Algorithms for Testing Boundary-Scan Equipped Circuits

by

Stephen Leslie Peters

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science

and

Master of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1992

Author . . . .
Department of Electrical Engineering and Computer Science
May 8, 1992

Certified by . . . .
Gordon D. Robinson
Company Supervisor (GenRad)
Thesis Supervisor

Certified by . . . .
Srinivas Devadas
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Algorithms for Testing Boundary-Scan Equipped Circuits

by

Stephen Leslie Peters

Submitted to the Department of Electrical Engineering and Computer Science
on May 8, 1992, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science
and
Master of Science in Computer Science

Abstract

Boundary-scan components are a relatively new development in the field of circuit board testing, yet their ability to test chips in isolation and to record the state of an entire printed circuit board indicate their potential for long-term application in the electronics industry. However, algorithms which can efficiently utilize this hardware, especially on printed circuit boards that have few boundary-scan components, have yet to be developed. In this thesis, several algorithms have been designed and analyzed, and either code samples (in C^{++}) or pseudocode have been provided for each. In the final chapter, the algorithms are compared to choose the "best" among the developed algorithms.

Thesis Supervisor: Gordon D. Robinson
Title: Company Supervisor (GenRad)

Thesis Supervisor: Srinivas Devadas
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank GenRad for the use of the equipment necessary to accomplish this thesis, as well as the project definition. Also, I would like to thank my parents, who paid roughly $6.15 for each word. Last, but far from least, I would like to thank Mary Linton – for saying yes.

Contents

1 Introduction **1**

1.1 Purpose . . . 1

1.2 Outline of Thesis . . . 1

2 PCB Testing Background **2**

2.1 Types of Printed Circuit Board Faults . . . 2

2.2 Traditional Approaches to Board Test . . . 3

2.3 The Need for Boundary-Scan . . . 4

3 Boundary-Scan Technology **6**

3.1 Overall Technological Description . . . 6

3.2 Boundary-Scan Test Techniques . . . 9

4 Scope of Thesis Research **13**

5 "Brute Force" Testing **15**

5.1 Algorithm Analysis . . . 15

6 Board-Level Fault Analysis **17**

6.1 The General Algorithm . . . 17

6.1.1 Description of Generalized Fault Analysis Algorithm . . . 18

6.1.2 Example . . . 19

6.1.3 Pseudocode Description of Algorithm . . . 20

6.2 Run-time Analysis . . . 21

6.2.1 Worst-case Analysis . . . 21

6.2.2 Random Analysis (Optimistic) . . . 21

6.3 Handling Mid-Range Values . . . 22

6.3.1 Impact on Run-time of Algorithm . . . 22

6.4 Detection of Shorts on Unscanned Nodes . . . 23

6.4.1 Impact on Run-time of Algorithm 24
6.5 Utilizing Logic Properties to Improve Test Validity 24
6.5.1 Impact on Run-time of Algorithm 25
6.6 Handling Bussed-Node Situations 26

7 Fault Dictionary Analysis 28
7.1 Fault Dictionary Background 28
7.1.1 Fault Simulation 29
7.2 Augmenting the FD with Boundary-Scan 29
7.2.1 Pinpointing Faults with Boundary-Scan 30
7.3 Pseudocode Description of Algorithm 32
7.4 Analysis of Algorithm 33

8 Comparison of Algorithms 34
8.1 Recommendations 35

A C++ Listings for Board-Level Fault Analysis 36

B The IntSet Class 45
B.1 Description of IntSet Class 45
B.2 Implementation of IntSet Class 47
B.2.1 Declaration – objects.H 47
B.2.2 Implementation – objects.C 49

List of Figures

3-1 Block Diagram of Boundary-Scan Device 7

3-2 Scan Path Through Three Boundary-Scan Devices 8

3-3 Simplified State Diagram for TAP Controller 8

6-1 Example of Open Fault on Bussed Node 26

7-1 Test Circuit for FD Analysis 31

7-2 Example of Generated Tree Structure 32

List of Tables

3.1 Sample Test Set Created by Modified Counting Sequence Algorithm 11
3.2 Test Set Created by Walking One's Algorithm 12

4.1 Table of Standard Symbols 13

6.1 Example Data Set for Shorts Analysis 19

8.1 Summary of Algorithm Efficiency 34

Chapter 1

Introduction

1.1 Purpose

Boundary-scan is a new method for structuring circuit components, such that testing access, either for the component or its surrounding connections, is much simpler. As boundary-scan is still relatively new, it is certain that the industry will see many boards in which boundary-scan components will be mixed with non-boundary-scan components. Because testing algorithms have only been created for testing boards that either have all boundary-scan components or no boundary-scan components, and also for mixed boards that have "enough" scan or "bed-of-nails" fixture access, algorithms such as these, which can test arbitrary mixed boards, are necessary.

As such, this project seeks to design, analyze, and implement new boundary-scan testing algorithms, with a special focus on algorithms used when tester access to the circuit board is minimal, and a test program is used to test the board. If successful, automated testers can use the algorithms set out here to quickly and easily test boards that contain boundary-scan circuit components.

1.2 Outline of Thesis

This document begins with a quick overview of both industry-standard techniques for testing printed circuit boards (PCBs), as well as an overview of the boundary-scan standard and its effects on testing. A short discussion of the existing boundary-scan test techniques will follow. The rest of this thesis is divided into sections describing the development of each new algorithm, accompanied with pseudocode descriptions and run-time analyses. ANSI-compliant C^{++} code used in some testing is also available in the appendices.

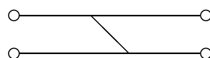
Chapter 2

PCB Testing Background

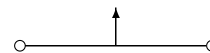
2.1 Types of Printed Circuit Board Faults

There are several types of faults that can occur during the manufacture of a typical printed circuit board (PCB). Some of the most common are:[6, p. 2-3]

- Incorrect installation – an incorrect component is installed, or the correct component is installed backwards (or not at all).
- Internal component faults – the component itself was manufactured incorrectly, or does not perform within requisite specifications. Today this tends to be extremely uncommon, with modern component manufacturing and quality control methods.
- Shorts – two tracks on the PCB are accidentally connected by a bridge of solder. This can also occur when soldering a component to a board, if the solder for one pin contacts another. This type of fault is the most common defect in through-hole boards.
- Opens – There may be a break in a PCB track, or a pin connection is not properly connected to the track (through a bent or broken pin). Note that input pins connected to an open tend to float either high or low, depending on the technology used to build the circuit. This type of faultis the most common in surface-mount technology (SMT) boards.
- Stuck-at faults – Similar to shorts, in that the node is accidentally connected to either a source voltage (often +5 volts) or a grounded pin, and therefore is no longer free-floating.


Shorts

Opens


Stuck-Hi (or Lo)

The first type of fault, incorrect installation, is for the most part either uncommon or easy to identify, given today's automated PCB manufacturing techniques. In addition, defects from incorrect installation often manifest themselves when checking for the other two types of faults. Thus, most testing methods try to target either shorts or opens when testing PCBs.

2.2 Traditional Approaches to Board Test

In the 1960s, circuit board testing meant manually putting the board through every possible state and checking that the outputs are correct for the given input. This was often done manually, and quickly became an extremely laborious process. With the advent of the integrated circuit (IC), the board complexity reached the point where manually running through all possible inputs on a single board took hours of the test technician's time. For this reason, testing quickly became the highest cost element in electronic products.

Therefore, the advent of automated testing equipment (ATE) came as a welcome addition to the testing workbench. With the ability to automate the test came speed, as the ATE could run through the possible inputs far faster than a human, and could automatically check the outputs against a small database.[6, p. 1-2]

The above approach to testing, in which a series of inputs are applied and the outputs are tested against a database of "correct" outputs, is known as functional testing. This is probably the oldest automated testing technique, and it works well in making sure the circuit board works to specification. However, in the event of a failure, functional testing was often unable to provide pointers as to where the problem lay, because it could only notice errors at the outputs. When bussed nodes (in which many ICs may be connected to a single node) became common, it became nearly impossible to tell which of the ICs had failed, as dozens of chips could have caused the problematic output.[6, p. 1-3] In addition, as complexity of ICs increased with the onset of large-scale integration (LSI), the cost of cataloguing the possible outputs became extremely high.

In order to offset these difficulties, methods for increasing the fault-locating facilities of the standard functional test were created. One of these is the use of a guided probe. The probe works by effectively providing a relocatable access point to the circuit. Guided by on-screen requests, the operator places the probe at different points, and the ATE runs through the functional test while cataloguing the values received at that node.

In order for guided probe testing to work, a database must be created containing the correct values for every node in the circuit. This is done by modelling the circuit in the ATE, and then simulating the operation of the circuit with the given test program, building up the database as the test progresses.

With the ability to simulate the circuit in software came yet another technique to increase the error-location abilities of functional testing. By running the same test program on a circuit with simulated faults, a database of failure patterns (called a "fault dictionary") can be created, in which a failure pattern can be matched to a subset of possible faults in the circuit. Thus, when an actual test of a board is run, the fault dictionary can be consulted to get an idea of where to look for faults, or to get a suitable test node for the guided probe.

In the mid-1970s, in-circuit testing was also developed. Functional testing often would take up large amounts of computer time, as well as operator time for diagnostics, while this new testing technique tended to work in a much more straightforward fashion. In-circuit testing operated on a different principle as functional testing: it operated by testing each individual component, and assumed that as long as the individual parts of the circuit worked, so would the entire board.

Modern in-circuit testing operates by connecting the board to a "bed of nails" – a series of probes which connect to the component pins after they are inserted through the PCB. Each of these nails also had the ability to override the value being driven to the node and drive its own value. This creates a connection to each and every node on the board, and gives the ATE the ability not only to read logic values from each node, but to drive logic values as well. Thus, every component can be tested in isolation, by providing descriptions of each component's operation.

This approach does give the ability to pinpoint errors on the board, but it also requires a cost in work – creating and wiring a nail fixture for each board, defining the nail locations for all the nodes, defining the operation for each single component, etc. In addition, it requires that the components be mounted such that the component leads extend through the board, or be routed to a space-consuming test access pad elsewhere on the PCB.

In the mid-1980s, combinational testers, with the ability to handle both the in-circuit and functional tests, appeared. These allowed full multi-strategy testing, so test technicians could pick and choose the most suitable method for testing PCBs.

2.3 The Need for Boundary-Scan

Unfortunately, new advances in technology have made the in-circuit and guided probe methods of testing difficult to perform. First, new surface-mounted devices (SMDs) are being produced. These ICs have their pins mounted directly to the surface of the circuit board, without drilling holes in the PCB, thus rendering standard bed-of-nails in-circuit testing ineffective, unless test pads were placed on the board.

In addition, the SMDs do not need nearly as much space. Because holes do not need to be drilled through the PCB for the component leads, it is possible to pack many more leads in the same

amount of space. Whereas the older, standard dual-inline package (DIP) form would have about 100 mils (1/10 of an inch) of space between leads, SMD devices routinely have only 25-50 mils of space, with some specialized devices packed as tightly as 12 mils. These small sizes make using a guided probe next to impossible, due to the inability of human operators to identify a single pin and hold the guided probe against it without slippage. Automated guided probing, using machines to control the probe, is expensive to implement. In addition, guided probes cannot be used on pin grid array (PGA) components (including modern CPUs), where most of the pins are unreachable on the operator side of the board.

