Test Pattern Generation Using Boolean Satisfiability

Tracy Larrabee*

Abstract

This article describes the Boolean satisfiability method for generating test patterns for single stuck-at faults in combinational circuits. This new method generates test patterns in two steps: 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. This approach differs from previous methods now in use, which search the circuit structure directly instead of constructing a formula from it. The new method is general and effective: it allows for the addition of heuristics used by structural search methods, and it has produced excellent results on popular test pattern generation benchmarks.

1 Introduction

To produce reliable computer systems, defect-free components must be available. Automatic test pattern generation (ATPG) systems distinguish defective components from defect-free components by generating input sets that cause the outputs of a component under test to be different if the component is defective than if it is defect-free. Existing algorithmic ATPG systems for single stuck-at faults in combinational circuits fall into two classes: the structural methods, which perform a topological search of the circuit under test, and the algebraic methods, which generate test patterns by manipulating algebraic formulas.

The Boolean satisfiability method is a new algorithm for test pattern generation for single stuck-at faults in combinational circuits that is neither a purely structural method nor an algebraic one[8, 9]. This method is not only practical but performs better than most systems now in use. Before describing the Boolean satisfiability method in detail we will briefly review the two classes of existing methods.

Structural search methods use a data structure representing the circuit to be tested. To generate a test pattern, they assign values that cause a discrepancy at the faulted line (the *fault site*) and then search for consistent values for all circuit lines such that the discrepancy is visible at a circuit output. The most successful ATPG systems use structural search methods. Of these, the most notable are the D-algorithm, Podem, FAN, and Socrates [6, 7, 16, 17].

Instead of performing a search on a data structure representing a circuit, algebraic methods produce an equation describing all possible tests for a particular fault and then simplify the resulting equation. The most famous algebraic method is the Boolean difference method.

The Boolean difference of any function F with respect to its variable x_i is equal to

$$F(x_1, ..., x_{i-1}, 0, x_{i+1}, ..., x_n) \oplus F(x_1, ..., x_{i-1}, 1, x_{i+1}, ..., x_n).$$

The notation for this quantity is dF/dx_i. The set of tests for x_i stuck at 0 is $X_i \cdot dF/dx_i$ and the set of tests for x_i stuck at 1 is $\overline{X_i} \cdot dF/dx_i$ (where X_i is the function representing the output of the subcircuit with output at x_i.

Note that the validity of the formula does not change if we introduce intermediate variables for any subformulas of $F(x_1, ..., x_n)$. If we introduce an intermediate variable, we do not change the permissable values for the original variables. This changes the solution set, but only because each satisfying binding will also contain bindings for the introduced variables that are consistent with the original variables. We could introduce intermediate variables for every line in the circuit.

*The author's current address is Computer Engineering Board of Studies, University of California, Santa Cruz, California 95064. When this work was performed, she was with the Department of Computer Science at Stanford University, Stanford, California 94305. This work was supported by a Digital Equipment Corporation Student Fellowship.

Once the formula using the Boolean difference is obtained, it is simplified using the basic laws of Boolean algebra or by using identities specific to the Boolean difference[1]. The tedious nature of the algebraic manipulations involved in solving formulas using the Boolean difference led to its disfavor as a practical tool for test pattern generation [12, 14, 11].

The Boolean satisfiability method generates a formula equivalent to that of the Boolean difference method, but instead of performing symbol manipulation, it runs a *Boolean Satisfiability* algorithm on the formula. Nemesis, an ATPG system using the new method is quite practical: it correctly tests or proves untestable every fault in the ISCAS-85 (Brglez-Fujiwara) benchmark set [3].

2 The Boolean Satisfiability Method

To generate a test pattern for a single fault, first extract a formula that defines the set of test patterns that detect the fault and then use a Boolean satisfiability algorithm to satisfy the formula.

2.1 Extracting the Formula

A directed acyclic graph represents the topological description of the circuit. The nodes of the graph are circuit inputs, outputs, gates and fanout points, the edges of the graph are circuit lines (wires), the sources of the graph are circuit outputs, and the sinks of the graph are the circuit inputs. Every edge has an associated variable. Figure 1 shows a circuit and its associated DAG.

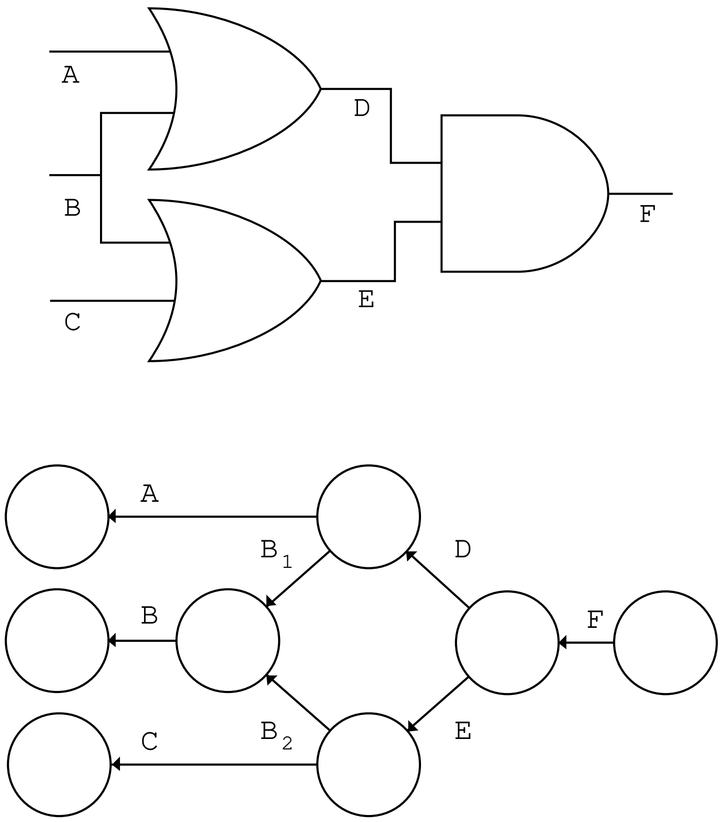


Figure 1: A circuit and its associated DAG

Every node of the DAG is tagged with a formula that represents the function performed by that gate or fanout point. For example, an inverter with an input X and an output Y will be be tagged with the formula

$Y = \overline{X}$; an AND gate with the inputs X and Y and the output Z will be tagged with the formula $Z = X \cdot Y$. Every node has a formula that contains only variables for its incoming and outgoing edges.

2.1.1 Translating Formulas into CNF

We will use conjunctive normal form, or CNF (also known as product of sums). Formulas written in CNF are easily manipulated programmatically. To get the CNF formula for an AND gate, we start with the formula

$$Z = X \cdot Y.$$

Because the formula $P = Q$ is logically equivalent to $(P \rightarrow Q) \cdot (Q \rightarrow P)$, we transform our original equality into

$$(Z \rightarrow (X \cdot Y)) \cdot ((X \cdot Y) \rightarrow Z).$$

Next, we transform all implications into disjunctions by using the fact that $P \rightarrow Q$ is logically equivalent to $\overline{P} + Q$ to get the formula

$$(\overline{Z} + X) \cdot (\overline{Z} + Y) \cdot (\overline{X} + \overline{Y} + Z).$$

This formula evaluates to 1 if and only if the values of the variables are consistent with the truth table for an AND gate. For comparison, the disjunctive normal form (sum of products) version of the same formula is

$$(X \cdot Y \cdot Z) + (\overline{X} \cdot Y \cdot \overline{Z}) + (X \cdot \overline{Y} \cdot \overline{Z}) + (\overline{X} \cdot \overline{Y} \cdot \overline{Z}).$$

In CNF formulas, one sum is called a clause. Clauses with only one, two, or three elements are unary, binary, or ternary clauses, respectively. A formula with no ternary clauses is said to be in 2CNF (2-element conjunctive normal form).

