1	7	36	39	1	18	2	2	15
C7552	2	120	**1.0**	-	9700	2	28	35	20	5	7	1400	13	57
S1494	8	-	**1.0**	-	-	-	16	-	71	18	-	-	-	-
S5378	6	-	**1.0**	-	-	8	71	64	67	24	-	19	19	170
S9234	3	-	**1.0**	-	-	2	45	58	56	7	-	1200	21	51
S13207	3	-	**1.0**	-	-	4	77	-	23	13	-	1000	21	87
S15850	3	-	**1.0**	-	-	4	91	-	49	14	-	130	11	180
S35932	28	-	**1.0**	-	-	120	120	-	220	41	-	180	180	290
S38417	5	-	**1.0**	-	-	11	230	-	35	34	-	4000	150	210
S38584	13	-	**1.0**	-	-	22	190	-	260	53	-	2200	150	330
Total	7	130	**1.0**	420	8800	20	100	42	81	22	9	700	63	140

[a] Times do not include preprocessing.
[b] Failed to complete on sixteen faults.
[c] Results using `Socrates 4.0`, including preprocessing time.
[d] Failed to complete on 1504 faults.
[e] Failed to complete on 133 faults.
[f] Time is for only four faults, and does not include preprocessing or fault simulation.
[g] Failed to complete on 988 faults.
[h] Failed to complete on 53 faults.
[i] Times are for only 169 faults and do not include preprocessing.
[j] Times are for only eleven faults, and do not include preprocessing or fault simulation.
[k] Failed to complete on 46 testable faults and an unreported number of redundant faults.
[l] Different networks were run with different backtrack limits and random pattern limits to improve results.
[m] Results for uniprocessor execution.
[n] Results for one phase ATPG without secondary branch-and-bound, redundant faults only.
[o] Results for one phase ATPG with secondary branch-and-bound, redundant faults only.
[p] Results for two phase ATPG with secondary branch-and-bound and fault simulation.

Table 4: Relative CPU times (normalized to TEGUS), DTG applied to only a subset of faults (- indicates not available).

	1989	1992			1994	
	JAC	KAJ	ABR	**TEG**	MEN	SIS
Network	[30]	[66]	[67][a]	[24]	[35][b]	[68][c]
C432	23	5	-	**1.0**	-	12
C499	7	7	-	**1.0**	-	14
C880	2	7	-	**1.0**	-	23
C1355	4	21	-	**1.0**	8	27
C1908	24	20	22	**1.0**	8	20
C2670	26	30	39	**1.0**	9	29
C3540	29	29	32	**1.0**	88	36
C5315	120	93	18	**1.0**	12	66
C6288	2	4	27	**1.0**	2	8
C7552	46	130	53	**1.0**	35	55
S1494	-	-	-	**1.0**	-	37
S5378	-	-	27	**1.0**	-	53
S9234	-	-	24	**1.0**	-	120
S13207	-	-	13	**1.0**	-	840
S15850	-	-	-	**1.0**	-	400
S35932	-	-	-	**1.0**	-	1400
S38417	-	-	-	**1.0**	-	170
S38584	-	-	150	**1.0**	-	1100
Total	33	64	57	**1.0**	28	460

[a] Results using a backtrack limit of 1000. The number of remaining redundancies is unreported.
[b] Failed to identify/remove 298 redundancies.
[c] Results using `sis` 1.2 combinational redundancy removal.

Table 5: Relative CPU times (normalized to TEGUS) for redundancy removal (- indicates not available).

or do not publish the total CPU times needed for comparison [70, 22, 71]. Others either make no experimental comparison [16, 17, 61, 65, 31, 35, 52, 29, 13, 54, 39], or compare against other heuristics of their own implementation [51, 71, 42, 43, 25, 33, 14, 57, 28]. These approaches vary too many factors to give good experimental results. A noteworthy exception is Cheng [49], who did a direct, normalized comparison between `CHE88`[49] and `CHN89`[40], and also reported absolute run times, permitting accurate future comparisons.

