FEBRUARY 1990

WRL Research Report 90/2

Efficient Generation of Test Patterns Using Boolean Satisfiability

Tracy Larrabee

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

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
DARPA Internet:	`WRL-Techreports@decwrl.dec.com`
CSnet:	`WRL-Techreports@decwrl.dec.com`
UUCP:	`decwrl!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Efficient Generation of Test Patterns Using Boolean Satisfiability

Tracy Larrabee

February, 1990

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

WRL Research Reports

‘‘Titan System Manual.’’
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’
David W. Wall.
WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’
David W. Wall.
WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’
Paul J. Asente.
WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’
Brian K. Reid.
WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’
Jeremy Dion.
WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’
Silvio Turrini.
WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’
David W. Wall.
WRL Research Report 89/17, September 1989.

‘‘Noise Issues in the ECL Circuit Family.’’
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 90.

‘‘Efficient Generation of Test Patterns Using Boolean Satisfiablilty.’’
Tracy Larrabee.
WRL Research Report 90/2, February 90.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’
Joel McCormack.
WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?’’
John Ousterhout.
WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’
Joel Bartlett.
WRL Technical Note TN-12, October 1989.

This report is a slightly revised version of a dissertation submitted in 1990 to the Department of Computer Science of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

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.

This dissertation describes a new method for generating test patterns: the Boolean satisfiability method. The 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 most programs now in use, which directly search the circuit data structure instead of constructing a formula from it. 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.

Acknowledgements

I have many people and institutions to thank for financial, technical, and emotional support that I have received during my work on this research and dissertation.

Digital Equipment Corporation supported me with a student fellowship during several years of graduate study. I am grateful to Sam Fuller for choosing me as the recipient of the fellowship and for being supportive as my research progressed.

My most important technical acknowledgement must go to Greg Nelson of Digital Equipment Corporation's Systems Research Center (DEC SRC). Greg's (mostly) patient explanations of everything from formal methods for attacking satisfiability problems to program methodology were crucial to my research. I have treated Greg as my own personal tutor; I hope to have the opportunity to help future students as Greg has helped me.

I was very lucky in my choice of advisor and dissertation reading committee. Each of these three men has contributed a tremendous amount to my graduate education.

John Hennessy agreed to be my advisor before I veered off into the realm of test pattern generation; I am very grateful to him for sticking by me and for many animated and instructive conversations.

To Forest Baskett I give the credit (or blame) for my choice of dissertation topic. I was working for Forest at Digital Equipment Corporation's Western Research Lab (DEC WRL) when he first suggested that I take a look at test pattern generation. If Forest had not recognized that the ideas generated by my first summer's work were worth pursuing, this dissertation would never have been produced.

Mark Horowitz joined my committee just as I was beginning to get results. Mark provided technical precision, and an impetus to useful technical conversations with

many members of Stanford's Center for Integrated Systems (Helen Davis, David Dill, Don Stark, and Daniel Weise immediately come to mind).

Many other members of the Stanford community gave me important ideas. From Stanford's Center for Reliable Computing, Ed McCluskey, John Udell, and Steve Millman provided not only feedback, but access to the Brglez-Fujiwara benchmark circuits. From the AI and theory communities, Tomas Feder, Ramsey Haddad, Donald Knuth, Evan Reid, and Ramin Zabih provided important ideas concerning satisfiability.

Greg was not the only person at Digital's Palo Alto research labs to lend me technical aid. Anna Karlin, Mark Manasse, Jim Saxe, and Mary-Claire van Leunen from DEC SRC, and David Boggs, Jeremy Dion, Mary Jo Doherty, Jud Leonard, and Jeff Mogul from DEC WRL provided interest and assistance. Aside from the support provided by first-class colleagues, DEC WRL also gave me access to several powerful machines. Special thanks are due to David Boggs, Brian Reid, and Mary Jo Doherty for allowing me to use their Titans.

Aside from those friends who have already mentioned for providing technical support, I thank Carolyn Bell, Linda Gass, Charles Haynes, Bruce Nelson, Glenn Reid, Karin Scholz, Andy Shore, and Larry Zwick for their encouragement and interest.

To my parents, Michael and Margaret Larrabee, I am indebted not only for the warmth and support of recent years, but also for the basic skills I needed to succeed as a graduate student at Stanford.

Finally, and most emphatically, I want to thank my husband, Daniel Mills, who has done much tutoring in the mechanics of page layout, much proof-reading, and much hand-holding. I intend to spend many years proving to him that I am worth the effort.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reliably functioning computers are not a luxury. A company perceived by its customers to sell unreliable digital systems will not be in business very long. Every customer expects a reliable product, but some customers need computer systems that are dependable in the extreme; these customers are willing to pay a premium for systems that can be counted on in situations of crucial importance—even of life and death. To produce reliable computer systems, defect-free components must be available. In this dissertation we present a method of distinguishing defective components from defect-free components.

More formally, we present a method of generating *test patterns* for *single stuck-at faults* in *combinational* circuits. The rest of this chapter will describe both the problem of test pattern generation and current approaches used to solve the problem. Our approach, generating a test pattern by extracting a formula for all possible tests for a fault and then satisfying that formula, will be described fully in Chapter 2 and Chapter 3.

1.1 The Problem

We will only be generating tests for *combinational* circuits, which have no feedback loops or memory elements. Figure 1.1 shows a simple combinational circuit. In a following section we will explain how techniques for testing combinational circuits

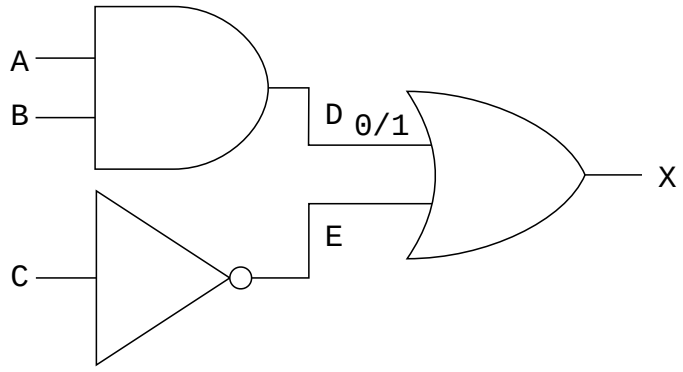


Figure 1.1: Combinational circuit with D stuck at 1

can be used to test sequential circuits.

A *test pattern* for a potentially defective circuit is a set of inputs for the circuit that will cause the circuit outputs to be different if the circuit is defective than if it is defect-free. To derive the input set, we must have some model for possible defects (faults) in the circuit. We will use the model most popular with existing testing systems: the *single stuck-at model*. In this model, a defective circuit is assumed to behave as if it were defect-free, with the exception of one wire that is tied to either a logic 0 or a logic 1 (instead of correctly varying as a function of the circuit inputs). Logically equivalent inputs may fail independently. For example, in Figure 1.2 A_1 can be stuck-at 1 while A_2 takes on the value 0.

To generate a test pattern for a circuit with a wire stuck at 1, we must ensure

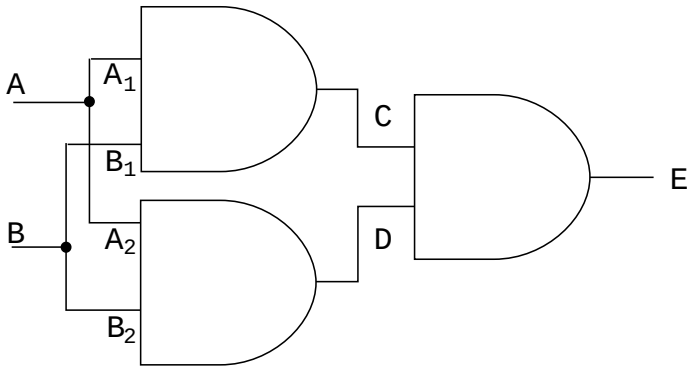


Figure 1.2: Line A_1 can fail independently from line A_2

that the wire in question would take on the logic value 0 in a correctly functioning circuit. If this is not the case, the circuit outputs will be the same whether or not the circuit is malfunctioning because the faulty circuit and the good circuit would carry the same values. In Figure 1.1, line D is labeled with a 0/1 to denote that line D is the site of a fault such that D will carry the value 0 if the circuit is functioning correctly (is *unfaulted*) and will carry the value 1 if the circuit is defective (is *faulted*). When a line has a different value in the faulted and unfaulted circuits, it is said to have a *discrepancy*. Figure 1.3 shows a test pattern that detects D stuck-at 1 and Figure 1.4 shows a test pattern that does not detect D stuck-at 1. We say that the test pattern of Figure 1.3 *covers* D stuck-at 1 and the test pattern of Figure 1.4 fails to cover D stuck-at 1.

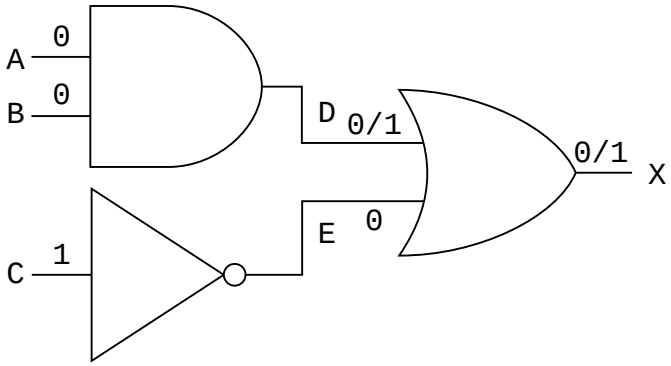


Figure 1.3: Test pattern covering D stuck-at 1

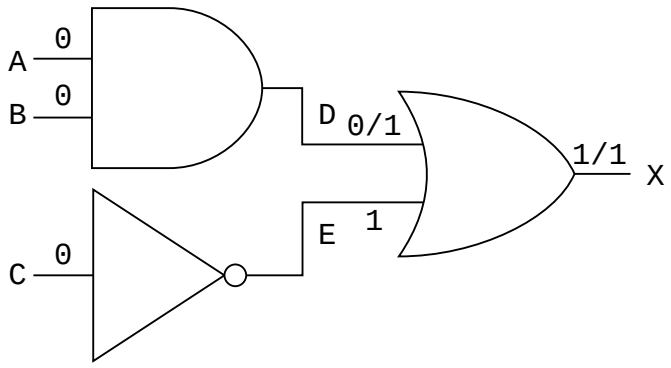


Figure 1.4: Test pattern not covering D stuck-at 1

As we will discuss in the next section, test pattern generation is known to be an *NP*-complete problem[FT82]. This does not mean that we can afford to give up on the problem. We must try to find methods of test pattern generation that have excellent expected case behavior on actual circuits. In Chapters 2 and 3 we will present such a method and the evidence that it is workable.

