WRL Research Report 90/3

Two Papers on Test Pattern Generation

Tracy Larrabee

digital Western Research Laboratory · 100 Hamilton Avenue · Palo Alto, California 94301 USA

Two Papers on Test Pattern Generation

Efficient Generation of Test Patterns Using Boolean Difference

A Framework for Evaluating Test Pattern Generation Strategy

Tracy Larrabee
March, 1990

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

A combinational circuit can be tested for the presence of a *single stuck-at fault* by applying a set of inputs that excite a verifiable output response in that circuit. If the fault is present, the output will be different than it would be if the fault were not present. Given a circuit, the goal of an automatic test pattern generation system is to generate a set of input sets that will detect every possible single stuck-at fault in the circuit.

These two papers describe a new method for generating test patterns: the Boolean satisfiability method. The new method is quite general and allows for the addition of any heuristic used by the structural search methods. The Boolean satisfiability method has produced excellent results on popular test pattern generation benchmarks. The first paper, *Efficient Generation of Test Patterns Using Boolean Difference*, gives an overview of a successful test pattern generation system using the Boolean satisfiability method. The second paper, *A Framework for Evaluating Test Pattern Generation Strategies*, describes potential test pattern generation heuristics and their efficacy in the Boolean satisfiability system.

Efficient Generation of Test Patterns Using Boolean Difference

Tracy Larrabee

March 1990

Abstract

Most automatic test pattern generation systems for combinational circuits generate a test for a given fault by directly searching a data structure representing the circuit to be tested. This paper describes a new system that divides the problem into two parts: First, it constructs a formula expressing the Boolean difference between the unfaulted and faulted circuits. Second, it applies a Boolean satisfiability algorithm to the resulting formula. The new system can incorporate any of the heuristics used by structural search techniques. It is not only quite general, but is able to test or prove untestable every fault in the popular Brglez-Fujiwara benchmark system.

This report is a slightly revised version of a paper appearing in the 1989 proceedings of International Test Conference.

Contents

1 Introduction

A combinational circuit can be tested for the presence of a single stuck-at fault by applying a set of inputs (a test pattern) that excite a verifiable output response in that circuit. Given a circuit, the goal of an automatic test pattern generation (ATPG) system is not only to generate a set of test patterns that will detect every possible single stuck-a fault in the circuit, but also to identify all untestable faults in the circuit.

The time needed to generate one test pattern that will detect the presence of a single stuck-at fault is, in the worst case, exponential in the size of the circuit under test [5]. Many test generation systems have been designed and implemented [9, 6, 4, 10] with the idea of avoiding those worst cases—of keeping the time needed to generate a test, in the expected case, within the realm of computational possibility. These systems all perform their search for a solution in a topological manner.

This paper describes a new approach to algorithmic generation of test patterns for combinational circuits. The method consists of two parts: First, given a circuit and a fault site in that circuit, construct a formula expressing the *Boolean difference* of a circuit with respect to a given fault. Second, apply a *Boolean satisfiability* algorithm to the resulting formula. Since the Boolean difference formula defines the complete set of tests capable of distinguishing between the faulted and unfaulted circuits, satisfying this formula will give us a set of inputs that detect the fault.

The Boolean difference of a circuit and a fault has long been considered theoretically important [11, 8], but the tedious nature of the algebraic manipulations involved in solving the Boolean difference led to its disfavor as a practical tool for test pattern generation. Fortunately, the Boolean difference need not be solved to be useful for generating tests: *satisfying* the formula is rewarding. Our system correctly tests or proves untestable every fault in the Brglez-Fujiwara benchmark system[2].

Our method of generating a test pattern for a stuck-at fault is to first generate a formula describing the set of possible patterns that detect that fault, and then satisfy the resulting formula. The following two sections describe these two steps in detail; the remainder of this section describes the complete test pattern generation system.

After wirelist translation, the first phase of test pattern generation begins. Pseudo-random vectors are produced 32 at a time and simulated using a Parallel Pattern, Single Fault Propagation (PPSFP) system modeled after the one reported by Waicukauski et al [12]. When one complete PPSFP pass produces fewer than a predetermined number of vectors (currently two), the second phase, algorithmic pattern generation, begins.

The attempt to generate a pattern for the first fault remaining on the fault list may terminate successfully or unsuccessfully. An attempt is successful if a pattern is generated or it is proven that no pattern exists; an attempt is unsuccessful if too much time has passed without reaching a successful conclusion. If a pattern is generated, this pattern is simulated against each of the remaining faults using a simple single pattern, single fault propagation fault simulator. Algorithmic pattern generation terminates when every fault has been declared covered, uncoverable, or aborted. The performance of the complete

system is reported in Section 4.

2 Extracting the Formula

A circuit is represented as a directed acyclic graph with the sources of the graph being the outputs of the circuit and the sinks being the inputs of the circuit. By walking the graph starting at any output, each of the nodes that can affect the value of that output are reached. Each node of the dag, a gate or a fanout point, is tagged with the logic formula, in 3-element conjunctive normal form, or 3CNF (also known as product of sums form). The formula associated with a logic element is a characteristic formula that is true if and only if the truth values assigned the variables representing the wires connected to the gate are consistent with the truth table for the element. For example, the formula for the AND gate shown in Figure 1 is $(\overline{D}+A)\cdot(\overline{D}+B)\cdot(D+\overline{A}+\overline{B})$. This formula is true if the varibles A, B, and D take on values consistent with the formula $D = A \cdot B$. For comparison, the disjunctive normal form (or sum of products) version of the same formula is $A\cdot B\cdot D+\overline{A}\cdot B\cdot\overline{D}+A\cdot\overline{B}\cdot\overline{D}+\overline{A}\cdot\overline{B}\cdot\overline{D}$.

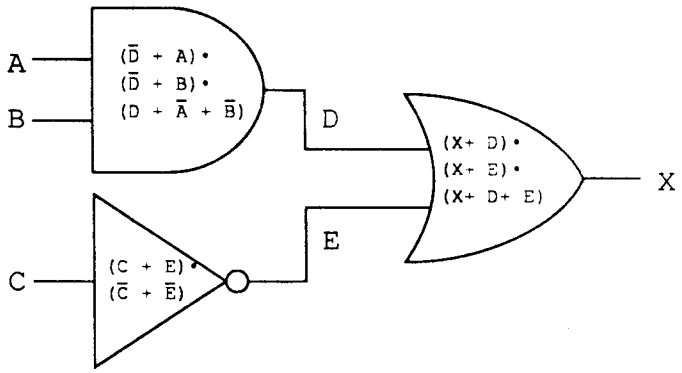

Figure 1: A Circuit

A formula describing the value of one of the circuit outputs in terms of its inputs can be obtained by walking the graph and taking the conjunction of all of the formulas for the visited nodes. Since variables are used to represent every wire of the circuit, the desired formula can be obtained in time and space that is linear in the size of the circuit. The formula for the output of the circuit in Figure 1 is $(X+\overline{D})\cdot(X+\overline{E})\cdot(\overline{X}+D+E)\cdot(\overline{D}+A)\cdot(\overline{D}+B)\cdot(D+\overline{A}+\overline{B})\cdot(C+E)\cdot(\overline{C}+\overline{E})$.