In addition, the functionality of components has increased at a much faster rate than has the number of pins used on typical components. As such, the ability to control an in-circuit test, even if other standard methods were possible, is being curtailed [5, pp. 3–5]. A technology tailored to testing board integrity, while minimizing costs in PCB real estate and tester hardware, was needed and therefore created. The result was boundary-scan, as defined in the next chapter.

Chapter 3

Boundary-Scan Technology

In February of 1990, the Institute of Electrical and Electronics Engineers released standard document 1149.1-1990 [2], specifying a standardized method of performing on-chip testing, called boundary-scan. This new technology is given a brief overview in the following sections. A far more thorough description can be found in [2], as well as the Maunder-Tulloss tutorial on boundary-scan [8].

3.1 Overall Technological Description

Every IEEE 1149.1-compliant boundary-scan component is equipped with three (3) dedicated testing inputs and one (1) dedicated testing output (refer to diagram in Figure 3-1). Two of the inputs, the test clock (TCK) and the test mode select (TMS) are used for control purposes. Together, they control a 16-state finite state machine existing on each compliant component. This FSM is known as the test access port (TAP) controller, and specifies the various operations being performed on the data traveling through the component. In addition, each component internally contains an instruction register (IR) and a bypass register.

The data itself is sent through using the two remaining access ports: the input/output combination of test data in (TDI) and test data out (TDO). Between TDI and TDO is a 1-bit wide shift register, with each register connected to an input or output pin. Thus, components can be strung together in a long chain via the TDI/TDO connections. Using this data path, all information can be shifted to and from the components serially. In addition, the path can be channeled through a component's bypass register, effectively shutting that component off from the data path (see Figure 3-2, in which the third component is bypassed, although still physically connected to the scan path).

All control on the chip is handled by the TAP controller. A simplified state diagram of the

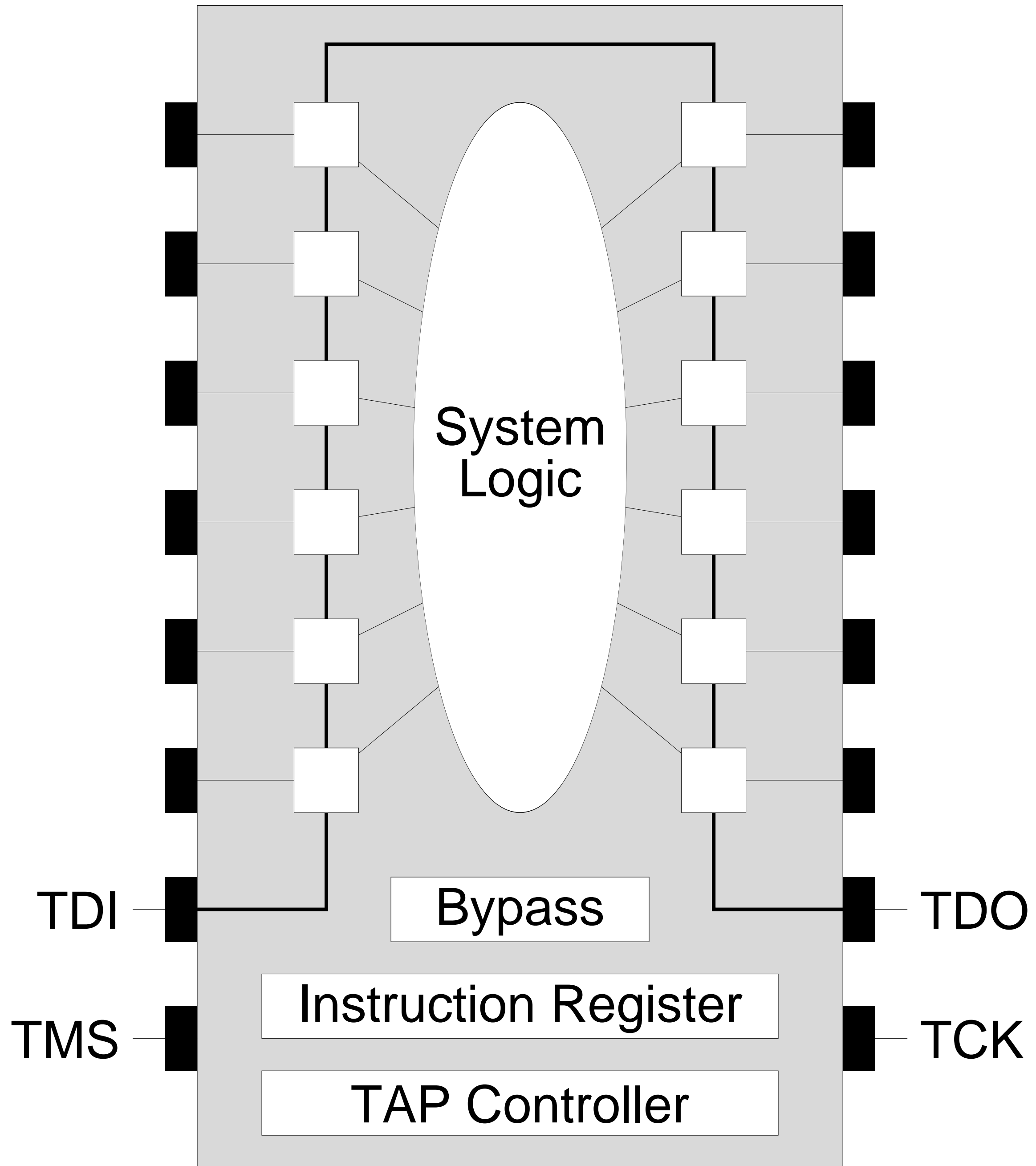


Figure 3-1: Block Diagram of Boundary-Scan Device (from [5, p. 13])

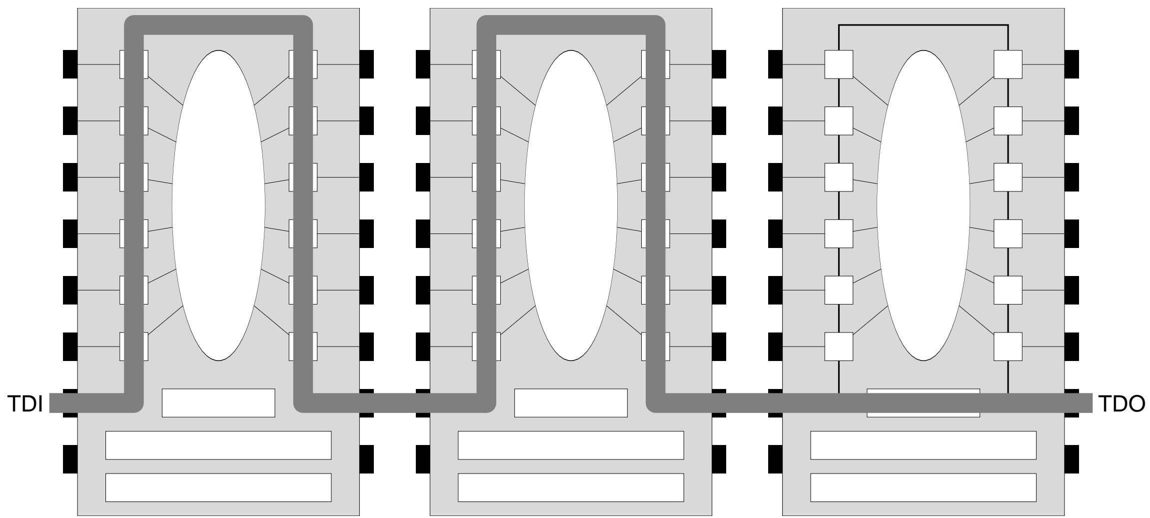


Figure 3-2: Scan Path Through Three Boundary-Scan Devices

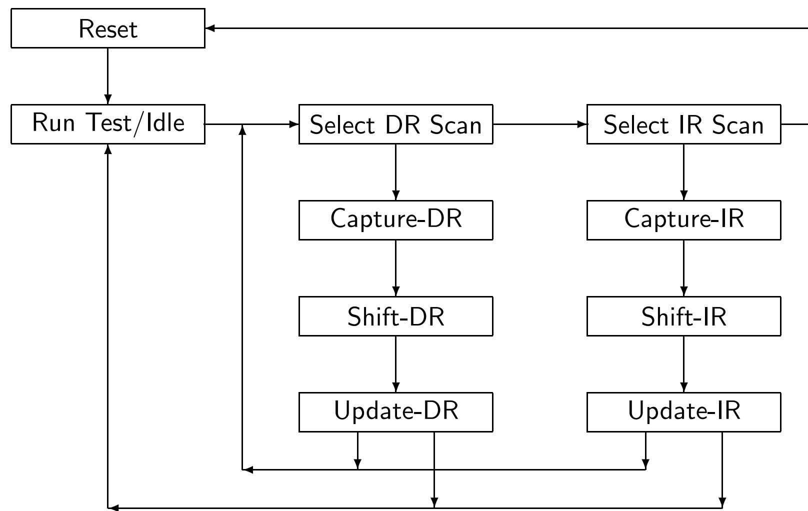


Figure 3-3: Simplified State Diagram for TAP Controller (from [9])

controller is shown in Figure 3-3. The finite state machine is able to control:

- Whether the boundary-scan circuitry is active or not (Reset, Run Test/Idle states)
- Whether the scan path goes through the instruction register (IR Scan) for loading instructions, or through the data registers (DR Scan) for information flow.
- Timing for information capture (Capture-DR) and information update (Update-DR).
- Timing for instruction fetch (Capture-IR) and load (Update-IR).

In addition, a standard set of mandatory instructions are used to control the boundary-scan circuitry:

- **Extest** – loads values into the output cells of the component and reads the values on the input cells. This is the prime instruction for most existing boundary-scan tests, which attempt to discover board-level faults by placing distinct patterns on the output cells.
- **Sample** – reads values from the input cells. This is used with existing test programs, to test the values at a given point in time. This is the instruction used most often in this thesis.
- **Bypass** – instructs boundary-scan circuitry to ignore this component, and channel the data path through the bypass register.

Other instructions are considered "optional", and may or may not be implemented by the manufacturer of the compliant chip. Thus, these three are the only instructions that the boundary-scan circuitry can be relied upon to handle with complete certainty. As they only perform simple data manipulation, the ATE must be used to perform all calculations relating to test generation, as well as issue commands to the boundary-scan circuitry during the actual test. The role of the boundary-scan controllers is relegated to sampling the circuit's node values at different points in time.

3.2 Boundary-Scan Test Techniques

The IEEE committee creating boundary-scan, in keeping with its mission to facilitate testing, have (along with participating individuals) already defined some simple methods of testing boundary-scan equipped PCBs. These are briefly described below (more detailed explanations can be found in [5, pp. 31–75], which provided much of the material in this section).

The first method uses one of the optional boundary-scan instructions: **Intest**. Using this instruction sets the circuitry into a mode where the boundary-scan component can be tested without affecting the rest of the circuit. During the "capture" state of the TAP controller, the component's

output cells are loaded. During the "shift" state, a new set of test data is shifted into the input cells and the output from the capture is shifted out. Finally, during the "update" state, the new input data set is applied to the component circuitry. Thus, all the ATE must do is provide a series of test vectors to apply to the component inputs, and compare the output vectors against a database or test program.

