Technical Report 1099

Generating Circuit Tests by Exploiting Designed Behavior

Mark Harper Shirley

MIT Artificial Intelligence Laboratory

Generating Circuit Tests by Exploiting Designed Behavior

by

Mark Harper Shirley
S.B., Massachusetts Institute of Technology, 1983
S.M., Massachusetts Institute of Technology, 1983

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

at the

Massachusetts Institute of Technology

December, 1988

Signature of Author _____

Department of Electrical Engineering and Computer Science
December 14, 1988

Certified by _____

Randall Davis
Associate Professor of Management Science
Thesis Supervisor

Accepted by _____

Arthur C. Smith, Chairman
Committee on Graduate Students

Generating Circuit Tests by Exploiting Designed Behavior

Mark Harper Shirley

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science at the
Massachusetts Institute of Technology, September, 1988

Abstract

Generating tests for sequential devices is one of the hardest problems in designing and manufacturing digital circuits. This task is difficult primarily because internal components are accessible only indirectly, forcing a test generator to use the surrounding components collectively as a probe for detecting faults. This in turn forces the test generator to reason about complex interactions between the behaviors of these surrounding components. Current automated solutions are becoming ineffective as designs grow larger and more complex. Yet, despite the complexity, human experts remain remarkably successful, in part, because they use knowledge from many sources and use a variety of reasoning techniques. This thesis exploits several kinds of expert knowledge about circuits and test generation not used by the current algorithms.

First, many test generation problems can be solved efficiently using *operation relations*, a novel representation of circuit behavior that connects internal component operations with directly executable circuit operations. Operation relations can be computed efficiently for sequential circuits that provide few operations at their interfaces by searching traces of simulated circuit behavior.

Second, experts write test programs rather than test vectors because programs are a more readable and compact representation for tests than vectors are. Test programs can be constructed automatically by merging test program fragments using expert-supplied goal-refinement rules and domain-independent planning techniques from AI. Additional leverage arises from giving the test generator knowledge of the capabilities of the tester hardware.

I describe two implemented programs based on these ideas, drawing examples from a simple microprocessor.

Keywords: Artificial Intelligence, Circuit Testing, Test Generation, Knowledge-based Systems, VLSI.

Thesis Supervisor: Randall Davis
Title: Associate Professor of Management Science

Acknowledgments

Many people contributed to this work and to the productive and fun environment that made it possible.

I owe the greatest debt to my advisor Randall Davis, who propelled me over many hurdles during my graduate career and who spent patient hours showing me how to question and how to explain and to Gordon Robinson, whose ideas about testing are the bones supporting the flesh of this thesis and whose unfailing good humor was always a delight. Thank you both. I hope I have learned a small fraction of the lessons you were teaching.

My readers Ramesh Patil and Paul Penfield provided incisive criticism and improved the thesis and its presentation immeasurably.

My friends Raúl Valdés-Pérez, Jeff Van Baalen, Reid Simmons, Howie Shrobe, Paul Resnick, Choon Goh, Walter Hamscher, Hal Haig and Meyer Billmers of the HT reading group shaped my ideas about AI and were a lively forum for discussion.

Special thanks go to Dan Weld, Brian Williams and Peng Wu and for many hours talking about anything and everything and for being great companions.

Jeff Siskind, Jerry Roylance, Jean-Pierre Schott and the rest of the crowd at MIT AI made it a wonderful place to work. Mark Norton, Mark Chilenskas, Mark Swanson, Mike Repeta, Pete Williamson, Wade Williams and Gordon Taylor of GenRad made my work there fun and productive. Yehudah Freundlich opened my mind to the history and philosophy of science. He knows lots of pretty good jokes too. Glenn Kramer, Narinder Singh, Dan Carnese, Marty Tenenbaum and Alex Miczo of Schlumberger supplied many good ideas and criticisms as did Mel Breuer of USC.

Andy Ressler wrote a simple and efficient implementation of Prolog for the lisp machine: Andy, the price was definitely right. Olin Shivers helped make learning about computers in high school fun and was always even more willing to datagrunt than me.

Finally, I would like to thank my family: Grant, Papa-san, Moma-sama, Hugh, Mary, Elizabeth, Ann, Emily, Hugh and especially my wife Lora. I love you all.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology, at GenRad Inc., and at Schlumberger CAS Palo Alto Research. Support for the AI laboratory's research on digital hardware

troubleshooting is provided in part by the Digital Equipment Corporation and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

Contents

Chapter 1
Introduction

Testing is an essential part of designing and manufacturing digital circuits. Without thorough testing, circuits cannot be relied upon to safely control airplanes, elevators and cars or to do any of the myriad tasks they do in our modern world. This thesis is concerned with generating tests to detect physical defects that can cause a circuit to malfunction.

Sequential VLSI circuits are one of the hardest kinds of circuit to generate tests for. They are difficult primarily because internal components are accessible only indirectly, forcing a test generator to use the surrounding components collectively as a probe for detecting faults. This in turn forces the test generator to reason about complex interactions between the behaviors of these surrounding components.

Current automated solutions are becoming ineffective as designs grow larger and more complex. Yet, despite the complexity, human experts remain remarkably successful, in part, because they use knowledge from many sources and use a variety of reasoning techniques. This thesis exploits several kinds of knowledge about circuits and test generation not used by the current algorithms.

This introduction summarizes the central aspects of test generation to show why the problem is difficult, then states the fundamental propositions of this thesis.

1.1 Test Generation is a Complex Planning Problem

At its core test generation is a classical planning problem. The goal is to cause patterns of internal activity that distinguish between a properly manufactured circuit and one that has physical flaws. The primitive actions available to testers are applying and observing voltages at the periphery of the circuit. Planning goals are related

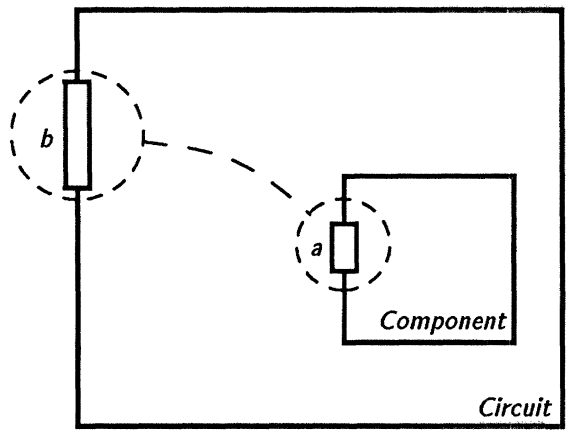


Figure 1.1: *The embedding problem: given a component test expressed in terms of the component interface (a), work out how to execute this test using manipulations of the circuit interface (b). The embedding problem is an instance of conjunctive planning, since it involves achieving multiple, potentially-interacting goals, i.e., the actions specified by the component test.*

to the primitive actions by the circuit schematic and behavioral descriptions of the components.

Tests are traditionally created by partitioning the design into components and generating a test for each component by (i) working out how to test the component as if it were alone – **the component test problem** – and (ii) working out how to execute that component test within the context of the larger circuit – **the embedding problem**. The component test problem has many interesting aspects, but it can always be solved by recursively dividing the component into smaller and smaller components until each can be tested exhaustively. The embedding problem (see figure 1.1) is fundamentally more difficult because it involves achieving a set of multiple, potentially-interacting goals, i.e., the actions specified by the component test. Thus the embedding problem is an instance of conjunctive planning [fikes71, sussman75, sacerdoti77, stefik80, vere83, chapman85].

The embedding problem is typically solved by using descriptions of component behavior to incrementally refine the goal of executing a component test into goals of controlling circuit inputs. Figure 1.2 shows an example of an embedding problem in a 16 function arithmetic logic unit (ALU). One test for component A involves applying 1 to A's upper input and 0 to the lower input, then observing whether the output is

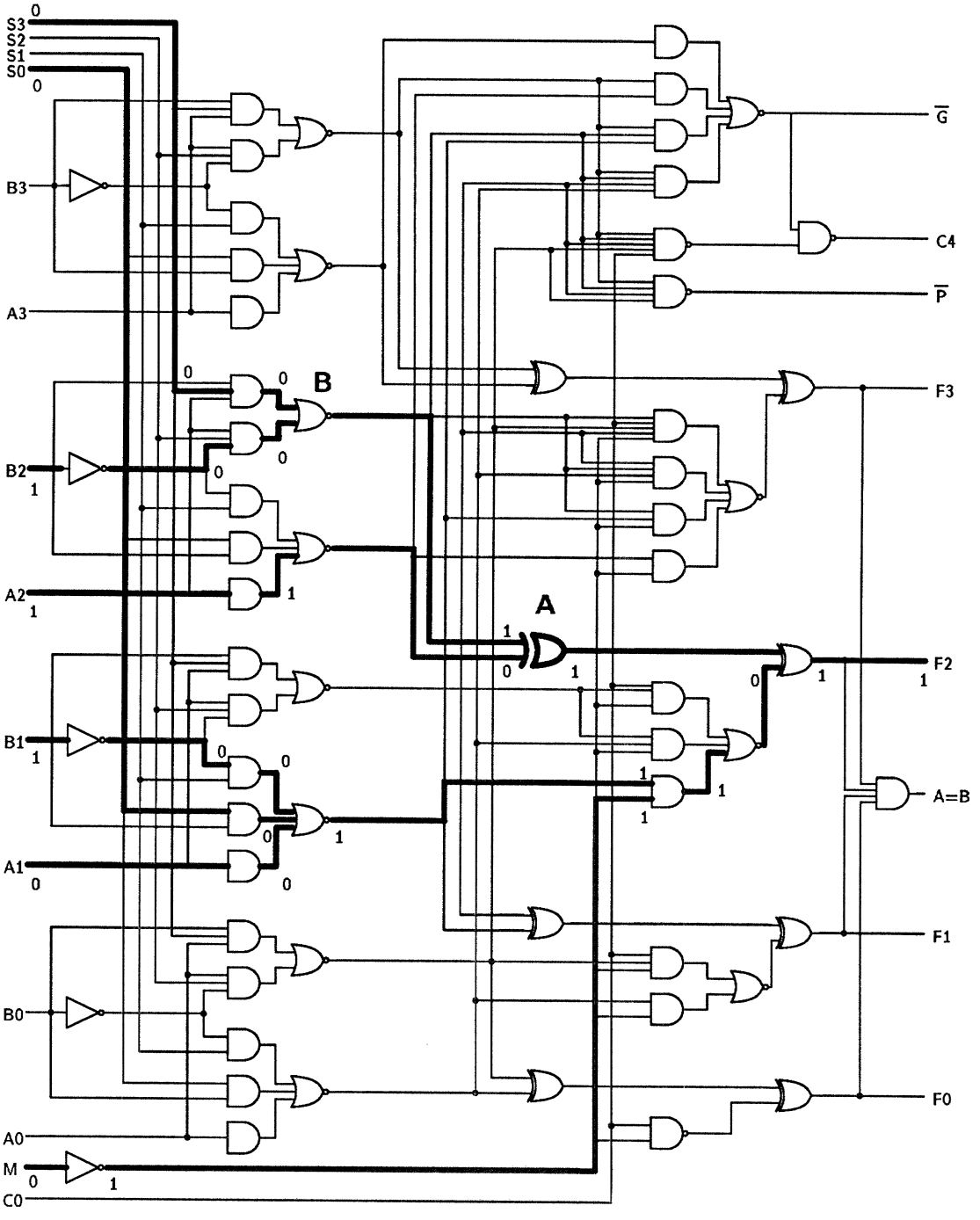


Figure 1.2: *A gate-level Embedding Problem*

a 1. The embedding problem involves working out how to execute this component test by manipulating the circuit inputs (on the left) and observing the circuit outputs (on the right). One solution is shown: the dark lines and the boolean values beside them indicate how controlling some of the inputs (S3, B2 and A2) causes the desired values on A's inputs, and how controlling other inputs (S0, B1, A1 and M) routes A's output to a place where it can be observed (F2). For instance, the goal of applying 1 to A's upper input is solved by applying 0's to B's inputs, and the goal of applying 0 to B's upper input is solved by applying 0 to S3. Since S3 is a circuit input, this last goal can be achieved directly.

Embedding problems are difficult in general because the methods for controlling the inputs and observing the output can interfere with each other via the circuit's many cross connections. Solving this problem for the ALU, although detailed, is within the capabilities of existing test generation algorithms. They are well able to handle this and similar combinational (memory-less) circuits built from thousands of gates. However, modern circuits, e.g., microprocessor-based systems, are several orders of magnitude larger and more complex than this ALU. And this is the root of a difficult problem: many circuits are so complex that the existing algorithms are, at best, of limited use because they take too long to execute. The algorithms get lost in the large search space extending over physical space (complex circuit structure) and time (many clock cycles).

Yet, despite the complexity, human experts can often design high-quality tests. This research is prompted by this performance gap and the differences in method that cause it. Studying the problems that people solve and how they solve them supplies clues for improving the algorithms and closing the gap.

1.2 Statement of the Thesis

Test generation is formally intractable – for combinational circuits the problem is NP-complete [ibarra75] – yet test generation is commercially important and must be solved for complex circuits. Testing practitioners use a collection of heuristics for partitioning the problem and techniques for handling important special cases, rather than a single, all-purpose method. These specialized techniques outperform general techniques by taking advantage of the characteristics of particular problem types. This document introduces and characterizes two new automated techniques inspired by the methods of the practitioners. In the first technique, test generation is organized to fit the characteristics of an important class of circuits. The second technique (suitable for a different class of circuits) applies a conventional planning

method to novel representations of testing goals and primitive actions.

The first technique is based on a new solution to the embedding problem, the key step in test generation for sequential circuits.

> *Embedding problems can be solved efficiently using* operation relations, *a representation of circuit behavior that connects internal component operations with directly executable circuit operations. Operation relations can be computed efficiently for circuits that provide few operations at their interfaces by searching traces of simulated circuit behavior.*

Traditional methods embed tests by repeatedly refining the goal of causing a specific internal behavior until the problem can be solved by direct action on the circuit inputs. Often a newly proposed subgoal conflicts with previous subgoals, forcing the test generator to backtrack and try again. This alternation of search and backtrack is characteristic of planners in general and test generators in particular.

Goal refinement is inefficient when solutions are infrequent and there is little guidance available to lead the test generator to them quickly. Unfortunately, test generation for complex, sequential circuits seems to be such a situation: a test generator is likely to propose and retract many potential solutions before finding one that meets all of the constraints imposed by the circuit structure and behavior. The difficulty of finding solutions is compounded by the potentially high cost of ruling out proposed solutions, since the test generator may reason about the circuit far backward or forward in time before discovering a constraint that causes it to backtrack.

It is, however, possible to avoid this pitfall if the circuit executes a small number of operations, as does, for instance, a processor with a small instruction set. An effective planning strategy for circuits in this class is to take the circuit operations as the planner's primitive actions and simulate them, looking for patterns of internal activity that could prove useful during testing.

This **simulate and match strategy**, in effect, turns goal-refinement planning on its head. Goal-refinement planning starts with a set of goals (e.g., a method to test a component) and asks "are these goals achievable?" Simulate and match instead asks "what is achievable and do any of those things meet our goals?" The approach focuses search on behavior *known to be achievable* rather than on potentially achievable behavior that must be verified via complex reasoning. Focusing on known-achievable behavior is effective for planning problems with large search spaces and few solutions caused by highly interacting subgoals, e.g., test generation for sequential circuits. These ideas are embodied in a test generation program called DB-TG, the Designed Behavior Test Generator.

The second technique is based on the following observation: testing practitioners write test programs rather than test vectors because programs are a more readable and compact representation for tests than vectors are.

> *Test programs can be constructed automatically by merging test program fragments using expert-supplied goal-refinement rules and domain-independent planning techniques from AI. Additional leverage arises from giving the test generator knowledge of the capabilities of the tester hardware.*

While conventional test generation and AI planning techniques are inefficient on problems with strongly interacting goals, they are effective on problems with weakly interacting goals. This kind of problem appears to correlate with moderately complex, sequential circuits that provide good accessibility from the outside. These conventional techniques provide a foundation for exploring ideas about representing tests as programs, creating tests by selecting and merging program fragments, and describing the capabilities of the tester (i.e., the agent that will perform the test) to the test generator. These ideas are embodied in a test generation program called PF-TG, the Program Fragment Test Generator.

The main contributions of this thesis are two novel methods of generating tests introduced above. These new methods plus the existing combinational, functional and special-purpose test generators (e.g., for memories) form a collection of tools that test engineers can draw from as appropriate. Our larger vision, of which this thesis is a part, is to build a collection of specialized testing tools that share circuit descriptions and work together autonomously or partially under human guidance to solve testing problems. This thesis is a step toward that goal.

1.3 A Reader's Guide

This thesis is aimed at two distinct audiences: circuit testing researchers and artificial intelligence researchers. Testing researchers may find this thesis useful as a description of two novel test generation techniques. Each achieves power by exploiting the characteristics of a class of circuits. Neither is powerful enough to solve the problem for all circuits. Specialized solutions of this kind are now and are likely to remain the state-of-the-art.

AI researchers may find this thesis useful for different reasons. At its core test generation is a classical planning problem that happens to be important in the real

world. The problem is formally intractable yet commercially important and must be handled by industry as best it can. AI researchers, particularly in the planning and engineering problem solving subfields, can view this thesis as a case study in identifying solution techniques appropriate for two broad classes of planning problem.

I would have liked to write this document in two colors – red for testing readers and blue for AI readers – and to have included colored glasses with each copy.[1] As it is, ideas from both fields are mixed together indiscriminately. Perhaps this is a good thing, as each field has something to say to the other. The engineering problem solving area of AI has developed a collection of methods for representing and reasoning about engineered systems. Circuit testing is a fertile source of hard problems, and its researchers have developed some similar ideas in parallel.

The rest of this introduction gives an overview of the thesis. Section 1.4 briefly contrasts how the algorithms and experts solve embedding problems and asserts that the algorithms can be made more effective by emulating some characteristics of the experts. Section 1.5 discusses the informal approach I have taken toward studying the experts, using their methods as sources of hints rather than as something to emulate in detail. Section 1.6 contains scenarios for two test generation systems that embody the main ideas of the thesis. Section 1.7 characterizes the problem we are interested in solving with respect to the broad space of circuit testing methods. Finally, section 1.8 summarizes the two new methods and contrasts their strengths.

The remainder of the document has roughly the same structure. Chapter 2 reviews the fundamental concepts and algorithms in the field of circuit testing. Chapter 3 describes how circuit testing and test generation is currently practiced. Together, these chapters deepen the contrast described in this introduction. The bulk of the document discusses the Designed Behavior Test Generator. Chapter 4 introduces the ideas behind this test generator and describes an example in detail. Chapter 5 analyzes its performance to determine the boundaries of its effectiveness and where its power comes from for problems within those boundaries, and chapter 6 answers some of the limitations by extending the basic method.

Chapter 7 describes the Program Fragment test generator, and appendix A contains additional examples. Chapter 8 describes how the ideas in this thesis connect to the literature and suggests future work, and chapter 9 summarizes the contributions of this thesis.

[1] The rose colored glasses for AI?

1.4 Existing Algorithms vs Human Experts

The need for test generation far outstrips the capabilities of existing algorithmic theory and is currently met by the application of human intelligence. This section briefly contrasts the existing algorithms with the methods of the expert practitioners.

1.4.1 Testing Theory: The Existing Algorithms

The classical test generation algorithms [roth66, goel81a, benmehrez83, fujiwara85] designed for combinational circuits solve the component test problem by using individual logic gates as components. Logic gates are simple enough that it is practical to work out tests by hand or to exercise them exhaustively. The classical algorithms then solve the embedding problem by propagating signals through the gate-level circuit model, going forward from component outputs and backward from component inputs and searching for consistent combinations of signals. These algorithms use highly optimized search heuristics and constraint propagation techniques, yet runtimes are excessive when applied to other than combinational circuits (up to roughly 10,000 gates) and simple state machines. Long runtimes are a consequence of the complexity of gate-level models of modern circuits.

Recent research [lai81, genesereth81, davis82a, shirley83b, khorram84, singh85, krishnamurthy87, chandra87] has identified several ways of increasing efficiency by using abstract representations of circuit structure and behavior to take larger steps during test generation. Some test generators have used a hierarchical circuit model and a strategy for selecting which level of the model to propagate signals through. Others have used a single, abstract circuit model suitable for a particular circuit type, e.g., a microprocessor.

The strategy of exploiting abstract representations is the most important, recent development in test generation. The designed behavior test generator introduced in this thesis is a step in this line of research: I identify and use a new kind of abstract representation of circuit behavior called **operation relations**. This representation is described in section 1.6.1.

1.4.2 Testing Practice: Human Experts

The following characteristics of expert test programming stand out:

1. *Experts understand how circuits work.* They know much of what circuit designers know, and their resulting understanding of circuit behavior enables them to focus on likely solutions to testing problems. For instance, they know what operations a circuit was designed to perform; they know which components implement which operations; they know the normal patterns of circuit activity; and they know relationships between the behavior of components that may be widely separated in the schematic. As we will see, all of these are useful in generating tests.

2. *Experts rely heavily on functional descriptions and block diagrams from databooks.* They are often forced to do this because they do not have access to detailed schematics. Furthermore, the tests they generate must use legal inputs expected by the designer, so the databook descriptions are applicable.

3. *Experts rely on past experience* (i.e., tricks of the trade). They know how to test commonly occurring components like registers, multiplexors, ROMs, etc, and they build circuit tests from these component tests rather than start from primitive gates.

4. *Experts write test programs not test vectors.* They use a more expressive representation for tests than the traditional algorithms do.

5. *Experts know the capabilities of the tester and can match them to testing problems.* A circuit tester[2] has special features for implementing commonly-needed tests efficiently. Existing test generation algorithms do not (and cannot easily) use knowledge about these features.

6. *Experts know when to use the traditional algorithms.* The traditional algorithms are extremely effective at solving certain parts of the problem. The experts know when and where these algorithms work well and use them judiciously.

7. *Experts can sometimes negotiate the boundaries of their problem.* As testability becomes an increasingly important design criterion competing with performance and cost, so test experts become more central members of the teams designing circuits. Often difficulties in test generation can be averted by changing the design, i.e., by negotiating with the advocates of other design criterion to simplify the problem that the test expert has to solve. The experts need test generators that are created with this design environment in mind.

[2]A tester is a tool for testing circuits. Testers are generally implemented as digital computers with special peripherals for interfacing with circuits.

1.4.3 Observations

For testing complex sequential circuits, experts are much more successful than any existing algorithm. How do the experts succeed?

Part of the answer lies in their use of abstract circuit descriptions. The circuit testing community has begun to explore this direction; this thesis extends this line of work by describing a novel kind of abstract circuit representation that is particularly suited to solving embedding problems in sequential circuits.

But abstraction is only part of the answer. Current programs for test generation are like current programs for playing chess [3]: they both succeed by dint of prodigious search. Yet in both fields, the best human practitioners still outperform the best programs. The central lesson that comes of comparing the algorithms with the experts is that the experts know more and search less. Domains of this type are familiar in AI,[4] and the central question to ask is "what is the knowledge and how can it be represented?" At its core, this is what this thesis is about: identifying several kinds of knowledge about test generation than have not been exploited by the algorithms so far.

This thesis describes two methods of applying expert-supplied how-to-test knowledge. The first method, DB-TG, concentrates on the first and second characteristics of expert test programming in the list above, i.e., understanding how circuits work and using block diagram descriptions. The second method, PF-TG, concentrates on the fourth and fifth characteristics, i.e., writing test programs and knowing the capabilities of the tester. These ideas are described in the scenarios below.

1.5 A Word About Methodology

Protocol analysis of expert problem-solving behavior has been a major source of the ideas in this thesis.[5] My goal, however, is to design effective test generation algorithms, not to duplicate the behavior of human test experts. Consequently, these protocol analyses were informal and are not emphasized in this thesis.

[3] Circuits and their designers often do seem like formidable antagonists to a test programmer.

[4] The chemical analysis domain of Dendral[lindsay80] is the classic example.

[5] The primary expert I have talked with is Gordon Robinson of GenRad Inc. I have studied Gordon's problem-solving methods on examples of several classes of circuits including a microprocessor, a processor bit-slice, a digital filter and a communications multiplexor. Short protocols of Mark Swanson of GenRad, Prof. Melvin Breuer of USC and Dr. Alex Miczo of Schlumberger have also been very helpful.

Instead, my approach has been to draw inspiration from the methods of experts and to combine their strengths with the strengths of the algorithms in the hope of eventually surpassing them both. Surpassing human test programming skills in a computer program is a broad and deep goal. I have by no means accomplished this task in this thesis, but I believe I have taken several significant steps along the path.

1.6 Scenarios

This section contains scenarios describing DB-TG, the Designed Behavior Test Generator, and PF-TG, the Program Fragment Test Generator. Scenario I is a shortened version of an example that appears in chapter 4, consequently there is some overlap. The example appears here in sufficient detail to make this chapter a reasonably complete and self-contained introduction to this thesis.

1.6.1 Scenario I: Exploiting Designed Behavior

DB-TG solves embedding problems: it transforms pre-written component tests expressed in terms of component I/O into directly executable tests expressed in terms of circuit I/O. The program inputs are: (i) a schematic at the level of a block diagram, (ii) descriptions of each operation the circuit is designed to perform (e.g., each instruction), (iii) simulation models of the components and (iv) expert-written component tests expressed in terms of component operations. The component tests include descriptions of the faults they are designed to detect. DB-TG outputs: (i) a test for the circuit consisting of one test per component and (ii) descriptions of the faults these tests are designed to detect.

This test generator was designed as part of an approach to integrate test generation and design for testability [shirley87, wu88]. It is intended primarily as a tool for use early during design to produce tests and to assess a design's testability. Consequently, we emphasize using the kinds of high-level circuit descriptions available early during design. A second consequence is that the circuits DB-TG generates tests for may not be completely finished, i.e., designers may still be trading off various design criterion. Therefore, we emphasize quickly identifying and solving testing problems that are straightforward (for people, not necessarily for existing programs) and not expending large amounts of time detecting every last fault. In particular, the failure of this program can be viewed as indicating a testability problem [wu88]. This issue is considered again at the end of the scenario.

DB-TG follows these steps: (i) lookup a component test from a library (the test is

in terms of a component operation), (ii) compute relationships between that component operation and the circuit operations from the circuit description (these so-called **operation relations** are a highly abstract description of the surrounding component behavior), (iii) use the relationships to embed the component test. These steps are the equivalent of propagating from the component backward to circuit inputs or memory cells and propagating forward to circuit outputs or memory cells. DB-TG then uses conventional AI planning technology (STRIPS) to plan sequences of circuit operations for controlling and observing circuit state.

This scenario focuses on: (i) the form of operation relations, (ii) how DB-TG uses operation relations to embed component tests and (iii) how DB-TG computes operation relations from schematics and component models. First, I introduce operation relations with a simple embedding problem.

1.6.1.1 An Easy Embedding Problem

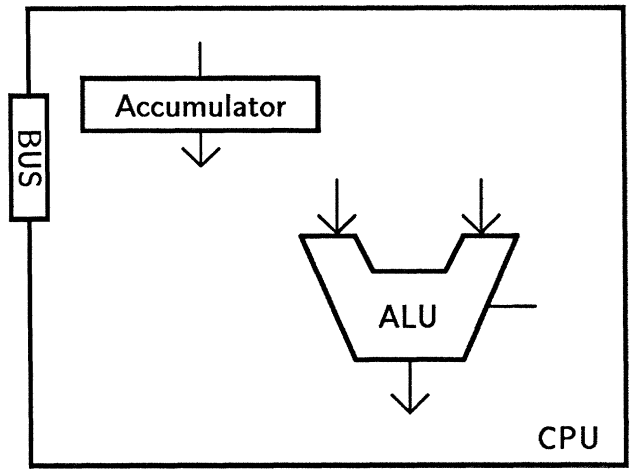
Figure 1.3 shows an easy embedding problem: test the ability of the processor's ALU to add numbers. Suppose an expert knows something about testing ALU's and says the way to test this ALU is to cause it to add several specific pairs of numbers (A_i, B_i) and then to observe the sums to make sure they are correct. Suppose also that this processor is implemented as a single chip, so we have no direct access to the ALU. Then the problem is to work out how to manipulate the bus to cause the processor to send the A's and B's to the ALU and to bring the sum back out to the bus for observation.

You may wish to stop reading and try to solve this problem. Make reasonable assumptions as needed about the way the circuit works, e.g., assume this processor provides the usual instructions for performing arithmetic and manipulating the accumulator.[6]

We find that testing experts and others familiar with computer architecture can easily suggest something like the following solution:

1. Load Accumulator with an A using the LOAD instruction.
2. Add a B to that using the SUM instruction.
3. Write the sum to the bus using the STORE instruction.

[6]This problem really is as simple as it may seem. The point is to ask what kinds of circuit representations make it simple, and how it might be possible to solve the problem without knowing detailed structural information like how datapaths connect the accumulator with the ALU.



Instructions

`LOAD value`	*– Load accumulator from memory*
`STORE value`	*– write accumulator to memory*
`ADD value`	*– increment accumulator by value*
`SUB value`	*– decrement accumulator by value*

Figure 1.3: *Work out how to make the ALU add numbers. Make reasonable assumptions about the way the circuit works, e.g., assume the processor provides instructions like those at the right for performing arithmetic and manipulating the accumulator.*

4. Repeat until all pairs of A's and B's have been used.

This solution relies on several assumptions, including: (i) that the SUM instruction actually uses the ALU shown in the figure[7], (ii) that the LOAD, STORE and SUM instructions manipulate the accumulator shown, and (iii) that the LOAD, SUM and STORE instructions can handle the test data required by the ALU. Before using this solution, we would have to check that these assumptions were warranted.

This example raises several interesting questions. What A's and B's will adequately test the ALU's ability to add? What descriptions of LOAD and STORE allow them to be easily recognized as useful for controlling and observing the accumulator? How is SUM identified as the key instruction to use? Associating the SUM instruction with the goal of making the ALU add is the key to solving this embedding problem, so we focus on this.

One telling observation is that testing experts can construct solutions *without* considering the detailed structure of the circuit. In particular, the routes taken by A and B from the bus interface to the ALU or the details of the microcode implementing the SUM instruction were not shown in the figure. Since the omission does not prevent

[7]While usually warranted, this assumption is questionable for heavily pipelined processors that can use several physical ALU's to implement a single "virtual" ALU.

an expert from arriving at a candidate solution, those details must not be essential. There must be some way of describing the circuit that allows candidate solutions like this to be proposed without a detailed examination of circuit structure.

Asking what such an abstract circuit description might be leads us to the idea of explicitly representing and manipulating relationships between operations of the circuit and operations of its components.

1.6.1.2 Operation Relations: Part-Whole Descriptions of Circuit Behavior

As a component is a part of a circuit, so is its behavior a part of the circuit's behavior. Knowing part-whole relationships about behavior is useful, because they provide a straightforward means of embedding component tests into the circuit.

Two kinds of part-whole relationships are useful for solving embedding problems: **causal connections** and **parameter relations**. In the example above, executing a SUM instruction *causes* the ALU to add, therefore we say the CPU's SUM instruction and the ALU's addition operation are causally connected. Parameter relations hold between the parameters of two causally connected operations. In the example there is a time during the execution of an addition instruction when the ALU does the real work. At that time, the two values being summed by the addition instruction are the *same two values* being summed by the ALU. In this case the parameters of the addition instruction and those of the ALU addition operation are related by identity functions. The term **operation relations** refers to both the causal connection between two operations and to relationships between their parameters.

Figure 1.4 shows these relationships between the ALU and the CPU. Each box contains a frame-like representation of the externally visible effects of an operation. The upper box describes the processor's SUM instruction and the lower box describes the ALU's ADD operation. The causal relation (c) says that executing a SUM instruction will cause the ALU to ADD. In this example, the parameter relations (d) are identities, i.e., the values of the corresponding variables must be the same.

1.6.1.3 Using Operation Relations to Solve Embedding Problems

Relationships between component and circuit operations exist because the designer used component behavior to implement circuit behavior in the first place. The relationships are useful because they provide a *direct link* from desired actions inside the circuit to actions executable by the tester hardware (see figure 1.5). Using this direct

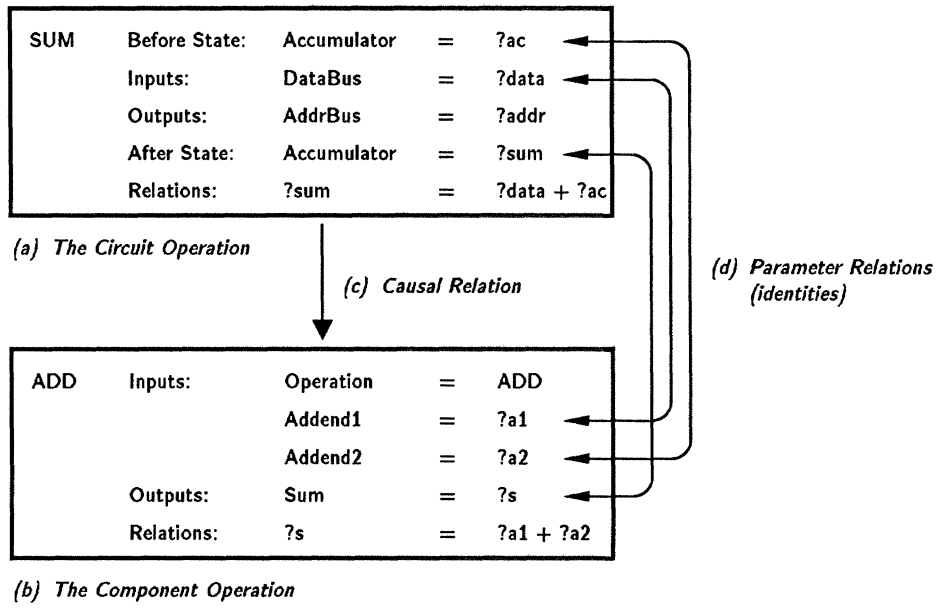


Figure 1.4: *One set of operation relations for the ALU example.*

link, DB-TG transforms component tests for the ALU (which a tester cannot manipulate directly) into equivalent tests expressed in terms of CPU operations (which the tester can execute directly).

To do this transformation, DB-TG substitutes component test data into the component side of the parameter relations and solves for the parameters of the circuit operation. For instance, the operation relations in figure 1.4 connect variables mentioned in the test data with the parameters of the SUM instruction. Figure 1.6 shows expert-supplied test data for an ALU addition operation. In this case, the operation relations happen to be identities so substituting test data in and solving for the parameters of the SUM instruction is trivial, and figure 1.7 shows the result for one line of test data.

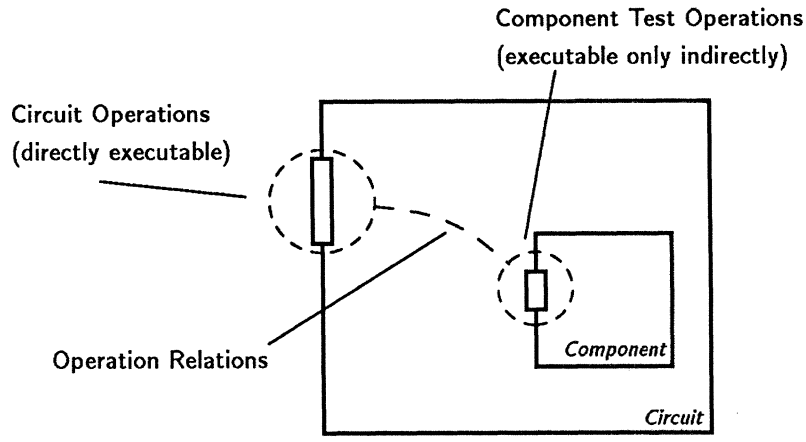


Figure 1.5: *Operation Relations are a direct link between the goals (i.e., desired component operations) and the primitive actions (i.e., directly executable circuit operations).*

?A1	?A2	?S
0	0	0
43690	43690	21844
1	65534	65535
1	65535	0
65534	1	65535
65535	1	0
21845	21845	43690
65535	65535	65534

Figure 1.6: *Expert-supplied test data for an ALU's addition operation.*

SUM	Before State:	Accumulator	=	65534
	Inputs:	DataBus	=	1
	Outputs:	AddrBus	=	?addr
	After State:	Accumulator	=	65535
	Relations:	65535	=	$1 \oplus_{16} 65534$

Figure 1.7: *Line 3 of the test data substituted into the SUM instruction.*

1.6.1.4 Computing Operation Relations via Simulation

DB-TG computes the relationship between a circuit operation (e.g., SUM) and a component operation (e.g., ADD) by simulating the circuit operation and then searching and extracting node values from the simulation trace. DB-TG uses an event-driven, symbolic simulator. It takes as input a circuit schematic, behavioral models of the components and descriptions of the instructions. (Figure 1.8 shows a simple microprocessor model that fleshes out the example above. The simulator uses this schematic and models of these components. Uninteresting components, e.g., latches in the datapaths, are not shown.) The simulator outputs a set of simulation traces, called **behavior graphs**, that describe what happens inside the circuit as the instructions execute.

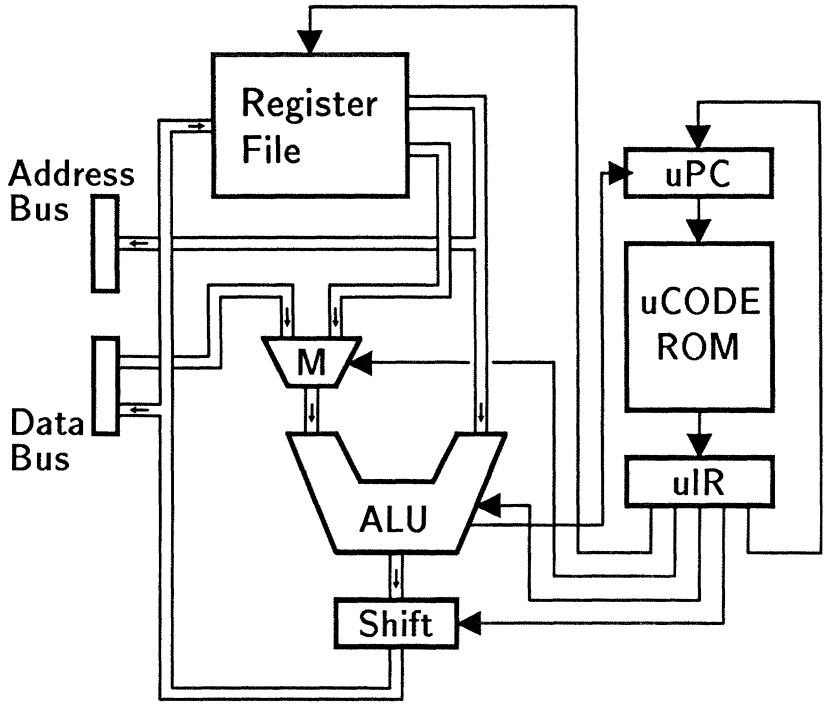


Figure 1.8: *The MAC-1 Microprocessor (some detail has been suppressed)*

Symbolic simulation is the process of propagating variables and algebraic expressions as well as numbers through the circuit. Doing this allows a single simulated operation to stand for an equivalence class of similar operations. For example, a LOAD instruction with symbolic data can stand for a LOAD of any specific constant.

Using symbolic simulation is important for two reasons: (i) the algebraic expressions that propagate through the circuit are what turn into operation relations and (ii) simulating equivalence classes of behavior rather than specific behaviors reduces the number of simulation runs needed and the size of the database that holds the results.

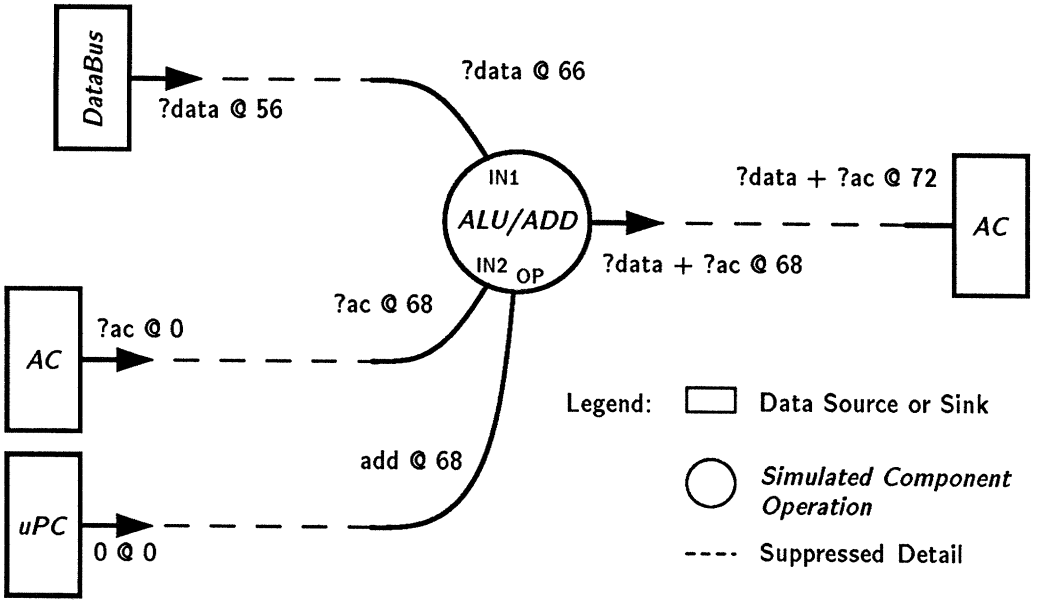


Figure 1.9: *The Behavior Graph for the SUM instruction. Time and data flow from left to right.*

To test the ALU's ADD operation, DB-TG searches the simulation trace of each instruction for simulated ADD operations. Figure 1.9 shows one such ADD operation generated by simulating the SUM instruction of the microprocessor in figure 1.8. This microprocessor has an ALU and an accumulator and is a complete version of the simple example above. A rectangle in simulation trace represents a source or a sink of values – either a memory element, an input or an output. A circle represents a component operation, and the dashed lines represent portions of the graph that have been omitted in order concentrate on the activity shown. Time and data flow from left to right through the figure. The value of a node is timestamped, e.g., a node value of *data@time* indicates that the node changed to *data* at the simulated time. Node values persist until they are caused to change by other circuit activity.

The figure shows that the accumulator contains ?ac at time 0, ?data is read from the databus at time 56 and the sum of ?ac and ?data is written into the accumulator

at time 72. Here, the ALU executes an ADD operation at time 68. It receives two expressions ?ac and ?data as inputs, adding them under the control of its operation input and outputs the expression (?data $\oplus_{16}$?ac). The value on the operation input (add) beginning at time 68 comes from the microprogram ROM and ultimately from the microprogram counter.

DB-TG extracts the operation relations between the SUM and ADD operations by examining node values around the ADD operation. In this case, the variable ?ac appears in the accumulator and at an ALU input, so there is an identity relationship there. ?data and (?data $\oplus_{16}$?ac) are handled similarly. The OP input is not mentioned in the test data, so its value is not needed to form the operation relations.

1.6.1.5 Finishing the Example

Figure 1.7 shows the result of embedding one line of test data using the operation relations generated above – executing this SUM instruction will cause the ALU inside to add this line of test data. Note, however that the accumulator must be loaded with 65534 before the SUM instruction is executed, and that the accumulator's value must be observed afterwards. DB-TG plans sequences of circuit operations for controlling and observing the simulator using a STRIPS planner. DB-TG generates the operations for this planner by summarizing behavior graphs, and the details of the planning and summarization processes are described in chapter 4.

The final solution is shown in figure 1.10 (the test data is marked by $\Leftarrow$). DB-TG also produces groups of three instructions for each of the 7 other rows of test data; these groups differ only on the marked lines. DB-TG's output is the total of 24 instructions plus descriptions of the faults they are designed to detect. Executing these instructions and checking the outputs will test the addition operation of the internal ALU.

1.6.1.6 Experimental Results

Figure 1.11 shows the results of running DB-TG over the whole microprocessor, which is equivalent to roughly 6500 gates. Simulation and test generation takes 6 minutes on a lisp machine, including both the time taken for successfully creating tests for some components and failing to do so for others. The highlighted components correspond to 85% of the stuck-at faults in this circuit. Additional techniques for designing component tests on-the-fly to fit the constraints of the particular circuit raise this coverage figure to 94%. These techniques take approximately 25 additional

LOAD	Before State:	Accumulator	=	?ac	
		ProgramCounter	=	?pc-1	
	Inputs:	DataBus	=	(LOAD ?addr)	
		DataBus	=	65534	⇐
	Outputs:	AddrBus	=	?pc-1	
		AddrBus	=	?addr	
	After State:	Accumulator	=	65534	
		ProgramCounter	=	?pc	

SUM	Before State:	Accumulator	=	65534	
		ProgramCounter	=	?pc	
	Inputs:	DataBus	=	(SUM ?addr)	
		DataBus	=	1	⇐
	Outputs:	AddrBus	=	?pc	
		AddrBus	=	?addr	
	After State:	Accumulator	=	65535	
		ProgramCounter	=	$?pc \oplus_{16} 1$	

STORE	Before State:	Accumulator	=	65535	
		ProgramCounter	=	$?pc \oplus_{16} 1$	
	Inputs:	DataBus	=	(STOD ?addr)	
	Outputs:	AddrBus	=	$?pc \oplus_{16} 1$	
		AddrBus	=	?addr	
		DataBus	=	65535	⇐
	After State:	Accumulator	=	?data	
		ProgramCounter	=	$?pc \oplus_{16} 2$	

Figure 1.10: *Program Output. This set of three instructions is the embedding for the test in the first row of figure 1.6.b. The numbers from that row appear in the output on the lines marked by ⇐, where they must be applied or observed by the tester. DB-TG also produces groups of three instructions for each of the 7 other rows that differ only on the marked lines. Executing the total of 24 instructions forces the ALU to perform the test.*

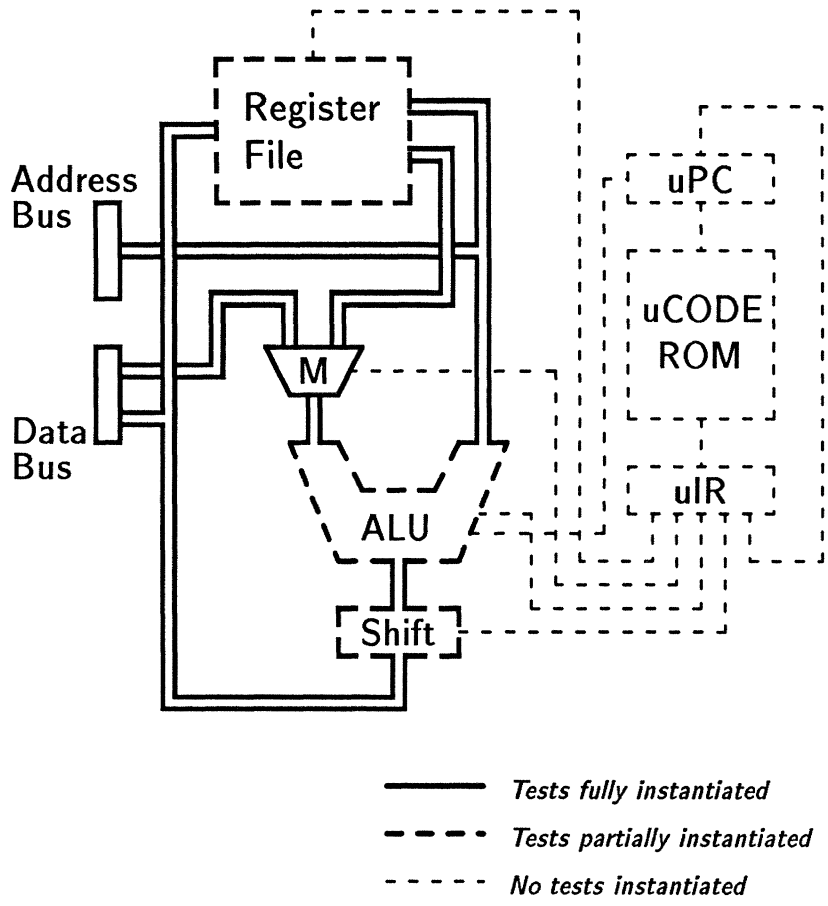


Figure 1.11: *Test generation results for the basic version of DB-TG*

minutes to run.

However, the real benefit lies in interfacing this test generator with an automated Design For Testability Advisor (e.g., [abadir85, zhu86, wu88]. We have done this with the system of [wu88], implemented its suggested modifications to the circuit (e.g., putting a scan path through the μIR and re-run the test generator to achieve 97% fault cover.