A faulted circuit is represented by a copy of its associated unfaulted circuit with renamed variables. Because the unfaulted and faulted circuits will have identical behavior except in those nodes that are affected by the fault, only the variables that are associated with wires that lie on a path between the fault site and a primary output need to be renamed. Figure 2 shows a faulted circuit corresponding to the unfaulted circuit in Figure 1 with line

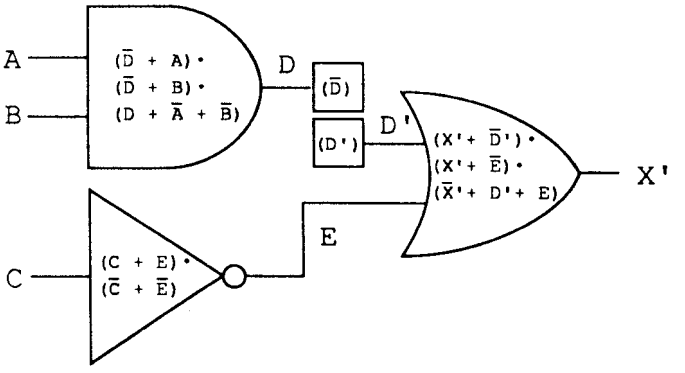

Figure 2: A Faulted Circuit

D stuck-at 1. The formula for the output of the circuit in Figure 2 is $(X' + \overline{D'}) \cdot (X' + \overline{E}) \cdot (\overline{X'} + D' + E) \cdot (D') \cdot (C + E) \cdot (\overline{C} + \overline{E})$.

The Boolean difference of a circuit with respect to a particular fault is defined as the XOR of the output of the unfaulted circuit and its associated faulted circuit output. The formula for the Boolean difference is obtained by walking the unfaulted circuit and the faulted circuit (as just described), and taking the conjunction of their formulas together with the formula for the XOR of the unfaulted output variable and the faulted output variable. The formula $BD = X \oplus X'$ is translated to $(BD = V_1 + V_2) \cdot (V_1 = X \cdot \overline{X'}) \cdot (V_2 = \overline{X} \cdot X')$ where V_1 and V_2 are automatically generated new variables.

The formula for the Boolean difference resulting from the XOR of the output of the unfaulted circuit of Figure 1 and the faulted circuit of Figure 2 is $(X + \overline{D}) \cdot (X + \overline{E}) \cdot (\overline{X} + D + E) \cdot (\overline{D} + A) \cdot (\overline{D} + B) \cdot (D + \overline{A} + \overline{B}) \cdot (C + E) \cdot (\overline{C} + \overline{E})$ (the clauses contributed by the unfaulted circuit) $\cdot (X' + \overline{D'}) \cdot (X' + \overline{E}) \cdot (\overline{X'} + D' + E) \cdot (D')$ (the clauses contributed by the faulted circuit) $\cdot (\overline{X} + \overline{V_1}) \cdot (X' + \overline{V_1}) \cdot (V_1 + X + \overline{X'}) \cdot (X + \overline{V_2}) \cdot (\overline{X'} + \overline{V_2}) \cdot (V_2 + \overline{X} + X') \cdot (\overline{BD} + V_1 + V_2) \cdot (\overline{V_1} + BD) \cdot (\overline{V_2} + BD)$, (the clauses contributed by $BD = X \oplus X'$).

Generating a test pattern for the given fault is now a matter of satisfying this 3CNF Formula. If the formula cannot be satisfied, the fault is undetectable.

3 Satisfying the Formula

The problem of satisfying a 3CNF formula, *3SAT*, can be viewed as searching a binary tree. Each node of the tree corresponds to a variable in the formula to be satisfied, and the two branches of a node correspond to the two possible Boolean assignments to the variable associated with that node. A path from the root node to any other tree node is *consistent* with the formula if the partial binding associated with that path causes no 3CNF clause of the formula to evaluate to false (in the sense that all its literals evaluate to false). In order to satisfy the formula, a consistent path from the root node to any leaf node must be found.

The 3SAT problem was one of the first to be proven NP-complete [3]. This means that we have transformed one potentially exponential problem into another. However, the class of formulas generated by combinational circuits is an interesting sub-class of all 3CNF formulas, and we can use this fact to try to avoid the worst case behavior of 3SAT: At least two thirds of the clauses generated for the Boolean difference of a combinational circuit have only two disjuncts (are in 2CNF). This is true because each two-input unate gate contributes two binary (2CNF) clauses and one ternary clause. Unate gates with more than two inputs contribute more than two thirds binary clauses, and fanout points, buffers, and inverters contribute only binary clauses. In practice we have found that 80% to 90% of the clauses are in 2CNF. The problem of satisfying a 2CNF formula, 2SAT, is satisfiable in time linear in the number of clauses plus the number of variables [1]. We may have an exponential number of 2SAT solutions, but we can use information from the ternary clauses to guide the iteration through the 2SAT assignments.

3.1 2SAT

We use a well-known algorithm for satisfying a 2CNF formula [1]. The first step is to construct an *implication graph*. Each 2CNF clause $(X+Y)$ can be viewed as two implications: $\overline{X} \Rightarrow Y$ and $\overline{Y} \Rightarrow X$. The implication graph for a 2CNF formula shows all of the constraints imposed by 2CNF clauses on the logic values of the variables involved.

More formally, for each variable X occurring in the 2CNF clauses, there are two vertices in the graph, labeled X and $\overline{X}$. For every 2CNF clause $(X+Y)$ there are two edges in the graph: one from $\overline{X}$ to Y, and one from $\overline{Y}$ to X. The edge represents the logical implication between the two literals. We can now bind logic values to the variables in the graph. Any assignment is legal as long as it does not cause a node labelled true to preceed (or imply) a node labelled false. Before we label the graph, we can simplify it by reducing *strongly connected components* to single nodes. A strongly connected component is a maximal set of nodes in a graph such that every node in the set is reachable from every other node in the set.

A strongly connected component represents a set of variables that are in an equivalence class. If any equivalence class contains both a literal and its negation, the formula is unsatisfiable. After each strongly connected component is reduced to a single node, the graph will not contain any cycles. Now we can find a binding for the 2CNF formula by visiting the vertices in any topological order. We choose a topological order that maximizes the number of variables in ternary clauses that are bound to false and thus narrowed to 2CNF or unary clauses.