This method gives many of the nice features of an in-circuit fixture, in that a component can be completely isolated from the rest of the circuit and tested, without physically removing the component from the board. Because no fixture board is needed, in-circuit testing using boundary-scan is a great deal cheaper than the older equivalent. The main drawback, however, is the speed with which the circuitry works. Because the TAP controller typically runs on a much slower clock than the rest of the circuit, and shifts data in series rather than the in-circuit's parallel access via the fixture, shifting data out for perhaps thousands of data vectors (given a fairly complex component) will take far longer to accomplish.

One option to this is another optional command, **Runbist**, for Run Built-In Self-Test. This will access a test program wired into the circuitry, and provide the results through the scan path. If many components contain this instruction, all component tests can be quickly run in parallel, and the data shifted out and analyzed.[5, pp. 24–27]

Neither of these commands, however, address board-level faults. They focus completely on faults occuring within the component itself (which are admittedly rare), and were created as optional instructions primarily for that reason.

Other testing methods for boundary-scan boards focus on the interconnections between two (or more) boundary-scan components, or between a boundary-scan component and an in-circuit fixture nail. For example, one fairly standard method is to use the **Extest** command to drive different patterns onto the output pins, and examine the input pins of other boundary-scan components to ensure that the values are correct. By driving a unique pattern onto each node, it is possible to identify exactly which two nodes were shorted together. In addition, open circuits can be detected whenever a constant pattern (all high values or all low values) is found on an input pin.

In order for this method to work, however, either boundary-scan nodes must exist on the entire circuit, or fixtures must be built to check values connected to the same nodes as the boundary-scan components, to ensure that there are no shorts or opens at the boundary-scan pins.

For situations where we have enough access, there has already been substantial work performed on diagnosis of faults. Much of this is summarized in the following paragraphs. More information and a far more in-depth technical analysis of these algorithms can be found in [7].

One of the major methods of using boundary-scan to test shorts between adjacent component

Node	Counting Vectors			
	t_1	t_2	t_3	t_4
n_1	0	0	0	1
n_2	0	0	1	0
n_3	0	0	1	1
n_4	0	1	0	0
n_5	0	1	0	1
n_6	0	1	1	0
n_7	0	1	1	1
n_8	1	0	0	0
n_9	1	0	0	1
n_{10}	1	0	1	0

Table 3.1: Sample Test Set Created by Modified Counting Sequence Algorithm (taken from [7])

pins is to use **Extest** to apply a set of independent test vectors to each node in the circuit. One of the most common is known as a "Modified Counting Sequence" algorithm, and operates by assigning a different number to each node, and then applying that number in binary to the node during the test step (See Table 3.1).

However, this method does not address two problems with the diagnosis of shorts:

1. *Aliasing syndromes* can occur if two nodes, when shorted together, respond with the same pattern as one of the other nodes. For example, in Table 3.1, nodes n_3 and n_4 have the same response as n_7 (assuming that a value of "1" will dominate when a short occurs – if a "0" is the dominating logic value, the same effect can be achieved by inverting each element of the test vectors). Thus, it is impossible to tell, when a short occurs, whether n_3, n_4, and n_7 are shorted together or whether it is just n_3 and n_4.

2. *Confounding syndromes* occur if two nodes, when shorted together, respond with the same pattern as two other nodes do when shorted together. For example, n_4 and n_{10}, when shorted, produce the pattern "1 1 1 0", which is the same pattern received when n_6 and n_8 are shorted. Thus, when this pattern occurs, it is impossible to tell whether all four are shorted together, or whether only one of the pairs is shorted together.

By modifying the test set used to find the shorts, these problems can be avoided. One example of this is in the use of the "Walking One's" algorithm to create the test set, as shown in Table 3.2. In this test, it is impossible to have any aliasing or confounding syndromes, because the patterns are designed such that any short between nodes results in an output pattern precisely identifying which nodes are shorted together (i.e. if nodes n_4 and n_7 are shorted, the output pattern is "0 0 0 1 0 0 1 0 0 0", with 1's in the fourth and seventh places).

Node	Walking One's Algorithm Vectors									
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
n_1	1	0	0	0	0	0	0	0	0	0
n_2	0	1	0	0	0	0	0	0	0	0
n_3	0	0	1	0	0	0	0	0	0	0
n_4	0	0	0	1	0	0	0	0	0	0
n_5	0	0	0	0	1	0	0	0	0	0
n_6	0	0	0	0	0	1	0	0	0	0
n_7	0	0	0	0	0	0	1	0	0	0
n_8	0	0	0	0	0	0	0	1	0	0
n_9	0	0	0	0	0	0	0	0	1	0
n_{10}	0	0	0	0	0	0	0	0	0	1

Table 3.2: Test Set Created by Walking One's Algorithm (taken from [7])

The two algorithms above, Modified Counting and Walking One's, have strong and weak points. The Modified Counting Algorithm works with a small number of tests, whereas Walking One's requires many tests, although it performs a better diagnosis. One way of getting the best of both worlds is to perform a Modified Counting algorithm, and examine the output to identify a set of nodes which may be creating aliasing or confounding syndromes. This set of nodes can then be tested using a Walking One's Algorithm. This method is called the "W-Test Adaptive Algorithm", and tends to require only slightly more tests than the Modified Counting Algorithm to get the diagnostic power of the Walking One's. The paper by Jarwala and Yau [7] presents this algorithm, as well as a similar one which only requires $C + \lceil \log(N+2) \rceil$ tests, where N is the number of nodes under test, and C is the highest degree of confounding (the maximum number of independent faults with the same syndrome).

Chapter 4

Scope of Thesis Research

As noted in the last chapter, the existing test methods only work for boards, or parts of boards, which have extremely good access to components, either through the boundary-scan devices or through fixtures. Although the components themselves can be checked (through the optional **Intest** and **Runbist** instructions), the interconnections exterior to the component can only be checked if a fixture nail or another boundary-scan component is connected to each pin of the component being tested.

Because it will most likely be several years before boundary-scan is utilized to the point of 100% coverage, it is necessary to look at ways to perform boundary-scan testing when this much coverage is not available, i.e., where boundary-scan components intermingle with ordinary components without access. Because it will not be possible to look at every single node in the circuit, these testing methods must use functional testing instead of in-circuit.

Thus, the main question for creating an efficient mixed-board test lies in finding ways to minimize the number of times that the functional test program is run. Presented in this thesis are methods of analyzing the circuit in an attempt to find a minimal set of test steps to examine.

Several standard symbols are used in this document. They are as shown in Table 4.1.

Note that for each method, the approximate run time for a single test is given as $O(b+t)$, where

Symbol	Description
n	Number of nodes on the circuit board
t	Number of test steps in the functional test program
b	Number of nodes containing boundary-scan access
b/n	Proportion of nodes containing boundary-scan access

Table 4.1: Table of Standard Symbols

b is the number of nodes with boundary-scan, and t is the number of test steps in the test program. The reason for this time is that the test itself will take $O(t)$ time to run on the circuit board, and the time to shift all the data collected through the scan path is $O(b)$. Hence, if a testing method requires that the test program be run $O(t)$ times, then the total time to test the board will be $O(bt + t^2)$.

One important fact to keep in mind for the following chapters is that the full estimates for the running time of the test are only valid when diagnostics are required. If the test passes (found by checking the outputs after one run of the test in time $O(b + t)$), no more time need be taken trying to track down defects in the board. Assuming that defects are rare, running the test will only take $O(b + t)$ per board, rather than whatever figures are given for worst-case or average-case run time.

Chapter 5

"Brute Force" Testing

Probably the most straightforward method of functionally testing a boundary-scan equipped board is to use a "brute force" method of checking every boundary-scan node at every possible test step to ensure that the values on the scan nodes are correct throughout the test program.

The major drawback to this system is its exhaustive nature: the test program must be run from beginning to end every time a test step is examined, which using the "brute force" method is once for every test step.

Although the concept at first glance may seem woefully inefficient (in that it threatens to examine every single data point in the system), in the long run it does a very good job at locating a fault swiftly. This is due to the fact that a fault tends to cause errors early in the test, and these errors often propagate relatively quickly to a visible point (in this case, any boundary-scan node or any output node).

5.1 Algorithm Analysis

As stated before, the function is extremely straightforward:

```
BRUTE-FORCE-TEST(n, t)
1   i ← 1
2   while i ≤ t do
3           if BSCAN-TEST(i) ≠ Good do
4              return i
5   return NIL
```

Obviously, the worst-case run time is going to be on the average of $O(t)$, multiplied by the time

to run the test ($O(b+t)$), thus giving a result of roughly $O(bt+t^2)$. If the test steps where we could detect an error was distributed more or less evenly across the entire range of the test program, then this value would also serve as a good approximation at the average run time.

However, most of the faults can easily be detected within the first few test steps. Roughly half of all possible pin faults will occur during the first test step, and be detected in the time it takes for the error to propagate to a scan point. As a rough estimate, the fault will be noticed in $O(n/b)$[1], thus giving $O(n + tn/b)$ as the average run time of the algorithm.

Unlike the other algorithms being presented in this paper, this one involves absolutely no time for preparation, beyond that of setting up a database of the "correct" values for the circuit. Unfortunately, there are a few basic deficiencies of the brute force algorithm. First, it is not at all a given that a fault will exhibit incorrect behavior at any of the scan points. The faulty behavior can get "swallowed" by another component and not propogate any further. In this case, no failure will be noticed at all. Second, the algorithm only tells the time and place where the fault is first noticed. It is up to the ATE to perform further diagnostics to locate the exact position of the fault, just as in standard functional testing. However, the brute force method is certainly an improvement over standard functional testing, as the boundary-scan nodes do provide much more access to the various components on the board.

[1] This is purely an estimate, based on the assumption that the propagation time will be roughly equivalent to the time that it takes to randomly locate a boundary-scan node, which of course is inversely proportional to the percentage of boundary-scan nodes. It is interesting to note that the actual time before a fault is noticed is extremely short. [4, pp.47-50] presents data showing that, for "typical" circuits with roughly a thousand components, 80-90% of the faults are located within 10 test steps at the outputs. Boundary-scan nodes would effectively create more outputs, so this figure can only be improved upon with the new technology.

Chapter 6

Board-Level Fault Analysis

One way to cut down on test time is to cut down the search to find specific faults. Because, due to methods of PCB creation, some faults appear more often than others, it makes sense to create a testing method which focuses on a small but common subset of faults. Analysis of the way in which these faults can occur can potentially cut down on the number of test steps examined. This is particularly useful when faults do not appear until late in the execution of the test, where algorithms such as the brute force method introduced in the last chapter tend to become quite inefficient.

The three most obvious faults to target are shorts, opens, and stuck-at faults (all of which are described in section 2.1). Note that these three can all be considered variations on a single fault category, for the following reasons: most opens usually act like a stuck-high or stuck-low, due to the tendency for an unconnected node to "float" in one direction or the other.[1] In addition, both stuck-high and stuck-low faults can be considered as shorts to either ground or the supply voltage. Hence, this document shall consider a quick and easy method for detecting shorts in a boundary-scan equipped circuit, as the other two types of faults can usually be detected from the same test.

6.1 The General Algorithm

Shorts occur when two nodes that should be distinct become connected (whether through a solder bridge, bent pins, etc.). Hence, the values on the nodes tend to be the values being driven by the stronger of the two connected nodes, rather than being independent as usual.