From Tables 3, 4, and 5, TEGUS performance is as good as or better than the best previous results. Without fault simulation, TEGUS is 10-100 times faster than the other published algorithms with no aborted faults. For efficient algorithms, Table 4 is primarily a comparison of fault simulator performance and says little about the test generation algorithm (unfortunately, results from applying DTG to all faults are not available for most of these algorithms). Nonetheless, TEGUS performance is also excellent under these conditions, and even TEGUS without fault simulation is faster than many of the algorithms with it (cf. Table 2). The combined benefits of efficiency and robustness are demonstrated by applying TEGUS to redundancy removal. As shown in Table 5, all redundancies are successfully removed from the benchmark networks in a fraction of the time taken by other algorithms. The comparisons in these tables are limited to stuck-at fault testing, but results using an earlier implementation of TEGUS have shown similar advantages for delay-fault testing [72].

Although run time is the main concern, memory usage can be a limiting factor for applying some algorithms to realistic VLSI networks. The memory requirements for TEGUS shown in Table 2 are reasonable and should scale well to larger networks. Even with the overhead of generating the characteristic equations, TEGUS is as practical as the best structural methods.

5.3 Simplicity

Simplicity of an algorithm is subjective but important because it balances the tradeoffs of robustness and efficiency. TEGUS would be more efficient if parts of the characteristic equations were reused when possible, but this is not as straightforward as simply creating a fresh equation for each fault. Applying the TEGUS heuristics in a structural algorithm reduces the overhead from generating the characteristic equations, but requires a more complicated implication procedure. Similarly, some algorithms in Tables 3, 4, and 5 can probably be made more efficient, but at the expense of a more complicated algorithm.

For example, the authors of the EST algorithm [42, 43] claim that their algorithm is 5.81 times faster than Socrates [29], and speeds up the identification of some redundancies by a factor of 200 000. However, as Table 4 makes evident, they do not actually compare with the published results for Socrates, but against their own reimplementation of the Socrates algorithm. When `GIR91` is compared directly to `SCH89` taking into account the difference in performance of the two computers used, Socrates is shown to be 14 times faster. If it is possible to improve the performance of algorithm `GIR91` by orders of magnitude, it will require a carefully optimized, more complicated, implementation. Thus algorithm simplicity is an important factor.

The branch and bound algorithm for SAT is simpler than for structural algorithms and the greedy heuristics are generally simpler than the testability measures and backtrace procedures used with other algorithms. Computing global implications is also simpler using the SAT reduction and, when iterated as in TEGUS, subsumes the unique sensitization conditions, etc., used in structural algorithms. TEGUS does not require many of the heuristics added to other algorithms (e.g., testability measures, single or multiple backtracing, implication graph processing, topological analysis, or dynamic global implications) and is one of the simplest algorithms in Tables 3, 4, and 5.

Our stand-alone implementation of TEGUS is is publicly available, allowing others to verify these experiments and also to use TEGUS as a base line for future experiments. We stress that such direct comparisons are more accurate (as well as easier) than trying to reimplement an algorithm, and are also to be preferred over the delayed comparisons we were required to use for Tables 3, 4, and 5. The use of TEGUS as a *benchmark program* to supplement the use of *benchmark networks* is especially valuable for experiments using examples which are not in the ISCAS benchmark set and which are unfamiliar or unavailable to others.

VI Conclusions

We have described TEGUS, an algorithm for combinational test generation using satisfiability which we argue is an excellent balance of robustness, efficiency, and simplicity. A combination of a simplified characteristic equation, DFS variable ordering, four fast greedy search strategies, and an iterated global implications procedure make TEGUS more robust and efficient than the best structural algorithms, without any testability measures or backtracing heuristics.

As shown in Table 3, TEGUS completes on every fault in the ISCAS networks without using fault simulation, and is over 10 times faster than other algorithms which have been shown to have no aborted faults under these conditions. Such comparisons without fault simulation are essential to determine the real robustness of an algorithm, and we hope that these data will be published for future algorithms. We have also shown in Tables 4 and 5 that TEGUS performance with fault simulation is comparable to the best published results although, for good algorithms, this measures the performance of fault simulation more than the efficiency of deterministic test generation.

Since all existing complete algorithms for test generation have the same worst case complexity, accurate comparisons of average case performance must be measured using implementations. Our publicly available implementation of TEGUS can be used as a base line for such experiments. The use of TEGUS as a benchmark program to supplement the ISCAS benchmark set avoids the pitfalls in reimplementing an algorithm, and allows meaningful comparisons despite variations in computer performance or the use of unfamiliar examples.

ACKNOWLEDGEMENT

The authors thank Hervé Touati for his insight and help, and are indebted to David Bultman, Abhijit Ghosh, Stuart Jarriel, Ron Neher, Jackie Patterson, Brian Reid, and Shirley Stephan for their generous assistance in collecting the data used for Tables 3, 4, and 5.