1.1.1 The Complexity of the Problem

We can quickly demonstrate that test pattern generation is an *NP*-hard problem by showing how a known *NP*-complete problem can be reduced to it. As our known *NP*-complete problem we will choose 3SAT [Coo71]: the problem of satisfying a boolean formula written in three-element conjunctive normal form (3CNF).

First, we take a 3CNF formula (also known as a product of sums formula where each sum has at most three literals) and naively build the circuit corresponding to it. We can do this by creating one OR gate for each clause and feed the outputs of all the OR gates into one AND gate. In Figure 1.5 we show a circuit corresponding to the 3CNF formula $(A + \overline{B} + C) \cdot (A + B + \overline{C})$. Next, we generate a test pattern for the output of the circuit stuck at 0. If it were possible to generate the test pattern in polynomial time, it would be possible to satisfy a 3CNF formula in polynomial time. Stated formally, 3SAT is polynomial-time reducible to test pattern generation.

It would be possible to generate a test pattern in linear time if it were not for

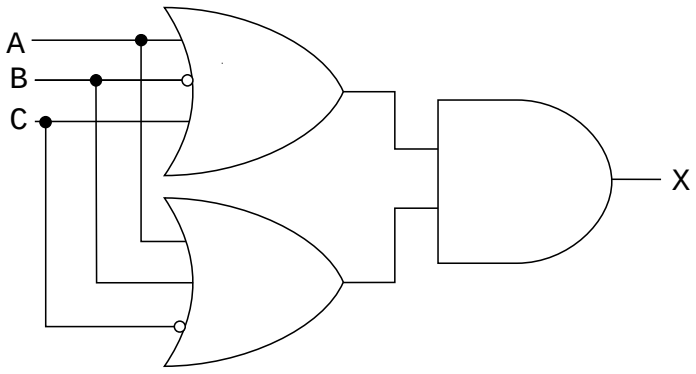
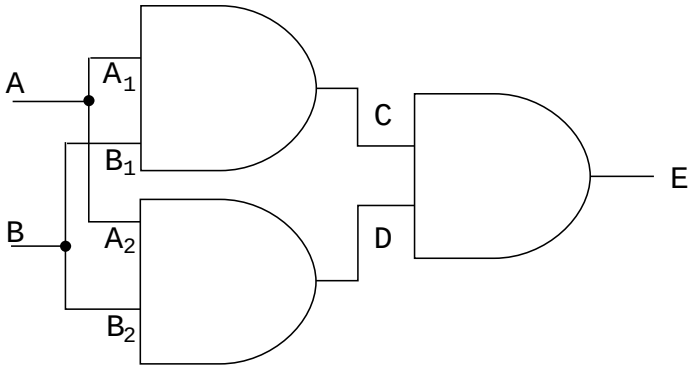


Figure 1.5: Circuit corresponding to the formula $(A + \overline{B} + C) \cdot (A + B + \overline{C})$

reconvergent fanout [IS75]. In a combinational circuit, reconvergent fanout occurs whenever there is more than one path of logic elements between any two lines in the circuit. For example, in Figure 1.5, there is more than one path between line A and line X. The presence of reconvergent fanout introduces potentially unsatisfiable dependencies into the problem of test pattern generation. We will discuss the nature of the potentially unsatisfiable dependencies when we discuss *backtracking* in Section 1.2.1.

It may not be possible to generate a test pattern for every single stuck-at fault in a circuit. Figure 1.6 shows an example of a circuit with an untestable fault (B_2 stuck at 1) and the truth table for the circuit both without the fault (the good machine) and with the fault (the bad machine). The truth table shows that the output of the circuit will be the same whether or not the circuit is faulted, so there is no test pattern that can detect the presence of B_2 stuck at 1: the fault is *untestable*. Because the circuit uses three AND gates to implement the function $A \cdot B$, it is not surprising that the presence of a single fault on an internal wire is undetectable. Circuits often have redundant circuitry that make the detection of certain single stuck-at faults impossible. We say that these untestable faults are *redundant faults*.



A	B	Good	Bad
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Figure 1.6: Circuit and truth table for B_2 stuck-at 1

The fact that a fault is redundant does not mean that it is uninteresting. Like test pattern generation, the question of whether or not a fault is redundant is an *NP*-complete problem, but identifying the redundant faults in a circuit can be crucial. We want to be able to ensure that we have generated a test for every possible fault

in a given circuit, so we must be able to identify redundant faults as part of our test pattern generation.

1.1.2 The Scope of the Problem

The ability to test combinational circuits can be used to test sequential circuits. In the 1960's and 70's, proponents of *design for testability*, who believe that a good logic design not only works but can be economically tested, began publishing schemes for segregating a sequential circuit into portions with and without state [CMPR64, WA73, EW77]. Today such *scan path techniques* are used by many industrial and academic organizations.

There are many variations on scan path techniques, but each involves using design rules and additional logic to allow the finished circuit to operate in two modes: normal mode, in which the additional logic is transparent, and test mode, in which the circuit state can be arbitrarily initialized or read out. In Figure 1.7 we show a circuit designed using a simple scan path technique.

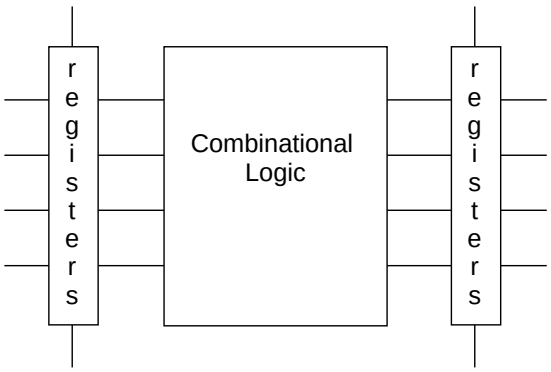


Figure 1.7: Scan Path: Isolating circuit state

Each memory element is a member of a shift register called the *scan chain*. When in normal mode, the registers act in parallel and their membership in the scan chain is undetectable. When in test mode, the registers act serially, and an arbitrary test pattern may be shifted into the memory elements. For our testing purposes, the circuit can be viewed as a set of initializable registers feeding into a collection of

combinational logic with results going back into another set of registers. To test the circuit, we use test mode to shift in a test pattern, use normal mode for one clock cycle, and then use test mode to shift out the result of the combinational operation.

1.2 Existing Solutions

There are several algorithmic *automatic test pattern generation systems* (ATPG systems) for combinational logic. Some ATPG systems work by generating algebraic equations and performing symbol manipulation on the equations, but the most successful ATPG systems perform their search for a solution in a topological, or structural, manner. In this section we will describe structural search and algebraic methods and give examples of each.

1.2.1 Structural Search Methods

Structural search methods work using a data structure representing the circuit to be tested. To generate a test pattern, they assign values corresponding to the 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 search is performed using three basic operations:

Fault sensitization is the process of generating a discrepancy at the fault site. As described in the first section, if we are looking for a test for a particular line stuck at 0, then this line must take on the value 1 in the correctly functioning circuit. Figure 1.8 shows the line values needed to sensitize the fault D stuck-at 1 in our sample circuit.

Fault propagation is the process of moving a discrepancy closer to a circuit output. Using our notation, we want to get a 0/1 or a 1/0 to a circuit output. Figure 1.9 shows the line values needed to propagate D stuck at 1 to an output of our sample circuit. Note that if line E were given the value 1 instead of the value 0, the discrepancy would not be visible at a circuit output.

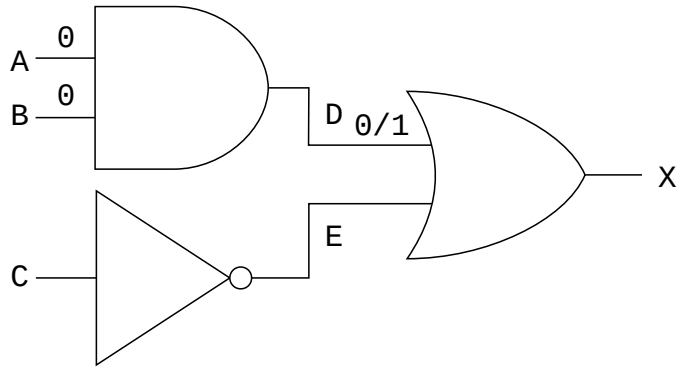


Figure 1.8: Fault sensitization

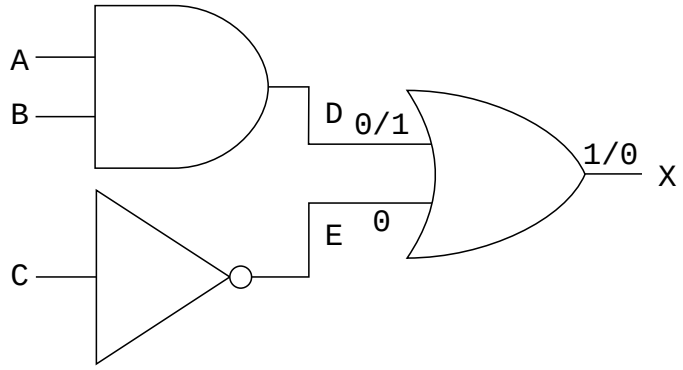


Figure 1.9: Fault propagation

Line justification is the process of assigning consistent values to all of the lines in the circuit that were not assigned values through fault sensitization or fault propagation. To perform line justification on our sample circuit after the assignment of the values given in Figures 1.8 and 1.9, the only thing we need do is give line C the value 1. The final assignment is shown in Figure 1.3.

If the circuit under test has no reconvergent fanout, the three steps described above yield a test pattern without searching. Because of reconvergent fanout, it is possible to assign a value to a line that will eventually be found to be inconsistent with previously assigned values. For example, using Figure 1.6 again, suppose that our ATPG system binds a logic value of 0 to the line E. At this point there are three different pairs of assignments that could be made to lines C and D. If the system

chooses to assign the value 0 to line C and 1 to line D, it will be forced to assign 1 to A_2 and B_2 and a 0 to either A_1 or B_1. Either assignment is inconsistent since A_1 must equal A_2 and B_1 must equal B_2. In this case the ATPG system must *backtrack* to the point when values were given to C and D and choose a different assignment. If the ATPG system then chooses to bind C to 1 and D to 0, it will eventually have to backtrack again. Only when C and D are both bound to 0 can a solution be found. Note that the system may have to backtrack during fault sensitization, fault propagation, or line justification.