This implies one method for finding the set of test steps needed to find the defects – locate the subset of test steps needed to differentiate each node from every other in the circuit. By testing just

[1]This does not happen when a node is bussed (i.e. attached to more than one driving pin). This situation is addressed later.

enough to determine if the signals are non-homogeneous, a very efficient board-level test is created.

Note that such a test does not presume to be able to tell *which* two wires have been shorted together. The test is prepared such that it is sufficient to ensure that no leads have been shorted, but does not contain enough information to specifically target the two (or more) involved nodes. However, it should in general identify at least one of the participants in a short, if there are no problems due to aliasing or confounding syndromes.

As before, t is the number of test steps within the test program, b is the number of scanned nodes, and n is the total number of nodes in the circuit. $\mathcal{N}$ represents the set of all accessible nodes in the circuit (thus, $|\mathcal{N}| = b$, because, for the most part, the only nodes that we can analyze are those with scan access).

6.1.1 Description of Generalized Fault Analysis Algorithm

The most straightforward method for determining a small subset of test steps needed to differentiate between faults is to successively split the set of nodes being tested into two smaller disjoint subsets, using the nodal values during a given test step as the criteria for a split. Repeating this operation on the subsets will eventually separate the entire system. If a test step effectively divides a set of nodes, it will be added to the set of tests used by the boundary-scan system to detect shorts.

For example, if we start with the set of nodes $\mathcal{N}$, where $|\mathcal{N}| = b$, the first test step will effectively divide $\mathcal{N}$ into two parts: those nodes which on the first step hold a "low" value ($\mathcal{N}_0$) and those nodes which on the first test step hold a "high" value ($\mathcal{N}_1$).[2] The second test step can further split these two sets into smaller sets ($\mathcal{N}_{00}$, $\mathcal{N}_{01}$, $\mathcal{N}_{10}$, and $\mathcal{N}_{11}$), and so on. If a test step can divide a set into two distinct, non-zero subsets, then that test step will be considered a useful test step, and will be included when the final set of tests is created. If not, that test step is skipped.

The algorithm stops when either (a) the set $\mathcal{N}$ has been divided successfully into $\{\mathcal{N}_a, \mathcal{N}_b, \ldots\}$ such that, for all $x, |\mathcal{N}_x| = 1$; or (b) all t test steps have been examined, and the set $\mathcal{N}$ is still not fully divided. If the latter has occurred, there are two or more nodes which maintain the same values throughout the given test program, and they cannot be distinguished from each other[3]. Running a test with these two nodes shorted may result in an undetected error. The tests that have been found and considered "useful" are still catalogued in this case.

[2] A note on syntax: The convention $\mathcal{N}_x$ stands for the set of nodes who, when examined on a given series of test steps, have a resulting pattern, or syndrome, of x. Thus, on the series of test steps, the nodes in $\mathcal{N}_x$ are all indistinguishable from another.

[3] This is a fairly common situation, since a board designer may need to provide a often-used signal, such as a clock, to many parts of the board. In order to prevent problems with fanning out a signal, the designer will create a series of buffers for the signal, in effect making many new nodes, each with exactly the same behavior

6.1.2 Example

Test Step	n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9
1	1	0	1	0	0	0	1	1	0	1
2	1	0	0	0	0	1	0	1	0	0
3	1	1	0	0	1	0	0	0	1	1
4	1	0	0	0	1	1	0	1	1	0
5	1	1	0	1	0	0	0	0	0	1
6	1	1	1	1	0	1	0	1	1	0
7	0	1	1	1	1	0	1	1	1	1
8	0	0	1	1	1	1	1	0	1	0
9	1	1	1	0	0	0	0	1	0	1

Table 6.1: Example Data Set for Shorts Analysis

By way of example, let us take a small circuit, with 10 nodes and a 9-step test program, with the data in Table 6.1 being the "expected" values in a board with zero defects:

Implied, but not shown, are two more nodes GND (which has solid 0's as an expected value) and VCC (which has solid 1's). The test run will then perform as follows: In the first pass through, we split the nodes on the basis of the information in test step 1. From this step, we split the nodes into two groups: the nodes $\{n_0, n_2, n_6, n_7, n_9, \mathrm{VCC}\}$ into one group (nodes with a logic 1 in test step 1) and the nodes $\{n_1, n_3, n_4, n_5, n_8, \mathrm{GND}\}$ in the other.

Taking the first group, we look at test step 2. This step will divide the group into $\{n_0, n_7, \mathrm{VCC}\}$ and $\{n_2, n_6, n_9\}$. Again taking the first group, test step 3 will divide $\{n_0, n_7, \mathrm{VCC}\}$ into the two parts $\{n_0, \mathrm{VCC}\}$ and $\{n_7\}$. Test step 7 must also be added in order to distinguish n0 from VCC. Step 3 will be able to divide $\{n_2, n_6, n_9\}$ by splitting off $\{n_9\}$. Test step 4 cannot divide $\{n_2, n_6\}$, because both nodes maintain equal values, as they do until test step 6. Thus, test step 6 will divide $\{n_2, n_6\}$ into its constituent parts. Hence, in order to ensure that no shorts exist between $\{n_0, n_2, n_6, n_7, \mathrm{VCC}\}$, the test must check test steps 1, 2, 3, 6, and 7.

In the second group, if we examine test step 2, we can divide the group into $\{n_1, n_3, n_4, n_8, \mathrm{GND}\}$ and $\{n_5\}$. Test step 3 will divide the $\{n_1 \ldots\}$ group into the two groups $\{n_1, n_4, n_8\}$ and $\{n_3, \mathrm{GND}\}$. Step 4 will split off $\{n_1\}$ from the first group. Test step 6 will split $\{n_4, n_8\}$ into its different parts. Finally, test step 5, will split $\{n_3, \mathrm{GND}\}$. Thus, for the second group, the testing must take place on steps 1, 2, 3, 4, 5, and 6.

Thus, we can ensure a lack of shorts if we test the first seven test steps, and no further. Note that this is far from the most efficient test. By testing only steps 1, 2, 3, 6, and 8, the entire data set can be split. However, performing an analysis to find the *smallest* subset could cost more test preparation time than would be desirable, especially when the number of nodes being analyzed

climbs into the thousands.

Note that one small adjustment can be made in order to improve the test generation: As the list of tests is created, that list is used for deciding on subsequent tests. That is, if we already have a list with test steps 1, 2, and 6 in it after working on one subset of nodes, step 6 will be used to split groups before steps 3, 4, and 5.

If this modification were used in the above example, then the first group would be split using test steps 1, 2, 3, 7, and 6 (in that order). When the second ($\{n_1 \ldots\}$) group was split, it would have divided into $\{n_1, n_4, n_8\}$ and $\{n_3, \text{GND}\}$ as before, but then would have examined test step 7, splitting $\{n_3, \text{GND}\}$ (but leaving $\{n_1, n_4, n_8\}$ alone). Examining test step 6 would have created $\{n_1, n_8\}$ and $\{n_4\}$. At this point, test step 4 would finally be examined, splitting the last group into its constituent parts. Doing this creates the set of test steps 1, 2, 3, 4, 6, and 7 (shaving off a test in the process).

This works by making sure that previous tests are used to their fullest extent before going on and using tests whose results could be duplicated elsewhere in the process.

6.1.3 Pseudocode Description of Algorithm

The generalized version of the algorithm is as follows. For explanation, $\mathcal{N}$ is the set of all nodes, t is the number of steps in the test program, and *testset* is the final set of tests to be analyzed when the test is run in practice. The $Data(i, j)$ subroutine finds the expected node value for node i at test step j.

```
SPLIT-SET(s, j)
1   if s = ∅ or j > t do
2      return
    ▷ Following splits set s using test step j
3   lo ← hi ← ∅
4   for i = each node in set s do
5       if Data(i, j) = 0 do
6          lo ← lo ∪ i
7       else
8          hi ← hi ∪ i
    ▷ Now add to testset if split
9   if lo ≠ ∅ and hi ≠ ∅ do
10     testset ← testset ∪ t
11  SPLIT-SET(lo, t + 1)
12  SPLIT-SET(hi, t + 1)
```

```
FAULT-ANALYZE()
1   testset ← ∅
2   SPLIT-SET(𝒩, 1)
```

Note that for this to work well, a method of dealing with sets needs to be created. Fault-Analyze initializes the *testset* set and starts the test running on the very first test step in the test program, by calling Split-Set.

Split-Set, in lines 1 and 2, makes sure that the inputs for the routine are reasonable. It then (lines 4-7) loops through every node and divides it according to logic values on test step j, placing the results into a *hi* and a *lo* set. If a useful split has occurred, *testset* has the current test step added. In any case, both *hi* and *lo* are subsequently analyzed using the Split-Set procedure.

6.2 Run-time Analysis

6.2.1 Worst-case Analysis

In the worst case, it will be necessary to check every test step during the run of the test. This makes it no worse than the brute-force method of testing, and so the test runtime grows as the function $O(bt + t^2)$.

In preparing for the test, the worst case scenario is that every single test step will have to be examined to split every single node (one obvious example is when a good number of the nodes do not change until extremely late in the test program). This will give a growth function of $O(bt)$.[4]

6.2.2 Random Analysis (Optimistic)

As a rough approximation, let us assume that all data within the test is random. This may be far from reasonable, as random data tends to change often and quickly, whereas user-generated tests may have long strings where values remain the same. However, the approximation serves to give a good "lower-bound" estimate on the processing time required. In addition, assume that the data already exists in a structure in which each element can be accessed in $O(1)$ time.

In running the test on randomized data, any given test step should be able to divide a set of nodes roughly in half. Thus, the number of test steps needed to accomplish the test at run time is only $\lg b$. Since each test takes time $O(b + t)$ (see Chapter 4), the total run time resolves to be $O((b + t) \lg b)$.

When generating the board-level test, every time we examine a test step, we effectively divide our work in two. With random data, these two sections will be roughly equal in size. Also, the processing needed at each step is roughly $O(n)$ (n here representing the number of nodes being

[4] The value of $O(bt)$ will hold even if the data is examined using a non-linear method, such as the group-splitting function described above. As long as each node is examined just once for each test step, the value will hold. Proof of this can be found by extending a proof of the run-time for a quicksort algorithm, found in [3, pp. 157-159]

analyzed at any step of the algorithm), as we must examine each node to decide in which section it belongs. Hence, the work performed in the test generation can be found using the recurrence formula:

$$\mathcal{T}(n) = \begin{cases} O(1) & \text{if } n \leq 1 \\ 2\mathcal{T}(n/2) + O(n) & \text{otherwise} \end{cases}$$

Evaluating the recurrence shows that, in a random-data case, the test can be created in time $\mathcal{T}(b) = O(b \lg b)$. As this is probably the most optimistic estimate of run-time ($O(bt)$ being the worst), we shall evaluate the total run-time of the test generator as being bounded below by $\Omega(b \lg b)$.

Unfortunately, as stated before, such random data is rare, as most circuits tend to vary only a small number of nodes from test step to test step. However, randomizing the algorithm may help things along towards this optimistic case. Rather than traversing the set of test steps in a sequential fashion, the algorithm can select an as-yet-unused test step at random. This will help the algorithm runtime to be more like the random case given above.

6.3 Handling Mid-Range Values

The algorithm as given in the preceding sections works only for a small subset of the possible boards – those which have perfectly well-defined logic values, and full scan access. This section and the later sections of this chapter address the real-world deficiencies of the algorithm and correct them.