There are many points of comparison between DB-TG and other approaches in the literature. One of the most interesting is comparing DB-TG and test generators that work from functional circuit descriptions (e.g., [lai83, khorram84, brahme85]). DB-TG can be viewed as a test generator that derives functional descriptions – behavior graphs – from structural descriptions – schematics, and consequently achieves the benefits of both. It benefits from using simple, high-level functional descriptions that abstract away from the details of how data moves through the circuit. At the same time, the high-level operations that move and transform data are connected with a low-level structural model that makes functional sharing apparent. For instance, arithmetic instructions and addressing calculations are implemented with the same ALU in the MAC-1, and hence need not be tested separately. This kind of sharing is represented in behavior graphs and consequently in operation relations.

1.6.2 Scenario II: Combining Test Program Fragments

Where DB-TG is targeted at embedding problems that give rise to highly interacting subgoals, our second program, the Program Fragment Test Generator (PF-TG), is targeted at embedding problems that give rise to weakly interacting subgoals. This second kind of embedding problem is characteristic of sequential circuits that provide relatively good access to internal components. Conventional planning technology from AI appears to be sufficient to solve many embedding problems of this type and provides a foundation for exploring several new ideas about circuit testing. This work has resulted in five specific claims:

1. **Produce Programs not Vectors**: Representing tests as programs rather than vectors makes them more compact and easier for people to understand and allows convenient access to special-purpose tester features.

2. **Merge Test Program Fragments**: Test programs can be created by merging program fragments. Goal decomposition rules and temporal constraints determine which program fragments are selected and how they fit together.

3. **Represent The Tester Explicitly**: Conventional test generators assume an impoverished model of the tester's capabilities. PF-TG uses an explicit and somewhat richer model, enabling the program to take advantage of special-purpose tester features.

4. **Propagate Typed Streams**: PF-TG can propagate tokens that represent typed streams of values, e.g., a `counting-stream`. Propagating typed streams can generate repetitive tests that are more efficient over a wider class of circuits than can propagating symbolic variables, the method of existing hierarchical test generators.

5. **Use Flexible Goal Structure**: The goal/subgoal structure of the test generator can profitably reflect the problem-solving methods of human test programmers as well as the structure of the circuit.

This scenario illustrates claims 1 and 2.

A test program is a sequence of instructions for testing a circuit that is executed by computer. Programs are a good representation for tests for several reasons. First, test programs are often more compact than the equivalent vectors. The size reduction stems from using looping constructs to encode repetitive tests. Second, tests have structure and test programs make that structure explicit, making them more readable by people than vectors. Readability is important when a test generator is used by an expert as a tool to help solve a complex problem: the expert must be able to understand, augment and modify the program's output. Third, test languages provide convenient access to special-purpose tester features, e.g., hardware for generating memory tests.

1.6.2.1 Structure of the Program

PF-TG generates test programs using these five steps:

1. **Problem Decomposition**: How-to-test rules decompose the problem of generating a test into subproblems. Decomposition continues until directly solvable subproblems are reached (e.g., controlling a circuit input or generating tests for a small combinational component) yielding a tree of rule invocations. Rules are stored in the **Rule Library** and are selected and executed by the **Rule Engine**.

2. **Fragment Collection and Constraint Posting**: In addition to breaking up test generation problems, rules can put program fragments into the output test

program. When the engine executes a rule, it copies any program fragments in the rule and passes them to the code manager. Rules also contain constraints controlling how the program fragments fit together. Constraints either control the execution times of program statements or the allocation of tester or circuit resources, and they are passed to the **Time Manager** and to the **Resource Manager** respectively.

3. **Constraint Satisfaction**: The Resource Manager reduces resource constraints to temporal constraints. These plus the temporal constraints sent directly to the Time Manager are reduced to a set of linear inequalities in two variables, where the variables represent execution times. The Time Manager solves these inequalities for integer values.

4. **Code Generation**: The Code Manager sorts the program fragments by execution time and assembles code for the tester.

How-to-test rules decompose testing problems into groups of simpler problems. PF-TG applies the rules using a backward-chaining rule engine based on Prolog. Each rule has four components: (i) a pattern describing what goals the rule can solve, (ii) a set of subgoals to introduce, (iii) a set of program fragments to include in the test program, and (iv) a set of constraints describing how the program fragments fit together with each other and with fragments posted by other rules.

The system has rules for solving the following kinds of problems: (i) how to test a component, (ii) how to control component outputs, (iii) how to move data through a component, (iv) how to make the tester drive circuit inputs, (v) how to make the tester observe circuit outputs, (vi) how to make a component inactive, (vii) how to initialize a component, (viii) how to move a state machine from one state to another, and (ix) how to take a state machine through a cycle. This scenario shows rule types (i) - (v). Consider, for example, the rule for testing a parallel datapath, i.e., a group of wires or components acting like wires that moves information from one part of the circuit to another (see figure 1.12). This rule breaks up the problem of how to test a datapath into the problems of selecting test data to send down the datapath, enabling it and getting the data to and from it.

Many subgoals involve writing test program code. For instance, the subgoal of enabling the datapath must eventually be solved by writing code to make the tester drive circuit inputs so they will enable the datapath in turn. The program writes this code by working out how to enable the datapath – via propagating back to circuit inputs – and combining the program fragments that appear in rules for controlling those inputs. As a very simple example, PF-TG would emit the following program

To test a parallel datapath:

1. Select a vector stream that has sufficient coverage characteristics.
2. Write a code fragment to enable the datapath.
3. Write a fragment to generate this vector stream.
4. Write a fragment to move each vector to the beginning of the datapath.
5. Write a fragment to pick up values as they reach the end of the datapath and move them to a place where the tester can observe them.
6. Write a fragment which observes the values on the primary output and compares them with expected values.
7. Create constraints to link these program fragments together. For example the time a vector is generated by fragment #3 must be the same time it starts to move to the beginning of the datapath in #4.

Figure 1.12: *How to test a parallel datapath.*

fragment to make the tester assign the circuit input called `in` the value 1:

```
in := 1;
```

More complex assignments can emit groups of statements or program loops. Depending on the intervening component types, the rules used at intermediate stages in the propagation sometimes include program fragments as well.

PF-TG uses three types of constraints to control how program fragments fit together. **Temporal constraints** control the execution times of program statements, e.g., statement A must execute before statement B. **Structural constraints** control the structure of the test program, e.g., assignment statement C must appear within loop D's body. **Resource constraints** control the allocation of scarce resources to different uses at different times, e.g., circuit node E has a certain value at time T and

cannot have any other.

The times of primitive tester actions are represented by integers. Each statement in a test program is associated with a temporal variable, and that variable is eventually bound to an integer representing the execution time of the statement. PF-TG controls timing relationships between program statements by controlling the relationships between the associated temporal variables. The relationships are specified by the how-to-test rules as they are used and are collected and solved by the Time Manager.

The Time Manager handles these relations between pairs of variables: $=$, $\neq$, $<$, $\leq$, $>$, $\geq$, plus **and** and **or** connectives between expressions. For convenience, PF-TG provides a macro language for expressing more complex relations such as `disjoint-intervals` and `overlapping-intervals`.

PF-TG handles structural constraints similarly, i.e., structural variables are constrained using the algebraic relations above and an equation solver assigns integer values that represent textual order in the final test program.

PF-TG goes over all resource assignments (e.g., node assignments made during propagation or assignments of tester hardware to subtasks of driving the circuit) and creates temporal constraints to make sure they do not conflict. Suppose, for example, that `A=1@T1`, meaning node A is assigned the value 1 at time T1, and `A=0@T2`. Because a single physical node cannot have more than one value at a time, PF-TG creates the temporal constraint ($\neq$ T1 T2).

1.6.2.2 An Example

The following example shows PF-TG generating a test program for the multiplexor AMUX in figure 1.13. The circuit is a simple 4 bit wide datapath. The ALU has four operations, among them a NOOP operation that copies data from IN1 to OUT. The register file is a single input, dual output memory with 16 cells. The address lines RF-AA and RF-BA select outputs for OUT-A and OUT-B respectively. RF-CA controls which register is loaded from the DATA-IN input, which happens on the rising edge of the clock. An enable input has been omitted from this example. All inputs are directly controllable, and all outputs are observable. All other nodes are internal to the circuit and are accessible to the tester only through intermediate components.

In this example, PF-TG generates a portion of a test program that verifies whether AMUX is working properly. The example is implemented, although a few details have been changed to clarify the explanation. In particular, the example shows program code that would result if PF-TG stopped at various times and merged several program

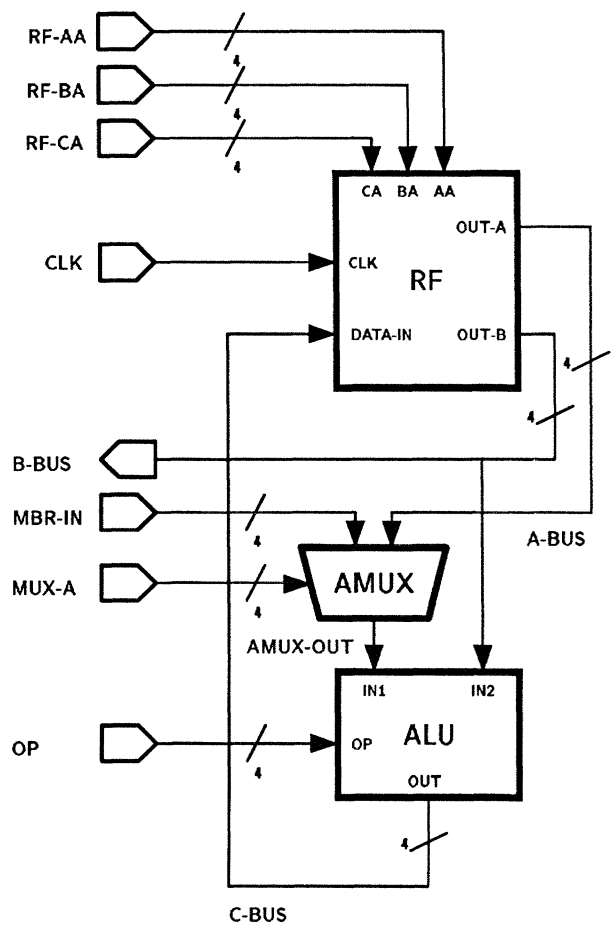


Figure 1.13: *A Simple Datapath*

To test a two-input mux:

Run one DATAPATH test from the mux's first input (in0) to its output. Then run another datapath test from the second input to the output.

Figure 1.14: *How to test a two-input multiplexor (simple method).*

fragments. The fragments actually remain separate until the end, when they are merged all at once. Also, the rules and program fragments are rendered in English and pseudo-Algol to improve readability. The actual rules and program fragments have a lisp-like syntax.

We start by asking PF-TG to write a test program for AMUX. The rule in figure 1.14 responds to this request. PF-TG maintains an agenda of independent programming tasks. Each task involves writing a section of the test program that exercises a single component or an aspect of a component's behavior. In this example, the rule above breaks up the problem of writing a test program for AMUX into two tasks, each of which involves testing the mux's ability to pass information from one of its inputs to its output. These top-level tasks can be solved separately, and PF-TG works on each of these programming tasks in turn. Both tasks are solved by using the DATAPATH rule shown earlier. Since the tasks are similar we describe only the second one, that of writing a datapath test from AMUX's right input (IN1) to its output.

First, PF-TG chooses test data to use, and in this case it chooses the diamond pattern (see figure 1.15). This pattern will detect stuck-ats and bridge faults in the datapath. Next, PF-TG works on writing code to enable the datapath. In this case, the datapath is very simple: it runs through the mux, from its right input to its output. This datapath is enabled by selecting the right input. Longer datapaths are handled by a rule that partitions datapaths into smaller parts and then constrains the parts to be enabled at the same time. Eventually, the problem is reduced to enabling through single components, which are handled directly. Because the select input of AMUX is directly controllable by the tester, this rule proposes the program fragment shown in figure 1.16.a.

This loop repeatedly selects the right input of AMUX, thus enabling the datapath.

vector #	1	2	3	4	5	6	7	8
Wire 1	0	1	1	1	1	0	0	0
Wire 2	0	0	1	1	1	1	0	0
Wire 3	0	0	0	1	1	1	1	0
Wire 4	0	0	0	0	1	1	1	1

Figure 1.15: *A diamond pattern for a 4 bit wide datapath. "Diamond" refers to the shape of the region of 1 bits.*

This enabling action can occur an arbitrary number of times, as indicated by the <. . .> placeholder for the iteration clause. These placeholders will be filled with code from other rules.

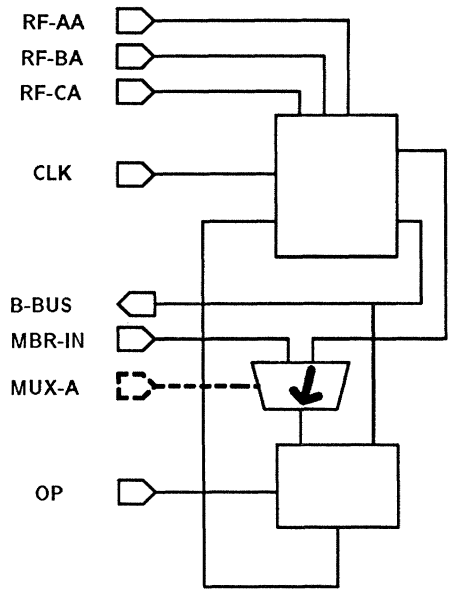
A temporal variable, TEST-TIME, is associated with the tester-assign statement. This temporal variable denotes the short interval during which a single test pattern passes through AMUX. The temporal variable was created by the DATAPATH rule and passed down to the ENABLE-THROUGH-MUX rule so that the resulting program fragments can be synchronized. [8]

The next subproblem involves writing code to generate a diamond pattern. A simple way to do this is to fill an array with the appropriate sequence of test vectors, then step an index through the array, fetching vectors and putting them on the circuit inputs. The **generate-diamond-stream** rule implements this method with the fragment in figure 1.16.b1.

The next subproblem is the most complex - move vectors from a primary input to the beginning of the datapath (i.e. the IN1 of AMUX). The path used and the resulting code is shown in figure 1.16.b2. The path is found via line justification, which involves searching backward from the MUX input to any primary input. Since we're moving a diamond pattern, the path chosen must be able to transmit parallel data, and the path shown is the only possible solution.

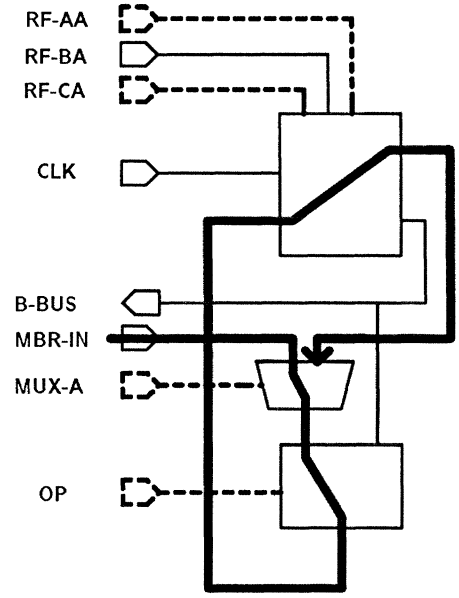
The code fragment is a combination of three fragments that enable the datapath to pass through AMUX, ALU and RF (the register file) respectively. The statement on line 17 of figure 1.16.b2 is from the body of a loop analogous to the one on line 4 of figure 1.16.a except that it selects the mux's other input. The statement on line 18

[8]There are several temporal variables not shown in the figure. For example, there are variables associated with the start and finish times of the loop.




```
1.  FOR <...> DO
2.      BEGIN
3.          <...>
4.          MUX-A := 1   ** at TEST-TIME
5.          <...>
6.      END;
```

(a) Enable the datapath through MUX-A.




```
7.  ARRAY DiamondPattern = [ ... data ... ];
8.  FOR index = 0 to 7 DO
9.      BEGIN
10.     <...>
11.     MBR-IN := DiamondPattern[index] ** at TIME-1
12.     <...>
13.   END
```

(b1) Have the tester apply a diamond stream to a circuit input.

```
14. FOR <...> DO
15.     BEGIN
16.         <...>
17.         MUX-A := 0        ** At TIME-1
18.         OP := ALU-NOOP    ** At TIME-1
19.         CA := 0           ** At TIME-1
20.         <...>
21.         AA := 0           ** At TEST-TIME
22.         <...>
23.     END;
```

(b2) Move the diamond stream from that input to the AMUX.

(b) Supply a diamond stream to AMUX's left input.

Figure 1.16: *PF-TG Scenario*

is from a loop that enables information to flow across the ALU. The statements one lines 19 and 21 come from a rule for moving a stream through a register file. This rule handles the address lines, and presupposes a free running clock and chooses which register file cell to use.

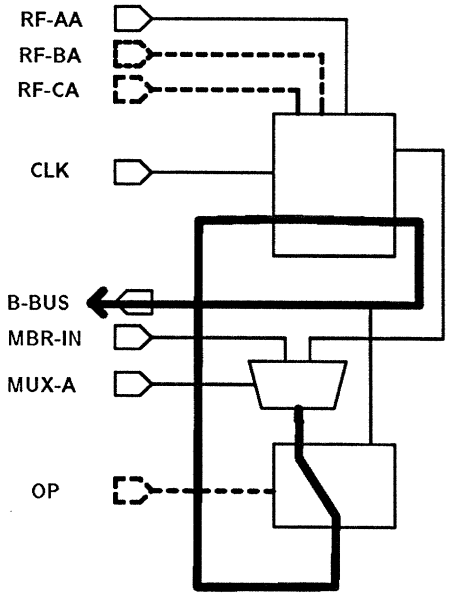
The next subproblem is to move the diamond stream from AMUX's output to a primary output of the circuit. PF-TG's solution to this problem is analogous to its solution for moving data in to the MUX's input and is shown in figure 1.17.c1. Finally, PF-TG uses the code fragment in figure 1.17.c2 to check the observed outputs against the expected values. At this point, PF-TG has expanded every subgoal introduced by the original DATAPATH rule.

So far, we've emphasized program fragments at the expense of the constraints that control how the fragments are put together, but both are equally important. PF-TG next collects all of the constraints from the rules it has used and solves them. There are about 30 temporal relations such as the one relating clock initialization time, *init*, with the clock's first use, *time-1*. The relation is *(< init time-1)* which says that the clock must be initialized before its value is used. The TESTER-ASSIGN statement at time *init* is a small program fragment separate from the loop, and is related to the loop by the temporal relation. In fact, most program fragments tend to be small and have many relations to other small fragments.

Finally, there are several resource constraints. For instance, the fragment in figure 1.16.a assigns MUX-A to 1 at TEST-TIME and the fragment in figure 1.16.b2 assigns MUX-A to 0 at TIME-1. Therefore ($\neq$ TEST-TIME TIME-1).

One particularly interesting resource is the stack of the computer inside the tester. Statements in the program fragments that use the stack must obey stack discipline. In particular, loops in this language allocate their iteration variables on the stack, hence two loops must appear in the program either one before the other, one inside the other, or merged to share the same iteration variable. This resource constraint is converted into temporal and structural constraints on the loop statements. (This constraint, together with the fact that TEST-TIME or TIME-1 appear inside all of the loops in the program fragments, is what causes PF-TG to merge all of the loops together in the final test program.)

Once the constraints have been collected and resource constraints converted into temporal constraints, an equation solver for systems of linear inequalities produces a solution assigning each temporal variable to an integer. If no solution is possible, the system backtracks and chooses different rules. The integers represent the execution times of the statements associated with the temporal variables; in the case of loops, they represent the times within a prototypical execution of the loop body.




```
24. FOR <...> DO
25.     BEGIN
26.         <...>
27.         OP := ALU-NOOP   ** At TEST-TIME
28.         CA := 0          ** At TEST-TIME
29.         <...>
30.         AA := 0          ** At TIME-2
31.         <...>
32.     END
```

(c1) Move the MUX outputs to a circuit output.

```
33. FOR <...> DO
34.     BEGIN
35.     <...>
36.     B-BUS = DiamondPattern[index] ** at TIME-2
37.     <...>
38.    END
```

(c2) Have the tester check the output stream (the DiamondPattern array has already been declared).

(c) Observe and verify the AMUX outputs.

```
1.   ** TEST-PHASE (:COMPONENT SELECT-A)
2.   ** Perform a DATAPATH test from IN1 to OUT
3.   BEGIN
4.       ARRAY DiamondPattern = [ ... data ... ];
5.       FOR index = 0 to 7 DO
6.           BEGIN
7.               MBR-IN := DiamondPattern[index] ** at TIME-1
8.               MUX-A := 0       ** at TIME-1
9.               OP := ALU-NOOP   ** at TIME-1
10.              CA := 0          ** at TIME-1
11.              CLK := 0;  CLK := 1;
12.              AA := 0          ** at TEST-TIME
13.              OP := ALU-NOOP   ** at TEST-TIME
14.              CA := 0          ** at TEST-TIME
15.              MUX-A := 1       ** at TEST-TIME
16.              CLK := 0;  CLK := 1;
17.              AA := 0          ** at TIME-2
18.              B-BUS = DiamondPattern[index] ** at TIME-2
19.              CLK := 0;  CLK := 1;
20.          END
21.  END
```

(d) The finished (merged) test program

Figure 1.17: *PF-TG Scenario (continued)*

The last step is to merge the program fragments in the order specified by the temporal variables. The end result is the test program in figure 1.17.d, which verifies that one path through the mux can transmit parallel data without any faults. Note that this program is much more readable than the equivalent test vectors in figure 1.18.

1.7 Where This Thesis Fits

This section shows where this thesis fits in the landscape of circuit testing approaches. Existing approaches fall into four broad categories:

1. **Combinational test generation algorithms**, as noted, are extremely effective with combinational circuits but are too slow to be useful with sequential circuits.

2. **In-circuit** test techniques solve the embedding problem by physically inserting probes into the circuit that can observe internal node voltages directly and control them by overriding the circuit's internal signals. This technique is extremely effective when it can be done without damaging the circuit. As circuits have gotten smaller, however, invasive testing has become more difficult and costly, and many modern circuits cannot be tested with this method (e.g., chips).

3. **Design for testability** and built-in test techniques help manage the test generation problem by accounting for it in the design process. One technique is to use extra circuitry to improve access to internal components and making it easier to embed component tests. Applied systematically, this technique can bring sequential circuits within reach of the existing combinational test generation algorithms.

   Circuits designed with testing in mind are increasingly common, but they nowhere near the norm yet. Moreover, the circuitry added to facilitate testing reduces performance and raises cost, so designers will sometimes want to avoid these penalties by using other testing methods that do not impose them.

4. **Expert test programming** covers the remaining circuits.

The fundamental problem that all of these techniques solve is how to access components internal to a circuit from the outside. Techniques 1 and 4 solve this problem by searching for ways to gain access by using the circuitry surrounding the component. Techniques 2 and 3 solve the problem by bypassing the surrounding circuitry. In

```
** CLK RF-AA RF-BA RF-CA MBR-IN AMUX OP B-BUS
** ------------------------------------------
   0   0000  0000  0000  0000   0    01 0000
   1   0000  0000  0000  0000   0    01 0000
   0   0000  0000  0000  0000   1    01 0000
   1   0000  0000  0000  0000   1    01 0000
   0   0000  0000  0000  0000   1    01 0000
   1   0000  0000  0000  0000   1    01 0000
   0   0000  0000  0000  1000   0    01 0000
   1   0000  0000  0000  1000   0    01 0000
   0   0000  0000  0000  1000   1    01 0000
   1   0000  0000  0000  1000   1    01 0000
   0   0000  0000  0000  1000   1    01 1000
   1   0000  0000  0000  1000   1    01 1000
   0   0000  0000  0000  1100   0    01 1000
   1   0000  0000  0000  1100   0    01 1000
   0   0000  0000  0000  1100   1    01 1000
   1   0000  0000  0000  1100   1    01 1000
   0   0000  0000  0000  1100   1    01 1100
   1   0000  0000  0000  1100   1    01 1100
   0   0000  0000  0000  1110   0    01 1100
   1   0000  0000  0000  1110   0    01 1100
   0   0000  0000  0000  1110   1    01 1100
   1   0000  0000  0000  1110   1    01 1100
   0   0000  0000  0000  1110   1    01 1110
   1   0000  0000  0000  1110   1    01 1110
   0   0000  0000  0000  1111   0    01 1110
   1   0000  0000  0000  1111   0    01 1110
   0   0000  0000  0000  1111   1    01 1110
   1   0000  0000  0000  1111   1    01 1110

   <15 lines suppressed>

   1   0000  0000  0000  0001   0    01 0011
   0   0000  0000  0000  0001   1    01 0011
   1   0000  0000  0000  0001   1    01 0011
   0   0000  0000  0000  0001   1    01 0001
   1   0000  0000  0000  0001   1    01 0001
```

Figure 1.18: *These test vectors are equivalent to the test program in figure 1.17.d. In this tabular representation, the structure of the test is difficult to see.*

technique 2, the tester circumvents the surrounding circuitry by physically inserting probes into the circuit, while technique 3 depends upon the designer including extra circuitry whose only purpose is to enable this bypass to occur.

The test generation techniques described in this thesis are targeted at category 4, i.e., at circuit testing problems that are currently solved by hand. This category is now and will continue (I believe) to be an important place to concentrate research effort. Tests for a significant percentage of all circuits are currently generated by hand (see figures 1.19 and 1.20). While these percentages should decrease in the next five years as design for testability (DFT) standards are adopted by industry, they are likely to remain significant because DFT standards often involve performance and cost penalties. Moreover, testing experts will constantly playing catch up to the latest design technologies, creating a time lag within which hand-written or human-assisted tests are needed. Barring the discovery of an effective in-circuit technology for small, high density circuits, the future of circuit testing will depend on a combination of advances in DFT and expert test generation.

1.8 Summary

This thesis makes several contributions to the field of circuit testing:

- It introduces operation relations, a representation of circuit behavior that often makes embedding problems easy.
- It describes a way to compute operation relations for sequential circuits by symbolic simulation. This method is efficient for circuits that offer a small number of operations at their interface.
- It introduces the designed behavior heuristic, i.e., test a circuit without going outside its normal operations, clarifies the issues surrounding this heuristic and provides empirical evidence that the heuristic is useful.
- It describes an automated method of creating test programs by combining test program fragments.
- It demonstrates how propagating typed streams of values can produce more efficient tests and introduces a vocabulary of stream types.
- It extends the goals of test generation to include using the capabilities of the tester well. To achieve this goal, PF-TG uses an explicit description of tester capabilities and resource limitations.

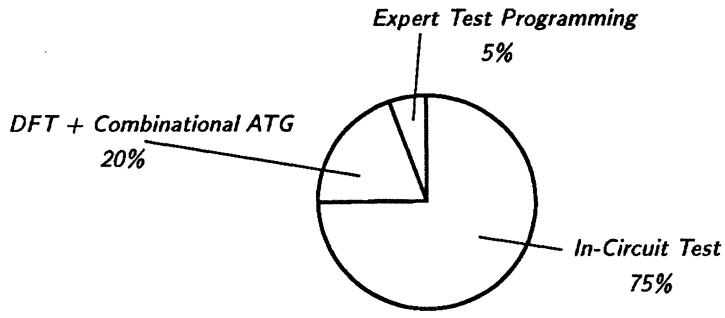


Figure 1.19: *Percentages of boards handled by various testing techniques*

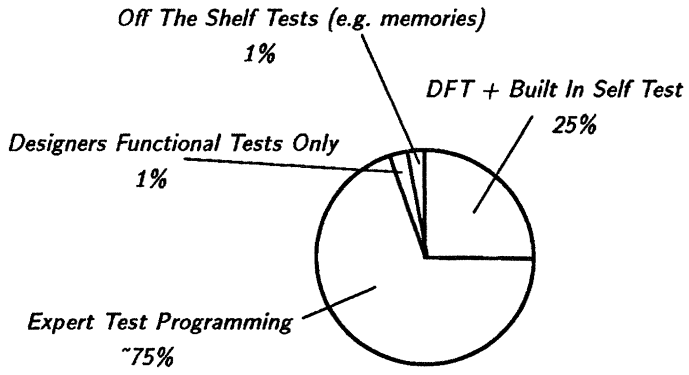
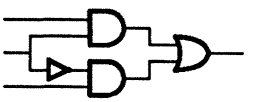


Figure 1.20: *Percentages of chips handled by various testing techniques*

DB-TG and PF-TG are two novel test generators that extend the range of techniques available to test engineers. DB-TG and PF-TG are complimentary: the first is effective on complex sequential circuits that display tightly interacting component behavior. Search in DB-TG is indexed and guided primarily by what is possible for the circuit to do rather what is desired to test a component. However, the cost of simulating circuit operations renders DB-TG inefficient for circuits where many operations are possible. PF-TG uses conventional goal-directed planning techniques, and is targeted at simpler sequential circuits. In PF-TG, search is indexed and guided primarily by specific testing goals, i.e., how to test a component, and is not limited by the number of circuit operations.

PF-TG's use of conventional planning techniques provides a testbed for experimenting with several other aspects of test generation. For instance, PF-TG produces test programs rather than test vectors to raise the level of the language between the test generator and the agent that will carry out the test. Using this richer language, together with using a simple model of the tester capabilities, helps PF-TG to design more efficient ways to test a circuit.



Chapter 2
Background I: Testing Theory

Summary: This chapter introduces the basic concepts and algorithms in the field of circuit testing. Section 2.1 gives an overview and answers fundamental questions like what is a test and how are they used? Section 2.2 covers the key issue of modeling circuits and faults for use by a test generation algorithm, and section 2.3 briefly presents the algorithms that form the core of testing theory.

Unfortunately, the need for test generation far outstrips the capabilities of existing algorithmic theory. This need is currently met by the application of human intelligence: for complex, sequential circuits, the experts are much more successful than any existing test generation algorithm. Chapter 3 describes how experts are actually solving problems now.

Chapters 2 and 3 serve two purposes. First, they bring up to speed readers who are unfamiliar with circuit testing. Second, they highlight differences in how the algorithms and the experts solve test generation problems. This thesis is prompted by these differences and the performance gap that results. Studying the problems people must solve (because the algorithms do not) and how they solve them supplies clues for improving the algorithms and closing the gap. Readers with circuit testing backgrounds may want to skip to the end of the next chapter for a summary of the differences that form the basis for the test generation methods introduced later in this thesis.

2.1 What is Circuit Testing?

Testing is a essential part of the process of designing and manufacturing circuits. Its primary objective is to detect physical faults resulting from the manufacturing process or from actual use, i.e., to detect discrepancies between a circuit design and its physical realization. A secondary objective is to locate faults precisely enough that the circuit can be repaired.

Figure 2.1 shows the basic connections between design, testing (test generation and application) and manufacturing. The boxes in the figure represent tasks to be

performed and the items between the boxes represent objects to be created or manipulated. Each task places constraints upon the others. For instance, the kinds of faults introduced by the manufacturing task determine what the test generation task has to accomplish. This thesis is about test generation, but first we briefly describe the other tasks that bear upon it.

Circuit Design

A circuit designer starts with an informal specification of the behavior desired and other requirements (e.g., cost) that the circuit must meet. The designer then decides how to implement this specification by selecting, connecting and modifying standard components, analyzing and optimizing the circuit's performance properties (e.g., speed and power consumption) and using any of the literally dozens of other methods of transforming the specification into something that can be manufactured directly. The end result is called a circuit **design description**, i.e. a description of the circuit as it should be manufactured. For our purposes, the important aspects of the design task are its input and output: the specification tells what an instance of the circuit has to do functionally and the design description tells what an instance of the circuit has to be physically.

Circuit Manufacturing

Figure 2.2 illustrates the steps of circuit manufacturing. The top portion of the figure shows the design description. The schematic will be used for test generation. The masks are like photographic slides. They correspond to the schematic and additionally specify the physical arrangement of materials in the chip. The assembly drawings also correspond to the schematic and additionally specify the physical arrangement of chips on a board.

The left-hand column shows the steps of chip manufacturing. First, a photolithographic process uses the masks, each of which corresponds to a layer of material on the chip, to successively lay down and etch away material forming complex, overlapping patterns on the surface of the chip.[1] For efficiency, these photographic and chemical processes are done on large wafers of silicon to form dozens of copies of the chip simultaneously. After separating the wafer into individual chips, the chips are put into ceramic or plastic packages to protect them during handling and to simplify connecting the microscopic chips with the macroscopic world.

The right-hand column shows the steps of board manufacturing. First, wires are laid down on the board in much the same way materials were patterned onto the

[1] For more detail, see for example [mead80].

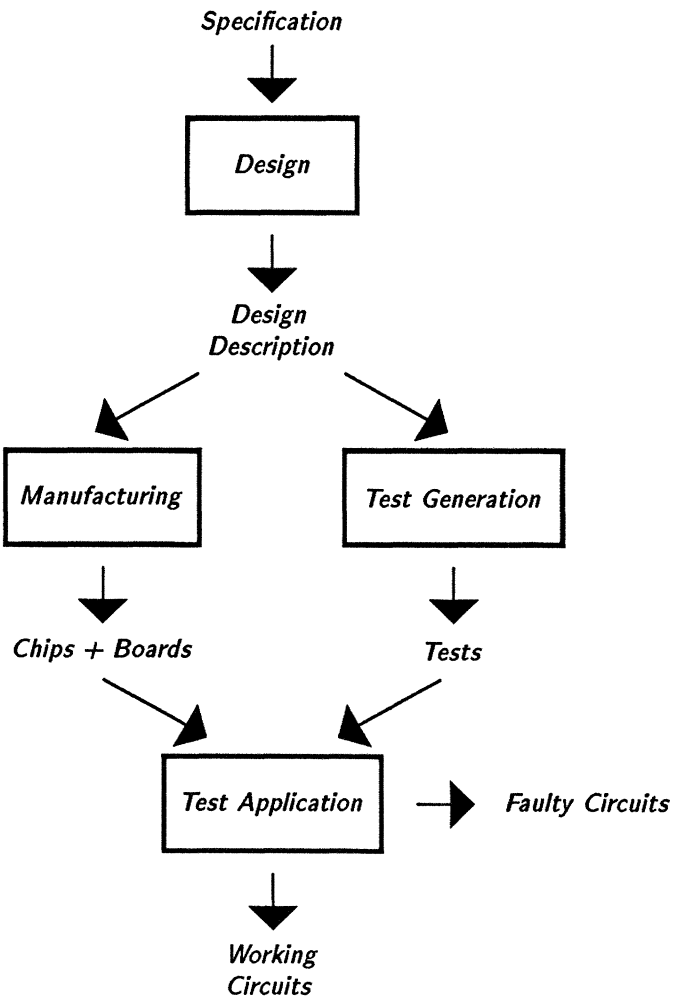


Figure 2.1: *This figure shows the basic connections between design, testing (test generation and application) and manufacturing. The boxes represent tasks to be performed and the items between the boxes represent objects to be created or manipulated.*

chip. Then holes are drilled into the board to hold the chip packages. These holes are arranged to place the wires coming out of the chip packages next to the appropriate wires on the board. When the board has been fully populated with chips, they are all physically and electrically bonded to the board at once in a process called wave soldering. This process holds the back side of the board close above a vat of molten metal and passes a wave in the liquid surface under the board. As the wave passes, some metal sticks to the chip wires and cools there, connecting them to the wires already on the board. The finished board can then be put into a cabinet with other boards to form a complete digital system.

If carried out imperfectly, any of these manufacturing steps can introduce faults into the chip or board that will prevent it from working properly. The different steps introduce different kinds of faults which must be tested for in different ways.

Test Application

A circuit board or chip is tested by applying stimuli to the circuit and verifying that its responses are correct (see figure 2.3). Correct responses are evidence that the circuit is fault-free, and the strength of this evidence depends upon how well the stimuli were designed. When a response is incorrect, we say a fault has been detected and we reject the circuit. The way the circuit fails can provide clues to the nature of the fault or faults inside, and the strength of these clues also depends upon how well the stimuli were designed.

The major issues in test application are test application speed and accessibility of internal components. Applying tests quickly is important because many circuits must be tested. Accessibility, e.g., finding technologies that allow control and observation of a circuit without damaging it, is important because increasing it generally increases test application speed and simplifies test generation.

Test Generation

Modern circuits are too complex to be exercised exhaustively, so a relatively small number of stimuli must be chosen which, if the circuit responds properly to them, will allow a high degree of confidence that the circuit is fault-free. Test generation is the process of selecting these stimuli. Test generation takes descriptions of a circuit design and of its potential faults and produces a set of tests designed to detect whether faults are present in an instance of the circuit. The major factors involved in doing this are:

1. **Accessibility**: Where can stimuli be applied to the circuit and where can responses be observed?

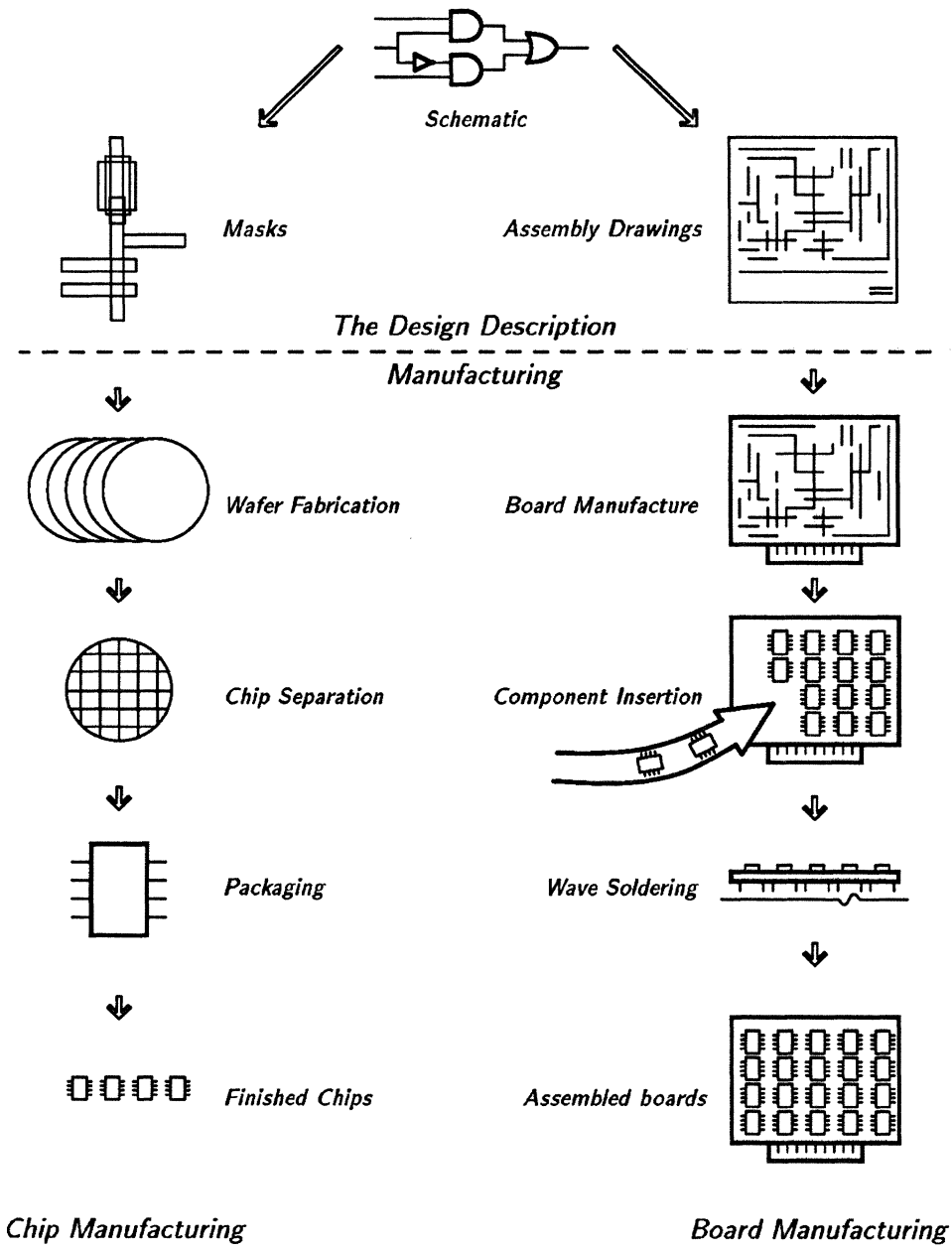


Figure 2.2: *The steps of circuit manufacturing*

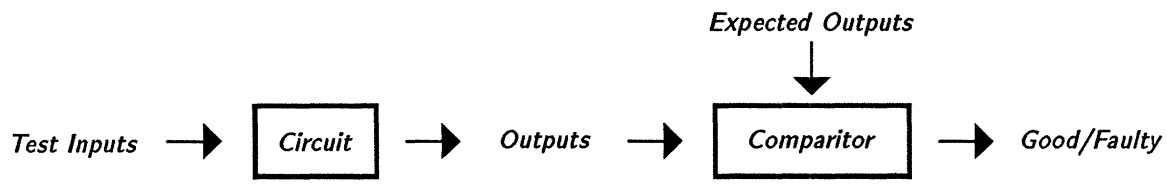


Figure 2.3: *Test Application*

2. **Complexity**: In general, the cost of test generation is exponential in the size of the circuit. Test generation strategies must manage this complexity for large circuits.

3. **Models**: How are the circuit's structure, function and potential faults modeled for the test generation program?

4. **Algorithms**: What algorithms can efficiently derive tests from the circuit and fault models

5. **Economics**: Circuit testing is fundamentally driven by economic factors, and cost provides common ground for negotiation between the design, manufacturing and testing tasks. For instance, some test generation problems are most effectively solved during the design task by making the design simpler or more amenable to test generation.

Accessibility and complexity are intimately related: limited accessibility increases the size and complexity of the portions of the circuit that must be tested as black boxes, i.e., without access to their internals. The larger and more complex these black boxes are, the more difficult it is for the test generation algorithm to select a small yet effective stimulus set. The next two sections introduce the fundamental models and algorithms developed by test generation researchers.

2.2 Modeling Circuits and Faults

Modeling the structure, behavior and faults of circuits is a central theme in test generation. The nature of the idealizations and abstractions in the models determines in large measure the cost of generating tests and their effectiveness when applied to real circuits. The following discussion introduces the notions of circuit models,

physical faults, behavioral errors and fault models. With this foundation, we can then examine several test generation techniques.

2.2.1 Circuit Models

A **circuit model** is an idealized description of a physical circuit in which some aspects of the physical reality (e.g., physical arrangement) have been ignored in order to make other aspects (e.g., electrical topology) more apparent. Figure 2.4 shows several kinds of circuit models for digital computers. Each level in this table is an idealization of the ones below it, and each describes a circuit using a different vocabulary. For example, a register transfer circuit model describes a circuit in terms of registers, boxes that compute arithmetic functions, and so on.

Modeling Level	*Vocabulary*
Instruction Set	arithmetic, addressing, conditionals ...
Register Transfer	registers, arithmetic functions, time ...
Logic Gate Level	boolean functions and values, time ...
Switch Level	pass transistors, time ...
Circuit Level	voltage, resistance, capacitance, time ...
Process Level	physical arrangement of material ...

Figure 2.4: *Modeling Levels for Digital Computers*

In principle, any of these kinds of models can be used for circuit testing and most of them have been. Choosing the model is important for test generation because it determines both the vocabulary for describing behavioral interactions between components and faults and the vocabulary for describing faults.

Most work on test generation for digital circuits has modeled circuits at the logic level as networks of boolean gates connected by ideal wires. However, many kinds of abstract, high-level models have been also used, e.g., register transfer models [shirley85], ISPS behavioral models [khorram84], petri-nets [lai81] and mixed gate-level schematics and state transition diagrams [hill77]. One recent line of research [genesereth81, davis82a, shirley83b, singh86, krishnamurthy87] has focused on using hierarchical circuit models which describe a circuit at multiple different levels of detail. Extremely detailed circuit models also exist, e.g., analog models [gray69, spice80] and

the models of the physical arrangement of silicon layers in a chip, but these models have mainly been used to analyze circuit performance rather than to generate tests.

2.2.2 Faults prevent a circuit from meeting its specification

So far, we have been using the term "fault" loosely to refer to a problem with a circuit. More precisely, a **fault** is a physical defect in a circuit that prevents it from meeting its specification, e.g., wire breaks, shorts between two wires, chips inserted backwards, or resistors with the wrong value. Physical faults can be subdivided into **logic** and **parametric** faults. Logic faults cause a portion of the circuit implementing one logic function to behave like a different logic function. For example, an inverter might misbehave and act like a buffer. Parametric faults involve deviations from their acceptable ranges of circuit parameters as voltage, capacitance and speed. Figure 2.5 shows some of the more common faults that can be introduced into a circuit during construction and use.

Manufacturing Step	*Potential Faults*
Chip Fabrication	silicon impurities, mask blemishes; over or under etching; silicon, metal or chip-to-package open-circuit or short-circuit defects,
Board Fabrication	incorrect interconnections; open-circuit or short-circuit defects; unintended crosstalk between adjacent wires; power supply defects; susceptibility to external electrical noise
Construction	incorrect IC packages; packages inserted backwards; bent IC pins; extra (splashed) solder causing bridge faults between wires and pins; unsoldered joints; thermal damage to chips
Environment / Use	component degradation due to high humidity, thermal conditions or electrical noise; component aging faults (metal migration in IC's, resistor or capacitor degradation); planned modifications

Figure 2.5: *Common faults introduced during circuit construction and use*

A fault is **intermittent** if it is present during some intervals of time and absent during others. This is typically caused by unstable physical problems or by envi-

ronmental conditions. A fault is **permanent** if it is continuous and unchanging. Detecting intermittent faults early is important, because intermittent faults often become permanent as physical damage to the circuit is increases. However, no theory of how to reliably test for intermittent faults exists because they can disappear during testing. In this thesis, we consider only permanent logic faults.

2.2.3 Physical Faults have Behavioral Effects

A **fault effect** describes the consequences of a fault in the language of a particular circuit model. For example, a physical short between the emitter and collector of the transistor in figure 2.6.a will pull the output voltage down to the ground voltage.

How a fault effect is described depends upon how the circuit is modeled. For instance, to describe the effect of this short in terms of a logic gate circuit model, we would say the output of the inverter is stuck at the boolean value 0, i.e., that it cannot move from 0 no matter what the input is. In this case, the physical description of the fault is a short and its behavioral description at the gate level – the fault effect – is a called a **stuck-at-0** fault effect. With the shift to the gate modeling level, comes a shift from voltage to boolean values, and with the disappearance of ground from the model comes the introduction of a stuck-node.

A related physical fault is a broken wire between IN and the base of the transistor. In this case the fault effect would be a stuck-at-1 on the inverter output. These two possibilities, stuck-at-0 and stuck-at-1, cover the possible **stuck-at** fault effects at the boolean level and are the most commonly used fault effects in circuit testing.

A fault effect can be caused by many different physical faults, e.g., there are many physical causes for stuck-at-0 faults, but a fault causes exactly one fault effect (for a given circuit modeling language). Therefore a fault effect defines an equivalence class of physical faults – those physical faults that are indistinguishable in their behavioral effect. Much of the reasoning test generators do that is ostensibly about faults is actually about fault effects in order to save time by reasoning about equivalence classes. This shortcut is useful enough that it often gets reflected in the terminology, for example we will say a stuck-at-0 fault to mean any fault that has a stuck-at-0 fault effect.

2.2.4 Faults don't always cause Errors

An **error** is a deviation from correct behavior caused by a fault. A fault may or may not cause an error depending upon the state of the circuit. The difference between

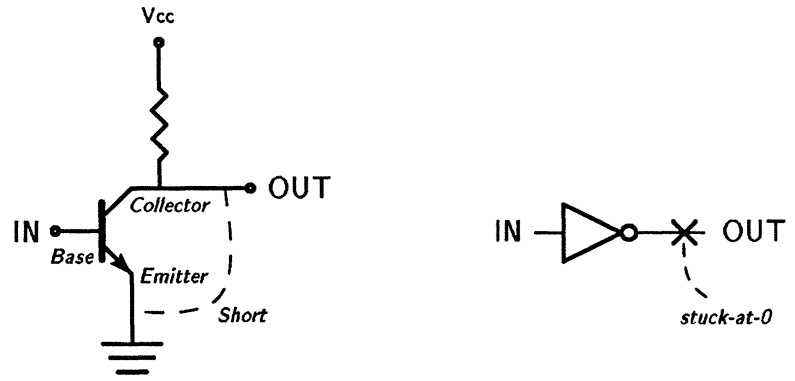


Figure 2.6: *A single physical fault (the short) and its behavioral consequence (the stuck-at-0 fault effect)*

faults and errors can be illustrated by this example from [bennetts82]:

> Consider a car carrying a spare tire, which, unknown to the driver, is flat due to a faulty valve. The driver may drive many miles before the need arises to change a tire. Until that time, the car has a fault but no error has occurred.