When generating a test pattern for a complicated circuit, an ATPG system is likely to spend all of its time in an unprofitable portion of the search tree unless it uses some heuristics to raise the probability of finding a solution. Structural search systems differ mostly in the heuristics that they use. We will mention four groundbreaking ATPG systems and describe their seminal contributions. In the following chapters we will describe how our system exploits the advances made by each of the four systems.

The D-algorithm [Rot66] (introduced in 1966 by Roth) was the first published algorithmic topological search method; it is still popular today. The D-algorithm is the basis for all existing structural search ATPG systems; it uses fault sensitization, fault propagation, and line justification as we have just described. The D-algorithm uses heuristics that try to guarantee that all line assignments are made with the goal of pushing a discrepancy toward a circuit output.

PODEM [Goe81] (introduced in 1981 by Goel) first described the problem of test generation as a classic search through the binary tree describing all possible circuit input values. PODEM uses the framework and formalisms of the D-algorithm, but always binds line values at the circuit inputs (all other line values are determined by implication). PODEM uses a *backtrace* operation that is similar to line justification to determine which circuit inputs should be bound (and to what value). PODEM is faster than the D-algorithm for most circuits.

FAN [FS83] (introduced in 1983 by Fujiwara) improves on PODEM through three new strategies:

1. FAN has an improved backtrace operation that follows each possible path back from the discrepancy to the circuit input: the improved backtrace procedure allows FAN to avoid assignments that will eventually be proved inconsistent.

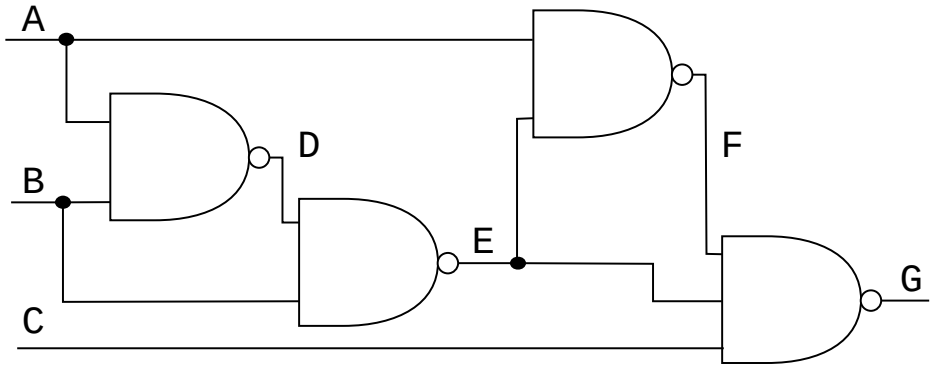


Figure 1.10: D and F are points of unique sensitization for a fault at B

2. FAN performs an analysis of the circuit topology to identify points of *unique sensitization*. When all paths from a discrepancy to a circuit output must pass through a given line, that line is a point of unique sensitization. In Figure 1.10, if there is a discrepancy at B, then E and G are points of unique sensitization. Points of unique sensitization are valuable because they must carry a discrepancy if the fault is detectable. This can imply values for many different circuit lines. For example, for a discrepancy at B in Figure 1.10 to be detectable, we know that A and C must be bound to 1.

3. FAN reduces its search space by taking advantage of the fact that it requires no search to determine if the output of a circuit that is free of reconvergent fanout can be set to a given value. Fujiwara divides all lines in the circuit into *free lines* and *bound lines*. Bound lines are lines that lie on any path between a fanout point and a circuit output. All lines that are not bound lines are free lines, and free lines that are adjacent to a bound line are *head lines*. In Figure 1.11, the bound lines, shown with bold lines,

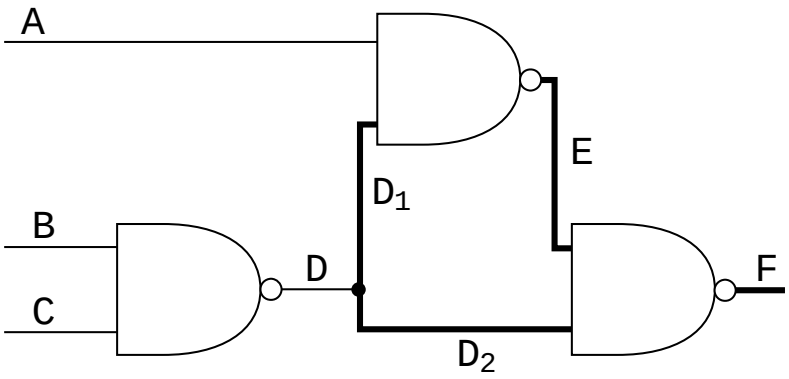


Figure 1.11: Circuit with head lines A and D

are D_1, D_2, E, and F; all other lines are free lines. Of the free lines, lines A and D are head lines.

Subcircuits composed only of free lines are fanout free and may be ignored until consistent values have been given to every bound line, so FAN stops its backtrace operation at head lines (instead of continuing to a circuit input as PODEM does).

FAN is faster than PODEM for most circuits [Fuj85a].

SOCRATES [STS88] (introduced in 1988 by Schulz et al) is an improvement on FAN. SOCRATES improves on FAN's unique sensitization procedure, but more importantly, SOCRATES is the first ATPG system that uses information from *non-local implications*. We will use Figure 1.12 to illustrate how non-local implications can be used by an ATPG system (this process is called *learning* by the designers of SOCRATES).

In Figure 1.12, if B is bound to 1, then by direct implication, D and E must be bound to 1, and therefore F must be bound to 1. However, if we bind F to 0, it is impossible to conclude anything about the value of D and E because they could be bound to any of three separate pairs of values (01, 10, or 00). The fact that we don't know the values of D and E does not mean that we don't know the value of B: if B bound to 1 implies F bound to 1, F bound to 0 implies B bound to 0. This means that if F is bound to 0 during any stage

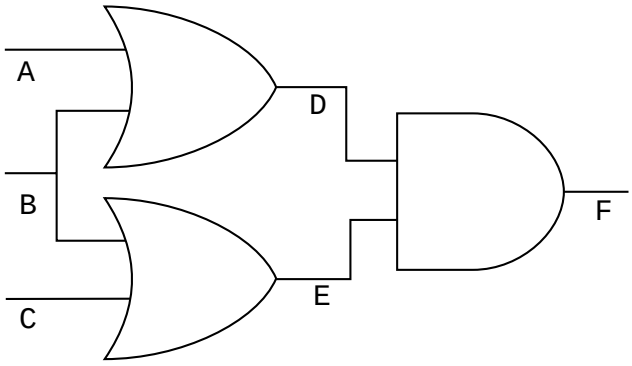


Figure 1.12: B = 1 implies F = 1, so F = 0 implies B = 0

of test pattern generation we can immediately conclude that B is 0. This kind of added information is very valuable when the ATPG system is attempting to determine if it has made an inconsistent assignment, and it can be crucial when it comes to identifying redundant faults.

Socrates collects non-local implications on the circuit as a whole (static learning) and non-local implications given a partial assignment of circuit values (dynamic learning). It uses heuristics to guess whether a given non-local implication is of value during test pattern generation.

SOCRATES is the most efficient test pattern generation system available today [SE88].

1.2.2 Algebraic Manipulation

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 that equation. To see how it is possible to generate a formula that describes a test set, we will look again at Figure 1.1. The formula for the output of this circuit is

$$X(A, B, C) = A \cdot B + \overline{C}.$$

If we use D as an intermediate variable, we can write the formula as

$$X(A, B, C) = F(D(A, B, C), A, B, C)$$

where $D(A, B, C) = A \cdot B$ and $F(D, A, B, C) = D + \overline{C}$.

Given this formula, the formula for the faulted circuit with D stuck-at 1 is

$$X'(A, B, C) = F(1, A, B, C)$$

To generate a test for D stuck-at 1, we want to ensure that X′, the output of the faulted circuit, is different from X, the output of the unfaulted circuit; we want

$$X(A, B, C) \neq X'(A, B, C).$$

Substituting in for X and X′ and substituting XOR for the difference operator, we rewrite the formula as

$$F(D(A, B, C), A, B, C) \oplus F(1, A, B, C).$$

Since we know that F(D(A, B, C), A, B, C) can only be different from F(1, A, B, C) if D(A, B, C) is not 1, we can transform the formula into

$$\overline{D}(A, B, C) \cdot (F(0, A, B, C) \oplus F(1, A, B, C)).$$

Substituting for F and D, our formula for the set of tests for D stuck-at 1 is

$$(\overline{C} \oplus 1) \cdot \overline{(A \cdot B)}$$

which simplifies to

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

So the set of tests for D stuck-at 1 are those where C is bound to 1 and at least one of A and B is bound to 0[1].

[1]One thing to notice here that will be important in the next chapter is that the validity of the formula does not change if we introduce intermediate variables. If we introduce an intermediate variable, we do not change the permissable values for the original variables. We do change the solution set, but only because each satisfying binding will also contain bindings for the introduced variables that are consistent with the original variables. In the example above, we could have assigned the new intermediate variable X to the output of X(A,B,C,0) and the new intermediate variable X′ to the output of X(A,B,C,1). Then the final formula for the test sets would be written

$$(X \oplus X') \cdot (X = X(A, B, C, 0)) \cdot (X' = X(A, B, C, 1)) \cdot \overline{D}(A, B, C).$$

If we want we can introduce intermediate variables for every line in the circuit.

We can generalize this procedure: Given any circuit with an output function of $F(X_1, ..., X_n)$ and a fault site with intermediate variable Y and function $Y(X_1, ..., X_n)$, we can rewrite the function as $F(X_1, ..., X_n, Y)$. Now the test set for Y stuck at 0 is

$$F(X_1, ..., X_n, 0) \oplus F(X_1, ..., X_n, 1) \cdot Y(X_1, ..., X_n)$$

and the test set for Y stuck at 1 is

$$F(X_1, ..., X_n, 0) \oplus F(X_1, ..., X_n, 1) \cdot \overline{Y}(X_1, ..., X_n).$$

Both the formula for Y stuck at 1 and Y stuck at 0 have in common the formula

$$F(X_1, ..., X_n, 0) \oplus F(X_1, ..., X_n, 1).$$

This formula is known as the *Boolean difference* of F with respect to Y and is written dF/dY. To restate it, the Boolean difference of any function F with respect to its variable X_i is written as

$$dF/dX_i \equiv F(X_1, ..., X_{i-1}, 0, X_{i+1}, ..., X_n) \oplus F(X_1, ..., X_{i-1}, 1, X_{i+1}, ..., X_n).$$

The 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 $\overline{X_i} \cdot dF/dX_i$.