REFERENCES

[1] J. P. Roth, "Diagnosis of automata failures: A calculus and a method," *IBM J. Res. Develop.*, vol. 10, pp. 278–291, July 1966.

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

[3] O. H. Ibarra and S. K. Sahni, "Polynomially complete fault detection problems," *IEEE Trans. Comput.*, vol. C-24, pp. 242–249, Mar. 1975.

[4] S. A. Cook, "The complexity of theorem-proving procedures," in *Proc. 3rd Ann. ACM Symp. on Theory of Computing*, pp. 151–158, 1971.

[5] D. Svanæs, "Using logic programming (PROLOG) as a tool for microelectronics CAD/CAM/CAT," Master's thesis, Univ. of Trondheim, Norway, Dept. of Electrical Engr., 1982.

[6] D. Svanæs and E. J. Aas, "Test generation through logic programming," INTEGRATION*, the VLSI Journal*, vol. 2, no. 1, pp. 49–67, 1984.

[7] E. Gullichsen, "Heuristic circuit simulation using PROLOG," INTEGRATION, *the VLSI Journal*, vol. 3, no. 4, pp. 283–318, 1985.

[8] R.-S. Wei and A. Sangiovanni-Vincentelli, "VICTOR-II: Global redundancy identification, test generation, and testability analysis for VLSI combinational circuits," in *Proc. 2nd Int. Symp. VLSI Technology, Systems, and Applications*, (Taipei, Taiwan), May 1985.

[9] K. Eshghi, "Application of meta-level programming to fault finding in logic circuits," in *Logic Programming and its Applications* (M. van Caneghem and D. H. D. Warren, eds.), pp. 208–219, Norwood, NJ: Ablex Pub. Corp., 1986.

[10] R. Gupta, "Test-pattern generation for VLSI circuits in a PROLOG environment," in *Proc. 3rd Int. Conf. Logic Programming*, (London), pp. 528–535, July 1986.

[11] P. Varma and Y. Tohma, "PROTEAN: A knowledge based test generator," in *Proc. IEEE 1987 Custom Integrated Circuits Conf.*, (Portland, OR), May 1987.

[12] Y. Tohma and K. Goto, "Test generation for large scale combinational circuits by using PROLOG," in *Proc. 6th Conf. Logic Programming*, (Tokyo), pp. 298–312, June 1987.

[13] H. Simonis, "Test generation using the constraint logic programming language CHIP," in *Proc. 6th Int. Conf. Logic Programming* (G. Levi and M. Martelli, eds.), (MIT Press, Cambridge, MA), pp. 101–112, June 1989.

[14] T. Larrabee, "Efficient generation of test patterns using Boolean difference," in *Proc. Int. Test Conf.*, pp. 795–801, Aug. 1989. Also see [18].

[15] S. T. Chakradhar, V. D. Agrawal, and M. L. Bushnell, "Automatic test generation using quadratic 0-1 programming," in *Proc. 27th Design Autom. Conf.*, pp. 654–659, 1990.

[16] S. T. Chakradhar and V. D. Agrawal, "A transitive closure based algorithm for test generation," in *Proc. 28th Design Autom. Conf.*, pp. 353–358, 1991.

[17] S. T. Chakradhar, V. D. Agrawal, and S. G. Rothweiler, "A transitive closure algorithm for test generation," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 1015–1028, July 1993.

[18] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 4–15, Jan. 1992.

[19] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Proc. Int. Symp. Circuits and Systems*, pp. 1929–1934, May 1989.

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

[21] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "Notes on combinational test generation using satisfiability," Tech. Rep. UCB/ERL M94/?, U. C. Berkeley, Nov. 1994.

[22] V. Sivaramakrishnan, S. C. Seth, and P. Agrawal, "Parallel test generation using Boolean satisfiability," in *Proc. Fourth CSI/IEEE Int. Symp. VLSI Design*, pp. 69–74, Jan. 1991.

[23] S. T. Chakradhar, M. A. Iyer, and V. D. Agrawal, "Energy minimization based delay testing," in *Proc. European Conf. Design Automat.*, pp. 280–284, Mar. 1992.

[24] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," Tech. Rep. UCB/ERL M92/112, U. C. Berkeley, Oct. 1992.