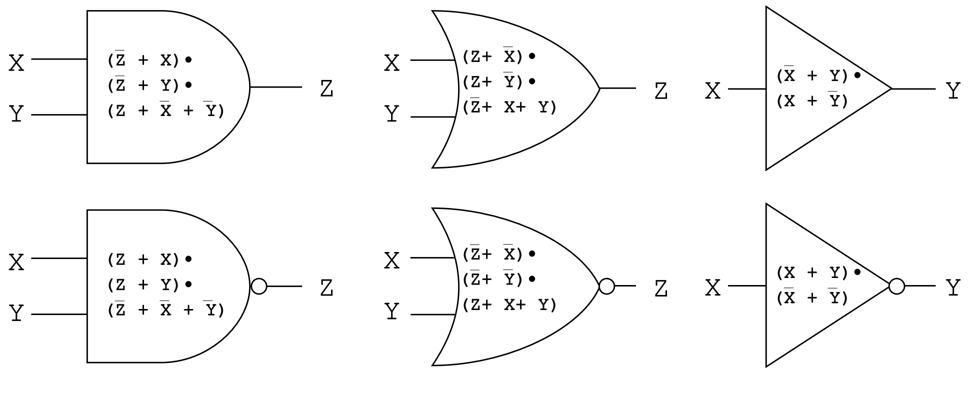


Figure 2: The formulas for the basic gates

The CNF formulas for the basic gates are shown in Figure 2, but the gates need not be basic to be included in this scheme: With the introduction of new variables, the CNF form of any formula can be produced in time and space linear in the size of the original formula. For example, the CNF formula for an AND gate with inputs X, Y, and W and output Z, is

$$(\overline{Z} + X) \cdot (\overline{Z} + Y) \cdot (\overline{Z} + W) \cdot (\overline{X} + \overline{Y} + \overline{W} + Z).$$

The formula for an XOR gate with inputs X and Y and output Z is

$$(\overline{X} + Y + Z) \cdot (X + \overline{Y} + Z) \cdot (\overline{X} + \overline{Y} + \overline{Z}) \cdot (X + Y + \overline{Z}).$$

2.1.2 Formulas for Unfaulted and Faulted Circuits

Because each gate and fanout point is tagged with a formula that must be independantly satisfied, we can extract a characteristic formula for any circuit output (or subcircuit output) by starting at the output and walking the graph, taking the conjunction of all of the formulas for the visited nodes. Since the formula for every component must be independantly satisfiable, the conjunction of the formulas must also be satisfiable. Figure 3 shows a circuit with each gate labeled by its characteristic formula. The formula for the output of this circuit is

$$(\mathrm{X} + \overline{\mathrm{D}}) \cdot (\mathrm{X} + \overline{\mathrm{E}}) \cdot (\overline{\mathrm{X}} + \mathrm{D} + \mathrm{E}) \cdot (\overline{\mathrm{D}} + \mathrm{A}) \cdot (\overline{\mathrm{D}} + \mathrm{B}) \cdot (\mathrm{D} + \overline{\mathrm{A}} + \overline{\mathrm{B}}) \cdot (\mathrm{C} + \mathrm{E}) \cdot (\overline{\mathrm{C}} + \overline{\mathrm{E}}).$$

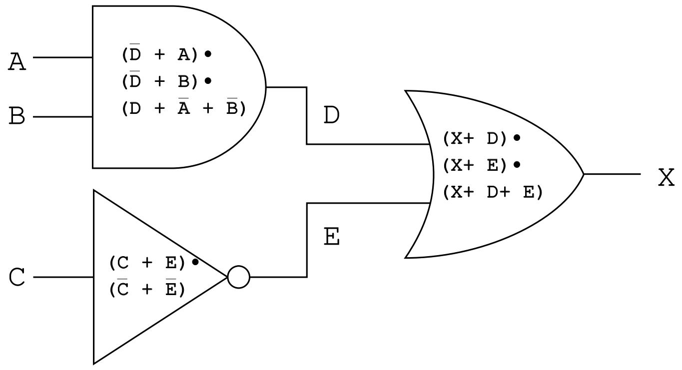


Figure 3: Combinational circuit with labeled gates

We can represent a faulted version of an unfaulted circuit by making a copy of the circuit, renaming the variables, and inserting two new nodes that represent the presumed disrupted connection in the faulted circuit. That is, if the circuit has the fault we want to test for, one value will be generated at the fault site, but another value will forwarded on to the rest of the circuit. We tag the new nodes with unary clauses that indicate the behavior of the fault we are interested in. For example, Figure 4 shows the faulted version of the circuit in Figure 3. Because wire D is stuck-at 1, we add the formula (D') to the node representing the faulted behavior at the fault site, and we add the formula $(\overline{\mathrm{D}})$ to the node representing the correct behavior at the fault site.

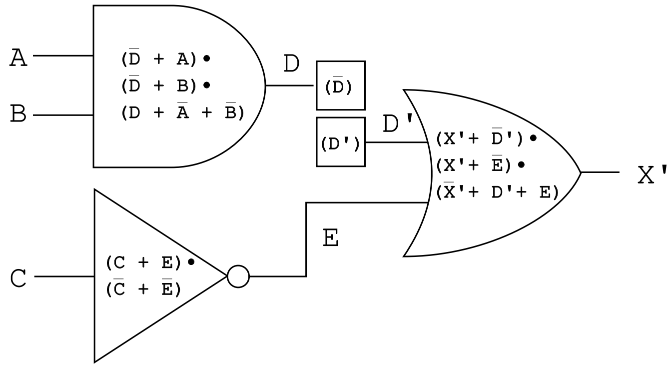


Figure 4: Circuit of Figure 2 with D stuck at 1

Because the unfaulted and faulted circuits will have identical behavior except at 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 circuit output need to be renamed.

We can extract a formula for the faulted output in the same way as we extracted a formula for the unfaulted circuit: by starting at the faulted output, walking the DAG, and taking the conjunction of all encountered nodes of the DAG. The formula for the faulted circuit of Figure 4 is

$$(\mathrm{X}' + \overline{\mathrm{D}'}) \cdot (\mathrm{X}' + \overline{\mathrm{E}}) \cdot (\overline{\mathrm{X}'} + \mathrm{D}' + \mathrm{E}) \cdot (\mathrm{D}') \cdot (\mathrm{C} + \mathrm{E}) \cdot (\overline{\mathrm{C}} + \overline{\mathrm{E}}).$$

We need not include the clause $(\overline{\mathrm{D}})$ in the formula for the faulted circuit because of the implied discontinuity at the fault site.

To test for the given fault, we need only find a set of inputs that cause the faulted output to differ from the unfaulted output. We will have a formula for all possible tests if we take the conjunction of the two extracted formulas and add an additional formula for the XOR of the faulted and unfaulted output. Using BD to represent the result of the final XOR, the formula resulting from the XOR of the output of the unfaulted circuit of Figure 3 and the faulted circuit of Figure 4 is

$$\begin{array}{l}(\mathrm{X}' + \overline{\mathrm{D}'}) \cdot (\mathrm{X}' + \overline{\mathrm{E}}) \cdot (\overline{\mathrm{X}'} + \mathrm{D}' + \mathrm{E}) \cdot (\mathrm{D}') \cdot (\mathrm{C} + \mathrm{E}) \cdot (\overline{\mathrm{C}} + \overline{\mathrm{E}}) \cdot \\ (\mathrm{X} + \overline{\mathrm{D}}) \cdot (\mathrm{X} + \overline{\mathrm{E}}) \cdot (\overline{\mathrm{X}} + \mathrm{D} + \mathrm{E}) \cdot (\overline{\mathrm{D}} + \mathrm{A}) \cdot (\overline{\mathrm{D}} + \mathrm{B}) \cdot (\mathrm{D} + \overline{\mathrm{A}} + \overline{\mathrm{B}}) \cdot \\ (\overline{\mathrm{X}} + \mathrm{X}' + \mathrm{BD}) \cdot (\mathrm{X} + \overline{\mathrm{X}'} + \mathrm{BD}) \cdot (\overline{\mathrm{X}} + \overline{\mathrm{X}'} + \overline{\mathrm{BD}}) \cdot (\mathrm{X} + \mathrm{X}' + \overline{\mathrm{BD}}),\end{array}$$

where the first line is contributed by the faulted circuit, the second line is contributed by the unfaulted circuit, and the third line is contributed by the final XOR. Figure 5 shows the circuit form of the formula to be satisfied. There are several clauses that appear in both the formulas for the faulted circuit and the unfaulted circuit, but they need not be repeated because AND is idempotent.