Once the formula using the Boolean difference is obtained, it is simplified using the basic laws of Boolean algebra. It can also be simplified using identities specific to the Boolean difference. For example, the following is an incomplete list of proven Boolean difference identities [Ake59]:

1. $dF/dX_i = d\overline{F}/dX_i$
2. $d(dF/dX_j)/dX_i = d(dF/dX_i)/dX_j$
3. $d(F \cdot G)/dX_i = F \cdot dG/dX_i \oplus G \cdot dF/dX_i \oplus dF/dX_i \cdot dG/dX_i$
4. $d(F + G)/dX_i = \overline{F} \cdot dG/dX_i \oplus \overline{G} \cdot dF/dX_i \oplus dF/dX_i \cdot dG/dX_i$
5. $d(F \oplus G)/dX_i = dF/dX_i \oplus dG/dX_i$

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. As examples of the prevailing attitude in the testing community toward the Boolean difference method, here are several quotations from major texts that touch on combinational logic test:

> The algebraic methods, relying on symbol manipulation, are not as readily implementable within the framework of a design automation system since the design automation data bases are generally topological. A. Miczo, *Digital Logic Testing and Simulation* [Mic86]

> The [algebraic algorithms are] not very practical since the heuristics required to tolerate the *NP*-completeness of the test generation problem are not available. Yet this group of algorithms serves the very important purpose of enlightening the fundamental nature of testing problems. J. A. Abraham and V. K. Agarwal, *Fault-Tolerant Computing Theory and Techniques* [Pra86]

> The Boolean difference is not used directly for practical test pattern generation. [...] The usefulness of the Boolean difference is as a tool for theoretical studies of testing and checking methods. E.J. McCluskey, *Logic Design Principles* [McC86]

> For all but very simple networks the calculation of the Boolean difference is not practical. E.J. McCluskey, *Logic Design Principles* [McC86]

We agree that the symbolic manipulation implicit in the Boolean difference method is impractical. However, our method, which is neither a purely topological method nor an algebraic one, involves using the topological structure of the circuit to extract a formula equivalent to the formula produced by the Boolean difference method; it is not only practical but performs better than most systems now in use. Instead of performing symbol manipulation on the extracted formula, we run a *Boolean Satisfiability* algorithm on it. We will describe the Boolean satisfiability method in full in the next chapter.

Chapter 2

The Boolean Satisfiability Method

In this chapter we describe our solution to the problem of generating a test pattern for a single stuck-at fault in a combinational circuit. Our approach is to divide the problem into two pieces: First, we extract a formula that defines the set of test patterns that detect the fault. Second, we use a Boolean satisfiability algorithm to satisfy the formula. The rest of this chapter is devoted to describing these two steps in detail.

2.1 Extracting the Formula

2.1.1 The Representation of the Circuit

We use a directed acyclic graph (DAG) as a 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 2.1 shows the DAG representation of the circuit in Figure 1.12. By walking the graph starting at any circuit output, we can reach every node that can affect the value of the output.

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

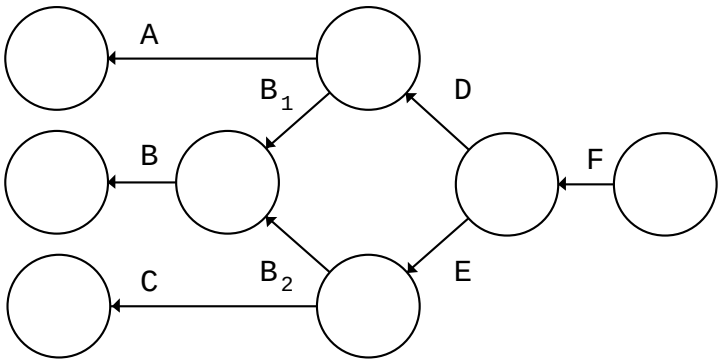


Figure 2.1: The DAG representation of the circuit in Figure 1.11

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.

2.1.2 Translating Formulas into 3CNF

We will use 3-element conjunctive normal form, or 3CNF (also known as product of sums form where each sum has at most three literals). Formulas written in 3CNF are easy to manipulate programmatically. To get the 3CNF formula for an AND gate, we start with the formula

$$Z = X \cdot Y.$$

Using the identity $P = Q \equiv (P \Rightarrow Q) \cdot (Q \Rightarrow P)$, we transform the formula into

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

Next, the identity $P \Rightarrow Q \equiv \overline{P} + Q$ transforms $Z \Rightarrow (X \cdot Y)$ into $(\overline{Z} + X) \cdot (\overline{Z} + Y)$ and $(X \cdot Y) \Rightarrow Z$ into $\overline{X} + \overline{Y} + Z$. So the final formula for an AND gate is

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

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 3CNF formulas, one sum is called a *clause*. A clause with only one element is a *unary clause*, a clause with two elements is a *binary clause*, and a clause with three elements is a *ternary clause*. A formula with no ternary clauses is said to be in 2CNF (2-element conjunctive normal form).

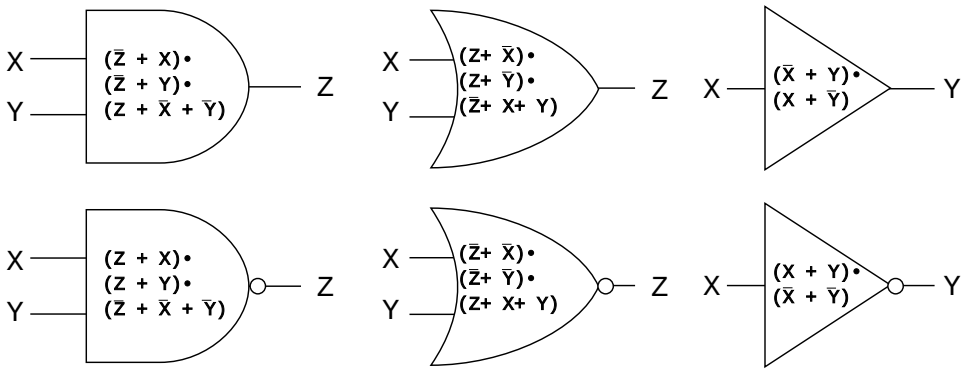


Figure 2.2: The formulas for the basic gates

We display the 3CNF formulas for the basic gates in Figure 2.2, but the gates need not be basic to be included in this scheme: With the introduction of new variables, we can produce the 3CNF form of any formula in time and space linear in the size of the original formula. For example, the 3CNF formula for an AND gate with inputs X, Y, and W and output Z, is

$$(\overline{\mathrm{Z}} + \mathrm{X}) \cdot (\overline{\mathrm{Z}} + \mathrm{Y}) \cdot (\overline{\mathrm{Z}} + \mathrm{W}) \cdot (\overline{\mathrm{X}} + \overline{\mathrm{Y}} + \mathrm{V}_1) \cdot (\overline{\mathrm{V}}_1 + \overline{\mathrm{W}} + \mathrm{Z}),$$

where V_1 is an introduced variable. The formula for an XOR gate with inputs X and Y and output Z is

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

Later in the chapter we will explain that it is easier for us to satisfy CNF formulas that have a preponderance of clauses in 2CNF, so we introduce intermediate variables and change $\mathrm{Z} = \mathrm{X} \oplus \mathrm{Y}$ into $\mathrm{Z} = (\mathrm{V}_1 + \mathrm{V}_2) \cdot (\mathrm{V}_1 = \overline{\mathrm{X}} \cdot \mathrm{Y}) \cdot (\mathrm{V}_2 = \mathrm{X} \cdot \overline{\mathrm{Y}})$. The final formula for the XOR gate looks like this:

$$(\mathrm{Z} + \overline{\mathrm{V}}_1) \cdot (\mathrm{Z} + \overline{\mathrm{V}}_2) \cdot (\overline{\mathrm{Z}} + \mathrm{V}_1 + \mathrm{V}_2) \cdot$$
$$(\overline{\mathrm{V}}_1 + \mathrm{X}) \cdot (\overline{\mathrm{V}}_1 + \overline{\mathrm{Y}}) \cdot (\mathrm{V}_1 + \overline{\mathrm{X}} + \mathrm{Y}) \cdot (\overline{\mathrm{V}}_2 + \overline{\mathrm{X}}) \cdot (\overline{\mathrm{V}}_1 + \mathrm{Y}) \cdot (\mathrm{V}_1 + \mathrm{X} + \overline{\mathrm{Y}}),$$

where the first line corresponds to $Z = (V_1 + V_2)$ and the second line corresponds to $(V_1 = \overline{X} \cdot Y) \cdot (V_2 = X \cdot \overline{Y})$.

2.1.3 Formulas for Unfaulted and Faulted Circuits

Because each gate and fanout point is tagged with a formula, 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. The circuit of Figure 1.1 is repeated here as Figure 2.3; the gates have been labeled by their characteristic formulas. The formula for the output is

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

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 2.4 shows the faulted version of the circuit in Figure 2.3. We add the formula $(\overline{D})$ to the node representing the correct behavior at the fault site, and we add the formula (D') to the node representing the faulted behavior at the fault site.