In most test programs, not all values are well-defined. During the start of the test, it is not uncommon for several nodes to be dependent on whatever values they held at power-up. Thus, these values could be a logic 1, a logic 0, or any value in between, until the circuit enters a definite state. In circuit discussions, these nodes are usually labeled "$\times$" at the appropriate times.

In order to handle these mid-range or "don't care" values, a slight modification to the algorithm must take place. First, the "don't care" values, when dividing the set $\mathcal{N}_x$, need to be added to both $\mathcal{N}_{x0}$ and $\mathcal{N}_{x1}$. Second, a test step is not added to the final test if the division results in all "don't care" values in either $\mathcal{N}_{x0}$ or $\mathcal{N}_{x1}$. Before this modification, a test was not added only if the split resulted in one of the two subsets being empty.

6.3.1 Impact on Run-time of Algorithm

If "don't care" values appeared with the same frequency as do logic 1's and 0's, then the algorithm would be much worse than normal – the recurrence formally given above as $\mathcal{T}(n) = 2\mathcal{T}(n/2) + O(n)$

would become:

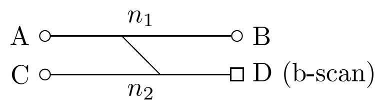
$$\mathcal{T}(n) = \begin{cases} O(1) & \text{if } n \leq 1 \\ 2\mathcal{T}(2n/3) + O(n) & \text{otherwise} \end{cases}$$

and thus would require a work time of $\mathcal{T}(b) = O(b^{\log_x 2})$ (where $x = 3/2$) $\approx O(b^{1.71})$ to calculate the test. Luckily, such mid-range values do not appear with anywhere near this frequency, and are usually involved only in power-up transients. Thus, the asymptotic growth of $\mathcal{T}$ would be far closer to $b \lg b$ than the exponential function above.

6.4 Detection of Shorts on Unscanned Nodes

In many boards (especially within the next few years, before boundary-scan reaches full acceptance), the number of nodes actually attached to boundary-scan components will remain low, in comparison with the rest of the board. By only examining the boundary-scan nodes, as we have been, we fail to recognize the possibility that a node lacking the boundary-scan capability will be shorted with a node containing the capability. Gaining the ability to detect these faults will present a substantial increase in power of our test.

The situation is as follows:



The only node which is boundary-scan capable is n_2, which has a boundary-scan lead at D. Node n_1 has no such capability. There is an unintentional short between them.

If both n_1 and n_2 had boundary-scan and were shorted together, we could detect it, because we would know a test step in which they differ and test them both, ensuring that the error be caught (regardless of whether the final values on the short are those of node n_1 or node n_2). However, because n_1 cannot be tested via boundary-scan, we can only detect the situation where n_1 dominates n_2.

We can modify the algorithm further to deal with when this situation occurs.

Assume that $\mathcal{B}$ is the set of boundary-scan equipped nodes, and $\mathcal{N}$ is the set of all nodes in the circuit. We will take for granted that $\mathcal{B} \subseteq \mathcal{N}$.

In order to detect situations where the non-boundary-scan nodes dominate the boundary-scan nodes, the modification is straightforward: stop processing a subset $\mathcal{N}_x$ when every member of the subset lacks boundary-scan capability, that is, $\mathcal{N}_x \subseteq \mathcal{N} - \mathcal{B}$. This modification keeps the algorithm

from exploring nodes which should not affect the outcome of the boundary-scan test.

6.4.1 Impact on Run-time of Algorithm

In terms of test generation, there is a slight impact on run-time, stemming from the fact that each element of a subgroup must be scanned for membership in $\mathcal{B}$. However, this is only $O(n)$, and because there is already a $O(n)$ term in the standard recurrence, the overall asymptotic situation remains bounded below by $\mathcal{T}(b) = \Omega(b \lg b)$.

6.5 Utilizing Logic Properties to Improve Test Validity

In the previous section, the algorithm was modified to allow for the situation where a node equipped with boundary-scan is dominated by a node without. This section describes a further analysis which uses properties of the electronic logic to target this situation.

In some systems, certain values are not driven out to nodes. Instead, resistors or other such devices are used to "pull" the values to a value when no driving is taking place. For example, components utilizing open-collector transistor-transistor logic (TTL) drive low '0' values out into the circuit, yet rely on resistors connected to supply voltages in order to pull the values high. Similarly, other component technologies drive the high '1' values, and have a passive low.

If two such TTL nodes are shorted together, they effectively create an AND gate – if either node is being driven with a '0' voltage, a '0' appears on the node. Otherwise, if both nodes are passive, a high or logic '1' will appear as the nodal value. If we can take into account this sort of logic, then the algorithm will be far more robust and accurate.

Note that this only comes into play when examining a node with boundary-scan and a node without. Two nodes that contain the boundary-scan technology will note a discrepancy on at least one of the two nodes, whereas the case where one of the nodes lacks the capability may result in the untested node bearing all the errors.

In order to make this modification, let us first note a feature of the "splitting" function of the algorithm. The split is such that members of a subset after a split are potentially indistinguishable from one another, given the series of tests made up to that point. Thus, in modifying the algorithm, unscanned nodes should remain in a set if they will be indistinguishable from the boundary-scan nodes within it.

For example, in splitting a TTL-logic set of nodes, the boundary-scan nodes are sifted into high's and low's as usual. "$\times$" (or "don't care") values on the nodes are placed into both sets, as these can be either '1' or '0' depending on the current state of the circuit. Unscanned nodes are placed

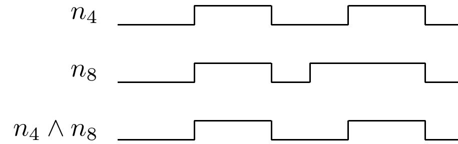
into the groups as follows:

- Unscanned nodes with a '1' are placed into both the high and low sets, as these will appear as either a '1' or a '0' dependent on the value of the nodes they could be potentially shorted with.
- Unscanned nodes with a '0' are placed only into the '0' set, as these will be indistinguishable only if the nodes they are being shorted with contain a '0' also.

Again, the algorithm stops when all nodes in a set are not equipped with boundary-scan.

Note that it is possible for a short between two nodes, one with boundary-scan and one without, to be undetected if this is used. A short will only be detected if, at some point in the test, the boundary-scan node contains a '1' and the non-boundary-scan node a '0',

For example, assume that n_4 as used in the example data set shown in section 6.1.2 above is boundary-scan capable and n_8 is not (cf. Table 6.1). If we short together the two, we get a result that is exactly equal to n_4:



Obviously, shorting together n_4 and n_8 is indistinguishable from the original n_4, thus creating an aliasing syndrome.

6.5.1 Impact on Run-time of Algorithm

The analysis for this modification is more difficult than the others, due to the lopsided divisions that the algorithm produces. However, we can attempt to create a very rough estimate on the running time of the generation algorithm.

Recall that b is the number of nodes being analyzed which contain boundary-scan capability. Assume also, and this is the point where rough estimation must take place, that the proportion of boundary-scan nodes to all nodes (b/n) is constant over each group to be analyzed.

Now, note that the recurrence equation can be given by

$$\begin{aligned}\mathcal{T}(n) &= \mathcal{T}\left(\frac{b}{2}+\frac{n-b}{2}\right)+\mathcal{T}\left(\frac{b}{2}+n-b\right)+O(n) \\ &= \mathcal{T}\left(\frac{n}{2}\right)+\mathcal{T}\left(n-\frac{b}{2}\right)+O(n)\end{aligned}$$

If we now search for the dominant term, we can see that, because $b \leq n$, $n - b/2 \geq n/2$ for all b. Therefore, let us take b to be a fairly low in comparison with n, and assume that processing the two halves will be twice the longer amount of time (thereby making this a worst-case scenario)

$$\mathcal{T}(n) = 2\mathcal{T}\left(\frac{2n-b}{2}\right) + O(n)$$

resulting in a work time of $\mathcal{T}(n) = O\left(n^{log_x 2}\right)$ (where $x = 2/(2n-b)$), which, for b a little less than, say, 20%, will result in a value of $O(n^7)$. The exponent of the n term increases substantially as b decreases.

Obviously, the hope is that the above assumptions are incredibly pessimistic.

6.6 Handling Bussed-Node Situations

There are occasions where the above utilization of the circuit logic will fail to work. For example, bussed nodes often drive both high and low values out, rather than use passive resistance to pull the nodal values high. These nodes often use tri-state drivers which enter a high-impedance state when not specifially driving values out to the node. These are fairly prevalent in microcomputer boards, which use the tri-state drivers as common buses.

Some of the preceding work in our analysis is still valid for this case. For example, if no drivers to a bussed node are active, the high impedance states can be considered an '×' state for all practical purposes. This analysis works for all shorts involving bussed or tri-state nodes in general.

However, if we consider the possibility of opens involving tri-state nodes, the situation becomes far more complex. For example, consider the situation shown in Figure 6-1.

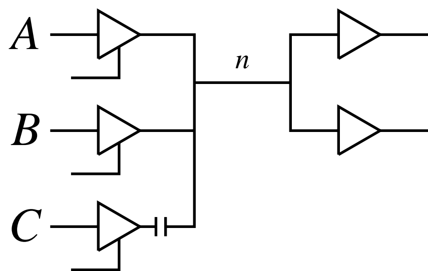


Figure 6-1: Example of Open Fault on Bussed Node

In the figure, the pins at A, B, and C are all tri-stated buffers connected to node n, which also

has a pair of output pins connected to it. In order for any analysis of this situation to be done at all, we need n to be boundary-scan capable.

In order to detect the open fault at node C, we need to know when C is driving both a high and a low value, and make sure that it is actively switching between the two.

For example, if we set the pins to drive the following sequence of values ('Z' represents the high-impedance state):

A	`ZZZZ`
B	`1Z0Z`
C	`Z1Z0`

There is absolutely no way for us to be sure that there is not an open at C in this case, since we are never actively driving a new value onto node n. If the open at C exists, then the node will tend to sustain its previous value when A and B enter their high-impedance state. The only way to be sure that an open does not exist is to actively find test steps where an active switch of the output occurs on a boundary-scan capable bus node, for each driving input of the circuit.

Changing the algorithm to implement this is actually not that hard – before we start splitting our initial set of nodes, we run through the boundary-scan capable tri-stated bus nodes, searching for all test steps in which an active switch of the output results from each of the nodes' drivers. This initial test creation takes roughly $O(tu)$, where u is the number of bussed nodes. Given this inital test set, all future splits will attempt to use the steps in this set as much as possible, hopefully minimizing the number of necessary test steps.

Note that the price we pay for this is twofold – some situations will still be impossible to test (namely, those such as the above example in which a driving node provides no active switching), and the number of test steps necessary for a single bussed node can get large, especially if bus nodes with many drivers exist.

Chapter 7

Fault Dictionary Analysis

As described in the last section of the preceding chapter, the board-level fault analysis does not work for all possibilities of fault occurrences. In order for the ATE to be able to detect situations such as those seen in Figure 6-1, the testing algorithm must have far more knowledge of the circuit. This can be addressed through an analysis of the fault dictionary for the test.