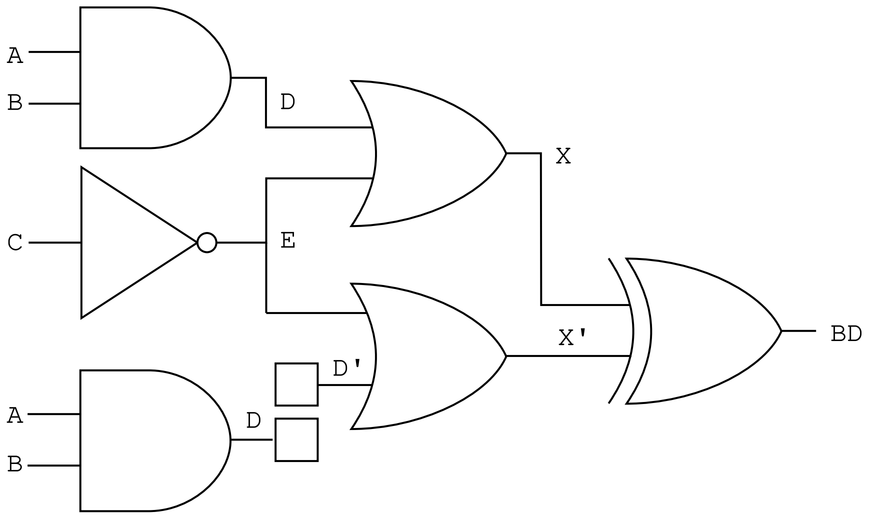


Figure 5: The XOR of the faulted and unfaulted circuits must be 1

The extracted formula is equivalent to the formula that would be produced by the Boolean difference method, in the sense that they are both satisfiable or both unsatisfiable: every set of satisfiable bindings for the formula produced by the Boolean difference method is consistent with a satisfying binding for the Boolean satisfiability method, and every set of satisfiable bindings for our formula is a superset of a satisfying binding for the formula produced by the Boolean difference method. The formula extracted by our system is not exactly the same as one that would be produced by the Boolean difference method because our formula has extra variables in it. These redundancies will be helpful in finding a satisfying assignment for the formula.

2.2 Satisfying the Formula

The problem of satisfying a CNF formula, SAT, is an *NP*-complete problem [4]. We have transformed one problem that in the worst case will take exponential time in the number of its circuit inputs into another problem that in the worst case will take exponential time in the number of its variables. Fortunately, the class of formulas generated by combinational circuits is an interesting sub-class of all CNF formulas, and we can use this fact to try to avoid the worst case behavior of SAT. Many researchers have recognized that the average behavior of a SAT algorithm can be improved dramatically if the set of formulas to be solved fit a restricted profile [5, 15]. The set of formulas produced by combinational circuits fits such a restricted profile.

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 (the basic unate gates are pictured in Figure 2). 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 [2]. 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.

2.2.1 Using 2SAT to Solve SAT

We use an algorithm from the 1970's for satisfying a 2CNF formula [2]. The first step is to construct an *implication graph*. Each 2CNF clause $(\mathrm{X} + \mathrm{Y})$ can be viewed as two implications: $\overline{\mathrm{X}} \rightarrow \mathrm{Y}$ and $\overline{\mathrm{Y}} \rightarrow \mathrm{X}$. 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 $\overline{\mathrm{X}}$. For every 2CNF clause $(\mathrm{X} + \mathrm{Y})$ there are two directed edges in the graph: one from $\overline{\mathrm{X}}$ to Y, and one from $\overline{\mathrm{Y}}$ 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 labeled 1 (true) to precede (or imply) a node labeled 0 (false).

Before we label the graph, we can simplify it by reducing each *strongly connected component*, a maximal set of nodes in a graph such that every node in the set is reachable from every other node in the set, to a single node. If any strongly connected component contains both a literal and its negation, the formula is unsatisfiable (because each strongly connected component represents a set of variables that are in an equivalence class). After each strongly connected component is reduced to a single node, the graph will not contain any cycles. Now we can find at least one satisfying binding for the 2CNF portion formula: First we visit the vertices in any topological order. For each variable, if the negated variable appears before the un-negated variable in the topological order, we bind the variable to 0 (false); otherwise, we bind the variable to 1 (true). We will discuss the details of iterating through all 2SAT bindings in the next section.

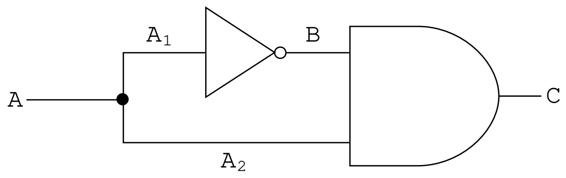
As an example of how 2SAT works, consider the small circuit in Figure 6. Imagine that we wished to iterate through all possible bindings to the variables A, A_1, A_2, B, and C. The formula for the circuit is

$$\begin{aligned}&(\overline{\mathrm{A}} + \mathrm{A}_1) \cdot (\mathrm{A} + \overline{\mathrm{A}_1}) \cdot (\overline{\mathrm{A}} + \mathrm{A}_2) \cdot (\mathrm{A} + \overline{\mathrm{A}_2}) \cdot \\ &(\overline{\mathrm{A}}_1 + \mathrm{B}) \cdot (\overline{\mathrm{A}_1} + \overline{\mathrm{B}}) \cdot (\overline{\mathrm{C}} + \mathrm{A}_2) \cdot (\overline{\mathrm{C}} + \mathrm{B}) \cdot \\ &(\overline{\mathrm{A}_2} + \overline{\mathrm{B}} + \mathrm{C}),\end{aligned}$$

where the first two lines are the 2CNF portion of the formula and the last line is the ternary portion of the formula. The implication graph of the 2CNF portion of this formula is shown in Figure 6. The graph has two strongly connected components: $\{\overline{\mathrm{A}_2}, \overline{\mathrm{A}}, \overline{\mathrm{A}_1}, \mathrm{B}\}$ and its complement, $\{\mathrm{A}_2, \mathrm{A}, \mathrm{A}_1, \overline{\mathrm{B}}\}$. We replace these strongly connected components with the unit nodes E_1 and $\overline{\mathrm{E}_1}$, which results in the graph shown in Figure 7. The final graph clearly shows that C implies $\overline{\mathrm{C}}$, and therefore C must be bound to 0. In the circuit from which the formula was extracted, C is equal to $\mathrm{A} \cdot \overline{\mathrm{A}}$, so it is reassuring that the system can determine that C must be bound to 0. Given the binding of C, only one unbound node in the graph remains, and it can assume either Boolean value and remain consistent with the ternary clause.

2.2.2 Iterating through 2SAT Bindings

We have 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 bound to 1 has a directed path to a node bound to 0.