Figure 2.7 illustrates this relationship between faults and errors using an inverter. As before, the output of the inverter is grounded, causing a stuck-at-0 fault effect. In case (a), the input is 1, so the expected output is 0 – the same value that the fault causes. Therefore, the inverter produces the correct output and there is no error. Note that the answer is correct even though the mechanism by which it is produced is wrong.

In case (b), the input is 0 and the expected output is 1. The short causes the output to be 0, which differs from the expected value. In this case, an error occurs, hence applying 0 to the input (case b) is one way test for the presence of this short. Applying 1 to the input does not test for the short.

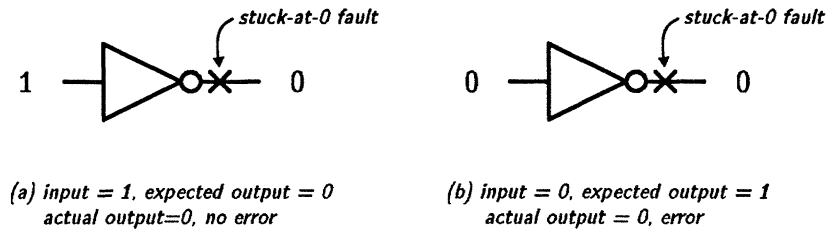


Figure 2.7: *Faults don't always cause misbehavior*

2.2.5 Errors are in the Eye of the Beholder

Depending upon conditions in the circuit, a fault may cause an error at the fault site. This error may subsequently propagate downstream in a chain of errors that eventually reaches a place (usually one of the circuit's outputs) where it is visible to an external observer. However, an error can fail to propagate too. It can be masked, again depending upon conditions in the circuit.

Figure 2.8 shows a situation where an error does not propagate to the output. The circuit is a selector whose behavior is to route either the value on D1 or the value on D2 to OUT depending upon the value of SEL. This selector has a stuck-at-0 fault on D1 and SEL is set to 0. Suppose we can only observe OUT.

The stuck-at-0 fault on D1 causes a 1 applied to that input to change to 0. Since the fault causes an error to occur inside the circuit, we have the beginnings of an effective test for the fault. However, because SEL is 0, the output of the upper AND gate is also 0. This is the correct value, hence no error occurs there and no error occurs at the output. An observer outside the circuit would say the circuit behaved properly, even though an error occurred inside.

The essence of test generation lies in figuring out under what circumstances a fault will cause an error at a place where the error can be observed. In this case, changing SEL to 1 would allow the error to propagate from the fault site through the upper AND gate and the OR gate to the output.

2.2.6 Fault Models are Closed-World Assumptions

We use a set of closed-world assumptions, called a **fault model**, to make test generation tractable by limiting the number of faults considered. Most test generation

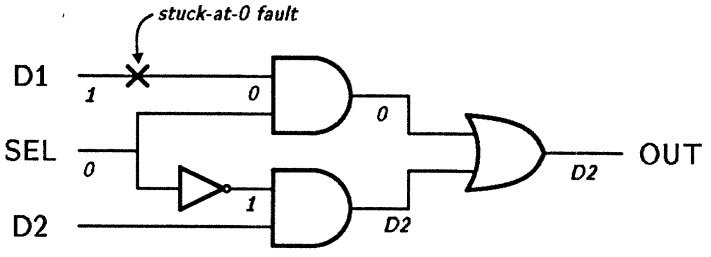


Figure 2.8: *The error at D1 is masked and does not propagate to OUT*

theory is based on the single stuck line (SSL) fault model, which is comprised of three assumptions:

1. There is at most a single fault.
2. Any physical fault causes a stuck-at fault effect.
3. Any fault is nonintermittent.

The first assumption rules out multiple faults. The second assumption rules out faults that do not manifest as a circuit node constantly holding a single boolean value, and the third assumption rules out faults whose effects change during testing. The only faults left are the stuck-ats, and we say these faults are included in the model.

2.2.7 Test Quality: Coverage and Resolution

Test quality is measured by how well the test detects and distinguishes between the faults in a circuit. A **fault list** is the list of fault effects that can occur in a given circuit under a given fault model. For example, applying the SSL fault model to a gate-level circuit with N nodes gives rise to $2N$ entries in the fault list, since each node can be stuck at 0 or at 1. If a fault causes an error that will be observed when a test is executed, then we say the fault is detected by the test. **Fault coverage** is the percentage of faults from the list detected by a given test. The higher the fault coverage, the better the test, and figures above 90-95% are considered good.

In addition to knowing whether a circuit is faulty, we sometimes want to know *how* it is faulty – which fault from the list is actually present – so we can repair the circuit. If two faults produce different outputs when a test is executed, then the test is said to distinguish or **resolve** them. A test has good **fault resolution** if it resolves most

pairs of faults from each other. This thesis is concerned primarily with production tests for VLSI circuits where repair is difficult, hence we focus on achieving good coverage and do not attempt to achieve good resolution.

Fault coverage and resolution are computed by a process called **fault simulation** that takes a model of the circuit and a fault list as input and predicts circuit outputs for each fault and for the good circuit. One way to do this is to simulate multiple copies of the circuit in parallel. When the program predicts a discrepancy between an output value of the fault-free circuit and one for a faulty circuit, the fault involved is marked as detected since an external observer would notice an error. To save work, a fault simulator usually drops a fault's simulation context immediately after it predicts the fault can be detected. The program finishes by reporting the percentage of detected faults, when and how each fault was detected and which faults were missed.[2]

2.2.8 Quality of a Fault Model

The quality of a fault model depends upon the cost of using it to generate tests and upon how well it approximates the real world. We consider these issues in turn.

The cost of using a fault model to generate tests for a circuit is roughly proportional the size of the fault list. For example, applying the SSL fault model to a gate-level circuit model produces a fault list that is linear (actually $2N$) in the number of circuit nodes.

A second fault model, called the **bridge fault model**, includes shorts between pairs of circuit nodes. A short can be cause by, for example, an oversized solder joint that is intended to connect one pin of a chip to a circuit board but also touches an adjacent pin by mistake. If all N^2 possible shorts are considered, then the bridge fault model subsumes the SSL model because any stuck node is indistinguishable from that node shorted either to power or to ground. This leads us to an important tradeoff involving fault models: the more comprehensive a fault model is, the more expensive it is to use.[3] Test engineers tend to prefer the less comprehensive SSL model because

[2]Modern fault simulators are highly optimized programs that share some ideas with AI programs that reason under multiple contexts, e.g., deKleer's ATMS [dekleer-ATMS86a].

[3]Some researchers define the SSL fault model to include stuck terminals, i.e., a wire break disconnecting a single component input from the circuit node that drives it. The behavioral effects of this kind of physical defect differ from those of a stuck-at if the driving node includes a fanout (i.e., a single output driving multiple inputs). The bridge fault model does not subsume an SSL model that includes stuck terminals, but the gist of the comparison between the SSL model and the bridge fault model still holds.

the resulting fault list is only linear in the circuit size.[4]

In principle, how well a fault model approximates faults in the real world can be determined by measuring what kinds of faults occur in practice and how often each kind occurs. For instance, if 95% of the faults that occur in practice were stuck-ats, then we would say the stuck-at model is a good approximation. Unfortunately, collecting a statistically significant number of samples is difficult because each fault must be carefully analyzed and categorized. Moreover, this process must be repeated often to track the frequent changes in circuit manufacturing methods.

A more practical method of judging a fault model's quality is to compare it against a more detailed and physically plausible model. [ferguson87] describes measurements for a set of physically plausible, fabrication faults in Metal Oxide Semiconductor (MOS) circuits. Only 45% of the faults corresponded to stuck-at fault effects. The rest were bridges and transistor defects of various kinds.

Conventional wisdom in the field has it that, while not all faults result in stuck-ats, tests generated for stuck-at fault effects are good enough. This conclusion has been called into question by Ferguson's results. A test that could detect 100% of the stuck-ats detected less than 90% over all of the physically plausible faults.

These results suggest that test generation algorithms should be flexible in the fault models they assume. Different technologies will have different kinds and distributions of physically plausible faults, and factoring the fault model out of the test generator will help it remain useful as technologies change.

2.2.9 Summary

This section introduced faults, fault effects and errors. Physical faults have behavioral consequences called fault effects that prevent a circuit from meeting its specification. Whether a fault actually causes an error and the error propagates to a place where it can be observed depends upon conditions in the circuit. Setting up the conditions properly is the goal of test generation.

Modeling is a central theme. The most commonly used type of circuit model for test generation is a network of boolean gates. This choice has largely been dictated by the kinds of circuit descriptions produced by existing design tools. Only recently has the rising cost of generating tests using gate level descriptions caused the use of gate-level models to be seriously questioned in industry.

[4] Also, a short can complicate matters by creating a feedback loop to turn a combinational circuit into a sequential one.

Fault models are sets of closed-world assumptions. The most commonly used fault model for test generation is the single stuck line (SSL) fault model, which was developed for 1960's board manufacturing technology. Chip manufacturing technology has changed the kinds of physically plausible faults sufficiently that the SSL fault model too has recently come under scrutiny. Still, this fault model remains the standard in industry.

The following two sections introduce test generation and test application techniques that have been developed for digital circuits. They briefly cover how the tests are generated from the circuit and fault models and how the responses are predicted, measured and compared.

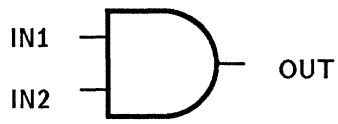
2.3 Generating Tests

Test generation takes a circuit model and a fault model as input and attempts to produce a set of exercises that will detect and optionally locate any of the faults covered by the fault model in an instance of the circuit. Each test specifies signals to be applied to the circuit's inputs and values to be observed at its outputs.

2.3.1 Representing Tests as Vectors

Figure 2.9.a shows a test for an AND gate. Each row of the table is called a **test vector** and describes two inputs to apply and one output to observe. To apply these tests, one steps through the rows of the table applying the inputs and looking for the expected outputs. Observing an output value that is different from one in the table indicates the presence of a fault. The term "test" is used variously to refer to a single test vector or to a group of them that share a common purpose, like the ones in the figure. Since the behavior of a combinational circuit is independent of its past inputs, each test vector is independent from the rest, and a set of vectors can be applied to a combinational circuit in any order.

Test vectors can be applied to a circuit automatically using a specialized piece of hardware called a **stored-pattern tester** (see figure 2.10). A stored-pattern tester is a computer with a large memory for holding test vectors and with special-purpose, parallel I/O electronics for driving circuit inputs and sensing circuit outputs. The computer first fills the memory with test vectors. Then the vectors are read out from the memory one-by-one. The electronics that interface the tester to the circuit, called driver-sensors, convert the input values to appropriate voltages and drive them onto the circuit inputs. Then they sense voltages on the circuit outputs, convert them into



IN1	IN2	OUT
0	0	0
1	0	0
0	1	0
1	1	1

(a) exhaustive tests (black box)

IN1	IN2	OUT
1	0	0
0	1	0
1	1	1

(b) SSL tests (model based)

Figure 2.9: *Two different tests for an AND gate*

boolean values and compare them with the expected outputs specified by the vector. If there are any discrepancies, the tester raises a flag to the human operator indicating the presence of a fault.

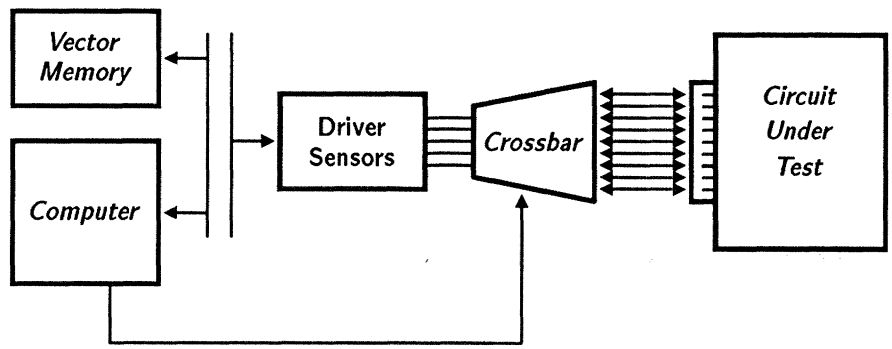


Figure 2.10: *A Stored Pattern Tester*

Classical testing theory views tests as vectors and testers as machines for applying vectors. The present-day reality is considerably more elaborate than this, as we shall see in chapter 7.

2.3.2 Test Generation Methods

This section describes two fundamentally different strategies for generating tests: **black box** test generation and **model-based** test generation. Black box test gener-

ation only requires information about the circuit's interface with the external world, i.e., its I/O behavior. The model-based approach exploits information about the circuit internal structure and behavior in order to produce higher-quality tests.

2.3.2.1 Black Box Test Generation

Black box test generation relies solely on descriptions of a circuit's interface. The two major forms of black box testing are random testing and exhaustive testing. In random testing inputs are selected at random from the set of possible circuit inputs. With this approach tests can be generated extremely cheaply and can be applied to a circuit as shown in figure 2.11. The pattern generator generates pseudo-random numbers and applies them simultaneously to the circuit under test and a reference standard that has been tested using another method, e.g., by hand. Any discrepancy between the outputs indicates the presence of a fault in the circuit under test.

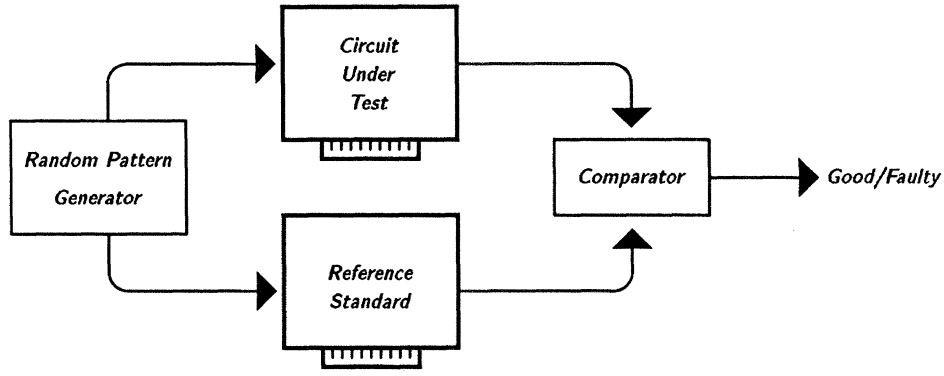


Figure 2.11: *Random Testing*

Exhaustive testing is the extreme case of black box test generation where one guarantees that all input combinations will be applied to the circuit. Therefore, all hard faults will be detected – a conclusion that does not require expensive analysis. The tests in figure 2.9.a are exhaustive, since they cover all four input combinations.

The primary disadvantage of both methods is the large number of vectors required to achieve good fault coverage. Random testing requires applying a large percentage of the circuit's possible inputs, and the more systematic approach involves applying all of them. Exhaustive tests for a combinational circuit with n inputs can require 2^n test vectors, one for each input combination. Exhaustive testing is, however, quite effective for small circuits since few vectors are needed and they are cheap to generate.

2.3.2.2 Model-Based Test Generation

Many modern circuits are too complex to test using black box methods. Using a model of the circuit and its faults, however allows a test generation program to produce high quality tests with relatively few vectors. The models allow the problem of testing the whole circuit to be divided into separate problems of testing each element of the model. For example, a model-based test generator can design one test vector for each SSL fault in a circuit. For the AND gate in figure 2.9.b, this yields 6 test vectors. Since a single test vector can usually detect several faults, eliminating redundant vectors reduces the number of vectors to the 3 shown in the figure. These 3 test vectors detect SSL faults on the inputs and the output of the AND gate using fewer vectors than the 4 required by an exhaustive test. Since the number of SSL faults rises linearly with circuit size while the number of exhaustive tests rises exponentially with the number of inputs, using circuit and fault models for large circuits can save significant effort and time.

There are two different strategies for model-based test generation: (i) check each behavior and (ii) check each fault. The strategies partition the test generation problem in different ways. The first strategy, known as **functional** testing, partitions the circuit by what it does. For example, a functional test for a microprocessor involves tests for each instruction. The second strategy, **fault-based** testing, designs a test for each potential fault in a model of the circuit.

A single test generation algorithm can combine the methods of checking behaviors and checking faults. For example, the test for a microprocessor might be partitioned by instruction, like a functional test, but the test for each instruction might then be generated using models of faults in the components used by that instruction.

Most of the recent work on test generation and both of the new methods introduced by this thesis fall within the model-based test generation framework. What distinguishes the different methods from each other is how circuits and faults are modeled and whether the testing problem is partitioned by function, fault or both. In order to make the discussion concrete, the next section describes a specific model-based test generation algorithm.

2.3.3 The D-Algorithm

The D-algorithm [roth66, roth80] is a test generation method for gate level circuits. This section shows the circuit and fault models traditionally used in test generation and the methodical and exhaustive search that is the foundation for generating tests.

The next section describes some of the search control heuristics and other methods of making test generation more efficient.

The D-algorithm generates tests for all SSL faults in a combinational circuit. This algorithm, the ancestor of most model-based test generation methods (e.g., [goel81a, benmehrez83, kramer83, fujiwara85]), illustrates three central ideas in test generation: (i) fault hypotheses, (ii) a symbolic vocabulary for signal flow and (iii) path sensitization.[5]

2.3.3.1 Hypothesizing faults

The D-algorithm generates one test vector for each potential SSL fault in the circuit. It does this by repeatedly hypothesizing the existence of a fault (e.g., that a single node is stuck-at-0) and then generating a vector which distinguishes a circuit containing that fault from a fault-free circuit. The discussion below assumes that a particular fault has been hypothesized.

2.3.3.2 A Symbolic Vocabulary for Signal Flow

The D-algorithm uses the extended boolean vocabulary for signal flow originally used by the D-algorithm. The D-vocabulary combines the fault-present case with the fault-absent case into single tokens in the following way:

Original Notation	*good/bad*	*Meaning*
1	1	boolean 1 in either case
0	0	boolean 0 in either case
D	1/0	1 if fault-free, 0 if faulty
$\overline{D}$	0/1	0 if fault-free, 1 if faulty
x	x	don't care

1/0 and 0/1 are called *sensitive values* because their value is sensitive to the presence or absence of the fault.[6] By combining the fault-present and fault-absent cases into a

[5]This section describes a slightly simplified version of the algorithm. The full version does not focus so closely on SSL faults. Rather, it has several complex mechanisms enabling it to test faults inside gate-level components in addition to the SSL faults on the circuit nodes. These mechanisms add considerable complexity to descriptions of the algorithm and are not of central concern here.

[6]The D-vocabulary is usually presented using the symbols D for 1/0 and $\overline{D}$ for 0/1. We use good/bad notation here to enhance readability.

single notation, the D-vocabulary makes reasoning about the propagation and interaction of errors in a circuit an analog of the familiar process of propagating boolean values.

The D-algorithm assigns sensitive values to circuit nodes following detailed rules dictated by the circuit structure and behavior. The goal of this process is to assign a sensitive value to one of the circuit outputs. When the real circuit is tested these sensitive nodes will either be 0 or 1 depending upon whether the fault is present, and measuring the value of the sensitive output will tell which situation holds.

Some of the rules governing node assignments stem from keeping them consistent with component behavior. Figure 2.12 shows a complete set of so-called D-rules that use the D-vocabulary for node assignments around an AND gate. The rules in group (a) describe the gate's normal behavior, i.e., its behavior in the absence of any faults inside the gate or errors on its inputs. The rules in group (b) describe a fault in the gate causing an error. In this case, the gate inputs are normal, fault-free boolean values and the output is a sensitive value. An output value of 0/1 indicates a stuck-at-1 fault while an output value of 1/0 indicates a stuck-at-0. The rules in group (c) describe how errors propagate through AND gates. The top four rules in group (c) show single errors propagating through the gate, while the bottom four rules show how pairs of errors (stemming from the same fault) interact. For instance, the sensitive values 1/0 and 0/1 on the inputs cancel out causing an insensitive 0 on the output because one or the other input is 0 in both the good and bad cases.

Within this framework, test generation is a process of choosing rules for the components to describe how the components will behave during a test. The rules chosen must agree where the components touch. For instance, if gate A drives gate B, then the output value of A's rule must agree with the input value of B's rule because the circuit node shared by both rules can only have a single value. The strategy for choosing rules to form a test is called path sensitization.

2.3.3.3 Path Sensitization

Path sensitization starts with each circuit node assigned to the don't care value, then changes node assignments to one of the other four values as necessary. The method involves three steps:

1. **Fault Sensitization**: Choose a way to make the fault cause an error at the fault site, i.e., choose assignments corresponding to a gate receiving correct inputs but having a potentially faulty output. If, for example, the hypothesized

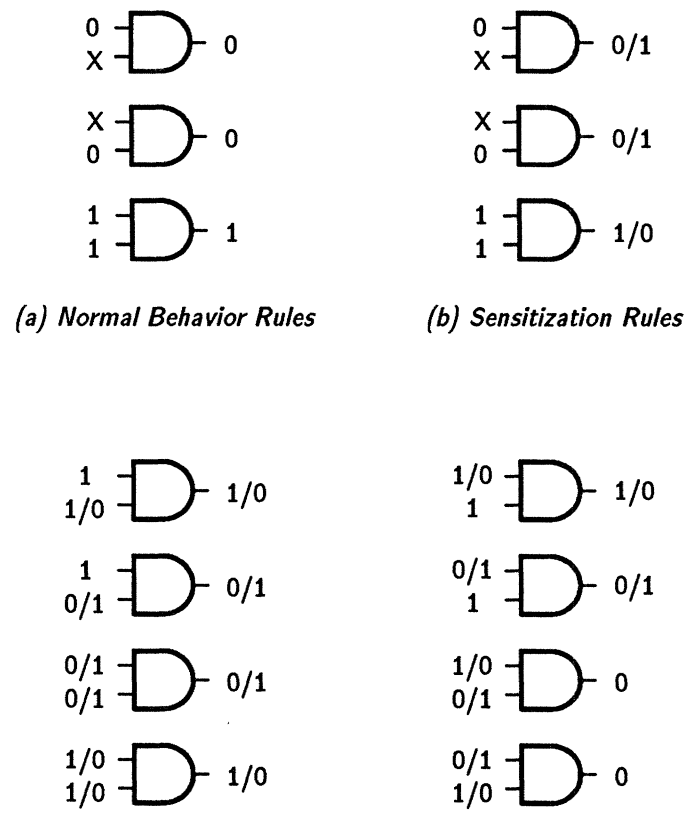


Figure 2.12: *D-rules for AND gates. A values of the form good/bad describe a circuit node whose value is different depending upon the presence of a fault. For instance, 1/0 indicates a node that is 1 if the circuit is good and 0 if it is faulty.*

fault is stuck-at-0 at a node, then assign the node a value of 1/0, i.e., the node's value should be 1 in the good circuit case and 0 otherwise. Then choose one rule that has 1/0 on the output from group (b) for component driving the node.

2. **Error Propagation**: Select a path to propagate the error signal from the fault site to an output. In effect, the path acts as a probe connecting the fault site to an output, thereby enabling an external observer to measure the node value inside the circuit. Select sensitive values for the nodes along the path and behavior rules from group (c) for the components along the path. Assign values to nodes adjacent to the path according to the rules chosen for the components in the path. These adjacent node assignments are conditions enabling the components to propagate the sensitive values from fault site to output.

3. **Line Justification**: Select values for the circuit inputs that will cause the values assigned in steps (1) and (2) to occur. This step uses behavior rules from groups (a) and (c).

All of these steps involve search. For example, there may be several ways to sensitize the fault, many ways to propagate the error and many ways to justify line values. Only some of these ways will be mutually consistent. The search involved in finding consistent ways of doing these steps is the main cost of running the D-algorithm and of doing test generation in general. This search is usually organized as constraint propagation from the fault site outward: forward to the circuit outputs during error propagation and backward to the circuit inputs during line justification.

2.3.3.4 The D-Algorithm and An Example

Path sensitization as described has a flaw: there can exist testable faults that are impossible to test using only a single sensitized path. The D-algorithm's major contribution over previous path sensitization techniques is a method of organizing the search for multiple sensitive paths so that no possibilities are overlooked. Thus, the D-algorithm was the first complete test generation algorithm: if a fault can be tested (given the circuit and fault models), then the D-algorithm can find a test for it.

The D-algorithm, shown in figure 2.13, uses chronological backtracking search over the space of D-rules with constraint propagation performed after every choice. AI readers should note that the D-algorithm used a clear though limited version of constraint propagation in 1966, long before Sussman and Stallman's EL program.

1. **Fault Selection**: Select a fault from the fault list. Here, we take the faults to be stuck circuit nodes.

2. **Fault Sensitization**: If the fault selected is stuck-at-1 (stuck-at-0), then attempt to set the node to 0 (1) by selecting inputs for the gate driving the node. Assign 1/0 (0/1) to the fault site.

3. **Constraint Propagation**: In step 1, some nodes may be assigned values that uniquely imply values on other nodes. When a node is assigned a value, put all gates that use that node either as an input or an output on an active list. During constraint propagation, remove gates from this list. If the assignments around a gate are inconsistent with all of its D-rules, then backtrack. If the surrounding assignments are consistent with exactly one D-rule, then assign new values to any unassigned nodes as specified by the rule. New assignments will cause gates to be added to the active list. Continue until the active list is empty.

4. **Error Propagation**: Drive the error out to an observable node. If any primary output has a D value, then go to step 4. The **D-frontier** is the set of gates which have sensitive values on their inputs but whose outputs are unassigned. The D-frontier is updated when assignments are made.

   Select a gate from the D-frontier and a D-rule consistent with the current node assignments and which has a sensitive value on the output. Perform constraint propagation for any new assignments. Repeat until a sensitive value appears on a primary output.

5. **Line Justification**: Construct a causal justification for all node assignments made in the steps above. A node assignment is *justified* if it is a primary input or if it is caused by the input assignments of the gate driving it.

   Select a node that is not justified. Choose a consistent D-rule for the driving gate, and make the new assignments. Perform constraint propagation. Repeat until all nodes are justified.

Figure 2.13: *The D-algorithm*

Figure 2.14 shows a simple example of the D-Algorithm in action. The first step is fault selection. Here we have chosen a stuck-at-1 fault on the output of gate A (figure 2.14.a). Fault sensitization assigns A's upper input to 0 (figure 2.14.b), although it could just as well have chosen the other input. This assignment will cause the gate's output to be 0 if fault-free and 1 if stuck-at-1.

Constraint propagation determines the consequences of the assignments made so far. Figure 2.14.c shows the first constraint propagation step: a 0 on the inverter's output implies a 1 on its input. Figure 2.14.d shows the end result of constraint propagation.

Error propagation attempts to propagate the error from the fault site to an observable output. In this case, the only possible path is through O2, and the algorithm looks for a propagation rule for OR gates that has a sensitive value on the output and also is consistent with the current node assignments. No such rule can be found, because the 1 on O2's output precludes any sensitive value there. The algorithm backtracks to the most recent choicepoint, which was the selection of a way to sensitize the fault in figure 2.14.b.

This time the algorithm sensitizes the fault by assigning the lower input to 0 as shown in figure 2.14.f. Constraint propagation does not add any new assignments, and error propagation propagates assigns the output of O1 to 0 (figure 2.14.g). At this point the values of the remaining nodes are uniquely determined, and constraint propagation works them out (figure 2.14.h). The final step, line justification, has no work to do in this example, because all assignments are justified by assignments to the primary inputs. The test vector for this fault is (IN1=0, IN2=0, IN3=0, OUT=0/1).

These steps are repeated for each SSL fault in the circuit yielding 14 test vectors. Since single test vectors often may serve to detect more than one fault, there is likely some redundancy in these 14 vectors. In fact, running a version of the D-algorithm modified to account for redundancy between test vectors yields 5 test vectors for this circuit.

Sometimes physical defects cannot be detected, e.g., the stuck-at-0 fault on N's output. This fault is untestable, because sensitizing the fault requires setting IN2=1, which in turn constrains OUT=1. Since 1 is not a sensitive value, no error can now propagate to the output. The D-algorithm fails is this case by exhausting its search without finding a solution. Failing here is not a problem, however, because there is no input combination where the presence of this fault would cause the circuit to produce a wrong output.

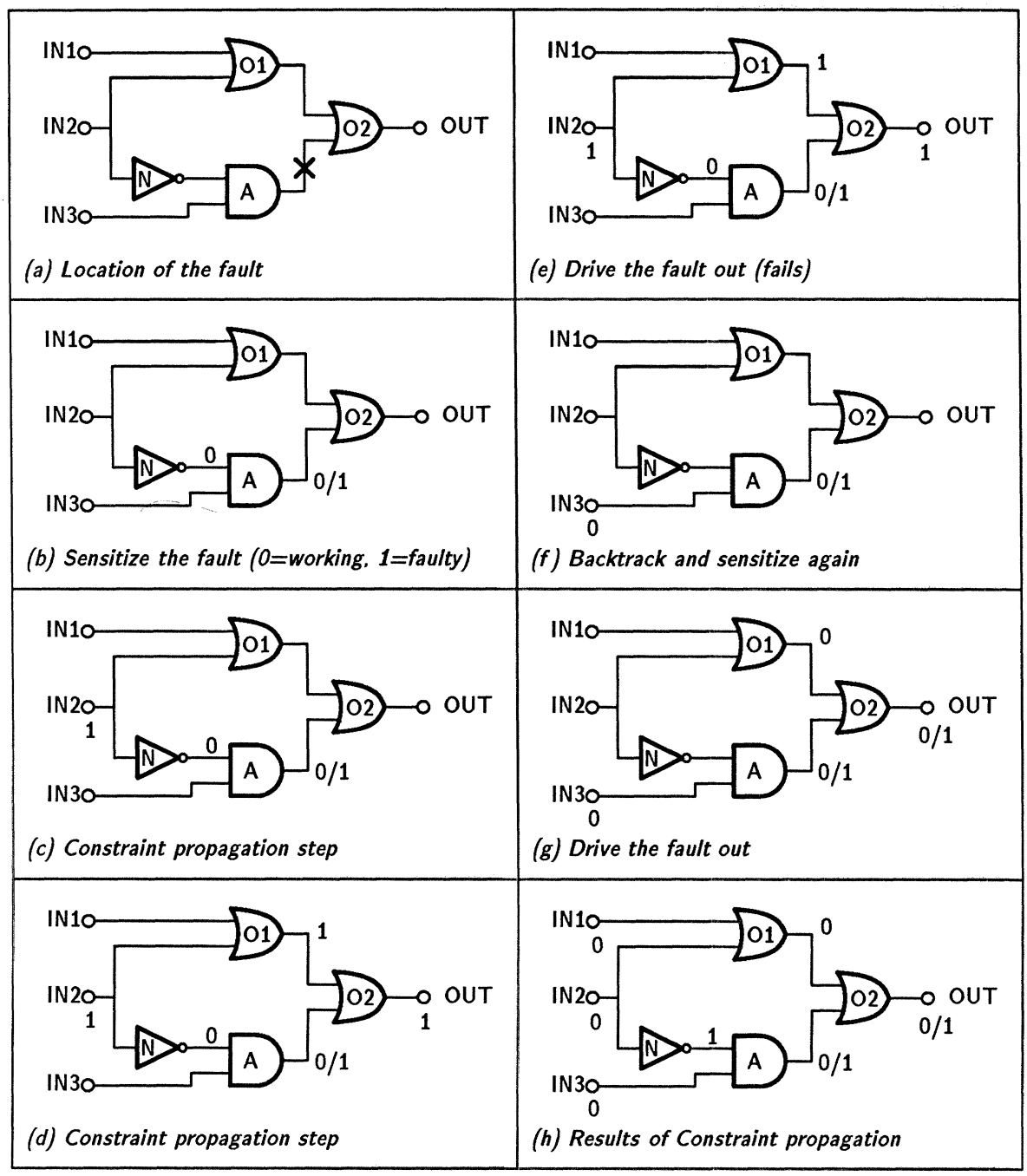


Figure 2.14: *D-Algorithm Example*

2.3.4 The Podem Algorithm

The Podem (Path Oriented Decision Making) algorithm [goel81a] is the first major successor to the D-algorithm. Podem illustrates in the testing domain two important ideas that are familiar to AI readers: (i) identification of strong constraints in the problem domain and (ii) search control heuristics. Podem is also an intellectual ancestor of the designed behavior test generator introduced in this thesis: its emphasis on behavior at the circuit inputs is a theme that we build on later.

Like the D-algorithm, Podem uses the D-vocabulary for describing circuit signals and D-rules for describing component behavior. The chief difference between Podem and its predecessor lies in the area of search control, i.e., which rules are used when. In practice, the D-algorithm's lack of a global view of circuit behavior often causes it to assign values to internal circuit nodes that are unachievable. When this occurs, the algorithm must backtrack and try again, eventually stumbling upon the correct assignments. This occurs especially when the values of the internal nodes bear complex and highly-constraining relationships to each other, e.g., in error correction circuitry. Podem uses the same kind of backtracking search, but organizes the search in a way that leads more directly to solutions or exposes conflicts more quickly.

Podem is predicated on the observation that assignments made to circuit inputs strongly constrain assignments on the internal nodes. Therefore, Podem tries hard to propagate signals back to the inputs first, before propagating elsewhere inside the circuit. In particular, Podem performs line justification immediately after making every internal assignment. If this succeeds in assigning a value to an input, then and only then does Podem take a constraint propagation step to deduce the consequences of the new input assignment on the values of internal circuit nodes. The strategy of pushing to the inputs quickly tends to expose global conflicts early, thereby reducing wasted effort.

Podem is guided toward good solutions by two fundamental search control heuristics:

1. *Conjunctive goals*: Try the hardest subproblem first, since this is likely to expose global conflicts quickly.

2. *Disjunctive goals*: Try the easiest subproblem first, since only one solution is needed.

These heuristics are instantiated in several ways in the algorithm. For instance, during error propagation, the sensitive value closest to a circuit output is pushed

forward because only one sensitive value need reach an output and the closest is likely to be the easiest to push there. Another example occurs during line justification. When using a component behavior rule requires controlling multiple inputs (e.g., setting the output of an AND gate to 1 requires that both inputs must be 1), the backward propagation proceeds first from the hardest input to control. Difficulty is approximated by distance to a circuit input. Similarly, when using a component behavior rule requires controlling only a single input (e.g., setting the output of an AND gate to 0 requires that either input must be 0), the backward propagation proceeds first from the *easiest* input to control. A rich line of recent research in test generation (e.g., [fujiwara85]) concerns refining and augmenting these heuristics and difficulty measures.

2.3.5 Test Generation with Hierarchical Circuit Models

One recent line of test generation research (e.g., [genesereth81, davis82a, shirley83b, singh86, krishnamurthy87]) has increased performance by using hierarchical circuit models. The algorithms are similar to the D-algorithm, i.e., they generally use models partitioned by structure rather than behavior, and they embed component tests using path sensitization and line justification. Their advantage comes from using high-level models to increase performance in two ways:

1. High-level models allow the test generator to take larger propagation steps with only a small increase in the cost of each step. Taking larger steps reduces the number of steps to get from an internal component to a circuit input or output.

2. Although there may be a choice of which high-level step to take, the act of taking a high-level step involves no search (just as taking a low-level step also involves no search). Taking a *single* high-level step, thus eliminates wasted search, i.e., backtracks, involved in taking the equivalent, *multiple* low-level steps.[7]

Hierarchical test generation algorithms can be described as refinements of the D-algorithm by showing: (i) how line justification and error propagation use high-level models, (ii) how the test generator shifts between modeling levels, and (iii) what strategy the test generator uses to control shifting between levels.

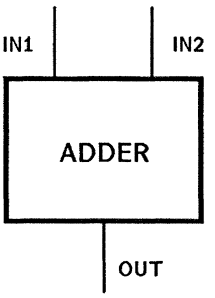
[7]This is a slight oversimplification, as some of these test generators allow search while taking a high-level step (e.g., [singh85]). However less search is generally required than in low-level propagation. The purpose of allowing search during a high-level steps is so the user may take advantage of nondeterminism to simplify writing the high-level circuit model.

First, rules for describing component behavior are augmented by high-level rules. Where D-rules describe boolean values propagating through a logic gate, high-level rules describe groups of signals representing, for example, integers propagating through an adder. One implementation [davis82a], represents high-level rules as sets of demons that activate when the value of one of their inputs changes (see figure 2.15). Techniques for doing this are detailed in [steele80]. Sensitive values are represented as pairs of values that are propagated together, one for the good circuit and one for the faulty circuit. Shifting between modeling levels is implemented with the same technique (see figure 2.15.d).

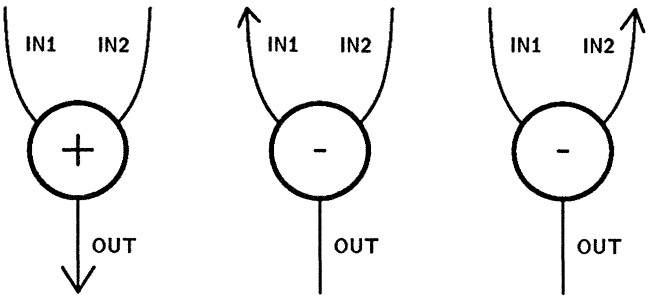
These test generators use strict hierarchical models and a simple strategy for controlling which level to propagate through: propagate at the highest level possible. This strategy assumes that high-level propagation is always less expensive than low-level propagation. These algorithms thus do line justification and path sensitization out from a fault site to the boundaries of the smallest component that contains the fault. Then they shift the values on the component inputs and outputs up one level and continue propagating to the boundary of the next containing component. The algorithms continue propagating and shifting upwards until they reach primary inputs and outputs.

Hierarchical test generation algorithms represent a significant advance over the classical algorithms, but they have several drawbacks: (i) they require abstract circuit models that are not produced or captured by current design tools[8], (ii) they have no model of the tester and (iii) they fall short of capturing the richness of expert test generation methods. On balance, hierarchical test generators are very promising, but much work remains before they can be said to work effectively in industry. The designed behavior test generator introduced in this thesis can be viewed as a step in this line of research that identifies and uses operation relations, a new kind of abstract representation for circuit behavior.

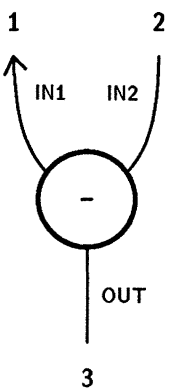
[8]The difficulty of modeling significant circuits at multiple levels of abstraction has been a hurdle in this line research. There is a chicken-and-egg problem here. These test generators require circuit descriptions that are not produced by current CAD tools, and the difficulty of creating significant test cases from scratch makes validating the research ideas difficult. On the other hand, it is difficult to justify including information a design description (and augmenting the CAD tool to handle it) that is not going to be used by a proven test generator. This problem is slowly being solved as the algorithms and the available design descriptions improve together, and efforts in industry to standardize on a rich circuit description language capable of expressing some of the needed information will accelerate this process.



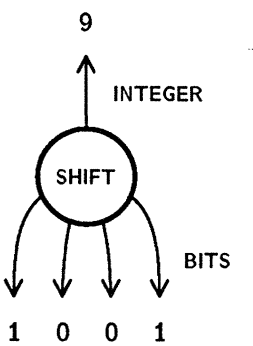
(a) structural model: one box instead of a complex, gate-level network



(b) Behavioral Model: Each rule is a function that activates when one of its inputs changes and computes a new value for its output.



(c) If IN2=2 and OUT recieves the new value 3, then this rule activates, computes 3-2 and outputs the result at IN1. If the rule cannot compute a unique output value, then it can guess among the possible values, enabling the test generator to search as well as propagate constraints. The other rules are analogous.



(d) Shifting between levels is handled by similar rules.

Figure 2.15: *High-level propagation rules*

2.3.6 Test Generation for Sequential Circuits

The D-algorithm and its successors can generate tests for combinational circuits, but they cannot directly handle sequential circuits, i.e., circuits with memory in them. This section describes a well known way to transform a model of a sequential circuit into a model of an equivalent, combinational circuit. This equivalent circuit model can then be used to generate tests for the sequential circuits using a combinational algorithm. While the transformation shows that test generation for sequential circuits is possible, testing them can be very expensive, and the form of the transformation illustrates why.

The D-algorithm is not directly applicable to sequential circuits because it models a test as occurring at an instant, whereas a test for a sequential circuit may require an interval of time, e.g., several clock cycles. For instance, to propagate an error from the fault site to the output, it may be necessary to route the error to a register, move forward by one clock cycle, and then route the error from the register to an output. In the process, signals may be routed differently in the two clock cycles. To test a given fault the D-algorithm can assign each node in a circuit at most one value, but to test a sequential circuit, each node may need a different value for each clock cycle. The D-algorithm has no place for these different values.

This problem can be solved by replicating the sequential circuit once for each clock cycle and connecting the copies to form a chain as in figure 2.16. Each copy represents the circuit during one clock cycle. A sequential circuit is composed of registers connected by combinational circuitry, and here, the R's denote copies of the registers and the C's denote copies of the combinational logic. Replacing the registers with buffers that pass information unchanged from one time step forward to the next completes the transformation from sequential to combinational circuit. The final step is to change the D-algorithm slightly to hypothesize a fault identically in all copies of the circuit, since faults are assumed to be the same at all times.

Test generation for combinational circuits is NP-complete [ibarra75] by a straightforward transformation from 3-SAT [garey79]. This means that the cost of test generation is exponential in circuit size in the worst case. In practice, however, search control heuristics and regularities in the kinds of circuits that actually are designed reduces the average cost to $O(n^3)$, where n is the number of gates in the circuit [williams79]. Hence test generation for combinational circuits is regarded as a solved problem.

Test generation for sequential circuits, on the other hand, is much more difficult. A "chain" equivalent circuit can require up to 2^m copies, where m is the number of

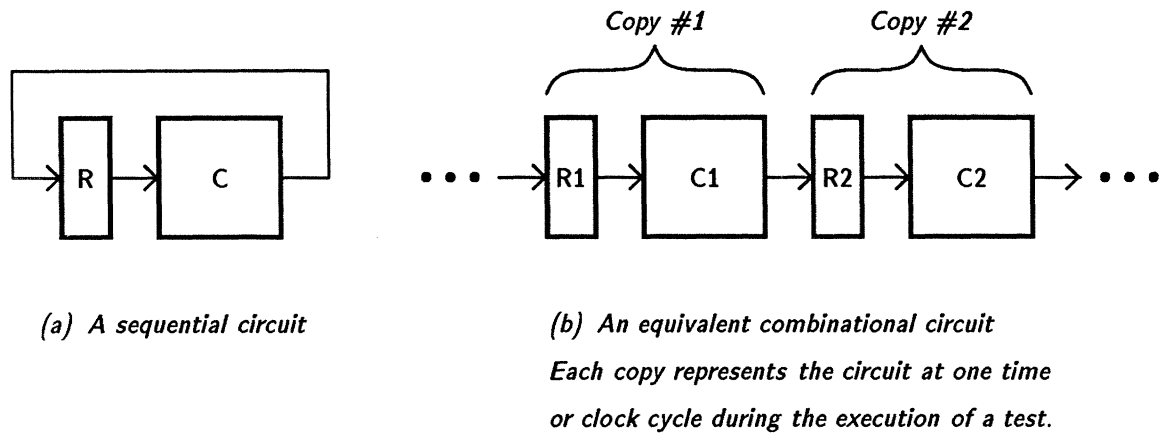


Figure 2.16: *The D-algorithm can be applied to a sequential circuit by converting the sequential circuit into an equivalent combinational one.*

bits of state, because every state may have to be visited. This yields a tight upper bound of $O((2^n)^{2^{m+1}})$ for the cost of test generation for sequential circuits [breuer76].[9]

2.4 Summary

This chapter introduced the basic concepts and algorithms in the field of circuit testing. Modeling circuits and their faults was the first major theme. Models are important because they establish the framework within which test generation occurs, i.e., faults are treated as perturbations of the circuit model. The kind of circuit model impacts the cost of doing test generation, and the kinds of closed-world assumptions made in the fault model determines how well the tests will detect real faults in the world.

The D-algorithm is a model-based test generation method that uses a gate-level model to generate tests for SSL faults. The chief characteristic of this algorithm is that it methodically and exhaustively searches through the space of circuit behaviors, making the D-algorithm complete with respect to its models. If a test for a fault exists, then the D-algorithm will find it.

[9]Some faults can add an extra bit of state by creating a feedback loop, hence the $m+1$. This is a tight upper bound, because circuits can be constructed whose testing requires traversing the entire state-transition graph.

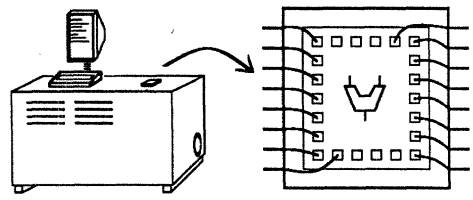
Since the development of the D-algorithm, much work has gone into sophisticated search control heuristics and careful attention to detail, resulting in a succession of more powerful algorithms. This work has effectively solved the test generation problem for quite large combinational circuits (up to 50,000 gates). However, generating tests for moderately complex sequential circuits still lies far beyond the capabilities of these algorithms.

Some recent work has applied similar propagation techniques to more abstract models of circuit structure and behavior. These methods represent an important step forward, but they do not completely solve the problem.

Two characteristics of classical test generation algorithms stand out:

- They are designed to be completely general, i.e., they can be used to test any digital circuit.
- They do not explicitly represent circuit behavior as a whole. Instead, they derive descriptions of global behavior via local propagation through behavioral descriptions of the components. In effect, they have an myopic view of the circuit, and succeed at finding tests by being methodical and exhaustive.

These characteristics are related. From a fixed and small set of components, e.g., the simple boolean gates, any circuit can be built by composition. From a fixed and small amount of domain knowledge, e.g., component behavioral descriptions, any circuit can be tested by local propagation. However, as in many things, there is a power vs generality tradeoff. The price of generality is high test generation costs as the algorithms get lost and miss the forest for the trees. As we shall see in the next chapter, human test experts use a large amount of special-purpose knowledge to focus their test generation effort. This works particularly well because real world circuits are not arbitrary but fall into distinct categories and display regularities that can be exploited.



Chapter 3
Background II: Testing Practice

Summary: Human test experts perform better than existing test generation algorithms. This chapter gives an abbreviated description of what test experts do and contrasts this with the algorithms described in the previous chapter. The differences suggest a research agenda for improving the algorithms of which this thesis is a part.

Human test experts perform better than existing test generation algorithms. An experienced test programmer can write a test program for a circuit board containing many VLSI chips in a period of weeks to months depending on the complexity and familiarity of the circuit. Such circuits contain hundreds of thousands of gates and thousands of bits of stored state. The structure of the resulting test program will reflect the structure of the circuit, with sections of the program devoted to sections of the circuit. If the circuit is changed slightly at the last minute to fix a bug, the test program will only require a small change too.

The classical test generation algorithms described in chapter 2 are far from this level of performance. They are ineffective on large sequential circuits. The flat sequences of test vectors they produce do not have a recognizable structure, making it difficult for an expert to understand, modify and combine their output with the output of other tools. Changes to the circuit are not related to changes in the tests in any meaningful way.

Understanding the differences in how experts and algorithms generate tests yields clues for improving the algorithms and closing this performance gap. This chapter gives a brief description of what test experts do and contrasts this with the existing algorithms. The differences are of two kinds. First, the experts solve a somewhat broader problem than the algorithms do. The core of the problem is the same – design circuit inputs to detect internal faults – but the boundaries of the problem are different. The following differences are central:

- **Negotiable vs Fixed Goals**: The goals for test generation can be established by negotiation with other factors of circuit design and manufacturing. This can make the testing problem easier. Test generation algorithms can and should reflect this possibility, yet current algorithms do not.

- **Realistic vs Simplistic Tester Models**: Tests must be designed to exploit the capabilities of the machine that will execute them. Test experts understand and design for the capabilities of complex, modern testers, while current test generation algorithms design tests only for extremely simple, vector-oriented testers.

Second, experts generate tests using methods that are quite different from the classical algorithms. The central differences are:

- **Specialized vs General Methods**: Experts have developed many special purpose techniques for testing specific kinds of circuits. The limited scope of these techniques enables them to be more effective or more efficient than the existing, general-purpose algorithms.

- **Task Understanding vs Search**: Experts understand the task that a circuit is designed to perform and the patterns of component activity inside the circuit that accomplish it. This understanding helps them to focus problem solving effort on potential tests that are likely to be achievable. Existing algorithms do not understand the circuit's task and rely upon search to discover what behaviors the circuit might possibly perform.

- **Rich vs Limited Test Representation**: Experts write test programs rather than test vectors. The programs provide convenient access to tester features and are a more compact and human readable description of how to test a circuit than are test vectors produced by existing algorithms.