As an example of how 2SAT works, consider the small circuit in Figure 3; imagine that we wish to iterate through all possible bindings to the variables $A, A1, A2, B$, and C. The formula for C is $(\overline{A}+A_1)\cdot(A+\overline{A_1})\cdot(\overline{A}+A_2)\cdot(A+\overline{A_2})\cdot(\overline{A_1}+B)\cdot(\overline{A_1}+\overline{B})\cdot(\overline{C}+A_2)\cdot(\overline{C}+B)\cdot(\overline{A_2}+\overline{B}+C)$. The implication graph of the 2CNF portion of this formula is shown in Figure 4. After the strongly connected components of this graph are reduced to unit nodes, the resulting graph is shown in Figure 5. The final graph clearly shows that $C \Rightarrow \overline{C}$, and therefore C

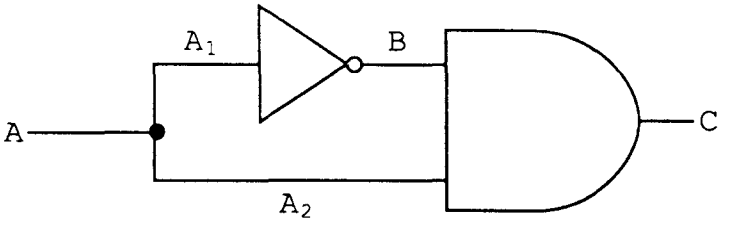

Figure 3: A Circuit

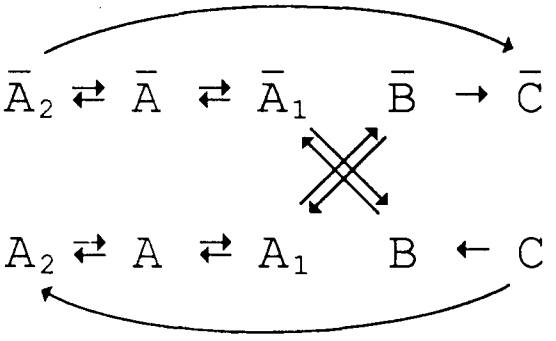

Figure 4: The Implication Graph

must be bound to false. Given this restriction, only one node in the graph remains, and it can assume either Boolean value and remain consistent with the ternary clause.

3.2 Iterating through 2SAT Bindings

We've just described a method for constructing a satisfying assignment for the 2CNF portion of the formula by assigning values to the literals so that no node labelled true has a directed path to a node labelled false. Clearly there are many such assigments. We would like to iterate through these assignments in an order that maximizes our chances of quickly discovering a 2SAT solution that can be extended to a satisfying assignment for the entire 3CNF formula.

We begin our iteration by ordering the variables by a metric that combines the number of constraints the variable places on the other variables within the implication graph and the number of times the variable appears in the ternary clauses. This defines a total order on the 2SAT solutions: One assignment preceeds another if the n-bit binary number representing the values of the variables (in the previously fixed order) preceeds the n-bit binary number for the other assignment. We consider the 2SAT solutions in this order, using standard search techniques to attempt to extend each 2CNF solution to a solution for the entire formula.

There is an intuitive rationale behind the metrics used to heuristically order the search:

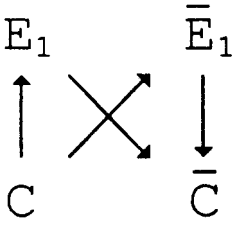

Figure 5: The Reduced Implication Graph

If a partial binding is going to lead to a contradiction, we want to discover this as quickly as possible. By beginning our construction of a 2SAT solution with those variables with the largest number of constraints on other variables, we are more likely to discover any conflicts early in the construction.

4 Minimizing the Search Tree

The algorithm we have described is complete: If a solution exists, the system will eventually find it. However, we can speed up the satisfier tremendously by making sure that it does not explore parts of the search tree that are known to contain no solutions. The search tree may be reduced both by subtracting clauses from the formula and by adding clauses to the formula.

4.1 Removing Clauses

We can remove a variable from the formula (along with all the clauses containing the variable) if we are guaranteed that removing the variable will not cause a satisfiable formula to appear to be an unsatisfiable one. We can use the *determines* relation presented below to identify and remove extraneous variables from the formula.

We say that variable V determines variable W if both assignments to V cause W to appear in the formula only negated or only unnegated. In this case, we may remove all clauses containing W from the formula and postpone the assignment of W until after the final assignment of V has been made.

This technique can be used to mimic the behavior of the FAN algorithm [4], which stops its backtrace operation at *head lines*, wires guaranteed not to be involved in reconvergent fanout loops.

As an example, in the Boolean difference formula presented for the circuit in Figure 1, E determines C but C does not determine E. In fact, every variable in the formula but BD is determined by some other variable. Since the circuit from which we produced the formula is completely fanout free, it is not surprising that a satisfying binding can be found with no search.

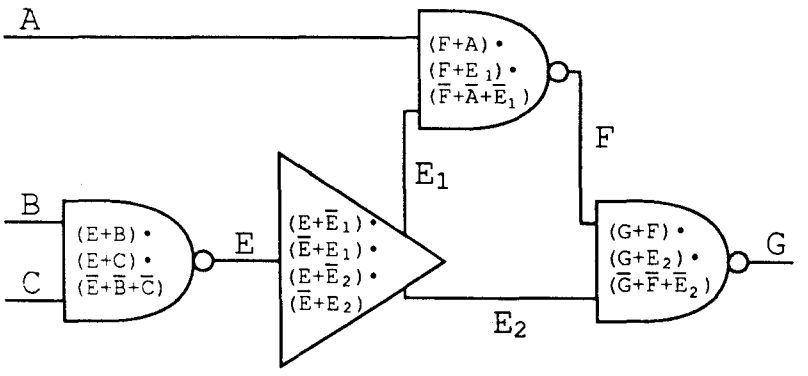

Figure 6: The Formula for G can be reduced to $(E + \overline{E_1}) \cdot (\overline{E} + E_1) \cdot (E + \overline{E_2}) \cdot (\overline{E} + E_2) \cdot (F + E_1) \cdot (G + E_2) \cdot (G + F) \cdot (\overline{G} + \overline{E_2} + \overline{G})$.

A more interesting example appears in Figure 6. The characteristic formula for G would normally consist of 13 clauses, but the removal of all clauses containing variables A , B , and C will leave only 8 clauses in the remaining formula because F determines A, and E determines B and C .