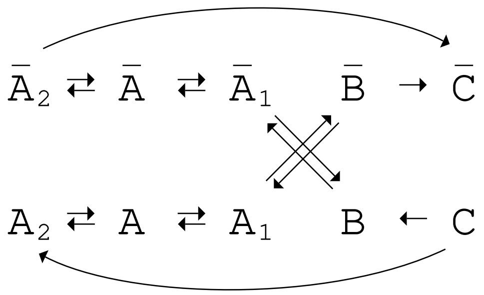


Figure 6: A simple circuit and its implication graph

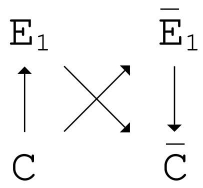


Figure 7: The reduced implication graph

Clearly there are many such assignments, but we want to construct a 2SAT assignment that is consistent with the ternary clauses. We will do this by defining an order for the 2SAT assignments and then constructing each assignment only so far as it is consistent with the ternary clauses.

We order the 2SAT assignments by ordering the variables that appear in the 2CNF clauses. This defines a total order on the 2SAT solutions: One total assignment precedes another if the n-bit binary number representing the values of the variables (in the previously fixed order) precedes the n-bit binary number for the other assignment. For partial assignments, we use lexicographic order with unbound variables treated as less than 0. We can consider the 2SAT solutions in either ascending or descending order, but we will assume (without loss of generality) that we consider them in descending order.

We start with V, the array of 2CNF variables (initially unbound), i, which points to the first unbound variable in the array (initially set to 0), and *dir*, which keeps track of whether or not we are backtracking (initially set to Forward). We call the *current prefix* of V the sequence of bound values *V[0], V[1], ..., V[i-1]*. All elements of V greater than or equal to 0 and less than i are bound. Our goal is to either find an assignment for the variables in V that is consistent with the ternary clauses or to prove that no such binding exists. Figure 8 shows a loop (with loop invariants) that achieves this goal.

Figures 9 through 11 show an example of 2SAT iteration. In Figure 9 we show an abbreviated version of a familiar constraint problem: the N-Queens problem. In this problem, we wish to place two queens on a board with two squares on one side and three on the other such that neither queen attacks the other. We can translate this problem into CNF in the following manner:

Each of the six squares is associated with a variable A, B, C, D, E, or F that is bound to 1 if a queen is placed in the square with the associated label and 0 if no queen is placed in that square. We can require that a queen must be placed in each row through two ternary *placement* clauses, and we can prevent a queen from attacking another by adding 13 binary *attack* clauses. For example, the attack clause $(\overline{\mathrm{A}} + \overline{\mathrm{B}})$ prevents

The loop invariant:

1. Any binding that precedes the current prefix falsifies the formula.
2. If dir = Backward, any complete binding that extends the current prefix falsifies the formula.
3. If dir = Forward, the current prefix is consistent with the formula.
4. Any variables that are bound but are not part of the current prefix are implied by the current prefix.

V ← all Unbound; i ← 0; dir ← Forward;

```
loop
    if dir = Forward then
        while i ≠ size(V) and V[i] is bound do i ← i + 1 end;
        if i = size(V) then exit successfully end;
        V[i-1] ← 0;
        Set direct implications of V[i-1];
        i ← i + 1
    elsif dir = Backward then
        if i = 0 then exit unsuccessfully end;
        temp ← V[i-1];
        Undo direct implications of V[i-1];
        V[i-1] ← Unbound;
        if temp = 0 then
            V[i-1] ← 1;
            Set direct implications of V[i-1]
        else
            i ← i - 1
        end
    endif
    if no clause falsified then dir ← Forward else dir ← Backward end
endloop
```

Setting or undoing direct implications: we keep a count of how many times each variable is set to 1 or set to 0; a variable with a count of 3 has been forced to 1 three times and a variable with a count of -3 has been forced to 0 three times. A variable is only bound when its count changes from 0 and is only unbound if its count goes to 0.

Figure 8: 2SAT iteration loop

queens from being simultaneously placed in squares A and B. The complete list of clauses is

$$
\begin{aligned}
&(\mathrm{A}+\mathrm{B}+\mathrm{C})\cdot(\mathrm{D}+\mathrm{E}+\mathrm{F})\cdot\\
&(\overline{\mathrm{A}}+\overline{\mathrm{B}})\cdot(\overline{\mathrm{A}}+\overline{\mathrm{C}})\cdot(\overline{\mathrm{A}}+\overline{\mathrm{D}})\cdot(\overline{\mathrm{A}}+\overline{\mathrm{E}})\cdot(\overline{\mathrm{B}}+\overline{\mathrm{C}})\cdot(\overline{\mathrm{B}}+\overline{\mathrm{D}})\cdot(\overline{\mathrm{B}}+\overline{\mathrm{E}})\cdot(\overline{\mathrm{B}}+\overline{\mathrm{F}})\cdot\\
&(\overline{\mathrm{C}}+\overline{\mathrm{E}})\cdot(\overline{\mathrm{C}}+\overline{\mathrm{F}})\cdot(\overline{\mathrm{D}}+\overline{\mathrm{E}})\cdot(\overline{\mathrm{D}}+\overline{\mathrm{F}})\cdot(\overline{\mathrm{E}}+\overline{\mathrm{F}}).
\end{aligned}
$$

Figure 10 shows the implication graph generated from the attack clauses.

A	B	C
D	E	F

Figure 9: Place a queen in every row of the board

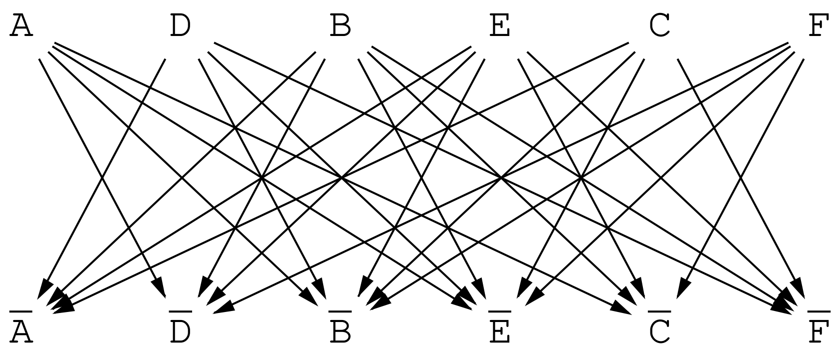


Figure 10: The implication graph from the 2-queens problem

From the implication graph we can see that variables B and E each have five outgoing implications, and variables A, C, D, and F each have four. Each of the six variables appears once in the ternary (placement) clauses. We want to order the variables so that the variables that place the most constraints on other variables appear first. Since variables B and E have more outgoing edges, this means that they must be assigned values before variables A, C, D, and F. The variable order B, E, A, C, D, F is acceptable. The affect of variable order on the ease of formula satisfiability is further discussed in Section 3.3.

B	E	A	C	D	F
0	0	1	0	0	1

Figure 11: A variable order for the 2-queens problem

Having determined a variable order, Figure 11 illustrates an attempt to search for a legal binding by stepping through the 2SAT bindings in descending order. The first legal binding for the implication graph, 100000, cannot be extended to satisfy the placement clauses because it allows for the placement of only one queen. The second legal binding, 010000, is similarly unsatisfactory. However, the third legal binding, 001001, satisfies the ternary clauses, and successfully concludes our search.

2.2.3 Terminating the Search

We terminate the search for a 2SAT binding that satisfies the entire formula in one of three ways:

1. We find a satisfying binding.
2. We prove that no binding exists.
3. We exceed the amount of computational effort we are willing to spend.

Though we can solve a 2SAT problem in linear time, there may be an exponential number of solutions. In the absence of significant theoretical advances, there will always be instances of *NP*-complete problems that take more time to complete than we want to wait; we would rather generate tests for all but one of the faults of a circuit in a small number of seconds than wait four hours and still not know if we will be given a successful test in the near future.