These differences form the starting point for this thesis. That experts use many specialized methods rather than a single general one suggests that knowledge engineering is an appropriate paradigm for studying circuit testing. That the testing problem is negotiable with the circuit designer is the basis for asserting that test generators should be fast, giving up completeness if necessary. That experts have a global understanding of circuit behavior while the algorithms do not is the impetus for DB-TG. The representations of circuit behavior DB-TG generates by simulating circuit operations provide a kind of global perspective and focus search. That test

experts understand tester features and programs are more expressive than test vectors is the basis for PF-TG, which generates tests in a rich test language.

The remainder of this chapter expands on these key differences and gives a broad outline of how one expert creates tests. The contrast between the algorithms and the experts suggests an agenda for testing research of which this thesis is a part.

3.1 Experts Solve a Broader Problem

This section describes these three ways in which real-world test generation extends beyond the scope of the classical algorithms: (i) available circuit descriptions often lack detail; (ii) experts express many tests as programs rather than as vectors; and (iii) testability can sometimes be negotiated between designers and test engineers.

3.1.1 Detailed Circuit Descriptions are Often Unavailable

Detailed structural (e.g., gate-level) circuit models are often unavailable to test experts. Instead, they must rely upon block diagram descriptions of the sort found in databooks. This situation can occur due to poor communications if the design and test teams are separated within the same organization and due to product evolution goals and competitive pressures when the design and testing teams are in different organizations. For example, chip manufacturers routinely withhold gate-level models of the components they produce so that the implementation may change with technology.

Component models may also be unavailable when design and test are integrated: if test generation is attempted early to gauge a developing design's testability, then detailed component implementations may not yet exist. Both loosely coupled and tightly coupled design and test environments need effective test generation when detailed structural models are not available.

Test experts address this problem in two ways. First, they use functional test techniques where practical. These techniques use behavioral descriptions that must be available, or designers could not use the circuit. As commodity components have become more complex, however, test experts have come to need more detailed information than designers. Consider, for example, an instruction cache in a modern microprocessor. A designer using the microprocessor as a component will reason about the stream of instructions that it executes. He will reason about the cache only in the most general terms, e.g., what is the hit ratio and will a particular inner loop

fit? To test most of the microprocessor, a test expert should also reason about the instruction stream. However, to test the cache, he must reason in detail about cache behavior. When this detailed information is unavailable, the test expert must resort to ad hoc approaches.

Test experts also reason by analogy when detailed structural models are unavailable. Unfortunately, functional test techniques often yield long tests. The key to generating short tests for complex circuits is the ability to divide testing problems based on the structure of the circuit description and to conquer each separately. Test experts can supply a finely structured description where none has been provided by assuming that a component whose structure they do not know is implemented like a component whose structure they *do* know. They then generate tests for the component they know and apply them to the component they do not know.

Generating tests by analogy works well for components whose designs are fairly well standardized, as similar implementations tend to fail in similar ways. Note, however, that there is some variability in how even common component types are implemented. Test experts handle this variability by developing tests that are more general than necessary for a particular circuit. Over time, experts have built up a how-to-test lore covering the most common implementation styles of familiar component types.

3.1.2 Experts Write Programs rather than Vectors

Test experts write test programs rather than test vectors. A **test program** is a test expressed in a version of a general-purpose programming language extended with statements for applying values to circuit inputs and observing values on circuit outputs. The pattern of inputs and predicted outputs that occurs when a test program is executed is equivalent to a set of test vectors. Test programs have several important advantages over vectors for human test programmers: (i) programs express repetitive tests compactly; (ii) the languages allow convenient access to tester features; (iii) test programs are easier to understand, modify and debug.

Consider the simple memory test program in figure 3.2. The Galloping Pattern Memory Test (GALPAT) tests for leakage between memory cells by writing a pattern and reading it back while repeatedly checking that reading and writing individual memory cells does not disturb other cells. Figure 3.1 illustrates how GALPAT tests a 16-bit memory by walking a 1 through a background pattern of 0's. To complete the test, GALPAT also walks a 0 through a background of 1's.

A tester will execute 1056 memory cycles when it runs this program for a 16 bit

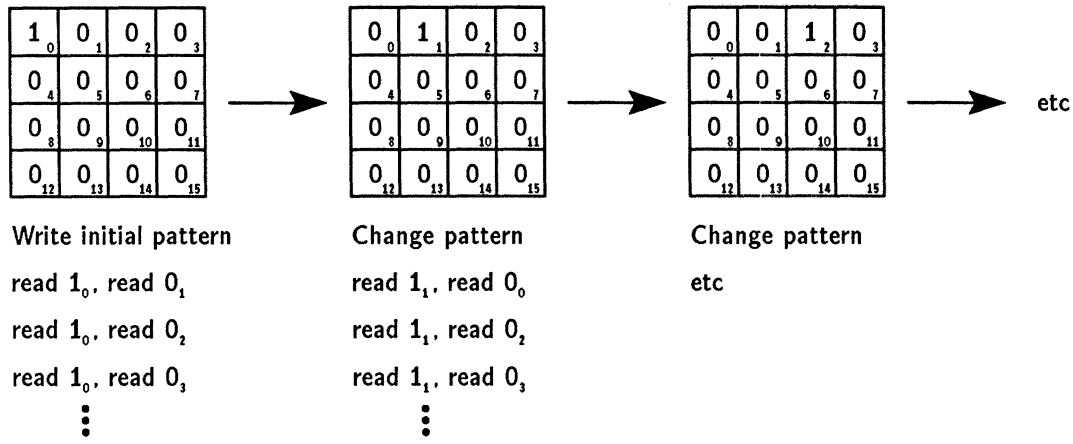


Figure 3.1: *GALPAT tests this memory by walking a 1 through a background pattern of 0's. After every change, GALPAT reads back the full pattern to make sure other cells were not effected.*

memory, and each memory cycle corresponds to several vectors (how many depends upon the details of the tester). A 1024 bit memory requires roughly 4 million cycles. Clearly the program is a more compact and more readable encoding of this test than the equivalent vectors would be.[1]

3.1.3 Testability can be Negotiated

The scope of the test generation problem can be negotiated when design and test are tightly coupled. This section discusses design for testability and its effect on test generation.

Section 2.3.6 showed one method of converting a sequential circuit into a combinational circuit in order to apply an existing test generation algorithm. This transformation only provides a way for a combinational test generator to model a sequential

[1]GALPAT takes time proportional to the square of the memory size. This test lies on a broad spectrum of memory tests. Some, with names like WALKPAT and MARCHPAT, are simpler, faster and somewhat less thorough. Others are more complex and more thorough. Still others are roughly equivalent in fault coverage but work faster by checking only physically plausible read/write disturbances, i.e., those between adjacent memory cells. Which of these tests an expert will use depends upon the kinds of faults a particular memory is susceptible to, whether the memory has been tested before, e.g., by the manufacturer, and to a certain extent upon the expert's experience and taste.

```
procedure GALPAT (MinAddress, MaxAddress)
  begin
    GALPAT_WITH_BACKGROUND (MinAddress, MaxAddress, 1);
    GALPAT_WITH_BACKGROUND (MinAddress, MaxAddress, 0);
  end;

** Execute GALPAT with a particular background value
procedure GALPAT_WITH_BACKGROUND (MinAddress MaxAddress CellValue)
  var BackgroundValue = 1 - CellValue;              ** CellValue is either 0 or 1
  begin
    for Address = MinAddress to MaxAddress          ** Fill the memory with the
      do MEMORY_WRITE (Address 0);                  **  background value
    for CellAddress = MinAddress to MaxAddress do   ** Gallop the cell-value through
      begin                                         **  the memory one cell at a time
        MEMORY_WRITE (CellAddress CellValue);       ** Write to a cell
        READ_PATTERN (CellAddress CellValue MinAddress MaxAddress); ** Disturbed?
        MEMORY_WRITE (CellAddress BackgroundValue); ** Erase the cell
      end;
  end;

** Check that the pattern has not been disturbed
procedure READ_PATTERN (CellAddress CellValue MinAddress MaxAddress)
var BackgroundValue = 1 - CellValue;
  begin
    for Address = MinAddress to CellAddress-1 do  ** Read the cell, Read a
      begin                                       **  background cell.  Repeat
        MEMORY_READ (CellAddress, CellValue);     **  for all background cells.
        MEMORY_READ (Address, BackgroundValue);
      end;
    for Address = CellAddress+1 to MaxAddress do
      begin
        MEMORY_READ (CellAddress, CellValue);
        MEMORY_READ (Address, BackgroundValue);
      end;
  end;
```

Figure 3.2: *The GALPAT memory test from [bennetts82].*

circuit and does not affect the circuit *design* at all. A second and more effective transformation involves changing the original design. Figure 3.3 shows an example of a design style called **scan design**. In this style, all state elements in the circuit are connected to form a large shift register called a scan path. The term "scan" refers to the ability to read or scan the circuit state by shifting out the contents of the state registers. Values can also be shifted in to put the circuit into any state.

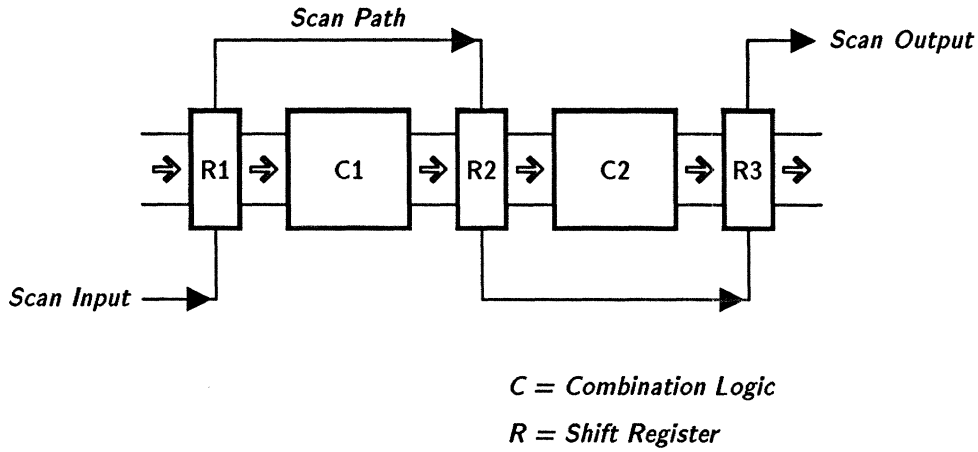


Figure 3.3: *The Scan Design Style: changing registers to shift registers simplifies testing by breaking up a sequential circuit into combinational pieces.*

This transformation enables a test generator to create tests for a sequential circuit as if it were combinational. Moreover, since the shift register partitions the combinational equivalent circuit into small pieces, the cost of generating tests for the equivalent circuit is reduced. These advantages of the transformation come at the cost of several disadvantages: (i) it slows the circuit down because shift registers are slower than normal registers and (ii) it increases circuit size which can increase manufacturing costs, (iii) it can increase test application times due to shifting in values via long scan paths, and (iv) it does not help when testing the circuit at the speed it will run in the field. Tradeoffs between these advantages and disadvantages are the causes of negotiation between designers interested in performance and test engineers needing access to internal components.

The scan design style is an instance of a large set of design styles and techniques that go under the name of Design for Testability (DFT) [williams83]. The details of the various DFT techniques are unimportant for our purposes. What is important, though, is that test generation is not a mathematical problem with fixed inputs and

outputs. Rather, it is a real-world problem whose parameters can be negotiated using the tradeoffs above. If a circuit's complexity makes generating or applying tests too expensive, then this cost can be traded off against circuit performance and manufacturing costs using DFT techniques.

Test experts understand DFT and how to use testable structures when they appear in circuits. Moreover, test experts form an increasingly important part of circuit design teams and can suggest modifications during early stages of design that will reduce testing costs later. In industry today, test generation is still strongly separated from design, but this separation will lessen in time as the advantages of tight coupling become widely recognized and inertia is overcome in circuit manufacturing organizations.

The possibility of design for testability affects the goals of a test generator. When the design can be changed, we want a test generator that can quickly separate portions of a circuit that are straightforward to test from portions that are more difficult and time consuming. The time consuming problems can then be simplified by modifying the design and giving the modified design back to the test generator. DB-TG was created to work in this kind of design environment. This issue is explored in section 6.5.

3.2 Task Understanding: How One Expert Generates Tests

This section describes how one expert generates tests. The method is in the form of advice from an expert test programmer to a novice.[2] The method is not detailed enough to follow by rote: experience and flexibility are essential to apply these steps in the context of a particular circuit. Consequently, this method cannot now be implemented directly by computer. The method, however, does suggest a way of looking at the problem that is quite different from the approaches taken by existing algorithms, and this perspective strongly influenced the design of DB-TG.

The expert follows four broad steps:

1. Understand the circuit.
2. Identify the test objectives.
3. Write the test program.

[2]The expert is Gordon Robinson of GenRad Inc.

4. Debug the test program.

The steps are covered in order.

3.2.1 Understand the Circuit

The first and most important step is to understand the circuit. What is the circuit designed to do and how does it do it? The expert puts it like this: "I learn the basic rhythms of the circuit, asking what was the circuit designed to do and can therefore do naturally? Then I work within those rhythms to generate tests." Understanding how the circuit works is crucial because it focuses the expert's search for ways of causing desired activity inside the circuit. Understanding consists of four components: recognizing high-level structure, categorizing the circuit, identifying what can be done and identifying what must be done.

3.2.1.1 Recognize High-Level Structure

Often the schematics contain little information about the high-level circuit structure and some analysis is needed to clarify it. For instance analyzing feedback paths may identify (possibly) independent state machines. This must be done with care, as such state machines which are topologically one can be considered as several if one or more crucial feedback lines are ignored. These lines may be activated only under rare conditions. Identify the internal interfaces between subsystems. Identify clocks, recognize signal names and naming conventions, look for small, well-known configurations (e.g., divide-by-N circuits) and notice proximity between components on the schematic. The expert pieces together a picture of the circuit from such clues.

3.2.1.2 Categorize the Circuit

By recognizing high-level circuit structure, the expert tries to build up a description in his mind that allows him to categorize the circuit as one of a small set of commonly occurring circuit types, e.g., a microprocessor board, a peripheral or memory board, a single processor board, or a microprogrammed peripheral controller. Sometimes the expert is told what kind of circuit it is, and this step is trivial. Categorizing the circuit suggests other information about the circuit that will be needed. For instance, if the circuit is a peripheral, then the expert looks for the range of addresses the circuit will respond to. Categorization and circuit recognition go hand-in-hand as the expert returns to the circuit to answer specific questions suggested by the circuit type.

3.2.1.3 Identify What Can Be Done

Identify the space of possible actions that can be taken to manipulate the circuit during testing. What is the basic set of actions? What variability is there within these actions? Where in the circuit can actions be initiated? An expert might consider, for example, the instructions of a processor to be its basic actions. The instructions have arguments whose values can be chosen. There is also variability in how instructions interact with the world, i.e., wait states during memory read cycles. Actions are initiated at the bus interface when the processor fetches an instruction.

3.2.1.4 Identify What Must Be Done

Certain actions must be taken while testing a circuit for the circuit to work as designed. A circuit is designed to operate in an environment. The environment is a contract between the circuit designer and the user: if the circuit is placed in its proper environment, it will behave as advertised. Supplying power to the circuit is a trivial example. Dynamic memory is a more complex example: the user must guarantee that the memory will be accessed at least N times per second. The test expert must understand the circuit's environment and decide how much of it must be replicated to test the circuit.

3.2.2 Identify the Test Objectives

After characterizing the space of possible and necessary actions, the expert decides what testing goals to work on. His central strategy is divide and conquer: partition the circuit into components and exercise and observe each major piece. Partitioning is done by function or structure as convenient. Partitioning can be guided by the circuit category, e.g., experience has shown that breaking up certain kinds of circuits in certain ways is useful. Often, the expert has a vague test plan associated with a way of partitioning the circuit. For example, the expert will break up a microprocessor-based system into the processor and the peripherals. He will then look for a way to disable the processor allowing access to the peripherals without interference. Alternatively, he will look for a way to load the processor itself with a program for testing the peripherals. In the absence of guidance, the expert breaks down the circuit into familiar components.

3.2.3 Write The Test Program

The test program is organized around the test objectives, one program section per objective. The expert usually orders the objectives from easiest to hardest so he can confirm or expand his understanding of the circuit as he goes along.

3.2.3.1 Component How-to-Test Knowledge

The expert divides the problem of testing the circuit into test objectives that are specific and familiar. The expert either already knows how to accomplish each objective assuming accessibility to the portion of the circuit involved or he knows straightforward ways to work out solutions, again assuming accessibility.

Experts develop and share many special-purpose techniques for testing specific component types. If, for example, the objective is to exercise a particular data register, then past experience suggests several patterns of data that are good for that purpose. The expert also knows the automated test generation techniques and under what circumstances they are effective. For combinational blocks of circuitry, the expert usually runs Podem or another test generation algorithm on just that block and uses the resulting vectors in his test program.

3.2.3.2 Accomplishing Test Objectives

If a test objective involves a component that is not directly accessible, then the expert must embed his component test using the surrounding components. The expert's understanding of circuit behavior is central in focusing his search for an embedding. Suppose, for example, that a UART[3] contains a one character buffer for holding incoming data while it interrupts and waits for the processor to fetch the character. If a second character arrives before the processor reads the first, then the UART signals an overrun condition. If the test objective is to cause the UART to raise an overrun condition, then one simple solution is to feed two characters into the serial input while not responding to the interrupt. Once the expert understands how the circuit works, his internal description of the circuit allows him to quickly solve problems like this. Only occasionally, does the expert resort to explicit signal tracing through the structure of the circuit in order to achieve test objectives.

[3]A Universal Asynchronous Receiver-Transmitter (UART) is a communications circuit that can, for example, connect a processor to a terminal. The UART translates between bus cycles on the processor side and characters on a serial line on the terminal side.

3.2.4 Debug the Test Program

Debugging a test program is much like debugging any kind of program. The expert runs the program on an instance of the circuit, or does the equivalent using a circuit simulator, and examines program trace information for problems. Two kinds of bugs are possible: (i) the test program drives the circuit in an unintended way or predicts outputs with the wrong value or at the wrong time, and (ii) the test program does not detect enough faults. The first kind of problem is an error of conception or execution. Tracking down and fixing this kind of bug in a test program is very similar to fixing a bug in any other kind of program. The second kind of problem is an error of omission. The expert can use a fault simulator to identify which areas of the circuit are insufficiently well tested. The expert usually leaves the existing program alone and fixes the program by adding new program code to catch the remaining faults. The expert stops adding tests when the test program meets the fault coverage target. A small percentage of bugs fall into both categories, e.g., the test program may create and propagate a fault effect to a circuit output but fail to look for it at the correct place and time. These bugs are fixed using a combination of the debugging techniques described above.

3.2.5 Summary

The method has four steps: (i) understand the circuit, (ii) identify the test objectives, (iii) write the test program and (iv) debug the test program. In the first two steps, the expert gathers information for use later. He asks what tasks does the circuit perform? What primitive actions can be used to build tests? What restrictions must he respect beyond those readily apparent from the circuit structure? In the last two steps, the expert generates tests by embedding component tests that he already knows. His knowledge of the circuit guides and focuses his search. He uses a combination of functional and structural test generation techniques. For instance, he does line justification and path sensitization, although he very rarely considers multiple sensitive paths. He uses a copy of the circuit or a simulator to check his predictions about circuit behavior, and he uses a fault simulator to grade the performance of his test program.

3.3 Experts Use a Collection of Skills

While circuit testing is a specific and well-defined problem, solving it efficiently remains something of a black art. The previous sections have touched on many distinct skills that test experts rely on to do this.

- **Classification:** Experts can choose how to model the circuit or which technique to use to solve a particular problem. For instance, experts comfortably switch between behavioral and structural models.
- **Cliche:** Experts can solve a small, common testing problem by recognizing the problem and recalling a solution (e.g., recalling how to test a register).
- **Explicit Search:** Experts know and occasionally resort to doing explicit path sensitization and line justification, usually when their automated tools fail.
- **Algorithms:** Test experts understand the capabilities of the existing test generation algorithms and use them when appropriate, often dividing problems into pieces that the algorithms can handle.
- **Reformulation:** Experts can change circuit representations (e.g., they can derive a state transition diagram from the schematic).
- **Specialization:** Experts can select parts of a general test that are applicable to a particular circuit. For example, if some component features are disabled in a particular design, then some portions of the standard component test can be omitted.
- **Tester Hardware:** Test experts understand and exploit the capabilities of the hardware.
- **Test Programming:** Test experts understand programming concepts like branching, iteration and subroutine calling and use a programming language to express complex tests.

Experts also have available the large array of techniques developed in industry and in the literature. Much of a test expert's knowledge covers what the techniques are and when they should be used.

- Test generation techniques distinguished by circuit type:

  - Random Logic [fujiwara85]
  - Memories [bennetts82, sarkany87]
  - Microprocessors [thatte80, brahme85]
  - Programmable Logic Arrays [williams87]
  - Systolic Arrays [rawat87]
  - ⋮

- Test application techniques:
  - In-Circuit
  - Edge-Connector
  - Memory Emulation [sargent83]
  - ⋮

The expert must choose among the techniques available in his test programming shop those that are best suited to a particular problem.

3.4 Summary and a Research Agenda

The contrasting pairs in figure 3.4 summarize the differences between classical test generation algorithms and expert test programmers. Each pair suggests one way that current algorithms might be extended.

In this thesis, I have concentrated on the differences marked by ⇒. These differences have been explored in the design of DB-TG and PF-TG: pairs 3 and 4 are the roots of the designed behavior test generator described in chapter 4, and pairs 4, 8 and 9 are the roots of the program fragment test generator described in chapter ChapFragments.

Some contrasting pairs above are well recognized by the testing community and are under investigation. For instance, work on hierarchical test generators has gone some distance in the direction suggested by pairs 1 and 2. Pair 6 suggests the test generator should know about DFT techniques, and some test generators are built specifically to work with certain testable design styles, e.g., scan circuits.

Some characteristics of expert test programming are inappropriate for emulation in a program because the tasks they solve can be accomplished more effectively in other ways. The most obvious characteristic in this category is learning about the circuit

1. Algorithms use gate level models
   Experts use block diagrams

2. Algorithms use simple behavioral descriptions
   Experts use abstract descriptions from the databook

⇒ 3. Algorithms rely on optimized search through large search spaces
   Experts know more about which solutions are plausible and search less

⇒ 4. Algorithms will generate any test the structure of the circuit will allow them to
   Experts learn the natural rhythms of the circuit, i.e., what can and must be done, and manipulate the circuit within those restrictions.

⇒ 5. Algorithms have no model of the tester
   Experts know how to use the tester's capabilities

6. Algorithms have no model of DFT techniques
   Experts understand how to use testable structures

7. Algorithms are completely general and work well on combinational but not sequential circuits
   Experts use specialized methods and work well on sensibly designed circuits but not on spaghetti[4]

⇒ 8. Algorithms produce test vectors
   Experts produce test programs

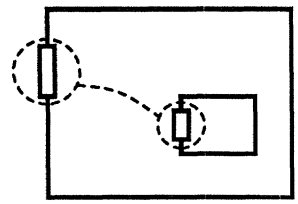
⇒ 9. Algorithms produce incomprehensible output[5]
   Experts document their code

10. Algorithms rely on single built-in methods
    Experts use a toolbag of ideas, including the algorithms

Figure 3.4: *Contrasting the Algorithms with the Experts. The contrasting pairs marked by ⇒ are emphasized in the design of two novel test generators introduced by this thesis.*

from poor circuit descriptions. Much of the difficulty of real-world test programming today is due to poor documentation. A test engineer must often interpret ambiguous and incomplete design information and reconstruct complete circuit descriptions from many sources simply to understand what the circuit does. He must recognize high-level structure in detailed, low-level schematics.

Recognizing circuit structure and learning about circuit behavior are potentially interesting areas for AI research, but they involve reconstructing information that the circuit designer already has. From a practical standpoint, these problems should be solved by improving communication between the circuit designer and the test engineer, e.g., by more comprehensive CAD tools [foyster84]. I have therefore not invested any effort in these problems.



Chapter 4
A Designed-Behavior Test Generator

Summary: Embedding component tests is the key step in test generation. Embedding problems can be solved efficiently using operation relations, a representation of circuit behavior that directly connects internal component operations with directly executable circuit operations. For sequential circuits that provide few operations at their interfaces, operation relations can be efficiently computed by searching traces of simulated circuit behavior. This approach is efficient because the search space is smaller. This approach is sufficient because circuits can be tested without going outside their normal operations. This chapter is self-contained and repeats some material from scenario I in chapter 1.

4.1 Introduction

Circuits are designed to perform specific tasks: gates compute boolean functions, registers load and hold values, disk controllers transfer blocks of data and microprocessors execute instructions. Informally, these operations at the circuit interface and the patterns of internal activity that implement them are what we mean by "a circuit's designed behavior."

By definition, a properly implemented circuit can carry out its designed behavior. Often a circuit can behave in other ways too, which correspond to input sequences outside its interface protocols. These incidental behaviors are usually irregular: what does a disk controller do if presented with a string of random numbers at its inputs? Whatever happens, it will not be as simple to describe as "transferring a block of data." We contrast a circuit's designed behavior with its incidental behavior. Designed behavior is easier to reason about than incidental behavior, because designed behavior can often be described in simpler, more abstract terms.

Understanding what a circuit is designed to do is one important way in which expert test programmers differ from existing test generation algorithms. Understanding enables an expert to focus quickly on candidate solutions for test generation problems by avoiding patterns of internal activity that cannot be achieved at all.

This chapter describes the Designed Behavior Test Generator (DB-TG), a test generator that uses representations of designed behavior to handle a class of complex, sequential circuits. The contributions of this work are:

- The identification of representations for a circuit's designed behavior at several levels of abstraction.
- The use of designed behavior to limit the search used in planning circuit tests.
- The construction of techniques for extracting descriptions of designed behavior from the commonly available descriptions of circuit behavior and structure, e.g., interface specifications, simulation models, and schematics.

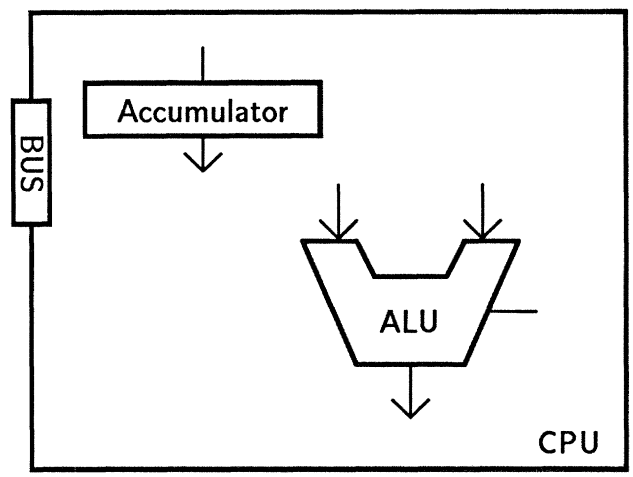
These representations are the basis of a heuristic and effective solution to the most difficult subproblem of test generation – embedding known tests for a component into a larger circuit.

Embedding a component test is the central step of the sample test generation problem shown next. Human test programmers can solve this problem easily. When we ask why, we are lead to consider circuit representations that make the problem straightforward. The remainder of the chapter introduces a test generator based on these circuit representations, which we describe by stepping through its solution to the sample problem. Chapter 5 considers the advantages and disadvantages of these circuit representations for doing test generation and place them among the armamentarium of techniques available to test engineers. Then, in chapter 6, we remove a simplifying restriction on information flow within the test generator and in the process integrate the test generator more closely with classical test generation techniques.

4.2 A Test Generation Example

Figure 4.1 reprises the simple test generation problem from the scenario in chapter 1. Testing experts and others familiar with computer architecture easily suggest the solution shown from the information shown in the figure. In particular, they are able to associate the SUM instruction with the goal of making the ALU add numbers without

seeing detailed datapaths. The sparse information required to solve the problem and the high-level form of the solution are typical of human test programmers but very unlike classical test generation algorithms. While we do not presume to know how human test programmers actually come up with such solutions, we do propose an automated method which generates the same kinds of answers. After introducing the ideas behind this method, we walk it through exactly this example, generating the sequence of instructions above.



Instructions

`LOAD Ai`	*– Load accumulator from memory*
`ADD Bi`	*– increment accumulator by Bi*
`STORE sum`	*– write sum to memory*

Figure 4.1: *Problem: test the ALU's ability to add pairs of numbers (A_i, B_i) by manipulating the BUS interface. The processor is accumulator based and provides (at least) the usual load and arithmetic instructions. Solution: repeat the three instructions shown to the right of the figure for each pair (A_i, B_i). This solution assumes that (i) the SUM instruction actually uses the ALU shown in the figure, (ii) the LOAD, STORE and SUM instructions manipulate the accumulator shown, and (iii) the LOAD, SUM and STORE instructions can handle the test data required by the ALU.*

4.3 Overview

This section describes the test generator's structure and briefly introduces the key ideas that lie behind it. Sections 4.5 through 4.9 elaborate on this description while walking through DB-TG's solution to the example problem above.

4.3.1 The Key Ideas

DB-TG is based on four ideas:

1. **Embedding Expert-Supplied Component Tests**: Testing experts have developed clever and effective methods of testing common components. Test generators should take advantage of them.

2. **Operation Relations**: Knowing about relationships between circuit operations (e.g., instructions) and component operations is the key to embedding component tests. Test generators should obtain and manipulate descriptions of these operations.

3. **Simulate and Match**: Operation relations can be obtained by simulating circuit behavior. The relations applicable to a particular situation can be found by searching simulation traces.

4. **The Designed Behavior Heuristic**: Test circuits using only patterns of usage anticipated by the designer. This means that a test generator can take the circuit's normal interface operations as its primitive actions. This in turn limits what must be simulated.

Each idea is a means of implementing the previous ideas or a justification for them. Together these ideas give a vertical slice through the test generator. The first idea has been used in other test generators. Here, we simply put the idea to work again. The other three ideas are unique to DB-TG and are described next.

4.3.1.1 Operation Relations

As a component is a part of a circuit, so is its behavior a part of the circuit's behavior. Knowing part-whole relationships about behavior is useful, because they provide a straightforward means of embedding component tests into the circuit.

Two kinds of part-whole relationships are useful for solving this problem: **causal connections** and **parameter relations**. In the example above, executing a SUM instruction *causes* the ALU to add, therefore we say the processor's addition instruction and the ALU's addition operation are *causally connected.* Parameter relations hold between the parameters of two causally connected operations. In the example there is a simple relationship between the inputs and outputs of the ALU when it is doing the work of the SUM instruction and the inputs and outputs of the instruction

itself. The term **operation relations** refers to the causal connection between two operations and any relationships between their parameters.

Operation relations exist because the circuit designer used the component operations to implement the circuit operations in the first place. The relations are often simple, especially in datapaths, because a circuit can rely at certain times almost directly on a single component to do its work. Knowing operation relations is useful, because they are the basis for a powerful, heuristic approach to solving the hardest subproblem in circuit testing: embedding a test for a component into the surrounding circuitry.

We describe operations, operation relations and the embedding process in detail later. Here, we simply assert that operation relations are a highly abstract form of circuit description that allow component tests to be embedded without working through the detailed structure of the surrounding circuitry. Therefore these relations should be obtained and manipulated explicitly during test generation.

There are several ways to supply operation relations to a test generator depending upon the type of circuit and the larger design environment the test generator sits in. For sequential circuits that execute a small number of well-defined operations, we use simulation working from descriptions of the circuit structure, the component behavior and the circuit interface. The strategy of computing operation relations via simulation is based on the following idea about planning, called simulate and match.

4.3.1.2 Simulate and Match

Test generation is fundamentally a planning problem [fikes71, sussman75, sacerdoti77, stefik80, chapman85]: how can we manipulate the circuit inputs to cause particular behaviors inside? Tests are usually planned by repeatedly refining the goal of causing a specific internal behavior until the problem can be solved by direct action on the circuit inputs. Often a newly proposed subgoal conflicts with previous subgoals, forcing the test generator to backtrack and try again. This alternation of search and backtrack is characteristic of planners in general and test generators in particular.

This strategy is inefficient when solutions are infrequent and there is little guidance available to lead the test generator to them quickly. Unfortunately, test generation for complex, sequential circuits seems to be such a situation: a test generator is likely to propose and retract many potential solutions before finding one that meets all of the constraints imposed by the circuit structure and behavior. The difficulty of finding solutions is compounded by the potentially high cost of ruling out proposed solutions, since the test generator may have to reason about the circuit far backward or forward

in time before discovering a constraint that causes it to backtrack.

It is however possible to avoid this pitfall if the circuit executes a small number of operations. For circuits in this class, an effective planning strategy is to take the circuit operations as the planner's primitive actions and to simulate them, looking for patterns of internal activity that could prove useful during testing.

This so-called **simulate and match strategy** turns goal-refinement planning on its head. Instead of proposing a behavior that tests a component and then asking "is this behavior achievable in the context of the surrounding circuitry?" this technique proposes a behavior known to be achievable and asks "is this behavior useful for testing anything?" The approach is essentially an effort to focus search on behavior *known to be achievable* rather than on potentially achievable behavior that must be verified via complex reasoning. The result can be a sharp reduction in the search necessary to achieve testing goals.

Identifying known-achievable behavior by simulating it is only practical for problems that have relatively few primitive actions, i.e., few operations at the circuit interface. Here, we take the primitive actions to be executing circuit operations (e.g., instructions) rather than the more fine-grained actions of controlling and observing to individual circuit I/O pins used by traditional test generators. This is the subject of the designed behavior heuristic, the last major idea upon which the test generator is built.

4.3.1.3 The Designed Behavior Heuristic

> *Test circuits without going outside the behavior they were designed to perform.*
> – Gordon Robinson

This statement is the result of much experience writing test programs. It says essentially that one need not do anything odd or ill-formed to a circuit in order to test it. Using this heuristic, a test generator can assemble tests out of standard circuit operations, albeit with carefully chosen parameters, and need not reason in detail about controlling each input wire individually. This can reduce the amount of search necessary to generate tests.

This heuristic is also useful as a problem decomposition strategy. If the circuit under test is in turn a component in a larger system, then tests that use standard circuit operations are likely to be achievable in the larger system. Tests that fall outside the standard operations, i.e., that do not meet the circuit's communication protocols with the rest of the system, are extremely unlikely to be achievable.

The Designed Behavior Heuristic is an integral part of DB-TG, and in this chapter, we take its use as a given. In chapter 5, we reexamine this heuristic and consider its implications in more detail.

4.3.2 Structure of the Program

Figure 4.2 shows the structure of DB-TG. The boxes in the figure represent either processes or databases and the arrows represent queries and responses. DB-TG follows the usual divide and conquer approach to test generation: (i) work out how to test each component as if it were in isolation, then (ii) work out how to execute these component tests acting only on the circuit inputs and outputs. In more detail, DB-TG follows these steps to generate tests for a circuit:

1. Pre-load the operation relation database by simulating the behavior of the circuit on each of its operations and recording what operations the components execute. The operation relation database describes known-achievable patterns of activity inside the circuit. (The circuit designer can also add entries to this database.)

2. The test generation top level steps through the components. For each component it looks up the component type in the library and fetches the corresponding component test, which is a procedure. It then "runs" the test.

3. Component tests capture expert how-to-test knowledge. When run, they try to work out how to test a component inside the circuit. The most primitive kind of component test is expressed in terms of a test operation that the component must be able to execute and specific test data for the parameters of the operation. Other component tests are combinations of primitive tests.

4. A primitive component test finds an instance of the test operation in the operation relation database and extracts the relations between that instance and the circuit operation that caused it.

5. It then substitutes the test data into these relations and solves for the parameters of the circuit operation. If successful, this process converts the component test from one expressed in terms of component operations to one expressed in terms of directly executable circuit operations.

The test generator works through these steps in order and always passes information forward from one step to the next, never backward. If it is impossible to solve

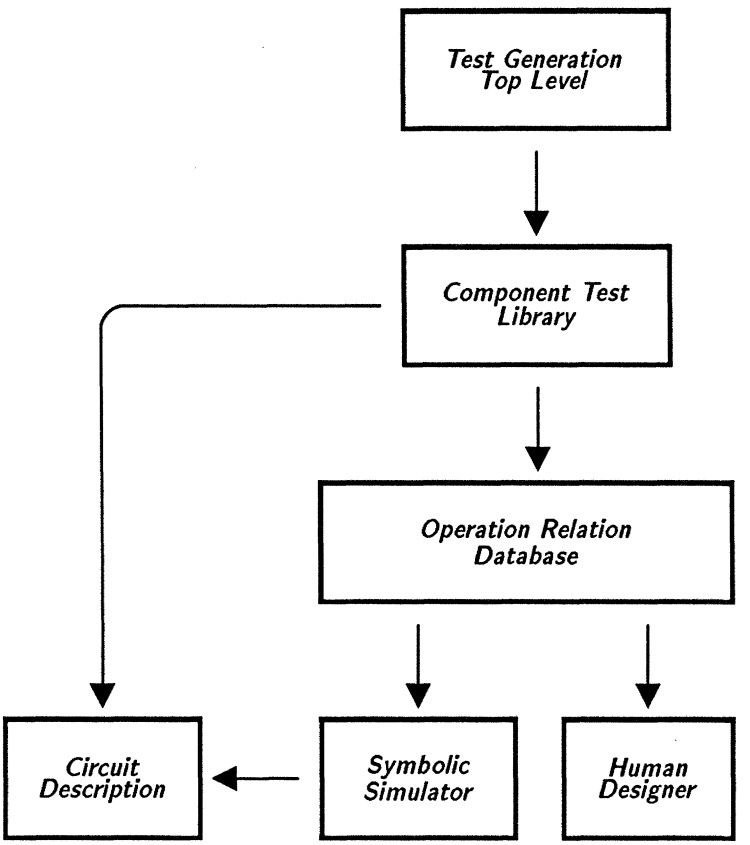


Figure 4.2: *DB-TG's structure. The boxes are either processes or databases and the arrows represent queries.*

for the parameters of the circuit operation, then the test generator backtracks and searches for other instance of the test operation to use. If it cannot find another, then it fetches another component test. No information other than failure passes back from later subproblems to earlier ones. Also, even though the operation relation database exists at the time the component test is fetched, the database is not consulted.

This limitation on the information flow between steps is a simplification: examining the operation relation database can help to select component tests that will pass the later steps or can help to design component tests on-the-fly. This elaboration on the basic test generator is described in chapter 6.

This broad outline of how the test generator works omits several subtleties that we get to later. First, we describe how component tests are represented for use by this test generator. Then we describe more precisely what these operations and the relationships between them are and how they are used to generate tests. Section 4.7 describes how a test generator can compute these relationships given descriptions of circuit structure and behavior. We illustrate each idea as we come to it by working through the problem of testing the ALU's addition operation.

4.4 The MAC-1 Microprocessor

Throughout the rest of this chapter, we expand on the example of testing a processor's ALU. To make the discussion concrete, we introduce a simple but fully functional microprocessor from [tanenbaum84] (see figure 4.3). This processor, called the MAC-1, executes a conventional set of instructions, including loading and storing the contents of its single accumulator, adding and subtracting from the accumulator, and stack operations. The central portion of the circuit is a 16-bit-wide datapath. The right-hand portion is a microcode sequencer whose ROM holds 80 lines of microcode and implements 23 instructions. The address and data busses and their associated signal lines are the only primary inputs and outputs. All internal nodes are inaccessible and must be controlled indirectly through intermediate components.

The MAC-1 is a fairly simple circuit, yet it illustrates many of the difficult problems in test generation: (i) it is highly sequential, (ii) it has to be tested from its bus interface, (iii) no detailed structural model is available, so we cannot use a technique which requires, for example, a gate-level description[1] and (iv) the MAC-1 has several

[1]Tanenbaum's textbook does not contain a detailed gate-level model; only the block diagram shown. Although I have constructed a gate-level model in order to measure the test generator's performance in the accepted way, the test generator does not use this model.

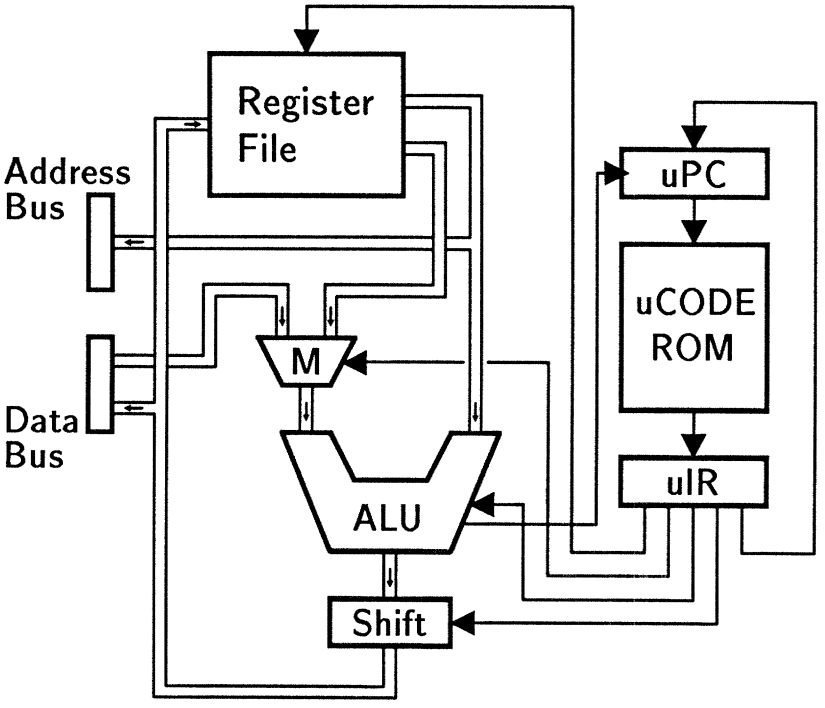


Figure 4.3: *The MAC-1 Microprocessor (some detail has been suppressed)*

testability problems.

This circuit is described to the test generator in three parts: (i) a schematic specifying how components are connected, (ii) behavioral descriptions of the components for use in the simulator described later, and (iii) one procedure per operation that interacts with the circuit during simulation to cause an operation to occur.

4.5 Component Tests

DB-TG solves the component test problem by fetching expert how-to-test knowledge from a library. This library was written by debriefing a test expert (Gordon Robinson) and contains exercises that the expert makes components perform to test them. The central issue here is how the expert's knowledge is represented for use by the test generator.

DB-TG uses operation relationships to embed component tests, so the test repre-

sentation is naturally based on component operations. There are two kinds of component tests: *primitive tests* and *compound tests*. A primitive test describes how to exercise a single component operation, for instance a REGISTER/LOAD operation or an ALU/ADD operation. Compound tests are procedures that assemble primitive tests in order to exercise a component more fully.

4.5.1 Primitive Tests

Figure 4.4 shows a primitive component test capturing the expert's method of testing 16-bit carry-chain adders.[2] This test will be used to test the addition operation of the MAC-1's ALU.

A primitive test has three parts: a test operation, test data, and a fault coverage description. The test operation specifies which component operation will be exercised, and the test data supplies parameters for that operation. Thus a primitive test is a specification to execute one component operation repeatedly. The number of repetitions depends on the amount of test data supplied.

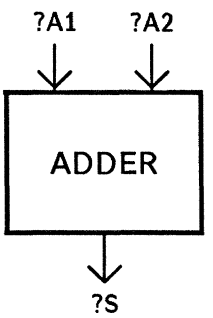
How well a test covers faults in a component depends upon the component implementation. Thus a fault description may have structural preconditions. For instance, the adder test will cover all stuck-at faults if the adder is implemented as a carry chain but will not detect all faults in a carry-lookahead implementation. This knowledge, expressed in English in the figure, is given to the program in a simple rule language. The test generator reports to the user fault descriptions for every primitive test that it successfully embeds.

When to use a particular component test is decided by a compound test, described next.

4.5.2 Compound Tests

Compound tests are collections of primitive tests that, taken together, exercise a component fully. In the current library there is one compound test per component

[2]Adding these eight pairs of numbers reveals any stuck-ats in a 16-bit carry-chain adder. To demonstrate this to yourself, consider the effect of adding these numbers on the single-bit adders inside a carry chain. Recall that a single-bit adder has three bits of input: two data inputs and one carry input. These eight pairs exhaustively cover all eight possible input combinations. A test expert created these pairs by cleverly interleaving exhaustive tests for each of the single-bit adders into a test for the larger adder that requires no extra additions. The method of interleaving generalizes to carry chain adders of arbitrary length.



(a) What the test looks like

?A1	?A2	?S
0	0	0
43690	43690	21844
1	65534	65535
1	65535	0
65534	1	65535
65535	1	0
21845	21845	43690
65535	65535	65534

(b) Test Data

```
ADD( ADDER, ?A1, ?A2, ?S )
```

(c) Test Operation

```
IF the adder is implemented as a carry chain
THEN all SSL faults will be detected
```

(d) Fault Description

Figure 4.4: *A primitive test for an ALU's addition operation. (a) shows what the behavior being tested would look like to a person. (b), (c) and (d) form the test generator's description: (b) is represented as a table datastructure, (c) is represented as a Prolog program that can find test operations within simulation traces of circuit behavior (described later), and (d) is canned text that is part the DB-TG's output.*

To test an ALU, test each of its operations:

1. Test *the-alu-executing-a-noop-operation* like a DATAPATH.
2. Test *the-alu-executing-an-add-operation* like an ADDER.
3. Test *the-alu-executing-a-boolean-and-operation* like a parallel collection of AND gates.
4. Test *the-alu-executing-a-boolean-not-operation* like a parallel collection of NOT gates.

Figure 4.5: *The ALU Compound Test*

type. Each compound test is implemented as a Prolog procedure that can either ask for primitive tests to be embedded into the circuit, call other compound tests as subroutines. Compound tests can decide at runtime which subroutines to call or which primitive tests to embed. In principle, they can do arbitrary computation to make these decisions. In the current implementation, they make simple choices based on the circuit structure around the component under test.

This very general mechanism is used in several specific ways. One way is to implement a boolean combination of primitive tests, i.e. *"embed primitive tests A and B"*, or *"embed A or B."* Another is to express a preference between primitive tests, for example *"Try to embed A. If that fails, then try B."* A third use is to choose among primitive tests based on local examination of the circuit structure.

Figure 4.5 shows a compound test for the simple type of ALU that appears in the MAC-1. This ALU implements four functions: NOOP, ADD, AND, and NOT. The actual compound test is implemented in Prolog but is expressed in English here for clarity.

This compound test is fairly simple and its structure is a common one for components that have multiple operating modes. The test exercises each ALU operation by treating the ALU as simpler kind of component and referencing a simpler test. Since the library already contained tests for DATAPATHs and ADDERs, the ALU test calls them as subroutines by specifying a mapping between the ALU inputs and outputs and the inputs and outputs of DATAPATH and ADDER.

4.5.3 Summary

Short and effective tests are known for many common components. This "how-to-test" knowledge represent the compiled and optimized expertise of the testing field. It is compiled and optimized in the sense that it is the end result of often subtle reasoning about the structure and behavior of common components and design styles. Once someone has worked out and published a clever test, it becomes part of the background of knowledge that experts can bring to new testing problems. Unlike classical test generation algorithms, this kind of knowledge can be captured in our component test representation and brought to bear on testing problems by DB-TG.

Primitive tests, organized around component operations to facilitate embedding, capture expert testing knowledge in a rigid format similar to a set of test vectors. Yet much of an expert's knowledge is less structured, more context dependent or more procedural in nature than can be expressed with primitive tests alone. Compound tests are a trap door, a procedural method of expressing the nuances of how-to-test knowledge.

The component tests of both kinds are the real drivers of this test generator. For each component the top level simply dispatches to the appropriate component test. The test "knows" how to exercise the component, but it cannot do so directly because the component is inaccessible inside the circuit. Instead the test must "rewrite" itself into an equivalent test expressed in terms of the circuit inputs and outputs. This process of "rewriting" is the purpose of path sensitization and line justification in a conventional test generator. DB-TG does this differently using operation relations.

4.6 Operation Relations

As a component is a part of a circuit, so is its behavior a part of the circuit's behavior and its operations a part of the circuit's operations. Two kinds of part-whole relationships have proven useful for solving this problem: causal connections and parameter relations. In the example above, executing an addition instruction *causes* the ALU to add, therefore we say the processor's addition instruction and the ALU's addition operation are causally connected. Parameter relations hold between the parameters of two causally connected operations.[3] In the example there is a time during the execution of an addition instruction when the ALU does the real work. At that time,

[3]In general, the parameters of any two operations can be related, either by coincidence or as consequences of some shared antecedent. However, all of the parameter relationships we will use in this chapter lie between two operations one of which causes the other.

the two values being summed by the addition instruction are the *same two values* being summed by the ALU. We say the parameters of the addition instruction and those of the ALU addition operation are related, in this case, by identity functions. We use the term **operation relations** to refer to both the causal connection between two operations and any relationships between their parameters.