7.1 Fault Dictionary Background

Fault dictionaries (or FDs) were created for use in functional testing, to ease diagnostic testing for faulty boards. Basically, the idea is to simulate running the test program on the circuit many times, using a "fault simulator" to create every possible fault that could occur. As each fault is created, the response resulting at the outputs is recorded and stored. Usually, rather than store the complete system response for a given fault f (R_f), the result of some function $S(R_f)$ is stored to conserve on space. The result of this function is called the "signature" of the fault.

After each possible fault has been simulated, the ATE can run the test on a real board. If the output is incorrect, the ATE compares the signatures in the fault dictionary with the actual faulty response, and locates a small subset of errors which could have caused the signature. These errors can then be tested utilizing a guided probe or an in-circuit algorithm, if such options exist. Otherwise, the list of potential faults can be reported back to the test technician.

The power of the fault dictionary is that, if both the fault simulator and circuit simulator are robust enough, it can effectively simulate and catalogue the signature of every possible fault on the board. There are cases where the fault dictionary method will also fail; for example, when multiple faults exist, causing very unexpected behavior at the outputs. However, we can often assume that manufacturing methods are reliable enough that multiple faults are a rarity. Other

potential problems with fault dictionary processing occur when the juxtaposition of PCB components cause capacitance effects. Such effects cannot often be adequately predicted through current circuit simulation methods.

7.1.1 Fault Simulation

The topic of fault simulation has already been thoroughly studied. A fairly thorough overview of current algorithms can be found in [1].

The most straightforward idea for fault simulation is to generate one fault at a time, running the fault simulator on our test program using the faulty circuit for each fault. If we assume that circuit simulation takes roughly $O(nt)$ each time the simulation is run, we find that a naive approach will take at most $O(nt|\mathcal{F}|)$.

However, we can certainly do much better than this. [1, pp. 146-155] describes several algorithms providing improved efficiency of the system. These improvements revolve around the fact that simulating a faulty circuit is the same as simulating a working circuit, with the fault only contributing small, localized changes in behavior. Using this fact, we realize that each node of the circuit is only affected by a very small number of faults, and thus running a fault simulation has a much smaller order of growth. Indeed, it is sometimes possible to make the entire simulation in just one pass, causing the order of growth to approach $O(nt)$. It has been found that such an algorithm can indeed run a full fault simulation for a "typical" PCB and test program in an appreciably short time.

7.2 Augmenting the FD with Boundary-Scan

In order to use the information in the fault dictionary to create a reasonably efficient boundary-scan testing algorithm, we take the output of the fault simulations (the set of fault signatures and their associated faults) and compare it to the output of the circuit under test. By matching the responses, we can find a set of potential faults.

Then, in order to pinpoint which fault is causing the specific problem, there needs to be a way to take this set of potential faults and select the "correct" fault or set of faults, using further boundary-scan tests if necessary. In order to do this, we can examine the output data and define a series of tests which will localize the faults.

To illustrate, consider the set of all possible faults, which we shall call $\mathcal{F}$. By running the fault simulator on each of these faults, we will create an output failure waveform for each of these faults. Note that some of these failure waveforms will be the same for different faults (that is, different faults will generate the same failure pattern at the outputs of the board). Hence, our set of faults $\mathcal{F}$

can be divided into a group of s disjoint equivalent fault sets $\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_s$. Each of the faults in a given fault set $\mathcal{F}_i$ will yield the same failure pattern when simulated on the circuit.

Dividing $\mathcal{F}$ into the equivalent sets will take some time, especially if all the values in the output were used to divide the set. However, as mentioned previously, a fault signature is used to cut down on the amount of space taken up by each failing response, and therefore it also will cut down on the time taken to divide up $\mathcal{F}$. One common method is to use the information for the first failing test step (FFTS) as the signature function, recording only when and where an error first occurred at the outputs to the circuit. Although using the FFTS or something similar will increase the size of each equivalent fault set, it will cut the time for the initial set division down to $O(|\mathcal{F}|)$ (assuming that set creation and insertion operations are optimized to be $O(1)$).

To this point, we have done no more work than is often done when generating a functional test for a given circuit board. Indeed, certain software systems for ATE already perform this work as a user-selectable test. The information that can be extracted from this functional test can be used to locate a "starting point" for a series of guided probe tests. Some more processing is needed to apply this to boundary-scan, however.

7.2.1 Pinpointing Faults with Boundary-Scan

For a boundary-scan test, the object is to find a series of boundary-scan tests that will further "pin down" which fault in an equivalent fault set $\mathcal{F}_i$ has occurred. Some of the equivalent fault sets will be singletons ($|\mathcal{F}_i| = 1$), and therefore no further processing needs to be done when a failure pattern for one of these fault sets appears. In this case, the single matching fault should just be given back to the user. However, if the matching equivalent fault set has more than one fault in it, the situation is more complicated.

For these, it is necessary to find test steps and boundary-scan accessible locations (nodes) in the circuit where two members of an equivalent fault set (that is, two different faults which would cause the same failure pattern at the outputs) differ, and then find a way to quickly pinpoint which fault in the set occurred. One of the most straightforward ways to implement this is by creating a node-labelled and edge-labelled tree structure, which would effectively provide a fast method of reaching the correct fault by performing only $O(\lg |\mathcal{F}_i|)$ tests. A binary tree structure also has the virtue of being easy to implement and store.

The main problem is then reduced to retrieving the differing test steps and locations. However, our fault simulation software can solve this problem for us, also. Nearly all general fault simulation techniques have the ability to run several different faults in parallel during one single pass through the circuit. Thus, the answer to finding differing test steps becomes simple: for each equivalent set,

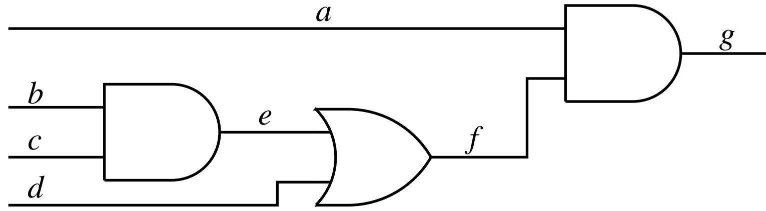


Figure 7-1: Test Circuit for FD Analysis

run a variant of the fault simulation software, but simulating only the members of the fault set. This variant should be modified so that it creates the binary tree structure "on the fly" as it discovers that two faults differ at a given node and test step.

Assuming that we can insert into the tree structure in $O(\lg |F_i|)$ time (see [3] for structures designed to facilitate this), the time to create a tree for each equivalent fault set will be $O(nt + |F_i| \lg |F_i|)$ (the time to perform a pass through the fault simulator, with some additional added time to create the tree). Clearly, the $O(nt)$ term dominates, and thus, creating the entire "forest" for all of the fault sets will be only $O(nt)$ times the number of fault sets, or $O(nts)$.

By way of example, take the circuit shown in 7-1. Although simplistic, it shows off the concept behind the fault dictionary analysis techniques mentioned above. Let us provide the following set of inputs to the circuit (also listed are the resulting values on e, f and g.

a	111
b	010
c	110
d	100
e	010
f	110
g	110

Now, we run the fault simulator on our circuit and attempt to simulate all possible faults that can occur. To simplify matters, we will only simulate the faults where one of the nodes gets stuck at either 0 or 1. When the fault simulator is run and each fault categorized by the FFTS, it will divide the set of faults $\{a_0, a_1, b_0, \ldots, g_1\}$ into four sets: Those faults noticed during the first test step ($\{a_0, d_0, f_0, g_0\}$), those noticed during the second step ($\{b_0, c_0, e_0\}$), those noticed during the third step ($\{d_1, e_1, f_1, g_1\}$), and, ultimately, those that are never noticed at all ($\{a_1, b_1, c_1\}$). Note that this test is probably not very robust, considering that it only has a 78% fault coverage. In actual situations, the test would need to have more test steps.

We now run the fault simulator on each of these fault sets and record boundary-scan nodes where we could detect the errors. As this occurs, a binary tree is created. In the above, assume

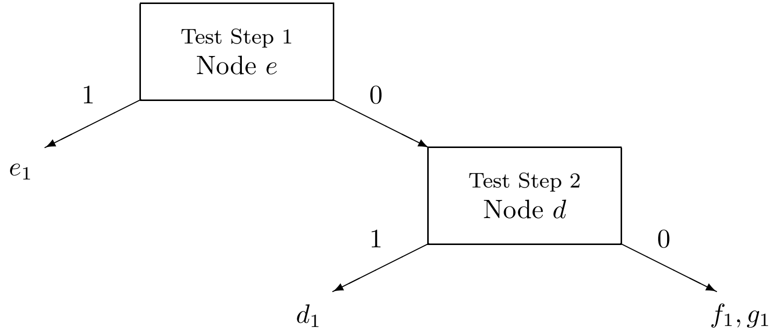


Figure 7-2: Example of Generated Tree Structure

that the or-gate is located on a boundary-scan component, and therefore nodes d, e, and f are all boundary-scan capable. If we run the fault simulator for the third set $\{d_1, e_1, f_1, g_1\}$, we will create the tree shown in Figure 7-2.

Now, in performing the test at runtime, the following operations occur: First, the entire test is run once, in order to locate which generated tree to use. This is done by matching the fault signature (the FFTS) of the output to the signatures of the equivalent fault sets. Then, the test is run several times, each time recording the information for a different test step (as specified by the tree structure). For example, if we discover that the FFTS is the third, we would go to the tree in Figure 7-2, and start by running the test and recording the boundary-scan information on test step 1. If node e is a high value, the system will report that e is stuck high. Otherwise, the system would go on to test whether d is high or low on test step 2. If it finds that d is low, then it will report that it cannot tell any difference between f stuck high or g stuck high.

7.3 Pseudocode Description of Algorithm

Pseudocode for this algorithm is difficult, considering that it is heavily dependent on integration with existing fault simulation code. However, the only major alteration to this code is the creation of balanced binary trees. This is fairly well-studied, however, and good references with pseudocode can be found in [3, pp.263-271]. Note that the entire tree creation should, as mentioned before, add only an $O(|\mathcal{F}_i| \lg |\mathcal{F}_i|)$ for each fault set.

The pseudocode for the run-time test, however, is extremely straightforward once these balanced trees are created:

```
RUNTIME-FAULT-TEST(T)
    ▷ given the tree for the appropriate fault set
```

```
1   if T is a fault or list of faults
2       return T
3   run boundary-scan test on test step test[T]
4   if value on node[T] = val[left[T]] do
5       Runtime-Fault-Test(left[T])
6   else
7       Runtime-Fault-Test(right[T])
```

7.4 Analysis of Algorithm

Like the board-level fault analysis algorithm, analysis using fault dictionaries has a precomputation process to run as well as the run-time processing liabilities. We shall deal with each in turn.

We have not calculated the time it takes for the fault simulator to operate. We can be fairly certain, however, that the fault simulator will be involved in the brunt of the precomputation time, since the other calculations can work in parallel with the simulator. As mentioned before, this is given roughly as $O(nts)$, where s is the number of equivalent fault sets being created.

During the runtime testing of the PCB, the test must be run once in order to discover which equivalent fault set the failing PCB belongs to. Discovering which set can, using just the FFTS, take just $O(t)$. After that, the number of extra tests is strictly dependent on the size and configuration of the tree structure generated for that fault set. Given a balanced tree as the "average" case, the number of extra tests will be $O(\lg |\mathcal{F}_i|)$. Each of these tests will run in the time $O(b+t)$ (see Chapter 4), so the overall runtime of the algorithm is $O((b+t)\lg |\mathcal{F}_i|)$.