[25] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Trans. Comput.*, vol. C-30, pp. 215–222, Mar. 1981.

[26] A. Lioy, "Adaptive backtrace and dynamic partitioning enhance ATPG," in *Proc. Int. Conf. Computer Design*, pp. 62–65, Oct. 1988.

[27] J. P. M. Silva and K. A. Sakallah, "Dynamic search-space pruning techniques in path sensitization," in *Proc. 31st Design Autom. Conf.*, pp. 705–711, June 1994.

[28] H. B. Min and W. A. Rogers, "Search strategy switching: An alternative to increased backtracking," in *Proc. Int. Test Conf.*, pp. 803–811, Aug. 1989.

[29] M. Schulz and E. Auth, "Improved deterministic test pattern generation with applications to redundancy identification," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 811–816, July 1989.

[30] R. Jacoby, P. Moceyunas, H. Cho, and G. Hachtel, "New ATPG techniques for logic optimization," in *Proc. Int. Conf. Computer-Aided Design*, pp. 548–551, Nov. 1989.

[31] Y. Matsunaga and M. Fujita, "A fast test pattern generation for large scale circuits," in *Proc. Synth. Simulation Meeting Int. Interchange*, pp. 263–271, Apr. 1992.

[32] W. Kunz and D. K. Pradhan, "Accelerated dynamic learning for test pattern generation in combinational circuits," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 684–694, May 1993.

[33] W. Kunz and D. K. Pradhan, "Recursive learning: A new implication technique for efficient solutions to CAD problems — test, verification, and optimization," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1143–1158, Sept. 1994.

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

[35] P. R. Menon, H. Ahuja, and M. Harihara, "Redundancy identification and removal in combinational circuits," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 646–651, May 1994.

[36] W. Kunz and D. K. Pradhan, "Recursive learning: An attractive alternative to the decision tree for test generation in digital circuits," in *Proc. Int. Test Conf.*, pp. 816–825, 1992.

[37] I. Pomeranz, L. N. Reddy, and S. M. Reddy, "COMPACTEST: A method

to generate compact test sets for combinational circuits," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 1040–1049, July 1993.

[38] K.-J. Lee, C. A. Njinda, and M. A. Breuer, "SWITEST: A switch level test generation system for CMOS combinational circuits," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 625–637, May 1994.

[39] J. Waicukauski, P. Shupe, D. Giramma, and A. Matin, "ATPG for ultra-large structured designs," in *Proc. Int. Test Conf.*, pp. 44–51, Aug. 1990.

[40] S. J. Chandra and J. H. Patel, "Experimental evaluation of testability measures for test generation," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 93–97, Jan. 1989.

[41] W. Mao and M. Ciletti, "DYTEST: A self-learning algorithm using dynamic testability measures to accelerate test generation," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 893–898, Aug. 1990.

[42] J. Giraldi and M. L. Bushnell, "EST: The new frontier in automatic test pattern generation," in *Proc. 27th Design Autom. Conf.*, pp. 667–672, June 1990.

[43] J. Giraldi and M. L. Bushnell, "Search state equivalence for redundancy identification and test generation," in *Proc. Int. Test Conf.*, pp. 184–193, 1991.

[44] D. Bryan, F. Brglez, and R. Lisanke, "Redundancy identification and removal," in *Proc. Int. Workshop Logic Synthesis*, May 1989.

[45] M. Abramovici, D. T. Miller, and R. K. Roy, "Dynamic redundancy identification in automatic test generation," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 404–407, Mar. 1992.

[46] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, ch. 10. Palo Alto, CA: Morgan Kaufmann Publishers, 1989.

[47] S. Patil and P. Banerjee, "A parallel branch and bound algorithm for test generation," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 313–322, Mar. 1990.

[48] S. K. Park and K. W. Miller, "Random number generators: Good ones are hard to find," *Commun. of the ACM*, vol. 31, pp. 1192–1201, Oct. 1988.

[49] W.-T. Cheng, "Split circuit model for test generation," in *Proc. 25th Design Autom. Conf.*, pp. 96–101, 1988.

[50] S. T. Chakradhar, M. L. Bushnell, and V. D. Agrawal, "Automatic test generation using neural networks," in *Proc. Int. Conf. Computer-Aided Design*, pp. 416–419, Nov. 1988.