How might a test generator represent operations and relations in order to reason about them? We begin with operations, then move to relations between operations.

4.6.1 Representing Operations and Operation Relations

Operations are represented as frames comprised of five slots: (i) the relevant state of the system before the operation occurs, (ii) the input of the system during the operation, (iii) the output of the system during the operation, (iv) the relevant state of the system after the operation, and (v) mathematical relationships between values occurring in the previous four slots. Here is the MAC-1's SUM instruction, which fetches a value from memory, adds it to the accumulator and stores the results back in the accumulator. We use the notation *device/name* to refer to an operation, e.g., MAC-1/SUM. Variables are preceded by ?'s.

MAC-1/SUM	Before State:	Accumulator	=	?ac
		ProgramCounter	=	?pc
	Inputs:	DataBus	=	(ADD ?addr)
		DataBus	=	?data
	Outputs:	AddrBus	=	?pc
		AddrBus	=	?addr
	After State:	Accumulator	=	?sum
		ProgramCounter	=	?pc1
	Relations:	?sum	=	?data $\oplus_{16}$?ac
		?pc1	=	?pc $\oplus_{16}$ 1

This frame shows input / output activity and state changes during MAC-1/SUM. In this case, the contents of the Accumulator and the ProgramCounter are relevant and their values are labeled ?ac and ?pc respectively. At the beginning of the instruction cycle, the contents of ProgramCounter are written to the address bus (AddrBus = ?pc) and the memory responds with the next instruction (DataBus = (ADD ?addr)). (ADD ?addr) represents an addition instruction with a variable in the address field. The MAC-1 then initiates a memory fetch by writing ?addr to the address bus (AddrBus = ?addr) and the memory responds with the data at that location (DataBus =

?data).[4] Finally, MAC-1 adds ?data and ?ac and stores the sum in the accumulator. The after state shows this sum and the incremented contents of the program counter. If an operation does not involve state changes, then we omit the before and after states as in the ALU's addition operation:

ALU/ADD	Inputs:	OP	=	ADD
		IN-1	=	?in1
		IN-2	=	?in2
	Outputs:	OUT	=	?output
	Relations:	?output	=	?in1 $\oplus_{16}$?in2

This frame says that during an ADD operation the ALU's control input (OP) sees a value (ADD) telling it to compute the sum of the values on its two data inputs (?in1 and ?in2) and put the answer on the output (OUT).

Each of these frames describes the corresponding operation in its most general form. They describe what can happen in terms of a set of internal constraints: the relations between the parameters, e.g., ?sum = ?data $\oplus_{16}$?ac. Because instances of components sit *inside* circuits, instances of component operations also have *external* constraints on what can happen. These external constraints correspond to relations between the parameters of a component operation and the parameters of a circuit operation. Here are the relations between MAC-1/SUM and the instance of ALU/ADD that actually does the work:

?data	=	?in1
?ac	=	?in2
?sum	=	?output

This says that the ?in2 parameter of this instance of ALU/ADD is equal to the ?ac parameter of MAC-1/SUM and so on for the other parameters. These relationships are enforced by the structure and behavior of the circuit.

This set of operation relations is unusual in that it is particularly simple (making it a good example). In general, operation relations can involve combinations of boolean and arithmetic functions, selection, bit-field extraction, concatenation and the like. While not always this simple, operation relations often involve straightforward formulas that are simpler to reason about than the structure of the circuit. (This issue is discussed further in chapter 5.)

[4]The detailed order and timing of bus events is not specified in these descriptions because it is not needed here. Order and timing is present in and only used by the procedures for driving the simulator described later.

4.6.2 Using Operation Relations

The operation relations between MAC-1/SUM and ALU/ADD can be used to transform component tests for the ALU (which a tester cannot manipulate directly) into equivalent tests expressed in terms of the MAC-1 Bus (which the tester can control directly). The operation relations are useful here because they provide a *direct link* from desired actions inside the circuit to actions directly executable by the tester hardware. Figure 4.6 illustrates this link.

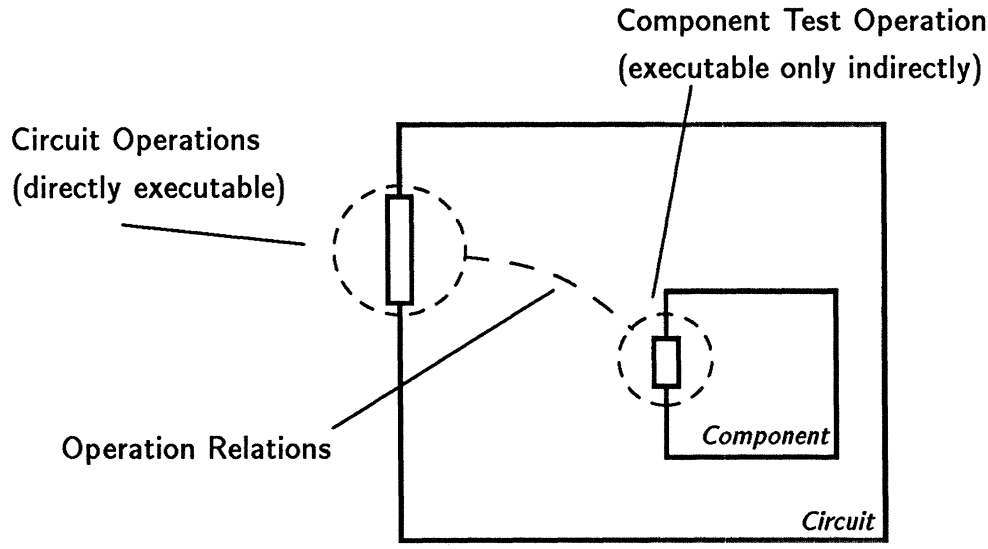


Figure 4.6: *Operation Relations are a direct link between the goals (i.e., desired component operations) and the primitive actions (i.e., directly executable circuit operations).*

DB-TG performs this transformation by substituting component test data into the component side of the operation relations and then solving for the parameters of the circuit operation. Carrying the example straight through, we would show this transformation next. However, note that there are other instances of ALU/ADD that occur during MAC-1/SUM, e.g., one increments the program counter, that have different parameter relations with the instruction. The key step that people make when generating tests for this ALU is to associate the SUM instruction with this particular instance of ALU/ADD, the one that does the work.

In order to focus more closely on the key step of the example, we detour to show where the program obtains its database of operation relations. When we return to the

example, we show how the test generator finds this particular instance of ALU/ADD and how the component tests are transformed.

4.7 Computing Operation Relationships via Simulation

Operation relations can be supplied to a test generator in several ways depending upon the type of circuit and the design environment the test generator sits in. For sequential circuits that execute relatively few operations, our solution is to compute the operation relationships using symbolic simulation of circuit behavior.

Symbolic simulation is the process of propagating variables and algebraic expressions as well as numbers through the circuit. Doing this allows a single simulated operation to stand for an equivalence class of similar operations. For example, a LOAD instruction with symbolic data can stand for a LOAD of any specific constant. Using symbolic simulation is important for two reasons: (i) the algebraic expressions that propagate through the circuit are what we turn into operation relations and (ii) simulating equivalence classes of behavior rather than specific behaviors reduces the number of simulation runs needed and the size of the database that holds the results. First we describe how the simulator works and then we show how operation relations are extracted from the results.

4.7.1 How the Simulator Works

DB-TG uses an event-driven simulator that is inspired by a similar program called MARS [singh83]. It takes as input a circuit schematic, behavioral models of the components and descriptions of the instructions and produces as output a set of simulation traces, called **behavior graphs**, that describe what happens inside the circuit as the instructions execute. These behavior graphs are an explicit representation of the circuit's designed behavior, i.e., the patterns of activity it was designed to carry out.

Before starting, the simulator initializes all memory cells in the circuit. In order to simulate the execution of an instruction in the middle of an arbitrary instruction stream, most of the memory cells are pre-loaded with variables. These variables correspond to the values that would have been left by a previous instruction (had that instruction been simulated too). For example, the accumulator is pre-loaded with ?ac and the stack pointer is pre-loaded with ?sp (the variable names are a debugging aid and are specified in the circuit description).

Certain registers are special and are pre-loaded with values that are contained in the circuit description. The MAC-1 MicroProgram Counter (uPC) is special because its value defines the instruction cycle. uPC is pre-loaded with a value corresponding to the beginning of an instruction cycle, which subsequently causes the processor to execute an instruction when the simulation starts. uPC also finishes each simulation run with that same value. This register is the only special case in the MAC-1.

As the simulator runs, the circuit model interacts with a program that emulates the environment in which the circuit sits. For the MAC-1, this program drives the circuit clock up and down and responds to bus cycles. The circuit description also contains one of these programs per instruction.

The simulator is event-driven: when the value on a node changes the components that use that value "wake up," decide what operation they should perform and recompute their outputs accordingly. Each component assembles an algebraic expression describing its operation and one or more expressions describing its output values as functions of its input values. These expressions are then run through a set of algebraic simplification rules and written to the outputs, thereby waking up other components.

Figure 4.7 shows part of the behavior graph for the MAC-1/SUM operation. The accumulator contains ?ac at time 0, ?data is read from the databus at time 56 and the sum of ?ac and ?data is written into the accumulator at time 72. Here, the ALU executes an ADD operation at time 68. It receives two expressions ?ac and ?data as inputs, adding them under the control of its operation input and outputs the expression (?data $\oplus_{16}$?ac). The value on the operation input (add) beginning at time 68 comes from the microprogram ROM and ultimately from the microprogram counter.

Behavior graphs are actually implemented with two levels: one, describing detailed propagation, is built by repeatedly composing the component functions to build up large expressions. A second level results from the structure of the simplified expressions that are propagated through the circuit. Much of the benefit of behavior graphs lies in the observation that these algebraic expressions are an alternative representation of dataflow that is often *simpler than the circuit structure.*

Figure 4.8 illustrates this idea. Consider a circuit that simply moves values from one end to the other without changing them (at a high level of abstraction, many communications networks can be viewed this way). The lower line of dots in the figure corresponds to a value propagating step-by-step through the circuit. The upper level corresponds to simplified values associated with the circuit nodes. To take a step, say from node 1 to node 2, the simulator composes the already simplified value on node 1 (labeled a) with the component function connecting node 1 to node 2 (labeled b)

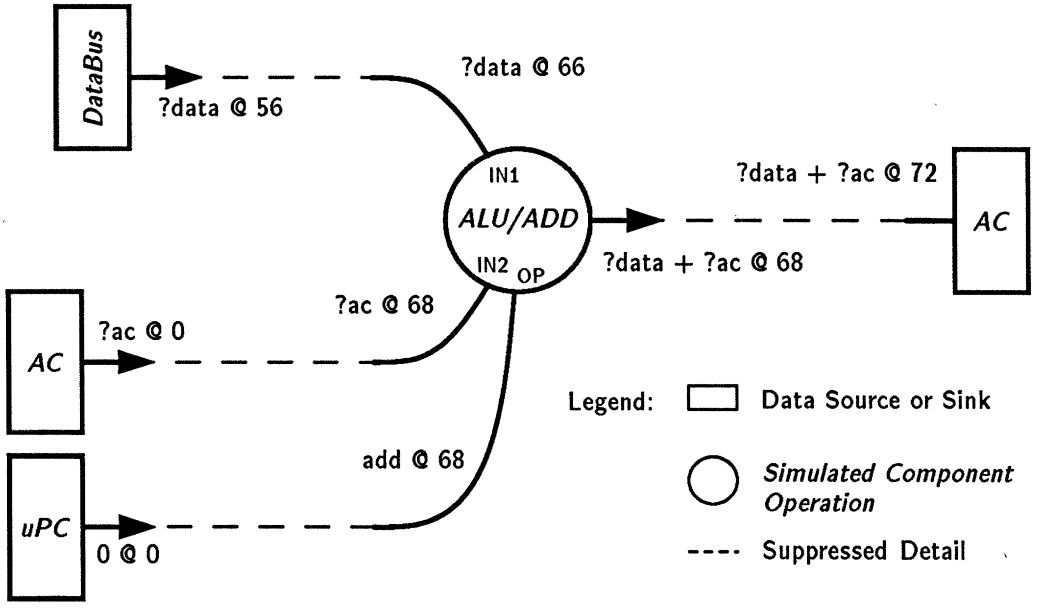


Figure 4.7: *The Behavior Graph for MAC-1/SUM. Time and data flow from left to right through the figure. The value of a node is timestamped, e.g., a node value of* data@time *indicates that the node changed to that value at the simulated time. Node values persist until they are caused to change by other circuit activity. Portions of the graph have been omitted; the full MAC-1/SUM behavior graph contains roughly 500 nodes corresponding to components performing operations or nodes changing value.*

and simplifies the result (labeled c).

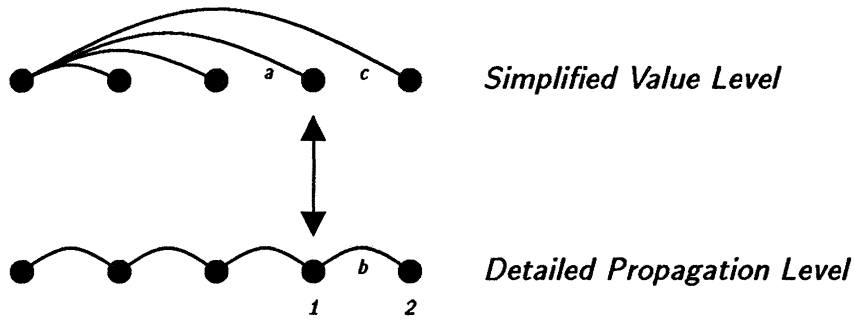


Figure 4.8: *Behavior graphs have two levels*

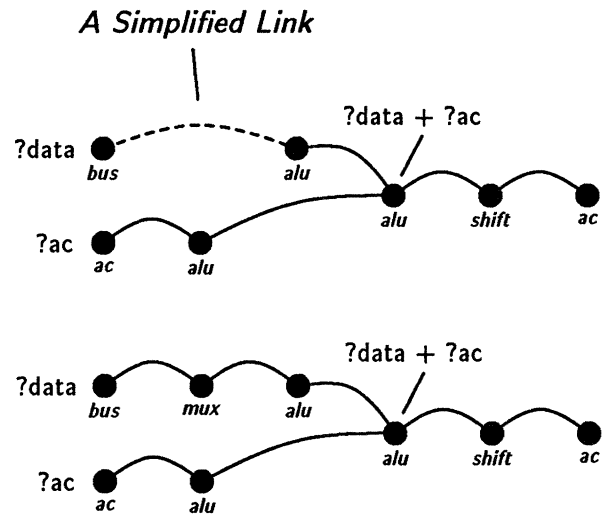


Figure 4.9: *(Part of) the two levels for the MAC-1/SUM behavior graph*

Since in this case a and b are identities, then c is one too. At the level of the lisp implementation, c is a pointer directly to the circuit input (actually to the value that was placed there at the start of simulation). The directness of this link is key: simulation and simplification has "recognized" that this circuit is simply moving a value around without changing it. Thus, by using the simplified values, little work will need be done later to propagate back from a component during test generation. Note that for any circuit the simplified values are never more complex than transcriptions of the circuit structure (assuming that the rewrite rules never make values more

complex).

Figure 4.9 instantiates this idea for the ALU example. The lower half of the figure shows ?data and ?ac propagating through the circuit to the ALU inputs where they are added and propagated to the accumulator. The upper half of the figure shows the simplified values; note that the two steps taken by ?data from bus to mux to alu are simplified to one step.[5]

4.7.2 Continuing the Example: Extracting Operation Relations

Now we return to the example of testing the ALU addition operation and the perspective of operation relations. The ALU component test calls the ADDER component test and specifies a mapping between ALU I/O pins and ADDER I/O pins so that the ADDER test knows how to embed itself. The ADDER component test is primitive, containing a test operation and test data. It searches the operation relation database for an instance of ALU/ADD that can be used to create a test.

The operation relation database shown in figure 4.2 consists of two sections: (i) a database of frames explicitly describing operations and operation relations like those shown earlier, and (ii) a database of behavior graphs that implicitly contain operation relations. When a component test looks for instances of component operations, it searches the first database. If that fails, it then searches the behavior graphs.

The purpose of the first database is to hold "advice" from the circuit designer, the test engineer or some other source in the form of useful operation relations. In some cases, this advice is sufficient for generating tests and symbolic simulation is unnecessary (the simulator is run on demand, triggered by search queries). If searching the frame database fails to turn up anything, the DB-TG runs the simulator and uses the behavior graphs to derive the operation relations.

Causal connections appear in the behavior graphs implicitly: the circuit operation is causally connected to every component operation that appears in its behavior graph. Therefore, to find circuit operations that cause the ALU to add numbers, we can search each behavior graph for instances of ALU/ADD.

Extracting parameter relations from a behavior graph is more involved. (Suppose, as it searches the behavior graphs, the component test comes across the instance of ALU/ADD shown in figure 4.7.) There are two interesting cases: (i) relations between

[5]Several "unessential" components have been omitted from the MAC-1 figure to simplify the discussion. For example, there are latches on the outputs of the register file. These components are reflected in the full behavior graph with a correspondingly greater reduction in steps.

circuit inputs and component inputs, and (ii) relations between circuit outputs and component outputs.

Parameter relations between circuit inputs and component inputs are the simplified values appearing on the component inputs, e.g., ?ac and ?data. More precisely, the simulated values denote functions of the primary inputs, and the parameter relations result from setting the expressions equal to the component's internal names for values on those inputs. For example:

General Case			*This Example*		
$f_1(CircuitInputs)$	=	$ComponentInput_1$	?data	=	?in1
$f_2(CircuitInputs)$	=	$ComponentInput_2$	?ac	=	?in2

Parameter relations between circuit inputs and component inputs tend to be simple, because the values are in simplest form (with respect to the rewrite rules).

Parameter relations between component outputs and circuit outputs are more difficult to extract from the behavior graph. We solve this problem using the unsimplified versions of each node value, i.e., the lower level of the behavior graph. We need the unsimplified versions, because they contain complete records of how they were produced, i.e., what values were combined to produce them and how those values moved through the circuit.

To compute the relation between a circuit output and a component output, we first fetch the unsimplified expression describing the circuit output's value. In this case, the circuit output in question is the data bus. Next, we find the subexpression corresponding to the component output, in this case (?data $\oplus_{16}$?ac), and substitute in a variable for this expression. At this point, we have a complex expression describing the parameter relationship we want. We then pass it through the simplifier. Here is the result:

General Case			*This Example*		
$f_1(ComponentOutputs)$	=	$CircuitOutput_1$	?output	=	?sum
$f_2(ComponentOutputs)$	=	$CircuitOutput_2$			

The output of a particular component operation can be related to values on several circuit outputs. DB-TG decides what circuit outputs it might be related to by walking forward through standard dependency records contained in the behavior graph. Connection via the dependency records is a very weak, necessary condition on the component output value and the circuit output value being related. Finding the right

subexpression inside the unsimplified circuit output value is a stronger condition, and finding the variable after simplification is a sufficient condition.

Computing the relation between a circuit output and a component output is somewhat expensive, so DB-TG does it on demand, i.e., only when a particular instance of a component operation is identified as potentially useful and the input parameter relations have already been handled.

4.8 Solving the Embedding Problem

Transforming the operations specified by a primitive component test into circuit operations involves solving a set of simultaneous equations. This process occurs in two steps: (i) substitute placeholders for the test data into the operation relations and solve for the parameters of the circuit operation and (ii) repeatedly substitute in lines from the actual test data. Operation relations can contain combinations of boolean and arithmetic functions, selection, bit-field extraction, concatenation and other combinators that appear in digital circuits, so solving for the circuit operation parameters can be expensive (but no worse than propagating through circuit structure). Doing this once with placeholders rather than repeatedly with each line of test data saves considerable time.[6]

After substituting in placeholders ADDEND-1, ADDEND-2 and SUM-1 for the test data, the equation solver solves for circuit operation parameters like so:

General Case			*This Example*		
$f_1(\mathit{CircuitInputs})$	=	ADDEND-1	?data	=	ADDEND-1
$f_2(\mathit{CircuitInputs})$	=	ADDEND-2	?ac	=	ADDEND-2
f_3(SUM-1)	=	$\mathit{CircuitOutput}_2$	SUM-1	=	?sum

Values for the circuit inputs are computed by inverting f_1 and f_2. This task is accomplished by a set of rules extending the Prolog unifier with an equality theory, i.e., a set of rules describing under what circumstances pairs of algebraic expressions are equal. The pairs of expressions are the left-hand and right-hand sides of the parameter relations. To invert f_1 and f_2, these rules take them apart level-by-level, moving parts of the expression from the right to the left. These functions need not be one-to-one in general, so the program must make choices about how to do the

[6]Sometimes, however, general solutions cannot be found for the placeholders where they could be found for the specific test data. This issue is discussed in section 5.3.1.4.

inversion. Since the functions may share inputs, choices must be made consistently. This process can be viewed as a kind of line justification through the structure of these algebraic expressions (this idea is expanded in section 5.4.1).

The value(s) for circuit output(s) is produced directly by substitution, however, DB-TG must check that the output(s) carries enough information to distinguish between a working and faulty component. The program currently checks the stronger condition that the output value be invertible, i.e., that it carry complete information about the component output(s). It does this check using the expression inversion mechanism described above.

Here is the result of solving for ?ac, ?data and ?sum and substituting into MAC-1/SUM. This instance of MAC-1/SUM differs from the generic description of this operation (page 101) in the slots marked with $\Leftarrow$, where placeholders for the actual test data appear.

MAC-1/SUM	Before State:	Accumulator	=	ADDEND-2	$\Leftarrow$
		ProgramCounter	=	?pc	
	Inputs:	DataBus	=	(ADD ?addr)	
		DataBus	=	ADDEND-1	$\Leftarrow$
	Outputs:	AddrBus	=	?pc	
		AddrBus	=	?addr	
	After State:	Accumulator	=	SUM-1	$\Leftarrow$
		ProgramCounter	=	?pc1	
	Relations:	?sum	=	?data $\oplus_{16}$?ac	
		?pc1	=	?pc $\oplus_{16}$ 1	

Before continuing with the example, consider what would have happened if the test generator had found one of the many other instances of ALU/ADD in the behavior graphs. Most of the other instances increment the program counter. A constant 1 appears at one of the ALU inputs in these instances. Working with the operation relations extracted from these instances, the equation solver fails (i.e., ADDEND-1 = 1 cannot be solved because ADDEND-1 must stand for any possible test data), hence test generator cannot use these instances and continues searching. Only a few instances of ALU/ADD are general enough to work: one instance that we have been using as an example is inside MAC-1/SUM and several other instances lie inside instructions that do address computations. We continue the example with the one in MAC-1/SUM.

4.9 Planning Control and Observe Sequences

At this point, the crucial steps have been taken, but the test generator has not quite finished designing a complete test. The test generator has ADDEND-2 coming from the accumulator and SUM-1 going there, but the accumulator is neither directly controllable nor directly observable. Extra operations must be used to control and observe it.

The answers are simple for a person: the LOAD instruction can put ADDEND-2 into the accumulator and the STORE instruction can write SUM-1 to the bus where the tester can observe it. How might a test generator identify these instructions as the relevant ones and add them to the SUM instruction to complete the test?

In DB-TG this is the task of the **State Planner**. The State Planner generates sequences of operations (in this case sequences of instructions) that set up the circuit state so that the single test operation will start with the right data. A second job of the State Planner is to move any results left in state registers by the test operation circuit outputs where they can be observed.

The State Planner is invoked when solving the operation relations results in at least one assignment to a state register. In this example, two values were assigned to the accumulator: ADDEND-2 at the start of the test operation and SUM-1 at the end of the operation. If solving the parameter relations does not assign a value to a state register, then that register's value does not affect the test, hence its value need not be controlled or observed.

The State Planner searches through the space of instruction sequences to find ones that move data around properly. It does this using simple STRIPS-like planning technology [fikes71]: the planner is implemented as a bounded depth-first search that moves forward in time to observe state registers and backward in time to control them. The interesting issue is how we supply the planner with descriptions of its operators.

One straightforward solution is to give the program another library that describes the overall effect each operation has on the circuit's state registers. This library would describe circuit behavior at the familiar register transfer level. This is what we do, except that DB-TG itself constructs the library by summarizing the behavior graphs. Summarization is done in two stages: (i) determining which state registers are relevant, and (ii) determining the relationships between values in those registers and values on the circuit I/O pins.

State registers are considered relevant with respect to a particular set of circuit operations. The program separates the relevant registers from the irrelevant ones in

order to reduce the amount of information the state planner must handle.

A state register is considered relevant if two conditions hold. First, some output of some operation must depend upon the value of the register. If no output ever depends on the register's value, then the register can safely be ignored. This can occur when considering a subset of the circuit operations, e.g., the stack pointer is irrelevant from the perspective of the simple arithmetic instructions. Second, the register must hold different values at the beginning and at the end of at least one operation. Note that this is more restrictive than saying the register's value must change. This condition selects registers whose value changes can be observed at the boundaries of the circuit operations: register changes inside the circuit operations are abstracted away. For example, this condition abstracts away the registers in the MAC-1's microengine, since they return to their instruction-fetch state at the end of every instruction.

Given a processor and its instruction set, these two conditions select the programmer-accessible registers. Other registers, in particular those in a micro-engine, are abstracted away. Once the program identifies the relevant registers, determining the relationships between register values involves collecting the algebraic expressions appearing on circuit outputs and those in relevant registers at the end of each simulation run.

Figure 4.10 shows an Effects Summary of MAC-1/LOAD. Values that enter the circuit inputs are at the top of the figure and values that leave the outputs are at the bottom. The values of state registers before the instruction starts are on the left and their values after the instruction finishes are on the right. The dashed lines indicate how values move and are transformed during the instruction. For example, the program counter's value is incremented and is also written to an output, while the stack pointer remains unchanged. It is coincidence that in this instruction no two values are combined (e.g., added), so none of the dataflow lines join together. The summaries include just the information in the figure, i.e., the relevant inputs, outputs and state registers, plus the actual functions that transform the values as they move.

The visual layout of figure 4.10 is intended to give a feel for what this process is like. Notice that all dataflow is downward and to the right. If the test generator has left any values in state registers (like SUM-1 in the accumulator) then the state planner attempts to fit together a sequence of summaries such that a continuous line is formed from the state register down and right to an output. Similarly, if any values are required to be in a state register at the start of the test (like ADDEND-1 in the accumulator), the planner fits together a sequence of summaries that allow the value to flow down from one of the inputs.[7] Figure 4.11 illustrates this process.

[7]Some operations can also create some values internally, hence these values need not come all the

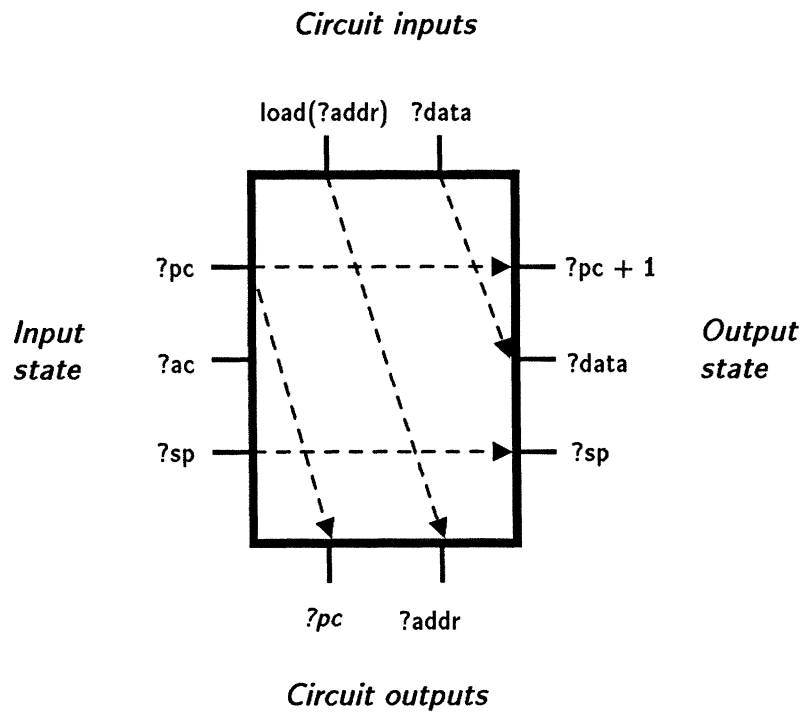


Figure 4.10: *Effects Summary for MAC-1/LOAD*

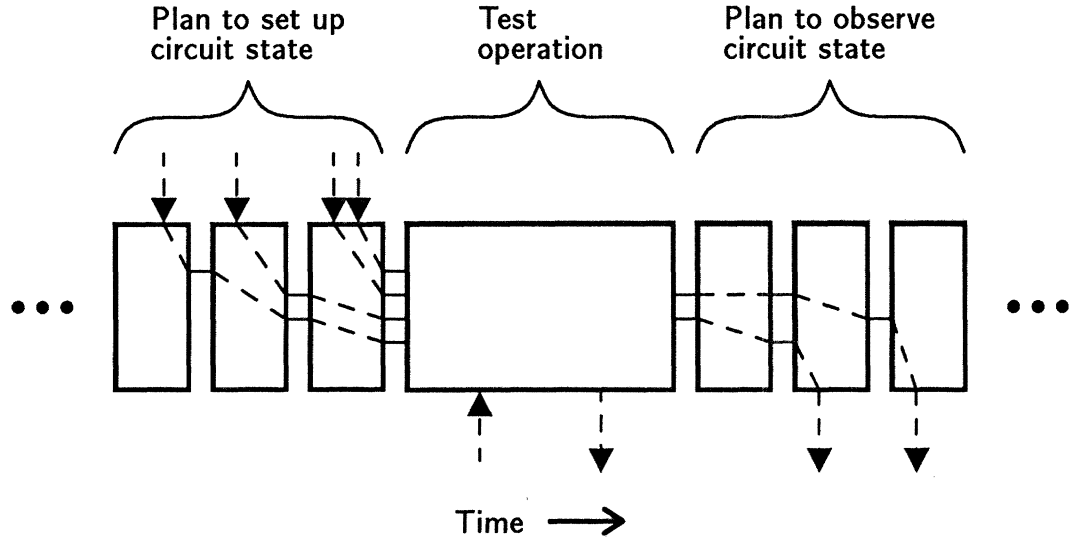


Figure 4.11: *State Planning Using Operation Effects Summaries*

The state planner must handle one final detail as it fits summaries together: it must ensure that the continuous lines running from input to state register and from state register to output correspond to invertible functions. This requirement guarantees that the placeholder can be transformed to appropriate values on the inputs and outputs by "pushing them through" the continuous lines. Requiring an invertible function on the output lines is more strict than necessary, but will help when the test generator is extended in chapter 6.[8]

Figure 4.12 shows the result of state planning for the example of testing the ALU. In this example, the control and observe sequences are only one instruction long. Both sequences involved identity functions, so ADDEND-2 and SUM-1 appear unchanged in the LOAD and STORE instructions (marked by $\Leftarrow$'s).

This sequence of three instructions is a test for ALU/ADD. We omit the final step of substituting in the test data (from figure 4.4). Were we to do this, there would be 24 circuit operations. Note that this solution generated by the program is the same one that is obvious to human test experts but not obvious to classical test generation methods.

4.10 Experimental Results

The MAC-1 has 16 components (not all of which appear in the simplified block diagram) equivalent to roughly 6500 gates. Each instruction is approximately 50 clock cycles long[9] and takes about 10 seconds of real time on a Symbolics 3640 to simulate. Test generation for this circuit takes 6 minutes, including both the time taken for successfully creating tests for some components and failing to do so for others. Figure 4.13 highlights the components for which the designed-behavior test generator successfully instantiated library tests.

Fault simulation reveals that the instantiated tests from the component library cover 85% of the gate-level stuck-at and open faults in the MAC-1. Measuring this takes a commercial quality fault simulator 30 minutes of cpu time on a SUN-2 com-

way from an input.

[8]To achieve fault coverage, the output function must map all erroneous component outputs to values different from the correct value. If the fault model limits the wrong values the component can possibly put out, then this function need not be invertible. However, an invertible output function can provide more information about the nature of a component fault.

[9]50 clocks is a long instruction cycle. This is due to the lack of circuitry for extracting opcodes directly and branching to the appropriate microcode sequence in this sample circuit used for teaching.

MAC-1/LOAD	Before State:	Accumulator	=	?ac
		ProgramCounter	=	?pc-1
	Inputs:	DataBus	=	(LOAD ?addr)
		DataBus	=	ADDEND-2
	Outputs:	AddrBus	=	?pc-1
		AddrBus	=	?addr
	After State:	Accumulator	=	ADDEND-2
		ProgramCounter	=	?pc

MAC-1/SUM	Before State:	Accumulator	=	ADDEND-2
		ProgramCounter	=	?pc
	Inputs:	DataBus	=	(SUM ?addr)
		DataBus	=	ADDEND-1
	Outputs:	AddrBus	=	?pc
		AddrBus	=	?addr
	After State:	Accumulator	=	SUM-1
		ProgramCounter	=	$?pc \oplus_{16} 1$

MAC-1/STORE	Before State:	Accumulator	=	SUM-1
		ProgramCounter	=	$?pc \oplus_{16} 1$
	Inputs:	DataBus	=	(STOD ?addr)
	Outputs:	AddrBus	=	$?pc \oplus_{16} 1$
		AddrBus	=	?addr
		DataBus	=	SUM-1
	After State:	Accumulator	=	?data
		ProgramCounter	=	$?pc \oplus_{16} 2$

Figure 4.12: *Final embedding of a test for ALU/ADD*

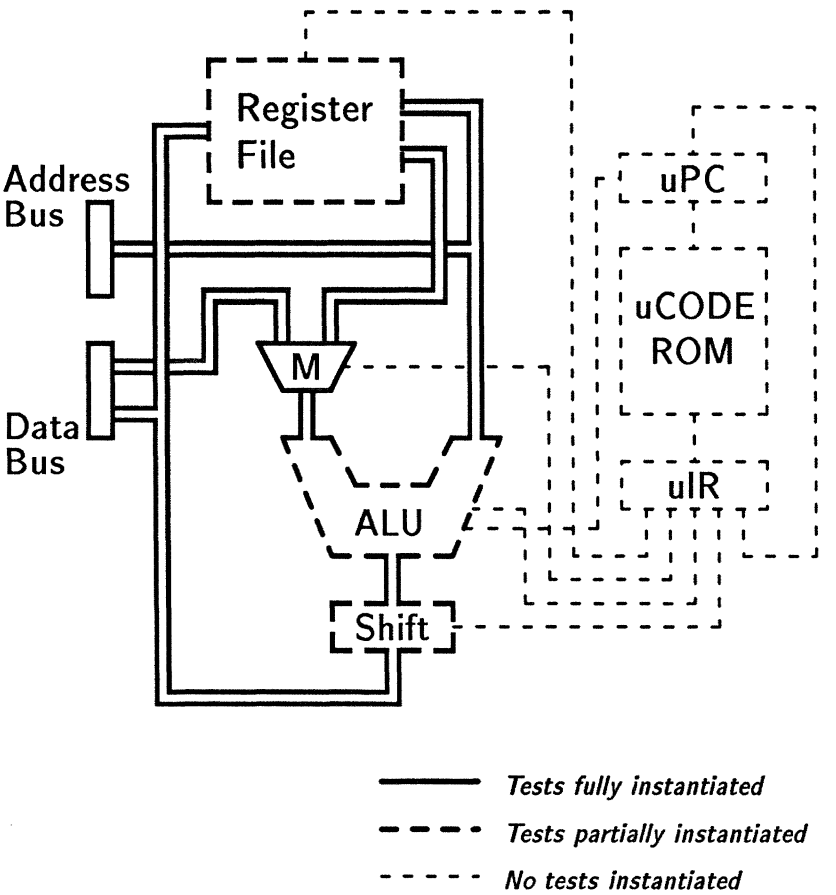


Figure 4.13: *Test Generation Results*

puter (therefore generating these tests takes substantially less time than fault simulating them). The tests actually catch somewhat more faults than they are designed for, because exercising components in the datapath partially exercises the sequencer too.

In a nutshell, the vanilla version of the designed-behavior test generator quickly instantiates tests for most of the components in the datapath, but it fails on the components in the microsequencer. This is good enough to be interesting – since the classical techniques do not work for such circuits – this but not good enough to solve the problem by itself. After analyzing why DB-TG fails, we will look at augmentations of its basic test generation strategy that raise coverage figures to 97%.

Note that DB-TG successfully tests the datapath despite the control and feedback paths between it and the sequencer. It is not the case that DB-TG fails to control some component(s) and therefore cannot control other components downstream. Something more interesting is going on, and we cover this issue in depth in chapter 5.

Success with the datapath and failure with the sequencer matches our expert's intuition about which parts of this circuit are easy to test and which are hard. The datapath is easy for people to manipulate, and the program finds it relatively easy too. The expert says the sequencer is testable with considerably more effort, but should be modified, if possible, to simplify testing and to reduce the test program's sensitivity to changes in the microcode. Thus a test generator's failure can be useful: as long as the test generator successfully handles the "easy" problems, then its failure points out areas of the circuit where design for testability techniques should be applied. For the origins and further development of this idea, see [wu88].

4.11 DB-TG: Additional Details

This section gives some additional details of the implementation.

4.11.1 Modeling and Simulation

We use a simple schematic entry system running on the Lisp Machine to enter and debug circuit descriptions. Figure 4.14 shows the full MAC-1 schematic. The simulator is tied closely with this graphical system, and we can probe nodes in the diagram to see their values in a behavior graph, see time histories of single nodes, single step the simulator seeing all node values and other useful debugging operations.

The simulator model is driven by programs describing how a tester would interact

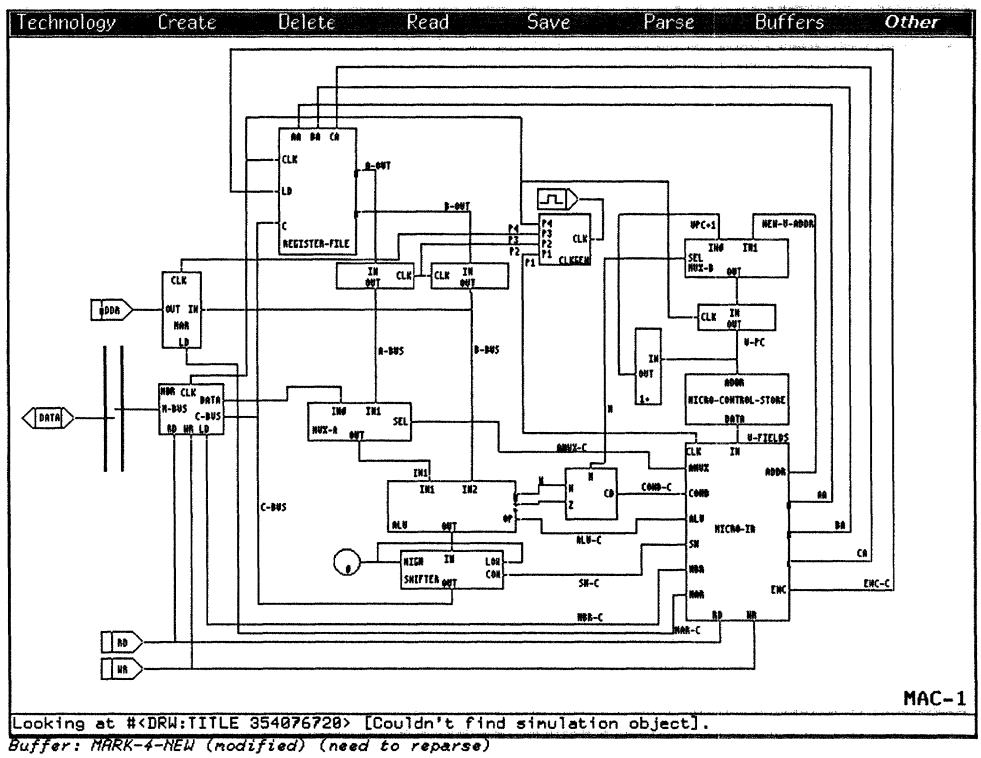


Figure 4.14: *A schematic entry system showing the full model for the MAC-1 processor.*

with the circuit to make it execute its operations. For instance, the program in figure 4.15 specifies how to drive the MAC-1 to execute a LOAD instruction. This program is written in simple embedded language that uses multitasking (e.g., if forks off a process to drive the clock on line 6) and synchronization primitives to interact with the circuit. The >> function references circuit nodes by name.

4.11.2 Focusing Search Through the Behavior Graphs

DB-TG focuses the process of searching the behavior graphs in two ways: (i) by not searching portions of the graphs that cannot contain solutions and (ii) by searching "simple" sets of behavior graphs before more ones.

DB-TG prunes the search space by summarizing features of the behavior graphs as they are generated and then skipping behavior graphs during test generation that do not have the appropriate features. The **Component Activation Summary** lists

```
1.   (define-stimulation-pattern (LOAD ?ADDR)
2.                               (:circuit-name 'mac-1
3.                                :operation-pattern '(LOAD
4.                                                     ((DATA ?data))
5.                                                     ((PC ?pc) (ADDR ?addr))))
6.     (with-clock-process-on-node (>> 'mac-1 'CLOCK)
7.       (tight-sequence
8.         (mac-1-read-cycle pc (make-LOAD '?addr))
9.         (mac-1-read-cycle '?addr '?data)
10.        (mac-1-clean-finish))))

    ;;; A wrapper for read-cycle that specifies which bus nodes it should
    ;;; look at.
11. (defun MAC-1-READ-CYCLE (address data)
12.   (read-cycle (>> 'mac-1 'rd) (>> 'mac-1 'addr) address (>> 'mac-1 'data) data))

    ;;; The general read-cycle primitive.  This function uses
    ;;; wait-for-node-to-assume-value to synchronize with the simulator.
13. (defun READ-CYCLE (rd-node addr-node address data-node data)
14.   (wait-for-node-to-assume-value rd-node 0 "Read Cycle A")
15.   (pause 5)
16.   (observe addr-node address)
17.   (assign data-node data)
18.   (wait-for-node-to-assume-value rd-node 1 "Read Cycle B")
19.   (assign data-node 'Z))
```

Figure 4.15: *This program specifies how to drive the MAC-1 to execute a LOAD instruction*

the kinds of component operations that occur within each behavior graph. The program constructs a component activation summary immediately after creating each behavior graph and checks the summary every time it starts to search for a simulated component operation. If the desired operation type does not appear in a graph's summary, then the graph need not be searched. For instance, the ALU/ADD operation appears several times with different arguments in the MAC-1/SUM behavior graph, but ALU/INVERT does not, therefore ALU/ADD appears in the summary but ALU/INVERT does not, and the program can skip this behavior graph if it is looking for simulated instances of ALU/INVERT.

DB-TG also focuses search by first looking for solutions using the circuit's "simple" operations, and then trying more and more complex or rare operations. In the MAC-1, for example, DB-TG first tries to embed tests using the core instructions LOAD, STORE, ADD, SUBTRACT and JUMP. If unsuccessful, it broadens its search to include the full instruction set. This strategy focuses search first on simple operations which are usually sufficient for testing most of a circuit.

Using a smaller set of circuit operations also simplifies the abstract descriptions (called Effects Summaries) used by the State Planner to construct sequences of circuit operations. The complexity of these descriptions depends primarily upon the amount of observable state in the circuit, and the amount of observable state in turn depends on the set of circuit operations that will be used during testing. For example, the stack register is not affected by or observable via any of the five instructions in the core set above, therefore the stack register is not observable state from the perspective of the core instructions. Only observable state is included in the Effects Summaries used by the State Planner. Currently, the user tells DB-TG which subsets of the circuit operations are useful and in which order to try them.

4.11.3 Relationships Between Component Operations

A behavior graph contains relationships between pairs of component operations as well as between many component operations and one circuit operation. These relationships between component operations are useful for solving embedding problems involving sequential components.

When the component under test is sequential, it can be critical that the exact sequence of test operations is executed in order and *with no other operations interposed.* That is because a library test for a sequential component carefully manipulates the component's internal state, and extra operations inserted in the middle of the sequence could render the test invalid. This can even occur when testing a combinational

component if the library test treats the component as if it were sequential, e.g., one that accounts for faults that could turn a (working) combinational component into a (faulty) sequential one. For instance, some open-circuit faults in MOS circuits can cause state behavior because floating circuit nodes can hold charge for a time. The library tests for multiplexors and several other components account for this possibility.

This kind of restriction is expressed in DB-TG using a compound component test: the compound test first finds a simulated instance of an appropriate component operation, extracts the operation relations and instantiates a primitive test containing the "combinational" test data. The compound test then searches from the simulated instance forward and backward through the behavior graph to make sure the component performs no other operations that might interfere. It must also examine the component activation summaries to make sure that the circuit operations used to control and observe circuit state do not interfere either.

Using behavior graphs beyond simply extracting operation relations is reminiscent of a different view of the designed behavior approach presented in [shirley86]: there the test generator searchs behavior graphs for patterns of activity that would be useful during testing. These "patterns of activity" were described by predicates in a rich language involving data and timing relationships and the patterns could potentially match multiple component operations. This thesis, taking a simpler and more direct approach, emphasizes of operation relations, which are by far the most important and useful kind of "pattern of activity" contained in the behavior graphs. Emphasizing operation relations also calls attention to the fact that part/whole relationships connecting circuit and component operations could potentially come from other sources, e.g., a design synthesis tool, and not just from symbolic simulation.

4.12 Review of DB-TG

DB-TG can be viewed at two levels of detail. The simpler version involves three steps: (i) the component test problem: solved here by looking up tests in a library supplied by an expert test programmer; (ii) the operation relation problem: derive relationships between the component operations mentioned in the test and circuit operations; and (iii) the embedding problem: transform internal component operations into directly executable circuit operations using the operation relations. Solving the operation relation problem is the key step where the test generator derives descriptions of circuit behavior for later use. The resulting operation relation database captures a "global view" of circuit behavior that is abstracted away from much of the detailed step-by-step dataflow and timing contained in circuit descriptions used by conventional test

generators.

While any source of operation relations is welcome (e.g., hints from the designer), we rely largely on computing operation relations by simulation. The simulator uses circuit schematics and component simulation models that can propagate algebraic expressions. In the process of refining the algorithm to use simulation, we partially combine the operation relation problem and the embedding problem. This second, more detailed version of the method takes the following information as input:

1. A Component Test Library: operations and specific test data supplied by a test expert
2. Component models for symbolic simulation
3. A Circuit Description consisting of
   - A block diagram schematic: component types and interconnect
   - A circuit interface specification: programs describing how a tester can drive the circuit to cause operations to occur.

and produces sequences of circuit operations as output. The program follows these steps (see figure 4.16):

1. Simulate the circuit's designed behavior: The program simulates the activity inside the circuit during each of its operations, capturing this activity as a set of behavior graphs.
2. Summarize the behavior graphs: The program summarizes the behavior graphs to create a set of abstract descriptions of the input and output of each operation.
3. Embed component tests: For each component in the circuit, the test generator tries to embed a test fetched from the component test library. Embedding involves (i) searching the behavior graphs for an appropriate instance of a component operation, (ii) extracting the relations between that component operation and the circuit operation that causes it, and (iii) substituting the test data into the relations solving for the parameters of the circuit operation. If this works, executing the circuit operation with these parameters will cause the component operation to occur. The circuit operation is now called the test operation.
4. Control and observe circuit state: Sometimes solving the operation relations causes values to be assigned to a state register. This means that either the

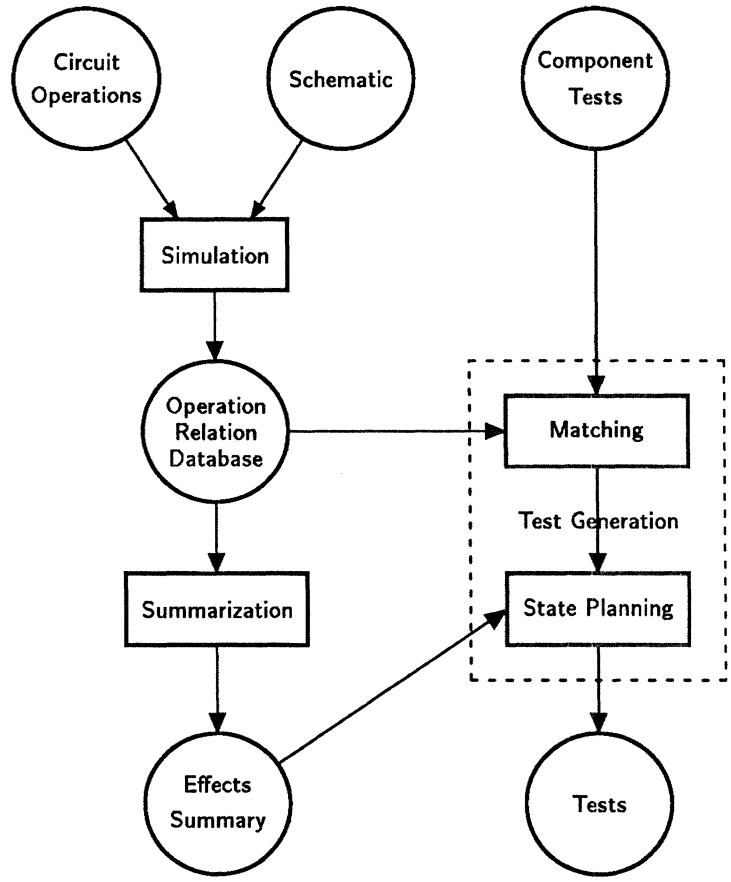


Figure 4.16: *Information Flow through the Test Generator*

value of the register must be preset before the test operation is executed or that the register's value must be observed after the test operation has finished. A simple STRIPS-like planner constructs sequences of operations that control and observe the circuit state.