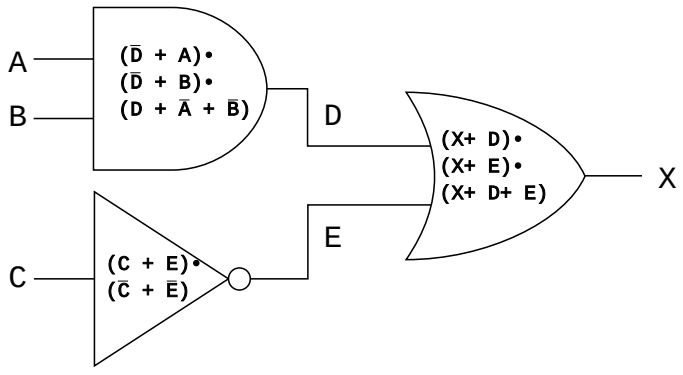


Figure 2.3: Combinational circuit with labeled gates

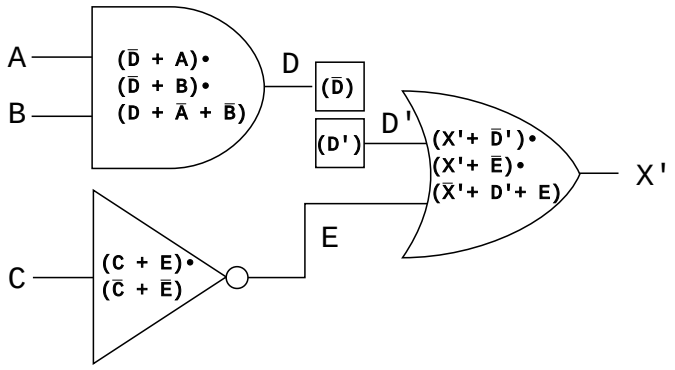


Figure 2.4: Circuit of Figure 2.3 with D stuck at 1

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 2.4 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. We will call the result of this final XOR, BD. The formula resulting from the XOR of the output of the unfaulted circuit of Figure 2.3 and the faulted circuit of Figure 2.4 is

$$\begin{aligned}
&(\mathrm{X}' + \overline{\mathrm{D}'}) \cdot (\mathrm{X}' + \overline{\mathrm{E}}) \cdot (\overline{\mathrm{X}'} + \mathrm{D}' + \mathrm{E}) \cdot (\mathrm{D}') \cdot (\mathrm{C} + \mathrm{E}) \cdot (\overline{\mathrm{C}} + \overline{\mathrm{E}}) \cdot \\
&(\mathrm{X} + \overline{\mathrm{D}}) \cdot (\mathrm{X} + \overline{\mathrm{E}}) \cdot (\overline{\mathrm{X}} + \mathrm{D} + \mathrm{E}) \cdot (\overline{\mathrm{D}} + \mathrm{A}) \cdot (\overline{\mathrm{D}} + \mathrm{B}) \cdot (\mathrm{D} + \overline{\mathrm{A}} + \overline{\mathrm{B}}) \cdot \\
&(\overline{\mathrm{X}} + \overline{\mathrm{V}_1}) \cdot (\mathrm{X}' + \overline{\mathrm{V}_1}) \cdot (\mathrm{V}_1 + \mathrm{X} + \overline{\mathrm{X}'}) \cdot (\mathrm{X} + \overline{\mathrm{V}_2}) \cdot (\overline{\mathrm{X}'} + \overline{\mathrm{V}_2}) \cdot (\mathrm{V}_2 + \overline{\mathrm{X}} + \mathrm{X}') \cdot \\
&(\overline{\mathrm{BD}} + \mathrm{V}_1 + \mathrm{V}_2) \cdot (\overline{\mathrm{V}_1} + \mathrm{BD}) \cdot (\overline{\mathrm{V}_2} + \mathrm{BD}),
\end{aligned}$$

where the first line is contributed by the faulted circuit, the second line is contributed by the unfaulted circuit, and the last two lines are contributed by the final XOR. Figure 2.5 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.

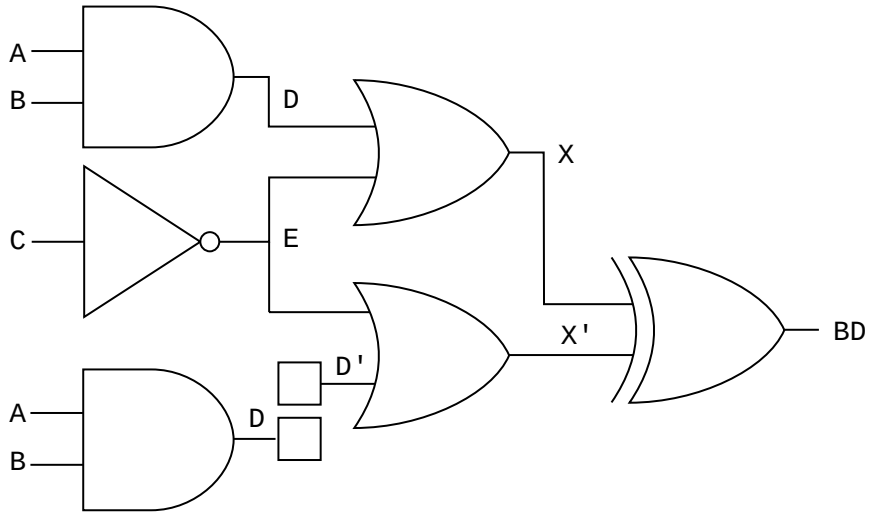


Figure 2.5: The XOR of the faulted and unfaulted circuits must be 1

2.1.4 Boolean Difference Method Formula Equivalence

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 our formula, 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.

Looking again at the formula produced by the Boolean difference method, if the

output of an unfaulted circuit is $F(X_1, ..., X_n, Y)$, and Y is the fault site, the formula for Y stuck at 1 is

$$F(X_1, ..., X_n, 0) \oplus F(X_1, ..., X_n, 1) \cdot \overline{Y}(X_1, ..., X_n),$$

as noted in the last section of Chapter 1. The formula extracted for the faulted circuit corresponds to $F(X_1, ..., X_n, 1)$; the formula extracted for the unfaulted circuit corresponds to $F(X_1, ..., X_n, 0) \cdot Y(X_1, ..., X_n)$. The conjunction of all of the formulas tagging nodes that lie between the fault site and any circuit input is $Y(X_1, ..., X_n)$; the conjunction of all of the formulas tagging nodes that were not used to produce $Y(X_1, ..., X_n)$ is $F(X_1, ..., X_n, 1)$.

To generate a test for the fault in question, we need to find a satisfying assignment for the formula; if it cannot be satisfied, the fault is undetectable.

2.2 Satisfying the Formula

As we saw in Chapter 1, 3SAT, the problem of satisfying a 3CNF formula, is an *NP*-complete problem [Coo71]. So 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. 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. Many researchers have recognized that the average behavior of a 3SAT algorithm can be improved dramatically if the set of formulas to be solved fit a restricted profile [DP60, PB82]. 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 2.2) . 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 [APT79]. 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 3SAT

We use an algorithm from the 1970's for satisfying a 2CNF formula [APT79]. 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 labeled 1 (true) to precede (or imply) a node labeled 0 (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 0 and thus narrowed to 2CNF or unary clauses.

As an example of how 2SAT works, consider the small circuit in Figure 2.6; imagine that we wish to iterate through all possible bindings to the variables A, A1, A2, B,

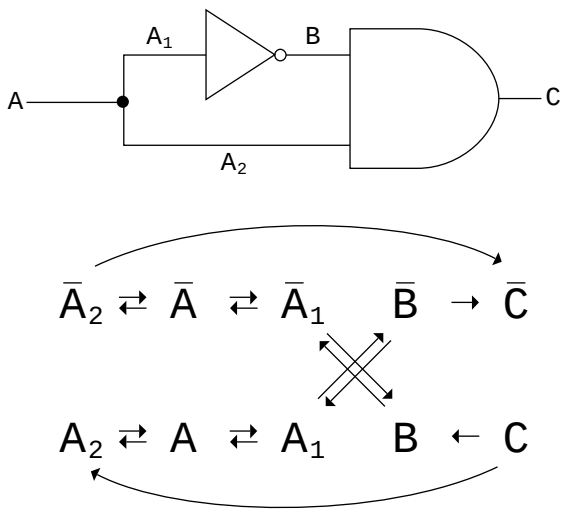


Figure 2.6: A simple circuit and its implication graph

and C. The formula for C is

$$(\overline{A} + A_1) \cdot (A + \overline{A_1}) \cdot (\overline{A} + A_2) \cdot (A + \overline{A_2}) \cdot$$
$$(\overline{A_1} + B) \cdot (\overline{A_1} + \overline{B}) \cdot (\overline{C} + A_2) \cdot (\overline{C} + B) \cdot$$
$$(\overline{A_2} + \overline{B} + C),$$

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 2.6. The graph has two strongly connected components: $\{\overline{A_2}, \overline{A}, \overline{A_1}, B\}$ and its compliment, $\{A_2, A, A_1, \overline{B}\}$: we will replace these strongly connected components with the unit nodes E_1 and $\overline{E_1}$, which results in the graph shown in Figure 2.7. The final graph clearly shows that C implies $\overline{C}$, and therefore C must be bound to 0. Given this restriction, 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

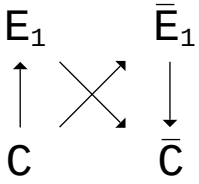


Figure 2.7: The reduced implication graph

labeled 1 has a directed path to a node labeled 0. Clearly there are many such assignments: 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 (we will discuss the metrics used to order the variables in Chapter 3). 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. On 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 until we discuss this further in Chapter 3, 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]. V[j] is bound for all $0 \leq$ j $<$ i. 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 2.8 shows a loop (with loop invariants) that achieves this goal.

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 2.8: 2SAT iteration loop

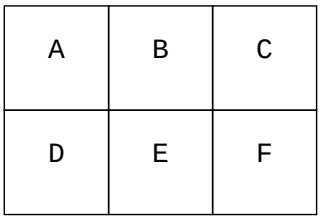


Figure 2.9: Place a queen in every row of the board

Figures 2.9 through 2.11 show an example of 2SAT iteration (another example will be presented in Section 3.3). In Figure 2.9 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 3CNF 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 $(\overline{A} + \overline{B})$ prevents queens from being simultaneously placed in squares A and B. The complete list of clauses is

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

Figure 2.10 shows the implication graph generated from the attack clauses.

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,

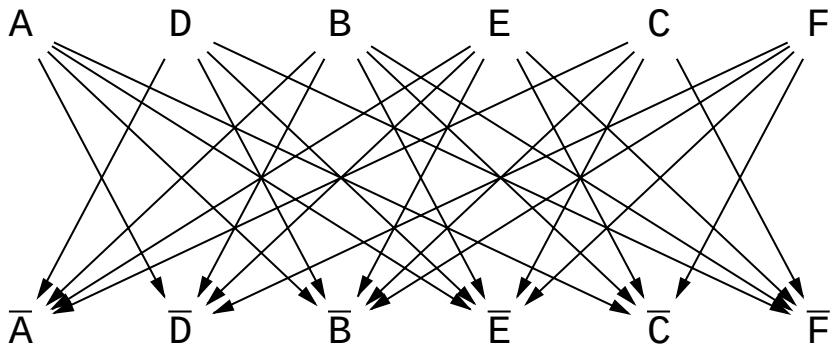


Figure 2.10: The implication graph from the 2-queens problem

C, D, F is acceptable.

Having determined a variable order, Figure 2.11 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.

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

Figure 2.11: A variable order for the 2-queens problem

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.

As we mentioned earlier, 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 3SAT solution. In the implementation that we describe in Appendix A, the number of unsuccessful 2SAT solutions we will tolerate is equal to the size 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 Chapter 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.

2.3 Results

In Appendix A we describe a system implemented using the Boolean satisfiability method and a set of benchmark circuits used by ATPG system designers. While the detailed breakdown of the system performance is presented in Appendix A, in Table 2.1 we present the summary of the system performance on the ten benchmark circuits. Table 2.1 contains the total time the system spent on each circuit, the percentage of total faults covered by the system, the percentage of faults proved redundant by the system, and the percentage of faults that the system aborted.