Pragmatics require that any implementation of our method stop searching for an answer after a certain number of 2SAT solutions have been unsuccessfully extended to a SAT solution. In the current implementation the number of unsuccessful 2SAT solutions we will tolerate is equal to the length of the variable array mentioned in Section 2.2.2. This backtrack limit was determined through experimentation and is not derived from the theoretical behavior of the search. In Section 3.3 we will discuss modifications to the satisfier so that instead of giving up when no solution is found after a given number of tries, we reorder the variables using a different metric and try again.

3 Minimizing the Search Tree

The algorithm we have just described is complete: If no test pattern for a fault exists, we will eventually prove it; if a test pattern exists, we will eventually find it. However, we can speed up the satisfier tremendously by figuring out how to quickly determine that some portions of the search tree contain no solutions. Like structural search ATPG systems, we can take advantage of topological information to avoid searching unprofitable sections of the search tree: we believe that any heuristic that can be stated in the structural search domain can be translated into a modification of the formula to be satisfied.

In this section we will describe how we translate several structural search heuristics into modifications to the basic Nemesis system described in Section 2. The effect that these modifications have on the efficiency of the base level system is described in detail in a previous publication [9, 10].

Each of the heuristics we will discuss is implemented in our system by adding to or subtracting from the formula to be satisfied. By adding or subtracting clauses we can avoid portions of the search tree. When we subtract variables we are making the search tree shorter, and when we add certain restrictive clauses we ignore branches of the search tree. In either case, we must ensure that the change preserves satisfiability.

3.1 Adding Clauses to the Formula

We can take the basic formula and add clauses that explicitly state information that the satisfier can eventually derive, but perhaps only after a great deal of search. The simplest example of such redundant information is the value of the faulted line with the unfaulted circuit. For example, the formula for the fault shown in Figure 5 contains the unary clause (D') (in English, the faulted value of the line is 1). The satisfier can derive that the variable D must take on the value 0 (for the XOR of the faulted and unfaulted circuits to be equal to 1), but we 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.

3.1.1 Non-local Implications

As noted by the designers of the SOCRATES system, it is possible to explicitly derive non-local implications by examining the reconvergent fanout in a circuit [17]. In Figure 12, we see that if line B has the value 1, line F has the value 1; conversely, if line F has the value 0, line B has the value 0. SOCRATES discovers this

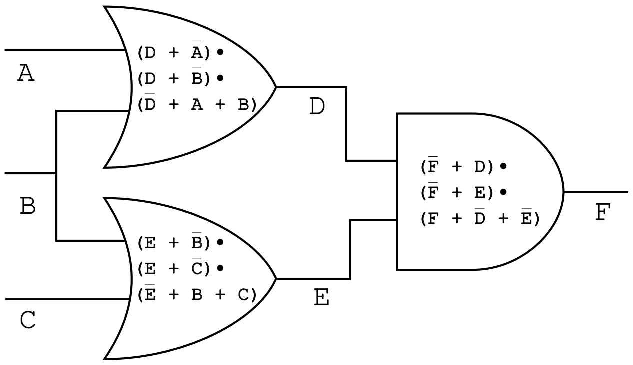


Figure 12: Non-local implications: Add $(\overline{\mathrm{B}} + \mathrm{F})$

implication by performing a structural analysis of the circuit; we find it by analyzing the formula representing the circuit.

Given the formula for an unfaulted circuit, we can list all 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. For example, the complete formula for the circuit in Figure 12 is

$$
\begin{gathered}
(\overline{\mathrm{F}} + \mathrm{D}) \cdot (\overline{\mathrm{F}} + \mathrm{E}) \cdot (\mathrm{F} + \overline{\mathrm{D}} + \overline{\mathrm{E}}) \cdot \\
(\mathrm{D} + \overline{\mathrm{A}}) \cdot (\mathrm{D} + \overline{\mathrm{B}}) \cdot (\overline{\mathrm{D}} + \mathrm{A} + \mathrm{B}) \cdot \\
(\mathrm{D} + \overline{\mathrm{A}}) \cdot (\mathrm{D} + \overline{\mathrm{B}}) \cdot (\overline{\mathrm{D}} + \mathrm{A} + \mathrm{B})
\end{gathered}
$$

Binding B to 1 causes the binary clauses $(\mathrm{D} + \overline{\mathrm{B}})$ and $(\mathrm{E} + \overline{\mathrm{B}})$ to be promoted to the unary clauses (D) and (E). When D and E are bound to 1, the ternary clause $(\mathrm{F} + \overline{\mathrm{D}} + \overline{\mathrm{E}})$ is promoted to a unary clause, which causes F to be bound to 1. The fact that a ternary clause was used to derive the direct implication that B bound to 1 implies F bound to 1 means that it is a non-local implication. By adding the explicit clause $(\overline{\mathrm{B}} + \mathrm{F})$ we insure that any time F is bound 0, B will also be bound to 0 without having to do any case-splitting.

We could add all the non-local implications for a circuit to every formula that we try to satisfy, but we only add the non-local implications if the satisfier fails on the original formula. The process of finding the implications can be time consuming, and we do not want to spend the time when the formula is easy to solve without the added information.

The great majority of patterns can be generated without non-local implications, but the few that could not be generated easily without non-local implications could not be generated even when the satisfier was allowed to run 1000 times as long as it normally does. Non-local implications are vital when it comes to processing difficult faults.

3.1.2 Active Clauses

When the D-algorithm was introduced, Roth concentrated on trying to get a discrepancy to a circuit output [16]. We can modify our formula so that the explicit need for a sensitized path can be used to speed up the satisfier. But there is a difference between the sensitized path of the D-algorithm and a sensitized path that we need for our formula: The D-algorithm searches for a solution by explicitly enumerating all possible combinations of sensitized paths, but we are only speeding up our search by taking advantage of the existence of at least one sensitized path for any detectable fault.

If a fault is detectable, there must be at least one sensitized path from the fault site to a circuit output. There may be more than one path, but we only need to find one: we will call this particular sensitized path the active path. Each line that is a member of the active path is an active line. Every active line must have a discrepancy, but since there may be other sensitized paths, not all lines with discrepancies are active lines.

To find an active path, we add clauses that describe how we would go about finding such a path manually: First, we know that the fault site is on the active path (if one exists). As for the other lines, if a line is on an acceptable active path and it is an input to a single output gate, the output must also be on the active path; if it is an input to a multiple output gate, one of the outputs must be on the active path. To put it formally, for each line that lies between the fault and a circuit output we allocate a variable (called the active variable for the line), and for each gate that lies between the fault and a circuit output we add several clauses (called the active clauses for that gate). We will use the notation that if a line has the name (variable) X, its active variable is $\mathrm{Act_X}$. For each single output gate with input X and output Y we add the clause $(\overline{\mathrm{Act_X}} + \mathrm{Act_Y})$ (in English, if X is active, Y is active). For each multiple output gate with input X and output Y and Z we add the clause $(\overline{\mathrm{Act_X}} + \mathrm{Act_Y} + \mathrm{Act_Z})$ (in English, if X is active, either Y is active or Z is active). Figures 13 and 14 show examples of these clauses.