4.13 Conclusion

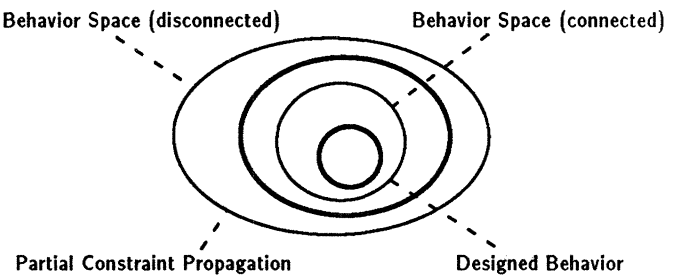
This chapter presented a simple testing problem and argued that a particular kind of abstract knowledge about circuit behavior – relationships between circuit and com-

ponent operations – can help to solve it. This observation formed the foundation of the DB-TG test generator.

Operation relations were supplied to the program either directly (e.g., by the designer) or computed from circuit structure and component behavior via symbolic simulation. Using symbolic simulation is based on an idea about planning called simulate and match. This particular strategy for computing operation relationships focuses the test generator on known-achievable behavior rather than on potentially achievable behavior that must be verified via complex reasoning.

This strategy also embodies the key ideas in the expert test generation method described in section 3.2. We, as algorithm designers, have identified what can be done – the circuit operations – and what must be done – stay within the designed behavior. We have identified the test objectives – embed component tests – and provided a strategy for doing the embedding. From this perspective, the novel step is restricting the test generator to what must be done, i.e., to follow established conventions between the circuit and its environment.

The next chapter discusses the advantages and drawbacks of DB-TG, explores the ideas underlying the test generator further and analyzes their individual contributions and relationships.



Chapter 5
Analysis

Summary: DB-TG is based on four ideas about test generation: (i) the designed behavior heuristic, (ii) embedding expert-supplied component tests, (iii) operation relations, and (iv) computing operation relations by simulation and matching. This chapter explores each of these ideas from the perspective of its effect on the test generator's soundness, completeness and efficiency.

DB-TG is a heuristic solution: it is neither guaranteed sound nor complete. There are situations where it can produce incorrect tests (it warns when this may have happened), and there are situations where it can fail to find a test. Sound and complete algorithms exist, but they are unusably slow for complex sequential circuits. We need an effective, fast, heuristic test generator for these circuits, i.e., the kind of solution human test experts currently provide. This chapter argues that DB-TG is such a solution.

5.1 Introduction

DB-TG is based on four ideas about representing testing expertise, representing circuit behavior and search in problem solving: (i) the designed behavior heuristic, (ii) embedding pre-written component tests, (iii) operation relations and (iv) simulate and match. Each idea is judged by its effect on three properties of the test generator: soundness, completeness and efficiency. A test generator is **sound** if the tests it produces are guaranteed to detect the faults they are supposed to detect. When a test generator is sound, its output can be used without fault simulation or other forms of independent verification. A test generator is **complete** if it is guaranteed to find a test for any fault if a test exists. When a test generator is complete, its failure indicates that no test exists. Running the program a little longer or modifying the

algorithm in some way would not have turned one up.[1] A test generator is **efficient** if it can generate tests in a timely fashion. An efficient test generator can, for instance, be used as an analysis tool during design to give feedback on the testability of the circuit. The tests generated by the program are themselves efficient if they can be applied to the circuit quickly.

Strong connections between the ideas prevent any linear text presentation from working completely; perhaps the best solution would be to arrange this chapter as a matrix covering the four major ideas and the three criterion for judging each idea. The current organization – a section for each idea with subsections for each property – reflects a compromise with the result that two important themes arise several times: (i) mismatches exist between the granularity of component tests and the granularity of the behavior graphs, and (ii) tension exists between needing abstract circuit descriptions for speed and needing specific predictions of fault effects for accuracy. These themes reflect tensions between efficiency and completeness on one hand and between efficiency and soundness on the other.

The central results of the analysis are as follows. The primary advantages of DB-TG are:

- Focusing on designed (known-achievable) behavior rather than potential behavior reduces the size of the search space.
- Generating tests within a circuit's designed-behavior yields tests that use the circuit according to its interface specification. If the circuit were in turn a component in a larger circuit, these tests would likely be achievable, while it is extremely unlikely that tests outside the interface specification would be achievable.
- Operation relations are a compact representation of the circuitry surrounding a component. Embedding tests by propagating signals out through operation relations rather than through circuit structure recovers the cost of generating the relations by simulation and algebraic simplification.

The primary disadvantage of DB-TG is:

[1] Note that soundness and completeness properties are relative to the idealizations made when modeling the circuit and potential faults. No set of tests, for instance, can account for all possible ways a circuit might fail. The best we can say is that a set of tests completely covers a particular class of (hopefully likely) faults.

- When writing the component test library, it is impossible for a testing expert to anticipate all of the ways standard components can be used in a circuit. Consequently DB-TG sometimes fails to generate tests for components in situations where a test generator that does not embed pre-written tests could do so or where the expert could by adapting the library tests to the constraints of the circuit.

In the final analysis, DB-TG is neither guaranteed sound nor complete, i.e., there are situations where it can produce incorrect tests (it warns when this may have happened) and situations where it can fail to find a test. DB-TG should be viewed as a heuristic solution to an exponential search problem. Sound and complete algorithms currently exist, but to achieve these properties the algorithms must search exhaustively. This renders them unusably slow on sequential circuits. We need instead an effective, fast, heuristic solution, i.e., the kind of solution human test experts currently provide. This chapter covers the trade–offs and compromises involved and shows where giving up exhaustivity causes problems.

When this analysis is behind us, we will then ask what specific problems did the test generator have with the MAC-1 processor and why. The answers to these questions suggest several extensions to the test generator and ways of combining its strengths with the strengths of the classical test generation methods. These extensions are developed in chapter 6.

5.2 The Designed Behavior Heuristic

This section considers how the strategy of searching a circuit's designed behavior affects the performance of the test generator. We begin analyzing this heuristic by describing the structure of the search spaces involved in test generation. This leads to the designed behavior space, a subset of the search space that contains solutions to all testing problems. We therefore want to search for tests inside the designed behavior space, and we want to waste as little effort as possible looking outside.

5.2.1 Search Spaces for Test Generation

A test generation algorithm searches through a circuit's behaviors for one that constitutes a test. The important aspects of test generation search spaces are: (i) how large is the space; (ii) if the space is a subset of the circuit's potential behavior, then

does it contain solutions to all testing problems; and (iii) how much effort does the test generation algorithm waste on search outside the space?

We call the search space involved the circuit's *behavior space* and define it in terms of the behavior spaces of the components. The behavior space of a combinational component is simply its truth table, i.e., the sets of values its inputs and outputs can consistently hold. The behavior space of a *sequential* component is the set of *sequences* of values its inputs and outputs can consistently hold.

The cross product of the component behavior spaces is the circuit's *disconnected behavior space* (see figure 5.1). We refer to the space formed by taking the cross product as "disconnected," because it does not reflect the physical constraints imposed by connecting the components. Enforcing these constraints rules out possible behaviors. The connected behavior space – or *potential* behavior space – is the subset of the disconnected behavior space that contains all globally consistent assignments of values to the circuit nodes.

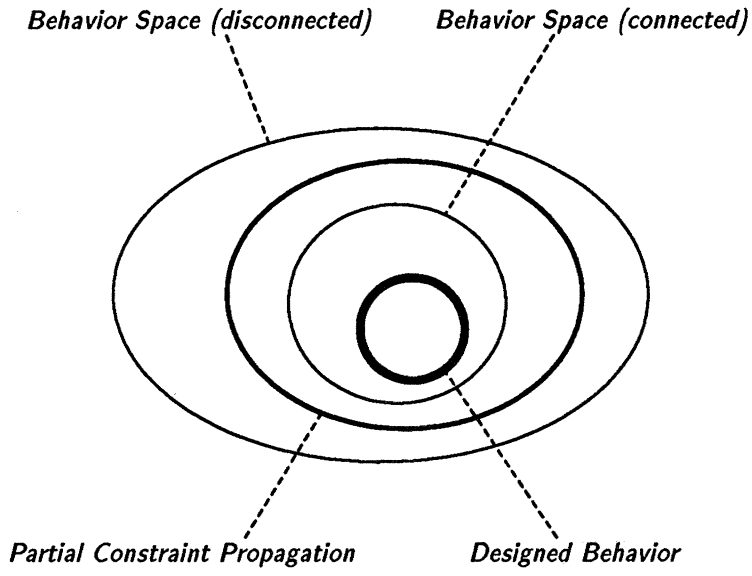


Figure 5.1: *Search Spaces for Test Generation*

Within the circuit's potential behavior space is its designed behavior space, i.e., those node assignments or sequences of node assignments that result from executing legal circuit operations. The legal operations are always a subset of the possible inputs. For many circuits, the legal operations are a *small* subset, so the designed

behavior is a small subset of the behavior space.

Constraint propagation techniques used in test generation are incomplete, i.e., they cannot detect all inconsistencies immediately as they construct tests. Classical test generators and DB-TG therefore search somewhat outside of their respective search spaces and backtrack inside when inconsistencies are noticed later. Although these constraint propagation techniques could be made complete (because the behaviors of digital circuits are finitely enumerable), the cost of doing so is prohibitive. It is cheaper to let the test generator backtrack somewhat while organizing search to minimize the amount of backtracking done.

5.2.1.1 The Boundary of Designed Behavior

How do we define the boundary of a circuit's designed-behavior? Clearly the circuit's normal operations are included, but are test modes and design-for-testability operations also included? What about behavior that is likely to change, e.g., behavior resulting from microcode?

We treat DFT operations and test modes as part of a circuit's designed behavior and handle them like the normal operations. An example of using a test mode and DFT features appears in section 6.5.

Handling behavior that is likely to change is a more subtle issue. Clearly we cannot define the space of designed behavior to include behavior that is unachievable now, simply because it *might* become achievable with the next circuit modification. Yet, at the same time, we would like DB-TG to produce tests that are relatively insensitive to planned design changes, e.g., we do not want to run the test generator again for every microcode change. To solve this problem, we rely on the designer to include DFT features for parts of the circuit that might change. Since the DFT features and the normal operations are unlikely to change, tests generated using them are likely to be insensitive to planned circuit modifications.

5.2.2 Completeness

While a circuit's designed behavior is smaller, is searching it sufficient to generate tests? We claim that it is.

Proposition 1 *The designed behavior space contains tests for every fault in a circuit.*

This proposition rests on defining faults as perturbations from a specification of correct behavior rather than as perturbations from a particular implementation. If a physical defect causes no perturbation from correct behavior, then a user will never notice it and it should not be considered a fault.

5.2.2.1 Implementation Defects May Not Cause Misbehavior

The distinction between faults as perturbations from correct behavior and faults as perturbations from an implementation is meaningful only if there are implementation faults that do not cause misbehavior. Physical defects that do not lie in electrically active areas certainly fall into this category. However more interesting cases exist if the implementation is not minimal, i.e., it contains unused functionality.

Circuit implementations are sometimes non-minimal because designers use components from component libraries (or from other designs) in order to save time and fabrication costs. When the component building blocks of the circuit are large units, e.g., an ALU, designers must sometimes choose a component that offers more functionality than they need. Unused functionality is implemented by unused structure inside the component, and defects in this unused structure are the defects that should not be treated as faults.

For example, figure 5.2 shows a plausible implementation for the MAC-1 ALU. This four function ALU is implemented here by four LS181's in a carry chain configuration. The combinational circuitry in C converts the 2-bit operation input (OP) into five control signals for the LS181's. The NEG output is the same as the high-order output bit. The ZER output is computed from the outputs (OUT) by a 16-input NOR gate (not shown in the figure).

There are several ways in which this implementation provides more functionality than necessary. First, the LS181 provides many more functional modes than the 4 needed by the MAC-1. Second, each LS181 contains unused generate and propagate circuitry.[2] Finally, the carry output of the high-order LS181 is unused. Physical defects that cause misbehavior only in this extra functionality are not faults.

[2]The generate and propagate inputs and outputs allow multiple LS181's to be connected together to form a large ALU. In simple usage, they are redundant with the carry input and output, but do their jobs more quickly when connecting many LS181's. However, using them requires adding an extra component. Here we connected the LS181's in the simpler carry-chain configuration.

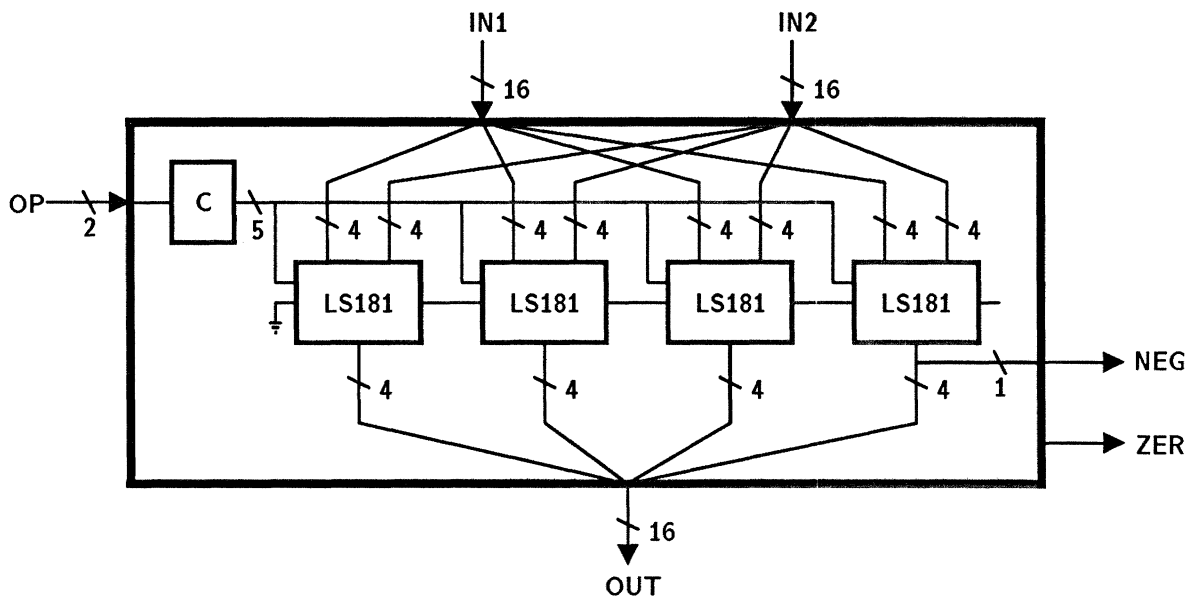


Figure 5.2: *An ALU model built out of LS181 components*

5.2.2.2 Granularity is Fundamental

This issue of functional granularity is fundamental. As components are chosen from a library of pre-sized functional units, so component tests are chosen from a library of pre-sized tests. Granularity of components and tests improves efficiency by limiting the number of cases that need to be considered during design and test generation, but it causes difficulties when there is a mismatch between the granularity of a component's functionality and the granularity of its tests. We consider this problem in detail when it arises in the section on embedding pre-written component tests (page 140).

5.2.3 Efficiency: Searching Designed Behavior can be Faster

5.2.3.1 The Designed Behavior Space is Smaller

An important efficiency advantage of DB-TG is that the space it searches – the designed behavior space – is smaller than the space classical test generators search – the potential behavior space. If the MAC-1 processor is represented at the gate level, the ratio between the spaces is roughly 2^{200}, corresponding to the ratio of legal in-

put sequences to possible input sequences. There is more to this story than ratios between the sizes of search spaces, because ratios say nothing about how the spaces are searched or about the frequency and distribution of solutions within the spaces. In particular, both DB-TG and classical test generators search somewhat outside of their respective search spaces as they construct tests and backtrack inside when inconsistencies in the tests are found. But based simply on size of the search space, DB-TG has substantial advantage.

5.2.3.2 Generating Tests for Structural Defects Can Waste Effort

Classical test generators can waste effort by attempting to generate tests for structural defects in unused portions of the circuit. For example, a simple implementation of the D-algorithm would attempt to generate tests for the carry output of the high-order slice in figure 5.2. After working out how to sensitize a stuck-at fault on this node, it would discover that the effects of the stuck-at cannot be observed, because the carry output is not connected to anything and is not a circuit output.

This is an horizon effect caused by the test generator failing to look ahead. It can be eliminated, in this case, by reordering the steps of the D-algorithm to do path sensitization first, but for any ordering there exist circuits where the worst-case behavior occurs.

A better solution is to examine the circuit structure before attempting to generate the test to see if a test is necessary. One commonly used method is to ignore stuck-ats on nodes that are not connected to an observable output by tracing forward through the schematic.

Unfortunately, structural analysis alone cannot always determine when component features are used by the rest of the circuitry. Circuit behavior must be taken into account. For instance, recognizing that 28 of the LS181 functional modes are unused cannot be done purely by structural analysis. This requires considering the behavior of the circuit. A test generator could recognize the unused modes by attempting to assign the control input of LS181 to one of them, propagating the control signals backwards through C (figure 5.2), and failing. In this case the backward propagation would fail quickly. In general, however, the test generator might need to propagate a long way before failing. In cases where propagation goes a long way, having an explicit and complete representation of the circuit's designed behavior, i.e., the behavior graphs, allows questions like this to be answered much more quickly. A test generator can search the behavior graph for examples of the control signals. This second search is bounded by the number of circuit operations times the length in time of each behavior

graph, rather than by an exponential of the propagation distance.

5.2.3.3 Searching Designed Behavior may not Find Efficient Tests

Searching for tests within a circuit's normal behavior may be insufficient to find *efficient* tests, i.e., tests that can be applied to the circuit quickly. While a test must exist within the designed behavior for every fault, that test may not be as efficient as one that lies outside the designed behavior. For instance, it is straightforward to construct an example where simpler tests exist outside the designed behavior than inside. Take a circuit with testability features and define its designed behavior so they are outside. Now tests for the faults in this circuit must exist within the designed behavior, but they are unlikely to be as simple as tests that use the testability features.

Whether simple tests exist within the designed behavior of real circuits often enough for the heuristic to be useful is an open question. Answering this question empirically is an obvious extension to the work in this thesis and is part of our future work. However, in lieu of a detailed study of hundreds of real circuits, we base our use of the heuristic on its use by our testing experts [bennetts82, robinson83].

5.2.4 Soundness

We have noted above that DB-TG is potentially unsound. However, this unsoundness is independent of the designed behavior heuristic, which determines the search space for tests but not how tests are constructed.

5.2.5 Summary: The Designed Behavior Heuristic

In this section we have seen that the space of a circuit's designed behavior is a subset of its potential behavior, and the difference in size can be orders of magnitude. Searching a circuit's designed behavior rather than its potential behavior makes test generation more efficient simply by reducing the size of the search space. Searching the designed behavior does not affect a test generator's completeness, because the designed behavior contains tests for all faults. However, it is not clear whether those tests are efficient. Based on the experience of expert test programmers, we believe that efficient tests can be found within a circuit's designed behavior often enough that searching designed behavior is a useful heuristic. This belief is born out by the limited experimentation we have done, but should be examined more closely by experimenting with a larger set of circuits.

5.3 Embedding Component Tests

This section shows how the strategy of embedding pre-written component tests into the circuit affects the performance of the test generator. The strategy has several strong advantages: (i) it allows use of efficient, expert-supplied tests, (ii) it amortizes component test generation costs, and (iii) it allows reasoning about faults in the aggregate. However, with these advantages come two strong disadvantages: (i) potential incompleteness, and (ii) potential unsoundness. Fortunately, the worst-case situations do not usually occur, and the test generator can warn the user when they might. In this section, I argue that the advantages of embedding pre-written component tests outweigh the disadvantages. As before, we consider the strategy's effect on completeness, efficiency and soundness in turn.

5.3.1 Completeness

The strategy of embedding component tests selected from a library is the primary source of incompleteness in DB-TG. The test generator fails to embed a component test when it cannot find a component operation causally connected to a circuit operation with parameter relationships that the test generator can solve. To understand why these failures occur, it is useful to think of component tests and simulated component operations as *sets* of behavior and the process of embedding tests as finding simulated component operations that are supersets of component tests.

A **behavior** of a component or a circuit is a set of lines from its truth table. Given two behaviors B and T, we say behavior B **subsumes** behavior T if $B \supset T$, i.e., if in the process of executing all of B the circuit also does T. This yields a strategy for embedding tests: if (i) T is a test; (ii) the test generator knows how to make the circuit do B, e.g., B corresponds to a simulated component operation in a behavior graph; and (iii) B subsumes T, then the test generator can perform the test T by executing behavior B (and ensuring the outputs are observable). For example, suppose T is the ADDER component test and B is the set of addition instructions with all possible data. B subsumes T, so T can be performed by executing B.

The behaviors the test generator knows how to execute, e.g., the set of all possible addition instructions, are often extremely large and involve many truth table lines. This has two consequences: (i) manipulating behaviors as explicit truth tables is far too unwieldy, so we represent them more compactly as algebraic expressions; and (ii) once the test generator has found a behavior that subsumes a test, it pares down the behavior to barely cover the test. The mechanisms inside the test generator that do

this have already been described, e.g., solving a set of operation relations. We review them briefly from this new perspective as they arise in the discussion of completeness.

Viewing embedding component tests as finding known-achievable behaviors that are supersets of the tests gives a framework for understanding how things can fail. Next we consider the three specific failure modes.

Incompleteness in the test generator stems from several sources:

- Component tests are selected from a limited set.
- Component functionality is sometimes unused and inaccessible from the outside.
- The representation of achievable circuit behavior, i.e., the behavior graphs, is partitioned into coherent sub-behaviors, and the test generator does not match component tests across sub-behaviors.
- The test generator saves work by solving the operation relations first for placeholder values and then later substituting test data in.

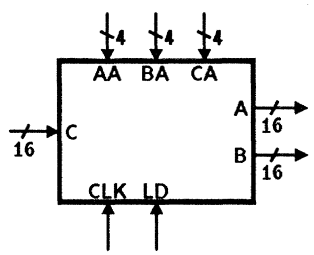
We consider each cause in order.

5.3.1.1 The Component Test Library is Incomplete

DB-TG embeds component tests by selecting one from the library and searching the circuit's designed behavior space for a way to carry out the test. While the designed behavior space contains at least one test for every fault, it may not contain the test selected from the library. Moreover, the library can contain multiple ways to test each component, but it is not feasible for it to contain all possible ways. Thus the component test library is incomplete in the sense that it cannot contain all possible test versions. This incompleteness in the library causes incompleteness in the test generator.

For instance, the BA address input of the MAC-1 Register File is not used fully by the microcode: it only uses addresses 0-10 (decimal) rather than the full range of 0-15. (See figures 5.3.a and 5.3.b.) In order to test the address input, DB-TG tries to embed the NODE component test shown in figure 5.3.c. Unfortunately this test cannot possibly be embedded; there is no way to set the node's value to 15 (1111 binary) as required by one line of the test data, since that value is outside the range 0-10.

A second test (shown in figure 5.3.d) can be embedded because all of the test data falls within the range 0-10 (decimal). However, in another circuit yet another



(a) The MAC-1 Register File

0000
0001
0010
0011
0100
0101 ←
0110
0111
1000
1001
1010 ←

(b) Achievable values for BA (used by the microcode)

Test Operation	value(Node, ?data)
Test Data	?data
	0000
	1111

(c) Test 1 (not achievable)

value(Node, ?data)
?data
0101
1010

(d) Test 2 (achievable)

Figure 5.3: *Subfigures (c) and (d) show two versions of a NODE test. (c) cannot be embedded for the B address input of the register file (BA) because its test data does not match the values achievable on that node. (d) does and can be embedded. This is a situation where having several versions of a component test in the library helped. However, it is impractical to store all possible versions.*

variation on the node test might be necessary. The component test library cannot contain all such variations – there are simply too many – so the expert anticipates several of the most likely ways components are used and includes tests for them.

As a result, the task of writing component tests cannot be completely separated from embedding concerns. We conjecture that digital circuits are stylized enough in their design that a small number of variations for each component test will cover most cases. This conjecture is very likely to be true of datapaths (as experiments with the MAC-1 indicate), however, whether it is also true of state machines and other complex circuitry needs to be explored by additional experimentation.

5.3.1.2 Unused Component Functionality Can Be Inaccessible

Section 5.2 described how component functionality can be unused. Any unused functionality can be inaccessible from the outside, which makes it impossible to embed a component test designed to exercise that functionality. While this unused functionality need not actually be tested, the library may not contain just the right piece of a component test that uses only accessible functionality.

For instance, the DB-TG fails to embed a test for the ALU's AND operation. The design of the circuitry around the ALU and the microcode in particular render the high-order bits of the ALU's AND functionality inaccessible. The inaccessible functionality is not an explicit design decision, but rather an inadvertent byproduct of explicit design decisions. In one decision, the designer chose to have the ALU/AND mask instruction fields as the MAC-1 decodes instructions, and these masks happen to be incompatible with the test data supplied by the test expert, i.e., the 8 bit and 12 bit masks do not match the test data. In another decision, the designer chose not to give this processor a general-purpose AND instruction. These decisions together make the AND operation difficult to test.

Figure 5.4 shows in detail why the library test for the AND operation fails. To save space and time, behaviors are represented in the test generator as compact symbolic expressions rather than as truth tables. Test operations in the library and the simulated operations in the behavior graphs are both of this form. These representations contain typed variables that range over the values of circuit nodes. For example, the simulated operation

```
AND( ALU1, (:fields (field 12 15 6) (field 0 11 ?addr)), 4095)
```

represents a subset of the ALU's truth table. The ":fields" subexpression describes a value that is comprised of several bit fields. Each bit field is described by a "field"

function that takes a low bit position, a high bit position, and a value as arguments. In this example, the variable ?addr ranges over the possible values on the AddressBus (0-4095). The values are shown in decimal. This expression represents the portion of component ALU1's truth table shown in figure 5.5.

5.3.1.3 The Fragmentation Problem

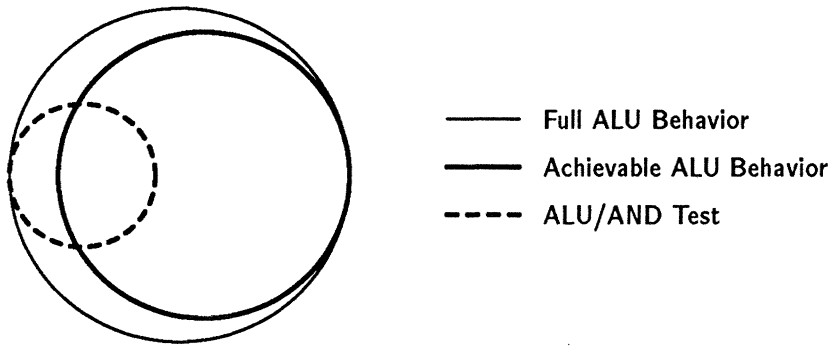
Even when all of a component's behavior is achievable, it may not be represented in the test generator in a form that allows component tests to be embedded. The advantage of representing behaviors with symbolic expressions is that the expressions are compact. They are compact because they exploit regularity of behavior, i.e., similar behaviors are represented by similar expressions. However, when achievable behavior is irregular – like the AND operation in the MAC-1 – this behavioral representation becomes fragmented, i.e, behavior must be represented with many symbolic expressions, as pieces of the irregular behavior are represented with separate expressions. The problem of finding achievable behaviors that are supersets of component tests is exacerbated when the achievable behaviors are fragmented, because fragmentation makes the individual sets of achievable behavior smaller, thus making the test generator less likely to find a single achievable behavior that subsumes a given component test. Note that failure due to fragmentation is a property of the *languages* used to describe component tests and achievable circuit behavior as well as of the circuit itself.

For example, the behavior graphs for the MAC-1 contain 6 different examples of ALU/AND, including the one above, that describe non-overlapping portions of the ALU's truth table (and many more that are subsets of one of the 6). These different examples correspond to different combinations of masks and data.

Contrasting the ALU and the microprogram counter (uPC) highlights this issue. Using the addition instruction, for instance, the ALU can be made to add different pairs of numbers *in the same way*, i.e., the activity inside the circuit differs only in the numbers. However, the μPC is not like this. There are 80 lines of microcode, and it is possible to load the μPC with any value in the interval 0-79. However, all of these values must be loaded in *different* ways, i.e., by executing different instructions, or branch instructions with different data. The test generator's description of the μPC's achievable behavior is spread out in small pieces (i.e., loads of constants rather than variables) all over the behavior graphs rather than as a general operations in a few places in the behavior graphs.

Unused component functionality and fragmentation of DB-TG's behavioral representation are the primary reasons that DB-TG sometimes cannot embed tests. These

(a) Sets of ALU Behavior



(b) Test Operation AND(ADDER, ?Data1, ?Data2)

(c) Test Data

Data1	Data2
0000000000000000	0000000000000000
1111111111111111	1111111111111111
1010101010101010	0101010101010101
0101010101010101	1010101010101010

Mask does not match any test data

(d) Simulated (Achievable) Behavior

Position	54321098	76543210
Data	11111110	?offset
Mask	00000000	11111111

Position	5432	109876543210
Data	0110	?addr
Mask	0000	111111111111

Figure 5.4: *Why the pre-written ALU/AND tests cannot be embedded: the ALU/AND operation is used only to extract bit fields by masking. DB-TG fails to embed the component test because it lies partially outside the ALU behavior achievable within the constraints of the larger circuit.*

OP	IN1 *(data)*	IN2 *(mask)*	OUT
AND	0110000000000000	0000111111111111	0000000000000000
AND	0110000000000001	0000111111111111	0000000000000001
AND	0110000000000010	0000111111111111	0000000000000010
AND	0110000000000011	0000111111111111	0000000000000011
⋮	⋮	⋮	⋮
AND	0110111111111111	0000111111111111	0000111111111111

Figure 5.5: *This portion of the ALU's truth table corresponds to the :fields expression on page 139. The values are in binary to make their structure apparent.*

problems are important; chapter 6 covers them in detail and describes several ways of addressing them.

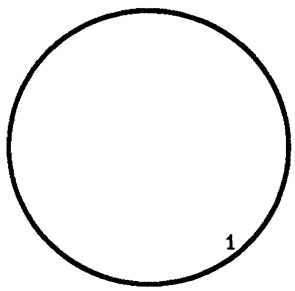
5.3.1.4 Inserting Placeholders Has Pros and Cons

DB-TG's method of determining whether a simulated operation subsumes a component test is a fast, approximate solution: it extracts the operation relations for the simulated operation, substitutes placeholders for the test data into them and solves for the parameters of the circuit operation. If the operation relations can be solved, then the simulated operation must subsume the test operation with the placeholders (which are arbitrary constants), therefore it must subsume the test operation with the test data. Once the operation relations have been solved, the placeholders are replaced with test data to create the actual test.

This method has two strong advantages:

- The cost of solving the operation relations is incurred once for the placeholders rather than repeatedly for each line of test data.
- Solving for circuit outputs that correspond to the component test data pares the test down to exactly what needs to be executed at the circuit inputs to cause the component test to occur inside. The test generator does not waste time executing long, known-achievable patterns of activity inside the circuit that happen to subsume a short component test.

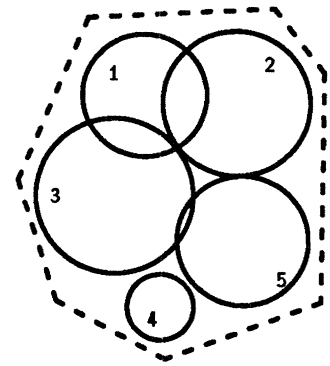
However, the method also can cause the test generator to fail to embed a component test that is actually achievable. Figure 5.8 illustrates this problem with the B



Description

Circle 1

(a) A regular component behavior (the description is compact)



Description

Circle 1
Circle 2
Circle 3
Circle 4
Circle 5

(b) An irregular component behavior (the rectangle) approximated by multiple, regular behavioral descriptions (the description is lengthy)

Figure 5.6: *The fragmentation problem: this figure illustrates the underlying cause of the fragmentation problem with an analogy. Here, the behavioral representation language can only describe circles compactly. When component behavior is regular, as indicated by the circle in (a), then that behavior can be described succinctly. When component behavior is irregular, as indicated by the rectangle in (b), then component behavior must be described using multiple, disjoint pieces. Fragmentation is a property of the* language *used to describe circuit behavior as well as of the circuit behavior itself.*

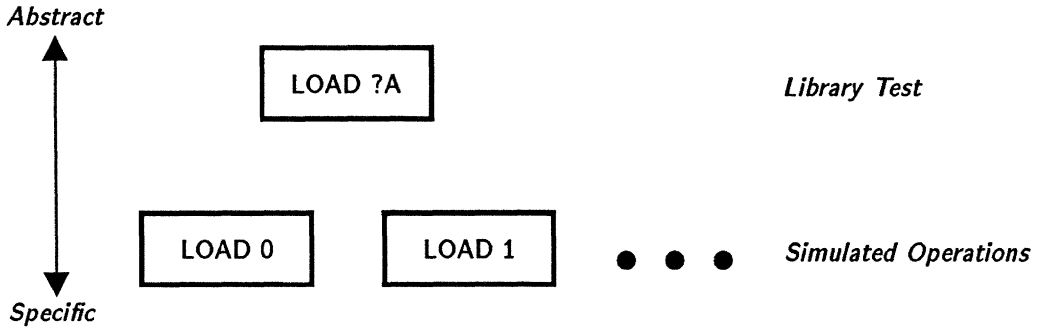


Figure 5.7: *An Example of Fragmentation: the REGISTER library test cannot be embedded for the uPC register because it involves a fairly large set of behavior (the test data is not shown) but no simulated instance of the register loading covers as large a set. Fragmentation of the representation for uPC's achievable behavior causes the test generator to fail here.*

address input of the MAC-1's register file. While this problem does occur in practice, it turns out to cause few failures in the MAC-1. The limited size of the component test library, inaccessible component behavior and fragmentation of the behavior graphs are the primary reasons that DB-TG fails to embed tests.

5.3.1.5 Granularity and Fragmentation Stem From the Design Process

Why is there a radical difference in coverage between the datapath and the sequencer? The difference is caused by regularity of behavior in the datapath and fragmentation of behavior in the sequencer. For instance, in the datapath there are several simulated ALU/ADD operations with variables for data, but in the sequencer the μPC is never loaded with a variable in any of the simulation runs. All examples of μPC behavior are more specific than the corresponding library test. The same story is repeated throughout the MAC-1. We conjecture that this unevenness in the amount of fragmentation over the circuit is a direct result of the design process.

Consider the following account of processor design: a designer starts with a specification for an abstract machine that includes the programmer accessible registers and the instruction set. His job is to implement this abstract machine in hardware while meeting myriad performance, reliability, and cost constraints.

The specification can be viewed as a set of dataflow graphs, one for each instruc-

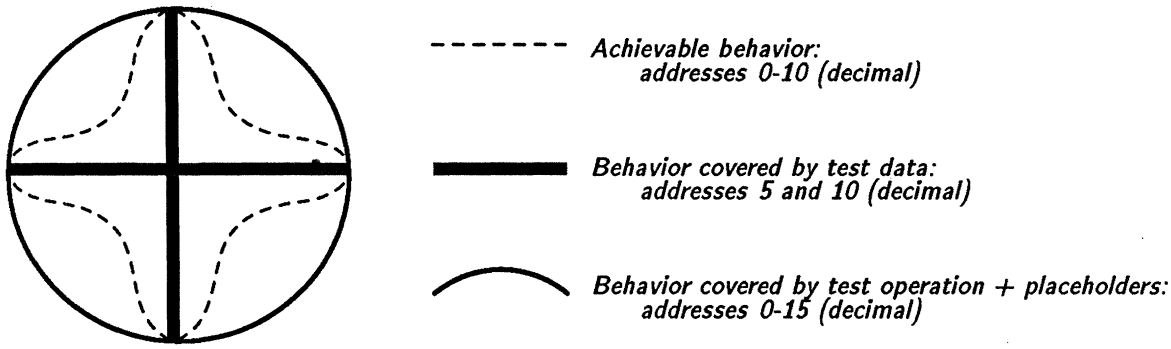


Figure 5.8: *The circular, solid, and dashed regions represent subsets of the behavior of the B address input of the MAC-1's register file. The dashed region represents behavior achievable within the MAC-1, i.e., addresses 0-10 decimal. The solid region represents component behaviors that the test would actually use, corresponding to test data from version 2 of the node test shown earlier in figure 5.3. Given a test with placeholders inserted, the circular region represents behaviors the test might possibly use if the placeholders are replaced with arbitrary values. The test is achievable since it lies entirely within the space of achievable behavior, i.e.,* $\{5, 10\} \subseteq \{0 \ldots 10\}$. *However using placeholders causes the test generator to fail because some values that could potentially replace the placeholders, e.g.,* $\{11 \ldots 15\}$, *are not achievable.*

tion, describing how data is transformed as it moves from register to register. The designer usually cannot implement these dataflow graphs directly in hardware: without some sharing there would be a wasteful duplication of functionality. So he looks for ways of merging the graphs together.

To do this, he adds to the graphs components for performing identity transformations. For example, he might insert a register in one graph to shift some of its operations later in time, thereby allowing components to be shared with another graph via time-multiplexing. In another situation, he might introduce identity boxes into two graphs and implement them using a single multiplexor. By adding these components to the flow graphs, he is able to fit them all together. When this process is complete, the designer collects all the control signals from all the graphs and creates a finite state machine (FSM) to provide these signals at the right times. The FSM is implemented using any one of the well-known methods (e.g., with a microcode engine).

By this account, the way the datapath is designed (incremental refinement and merging) is very different from the way the controller is designed (stylized implementation of a state machine). The availability of components which directly implement large portions of individual processor operations (e.g., ALU chips), plus the fact that the merging process does not normally change existing components (it just adds new identity boxes), means that many datapath component operations tend to "very directly implement" circuit operations. The process of designing a state machine, however, need yield no such simple part-whole relationships. The behavior of the whole controller (a state machine) is very different from the behavior of any single controller component (e.g., a register, ROM, or MUX).

5.3.1.6 Conclusions

DB-TG is incomplete in its ability to find tests. This incompleteness stems from (i) selecting component tests from a limited set supplied by an expert and (ii) interactions between the granularity of the component tests and granularity and fragmentation of the representation of achievable circuit behavior (i.e., the behavior graphs). However, when embedding pre-written component tests succeeds, it has several very important efficiency advantages described in the next section. When embedding pre-written component tests fails, DB-TG can turn to several methods of generating component tests upon demand, achieving the effect of having a larger library or one designed specifically for a particular circuit. These extensions are described in chapter 6.

5.3.2 Efficiency

Embedding pre-written component tests has several important efficiency advantages. First and foremost, the expert-supplied tests are themselves efficient, being the product of the intelligence and experience of a human expert. The carry-chain adder test on page 98, for instance, uses a minimum of test data – no gate level test generation algorithm could do better, and many do worse. Moreover, this and similar tests are designed to cover potential faults not considered by classical test generators.[3] In addition to tests designed by the expert, the library can also include the best available tests published in the literature and generated by any means whatsoever.

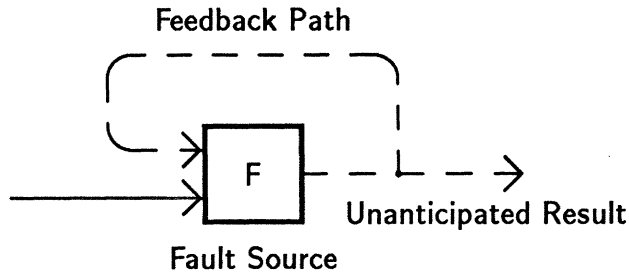
Embedding pre-written component tests amortizes component test generation costs. Because a component test will be used in many circuits, it is feasible to invest more effort designing it than in designing a test that will be used only once. This extra effort can go toward shortening the test or increasing its coverage.

Finally, embedding pre-written component tests saves time by allowing the test generator to work with faults in the aggregate. Classical test generators in effect ask "suppose this node is stuck?", generate a test for that fault and move on to the next fault. DB-TG in effect asks "suppose this component operation is faulty?" and proceeds to embed a test for the operation. For example, when embedding the ALU/ADD test, the test generator need not explicitly consider all of the possible ways in which the ADD operation could go wrong. DB-TG simply builds on top of the work of the expert (who did consider all of the failure modes). Since there are far fewer component operations than circuit nodes, this approach saves effort. The cost of considering faults individually is incurred only during the design of the component test. Not considering them individually when embedding tests saves work each time the component test is used.

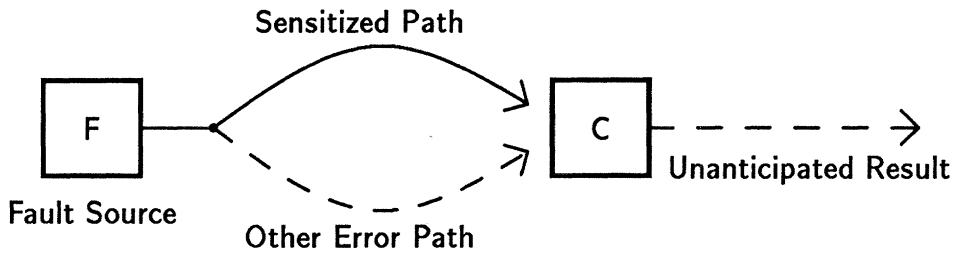
5.3.3 Soundness

Unfortunately, the efficiency advantage gained by working with faults in the aggregate is at odds with the goal of generating sound tests. The key issue is when can a component be used to help test itself? The need to use a component in this way arises in circuits that exhibit a structural configuration known as reconvergent fanout. Figure 5.9 shows the two prototypical cases. In the first case, there is a feedback path from the output of F, the component under test, back to one of its inputs. In the second case, two or more paths from F reconverge on another component.

[3]For example, bridge faults within each single-bit adder in the carry chain



Case 1: A Reconvergent Feedback Path



Case 2: Interference with a Sensitized Path

Figure 5.9: *This figure shows two examples of reconvergent fanout. In these configurations, the component F must be used to help test itself. In case 1, F is used to set up its own inputs. In case 2, F is used to help observe its own output value. Generating tests in either situation requires very precise predictions of fault effects.*

In both cases component F must be used to help test itself, and herein lies the problem. How can a test be sound if one of the components used to carry it out is potentially faulty? In case 1, F must be used to set up its own inputs. For example, the ALU in the MAC-1 is used to decode instructions, hence any test for the ALU that involves executing instructions must itself rely on the ALU. How can we be sure the ALU has decoded the instruction correctly so that the rest of the test proceeds as planned? In case 2, a potentially wrong signal from F could interfere with the sensitive path from F to an output. A test generator needs more precise information to predict how two error values will interact, than it needs to propagate a single error value.

The need for precise information about fault effects in the event of reconvergence is at the heart of this problem. When a test generator hypothesizes a specific fault, e.g., a node stuck at 0, it has precise information about the effects of that fault, i.e., if the fault is present then the node's value will be 0. When a test generator hypothesizes an abstract fault, e.g., a component is broken, it has only vague information about the effects of that fault, e.g., if the fault is present then the component output will not be correct. Since the fault hypothesis does not specify *how* the component is faulty this can leave a lot of possible error values. For a test that uses this component to be sound, the test must be designed to work properly for every one of those wrong outputs.

DB-TG detects when it has used a component to help test itself by examining the dependency records on all operation relations it used. In this situation, the test may be unsound, i.e., it may not work as planned, and must be verified by fault simulation. We believe that most of the time the test will work as planned, and express this belief as the reconvergence heuristic, described next.

5.3.3.1 The Reconvergence Heuristic

Our approach to this problem is heuristic and follows the practice of human test programmers and functional test methodologies: the test generator assumes that the component under test is working properly whenever it is used to help test itself, i.e., to set up its own inputs or to observe its own outputs. This strategy is made explicit as the reconvergence heuristic.

Proposition 2 *For complex, sequential devices, ignoring reconvergent fanout is extremely unlikely to cause a test program to miss faults.*

This proposition is probabilistic, and as a result the method can occasionally produce unsound tests. Theoretical analysis of this proposition is difficult due to the unconstrained nature of the class of circuits it is intended to cover. Experimentation with real circuits is thus the strongest candidate for a method to corroborate or discount this proposition. While we have not yet collected a large amount of empirical evidence, we believe the proposition is likely to be true for several reasons:

- If a component combines the values of multiple input bit positions to create its output, e.g., a multiplier, then it is unlikely to mask a single bit error that it has previously caused. This tends to become more unlikely as the component's word size increases. (This likelihood does not change for bit-parallel components.)
- The likelihood that a fault will mask its own effects at the planned test output *and also not appear at any other circuit output* tends to decrease with increasing reconvergence and circuit complexity.
- In circuits where many components are involved in many operations, the likelihood that a fault will remain undetected by every component test tends to decrease with increasing test program length, and hence with increasing circuit complexity.

5.3.3.2 An Experiment

The reconvergence heuristic appears to be borne out by the MAC-1 example. Fault simulation indicates that the tests generated by DB-TG actually do detect all of the faults they were designed to detect. I determined this by fault simulating the component tests twice, once with the components outside the circuit and once with them inside. In both the single component and full circuit simulations the fault simulator listed when and where each fault was detected at an output. These lists were then compared to see whether every time a fault was detected with the component outside the circuit, there was a corresponding detection with the component inside the circuit.

The details of the experiment are somewhat involved because they are designed to minimize noise in the results arising from lucky, unplanned routes from faults to circuit outputs. The comparison was done at the times and circuit outputs where the test generator expected errors to appear. This involved working out the temporal mapping between the two simulation runs. The fault simulator also listed the values of selected internal circuit nodes, e.g., the operation inputs of the ALU, so we could

determine whether the test had progressed as planned. The faults in this circuit fell into three categories:

1. Internal circuit activity, faulty and not, occurred as planned. These faults were detected.

2. The fault caused the circuit to deviate far enough from the plan that the component test operation never occurred. These faults were detected early due to the wide deviation. The faults were also detected at the expected times because the circuit had wandered far from the correct behavior by then.

3. The fault caused the circuit to deviate so far from the plan that the fault simulator did not model the faulty behavior properly. The simulator detected this situation, warned the user and aborted the simulation. In all cases, these faults were detected early, before the simulation run was aborted.

A fourth case is possible: the test might not proceed as planned, with the fault masking itself and yet causing an error at the proper time and place via an unplanned route. We observed no such situations in this experiment and believe it to be an extremely rare occurrence.

5.3.3.3 Soundness can be regained by case splitting

A sound method of using a component to help test itself is to revert to a more specific (e.g., gate-level) fault model. This method achieves specific predictions of fault effects by splitting the fault hypothesis (e.g., a component is faulty) into many, very small cases (e.g., a node inside the component is stuck at 0) and solving each case separately. Each small case can be solved because it makes a specific prediction about the effect of the fault that can be relied upon to plan a test.

The techniques involved come from hierarchical test generators and are well understood [genesereth81, shirley83b, singh86, krishnamurthy87]. We have chosen not to implement this method in DB-TG because DB-TG is a demonstration system and these techniques are neither new nor necessary to demonstrate the main ideas.

Even if DB-TG were a production test generator, it is not clear whether modifying it to achieve soundness by case-splitting would be worth the cost. Splitting fault hypotheses into many cases is expensive, and, as the experiment above indicates, this technique would not have improved the coverage of the tests generated for the MAC-1.