The performance reported is that of a system that includes heuristics not yet described. Specifically, this system includes active clauses, critical value clauses, non-local implication clauses, and repeated modification of 2SAT variable order. Each of these heuristics will be described in detail in the next chapter.

Circuit	Time (seconds)	Percent of Faults Covered	Redundant	Aborted
C0432	10.7	99.09	0.97	0.00
C0499	10.6	98.73	1.27	0.00
C0880	44.2	100.00	0.00	0.00
C1355	28.1	99.45	0.55	0.00
C1908	128.9	99.49	0.51	0.00
C2670	380.9	95.08	4.92	0.00
C3540	354.3	95.84	4.16	0.00
C5315	111.2	98.79	1.21	0.00
C6288	197.5	99.55	0.45	0.00
C7552	639.0	98.18	1.82	0.00

Table 2.1: Base level system summary

Our system is one of only two published that have produced tests for or proved redundant every fault in the benchmark circuits.

Chapter 3

Heuristics

The algorithm we described in the previous chapter 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 and therefore do not need to be searched. Like topological ATPG systems, we can take advantage of structural information to avoid searching unprofitable sections of the search tree: any heuristic that can be stated in the topological domain can be translated into a modification of the formula to be satisfied.

In this chapter we will describe how we translate several topological heuristics into modifications to our algorithm, and we will describe the effect that these modifications have on the efficiency of the base level system. When we conclude that a change improves the efficiency of our method, we will be drawing upon experiments run with a complete ATPG system using the Boolean satisfiability method. At the end of Chapter 2 we presented a system summary for a system that includes many of the heuristics we will be presenting. For each heuristic included in the base-level system, we will give a performance summary for the system without the chosen heuristic. For each heuristic not included in the base-level system, we will give a performance summary for a system that includes the heuristic. In Appendix A we will describe the complete system, describe a set of benchmarks used to evaluate our system (and other systems), and give precise performance measures for the system as a whole.

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 2.5 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. In this section we will describe several different kinds of redundant information as well as what effect the added redundancies have on our system.

3.1.1 Non-local Implications

As we discussed in Chapter 1, it is possible to explicitly derive non-local implications by examining the reconvergent fanout in a circuit. Figure 1.12 is repeated here (with tagged gates) as Figure 3.1. Once again, 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 implication by performing a structural analysis of the circuit [STS88]; 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

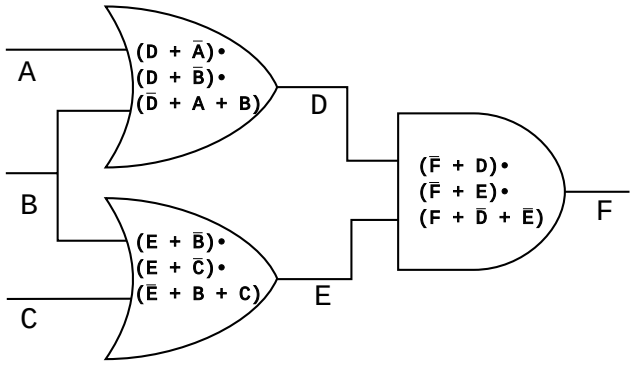


Figure 3.1: Non-local implications: Add $(\overline{B} + F)$

must come from reconvergent fanout. For example, the complete formula for the circuit in Figure 3.1 is

$$(\overline{F} + D) \cdot (\overline{F} + E) \cdot (F + \overline{D} + \overline{E}) \cdot$$
$$(D + \overline{A}) \cdot (D + \overline{B}) \cdot (\overline{D} + A + B) \cdot$$
$$(D + \overline{A}) \cdot (D + \overline{B}) \cdot (\overline{D} + A + B)$$

Binding B to 1 causes the binary clauses $(D + \overline{B})$ and $(E + \overline{B})$ to be promoted to the unary clauses (D) and (E). When D and E are bound to 1, the ternary clause $(F + \overline{D} + \overline{E})$ 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 $(\overline{B} + F)$ 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 would be easy to solve without the added information.

As we can see by comparing Table 3.1 with Table 2.1 on page 31, 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.

	Time	Percent of Faults		
Circuit	(seconds)	Covered	Redundant	Aborted
C0432	10.7	99.09	0.91	0.00
C0499	10.6	98.73	1.27	0.00
C0880	44.2	100.00	0.00	0.00
C1355	28.0	99.45	0.55	0.00
C1908	128.9	99.49	0.51	0.00
C2670	301.5	94.84	4.62	0.44
C3540	354.3	95.84	4.16	0.00
C5315	110.8	98.79	1.21	0.00
C6288	195.8	99.55	0.45	0.00
C7552	638.3	98.18	1.82	0.00

Table 3.1: System performance without non-local implications

3.1.2 Active Clauses

When the D-algorthm was introduced, Roth concentrated on trying to get a discrepancy to a circuit output [Rot66]. We might say that of the three steps for test pattern generation, Roth saw fault sensitization and propagation as more important than line justification. This approach made intuitive and practical sense: first discover if it is possible to create a path from the fault site to a circuit output such that every line on that path has a discrepancy (find a sensitized path to a circuit output) and then try to justify the values needed for the path.

We can modify our formula so that our approach also looks for a sensitized path. 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. Figure 3.2 shows active paths for two circuits with a fault at line D.

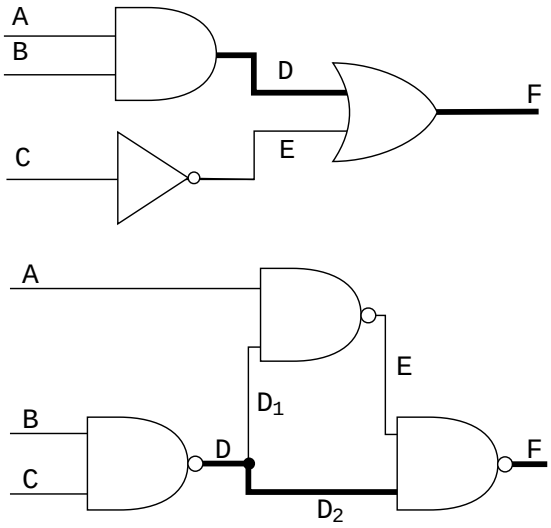


Figure 3.2: An active path for D faulted is shown with thick lines

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 3.3 and 3.4 show examples of these clauses.

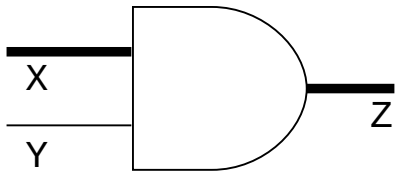


Figure 3.3: If X is active, Y must be active: $(\overline{\mathrm{Act_X}} + \mathrm{Act_Y})$

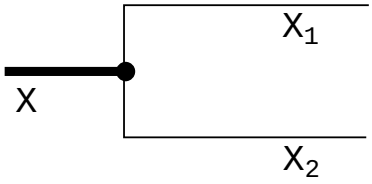


Figure 3.4: If X is active, either X_1 or X_2 must be active: $(\overline{\mathrm{Act_X}} + \mathrm{Act_{X_1}} + \mathrm{Act_{X_2}})$

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 2.5 we allocate the variables $\mathrm{Act_D}$ and $\mathrm{Act_X}$, and add the formula

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

to the basic formula we mentioned in Chapter 2.

As we can see by comparing Table 3.2 with Table 3.1, by adding the active implication clauses, we greatly increase the efficiency of our system. Without the implication clauses we abort on many of the faults.

Circuit	Time (seconds)	Percent of Faults Covered	Redundant	Aborted
C0432	9.6	99.03	0.73	0.24
C0499	109.6	98.64	0.00	1.36
C0880	63.9	99.60	0.00	0.40
C1355	137.8	99.45	0.00	0.55
C1908	344.1	98.62	0.52	0.86
C2670	1410.6	94.22	0.67	5.11
C3540	8105.2	93.21	1.72	5.07
C5315	615.4	98.34	0.71	0.95
C6288	14007.3	99.55	0.24	0.21
C7552	3384.4	97.14	0.14	2.72

Table 3.2: System performance without active clauses or non-local implications

3.1.3 Requiring Critical 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 critical 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 3.5 shows two legal critical assignments for a 4-input AND gate (the active path is shown by a bold line), and Figure 3.6 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 static critical value (AND and NAND gates require static critical values of 1, and OR and NOR gates require static critical values of 0). 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).

Given these requirements, we can now add clauses requiring critical values for every gate between the fault site and a circuit output. For example, the OR gate in Figure 2.4 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.

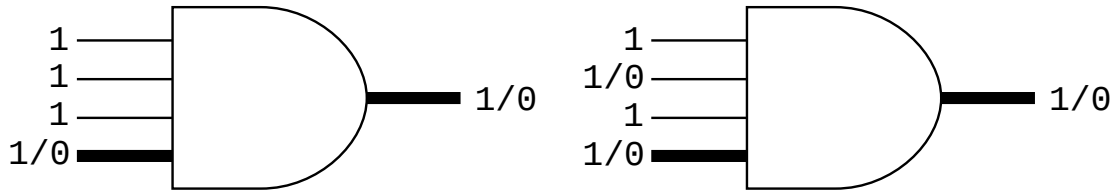


Figure 3.5: Legal critical assignments

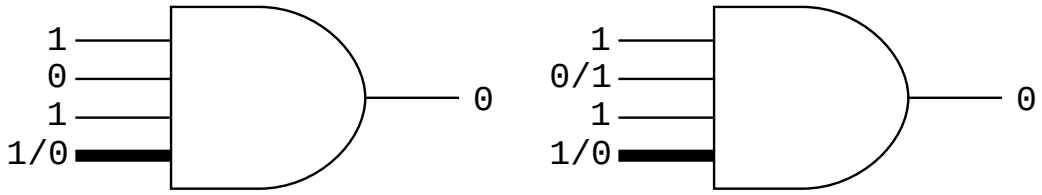


Figure 3.6: An illegal critical assignment

Explicitly requiring critical 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 critical clauses is inexpensive, and they can never retard the search for a solution, so we add the critical clauses to our base level system. As we can see by comparing Table 3.3 with Table 3.1, there are cases where critical value clauses are useful.