[51] M. Abramovici, D. T. Miller, and R. Henning, "Global cost functions for test generation," in *Proc. Int. Test Conf.*, pp. 35–43, 1990.

[52] J. Rajski and H. Cox, "A method to calculate necessary assignments in algorithmic test pattern generation," in *Proc. Int. Test Conf.*, pp. 25–34, 1990.

[53] S. T. Chakradhar, M. L. Bushnell, and V. D. Agrawal, "Toward massively parallel automatic test generation," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 981–994, Sept. 1990.

[54] T. Stanion and D. Bhattacharya, "TSUNAMI: A path-oriented scheme for algebraic test generation," in *FTCS-21: Int. Symp. Fault-Tolerant Computing*, pp. 36–43, June 1991.

[55] M. Teramoto, "A method for reducing the search space in test pattern generation," in *Proc. Int. Test Conf.*, pp. 429–435, Oct. 1993.

[56] H. Cox and J. Rajski, "On necessary and nonconflicting assignments in algorithmic test pattern generation," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 515–530, Apr. 1994.

[57] U. Mahlstedt, T. Grüning, C. Özcan, and W. Daehn, "CONTEST: A fast ATPG tool for very large combinational circuits," in *Proc. Int. Conf. Computer-Aided Design*, pp. 222–225, Nov. 1990.

[58] K. K. Jain, J. Jacob, and M. K. Srinivas, "ATPG with efficient testability measures and partial fault simulation," in *Proc. Fourth CSI/IEEE Int. Symp. VLSI Design*, pp. 35–40, Jan. 1991.

[59] S. Kundu, I. Nair, L. Huisman, and V. Iyengar, "Symbolic implication in test generation," in *Proc. European Conf. Design Automat.*, pp. 492–496, Feb. 1991.

[60] D. M. Wu and R. M. Swanson, "Multiple redundancy removal during test generation and synthesis," in *Proc. IEEE VLSI Test Symp.*, pp. 274–279, Apr. 1992.

[61] S. Kundu, L. M. Huisman, I. Nair, V. Iyengar, and L. Reddy, "A small test generator for large designs," in *Proc. Int. Test Conf.*, pp. 30–40, 1992.

[62] S. Srinivasan, G. Swaminathan, J. H. Aylor, and M. R. Mercer, "Combinational circuit ATPG using binary decision diagrams," in *Proc. IEEE VLSI Test Symp.*, pp. 251–258, Apr. 1993.

[63] R. H. Klenke, L. Kaufman, J. H. Aylor, R. Waxman, and P. Narayan, "Workstation based parallel test generation," in *Proc. Int. Test Conf.*, pp. 419–428, Oct. 1993.

[64] M. H. Konijenburg, J. T. van der Linden, and A. J. van de Goor, "Test pattern generation with restrictors," in *Proc. Int. Test Conf.*, pp. 598–605, Oct. 1993.

[65] W. Li, C. McCrosky, and M. Abd-El-Barr, "Reducing the cost of test pattern generation by information reusing," in *Proc. Int. Conf. Computer Design*, pp. 310–313, Oct. 1993.

[66] S. Kajihara, H. Shiba, and K. Kinoshita, "Removal of redundancy in logic circuits under classification of undetectable faults," in *FTCS-22: Int. Symp. Fault-Tolerant Computing*, pp. 263–270, June 1992.

[67] M. Abramovici and M. A. Iyer, "One-pass redundancy identification and removal," in *Proc. Int. Test Conf.*, pp. 807–815, 1992.

[68] E. M. Sentovich *et al.*, "Sequential circuit design using synthesis and optimization," in *Proc. Int. Conf. Computer Design*, pp. 328–333, Oct. 1992.

[69] T. Fujino and H. Fujiwara, "An efficient test generation algorithm based on search state dominance," in *FTCS-22: Int. Symp. Fault-Tolerant Computing*, pp. 246–253, June 1992.

[70] T. Kirkland and M. R. Mercer, "A topological search algorithm for ATPG," in *Proc. 24th Design Autom. Conf.*, pp. 502–508, June 1987.

[71] D. Bhattacharya and P. Agrawal, "Boolean algebraic test generation using a distributed system," in *Proc. Int. Conf. Computer-Aided Design*, pp. 440–443, Nov. 1993.

[72] A. Saldanha, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Equivalence of robust delay-fault and single stuck-fault test generation," in *Proc. 29th Design Autom. Conf.*, pp. 173–176, June 1992.