5.3.4 Summary: Embedding Component Tests

The strategy of embedding pre-written component tests into a circuit has several important advantages: (i) it allows use of efficient, expert-supplied tests, (ii) it amortizes component test generation costs, and (iii) it allows reasoning about faults in the aggregate. These advantages are accompanied by two disadvantages: (i) potential incompleteness due to the limited size of the component test library and to functional granularity of the representations for tests and achievable behavior, and (ii) potential unsoundness due to the tension between achieving efficiency through abstraction and achieving soundness through specificity.

The tension between efficiency and soundness is a fundamental one. This tension arises when embedding component tests because component tests can handle many faults at once but in doing so prevent precise predictions about the effects of these faults. Precise predictions are needed for a test generator to soundly and exhaustively plan for all contingencies that might interfere with a test. This tension will arise again in connection with operation relations.

Fortunately, for realistic circuits the worst-case situations do not occur often (save for the functional granularity problem), hence we optimize for the advantages listed above. In situations where functional granularity causes incompleteness it is possible to generate library tests upon demand, achieving the effect of having a larger component test library. This extension is described in chapter 6. In the situations involving potential unsoundness, the test generator can warn the user, a fault simulator can determine whether the test is actually unsound, and the user can then revert to low-level circuit and fault representations. These extensions have not been implemented; the ideas are described in section 8.3 under future work.

5.4 Operation Relations

This section considers the utility of representing circuit behavior as relationships between circuit and component operations. Using operation relations provides an important advantage: they are a more compact representation of the circuitry surrounding a component – its behavioral context – than are structural circuit models. This simplifies the task of embedding component tests. Using operation relations has a related disadvantage: potential unsoundness. Operation relations are compact because they abstract away from the details of data movement in both space and time. Abstraction is inappropriate when detailed descriptions are necessary, as when a component is used to help test itself. As with the strategy of embedding component tests, test

generated with operation relations can occasionally be unsound. Again we argue that the advantages outweigh the disadvantages.

5.4.1 Efficiency

Operation relations contribute to the efficiency of the test generator in two ways. First, causal connections allow the test generator to quickly identify candidate solutions for the embedding problem. Each candidate solution is an instance of a component operation appropriate to the component test, so part of the work has already been done. A candidate solution is turned into a real solution by substituting test data into the parameter relations and solving them for values on the circuit inputs and outputs. The bulk of the advantage lies in this second step.

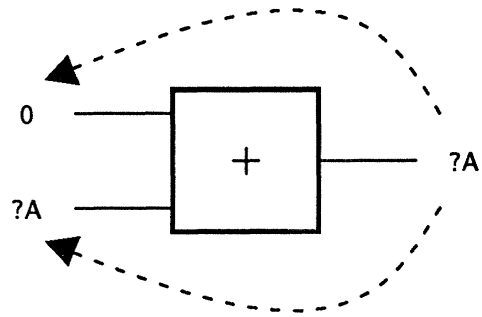
Solving parameter relations is analogous to line justification and path sensitization in a conventional test generator but with a twist: parameter relations are solved by propagating through the structure of pre-simplified algebraic expressions rather than through the structure of the original circuit. Figure 5.10 illustrates this process. Figures 5.10.a and 5.10.b show two rules for propagating values backwards through an addition operator in an operation relationship. These rules are completely equivalent to rules for propagating through components in conventional test generators except that the vocabulary for values has been expanded to include algebraic expressions.[4] Figure 5.10.c shows an example operation relation as produced by the simulator,[5] and figure 5.10.d shows the same expression as a network and the results of propagating a value (VAL) backward using the rule in figure 5.10.b. The test generator never actually constructs networks like this but instead works directly with the representation of the operation relation.

The simplicity of the algebraic expressions is the source of the technique's power. In the parameter relations identity data transfers are simplified away. Figure 5.11.a shows an example of this simplification. Signals flowing through this subcircuit are inverted as they go onto the backplane and re-inverted as they come off. When propagating through this subcircuit, a test generator must invert the signal twice. In total, propagating across this subcircuit takes 5 units of work, one for each node and component along the path.

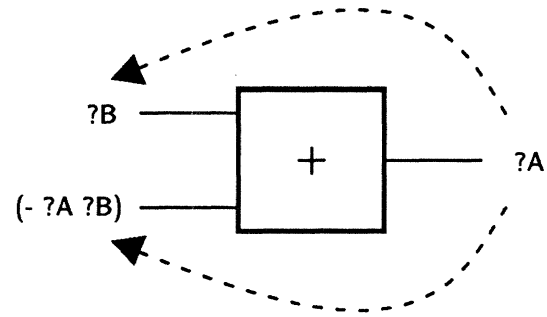
Figure 5.11.b shows the simulated behavior of this subcircuit. Figure 5.11.c shows the corresponding parameter relation, which is an equality that takes 1 unit of work to propagate through. Propagating through the parameter relation is more efficient be-

[4]DB-TG includes both rules shown in the figure and tries the simpler one first.

[5]DB-TG canonicalizes to two argument addition operators.



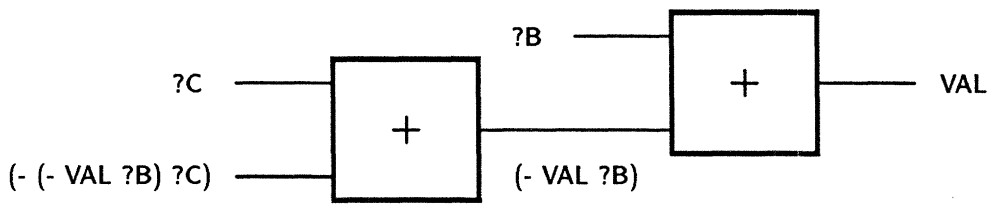
(a) A simple and fast backward propagation rule.



(b) The most general backward propagation rule.

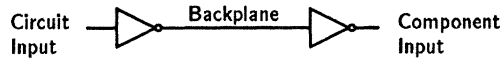
(+ ?A (+ ?B ?C)) = ComponentInput

(c) A parameter relation (?A ?B and ?C are circuit inputs). This relation can be visualized as the network of two adders below.

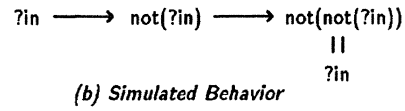


(d) This network, corresponding to (c), shows how VAL is propagated backwards using the general rule from (b).

Figure 5.10: *Propagating through operation relations is analogous to propagating through circuit structure.*

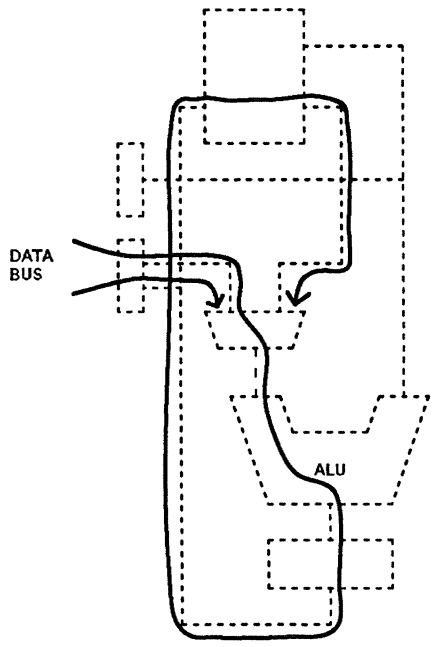


(a) Circuit Structure

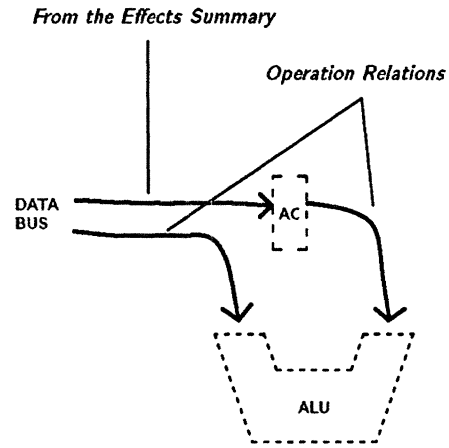


(b) Simulated Behavior

(c) Corresponding Operation Relation



(d) Structure of the MAC-1 Datapath. The bold lines show datapaths used to supply inputs to the ALU.



(e) The bold lines show the simplified circuit representation DB-ATG uses to decide how to supply ALU inputs.

Figure 5.11: *Parameter Relations are Simplified Paths for Propagation*

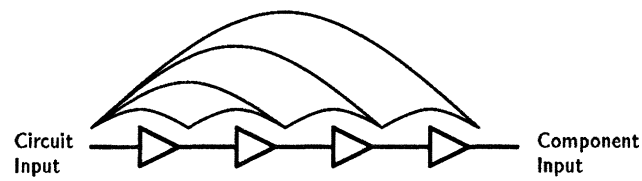
cause it explicitly represents the fact that the backplane moves data without changing it.

Parameter relations also abstract away from time. Circuits often store values temporarily in registers. If the actions of storing and retrieving a value are built into the circuit behavior such that the test generator cannot affect them, as the inverters are built into backplane structure, then the test generator need not reason about them. Memory is just another identity transformation stripped away by the simplifier during simulation. Figures 5.11.d and 5.11.e show an example of this. Both figures show the paths used to supply inputs to the ALU in bold. Values in (d) enter from the DataBus, pass through a register and on into the circuit. This temporary storage of the value in the register is built into the circuit behavior and cannot be affected. The operation relations in (e) abstract away from that detail. The operation relations do represent the temporary storage of one value in the Accumulator. The test generator must reason about this because it occurs at the boundary of a circuit instruction and can be affected.

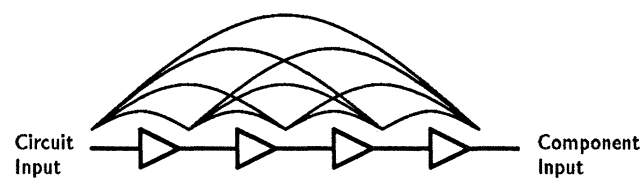
It is possible to construct circuits that will generate identity expressions that cannot be reduced by a given set of rewrite rules. As a consequence, the simplifier will not be effective in all cases. For instance, a DFFT circuit that transforms a time varying signal into the frequency domain and back again implements (ignoring sampling errors) a very complex wire from one place to another, but DB-TG will not recognize this. However, most circuits do not move data in such a convoluted way. For circuits that do, e.g., signal processing circuits, one can augment the simplifier with rules suitable for the circuit type.

So far we have described the cost of using operation relations. When they come from an outside source such as a human designer or a silicon compiler, nothing more need be said. However, when the test generator derives them for itself then the derivation cost must be considered. The cost of propagating a value through the circuit during simulation is equivalent to the cost of propagation during test generation. Simplifying the value at each step does not add to the cost, since maintaining the value in simplest form would also be done during test generation. However, by simulating and simplifying in one forward pass, DB-TG builds on simplified partial results at each step. For instance, the cost of computing operation relations for the components in a straight path is linear in the number of components (see figure 5.12.a). The cost of propagation during test generation rises as the square of the number of components (see figure 5.12.b), unless the test generator caches partial results during propagation, with the attendant costs and bookkeeping complexity that implies. Getting this caching right is complex; the bookkeeping required for one simulation pass is much

simpler.



(a) Computing the relationships in one forward pass during simulation involves O(n) work (n is the number of components here).



(b) Computing the relationships piecemeal during test generation involves O(n^2) work. O(n) can be achieved with complex (and costly) caching and bookkeeping.

Figure 5.12: *Computing relationships between circuit inputs and component inputs in one forward pass saves work over doing it piecemeal during test generation.*

5.4.2 Soundness

In this section we consider how using operation relations can go wrong. The efficiency advantage gained by working with operation relations is unfortunately at odds with the goal of generating sound tests. As with embedding component tests, the key issue is when a component can be used to help test itself.

The process of computing operation relations uncovers the simplicity and order built into a circuit by its designer. However faulty circuits are much more complex to describe and to reason about, and the simple descriptions are no longer correct. For instance, a component which normally adds integers will compute some complex boolean function of its inputs when a fault is inserted. In the presence of a fault, the abstraction shift from boolean operations on bits to arithmetic operations on integers is not valid. All of the special, timesaving rules we have for reasoning about addition

operators may no longer be applicable.[6]

In DB-TG, this problem manifests as a failure of the propagation rules used during simulation. For instance, when an adder is faulty, the simulation rule that predicts that the output will be the sum of the inputs is inaccurate. This inaccuracy is further compounded by simplification rules that combine the output of the faulty adder with other values. Since the test generator records what rules were used to derive each expression during simulation, it is a simple matter to know when there is a potential for inaccuracy. The question is, what are we to do about it?

A sound solution is to revert to the un-simplified versions of all suspect expressions and to revert to specific circuit and fault models for the potentially faulty component as was discussed in the section on embedding component tests on page 151. However, it is often the case that much of the behavior of the circuit depends on much of the circuit, hence the simplified, abstract representations of circuit behavior are often inaccurate. This seems to be especially true of complex sequential circuits with global feedback like the MAC-1. But test experts generate effective tests without investing much effort reasoning about these inaccuracies.

Soundness requires considering all possible ways a fault might interact with the plan for a test. Considering all possible interactions is expensive, and the question of whether doing so is worth the cost is fundamentally an economic one based on the cost and the quality of tests desired. Here, we again follow the reconvergence heuristic and the practice of human test experts. DB-TG generates tests using the operation relations and tells the user which tests use expressions that depend upon the component under test and hence may be unsound.

5.4.3 Completeness

The use of operation relations does not reduce the test generator's completeness as long as all circuit operations are included. If DB-TG is given an incomplete list of circuit operations (e.g., there are too many and simulation costs would be too high), the program can still generate tests, although with degraded performance. In the MAC-1, the program achieves almost the same coverage on the datapath using just four instructions (LOAD, STORE, ADD and SUBTRACT) as it does with the full instruction set. This is primarily due to the granularity problem: this ratio changes

[6]Certain classes of faults do result in simple misbehaviors that can be described abstractly. For example, stuck-at faults on the inputs or output of an adder can be described as perturbing the correct value by a power of 2. However, we know of no abstract descriptions for misbehavior in the general case that remain detailed enough to guarantee soundness.

with the extensions discussed chapter 6.

5.4.4 Summary

The strength of operations relations is that they are *simplified* representations of circuit behavior that allow more efficient and direct solution of testing problems. Their weakness is that they are *simplified* representations. It is exactly the assumptions that make the operation relations simple and effective, namely that the circuit is behaving as it was designed to behave, that get in the way when considering what might happen when the circuit is faulty.

However, the utility of operation relations for test generation need not rest on formal soundness. If tests generated using them are usually valid as verified by fault simulation, then using operation relations is a good heuristic. Experimental evidence from the MAC-1 suggests that this is the case: using operation relations produced tests that detect the faults they are supposed to detect.

5.5 Simulate and Match

This section shows how Simulate and Match aids the performance of the test generator. Simulate and Match is the idea that operation relations can be obtained by simulating circuit behavior, and the relations applicable to a particular situation can be found by searching simulation traces. The previous section considered this issue from the perspective of using operation relations as a compact representation of the circuitry surrounding a component. This section views operation relations as an explicit representation of a circuit's designed behavior and simulation as a method of computing them that focuses the test generator very closely on designed behavior rather than potential behavior. The argument is based the technique's ability to avoid proposing inconsistent partial plans as it searches for a test.

Tests are planned by repeatedly refining the goal of causing a specific internal behavior until the problem can be solved by direct action on the circuit inputs. In this process, the test generator chooses a way to refine the goal and propagates the consequences of that choice in order to focus search by constraining later choices. However, existing constraint propagation techniques are imperfect: they sometimes cannot immediately detect when a new choice is inconsistent with previous choices or when a new choice is a dead end that will preclude finding a solution later. Until the problem is discovered and the test generator backtracks, further choices depending on the erroneous one represents overhead effort that does not lead directly to a solution.

When the test generator makes an inconsistent choice, it has wandered outside of what the circuit can possibly do.

DB-TG expends little effort considering globally inconsistent solutions or solutions outside a circuit's designed behavior. The behavior graphs represent the globally consistent, known-achievable behavior of a circuit. Matching component tests against the behavior graphs is a search process, and one can view a failed match as wasted search outside the designed behavior. However, failed matches, i.e., proposed tests that are globally inconsistent, tend to be identified quickly for two reasons:

1. *Simple Expressions*: The mechanism for detecting a failed match (i.e., an inconsistency) is to extract and solve the operation relations. This tends to terminate quickly because the expressions are often simple.

2. *Query ordering*: Component tests that use conjunctive patterns can be expensive to match against the behavior graphs. DB-TG saves work by using query ordering techniques, e.g., by organizing the match so that inexpensive choices are made before expensive ones. Having a complete set of behavior graphs facilitates this.

   For instance, one measure of how cheap a choice will be is how many alternatives there are. The pattern for the ALU addition test, for example, involves matching four items: two data inputs, the operation input and the output. The program estimates the number of choices for each part of the match, say for the operation input, by counting each value in the behavior graphs that could match that part. Variables and complex algebraic expressions count more than constants, since they are more costly to match against. The program does this for each part of the match (caching its results for use by subsequent matches). In the case of the ALU, the operation input has the smallest space of choices, hence the program matches the operation input before the other inputs and the output. The information used to do this query ordering is readily available in the behavior graphs but is only implicit in the structure and component behavior of the circuit.

Simulate and match is a cost effective technique when the cost of the wasted search performed by conventional test generation approaches is enough to offset the initial cost of doing the simulations. Simulate and match is thus appropriate for circuits which execute a few complex operations.

5.6 An Estimate of Computational Complexity

The cost of generating tests with DB-TG is

$$O(n \times I \times L \times E^3 + 2^E)$$

where:

n = # of nodes (or components)
I = # of circuit operations
L = Average duration of a circuit operation in simulated time (e.g., clock cycles)
E = Average token size of simulated expressions passing through the circuit under the assumption that E is independent of n, I and L.

The cost of running DB-TG is the sum of the costs of three steps: (i) simulation, (ii) finding simulation operations that are candidates for matching, and (iii) extracting and solving the operation relations. Assuming that each node is active at each simulated time step, the cost of simulation is $O(n \times I \times L \times E^3)$. The cost of finding candidate simulated operations is $O(n \times I \times L)$, corresponding to the worst case number of operations in the behavior graphs. The cost of extracting and solving the parameter relations is $O(2^E)$.

Considering just factors that change with circuit size, the cost of running DB-TG is $O(n \times I \times L)$. Important here is the assumption that the average size of expressions is independent of circuit size and the duration of operations. This assumption depends upon the simplification rules having the right vocabulary, i.e., that they simplify. This assumption is supported by measurements on the MAC-1. The average size of the expressions propagated while simulating the MAC-1 instruction set is 6.4 tokens, where a token is a constant, a variable, an operator or a parenthesis in the printed representation of the expression. Figure 5.13 shows a histogram of the average sizes by time (over the whole instruction set).

The expressions are large early in the simulation runs due to an initialization phenomenon. When a simulation run starts, many state registers are preset with variables representing their values at circuit power up or from a previous instruction. The outputs of combinational circuitry downstream of these registers have values that represent the potential behavior of the combinational circuitry. These complex values generally do not participate in the rest of the simulation, but are replaced by more specific values as the circuit fetches an instruction and determines what it is going to

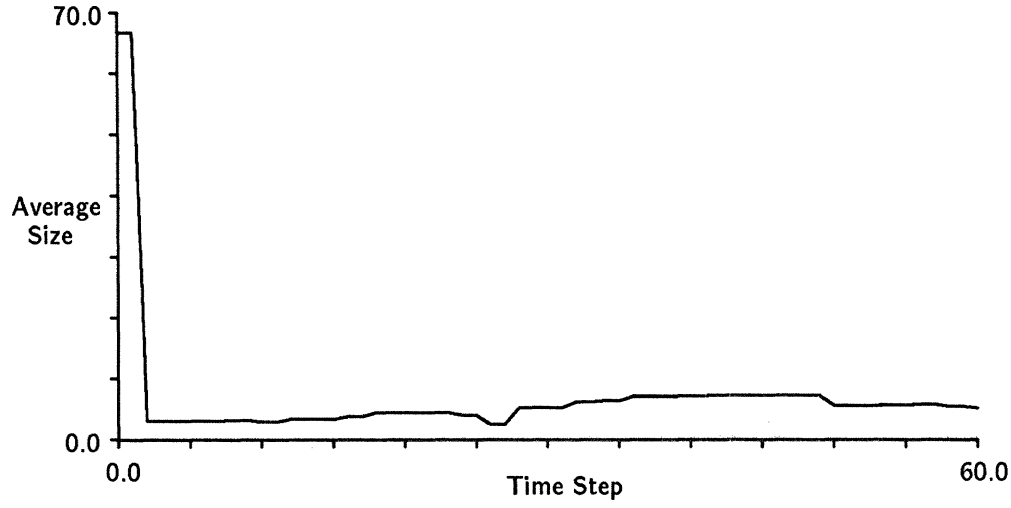


Figure 5.13: *Average size of expressions vs time*

do. Shortly into the instruction fetch, the average expression size settles down to stay roughly constant or grows very slowly. Therefore the 2^E and E^3 terms in the cost can be approximated as constants.

For purposes of comparison, combinational test generation is NP-Complete, so we take it to be $O(2^n)$ in the worst case. In practice, i.e., for realistic circuits, combinational test generation is roughly $O(n^3)$ [williams79]. Sequential test generation is much worse. Since it may be necessary to visit each of the 2^m states where m is the number of bits of memory in the circuit, and the combinational problem associated with each state may require exponential work, test generation for sequential circuits is $O((2^n)^{(2^m)})$.[7]

5.7 Summary

DB-TG is based on four ideas: (i) the designed behavior heuristic, i.e., that a circuit should be tested using its normal operations, (ii) the strategy of embedding expert-supplied component tests, (iii) that the key to embedding component tests into a

[7] [breuer76] gives 4^m as an upper limit on the number of states required, because the D-vocabulary involves 4 non-X values (i.e., 0, 1, D $\overline{D}$), implying $O((2^n)^{(4^m)})$ work is needed. However, the lower bound holds, because the brute-force approach of exhaustively fault simulating all input sequences up to 2^m in length fits under the lower bound.

circuit is knowing relationships between the circuit operations and the component operations, and (iv) computing operation relations by simulation and matching focuses the test generator on realizable circuit behavior rather than on potential behavior. We introduced the notion of behavioral subsumption to explain how operation relations help embed component tests: the test generator finds a simulated component operation, i.e., an example of known-achievable circuit behavior, that subsumes the test and then converts the simulated operation into a test by solving the operation relations.

Each of these ideas was explored from the perspective of its effect on the test generator's soundness, completeness and efficiency. In the final analysis, DB-TG is neither guaranteed sound nor complete: there are situations where it can produce incorrect tests (it warns when this may have happened) and where it can fail to find a test. Sound and complete algorithms exist at the moment, but they are unusably slow for complex sequential circuits. An effective, fast, heuristic solution is needed for generating tests for these circuits, i.e., the kind of solution human test experts currently provide. DB-TG is such a heuristic solution whose primary advantages are: (i) focusing on designed (known-achievable) behavior rather than potential behavior reduces the size of the search space; (ii) operation relations are a compact representation circuitry surrounding a component, reducing the cost of embedding component tests and (iii) the test generator uses abstract representations (e.g., operation relations and effects summaries) and computational steps that perform a lot of work at once (e.g., embedding pre-written tests and solving operation relations with placeholders).

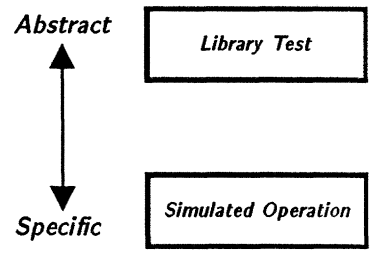
The analysis exposed two fundamental tensions in test generation: abstraction vs. specificity and regular vs. irregular behavior. The first tension lies between needing abstract circuit descriptions for speed and needing specific predictions of fault effects for accuracy. This issue arises with the need to use components to help test themselves in circuits with feedback or other forms of reconvergent fanout. When embedding component tests at the block diagram level, the hypothesis that a particular component may be faulty does not always provide enough information to accurately predict fault effects. Similarly, operation relations are inaccurate when their derivation depends upon the component under test. For these reasons, DB-TG can occasionally generate unsound tests.

Relinquishing soundness goes to the heart of what is gained and lost by using abstract descriptions. The strength of abstract representations is that they are *simplified* for solving testing problems more directly. Their weakness is that they are *simplified* representations, leaving out detail that may occasionally turn out to be important. Sound alternatives to our methods exist, but they are much more expensive. Given

the experience of testing experts and the coverage experiment with the MAC-1, it is not at all clear that they are worth the expense.

The second tension lies between regular and irregular behavior. The test generator exploits regularity in the behavior of a circuit. It does this by representing behavior with algebraic expressions that can compactly describe large numbers of similar behaviors, e.g., all of the addition instructions. This compactness stems partly from removing identity data transfers (in both space and time) from the representation and partly from using symbolic variables in the expressions to denote sets of similar logic values. Regularity of behavior saves time by allowing the test generator to propagate a placeholder value once rather than specific data many times. Compactness of representation saves time by shortening the distance values have to propagate. Which behaviors are similar to each other depends critically upon on the vocabulary of operators in the expressions in a way that we do not fully understand yet, and further work needs to be done in this area.

Tuning the test generator for regular behavior causes problems when the circuit behavior is irregular. We have labeled this the granularity problem because the test generator's representations for behavior become fragmented, making it more difficult to match a fixed set of pre-written component tests against achievable circuit behavior. The granularity problem is the prime source of incompleteness in DB-TG. The next chapter suggests two distinct approaches for solving the granularity problem. The key ideas are to reduce the granularity of component tests or to increase the granularity of the achievable component behavior.



Chapter 6
DB-TG: The Fragmentation Problem

Summary: This chapter describes additional methods used by the designed behavior test generator to overcome the fragmentation problem, which can cause the methods in chapter to 4 fail. The central insight is that the test generator's representations of desired behavior (component tests) and achievable behavior (circuit simulations) lie on a continuum of representations of greater or lesser generality. The fragmentation problem can be solved by moving component tests and simulated operations along this spectrum. One class of method involves making component tests more specific by splitting them into cases, and a second class involves making simulated operations more general by modifying the circuit using DFT techniques.

6.1 The Fragmentation Problem

The fragmentation problem arises in two ways. First, unused and inaccessible component functionality can make it impossible to fully execute a library component test. If DB-TG cannot execute a test fully, then it rejects the test and looks for another in the library. If no others are present, the test generator fails.

DB-TG has a language for describing achievable circuit behavior, and behavior graphs are expressed in this language. This language includes sets of simulated operations and node values expressed as functions of primary circuit inputs. Functions can be expressed as compositions of simple arithmetic functions, selection, concatenation and bit-field extraction. Like all languages, some things can be expressed compactly in the language and other things cannot be. The fragmentation problem also arises when circuit behavior is not regular in a way that can be expressed compactly in this

language. In this situation, DB-TG's description of a circuit's achievable behavior must be broken into many small pieces, i.e., it is fragmented. When the pieces are small enough, every piece associated with a component is insufficient to support full execution of the component test, again causing DB-TG to reject the test.

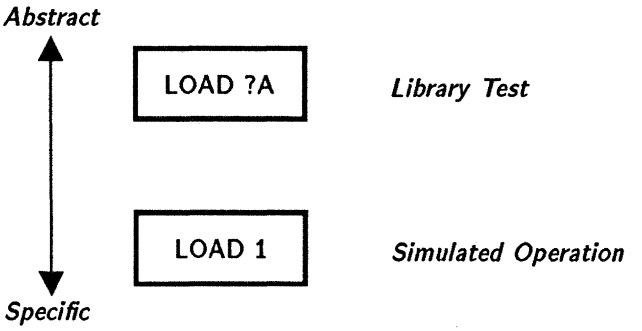


Figure 6.1: *The fragmentation problem causes simulated component operations to be more specific than the component tests, preventing the system from embedding the tests.*

Figure 6.1 illustrates the result of the fragmentation problem: a simulated component operation that is more specific than the component test. The layout of this figure suggests two approaches to solving this problem: (i) make the test more specific, i.e., conform to the constraints of the circuit by choosing more specific component tests that can be embedded using the available simulated operations, or (ii) make the simulated operation more abstract, i.e., modify the circuit design making it possible to embed the test. Sections 6.2 through 6.4 introduce three new ways to make component library tests more specific:

1. **Fine Grain Component Tests**: Try embedding several simple component tests instead of a single, complex one. Each simple test demands less accessibility and is more likely to be achievable. Two techniques are introduced: (i) break up an existing test into simpler tests and (ii) have the expert supply a range of progressively simpler tests.

2. **Parameterized Library Tests**: It is not feasible to provide component tests for every size of a bit-parallel component like an adder. We therefore capture expert testing knowledge in programs that can examine a component *and the surrounding circuitry* and write a component test on-the-fly. Two techniques are introduced that differ in the kind of information they gather about the

surrounding circuitry: (i) gather no information about context, i.e., examine component structure only (e.g., size) and (ii) examine fully-instantiated (no variables) operations in the behavior graphs.

3. **Focussed Application of Gate-Level ATG**: We can combine the strengths of DB-TG and gate-level test generation to test combinational components whose gate-level models are available. In this marriage, the DB-TG handles the sequential aspects of the circuit and the gate-level test generator handles the details of the component, augmenting or replacing the component test library.

These techniques provide a counterpoint to the way hierarchical test generators work. Hierarchical test generators work bottom-up, from gate-level, fine-grained representations of the circuit to more abstract representations. These three techniques work top-down and rely on the heuristic that the abstract representations will be sufficient most of the time and detailed examination of the circuit will be unnecessary. These techniques raise fault coverage in the MAC-1 from 85% to 94%.

Section 6.5 takes the opposite viewpoint. Wu's Design-For-Testability Advisor [wu88] can add test mode operations to a circuit, e.g., to load register values via a scan path. The additional circuit operations result in more abstract simulated component operations in the behavior graphs that can be used to embed additional tests. Running this program on the MAC-1 yields suggested circuit modifications that enable DB-TG to reach 97% of the stuck-at faults.

6.2 Fine Grain Component Tests

This section describes two techniques for breaking up component tests into smaller pieces. By splitting up a component test, it may be possible to find *separate* simulated operations that subsume separate pieces of the test.

6.2.1 Test Specialization: Substitute Test Data in Early

When embedding a primitive component test, DB-TG enforces the following relationships:

$$\text{SimulatedOperation} \supseteq \text{TestOperation} \supseteq \text{TestData}_i$$

where SimulatedOperation is an example of known-achievable behavior appearing in a behavior graph, TestOperation and TestData_i are the operation and data parts of the test from the component test library, and $\supseteq$ denotes behavioral subsumption (see

section 5.3.1). DB-TG embeds TestOperation once, makes m copies of the solution, where m is the amount of test data, and substitutes each element of the test data TestData_i into one copy of the solution. Separating the test into TestOperation and TestData reduces work by having the test generator embed all test data in the same way: SimulatedOperation must subsume the test operation and every element of the test data for this method to work.

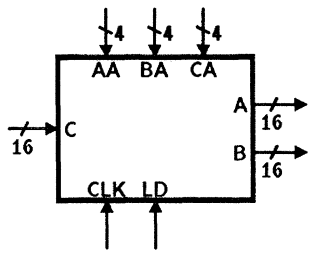
This optimization can be given up to improve coverage by substituting TestData into TestOperation *before* trying to embed TestOperation into SimulatedOperation. This creates simpler component tests – one for each line of test data – which can be embedded separately thereby increasing the likelihood of success. Substituting TestData in early is called **specializing** the component test.

For instance, the NODE test in figure 6.2.c cannot be used directly to exercise the BA address input of the MAC-1 Register File (figure 6.2.a) because there is no *single*, simulated example of the node holding a value that is general enough. Figure 6.2.b shows the actual values that BA can be set to. Note that each of these values appears separately in the behavior graphs.

Since the test data `0101` and `1010` appear among the list of achievable values in figure 6.2.b, this problem can be solved by substituting the test data into the test operation to create two new component tests (figure 6.2.d). Each of the new tests partially exercises the node, and they do together what the original component test would have done. The test generator can now succeed by embedding these two component tests into *different* parts of the behavior graphs.

Specializing a test can affect its fault coverage description, hence some tests are not candidates for specialization. In the carry chain adder test, for instance, each group of test data is intended to be a completely separate exercise. Specializing this test does not affect its coverage. However, in tests for sequential devices, the sequence of test data is important. It can be critical that a component execute no extra operations during the test, and specializing such a test could drastically reduce its coverage. These tests are marked by the expert so the program will not specialize them.

When the current implementation does specialize a test, it embeds as many of the separate pieces as it can – it does not require that they all are successfully embedded – on the assumption that a partial test is better than none. This also affects the fault coverage description. I currently have no good way to describe the fault coverage of a broken up test that has been partially embedded and instead rely upon fault simulation to determine the coverage.



(a) The MAC-1 Register File

0000
0001
0010
0011
0100
0101 ←
0110
0111
1000
1001
1010 ←

(b) Addresses 0-10 can be supplied to the BA address input of the register file. These addresses appear in separate simulated operations. Other addresses are not achievable with the current microcode.

Test Operation value(Node, ?data)

Test Data

0101
1010

specialization →

value(Node, 0101)

value(Node, 1010)

No separate test data

(c) The NODE Test (one version)

(d) Two partial NODE tests

Figure 6.2: *Embedding partial tests: the test generator cannot embed the NODE test (c) for the BA input of the register file (a), because none of the simulated operations of that node (b) are general enough. Subfigure (d) shows the set of partial NODE tests that come from specializing the NODE test. Together these partial node tests are equivalent to the original, but DB-TG can embed each one separately.*

6.2.2 A Hierarchy of Component Tests

The technique of specializing component tests can break up a single component test into a potentially large set of very specific tests so that they can be embedded separately. It is not always necessary to break up component tests so finely. A less extreme solution is to (i) ask the experts for a variety of tests for each component that exercise pieces of component behavior and exercise components in different ways, (ii) place these tests into an and/or hierarchy with the tests that demand the most access at the top, and (iii) have the test generator start at the top of the hierarchy trying to embed a test and work its way down until it succeeds or runs out of tests.

For example, consider how the expert says to test a multiplexor (figure 6.3). This test has a complex structure. The expert tests the ability of the MUX to propagate values from each input to the output, referencing a test for another kind of component (a DATAPATH) to express this. The MUX test has optional parts: while testing the path from one input to the output, the expert would like to hold the other inputs constant to reveal subtle flaws in an MOS implementation that cause feedback and state behavior in the MUX. If holding the other inputs constant is impossible, however, the expert will still be satisfied with the rest of the test.

Figure 6.3 shows an and/or tree of component tests that corresponds loosely to the expert's test for a two input MUX. Test trees are implemented using primitive and compound tests (described in section 4.5). Solid boxes represent primitive component tests, and dashed boxes represent compound component tests that refer to the tests below them as subroutines. If the lines connecting a test with its subroutines are not connected by an arc (e.g., MUX-2 IN-0 in the third row), then subroutines are tried in left-to-right order and the first than can be successfully embedded is used. If the lines connecting a test with its subroutines are themselves connected by an arc (e.g., MUX-2 FULL-CONTROL), then all subroutines are tried and all that succeed are used. Figure 6.5 describes what the primitive and compound tests in this figure do.

Due to the way the MAC-1 microcode controls the flow of data through the datapath, the datapath mux in the MAC-1 cannot be controlled arbitrarily. Thus MUX-2 FULL-CONTROL cannot be embedded. However, more specific tests lower in the hierarchy can be embedded. The starred boxes show the tests that DB-TG actually uses to exercise this MUX. These tests catch 100% of the stuck-at faults in a boolean implementation and some of the stuck-open faults in an MOS implementation. DB-TG does as well as the expert would have by selecting pieces of the component test from the library. It does better does better here than a gate-level test generator would have because it applies component test knowledge (e.g., about

To test a MUX, test the path from each input to the output as a DATAPATH while holding the other inputs at a constant background value, (e.g., all 0's). Repeat this after inverting the background value. The datapath test must be successfully embedded. However, if the other inputs cannot be held constant, they can be ignored.

Figure 6.3: *A multiplexor component test (in English)*

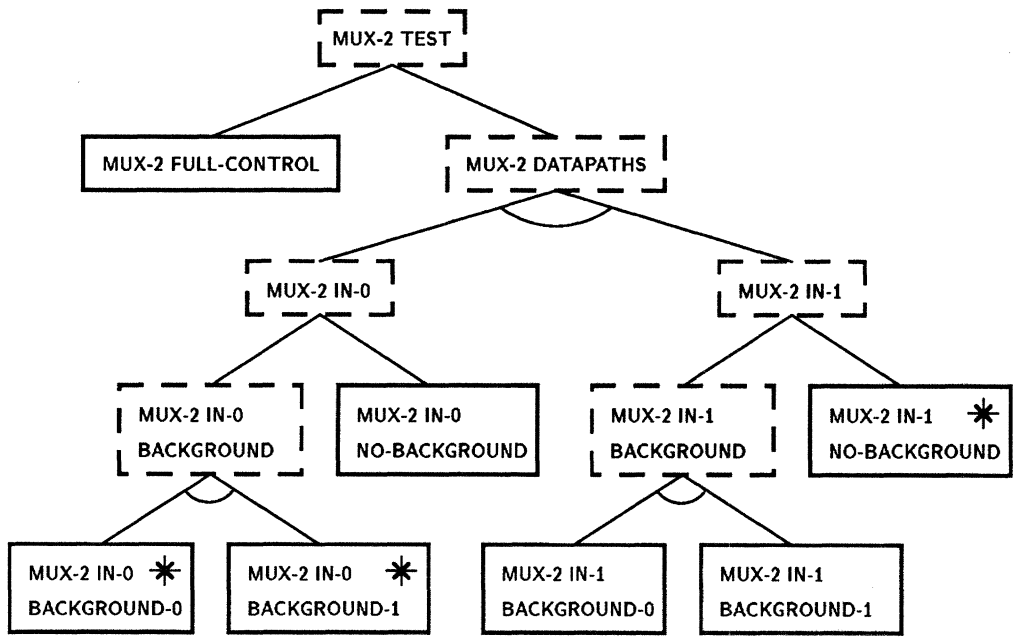


Figure 6.4: *This hierarchy of component tests corresponds to the expert-supplied test in figure 6.3. Solid boxes hold primitive component tests, and dashed boxes hold compound component tests that refer to the tests below them as subroutines.*

- MUX-2 TEST is the top-level entry. It first tries to instantiate MUX-2 FULL-CONTROL. Failing that, it calls MUX-2 DATAPATHS.
- MUX-2 FULL-CONTROL exercises a MUX fully. It can be instantiated if both inputs are completely controllable and the output is observable. The test data was generated by a conventional test generator from a gate-level model of a MUX.
- MUX-2 DATAPATHS calls MUX-2 IN-0 to test the datapath from IN-0 to the output and MUX-2 IN-1 to test the datapath from IN-1 to the output.
- MUX-2 IN-0 first tries MUX-2 IN-0 BACKGROUND. Failing that, it tries MUX-2 IN-0 NO-BACKGROUND.
- MUX-2 IN-0 BACKGROUND calls two primitive tests that differ in the background values they use.
- MUX-2 IN-0 NO-BACKGROUND exercises the path from IN-0 to the output and does not try to control the other input. The test can be instantiated if IN-0 can be selected, IN-0 is controllable, the output is observable. This test applies a diamond pattern to the input and observes it at the output.
- MUX-2 IN-0 BACKGROUND-0 is identical, except that it sets the other input to all 0's.
- MUX-2 IN-0 BACKGROUND-1 is identical, except that it sets the other input to all 1's.
- And so on. The primitive tests in the right half of the tree correspond to those in the left half, except that they operate on IN-1, the other data input.

Figure 6.5: *This figure describes what the component tests in figure 6.3 do.*

background values) that conventional test generators do not have.

6.3 Parameterized Component Tests

This section describes the technique of storing parameterized tests in the component test library. Section 6.2.2's solution of expanding the library to include multiple versions of tests could conceivably open pandora's box. How many test versions must we include for components that are found in many sizes, like adders, or components that manipulate bit-fields? Consider the ALU/AND operation in the MAC-1: there are 120 contiguous bitfields[1] within the 16 bit ALU, and any of them could potentially be used by the microcode to do masking operations. Must the library include AND tests for every possible bitfield?

Our solution is to write tests for components like adders and bit-parallel AND with their size as a parameter. More generally, we capture expert testing knowledge in programs that can examine the component and the surrounding circuitry and then write a component test based on what it finds. This technique allows tests to be created on-the-fly, effectively making the library much larger than could be stored explicitly.

6.3.1 Example #1: A Parameterized Adder Test

Figure 6.6 shows a program that generates test data for carry chain adders of arbitrary width. The test generator uses this program to test an adder by first examining the circuit model to determine the adder's width, running this program and encapsulating the result as a newly created component test and embedding the component test as usual. Sample output from this program for 16-bit adder appears in figure 4.4.b on page 98.

The important point here is that some component types are implemented in so regular a way that simple, short programs like this one can express how to test them. Components such as bit-parallel logic functions, adders, comparators, parity generators and memories fall into this category.

6.3.2 Example #2: Designing a Register Test On-The-Fly

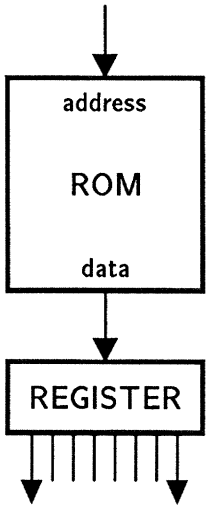
[1] Fields at least two bits wide, e.g., bits 0-11.

```
(defun TEST-DATA-FOR-CARRY-CHAIN-ADDER (size)
  (labels ((recur (pattern element size)
             (if (< size 0)
                 0
                 (+ (recur pattern element (- size 1))
                    (* (expt 2 size)      ; the slow way to shift
                       (bit pattern (mod (+ size element) 8)))))))
    (loop for i from 0 to 7
          collect (list (recur #*01110001 i (- size 1))
                        (recur #*01101010 i (- size 1))
                        (bit   #*00111100 i)))))
```

Figure 6.6: *This function creates a list of 8 test cases for exercising a carry chain adder that is SIZE bits wide (figure 1.6 shows the output for SIZE=16). This program illustrates how succinctly tests for some regular component types can be expressed. Tests for bit-parallel logic functions, adders, comparators, parity generators and memories are all roughly this short.*

In the previous example, DB-TG examined component structure (i.e., bit width) to design a test on-the-fly. In this example, DB-TG designs a component test by examining component *behavior* achievable in the context of a larger circuit. Figure 6.7.a shows a simple circuit. The register's output is directly observable, but its input is controllable only indirectly through the ROM. The 7 operations of this circuit all involve applying a specific ROM address, clocking the register to load the contents of the ROM at that address and observing the register outputs. The circuit operations give rise to the simulated register LOAD operations in figure 6.7.b. DB-TG cannot embed the normal register test because there is no sufficiently general simulated operation, e.g., a load with variable instead of constant data. Test specialization does not work because none of the test data from register tests (figure 6.7.c) matches the ROM contents.

To solve this problem, we return to one of the basic ideas behind the designed behavior approach: look at behaviors known to be achievable and ask if they constitute a test. A simple way to test a register is to see whether each bit position can hold both 0 and 1. This can be accomplished by selecting a subset of the simulated LOAD operations that cover the possible bit values. DB-TG has a simple algorithm for finding near-minimal sets of values that cover possible bit-position values, and in this case, the program selects the three LOAD operations marked with arrows. Each of



ROM contents

00000000
01100110
01101001
11001001
11011110
00100110
00100100

(a) *The ROM provides only restricted inputs to the register making it difficult to test*

LOAD(register, 00000000)
LOAD(register, 01100110)
LOAD(register, 01101001)
LOAD(register, 11001001) ←
LOAD(register, 11011110) ←
LOAD(register, 00100110)
LOAD(register, 00100100) ←

(b) *Here are the simulated load operations. The marked operations form a covering set. Executing them and observing the outputs accomplishes the goal of the REGISTER test, i.e., seeing each bit position hold both 0 and 1.*

Test Operation LOAD(register, ?data)

Test Data

00000000
11111111

(c) *One version of an 8 bit REGISTER Test*

LOAD(register, 00000000)
LOAD(register, 00000000)
LOAD(register, 11011110)
LOAD(register, 01101001)
LOAD(register, 00000000)
LOAD(register, 11011110)
LOAD(register, 11011110)
LOAD(register, 01101001)
LOAD(register, 01101001)

(d) *This sequence of load operations implements a more sophisticated test by covering all 0-1, 1-0, 0-0 and 1-1 transitions.*

Figure 6.7: *Designing a register test on the fly*

these load operations is then converted into a test operation with no extra data and separately embedded.

A more sophisticated version of the register test checks the ability of each bit position to make 0-1, 1-0, 0-0 and 1-1 transitions. DB-TG can also design register tests like this on-the-fly by finding near-minimal sequences of operations that cover all transitions on all bit positions. Such a sequence is shown in figure 6.7.d. Similarly, an even more sophisticated test could be written to construct sequences to detect shorts between adjacent bits.

The program can do these examples. However, it cannot currently do the similar example of testing the MAC-1 μIR because the algebraic rules for manipulating bit field expressions (describing where the μIR outputs go) are insufficiently powerful.[2]

6.4 Focussed Application of Gate-Level Test Generation

This section shows how to combine the strengths of DB-TG with those of a gate-level test generator to create tests for combinational components on-the-fly. In this marriage the gate-level test generator handles the details of exercising the component and DB-TG handles the sequential aspects of the surrounding circuit.

The method works by putting the component into simple combinational circuits that, taken together, are equivalent to the sequential circuit in terms of access. There are several steps: DB-TG first picks a simulated component operation and extends the operation relations associated with it all the way to circuit inputs and outputs. The extended operation relations are called an **embedding**. Since they involve purely functional relationships between circuit and component I/O, the embedding is effectively a slice of combinational behavior cut from the circuit's sequential behavior (see figure 6.8). The next step is to convert the embedding into an equivalent circuit, called a **combinational equivalent**, that fits around a gate-level model of the component. Finally, the combinational equivalent and the component model are given to

[2]An expert might test this register in yet another way. Assuming that the ROM must also be tested and knowing that ROMs are tested by exhaustively reading out their contents, an expert would realize that testing the ROM would also test the register as a free side-effect. (Depending on the ROM contents, testing the ROM may not test the register exhaustively, but it would test the register as well as possible and as well as is necessary to ensure it will work properly in the field.) Hence the expert would "test" the register by testing something else. The process of combining goals during problem solving is well understood in the AI and testing literatures. However, the details of doing this with DB-TG's representations have not yet been worked out. In particular, we need a kind of qualitative fault simulation, discussed under future work in section 8.3.5.

the gate-level test generator to produce tests for component faults.

Section 6.4.1 shows an example of an embedding and the combinational equivalent it gives rise to. Section 6.4.2 describes how embeddings are produced. Sequential circuits of only moderate complexity can give rise to very many embeddings. Section 6.4.3 describes several ways to avoid redundant embeddings and to simplify non-redundant embeddings further.

The problem of interfacing an embedding with a gate-level component model raises the question of when should a test generator shift between levels of abstraction during signal propagation? This is a standard question (here labeled the level-shift problem) that implementors of hierarchical test generators must answer, although the answer takes an unusual turn in our application. Section 6.4.4 covers this issue.

6.4.1 An Example of a Combinational Equivalent

DB-TG derives combinational embeddings from its operation relations and effects summaries. For instance, figures 6.10.a and 6.10.b show the operation relations between ALU/AND and MAC-1/JUMP, and figure 6.10.c shows the Effects Summary for MAC-1/STORE. Figure 6.11 shows a combinational embedding derived from this information.

The MAC-1/JUMP instruction branches to an address contained directly in the instruction, and the ALU is used to mask out the opcode before storing that address in the program counter. The operation relations shown are those between the jump instruction and the instance of ALU/AND that does the masking. The :FIELDS expression describes a word made up of separate bit fields; figure 6.10.b shows this :FIELDS expression in a more familiar notation. Each field in the expression is represented as (FIELD low high data), where low and high describe a range of bits. (NBITS 12 ?ADDR) says that the variable ?ADDR can hold 12 bit values. The :FIELDS expression shown here describes the jump instruction itself before the opcode is masked out.

The JUMP instruction puts the output of the ALU in the program counter, shown in 6.10.b by the parameter relation ?OUT = ?PC1. If a STORE instruction follows the JUMP, then this value will be written to the address bus as shown by the effects summary in figure 6.10.c.[3]

Figure 6.11 shows a circuit that is equivalent, from the perspective of one ALU

[3]In fact, any instruction can follow the JUMP with the same effect. DB-TG does not realize this, though and just uses the first method it can find for observing the value of the PC.