Unfortunately, our technique as stated will not remove as many variables as may be safely removed from the formula: the technique will not remove variables corresponding to input wires for XOR gates not involved in reconvergent fanout loops. We are working on a modification that will correctly deal with XOR gates.

4.2 Adding Clauses

We can take the basic formula to be satisfied and add clauses that explicitly state information that can be derived by the satisfier, but may only be derived after a certain amount of case-splitting. The simplest example of such redundant information is the value of the unfaulted wire with the fault. For example, the formula for the fault shown in Figure 2 contains the unary clause (D'). The satisfier can derive that the variable (D) must take on the value false in order for the XOR of the faulted and unfaulted circuits to be equal to true, but can add that information explicitly, by adding the clause $\overline{D}$ to the formula. Adding this kind of derivable information can speed up the satisfier by an order of magnitude. The effects of the added clauses mentioned here are fully described in another paper [7], but three different kinds of redundant information to be added are described in this section.

4.2.1 Active Clauses

One important class of redundant information is a set of clauses that speed fault propagation. If a fault is detectable, there must be at least one path from the fault site to a primary output such that every wire on that path has different unfaulted and faulted values. There may be more than one such path, but we only need to find one.

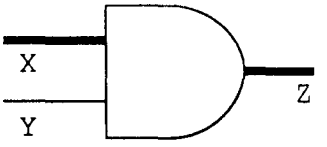

Figure 7: If X is active, Z must be active: $(\overline{Act_X} + Act_Z)$

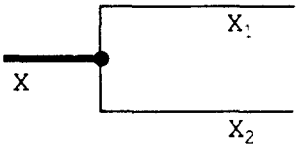

Figure 8: If X is active, either X_1 or X_2 must be active: $(\overline{Act_X} + Act_{X_1} + Act_{X_2})$

For every wire that lies on a path between the fault site and a primary output, we allocate an *active* variable. If the active variable for a wire is true, then the faulted value of the wire differs from the unfaulted value. For example, for the circuit in Figure 2 we allocate the variables Act_D and Act_X , and add the clauses $(\overline{Act_D} + D + D')$, $(\overline{Act_D} + \overline{D} + \overline{D'})$, $(\overline{Act_X} + X + X')$, and $(\overline{Act_X} + \overline{X} + \overline{X'})$ to the formula to be satisfied. We also add clauses that state that if an input to a single-output node is active, the output is active; if an input to a multi-output gate is active, one of the outputs is active; and the fault site is always active. Figure 7 and Figure 8 show the clauses that are added for circuit elements with fanin and for circuit elements with fanout.

It is important to notice that the active implication clauses (as shown in Figures 7 and 8) for the circuit as a whole can be used to determine all of the unique sensitization points (points of total reconvergence) in a circuit.

4.2.2 Critical Clauses

If a node is on the active path, we know that node must propagate the fault (discrepancy). This means that the non-active inputs to the node must have taken on critical values that allow the fault to be propogated. Non-active inputs to XOR and *XNOR* gates on the active path must not have a discrepancy. Non-active inputs to gates implementing monotonic functions must either have a discrepancy identical to that of the active input, or have no discrepancy and assume a static critical value (AND and NAND gates require static critical values of true, and OR and NOR gates require static critical values of false). Figure 9 shows two legal critical assignments for a 4-input *AND* gate (the active path is shown by a bold line), and Figure 10 shows an illegal assignment for the same gate. In these illustrations, the notation 0/1 means that the wire takes on the value 0 (false) in the unfaulted circuit, and 1 (true) in the faulted circuit. For example, the OR gate in Figure 2 will cause the clause $(\overline{Act_D} + \overline{E})$ to be added to the formula to be satisfied.

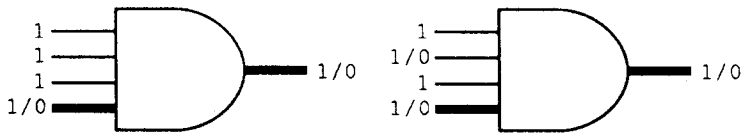

Figure 9: Legal critical assignments

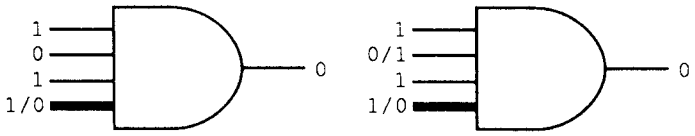

Figure 10: Illegal critical assignment

4.2.3 Non-local Implications

As pointed out by Schulz et al [10], it is possible to explicitly derive non-local implications by examining the reconvergent fanout in a circuit. Figure 11 presents a circuit that demonstrates this idea: If wire B has the value true, wire F has the value true. This means that we know that if wire F has the value false, wire B has the value false. This implication can be discovered by performing a structural analysis of the circuit, or it may be found by analyzing the formula representing the circuit.

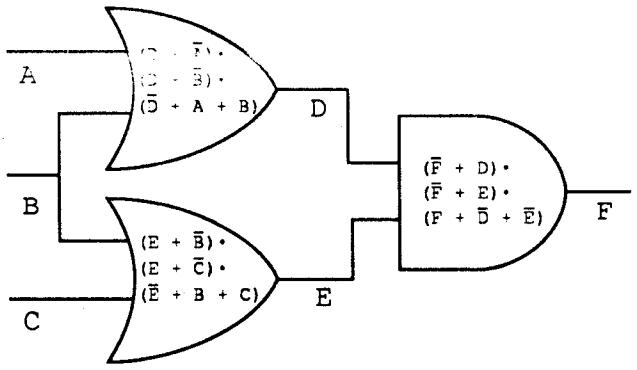

Figure 11: Non-local implications: Add $(\overline{B} + F)$.

We can list all of the non-local implications of a given variable assignment by binding the variable and then noting the implications that use a ternary clause. Any implication that involves a ternary clause must come from reconvergent fanout. If variable B non-locally implies variable F , the clause $(\overline{B} + F)$ may be added to the formula to be satisfied.

5 Experimental Results

We produced test sets for ten sample circuits collected by Franc Brglez [2] and described by a paper in the Proceedings of the 1985 ISCAS Conference.

	Time in Seconds	
Circuit	Random	Algorithmic
C0432	.9	9.0
C0499	1.0	8.2
C0880	2.1	38.1
C1355	9.6	16.2
C1908	10.5	109.3
C2670	5.4	350.0
C3540	38.9	271.3
C5315	8.7	92.1
C6288	130.8	63.1
C7552	24.2	570.4

Table 1: Timing

The test generation was run on a Titan, an experimental RISC machine developed at the Digital Equipment Corporation Western Research Laboratory. A Titan is about 10 times faster than a VAX-11/780. The implementation is written in Modula-2.