Circuit	Time (seconds)	Percent of Faults Covered	Redundant	Aborted
C0432	9.5	99.03	0.97	0.00
C0499	10.3	98.64	1.36	0.00
C0880	42.5	100.00	0.00	0.00
C1355	27.5	99.45	0.55	0.00
C1908	177.2	99.43	0.51	0.06
C2670	528.6	94.84	4.62	0.44
C3540	351.2	95.89	4.11	0.00
C5315	103.7	98.76	1.24	0.00
C6288	185.8	99.55	0.45	0.00
C7552	658.3	98.15	1.82	0.03

Table 3.3: System performance without critical clauses or non-local implications

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 3.1 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 topological ATPG systems place great importance on preprocessing the circuit structure to derive the unique sensitization points (points of total reconvergence) in the circuit [FS83, STS88], 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.

3.2 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 mimic structural heuristics that avoid searching some portions of the circuit by removing variables from the formula.

3.2.1 Avoiding Fanout-Free Subcircuits

As described in Chapter 1, the FAN algorithm stops its backtrace procedure at head lines so that it can avoid searching fanout-free portions of the circuit [FS83]. We can can restrict our search space in a similar manner 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 either an assignment of 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 2.5, 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.

A more interesting example appears in Figure 3.7 (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.

Unfortunately, our technique as stated will not remove as many variables as may

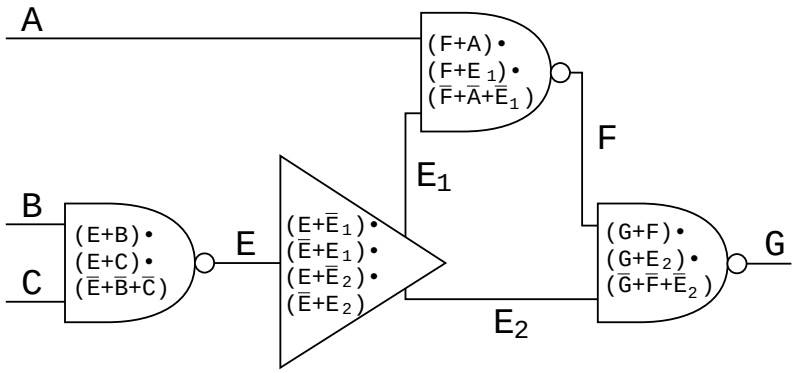


Figure 3.7: All clauses containing A, B, or C can be removed from the formula

be safely removed from the formula. In Chapter 2 we mentioned that we had a choice of using a 4-clause or a 9-clause characteristic formula for an XOR gate: in neither representation can we determine that the output of the XOR gate determines the input. For example, the 4-clause formula for an XOR gate with inputs X and Y and output Z is

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

Binding Z to 1 leaves the formula $(\overline{\mathrm{X}} + \overline{\mathrm{Y}}) \cdot (\mathrm{X} + \mathrm{Y})$, in which both X and Y appear negated and unnegated (the formula says X is not equal to Y). Similarly, binding Z to 0 leaves the formula $(\overline{\mathrm{X}} + \mathrm{Y}) \cdot (\mathrm{X} + \overline{\mathrm{Y}})$, which also leaves X and Y appearing negated and unnegated (the formula says X equals Y). Even though we know that given the output of an XOR gate, we can set the inputs without searching, we can't get that information using the determines relation. This seems to be a case of one variable determining two variables jointly instead of one variable at a time.

As we can see by comparing the times from Table 3.4 with the times from Table 3.1, even on circuits with no XOR gates (only the C0432 and C0499 circuits contain XOR gates), we have found that minimizing the search tree by removing determined variables does not help our system (neither does it hurt it). 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.

Circuit	Time (seconds)	Percent of Faults Covered	Redundant	Aborted
C0432	12.7	99.09	0.97	0.00
C0499	13.1	98.73	1.27	0.00
C0880	55.3	100.00	0.00	0.00
C1355	33.1	99.45	0.55	0.00
C1908	149.5	99.49	0.51	0.00
C2670	244.1	95.08	4.92	0.00
C3540	403.3	95.84	4.16	0.00
C5315	131.9	98.79	1.21	0.00
C6288	218.8	99.55	0.45	0.00
C7552	742.1	98.18	1.82	0.00

Table 3.4: System performance with FAN-reduced formulas

3.3 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 3CNF formula. In Chapter 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 [MR89], 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.

Each of the three heuristics for variable order that we use make intuitive sense because each attempts to keep the search as constrained as possible. If the search is not constrained, we often find that the first half of the variables are given an initial assignment, and then the second half of the variables fluctuate wildly, never counting enough to exit the first half of the search tree.

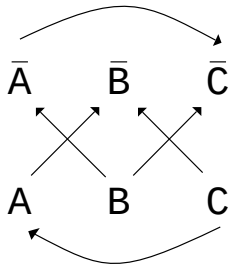


Figure 3.8: Implication graph for $(\overline{A} + \overline{B}) \cdot (\overline{B} + \overline{C}) \cdot (A + \overline{C})$

As we explain the three strategies, we will use the following example: Given the 3SAT formula

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

Figure 3.8 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. 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. Figure 3.9 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.

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

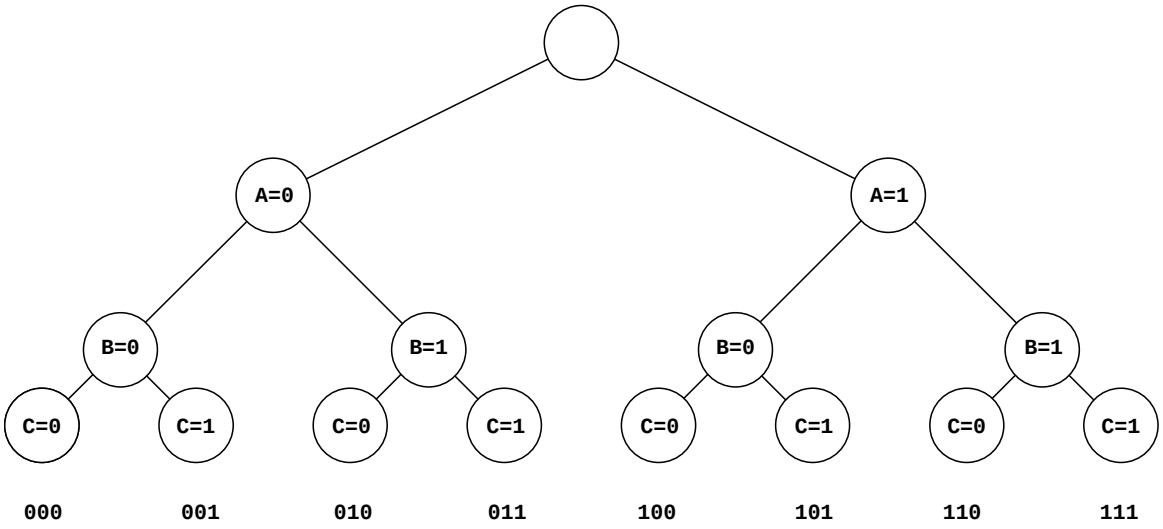


Figure 3.9: The search tree for the first two strategies

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 3.10 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.

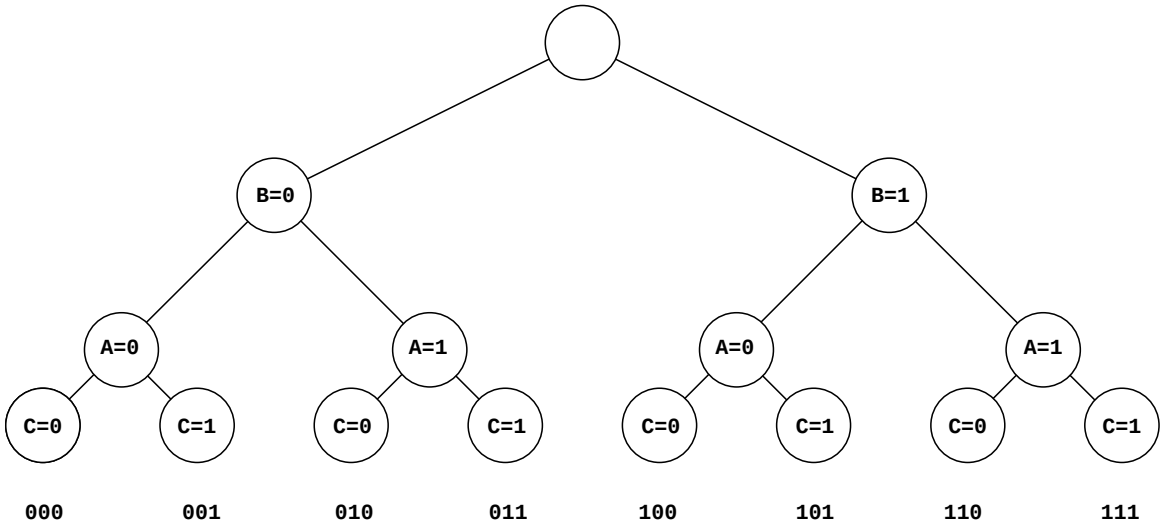


Figure 3.10: The search tree for the third strategy

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. Table 3.5 shows how many faults are aborted by the system using each strategy without the other two strategies.

Circuit	Strategy 1	Strategy 2	Strategy 3	Composite
C0432	2	6	2	0
C0499	0	25	2	0
C0880	6	0	0	0
C1355	0	0	0	0
C1908	2	1	11	0
C2670	10	14	14	10
C3540	0	4	0	0
C5315	4	15	2	0
C6288	0	0	0	0
C7552	27	18	84	0

Table 3.5: Aborted Faults for three strategies (without non-local implications)

Chapter 4

Conclusions

In this dissertation we presented the Boolean satisfiability method for generating test patterns for single stuck-at faults in combinational circuits: we extract a formula for the test set of a fault and then we satisfy that formula. This chapter summarizes the main conclusions and suggests possible future work.

4.1 Summary

The Boolean satisfiability method 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. As we discussed in Chapter 3, we can translate traditional structural heuristics into our domain, and as we discussed in Section 2.2, we can incorporate heuristics that would be difficult to implement in a structural search system.

Our system achieves total test coverage: SOCRATES is the only other system to achieve 100% coverage on the Brglez-Fujiwara benchmarks. It is not the fastest system available today (that honor goes to the SOCRATES system), but the strength of our model leads us to believe that we will gain significant performance improvements as the system matures. The structural search methods have had the benefit of more than a decade of program development and craftmanship; we look forward to the benefit the Boolean satisfiability method will obtain through similar attention.

4.2 Future Work

We would like to exploit the flexibility of our method by extending the system to handle additional fault models. By separating the form of the problem from the form of the solution, we can solve a larger class of problems than those we originally designed the system to handle. We already know how to extract a logical formula for the set of tests for single stuck-at faults. If we can extract formulas for non-classical faults, we can use our Boolean satisfier to produce test patterns for non-classical faults.