Note that, in addition to the usual computation overhead, this algorithm requires a substantial amount of space to be stored for the test. When you store the information for each test, the amount of space to store for each tree is $O(|\mathcal{F}_i|)$ for each equivalent fault set, or $O(|\mathcal{F}|)$ overall for the entire forest created by all the fault sets.

Chapter 8

Comparison of Algorithms

In deciding which of the previous algorithms is "best", we need to use the following points of comparison:

- The algorithm's ability to detect that a fault has occurred.
- The algorithm's ability to pinpoint the location of the fault.
- The efficiency of the algorithm (taking into consideration both precomputation time as well as test runtime).

The three algorithms have already been analyzed for their efficiency, each in its own section. These results have been tabulated and presented in Table 8.1. Note that, because of the wildly disparate methods of computation, it is not at all easy to compare these values directly. How, for example, does the number of boundary-scan nodes b compare with the number of faults in a typical equivalent fault set $\mathcal{F}_i$? Although we can assume that the brute-force algorithm is least efficient in terms of runtime, it does not pay a hefty price in precomputation as do the other two methods. One note about test generation: in nearly all cases, it is better to minimize the time that it takes to run the test, rather than worry about the time for test preparation. This is because the same test will often be run on the same PCB multiple times, whereas the precomputation only needs to

Algorithm	Precomputation	Runtime
Brute-Force	none	$O((b+t)n/b)$
Board-Level	$O(n^{\log_x 2})$	$O((b+t)\lg b)$
Fault-Dictionary	$O(nts)$	$O((b+t)\lg \lvert\mathcal{F}_i\rvert)$

Table 8.1: Summary of Algorithm Efficiency

occur once. Thus, the exponential growth of the board-level fault analysis may tend to indicate an additional value in the fault-dictionary approach.

In comparing the ability of each test to identify an error that has occurred, we quickly find that the different tests vary in their thoroughness. For example, the brute-force is extremely good, in that any error that will propagate to a boundary-scan node will inevitably be detected and the error flagged. As we have seen in the opening paragraphs of Chapter 7, the board-level analysis may be incapable of fully testing a bussed-node situation, thus making it a perhaps less than optimal solution. Finally, the fault dictionary algorithm will be able to detect any single fault on the board, provided that the fault simulator is robust and correct enough.

In terms of ability to pinpoint an error, the fault dictionary algorithm is perhaps the best (again dependent on whether the fault simulator is complete). Roughly equal are the brute-force and board-level analysis algorithms, since both algorithms will rarely be able to give any more information beyond the boundary-scan node and the test step where the error occurred.

8.1 Recommendations

> Simplify, simplify.
>
> — Henry David Thoreau, *Walden*

> Why didn't he just say one 'simplify'?
>
> — Nicholas Colasanto, *Cheers*

In the final analysis, the choice of algorithm is highly dependent on the accuracy that the user/implementor desires. If there is a real need for pinpointing the exact error in the board without resorting to guided probe (and if there is a well-defined fault simulator), the fault dictionary is perhaps the best, despite the potentially heavy emphasis on pre-test computation. In addition, a need for speed would also seem to indicate the fault-dictionary approach, assuming that the typical size of a fault set $\mathcal{F}_i$ is a good deal lower than the number of boundary-scan nodes b.

However, for most applications, in which it is all right to merely get a rough pass at the fault location (due to guided probe access to other components), the best test to perform is probably that of the far simpler brute-force algorithm. Because this algorithm is guaranteed to detect 100% of the errors that can appear on the boundary-scan nodes, it is probably the most robust of all the algorithms (no such guarantee is given for fault-dictionary testing, due to the inherent complexity of the fault simulation program).

Appendix A

C^{++} Listings for Board-Level Fault Analysis

The main code for the algorithm is shown below. Values for n (the number of nodes), *bscan* (the number of nodes which contain boundary-scan), t (the number of test steps), and *data* (the array of all node values) are created within `gen_data.C`, which also handles the processing for data creation. Common declarations are found in `tester.H`.

```
1   #include "tester.H"
    #include <sys/types.h>
    #include <sys/time.h>

5   intset *nodeset, *non_bscan, *testset;

    /**********************
    Routines to create test
    **********************/
10  // *now can handle non-BS nodes*
    // note: assumes TTL logic.
    // divideset takes two arguments -- the set being divided, and the
    // list of nodes not yet tested.
    void divideset(intset *set, intset *untested)
15  {
      int el;
```

```
     if (set->size<2) return;
     // check: if no nodes have B/S capability, exit routine
20   int flag = 1;
     set->iterate(el);
     while (set->ok(el) && flag == 1)
       flag = (non_bscan->member(set->next(el)));
     if (flag) return;
25   if (untested->size == 0) {
   #ifndef MMA
       cerr << "warning: nodes " << *set << " are similar; cannot distinguish\n";
   #endif
       return;
30   }

     // get next test step to search
     int test;
     flag = 0;
35   if (testset->size > 0) {
       testset->iterate(el);
       // next line loops through testset until it finds an untested node
       while(testset->ok(el) && flag==0) {
         test = testset->next(el);
40       if (untested->member(test))
           flag = 1;
       }
     }

45   if (flag==0)
       test = untested->random(); // if not found, set to a random element.

     // divide set and process each half using current test step
     // note that nodes that are not BS are placed in both groups if they
50   //   have a "1", in the lo groups if they have a "0"
     intset hi_set(n), lo_set(n);
     set->iterate(el);
```

```
      while(set->ok(el)) {
55      int elt = set->next(el);
        if (Data(elt,test) != lo)
          hi_set.insert(elt);
        if (Data(elt,test) != hi ||
            (non_bscan->member(elt) && !bus_nodes->member(elt)))
60        lo_set.insert(elt);
      }

    //  cout << "dividing " << *set << " on " << test << " --> " << hi_set << lo_set << endl;

65    // test to decide whether to add current test to test set.
      if (!testset->member(test) &&   // if already a member, don't bother looking.
          hi_set.size != set->size &&
          lo_set.size != set->size) {  // make sure something split.

70      int check = 0;

        hi_set.iterate(el);
        while(hi_set.ok(el) && check == 0)
          if (Data(hi_set.next(el),test) != no_care) check = 1;
75      if (check) {
          check = 0;
          lo_set.iterate(el);
          while(lo_set.ok(el) && check == 0)
            if (Data(lo_set.next(el),test) != no_care) check = 1;
80
          if (check) testset->insert(test);
        }
      }

85    //remove test from untested list.
      untested->remove(test);

      // make a copy of untested for second call
      intset untested2(t);
```

```
90    untested->iterate(el);
      while(untested->ok(el))
        untested2.insert(untested->next(el));

      //divide and check the halves.
95    if (hi_set.size>1) divideset(&hi_set, untested);
      if (lo_set.size>1 &&
          !(lo_set==hi_set)) divideset(&lo_set, &untested2);
    }

100 void MakeShortsTest()
    {
      // This routine operates by dividing the set in two and operating on
      // each half.  This is accomplished in divideset.

105   // make set of nodes
      nodeset = new intset(n);
      testset = new intset(t);

      for(int i=0;i<n;i++)
110     nodeset->insert(i);

      // make list of untested test steps
      intset untested(t);
      for(int j=0;j<t;j++)
115     untested.insert(j);

      divideset(nodeset,&untested);

      // at this point, testset has the list of tests.
120 }
    inline void FindNonBS()
    {
      non_bscan = new intset(n);

125   for(int i=(bscan_pct*n)/100;i<n;i++)
```

```
        non_bscan->insert(i);
    }



130 main(int argc, char *argv[])
    {
      time_t start,finish;

      // create the list of non-boundary-scan nodes
135   CreateData(argc,argv);
      FindNonBS();
    #ifndef MMA
      PrintDataSet();
      cout << "\nMaking Test...\n";
140   cout.flush();
    #endif
      start = time(NULL);
      MakeShortsTest();
      finish = time(NULL);
145
    #ifdef MMA
      cout << form("{%d,%d,%d,%d,%d}",n,t,bscan_pct,testset->size,finish-start);
    #else
      cout << "Test: (n,t,b) (" << n <<","<< t <<","<< bscan_pct << "%)\n";
150   cout << *testset << ", size " << testset->size <<  endl;
      cout << form("Time to generate test is %d seconds\n",
                   (int) (finish - start));
    #endif
    }
```

```
  1 #include "tester.H"
    #include "math.h"
    #include "rand48.h"
    #include "malloc.h"
  5
    int n, bscan_pct, t, bus_pct;
    int *data;
```

```
    intset *bus_nodes;

10  void randomize()
        // randomize the data values for testing
        // don't cares appear with variable probability which drops off quickly
        // as the test continues...
    {
15    bus_nodes = new intset(n);
      // loop through nodes
      for (int i = 0; i<n; i++) {
        // set number of X's from 0...5
        int x = (int) lrand48()%05;
20      int j=0;
        while(j<x) Data(i,j++) = no_care;
        // set length of trends to low, med, or hi
        double threshold;
        x = (int) lrand48()%04;
25      if (x==0) threshold = 0.7;
        else threshold = 0.05;
        x= (int) lrand48()&01;
        for (;j<t;j++) {
          if (drand48() < threshold)
30          x= (x==hi?lo:hi);
          Data(i,j) = x;
        }
        // add to bus_node list
        if (lrand48()%100 < bus_pct)
35        bus_nodes->insert(i);
      }
    }

    void PrintDatum(int i,int j)
40  {
      switch (Data(i,j)) {
      case no_care:
        cout << "X"; break;
```

```
      case lo:
45      cout << "0"; break;
      case hi:
        cout << "1"; break;
      }
    }
50
    void PrintLine(int j)
    {
      cout << " Test " << dec(j,3) <<": ";
      for(int i = 0;i<n;i++) {
55      PrintDatum(i,j);
        if (i%5 == 4) cout << " ";
      }
      cout << "\n";
    }
60
    void PrintDataSet()
    {
      for(int j=0; j<t; j++)
        PrintLine(j);
65    cout << "   B-scan: ";
      for(int i=0; i<n; i++) {
        if (i<(bscan_pct*n)/100) cout << "*";
        else cout << " ";
        if (i%5 == 4) cout << " ";
70    }
      cout << "\nBus Nodes: ";
      for (i=0; i<n; i++) {
        if (bus_nodes->member(i)) cout << "*";
        else cout << " ";
75      if (i%5 == 4) cout << " ";
      }
      cout << endl;
    }
```

```
80   void CreateData(int argc, char *argv[])
     {
       if (argc < 5)
         {
           cout << "Enter the following parameters:\n\tNumber of nodes: ";
85         cin >> n;
           cout << "\tNumber of test steps: ";
           cin >> t;
           cout << "\tPercentage of boundary-scan capable nodes: ";
           cin >> bscan_pct;
90         cout << "\tPercentage of bussed nodes: ";
           cin >> bus_pct;
         } else {
           n = atoi(argv[1]);
           t = atoi(argv[2]);
95         bscan_pct = atoi(argv[3]);
           bus_pct = atoi(argv[4]);
         }

       data = new int[n*t];
100    randomize();
     }
```

```
 1   \begin{verbatim}
     /* tester.H  -- generic stuff for tester routines... */

     #include "objects.H"
 5   #include "stream.h"

     extern int n, bscan, t;
     enum { lo = 0, hi, no_care};

10   extern int *data;
     inline int& Data(int x, int y)
     {
       return (data[x*t+y]);
     }
```

15

```
void CreateData(int argc, char *argv[]);


void PrintDataSet();
\end{verbatim}
```

20 \newpage

Appendix B

The IntSet Class

B.1 Description of IntSet Class

The IntSet class is for the purpose of defining integer sets, and has the following functions and variables defined: ($\mathcal{S}$ is the set for which the operation is being applied.)

intset(int $maxsize$**)**

Creation function for integer sets, where $maxsize$ is the maximum number of elements that the set can contain. The current version also treats this as the maximum value within the intset, so there can be at most $maxsize$ elements, in the range $(0 \ldots maxsize)$. If m is less than 1, an "illegal size" error is returned and the program exits. This procedure runs in time $O(1)$.