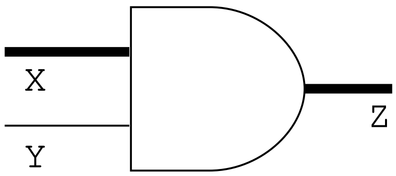


Figure 13: If X is active, Y must be active: $(\overline{\mathrm{Act_X}} + \mathrm{Act_Y})$

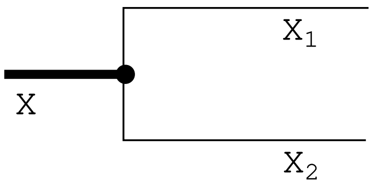


Figure 14: If X is active, either $\mathrm{X_1}$ or $\mathrm{X_2}$ must be active: $(\overline{\mathrm{Act_X}} + \mathrm{Act_{X_1}} + \mathrm{Act_{X_2}})$

If we only added the clauses we have described so far, we would find any path from the fault site to a circuit output and call it the active path (whether or not it was possible to sensitize it). We must also add clauses that ensure that the path is made up entirely of lines with discrepancies. For each potentially active line X, we add the formula $(\overline{\mathrm{Act_X}} + \mathrm{X} + \mathrm{X}') \cdot (\overline{\mathrm{Act_X}} + \overline{\mathrm{X}} + \overline{\mathrm{X}}')$ (in English, if X is active, the unfaulted value of X differs from the faulted value of X). For example, for the circuit in Figure 5 we allocate the variables $\mathrm{Act_D}$ and $\mathrm{Act_X}$, and add the formula

$$(\overline{\mathrm{Act_D}} + \mathrm{Act_X}) \cdot (\overline{\mathrm{Act_D}} + \mathrm{D} + \mathrm{D}') \cdot (\overline{\mathrm{Act_D}} + \overline{\mathrm{D}} + \overline{\mathrm{D}'}) \cdot (\overline{\mathrm{Act_X}} + \mathrm{X} + \mathrm{X}') \cdot (\overline{\mathrm{Act_X}} + \overline{\mathrm{X}} + \overline{\mathrm{X}'})$$

to the basic formula we mentioned in Section 2.

Adding the active implication clauses greatly increases the efficiency of our system. Without the implication clauses we abort on many of the faults.

3.1.3 Requiring Non-Controlling Values

If a gate is on the active path, we know that it must propagate the discrepancy. This means that the gate inputs not on the active path must take on certain non-controlling values that will allow the fault to be propagated. For example, if an AND gate is on the critical path, none of its non-active inputs can take on the value 0: if they did, the AND gate would always have the output 0, and no discrepancy could be propagated. On the other hand, a non-active input to an AND gate on the active path could have a discrepancy. In this case, if the non-active discrepancy is the same as the active discrepancy, the fault is propagated (0/1 AND 0/1 is 0/1); if the discrepancy is the opposite of the active discrepancy, the fault is not propagated (0/1 AND 1/0 is 0/0). Figure 15 shows two legal critical assignments for a 4-input AND gate (the active path is shown by a bold line), and Figure 16 shows illegal assignments for the same gate.

We can come up with similar rules for all the basic gates: 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 non-controlling value. For XOR and XNOR gates on the active path, we must require that their non-active inputs have no discrepancies (0/1 XOR 1/0 is 1 and 0/1 XOR 0/1 is 0).

We can add clauses requiring non-controlling values for every gate between the fault site and a circuit output. For example, the OR gate in Figure 4 is on the active path, and its input E cannot carry a discrepancy. We could add the clause $(\overline{\mathrm{Act_D}} + \overline{\mathrm{E}})$ (in English, if D is active, E must be 0) to the formula to be satisfied.

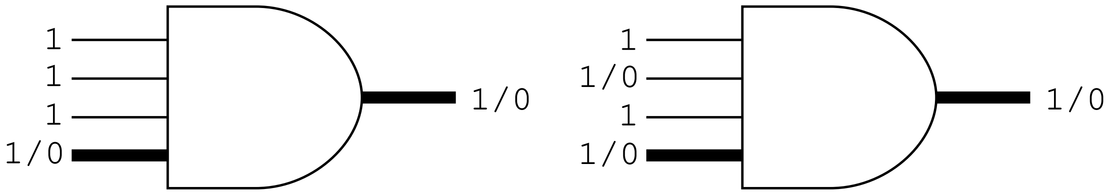


Figure 15: Legal critical assignments

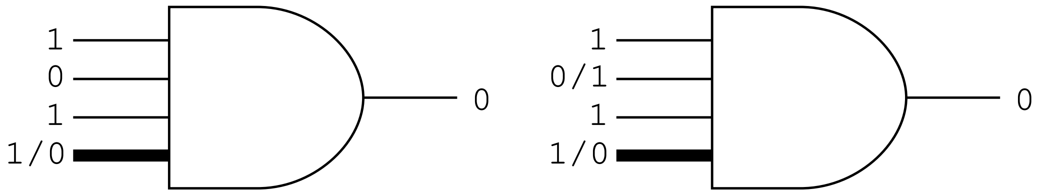


Figure 16: An illegal critical assignment

Explicitly requiring non-controlling values for gates propagating a discrepancy is of great value for the topological ATPG systems; in our case, the added clauses are not as valuable. The added clauses not only add redundant information, but the information they add is usually derived by the satisfier in a few simple steps. Adding the non-controlling clauses is inexpensive, and they can never retard the search for a solution, so we add the non-controlling clauses to our base level system.

3.1.4 Determining Unique Sensitization Points

We can build a preprocessor that identifies all of the unique sensitization points for each possible fault site by generating the active clauses for every gate in the circuit and determining the non-local implications of the active clauses. For example, looking at Figure 12 again, just as we generated the non-local implication $(\overline{\mathrm{B}} + \mathrm{F})$ from the formula for the circuit, we can also generate $(\overline{\mathrm{Act_B}} + \mathrm{Act_F})$. That is, we can derive that if B is active, F must be active.

Many authors of structural search ATPG systems place great importance on preprocessing the circuit structure to derive the unique sensitization points (points of total reconvergence) in the circuit [6, 17], but such a preprocessing step is not necessary for us. In the process of finding an active path, our satisfier will always find all the unique sensitization points without explicitly searching for them. We have never found a case where explicitly deriving the unique sensitization points improved the performance of our system.

3.1.5 Removing Clauses from the Formula

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 (even if removing a variable will remove some satisfying bindings from the solution set for the original formula). We don't need to find all satisfying bindings—we only need to find one.

We can avoid searching fanout-free portions of the circuit by removing variables and clauses corresponding to fanout-free portions of the circuit. To explain our method, we must first explain the *determines* relation.

We say that variable V determines variable W if an assignment of either 0 or 1 to V will 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.

As an example, in the Boolean difference formula presented for the circuit in Figure 5, E determines C but C does not determine E. This is true because when E is bound to either 0 or 1, C will appear only negated or only unnegated in the remaining clauses, but C can not be bound to any value that will leave E appearing only negated or only unnegated. 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.

A more interesting example appears in Figure 17 (where the triangle with input E and output E_1 and E_2 represents a fanout point). 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.

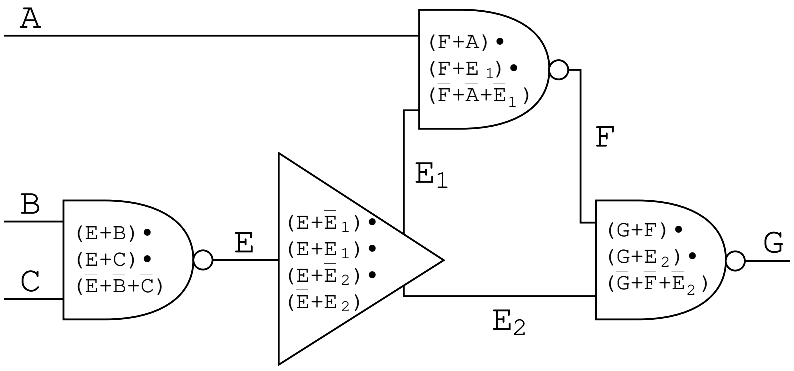


Figure 17: All clauses containing A, B, or C can be removed from the formula

3.2 Modifying 2SAT Variable Order

We want to iterate through the 2SAT solutions in an order that maximizes our chances of quickly discovering a solution that can be extended to a satisfying assignment for the entire CNF formula. In Section 2.2 we explained how we use a metric to determine the order of variable assignment. In fact, we do not use one metric, we use three. Like others who produce ATPG systems [13], we have noted that independent search strategies are often effective on different classes of faults. To use the terminology of Min and Rogers, search strategies that have largely disjoint solution sets (with a given search or backtrack limit) are called *orthogonal* search strategies. By limiting the search with a given strategy and switching to a new strategy when no perceivable progress is made in a given period, we can increase our coverage.