Table 1 shows the total time spent for each individual circuit. Tables 2 and 3 report on the faults tested, and proved untestable by the different phases of the program (every fault was either covered or proved untestable). In each case, the largest percentage of the faults were covered by the patterns generated by the first phase of the system. (For the C6288 circuit, every pattern was produced by the pseudo-random phase, and the algorithmic phase was restricted to proving that all remaining faults were uncoverable.) Table 4 shows the number of patterns produced by each phase.

We believe that these experimental results show that the generation of test patterns by extracting a formula and then satisfying it is a general and practical alternative to traditional structural search methods.

6 Acknowledgements

This paper grew out of work begun with Greg Nelson of Digital Equipment Corporation's Systems Research Center. Guidance was provided by the author's advisor, Professor John Hennessy. The wirelists for the Brglez circuits were provided to the author by Jon Udell, who was then a student in Stanford's Center for Reliable Computing.

Circuit	Percentage of Faults		
	Covered	Redundant	Aborted
C0432	99.03	0.97	0.0
C0499	98.64	1.36	0.0
C0880	100.00	0.0	0.0
C1355	99.45	0.55	0.0
C1908	99.48	0.52	0.0
C2670	94.84	5.16	0.0
C3540	95.89	4.11	0.0
C5315	98.76	1.24	0.0
C6288	99.55	0.45	0.0
C7552	98.15	1.85	0.0

Table 2: Coverage

The author was supported by a Digital Equipment Corporation Student Fellowship during the course of this research. In addition to monetary support, Digital, through its Western Research Lab, provided access to many powerful machines and helpful colleagues. Special thanks are due to Mary Jo Doherty and David Boggs for proof-reading this manuscript.

References

[1] Aspvall, B. and Plass, M. and Tarjan, R., "A Linear-time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas," *Information Processing Letters,* vol. 8, October 1979, 121-123.

[2] F. Brglez, H. Fujiwara, "A Neutral Netlist of 10 Combinatorial Benchmark Circuits and a Target Translator in FORTRAN", Special Session on Recent Algorithms for Gate-Level ATPG with Fault Simulation and Their Performance Assessment, *International Symposium on Circuits and Systems* June 1985.

[3] S. A. Cook, "The Complexity of Theorem Proving Procedures," *Proceedings of the Third Anual ACM Symposium of Theory of Computing,* 1971 151-158.

[4] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Transactions on Computers,* vol.C-30, December 1983, 1137-1144.

[5] H. Fuiwara and S. Toida, "The Complexity of Fault Detection Problems for Combinational Logic Circuits," *IEEE Transactions on Computers,* vol. C-31, June 1982, 555-560.

Circuit	Original	Covered by Random	Covered by Algorithmic	Proved Untestable
C0432	411	394	13	4
C0499	588	555	25	8
C0880	748	710	38	0
C1355	1444	1389	47	8
C1908	1740	1452	279	9
C2670	2250	1810	324	116
C3540	3136	2906	101	129
C5315	4761	4665	37	59
C6288	7616	7582	0	34
C7552	7069	6487	451	131

Table 3: Number of Faults

[6] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers,* vol.C-30, March 1981, 215-222.

[7] T. Larrabee, "A Framework for Evaluating Test Pattern Generation Strategies," *Proceedings of the International Conference on Computer Design,* 1989.

[8] E. J. McCluskey, *Logic Design Principles*, Prentice-Hall Publishing, 1986.

[9] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development,* vol. 10, July, 1966, 278-291.

[10] M. H. Schulz, E. Trischler, and T.M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Transactions on CAD*, January 1988, 126-137.

[11] F. F. Sellers, M. Y. Hsiao, And L. W. Bearnson, "Analyzing errors with the Boolean difference," *IEEE Transactions on Computers,* vol.C-17, July 1968, 676-683.

[12] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, T. McCarthy, "Fault Simulation for Structured VLSI," vol. VI, December 1985, 20-32.

Circuit	Number of Patterns	
	Random	Algorithmic
C0432	70	7
C0499	53	19
C0880	93	21
C1355	90	18
C1908	64	110
C2670	94	79
C3540	197	71
C5315	191	27
C6288	47	0
C7552	211	137

Table 4: Number of Patterns

A Framework for Evaluating Test Pattern Generation Strategies

Tracy Larrabee

March 1990

Abstract

This paper presents a formal approach for the analysis of heuristics used in automatic test pattern generation for combinational circuits. We start with a test pattern generation system that constructs a satisfying assignment for a Boolean formula describing the legal set of tests. We then describe heuristics as modifications to the formula or to the satisfier acting on the formula. We provide experimental results for the system as a whole, and for the effects of four heuristics.

This report is a slightly revised version of a paper appearing in the 1989 proceedings of International Conference on Computer Design.

Contents

1 Introduction

A combinational circuit can be tested for the presence of a single stuck-at fault by applying a test pattern that excites a verifiable output response in that circuit. If the fault is present the output will differ from the value required by the test. Given a circuit, the goal of an automatic test pattern generation (ATPG) system is to generate a set of tests that will detect every possible single stuck-at fault in the circuit.

Our method for generating test patterns consists of two parts: First, construct a formula expressing the *Boolean difference* between the unfaulted and faulted circuits. Second, apply a *Boolean satisfiability* algorithm to the resulting formula. This differs from most programs now in use [3, 4, 6, 7], which directly search the circuit data structure instead of constructing a formula from it. Our method is quite general and allows for inclusion of a variety of heuristics—including those used by programs now in use.

This paper describes how four common test pattern generations strategies may be incorporated into our new ATPG system, and how these strategies affect the behavior of the new system. First we give an overview of the complete system and report its performance on the Brglez-Fujiwara test circuits [1]. Next, we describe how the four strategies are incorporated in our system, and how they affect the system's performance.

2 The Framework

Our system consists of a wirelist translator, two phases of test pattern generation, and two simulators [5]. The ATPG method used by the second stage—extracting a formula and then satisfying it—has been the focus of our research. We describe each of the test pattern generation strategies as modifications to this second stage, so we will review it before discussing the strategies.

2.1 Extracting the Formula

A circuit is represented as a directed acyclic graph (dag) with the sources of the graph being the primary outputs of the circuit and the sinks being the primary inputs of the circuit. Each node of the dag, a gate or a fanout point, is tagged with a characteristic logic formula, in 3-element conjunctive normal form (3CNF), describing the behavior of that node in terms of its graph edges (or circuit wires). A formula describing the value of one of the circuit outputs can be obtained by walking the graph starting from the output and taking the conjunction of all of the formulas for the composite nodes. Figure 1 shows an unfaulted circuit and its associated formula.