Faults that can be described as additional circuit logic can be more easily described in our system than in the traditional structural search methods. We can easily generate formulas for multiple stuck faults, or for bridging faults without feedback (commonly modelled as wired-and or wired-or failures). Bridging faults that cause feedback may also be possible. Using the Boolean satisfiability method, we can add constraints that prevent or cause oscillation, depending on our goal (for some technologies, the mere presence of a bridging fault causing oscillation could be detected by a change in circuit power consumption).

Another fault model receiving increasing attention is delay fault testing. In order to test for delay faults, we need a pair of input patterns. An algebra for test generation for delay faults has been developed by Iyengar et al. [IRS88a, IRS88b]. This algebra may be translatable into the Boolean satisfiability domain; our preliminary assessment is that it is a good match for our technique because of the ease of including constraints. For example, given a pair of patterns, it is possible to add constraints that avoid generating hazards on the lines that propagate the delay fault error to a circuit output.

One major enhancement to our system would be to modify it to detect faults in sequential circuits. Our model lends itself to application on sequential circuits by having a variable represent the value of the faulted or unfaulted circuit during a given clock cycle. We look forward to discovering if this natural extension can be made as efficient as it is conceptually simple.

One final enhancement to the current system that we would like to work on is

the development of a parallel version of the system. Much theoretical work has been done on the parallelization of satisfiability algorithms: the use of some of these ideas for something as practical as test pattern generation would be quite educational.

Appendix A

The Data

This appendix contains performance numbers for the Boolean satisfiability method described in Chapters 2 and 3 of this dissertation. Before we present the results, we need to describe the remaining pieces of the system and how they work in concert with the algorithmic test pattern generation.

A.1 The System as a Whole

Our system accepts circuit descriptions in Tegas Description Language (TDL). Before test pattern generation begins, we translate the TDL into an internal form and produce a collapsed fault list.

We collapse the faults by visiting each gate in the circuit and replacing equivalence class members with one representative. For example, since AND gate inputs stuck-at 0 and an AND gate output stuck-at 0 cause the same behavior in the output of the AND gate, any inputs stuck-at 0 may be replaced by the output stuck-at 0. Other gates are treated similarly: NAND gates with inputs stuck-at 0 and output stuck-at 1, OR gates with inputs stuck-at 1 and output stuck-at 1, and NOR gates with inputs stuck-at 1 and output stuck-at 0. Faults on the inputs of buffers or inverters are translated to the outputs. Since we never visit a circuit output before visiting all of its inputs, it is common for faults to be translated through many gates. Figure A.1 shows the faults remaining on the circuit from Figure 1.6 after fault collapsing; the

stuck-at 0 faults on the inputs of the first two AND gates are translated to the outputs, and then these two faults are translated to the output of the final AND gate. Although the circuit has 18 potential stuck-at faults to be tested, it can be completely tested for these faults by generating test patterns for only 12 faults.

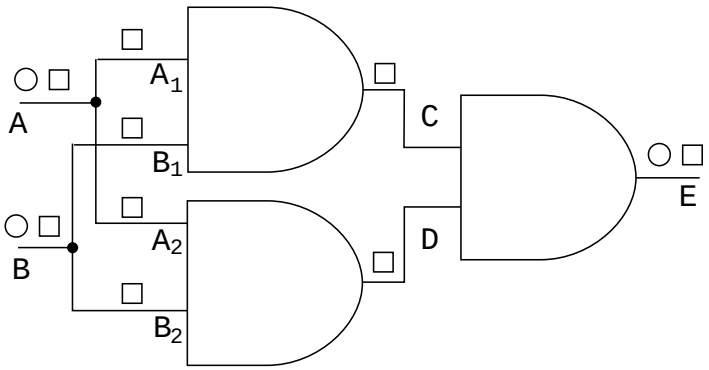


Figure A.1: Collapsed Faults: □ for stuck-at 1 and ○ for stuck-at 0

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 system is modeled after the parallel-pattern, single fault propagation (PPSFP) simulator reported by Waicukaski et al [WEF+85]. Figure A.2 shows how a 3-bit word can be used to simulate three patterns on the circuit from Figure 1.1. 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.

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 (described in Chapter 2), the fault is abandoned.

We produced test sets for ten sample circuits collected by Franc Brglez and Hideo

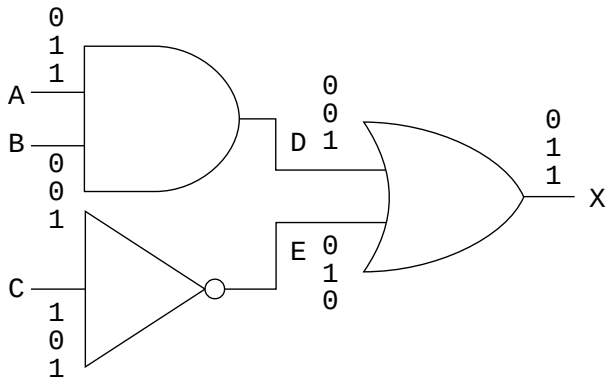


Figure A.2: Using word operations to simulate three patterns simultaneously

Fujiwara and distributed at the 1985 ISCAS Conference [BF85].

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.

A.2 The Numbers

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

Table A.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 falsfying a formula.

Table A.3 shows the number of patterns produced by each phase and the percentage of faults covered, proved redundant, or aborted by the complete system.

	Time in Seconds					
Circuit	Parsing	Random TPG	Extraction	Satisfying	Simulation	Total
C0432	.5	.9	.5	8.7	.1	10.7
C0499	.6	1.0	.9	7.9	.2	10.6
C0880	.9	2.1	1.2	39.5	.5	44.2
C1355	1.4	9.6	1.9	14.7	.5	28.1
C1908	2.0	9.7	9.7	101.8	5.7	128.9
C2670	2.9	5.3	19.5	350.0	3.2	380.9
C3540	3.7	37.3	41.6	260.7	11.2	354.3
C5315	5.4	8.5	10.8	84.7	1.8	111.2
C6288	6.9	128.1	27.0	35.5	0.0	197.5
C7552	7.9	20.9	51.0	536.9	22.3	639.0

Table A.1: Base level system timing

	Faults in Circuit		Faults covered by		Proved	
Circuit	Uncollapsed	Collapsed	Random	Algorithmic	Redundant	Aborted
C0432	864	438	421	13	4	0
C0499	998	628	595	25	8	0
C0880	1660	798	760	38	0	0
C1355	2710	1436	1389	39	8	0
C1908	3816	1754	1469	276	9	0
C2670	5340	2357	1890	351	116	0
C3540	7080	3292	3036	119	137	0
C5315	10630	4858	4762	37	59	0
C6288	12576	7616	7582	0	34	0
C7552	15104	7170	6518	521	131	0

Table A.2: Base level system number of faults

Circuit	Number of Patterns		Percentage of Faults	
	Random	Algorithmic	Covered	Redundant
C0432	70	7	99.09	0.91
C0499	53	19	98.73	1.27
C0880	94	21	100.00	0.0
C1355	90	18	99.45	0.55
C1908	64	110	99.49	0.51
C2670	95	81	95.08	4.92
C3540	190	79	95.84	4.16
C5315	191	27	98.79	1.21
C6288	47	0	99.55	0.45
C7552	297	146	98.18	1.82

Table A.3: Base level system patterns and percentage coverage

Bibliography

[Ake59] S. B. Akers. On a theory of boolean functions. *Journal of the Society for Industrial and Applied Mathematics*, 7, 1959.

[APT79] 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.

[BF76] M. Breuer and M. Friedman. *Diagnosis and Reliable Design of Digital Systems.* Computer Science Press, 1976.

[BF85] 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.

[CMPR64] W. C. Carter, H. C. Montgomery, R. J. Preiss, and H. J. Reinheimer. Design of serviceability features for the IBM system/360. *IBM Journal of Research and Developement*, 8:115–126, 1964.

[Coo71] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Anual ACM Symposium of Theory of Computing.* ACM, 1971.

[DP60] M. Davis and H. Putman. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[EW77] E. Eichelberger and T. Williams. A logic design structure for LSI testability. In *Proceedings of the 14th Design Automation Conference*. IEEE, 1977.

[FS83] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, C-31:1137–1144, 1983.

[FT82] H. Fuiwara and S. Toida. The complexity of fault detection problems for combinational logic circuits. *IEEE Transactions on Computers*, C-30:555–560, 1982.

[Fuj85a] H. Fujiwara. Fan: A fanout-oriented test pattern generation algorithm. In *Proceedings of the International Symposium on Circuits and Systems*. IEEE, June 1985.

[Fuj85b] H. Fujiwara. *Logic Testing and Design for Testability*. The MIT Press, 1985.

[Goe81] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Transactions on Computers*, C-31:215–222, 1981.

[IRS88a] V.S. Iyengar, B.K. Rosen, and I. Spillinger. Delay test generation 1 – concepts and coverage metrics. In *Proceedings of the International Test Conference*, pages 857–866. IEEE, 1988.

[IRS88b] V.S. Iyengar, B.K. Rosen, and I. Spillinger. Delay test generation 2 – algebra and algorithms. In *Proceedings of the International Test Conference*, pages 867–876. IEEE, 1988.

[IS75] O. H. Ibarra and S. K. Sahni. Polynomially complete fault detection problems. *IEEE Transactions on Computers*, C-24:242–249, 1975.

[McC86] E. J. McCluskey. *Logic Design Principles*. Prentice-Hall Publishing, 1986.

[Mic86] A. Miczo. *Digital Logic Testing and Simulation.* Harper and Row, Publishers, 1986.

[MR89] H. B Min and W. A. Rogers. Search strategy switching: An alternative to increased backtracking. In *Proceedings of the International Test Conference.* IEEE, 1989.

[PB82] 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.

[Pra86] D. K. Pradhan. *Fault-Tolerant Computing Theory and Techniques.* Prentice-Hall Publishing, 1986.

[Rot66] J. P. Roth. Diagnosis of automata failures: A calculus and a method. *IBM Journal of Research and Development*, 10:278–291, 1966.

[SE88] M. H. Schulz and Auth E. Advanced automatic test pattern generation and redundancy identification techniques. In *Proceedings of the International Fault Tolerant Computing Seminar.* IEEE, December 1988.

[STS88] 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.

[WA73] M. J. Y. Williams and J. B. Angell. Enhancing testability of large-scale integrated circuits via test points and additional logic. *IEEE Transactions on Computers*, C-22:46–60, 1973.

[WEF+85] 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.