As we explain the three strategies, we will use the following example: Given the SAT formula

$$(\overline{\mathrm{A}} + \overline{\mathrm{B}}) \cdot (\overline{\mathrm{B}} + \overline{\mathrm{C}}) \cdot (\mathrm{A} + \overline{\mathrm{C}}) \cdot (\mathrm{A} + \overline{\mathrm{B}} + \overline{\mathrm{C}}),$$

Figure 18 shows the implication graph for the 2SAT portion of the formula. We will describe how the search for a satisfying assignment for this formula would differ under the three strategies. The three orderings are:

1. We order the variables from high to low by the number of other variables they directly force to 0 when bound, and we then step through the 2SAT solutions in descending order. That is, if we are not forced to assign a given variable to 0, we will bind to 1. Each of A and B force two other variables to 0 when bound, but C only forces one variable to 0, so A and B must appear before C. Given the order A, B,

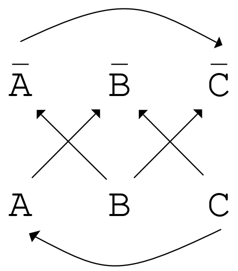


Figure 18: Implication graph for $(\overline{A}+\overline{B})\cdot(\overline{B}+\overline{C})\cdot(A+\overline{C})$

C, Figure 19 shows the search tree for our example. First A will be bound to 1, which will force B to be bound to 0. After we bind C to 1, the final binding is A = 1, B = 0, C = 1.

By using this strategy, we are attempting to assert the strongest constraints at every opportunity—whether the variable is bound to 1 or to 0. The more constraints we trigger at the beginning of a search, the fewer guesses we will have to make because so much of our search will be directed.

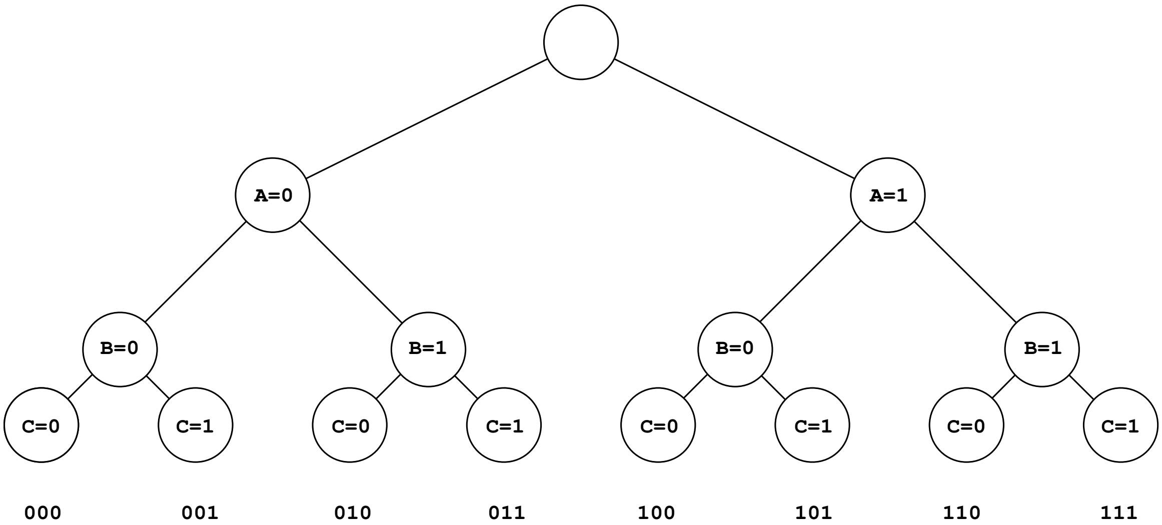


Figure 19: The search tree for the first two strategies

2. We use the same variable order as in Strategy 1, but we step through the solutions in ascending order. That is, if we are not forced to assign a given variable to 1, we will bind it to 0. The search tree is the same as for Strategy 1, except that instead of searching the tree from right to left, we search it from left to right. We didn't find a solution in the high-ordered section of the tree, so we look in the low ordered section. For our example, first A will be bound to 0, which will force C to be bound to 0. Upon binding B to 0 we have a solution consistent with our ternary clause: A = 0, B = 0, C = 0.

3. Like Strategy 1, we order the variables by the number of other variables that they force to 0, but unlike Strategy 1, we are interested only in the number of other variables that are forced to 0 when the variable is bound to 1. An additional difference with Strategy 1 is that this ordering is a lexicographic ordering: variables that force an equal number of other variables through 2SAT implications are ordered by their occurrence in the ternary clauses. We step through the 2SAT solutions in descending order. Figure 20 shows the search tree for our example. First we bind B to 1, which will force A and C to be bound to 0, leaving us a solution consistent with the ternary clause: A = 0, B = 1, C = 0. By using this strategy, we are also attempting to assert the strongest possible constraints at every opportunity, but this time

we will trigger the most constraints only if the variables are bound to 1. Since we are stepping through the bindings in descending order, the constraints triggered by binding a variable to 1 are more likely to come into play.

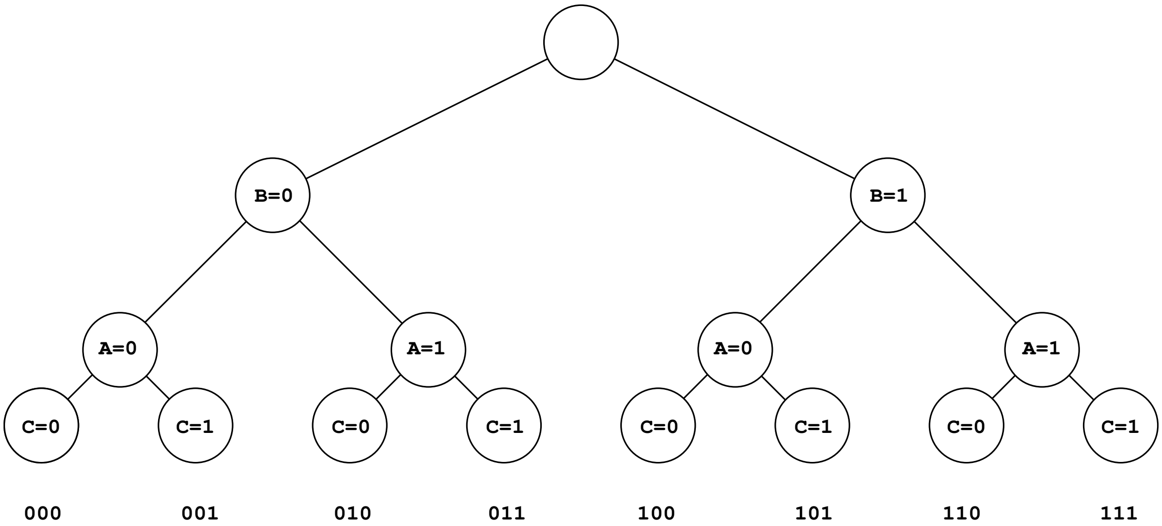


Figure 20: The search tree for the third strategy

The strategies we have just described are only three of the many possible search strategies we could have used. In practice, we have found the three strategies work well in concert. Strategy 2 will often find a solution when Strategy 1 will not. Since they explore the same solution space, but in opposite orders, it is easy to see that they are orthogonal searching strategies. Strategy 3 builds a markedly different tree from the first two only in cases where the assumptions used to build the first tree were invalid. That is, Strategy 1 may place a variable high in the ordering because it causes many constraints when it is bound to 0, but if the variable is only ever bound to 1, those constraints do not direct the search. By switching to an ordering that will strongly direct the search in the expected case, we can come up with a different solution set.