A faulted circuit is represented by a copy of its associated unfaulted circuit with renamed variables. Because the unfaulted and faulted circuits will have identical behavior except in those nodes that are affected by the fault, only the nodes that lie on a path between the

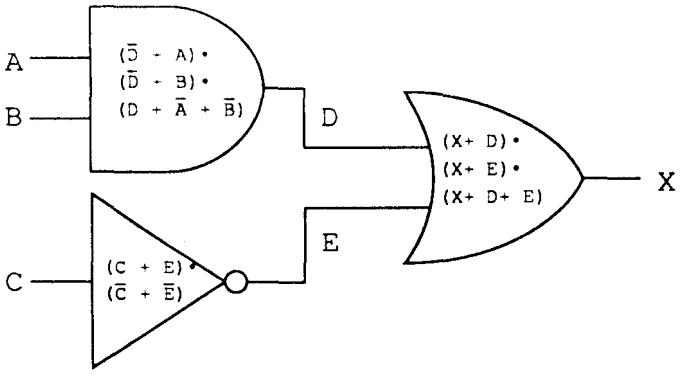

Figure 1: The formula for the output is $(X + \overline{D}) \cdot (X + \overline{E}) \cdot (\overline{X} + D + E) \cdot (\overline{D} + A) \cdot (\overline{D} + B) \cdot (D + \overline{A} + \overline{B}) \cdot (C + E) \cdot (\overline{C} + \overline{E})$.

faulted node and a primary output need to be renamed. Figure 2 shows a faulted circuit corresponding to the unfaulted circuit in Figure 1 with line D stuck-at 1.

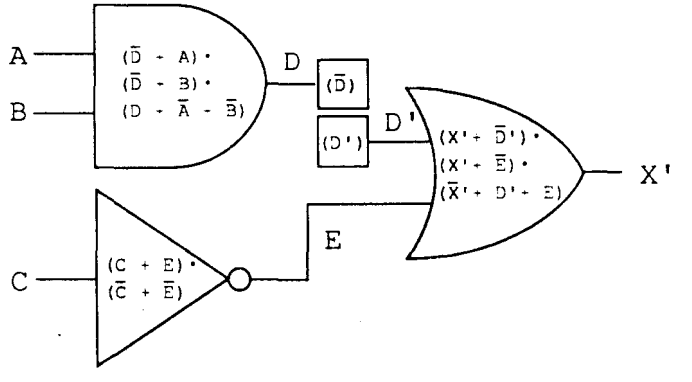

Figure 2: The Formula for the output is $(X'+\overline{D'})\cdot(X'+\overline{E})\cdot(\overline{X'}+D'+E)\cdot(D')\cdot(C+E)\cdot(\overline{C}+\overline{E})$

A fault is detectable if there exists some set of inputs that cause the output of the unfaulted and faulted circuits to differ. A formula describing all legal tests for a given fault can be found by taking the XOR of the two outputs: this formula is called the Boolean difference of a circuit with respect to a fault. The formula for the Boolean difference is obtained by walking the unfaulted circuit, walking the faulted circuit, and taking the conjunction of all formulas encountered together with the formula for the XOR of the two circuit output variables.

Generating a test pattern for the given fault is now a matter of satisfying this formula. If the formula cannot be satisfied, the fault is undetectable. Note that this formula is in conjunctive normal form, since the formulas describing the circuit elements are CNF formulas.

2.2 Satisfying the Formula

The problem of satisfying a 3CNF formula (3SAT) was one of the first to be proven NP-complete [2]. This means that in the worst case a 3SAT problem could take time exponential in the number of its variables, but it does not mean that the average time is as extreme. Our system exploits the fact that 2SAT , unlike 3SAT , is satisfiable in time linear in the number of clauses plus the number of variables. We can do this since a minimum of two-thirds of the clauses in formulas generated as the Boolean difference of a combinational circuit have only two disjuncts (are in 2CNF). If the circuit contains many inverters or gates with fanout greater than one, the percentage of 2CNF clauses increases. In practice, we have found that 80% to 90% of the clauses are 2CNF .

We iterate through the satisfying assignments of the 2CNF portion of the formula until we find a partial binding that is consistent with the ternary clauses and can be extended to satisfy them as well. We give priority in the iteration order to 2SAT assignments that cause many ternary clauses to be narrowed.

An attempt to satisfy a formula may terminate successfully or unsuccessfully. An attempt is successful if the formula is satisfied or it is proved that the formula is unsatisfiable; an attempt is unsuccessful if too much time has passed without reaching a successful conclusion.

2.3 Simulation

The first phase of test pattern generation produces random test patterns that are simulated against each fault. The first phase allows us to cover (generate a test for) 80% to 90% of the faults in less than one tenth of the time required by the second phase. We take advantage of the available word operations to generate and simulate patterns 32 at a time. This approach to simulation is similar to one reported by Waicukauski et al [9], who named it Parallel Pattern, Single Fault Propagation (PPSFP) simulation. Each pattern generated by extracting and satisfying a formula is simulated against each of the remaining faults using a simple single pattern, single fault propagation fault simulator.

2.4 System Performance

In Table 1 we present the results of our base-level system (which exploits the first three heuristics described in the next section but not the last one) when run on the ten sample circuits collected by Franc Brglez and described by a paper in the Proceedings of the 1985 ISCAS Conference. The first column of the table reports how many seconds elapsed processing the circuit, and the last three report on the percentage of total faults for which we generate a test (covered), prove that no test exists (proved redundant), or fail to test or prove untestable (aborted).

The test generation was run on a Titan, an experimental RISC machine developed at the Digital Equipment Corporation Western Research Laboratory. A Titan is about 10

times faster than a VAX-11/780. The implementation is written in Modula-2. The most memory that the system has ever used is 15 megabytes (on the largest input set).

Our system either tests or proves untestable every fault in the Brglez-Fujiwara benchmark.

	Time	Percent of Faults		
Circuit	(seconds)	Covered	Redundant	Aborted
C0432	10.5	99.03	0.97	0.0
C0499	10.2	98.64	1.36	0.0
C0880	41.3	100.00	0.00	0.0
C1355	27.6	99.45	0.55	0.0
C1908	127.3	99.48	0.52	0.0
C2670	365.5	94.84	5.16	0.0
C3540	314.5	95.89	4.11	0.0
C5315	107.8	98.76	1.24	0.0
C6288	200.8	99.55	0.45	0.0
C7552	603.3	98.15	1.85	0.0

Table 1: Base-level System Performance

3 The Strategies

In this section we will describe four heuristics used in test pattern generation, describe how they can be incorporated into our system, and report their effect on the system's performance.