~intset()

Automatic destruction function for integer sets. This function is automatically called by the C^{++} libraries when the integer set goes out of scope. (see C^{++} manual for more details.) This procedure runs in time $O(1)$.

int size

Integer variable describing the current size of the integer set $\mathcal{S}$. If $\mathcal{S} = \emptyset$, $\mathcal{S}$`.size` is equal to zero, otherwise $\mathcal{S}$`.size` $= |\mathcal{S}|$. As this is just a variable, it can be accessed in time $O(1)$.

int member(int t**)**

Tests whether or not the integer t is a member of the integer set $\mathcal{S}$. Calling $\mathcal{S}$`.member(x)` returns 1 if `x` $\in \mathcal{S}$, returns zero otherwise. This is time-critical, so it must run in time $O(1)$.

void insert(int t)

Inserts the member t into the integer set $\mathcal{S}$ by performing the function $\mathcal{S} \leftarrow \mathcal{S} \cup \{t\}$. $\mathcal{S}$`.insert(x)` will add the integer `x` into the set, except when a) `x` already exists; or b) $\mathcal{S}$`.size` is equal to $maxsize$, the value given during object creation. If the second condition is true, a "too many elements" error is returned and the program exits. This is time-critical, so it must run in time $O(1)$.

void remove(int t)

Removes the element t from the integer set $\mathcal{S}$by performing the function $\mathcal{S} \leftarrow \mathcal{S} - \{t\}$. $\mathcal{S}$`.remove(x)` searches for `x` in the set, and deletes it from the set if it is able. If `x` $\notin \mathcal{S}$, then an "element not found" error is returned and the program exits. This must also run in time $O(1)$.

int random()

Function to select a random element from $\mathcal{S}$. $\mathcal{S}$`.random()` will select a random number from 1 to $\mathcal{S}$`.size`, and returns the element in that position within the set. If $\mathcal{S} = \emptyset$, then the function exits with an "empty set" error. This function currently runs in time $O(maxsize)$.

void iterate(int& i)

Initializes external counter i as preparation for an iteration routine. The function $\mathcal{S}$`.iterate` works in conjunction with $\mathcal{S}$`.ok` and $\mathcal{S}$`.next` to create a routine which iterates over the entire set $\mathcal{S}$. These three procedures are designed such that a typical iteration will run in time bounded by $O(maxsize)$.

int ok(int& i)

Returns a '1' if the external counter i has not yet reached the last value in the set. If a '1' is returned, a $\mathcal{S}$`.next` function call can safely take place. Using $\mathcal{S}$`.next` without checking first $\mathcal{S}$`.ok` could result in spurious results.

int next(int& i)

Returns the value of the set specified by the external counter i. As a side effect, it increments the counter. If i is set by the calling program, rather than through the use of $\mathcal{S}$`.iterate` and $\mathcal{S}$`.next`, then the results may be undefined.

Example use of `iterate`, `ok`, and `next`:

```
/* loop through intset s and add together elements */
int sum = 0;
int i;
```

```
s.iterate(i);   // initialize counter
while(s.ok(i))  // loop until done
  sum += s.next(i);  // add elements until done.

cout << sum;    // print out sum
```

ostream& operator<<(ostream& s, intset& x)

Adds a text representation of the integer set x to the output stream s. The text representation consists of the open brace {, the text representation of each member of s, separated by commas, and the closing brace }. This operates in time $O(maxsize)$.

B.2 Implementation of IntSet Class

B.2.1 Declaration – objects.H

```
1   /*
     *  Header -- Standard classes for algorithms
     */

5   #include<stream.h>

    //doubly linked with pointers to elements
    class dlink
    {
10  friend class dlist;
      int e;
      dlink *next;
      dlink *prev;
      dlink (int a, dlink* p, dlink* n) {e=a; next = n; prev = p;}
15  };

    class dlist
    {
      dlink *head;
20  public:
      dlink *search(int a);
```

```
    void insert(int a);
    void remove(int a);
    void walk(void func(int));
25  void walkandkill(int func(int));
    void clear();
    int empty() {return(head == 0);}
    dlist() { head = 0; }
    dlist(int a) { head = new dlink(a,0,0); }
30  ~dlist() { clear(); }
  };

  //class for integer sets (used in testsets)

35  class intset {
    int maxsize;
    int *x;
  public:
    int size;
40  intset(int m);  // at most m ints
    ~intset();
    int member(int t);
    void insert(int t);
    void remove(int t);
45  int random();
    void iterate(int& i);
    int ok(int& i) {return i<maxsize;}
    int next(int& i);
    intset& operator=(intset& b);
50
    friend ostream& operator<<(ostream&, intset&);
    friend int operator==(intset&, intset&);
    friend intset& operator+(intset&, intset&);
    friend intset& operator+=(intset&, intset&);
55  };

  ostream& operator<<(ostream& s, intset& x);
```

```
    int operator==(intset& a, intset& b);
    intset& operator+(intset& a, intset& b);
60  intset& operator+=(intset& a, intset& b);
```

B.2.2 Implementation – objects.C

```
 1  #include "objects.H"
    #include "stream.h"
    #include "rand48.h"

 5  dlink *dlist::search(int a)
    {
      dlink *x = head;
      while(x!=0) {
        if (x->e == a)
10        break;
        x = x->next;
      }
      return x;
    }
15
    void dlist::insert(int a)
    {
      dlink *x = new dlink(a,0,head);
      if (head != 0)
20      head->prev = x;
      head = x;
    }

    void dlist::remove(int a)
25  {
      dlink *x = search(a);
      if (x) {
        if (x->prev != 0)
          x->prev->next = x->next;
30      else head = x->next;
```

```
      if (x->next != 0)
        x->next->prev = x->prev;
      delete x;
    }
35  }

    void dlist::walk(void func(int))
    {
      dlink *x = head;
40    while (x) {
        func(x->e);
        x = x->next;
      }
    }
45
    void dlist::walkandkill(int func(int))
        // walks through list.  If func(element) is 1, then it deletes the elt.
    {
      dlink *x = head;
50    while(x) {
        dlink *y = x;
        x = y->next;
        if (func(y->e) == 1) {
          if (y->prev != 0) y->prev->next = y->next;
55        else head = y->next;
          if (y->next != 0) y->next->prev = y->prev;
          delete y;
        }
      }
60  }

    void dlist::clear()
    {
      dlink *x = head;
65    while (x) {
        dlink *l = x;
```

```
    x = x->next;
    delete l;
  }
70  head = 0;
}

// definitions for integer sets.
/*************************************************
75  * NOTE: the new intset implementation assumes that *
 * the number provided to the creation function is  *
 * such that only elements from 1...n will be used  *
 * in the integer set, without duplicates.          *
 *************************************************/
80

void error(char *s)
{
  cerr << s << "\n";
  exit(1);
85 }

intset::intset(int m)
{
  if (m<1) error("intset: illegal size");
90  size = 0;
  maxsize = m;
  x = new int[maxsize];
  for(int i = 0; i< maxsize; i++) x[i] = -1;
}
95

intset::~intset()
{
  delete x;
}
100

void intset::insert(int t)
{
```

```
    if (x[t] == -1) size++;
    x[t] = t;
105 }

  int intset::member(int t)
  {
    if (x[t] == -1) return 0;
110 else return 1;
  }

  void intset::remove(int t)
  {
115 if (x[t] == t) size--;
    x[t] = -1;
  }

  void intset::iterate(int& i)
120 {
    i=0;
    while(x[i]==-1) i++;
  }

125 int intset::next(int& i)
  {
    int r = x[i++];
    while(x[i]==-1) i++;
    return r;
130 }

  int intset::random()
      // locates a random element within the set, if possible.
  {
135 if (size == 0) error("intset: empty set");
    int i = (int) lrand48()%maxsize;
    while(x[i] == -1)
      i=(i+1)%maxsize;
```

```
  return x[i];
140 }

intset& intset::operator=(intset& b)
    // creates a copy of the integer set
{
145   if (this == &b) return *this;
  delete x;
  x = new int[maxsize];
  memcpy(x,b.x,(maxsize>b.maxsize?b.maxsize:maxsize));
  return *this;
150 }

intset& operator+=(intset& a, intset& b) // union operator
{
  if (b.maxsize>a.maxsize)
155     a.maxsize = b.maxsize;
  int i;
  b.iterate(i);
  while(b.ok(i))
    a.insert(b.next(i));
160   return a;
}

intset& operator+(intset& a, intset& b) // union operator
{
165   intset *temp;
  if (a.maxsize>b.maxsize)
    temp = new intset(a.maxsize);
  else
    temp = new intset(b.maxsize);
170   *temp = a;
  *temp += b;
  return *temp;
}
```

```
175  ostream& operator<<(ostream& s, intset& x)
     {
       s << "{";
       int found = 0;
       for(int i = 0; i<x.maxsize; i++)
180      if (x.x[i]!=-1) {
           if (found == 0)
             found = 1;
           else s << ",";
           s << x.x[i];
185      }
       return s << "}";
     }

     int operator==(intset& a, intset& b)
190  {
       if (a.maxsize != b.maxsize) return 0;
       for(int i = 0; i < a.maxsize ; i++)
         if (a.x[i] != b.x[i]) {
           return 0;
195      }
       return 1;
     }
```

Bibliography

[1] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

[2] IEEE Standards Board. IEEE Standard Test Access Port and Boundary-Scan Architecture. Standard 1149.1-1990, Institute of Electrical and Electronics Engineers, 345 E.47th St., New York, NY 10017-2394, February 1990.

[3] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Co & MIT Press, 1990.

[4] Edward B. Eichelberger, Eric Lindbloom, John A. Waicukauski, and Thomas W. Williams. *Structured Logic Testing*. Prentice Hall, Edgewood Cliffs, NJ, 1991.

[5] GenRad, Inc. *Meeting the Challenge of Boundary Scan*. GenRad, Inc., Concord, MA 01742, August 1991.

[6] Deborah Graham. *Introduction to Multi-Strategy Testing*. GenRad, Inc., Concord, MA 01742, July 1989.

[7] Najmi Jarwala and Chi W. Yau. A new framework for analyzing test generation and diagnosis algorithms for wiring interconnects. In *IEEE Proceedings 1989 International Test Conference*, pages 63–70, 1989.

[8] Colin M. Maunder and Rodham E. Tulloss. *The Test Access Port and Boundary Scan Architecture*. IEEE Computer Society Press, 1990.

[9] Gordon D. Robinson. Boundary scan impact on board test strategies. In *ELECTRO/90*, May 1990.