4 Results

Before we can present the measurements taken from Nemesis we must provide further information about how the features described in this paper fit into the system as a whole as well as what kind of input we are using to evaluate Nemesis's performance. Nemesis is written in C and runs on a Sun Sparcstation 1+. We used the ten sample circuits collected by Franc Brglez and Hideo Fujiwara and distributed at the 1985 ISCAS Conference as input to Nemesis [3]. We used the Tegas Description Language (TDL) version of the ISCAS circuits.

Before test pattern generation begins, Nemesis translates the TDL into an internal form and produces a collapsed fault list. After wirelist translation and fault collapsing, two phases of test pattern generation follow: random and algorithmic.

The first phase of test pattern generation is the random phase: We use the logic word operations of the computer to simulate 32 pseudo-random patterns against one target fault. The simulator is modeled after the parallel-pattern, single fault propagation (PPSFP) simulator reported by Waicukaski et. al. [18]. In this way we generate patterns for the easily tested faults (generally 80% to 99% of the total faults). When one complete PPSFP pass produces fewer than a predetermined number of patterns (currently two), the second phase, algorithmic pattern generation, begins.

The algorithmic test pattern generation uses the Boolean satisfiability method described in this article in conjunction with all of the heuristics described except for the heuristic that avoids search of fanout-free subtrees. During the algorithmic pattern generation phase, each pattern generated is simulated (using a simple single pattern, single fault propagation simulator) so that any faults detected by the new pattern may be removed from the fault list. If the system backtracks too many times during the 2SAT iteration, the fault is abandoned.

	Time in Seconds						
Circuit	Parse	PPSFP	Extract	Satisfy	SPSFP	Compact	Total
C0432	0.5	0.4	0.4	7.0	0.0	0.3	8.5
C0499	0.6	0.4	0.6	1.9	0.0	0.4	3.9
C0880	0.9	0.5	1.4	34.1	0.2	0.4	37.5
C1355	1.4	1.4	2.2	15.0	0.2	1.8	22.0
C1908	2.0	2.8	6.1	56.4	0.7	1.8	69.8
C2670	2.9	2.8	22.1	339.6	2.1	1.7	371.2
C3540	3.7	6.9	42.4	204.1	1.6	6.0	264.7
C5315	5.5	5.1	9.4	50.6	0.6	3.6	74.8
C6288	6.2	35.0	30.6	39.8	0.0	35.9	147.6
C7552	8.1	8.2	53.7	668.0	8.2	6.5	752.6

Table 1: Nemesis timing

Table 1 shows the time spent for each individual circuit during each of six phases: translation of the wirelist into internal form, generating and simulating semi-random test patterns, extracting formulas, satisfying formulas, simulating the patterns found by formula satisfaction, and compaction of the resultant vectors. For all circuits but the C6288, Nemesis spends most of its processing time satisfying extracted formulas.

	Faults in Circuit		Faults covered by		Proved	
Circuit	Uncollapsed	Collapsed	Random	Algorithmic	Redundant	Aborted
C0432	864	420	410	6	4	0
C0499	998	652	641	3	8	0
C0880	1660	765	727	38	0	0
C1355	2710	1444	1389	47	8	0
C1908	3816	1740	1643	88	9	0
C2670	5340	2516	2090	309	117	0
C3540	7080	3150	2931	90	129	0
C5315	10630	4909	4828	22	59	0
C6288	12576	7619	7585	0	34	0
C7552	15104	7194	6566	497	131	0

Table 2: Nemesis number of faults

Table 2 shows the number of faults that require test patterns, the number of faults after fault collapsing, the number of faults covered by the semi-random test pattern generation and simulation, the number of faults covered by extracting and satisfying a formula, and the number of faults proved redundant by extracting and falsifying a formula.

Table 3 shows the number of patterns produced by each phase and the percentage of faults covered, proved redundant, or aborted by the complete system. Nemesis was the second system to successfully produce tests for or prove redundant every fault in the benchmark circuits.

	Number of Patterns		
Circuit	Random	Algorithmic	Compacted
C0432	70	7	68
C0499	53	19	60
C0880	94	21	72
C1355	90	18	92
C1908	64	110	138
C2670	95	81	149
C3540	190	79	202
C5315	191	27	161
C6288	47	0	45
C7552	297	146	245

Table 3: Nemesis patterns

5 Conclusions

The Boolean satisfiability method for generating test patterns for single stuck-at faults in combinational circuits—extracting a formula for the test set of a fault and then satisfying that formula—is general, flexible, and effective. By separating the solution from the exact form of the problem, we can solve a larger class of problems than can more restrictive systems. Not only can we translate traditional structural heuristics into our domain, but we can incorporate heuristics that would be difficult to implement in a structural search system.

The Nemesis system using the Boolean Satisfiability method achieves total test coverage of the ISCAS-85 benchmark circuits: it was the second system (after SOCRATES) to correctly process all the faults. The structural search methods have had the benefit of more than a decade of program development and craftmanship; the strength of our model leads us to believe that we will gain significant performance improvements as the system matures.

References

[1] S. B. Akers. On a theory of boolean functions. *Journal of the Society for Industrial and Applied Mathematics*, 7, 1959.

[2] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8:121–123, 1979.

[3] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinatorial benchmark circuits and a target translator in fortran. In *International Symposium on Circuits and Systems*. IEEE, June 1985.

[4] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Anual ACM Symposium of Theory of Computing*. ACM, 1971.

[5] M. Davis and H. Putman. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[6] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, C-31:1137–1144, 1983.

[7] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Transactions on Computers*, C-31:215–222, 1981.

[8] T. Larrabee. Efficient generation of test patterns using boolean difference. In *Proceedings of the International Test Conference*. IEEE, August 1989. Also available as part of Digital Equipment Corporation Western Research Lab Research Report WRL-90/3.

[9] T. Larrabee. A framework for evaluating test pattern generation strategies. In *Proceedings of the International Conference on Computer Design*. IEEE, October 1989. Also available as part of Digital Equipment Corporation Western Research Lab Research Report WRL-90/3.

[10] T. Larrabee. *Efficient Generation of Test Patterns Using Boolean Satisfiability*. PhD thesis, Stanford University, 1990. Also available as Stanford Technical Report STAN-CS-90-1302 and as Digital Equipment Corporation Western Research Lab Research Report WRL-90/2.

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

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

[13] H. B Min and W. A. Rogers. Search strategy switching: An alternative to increased backtracking. In *Proceedings of the International Test Conference*. IEEE, 1989.

[14] D. K. Pradhan. *Fault-Tolerant Computing Theory and Techniques*. Prentice-Hall Publishing, 1986.

[15] P. W. Jr. Purdom and C. A. Brown. Evaluating search methods analytically. In *Proceedings of the National Conference on Artificial Intelligence*, pages 124–127, 1982.

[16] J. P. Roth. Diagnosis of automata failures: A calculus and a method. *IBM Journal of Research and Development*, 10:278–291, 1966.

[17] M. H. Schulz, E. Trischler, and T. M. Sarfert. Socrates: A highly efficient automatic test pattern generation system. *IEEE Transactions on CAD*, pages 126–137, January 1988.

[18] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, and T. McCarthy. Fault simulation for structured VLSI. *VLSI Design*, VI:20–32, 1985.