3.1 Speeding Fault Propagation

We wish to take advantage of heuristics, first implemented for the D-Algorithm [6], that order their operations in an attempt to propagate the effect of the fault to an output. For a fault to be successfully propagated to an output, there must be at least one path from the fault to that output such that every line on that path has a discrepancy (the unfaulted value differs from the faulted value). We would like to find one such path, which we call an active path. Each line that is a member of the active path is an active line. Every active line must have a discrepancy, but not all lines with discrepancies are active wires.

To find an active path, we add clauses describing the restrictions of the active path to the formula to be satisfied. For each line that lies between the fault and a primary output we allocate a variable (called the active variable for the wire), and for each gate that lies between the fault and a primary output we add several clauses. These clauses ensure that if

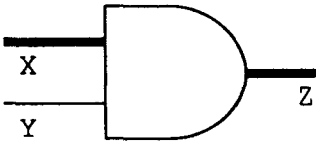

Figure 3: If X is active, Z must be active: $(\overline{Act_X} + Act_Z)$

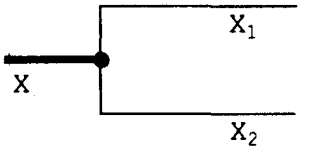

Figure 4: If X is active, either X_1 or X_2 must be active: $(\overline{Act_X} + Act_{X_1} + Act_{X_2})$

an input to a single output node is active, the output is active; if an input to a multi-output gate is active, one of the outputs is active; the fault site is always active. Figures 3 and 4 show examples of these clauses.

We also add clauses that guarantee that if a line is on the active path, the wire must have different faulted and unfaulted values. For example, for the circuit in Figure 2 we allocate the variables Act_D and Act_X , and add the clauses $(\overline{Act_D} + D + D')$, $(\overline{Act_D} + \overline{D} + \overline{D'})$, $(\overline{Act_X} + X + X')$, and $(\overline{Act_X} + \overline{X} + \overline{X'})$ to the formula to be satisfied.

	Time	Percent of Faults		
Circuit	(seconds)	Covered	Redundant	Aborted
C0432	10.6	99.03	0.38	0.76
C0499	155.1	98.64	0.00	1.36
C0880	75.9	99.87	0.00	0.13
C1355	310.2	99.45	0.00	0.55
C1908	525.6	98.79	0.52	0.69
C2670	2234.6	94.49	1.08	4.53
C3540	37061.0	93.34	1.72	4.94
C5315	1826.7	98.48	0.71	0.81
C6288	84535.1	99.55	0.24	0.21
C7552	25983.5	97.66	0.14	2.20

Table 2: System Performance without Active Clauses

The effectiveness of this strategy is shown in Table 2, which shows the system's performance when no clauses containing active variables are added to the satisfied formula. Clearly the active clauses are a very important addition to our system.

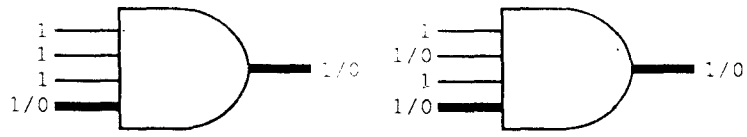

Figure 5: Legal critical assignments

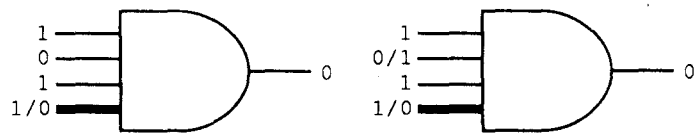

Figure 6: Illegal critical assignment

Many systems place great importance on preprocessing the circuit structure to derive the unique sensitization points (points of total reconvergence) in the circuit [3, 7]. We can use the active implication clauses (as shown in Figures 3 and 4) for the circuit as a whole to determine all of the unique sensitization points. We have experimented with explicitly adding information concerning the unique sensitization points, but we have found that this information never improves performance over merely adding the active implication clauses and letting the satisfier derive the active path as it runs.

3.2 Requiring critical values

If a node is on the active path, we know that that node must propagate the discrepancy. This means that the non-active inputs to the node must have taken on critical values that allow the fault to be propagated. Non-active inputs to XOR and XNOR gates on the active path must not have a discrepancy. Non-active inputs to gates implementing monotonic functions must either have a discrepancy identical to that of the active input, or have no discrepancy and assume a static critical value (AND and NAND gates require static critical values of true, and OR and NOR gates require static critical values of false). Figure 5 shows two legal critical assignments for a 4-input AND gate (the active path is shown by a bold line), and Figure 6 shows illegal assignments for the same gate. In these illustrations, the notation 0/1 means that the wire takes on the value 0 (false) in the unfaulted circuit, and 1 (true) in the faulted circuit.

We can see from Table 3, which reports on the performance of the system when no clauses requiring critical value assignments are included, that this heuristic has little effect on system performance. Even though the clauses added to insure critical value assignments can provide for many more assignments of values indispensable for propagating the fault than a structural analysis during a preprocessing phase, the information added by the critical clauses is usually easily derived by the satisfier.

Circuit	Time (seconds)	Percent of Faults Covered	Redundant	Aborted
C0432	11.6	99.03	0.97	0.00
C0499	9.9	98.64	1.36	0.00
C0880	41.5	100.00	0.00	0.00
C1355	27.2	99.45	0.55	0.00
C1908	225.6	99.43	0.51	0.06
C2670	582.0	94.84	5.17	0.00
C3540	340.5	95.89	4.11	0.00
C5315	101.8	98.76	1.24	0.00
C6288	185.9	99.55	0.45	0.00
C7552	886.5	98.15	1.82	0.03

Table 3: System Performance without Critical Clauses

3.3 Non-local Implications

As pointed out by Schulz et al [7], it is possible to explicitly derive non-local implications by examining the reconvergent fanout in a circuit. Figure 6 presents a circuit that demonstrates this idea: If wire B has the value true, wire F has the value true; therefore if wire F has the value false, wire B has the value false. This implication can be discovered by performing a structural analysis of the circuit, or it may be found by analyzing the formula representing the circuit.

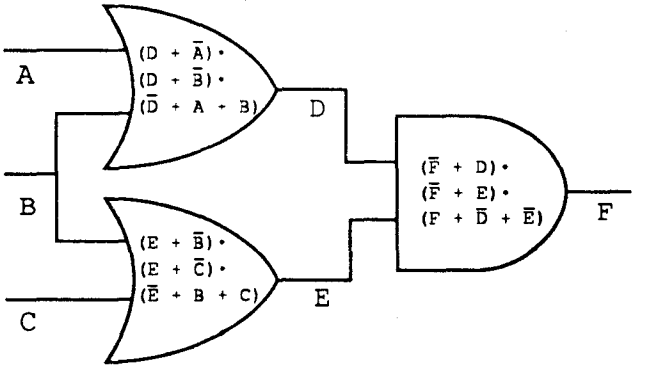

Figure 7: Non-local implications: Add $(\overline{B} + F)$.

We can list all of the non-local implications of a given variable assignment by binding the variable and then noting the direct implications that use a ternary clause. Any implication that involves a ternary clause must come from reconvergent fanout. All non-local implications can be added to the formula to be satisfied. We only add the non-local im-

plications if the satisfier fails to satisfy or prove unsatisfiable the original formula. Table 4 reports that the non-local implications were needed for only one of the benchmark circuits.

Circuit	Time (seconds)	Percent of Faults Covered	Redundant	Aborted
C0432	10.5	99.03	0.97	0.00
C0499	10.2	98.64	1.36	0.00
C0880	41.3	100.00	0.00	0.00
C1355	27.6	99.45	0.55	0.00
C1908	127.3	99.48	0.52	0.00
C2670	301.5	94.84	4.62	0.44
C3540	314.5	95.89	4.11	0.00
C5315	107.8	98.76	1.24	0.00
C6288	200.8	99.55	0.45	0.00
C7552	603.3	98.15	1.85	0.00

Table 4: System Performance without Non-local Implications

3.4 Avoiding Fanout-Free Subcircuits

We can also take advantage of topological features first exploited by the FAN algorithm [3]. By stopping its backtrace operation at *head lines*, wires guaranteed not to be involved in any fanout loop, FAN restricts its search-space size. We can restrict our search space in a manner similar to FAN's method.

We say that variable V *determines* variable W if both assignments to V cause W to appear only negated or only unnegated. In this case, we may remove all clauses containing W from the formula and postpone the assignment of W until after the final assignment of V has been made.

In the circuit shown in Figure 8, F determines A and E determines B and C. The characteristic formula for G would normally consist of 13 clauses, but the removal of all clauses containing variables A , B , and C will leave only 8 clauses in the remaining formula.

Unfortunately, our technique as stated will not remove as many variables as may be safely removed from the formula: the technique will not remove variables corresponding to input wires for XOR gates not involved in reconvergent fanout loops. We are working on a modification that will correctly deal with XOR gates.

Table 5 shows us that the FAN heuristic does not help our system. We find this result surprising, and have not completely determined the underlying reason. We speculate that our satisfier does not spend much time in the portion of the search tree being eliminated by the FAN heuristic, but we need to design further experiments to confirm this. The deficiency of our technique in the presence of XOR gates might be partially blamed for our

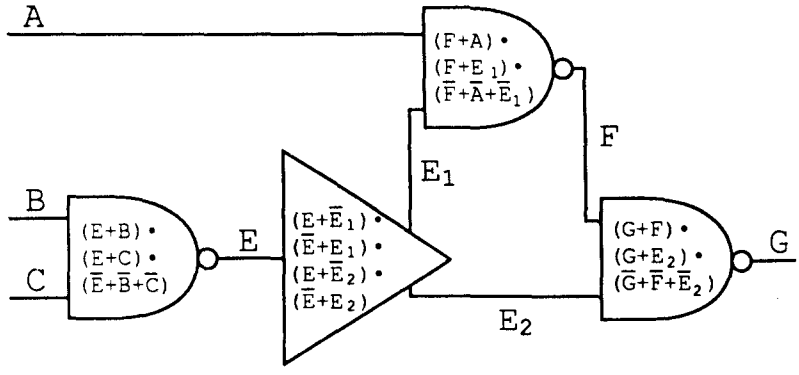

Figure 8: The Formula for G can be reduced to $(E + \overline{E_1}) \cdot (\overline{E} + E_1) \cdot (E + \overline{E_2}) \cdot (\overline{E} + E_2) \cdot (F + E_1) \cdot (G + E_2) \cdot (G + F) \cdot (\overline{G} + \overline{E_2} + \overline{G})$.

lack of success, but XOR gates appear only in the two smallest circuits, so this cannot be an acceptable explanation.

4 Conclusions

We have presented a flexible system that generates test patterns by extracting formulas and then satisfying them. Using this system, we have experimented with four heuristics that have been reported in the literature: fault propagation acceleration, requiring critical values, adding non-local implications, and avoiding fanout-free subcircuits. We found that the first and third of these were very valuable, the second was of some slight value on large circuits, and the last was of no value.

5 Acknowledgements

This paper grew out of work begun with Greg Nelson of Digital Equipment Corporation's Systems Research Center. Guidance was provided by the author's advisor, Professor John Hennessy. The wirelists for the Brglez-Fujiwara circuits were provided to the author by Jon Udell, who was then a student in Stanford's Center for Reliable Computing.

The author was supported by a Digital Equipment Corporation Student Fellowship during the course of this research. The author wishes to thank Mary Jo Doherty for once again proof-reading one of her manuscripts.

References

[1] F. Brglez, H. Fujiwara, "A Neutral Netlist of 10 Combinatorial Benchmark Circuits and a Target Translator in FORTRAN," Special Session on Recent Al-

Circuit	Time (seconds)	Percent of Faults Covered	Redundant	Aborted
C0432	12.7	99.03	0.97	0.0
C0499	13.1	98.64	1.36	0.0
C0880	55.3	100.00	0.00	0.0
C1355	33.1	99.45	0.55	0.0
C1908	149.5	99.48	0.52	0.0
C2670	244.1	94.84	5.16	0.0
C3540	403.3	95.89	4.11	0.0
C5315	131.9	98.76	1.24	0.0
C6288	218.8	99.55	0.45	0.0
C7552	742.1	98.15	1.85	0.0

Table 5: System Performance with FAN-Reduced Formulas

gorithms for Gate-Level ATPG with Fault Simulation and Their Performance Assessment, *International Symposium on Circuits and Systems* June 1985.

[2] S. A. Cook, "The Complexity of Theorem Proving Procedures," *Proceedings of the Third Anual ACM Symposium of Theory of Computing,* 1971 151-158.

[3] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Transactions on Computers,* vol.C-30, December 1983, 1137-1144.

[4] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers,* vol.C-30, March 1981, 215-222.

[5] T. Larrabee "Efficient Generation of Test Patterns Using Boolean Difference," *Proceedings of the International Test Conference,* August 1989.

[6] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development,* vol. 10, July, 1966, 278-291.

[7] M. H. Schulz, E. Trischler, and T.M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Transactions on CAD*, January 1988, 126-137.

[8] F. F. Sellers, M. Y. Hsiao, And L. W. Bearnson, "Analyzing errors with the Boolean difference," *IEEE Transactions on Computers,* vol.C-17, July 1968, 676-683.

[9] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, T. McCarthy, "Fault Simulation for Structured VLSI," vol. VI, December 1985, 20-32.