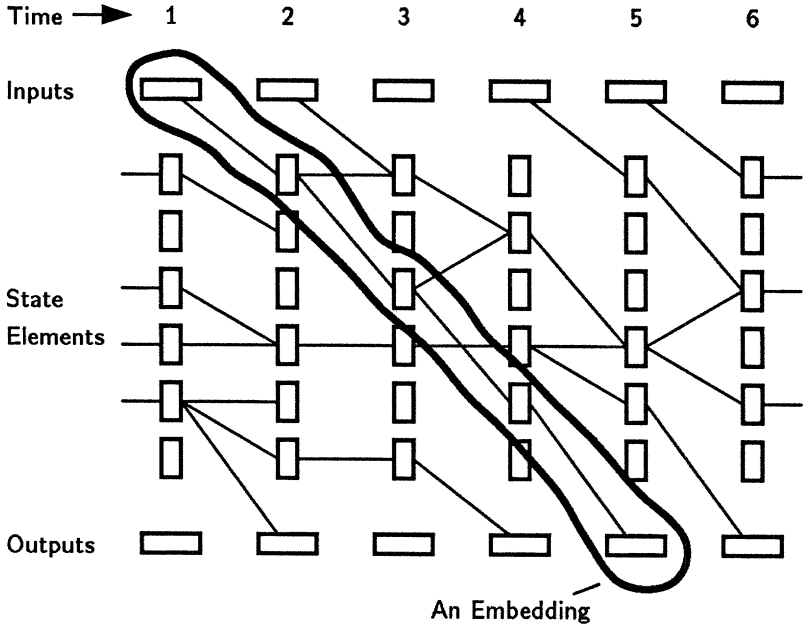


Figure 6.8: *An embedding: the figure shows a circuit at six points in time. Each column corresponds to the state of the circuit between one operation and the next. The circuit inputs are at the top of the figure, replicated once for each time, and the circuit outputs appear similarly at the bottom. The circuit state registers appear in columns between the inputs and outputs, again once per time. The lines represent data being operated on as it flows through the circuit. The bold line shows an embedding surrounding the operation where the third state register loads a value at time 3.*

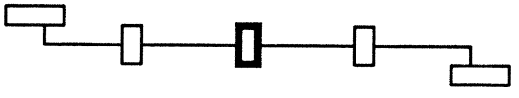
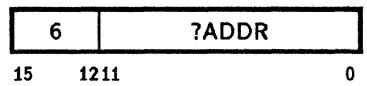


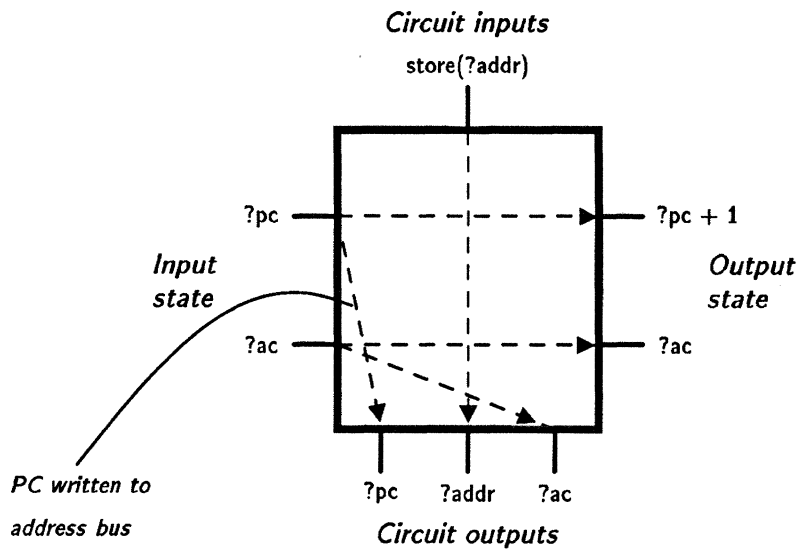
Figure 6.9: *The same embedding: the component under test (a state element) is in bold. All other state elements are treated as noops. Again, the lines represent data being operated on. This combinational "circuit" is a piece of the sequential circuit above and offers a subset of its access to the component.*



(a) The :FIELDS expression in (b) represents a word composed of a 4-bit field and a 12-bit field. In this case the 6 is the JUMP opcode and the word is the JUMP instruction itself.

?IN-1 = (:FIELDS (FIELD 12 15 6) (FIELD 0 11 (NBITS 12 ?ADDR)))
?IN-2 = 4095
?OUT= ?PC1

(b) Operation Relations between ALU/AND and MAC-1/JUMP



(c) The Effects Summary for MAC-1/STORE. Here, the PC gets written to the address bus.

Figure 6.10: *An Embedding: the operation relations (b) extend from the ALU back to a primary input (the address bus) and forward to a state register (the program counter). These relations are extended using the effects summary (c) to a primary output (the address bus) to form an embedding for the ALU/ADD operation. The combinational equivalent is shown in figure 6.11.*

operation, to the MAC-1 executing a JUMP followed by a LOAD. In this case, the network surrounding the ALU is very simple, just some wires and constants on some of the inputs. After inserting a gate-level model of the ALU, this equivalent circuit is given to a combinational test generator to generate tests for the ALU without having to deal with the complexity and sequentiality of the processor that surrounds it. If any undetected stuck-at faults remain in the ALU after test generation, then DB-TG finds another set of operation relations, constructs another equivalent circuit, and runs the gate-level test generator again on the remaining faults. As there can be many sets of operation to choose from, DB-TG uses several heuristics to limit the choice. These heuristics are described later.

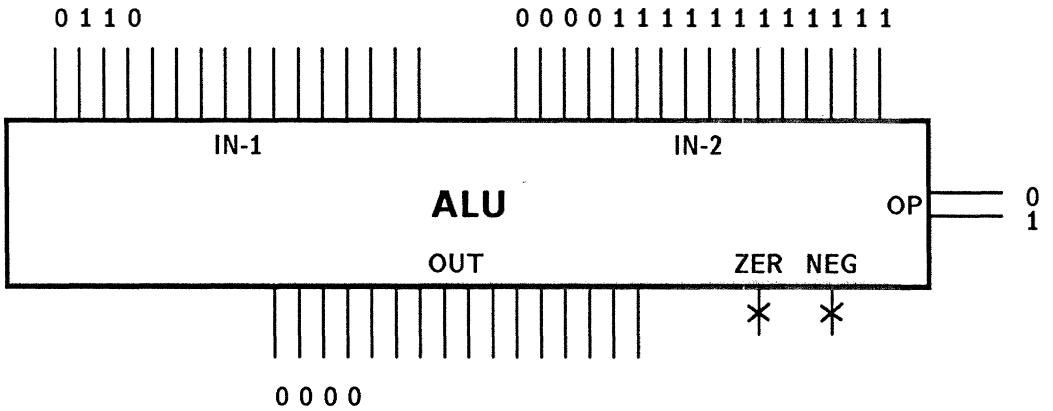


Figure 6.11: *This figure shows a combinational equivalent surrounding the ALU. In this case, the equivalent is a set of assignments to the some of the ALU's inputs (which constrain some outputs in turn) and has no internal structure of its own. The ALU's NEG and ZER outputs are not included in the equivalent, so they are not observable.*

The purpose of computing embeddings and combinational equivalents is to put the component into simple, equivalent, combinational circuits that provide the same controllability and observability that the sequential circuit provides. Having a complete set of combinational embeddings would allow us to reduce the problem of testing a component in a sequential circuit to the problem of repeatedly testing that component in each of a set of combinational circuits. However, we do not actually take the idea this far because, in the worst case, the cost of doing so is greater than conventional test generation applied to sequential circuits.

Instead, we use embeddings and combinational equivalents as another method of custom designing component tests when the pre-written tests do not fit. For this

purpose, the combinational equivalents act as descriptions of the controllability and observability of a component in a sequential circuit in a form that a conventional gate-level test generator can use. The next section describes: (i) how embeddings and combinational equivalents are derived, (ii) how embeddings can sometimes be simplified beyond the operation relations they come from, further reducing the cost of running the combinational test generator, and (iii) how redundant embeddings can be bypassed.

6.4.2 Computing Embeddings

An embedding is *roughly* equivalent to an operation relation. The difference is that an embedding goes from component inputs all the way to circuit inputs and component outputs to circuit outputs, whereas an operation relation, by definition, stops at circuit state registers.

Embeddings are computed by using a circuit's effects summaries and the State Planner to extend operation relations all the way to circuit inputs and outputs. The process first extracts the operation relations for each simulated component operation. If the operation relations connect some component inputs to some state registers, then it uses the State Planner to work out how to control the registers and substitutes the transformation functions composed by the planner into the operation relations. This gives a single set of functions connecting circuit inputs to component inputs. It then uses the State Planner similarly if the relations connect some component outputs to some registers. The end result is a set of functions relating circuit inputs to component inputs and component outputs to circuit outputs.

When running the planner, the process limits the length of the operation sequences it is allowed to generate. This limit becomes a property of the embedding, e.g., a first order embedding must involve control and observe sequences one circuit operation long. Also, when the State Planner can generate several ways to control a set of registers and several ways to observe them, the process takes the cross product. For instance, if OP is the circuit operation we are starting from, A and B are two sequences for controlling the registers, and C and D are two sequences for observing them, then we generate embeddings corresponding to the sequences A-OP-C, A-OP-D, B-OP-C and B-OP-D.

6.4.3 Reducing the Set of Embeddings

This section describes techniques for reducing the set of embeddings that are considered. The method for generating embeddings described above can produce very many embeddings. For example, there are 109 simulated ALU operations within the core group of MAC-1 instructions (load, store, add, subtract and jump); these give rise to 2725 first order embeddings. Some way to reduce the set of embeddings examined by the program is necessary.

To reduce this large set of embeddings, we first apply two control heuristics: (i) restrict the set of circuit operations under consideration, which in turn restricts the set of state registers and (ii) restrict the kinds of operation relationships and effects summaries allowed to only those using identity or bit-parallel relations. These heuristics are currently under human control, since we do not yet have enough experience with their effect on the tradeoff between test generation cost and coverage. When we restrict the set of circuit instructions under consideration to the core group of MAC-1 instructions (i.e., LOAD, STORE, ADD, SUBTRACT), the program determines that there are only two relevant state registers: the program counter and the accumulator. When we tell the program to look only for first order embeddings generated from operation relations using bit-parallel functions, it finds 9 embeddings.

Next, the program removes embeddings that provide less controllability, observability, or both than another embedding. The program identifies redundant embeddings using a method based on behavioral subsumption. The expressions on the input side of the embedding represent the set of values that can be achieved on the component inputs and the expressions on the output side represent the set of values that can be observed at a circuit output. For instance, an identity relation between the circuit input and the component input means that all values are achievable, i.e., all possible bit patterns on that input. The following fields expression represents the set of values with a 1 in the low order bit, i.e., all 16 bit odd numbers if the bit patterns are interpreted as integers.

```
?ComponentInput = (:FIELDS (FIELD 0 0 1) (FIELD 1 15 ?CircuitInput))
```

The inputs side of each embedding represents a set of achievable values. We would like the program to compute a minimal covering set. However, this problem requires exponential time with a subset comparison counting as one time step. The complexity is actually worse, since we cannot do subset comparisons in unit time. Therefore the program computes a near-minimal covering set. The program can compare embeddings that fall into several categories and remove redundant ones.

```
;;; If a pair of expressions denote the same set, we want
;;; to remove one. So anything subsumes itself.
subsumes(?a, ?a).

;;; Variables subsume constants.
subsumes(?a, ?b) :- variable(?a), constant(?b).

;;; If two addends of one expression share no variables, and they subsume
;;; the addends of another expression, then the first expression subsumes
;;; the second. This rule catches the case of the addition instruction
;;; allowing higher controllability of ALU/ADD than incrementing the PC
;;; does.
subsumes([+ ?a ?b], [+ ?d ?e]) :- independent(?a, ?b),
                                  subsumes(?a, ?d),
                                  subsumes(?b, ?e).
```

Figure 6.12: *Three rules for identifying redundant embeddings*

The categories are (i) compositions of addition operators (A ... A), (ii) compositions of bit-parallel functions (B ... B), (iii) composition of addition operations and bit-parallel operators where the A's come first (A ... A, B ... B), and (iv) the previous cases with selection operators (i.e., IF statements) interspersed.

We have implemented these subsumption categories written in a set of about 20 Prolog rules. Figure 6.12 shows several of the rules together with comments describing their purpose. These categories and rules are clearly not complete, but they are sufficient for a large class of datapath circuits described in terms of bit-parallel operations (e.g., AND, OR and NOT), addition and selection. For the ALU/AND example in the MAC-1, they determine that 6 of the nine first order embeddings are redundant, reducing the list of embeddings the program has to consider in detail to 3.

6.4.4 Connecting the Embedding to a Gate-Level Test Generator

The key problem when connecting an embedding to a gate-level component model is that they are usually at different levels of abstraction. A gate-level component model describes how boolean values propagate whereas the algebraic expressions comprising

our embeddings describe how integers propagate.[4] This is an instance of a classic problem that all hierarchical test generators [genesereth81, shirley83b, singh86, krishnamurthy87] encounter: when and how are values propagated between abstraction levels? For instance, when are bits combined into integers and vice versa? We label this the **level-shift problem** and describe a novel approach to it. We describe the problem in terms of a gate-level component and a higher-level embedding, but the ideas generalize to higher-level component models.

6.4.4.1 The Level-Shift Problem

There are two competing goals to be considered when solving the level-shift problem. First, propagation should occur at as high a level of abstraction as possible (e.g., propagate one integer rather than many bits) in order to save work. This is the lesson of the hierarchical test generators. Second, assuming the gate-level test generator can backtrack to the most recent relevant choice, it should have independent control over bit-level component inputs and outputs so it has maximum freedom to find effective tests. We want the choices of bit-level assignments to be independent because backtracking is likely. Recall that we invoke the gate-level test generator only in situations where there is limited controllability and observability, i.e., when DB-TG could not embed any pre-written component tests.

The problem lies in the conflict between these goals: propagation efficiency is improved at higher levels because the test generator takes larger steps, but each step is a larger choice that may be wrong. When it is only partially wrong, it cannot be partially retracted, reducing backtracking efficiency.

The Saturn test generator [singh86] is an example of the standard solution to the level-shift problem. This program changes levels at component boundaries. Given a fault hypothesis inside the component, Saturn designs a test for it and propagates signals forward and backward to the component boundary. Once all propagation settles down, all assignments necessary for detecting the fault have been made to the component input and output wires, but the values of some of these wires may remain unspecified. In order to abstract upward from a collection of bits to an integer, these unassigned wires must have values. Saturn chooses these values randomly and propagates constraints within the component to make sure the random choices are locally consistent. Then it converts the bits into an integer and continues propagating at the higher level.[5]

[4] There is no in-principle restriction of embeddings to integers rather than other objects, e.g., ethernet packets. Our examples, however, all deal with integers.

[5] A collection of bits, some of which are unassigned, could correspond to many different integers.

This solution implicitly assumes that most values will be achievable at the higher level most of the time. This heuristic has not been made explicit in the literature. While we agree with the heuristic in general and DB-TG uses it, the heuristic does not hold in situations of limited controllability, and the very situations we are considering here.

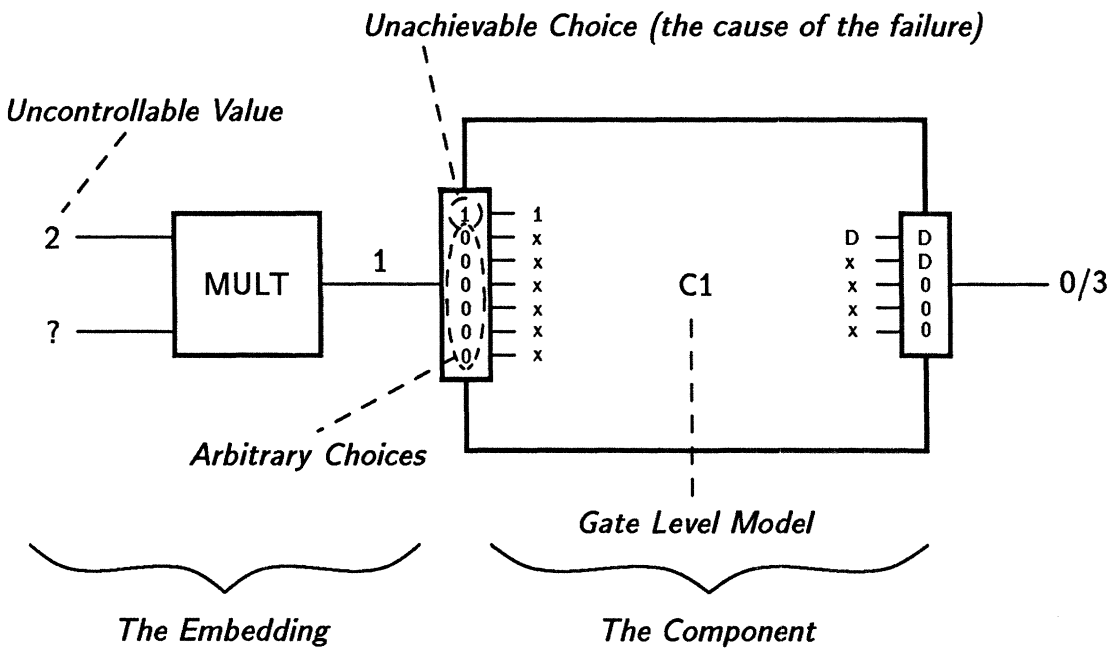


Figure 6.13: *This figure illustrates the standard solution to the level shift problem and how it can get into trouble.*

Figure 6.13 illustrates the problem with the standard solution. Saturn's task is to generate a test for a fault inside C1, which is modeled at the gate level. Suppose also that the upper input of the multiplier (MULT) has previously been assigned the value 2. The program will run into difficulty assigning blame for a propagation failure.

Saturn sensitizes a fault inside C1 and propagates forward and backward to the component boundary. Suppose, due to choices made in sensitization and line justi-

If the vocabulary of node values at the higher level includes integers but not sets of integers, then abstraction upward cannot occur unless the unassigned bits are given values. Typically, these values are chosen randomly. An alternative to choosing values for the unassigned bits is to enlarge the higher-level vocabulary to include integer sets. This can improve efficiency in some cases, but increases the complexity of the algorithm and is of dubious benefit in general.

fication, the low-order input bit is assigned the value 1. (This choice will eventually cause the test generator to fail and backtrack, since the MULT output must be an even number. The problem is that the guilt of this choice may be obscured by other choices.) Suppose also that line justification fails to specify values for rest of the input bits. In order to shift upward one level of abstraction and continue test generation, Saturn chooses values for those inputs arbitrarily. In this case, it chooses 0 for all of them. Once it has fully specified C1's inputs, it abstracts the input string of bits into the integer 1 and propagates it backwards through the multiplier. At this point, the test generator fails.

Which choice is responsible for the failure: the low-order bit or one of the high-order bits? The program cannot assign blame appropriately because it has combined seven choices made at a low level – six of which were arbitrary and necessary only to shift levels – into one choice at a higher level.

Hierarchical test generators like Saturn backtrack and retry the arbitrary choices made to shift levels. In this case, the program would have to work through 2^6 sets of choices for the high-order bits before it would succeed by trying a new value for the low-order bit.

We know that the real problem lay in choosing 1 for the low-order bit, since this made the number odd. The program, however, cannot assign blame appropriately because it has combined seven choices made at a low level into one choice at a higher level. The program must backtrack inside C1 and try other tests with *no guidance* about what went wrong. In particular, since the random choices for unassigned bits could have been the problem, the test generator must go back and try all possible random choices to ensure completeness. This is clearly unacceptable.

The problem of credit assignment would not have occurred had the program remained at the bit level. We suggest that, contrary to the lesson of hierarchical test generators, it is sometimes better to stay at a lower level of abstraction when limited controllability or observability is likely to cause backtracking. Our approach is an attempt to have our cake and eat it too, i.e., to have the propagation efficiency of high-level representations with the subgoal independence of low-level ones.

6.4.4.2 Controllability and Observability Preserving Transforms

Our approach is to apply a set of transformations to the embedding that simplify its structure while preserving its controllability and observability properties. The test generator runs with the simplified embedding, efficiently searching through the component's limited space of controllability. When it is finished, the values assigned

to the component inputs are known to be achievable and observable. These values are then propagated through the original embedding to determine values for the circuit inputs and outputs.

Because we are using a gate-level test generator, wanting independent subgoals means that we want bit-parallel embeddings. DB-TG has a set of rewrite rules for transforming bit-mixing embeddings (e.g., ones using addition or multiplication) into bit-parallel embeddings. Figure 6.14 shows 3 examples. When not otherwise specified, these rules apply to 16-bit values. The first rule converts addition of a variable and a constant into a new variable. The sum of a variable and a constant covers the entire 16-bit range of values because modulus arithmetic allows the sum to wrap around and take on values less than the constant. `?b` is a completely new variable that does not appear elsewhere in the original or any other expression. This has been the most useful transformation in the examples we have run because a common way the ALU is used with limited controllability involves incrementing or decrementing the program counter or stack pointer. Rule #2 holds as long as `?a` and `?b` are independent variables, and again `?c` is a new variable. Rule #3 captures the idea that doubling a number in base two shifts the number 1 bit to the left. There is a similar rule for multiplication by a power of 2. These rules have also been very useful because the microcode uses addition in several places to perform shifting actions.[6]

1. `(+ ?a constant)` $\mapsto$ `?b`
2. `(+ ?a ?b)` $\mapsto$ `?c` *if ?a and ?b are independent variables*
3. `(+ ?a ?a)` $\mapsto$ `(:fields (field 1 15 ?a) (field 0 0 0))`

Figure 6.14: *Three transformation rules that simplify the function while preserving controllability and observability.*

DB-TG currently uses roughly 10 of these transformations to simplify embeddings. They are applied as soon as the embeddings are generated, because they make it easier to identify and remove redundant embeddings. Figure 6.15 shows the set of first-order

[6]There are 8 rules for transforming bit-mixing embeddings into bit-parallel embeddings in the current implementation. These rules are widely applicable, however, it is not clear how close we are to having rules that cover a large enough class of circuits. Note that a missing rule will not cause the gate-level test generator to fail, it just reduces backtracking efficiency.

Solving this problem in a completely general way – not using a rewrite system – depends the ability to determine whether two functions are equivalent. This subproblem is decidable for finite domain functions (as are these) but can be quite expensive. The rewrite system is an approximate solution that executes quickly.

embeddings involving ALU/AND after the transformations have been applied and redundant embeddings removed.

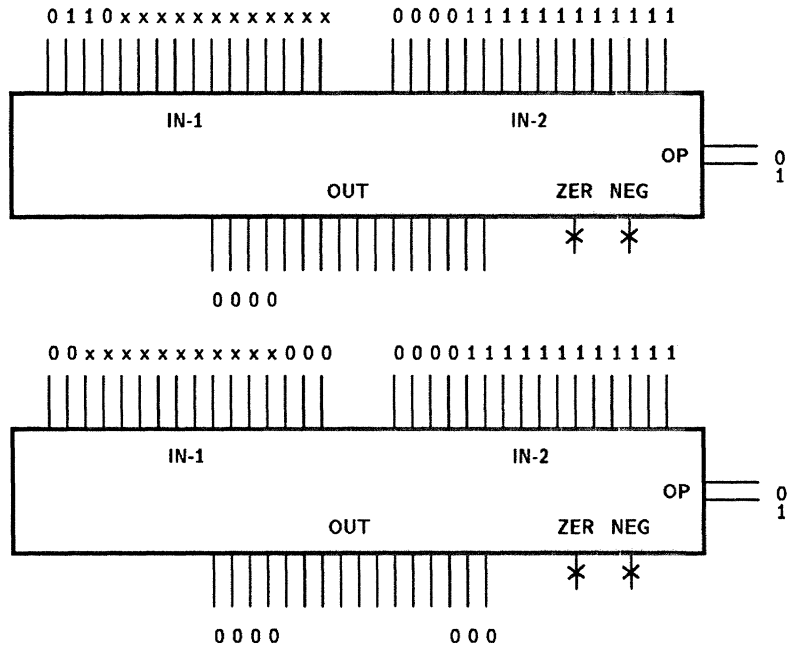


Figure 6.15: *The set (2) of first-order embeddings involving ALU/AND after the transformations have been applied and redundant embeddings removed.*

6.4.5 Implementation Status and Experimental Results

DB-TG currently computes and simplifies embeddings designed for gate-level combinational test generators. We have performed experiments on the MAC-1 ALU using two programs: (i) a commercial quality test generator supplied by GenRad Inc (the implementation of Podem in HITEST) and (ii) a lisp implementation of the D-algorithm that uses testability metrics to guide search plus a fault simulator to identify accidental detections. In both cases, we converted the embeddings into combinational equivalents by hand. This process should be straightforward to automate as it involves only assembly of gate-level models from a library.

This technique improves coverage of ALU faults by 6.7% (from 89.5% of ALU faults detected by the simple version of DB-TG to 96.2%). This technique achieves higher coverage because it can exercise the ALU/AND whereas the simple version of

DB-TG cannot. The final coverage of 96.2% of ALU faults is the maximum possible given the redundancy in the ALU model.

6.4.6 Discussion

Using a gate-level test generator to design component tests on-the-fly within the constraints of a sequential circuit continues the line of development begun with parameterized tests. This strategy is an instance of trying to match the tool to the problem: gate-level, combinational test generators like Podem are extremely effective at what they do. If they are applied in a focussed way on appropriate pieces of the problem, they can be an effective tool for helping to test sequential circuits too.

This section also covered the level-shift problem – a classic problem encountered in all hierarchical test generator – and identified a difficulty with the standard solution. The essence of the difficulty is that abstracting upwards to increase propagation efficiency can reduce backtracking efficiency. Backtracking efficiency is important in our application of generating component tests under conditions of limited controllability and observability. We increased backtracking efficiency with two strategies: (i) continue propagation at a low level where subgoals are more independent so that blame for failure can be assigned appropriately and (ii) transform the circuit representation to help keep subgoals independent. For our gate-level application, the most effective transformations were those that converted bit-mixing functions into bit-parallel functions.

If gate-level component models are not available, DB-TG can use either models for similar components when available or parameterized tests. Parameterized tests are likely to be more effective because they can be designed by testing experts to cover the likely component implementation styles. Control of these heuristics and methods is currently left to the user, because we are still experimenting to determine their relative merits.

6.5 A Synergistic Combination of Test Generation and Design for Testability

Test generation and design for testability should be equal partners in solving circuit testing problems. With several colleagues, this author has proposed an automated methodology for circuit testing [shirley87] that attempts to strike a balance between these two approaches – to apply each in its proper place. DB-TG is the test generation

portion of this methodology; the DFT portion is described in detail in [wu88]. This section briefly describes (i) the central ideas in the methodology, (ii) the ideas behind Wu's DFT advisor, (iii) one of several sets of DFT modifications to the MAC-1 suggested by Wu's program, and (iv) DB-TG's performance on the circuit after making the DFT modifications.

The idea of separating the easy testing problems from the hard and attacking just the hard ones with DFT techniques is not new to us: this basic approach has been used for years by experienced designers and reported on in the literature (e.g., [daniels85]). Our contribution lies in refining the methodology and in automating portions of it.

Designers usually know how to test most of their circuit in straightforward ways using only the operations it was designed to perform. Thus, DFT techniques are unnecessary here. The remainder of the circuit, often control circuitry, is much more difficult to handle due to limited accessibility. DFT techniques, greatly increased test generation effort or both are required to ensure testability here.

Following this lead, DB-TG is designed to generate tests quickly for the parts of sequential circuits that test engineers would find straightforward – this itself represents an advance over the state-of-the-art – and to give up quickly on the rest, deferring to DFT techniques. Subsequently, Wu's DFT advisor suggests circuit modifications to increase testability, while attempting to minimize the extra circuitry needed.[7]

Clearly, it would be preferable if DB-TG could generate easy-to-apply, high coverage tests for 100% of the circuit, and for it to do so without the designer having to take testability into account. In the past, however, this goal has been attained only for small circuits. Moreover, there is much theoretical justification for saying that it will never be attained for large sequential circuits. Combinational test generation for gate level circuits is NP-complete, and testing sequential circuits is even more difficult. Our methodology avoids the worst cases by bounding the search done during test generation to embedding pre-written component tests or the limited classes of parameterized tests.

We suggest that there is a point of diminishing returns, beyond which effort spent on test generation yields a lower return than effort spent on making the circuit more testable. Rather than expend large amounts of effort searching for increasingly subtle ways to test the existing circuit, we have designed DB-TG to stop and report its

[7]While the ideas in DB-TG and Wu's DFT Advisor fit together nicely, neither program depends on the other. From our perspective, the DFT systems of [abadir85, zhu86] would suffice, and the DFT Advisor should be able to work with another test generator capable of handling (many problems involving) sequential circuits too.

failures to a DFT advisor.

6.5.1 Wu's DFT Advisor

When DB-TG fails, we send descriptions of that failure to Wu's DFT advisor. Currently, these descriptions are lists of the component operations and circuit nodes that DB-TG could not test. The DFT advisor uses these failure descriptions to focus on circuit modifications which improve testability. The key idea in Wu's system is this: detailed descriptions of a test generator's failure provide goal-oriented and fine-grained indexes into a database of DFT suggestions.

The advisor suggests modifications by first running a simple, test generator on the components that DB-TG could not test. This test generator uses a high-level circuit model (e.g., it propagates signals through multi-bit paths) but does not take time into account. For example, one way this test generator might fail involves being unable to control the value of a register. One of the modifications indexed by this failure is "put a scan path through the register." In effect, Wu's DFT program helps the test generator by modifying the circuit just enough to get it past the difficult spots. After a designer has reviewed and approved the advisor's suggestions, the circuit description is changed and the test generator run again. This cycle continues until the testing goals are met.

DB-TG and Wu's test generator use different circuit representations and complement each other as a result. DB-TG goes to great lengths to avoid working with the detailed structure of the circuit. Therefore its failure messages often contain insufficient detail to propose useful structural changes. Wu's test generator is much less powerful than DB-TG for sequential circuits, but, because it propagates through the circuit structure, it produces failure descriptions that can be closely indexed to useful structural changes. DB-TG's chief role is therefore to focus the DFT Advisor on the really hard test generation problems.

Combining ATG and DFT techniques yields several benefits. First, it focuses the cost of extra hardware to support testing only on the most difficult testing problems. The key here lies in having an effective test generator: we do not pay for testability if the test generator can find a solution without changing the circuit. Contrast this with the scan design style which incurs cost uniformly over a circuit, because is tuned to exploit the limited capabilities of purely combinational test generators.

Second, not requiring pure ATG solutions to the small number of very hard problems can reduce test generation time by a disproportionately large amount. Moreover, the resulting tests tend to be simple and direct, thus they can be applied quickly.

These first two benefits result in a reduction of the total cost of the test-related aspects of design.

Third, this combination of factors makes the test generation problem more predictable and manageable in the face of design changes, improving the designer's ability to trade off testing costs against, for example, performance and area costs. Testing should be one factor among many that designers can balance, no more nor no less important a priori than the rest.

6.5.2 A Set of DFT Modifications

When run with the pre-written component test library, DB-TG successfully embeds tests for most of the datapath but fails to do so for the sequencer. As we have not yet automated the links in either direction between DB-TG and the DFT Advisor, we transfer the list of failing operations by hand to the advisor. Among other possibilities, the Advisor suggests that we: (i) put a scan path through the μIR, which adds shift-in and shift-out operations to this register, and (ii) provide two new pins which determine whether the circuit is in test-mode and control μIR shifting. We modified the circuit model accordingly.

Figure 6.16 shows the consequences for ATG. With the DFT modifications, DB-TG was able to achieve 97% coverage over stuck-at faults. This figure has been normalized to remove defects in the unused portions of the ALU and the register file. The raw coverage figure from fault simulation is 94%.

The new operations for loading and observing arbitrary values in the μIR plus the single-shot mode allow the test generator to properly embed its library tests for the components in the sequencer. In effect, the test generator is able to build very simple microcode sequences and execute them by shifting them, one at a time, into the μIR.

Of the remaining undetected faults, most are ALU faults that are detectable but are not caught due to the granularity problem with the ALU/AND operation. Given complete control over the microinstruction register, the test generator should be able to build microcode sequences that fully exercise the ALU/AND operation, for example, by implementing a general AND instruction. The program cannot do this at the moment, because the sequences required are on the order of 10 microinstructions long, and the simple STRIPS planning technology used by the state planner to construct operation sequences is not up to it. Section 8.3 suggests one promising way to improve the state planner for the purpose of attacking this sort of complex sequence planning problem.

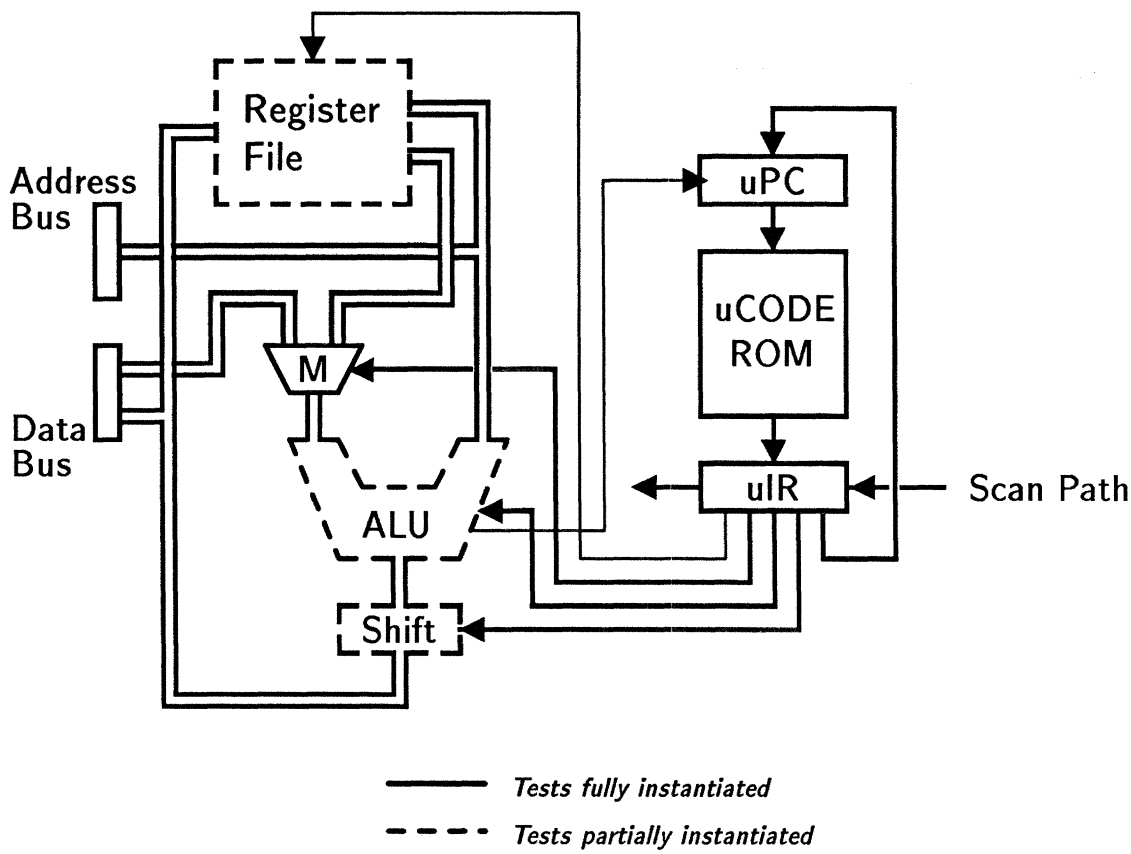


Figure 6.16: *Test generation results after adding DFT modifications: 97% of the detectable stuck-at and open circuit faults.*

6.6 Summary

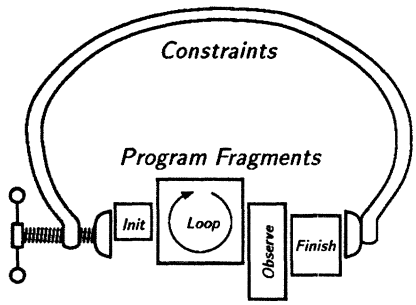
This chapter also described several extensions to DB-TG that address the fragmentation problem, the primary reason that DB-TG is incomplete. The first extension involved selecting tests that require only limited access to the component. These tests were organized in a subsumption hierarchy to facilitate trying them in order.

The second extension was to store parameterized versions of the expert-supplied tests. One way to do this involved asking the expert to describe a general pattern describing data that can be used to test a component type and then writing a program that reproduced the pattern upon demand (e.g., the carry-chain adder test). Another way to do this involved breaking up the problem of testing the component into multiple subgoals and writing a special-purpose program to search for ways of achieving these subgoals independently (e.g., the register test.)

The third extension involved combining the strengths of DB-TG and gate-level ATG to effectively test combinational components whose gate-level models are available. In this marriage, the DB-TG handles the sequential aspects of the circuit and the gate-level test generator and handles the details of the component, augmenting or replacing the component test library.

Combining DB-TG and gate-level ATG exposed an important difficulty encountered by hierarchical test generators called the level-shift problem. The essence of the problem is that aggregating values, e.g., bits into integers, is sometimes a disadvantage because it takes test generation goals that were independent (e.g., controlling individual bits) and ties them together (e.g., controlling the integer). When goals are expected to interfere, it is sometimes more efficient to continue using the lower level circuit representation. This extension also used a set of domain-specific problem transformations that preserve the essential properties needed for test generation (i.e., controllability and observability), but rendered test generation subgoals independent.

Finally, test generation is a problem whose specification is often negotiable: often the best way to solve a testing problem is to change the circuit rather than search a very long time for a solution or generate a solution that is expensive to use. DB-TG can quickly separate testing problems that would be straightforward for an expert test programmer from testing problems what would be difficult and expensive to solve. The difficult problems can then be attacked by a design for testability advisor that suggested changes to make the circuit easier to test. Intelligently combining test generation and design for testability can reduce testing costs.



Chapter 7
Generating Tests by Merging Program Fragments

Summary: Experts write test programs rather than test vectors because programs are a more readable and compact representation for tests than vectors. Test programs can be constructed automatically by merging test program fragments using expert-supplied goal-refinement rules and domain-independent planning techniques from AI. Giving the test generator knowledge of the capabilities of the tester provides additional leverage.

7.1 Introduction

Whereas DB-TG was targeted at embedding problems that give rise to highly interacting subgoals, our second program, the Program Fragment Test Generator (PF-TG), is targeted at embedding problems that give rise to weakly interacting subgoals. This second kind of embedding problem is characteristic of sequential circuits that provide relatively good access to internal components. Conventional planning technology appears to be sufficient to solve many embedding problems of this type and provides a base for exploring several new ideas about circuit testing. This work has resulted in the four claims shown in figure 7.1. There is a fair amount of mechanism needed to implement these ideas, and this section presents the ideas without diving into details. Section 7.2 describes the mechanisms needed to work through a detailed example, and section 7.3 describes the implementation in further detail.

7.1.1 The Conventional Aspects of PF-TG

PF-TG is novel in some ways and conventional in others. This section briefly describes the conventional parts. At the coarsest level of detail, PF-TG solves embedding problems in the usual way: it starts with a component test and propagates backward and forward from the component to circuit inputs and outputs to create an embedding. However, PF-TG is unusual in the representation it uses for component tests (test programs), in the kinds of propagation rules it uses (those that contain program fragments), in some of the signal tokens that are propagated (those that represent streams), and in how it fits pieces of a solution together (via late commitment and temporal constraint posting).

PF-TG can be likened to a conventional test generator in the following way: as the D-Algorithm combines D-Rules to propagate values from the component to circuit inputs and outputs, so PF-TG combines test program fragments. When the D-Algorithm finishes, all node assignments are consistent and the values assigned to circuit inputs and outputs constitute a test vector. When PF-TG finishes, all the program fragments fit together to form a test program.

PF-TG has three intellectual ancestors: Joyce's extensions to DART [joyce83], HITEST [robinson83] and the test generator described in [shirley85]. How PF-TG compares wit these ancestors is explored in chapter 8.

7.1.2 Test Programs Have More Explicit Structure Than Vectors

A test program is a sequence of computer-executable instructions for testing a circuit. Test programs are equivalent to sets of test vectors in one sense: both specify sequences of input/output pairs. Programs, however, may have more structure than the equivalent vectors, including, for instance, looping constructs and conditional statements.

This structure makes programs a good representation for tests for several reasons. First, test programs are often more compact than the equivalent vectors, because looping constructs can efficiently encode repetitive tests.

Second, test programs are more readable. (Recall the comparison between a test program and the equivalent vectors in section 1.6.2.) Explicit structure makes the test programs easier to understand, to augment, and to modify. Making the test generator's output more readable makes it more accountable. Accountability is important in turn, because circuit testing is still something of an art: fully automated techniques do not yet exist for testing the most complex circuits. Tools will solve some portions

1. **Produce Programs not Vectors**: Representing tests as programs rather than vectors makes them more compact and easier for people to understand and allows convenient access to special-purpose tester features.

2. **Merge Test Program Fragments**: Test programs can be created by merging program fragments. Goal decomposition rules and temporal constraints determine which program fragments are selected and how they fit together.

3. **Represent The Tester Explicitly**: Conventional test generators assume an impoverished model of the tester's capabilities. PF-TG uses an explicit and somewhat richer model, enabling the program to take advantage of special-purpose tester features.

4. **Propagate Typed Streams**: PF-TG can propagate tokens that represent typed streams of values, e.g., a `counting-stream`. Propagating typed streams can generate repetitive tests that are more efficient over a wider class of circuits than can propagating symbolic variables, the method of existing hierarchical test generators.

Figure 7.1: *PF-TG illustrates four new ideas about test generation.*

of the problem and human experts will solve others for the foreseeable future. While tools like fault simulators can grade test programs and vectors equally well, they do little to help the test expert modify tests to increase fault coverage. Therefore, in an environment where the expert modifies and combines the tests generated by his tools, the expert must understand what the tools are doing.

Test engineers currently write tests in variants of conventional programming languages (e.g., Basic and Pascal). PF-TG uses a simple block-structured language based loosely on the Design Waveform Language (DWL) used in the HITEST test generator [robinson83].

Test programming languages also provide convenient access to special-purpose tester features, e.g., hardware for generating memory tests or streams of counting

numbers. Test vectors – lists of circuit inputs and outputs – are insufficient for this purpose because part of what must be controlled is the tester itself. Programs are a convenient output format for a test generator that exploits tester capabilities, as described next.

7.1.3 Exploiting Tester Capabilities

Classical test generators have extremely limited models of tester capabilities. They assume a machine that can only step through a table of inputs and outputs and raise a flag when one of the measured outputs differs from the table entry. This model is simple, general and has aided the development of test generation algorithms, but it is a far cry from the machines programmed by test engineers today. Modern testers have numerous special features that are included to handle common testing situations efficiently. PF-TG has a description of these features and uses them where appropriate.

PF-TG assumes a tester architecture like the one shown in figure 7.2.a. This tester is driven by programs written in a high-level test language and running on a general-purpose computer. PF-TG also assumes that:

1. There is a single clock. The clock is either generated by the tester or by the circuit. In the second case, the tester must synchronize to the circuit clock.

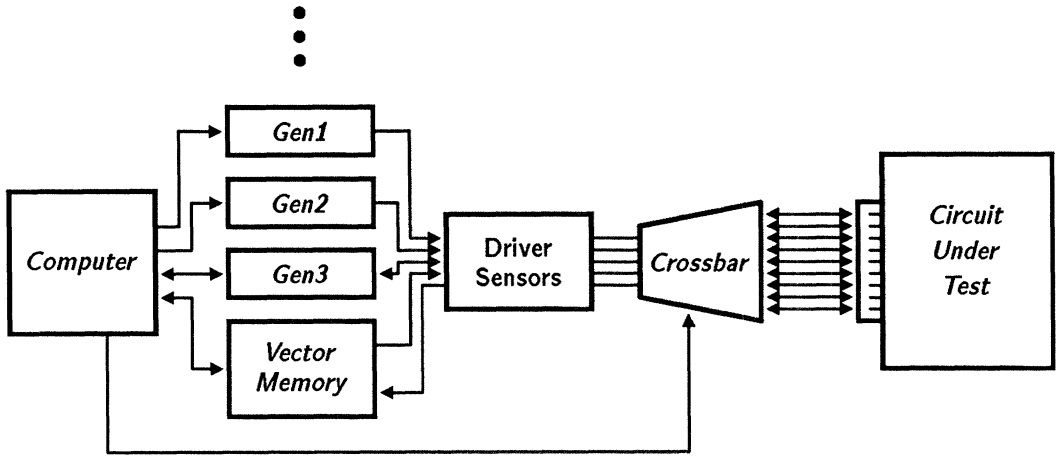
2. The tester has a vector memory for holding test patterns declared as arrays in the test language.

3. The tester has several special-purpose pattern generators (e.g., for memory tests) that create streams of values on-the-fly. These streams can be applied to the circuit or compared with circuit outputs.

4. The tester has a limited number of driver-sensors (e.g., 256) that are connected to the circuit via a crossbar switch.

This architecture describes a class of testers, and particular instances may have only a subset of these features. Some features may be implemented directly in hardware, while others may be implemented in the software that drives the tester,[1] but from PF-TG's perspective, both kinds of features are the same.

[1]Some testers are so complex that sophisticated software interfaces are required to fully utilize their capabilities, e.g., to handle detailed formatting and resource constraints. Thus, the combination of tester hardware and *software* is the real target of test generators.



(a) PF-TG assumes the tester is a general-purpose processor with a vector memory, a (possibly empty) set of special-purpose stream generators, and a crossbar switch between the driver-sensors and the circuit.

RULE 1: To supply a counting stream to ?node use this code fragment:

```
InitializeCounter(Gen1, ?low, ?high, 1);
ConnectStreamGenerator(Gen1, ?node);
  <...>
StartCounter(Gen1);
```

RULE 2: To supply a counting stream to ?node use this code fragment:

```
FOR value FROM ?low TO ?high DO
  BEGIN
    ?node := value;
  END
```

(b) These rules describe two different ways for a tester to generate a counting stream. The rules are pseudo-code equivalents to the real rules in PF-TG's (less readable) rule language. The ellipses "..." here is shorthand for a temporal constraint that allows an arbitrary gap between the parts of the first fragment.

Figure 7.2: *The tester model and two rules for using it.*

Figure 7.2.b. shows two sample rules, one of which involves a tester feature. If PF-TG discovers that it needs to generate a counting stream, it considers using these rules. Rule 1 solves the problem using a binary counter called Gen1. The first statement in the code fragment initializes Gen1, tells it the range of values to count through and the length of each step in clock cycles (1 in this case). The second statement is an instruction to the crossbar to connect the output of Gen1 with a particular circuit input, and the third statement starts the counter.

Rule 2 solves the same problem using a FOR loop that assigns the value of the iteration variable to the circuit input. By expressing the method in simple test programming language constructs, this rule passes the problem off to the test language compiler, which can generate the counting stream by: (i) executing the loop at run-time in the uniprocessor or (ii) converting the loop into vectors at compile time and applying them to the circuit via the vector memory.[2]

These solutions have different characteristics. Rule 1 produces a solution that is space efficient and fast because these circuit inputs are generated in hardware as the test is being applied. Rule 2 produces a solution that can be executed by a simple tester with no stream generators. Executing the loop directly in the tester's processor is relatively slow. In this case, the test can usually be executed more quickly by unrolling the loop into a sequence of vectors, unless the vectors do not fit in vector memory. PF-TG prefers rule 1 where applicable.

This tester model, while more accurate than the model used by conventional test generators, is still a simplification of a real tester. For instance, the stream generators in this model are connected to the pins via the crossbar switch which, due to fairly long setup times, cannot be changed on every time step. Instead stream generators are assigned to groups of pins for the duration of major pieces of the test program. In some real test generators, the situation is more complex: the value for single pins can be selected from either vector memory or a stream generator at each time step. This choice is controlled by the value coming from the vector memory (e.g. one value from the memory means apply a 1, another means apply 0 and a third means use the value from the stream generator). This capability allows streams of values to be embedded into complex input patterns. The point of this section is not that the particular tester model used by PF-TG is a complete and accurate representation of an existing tester. It is not. Rather, we are illustrating with a model that is more sophisticated than those used traditionally that a test generator can benefit from knowing the capabilities

[2]In principle, the compiler could also deduce the goal of applying a counting stream from the code and allocate a stream generator to this problem like Rule 1. However, this is a difficult problem, and existing compilers do not do this.

of the agent that will carry out the test.

7.1.4 Typed Streams: Test Data Has Structure Too

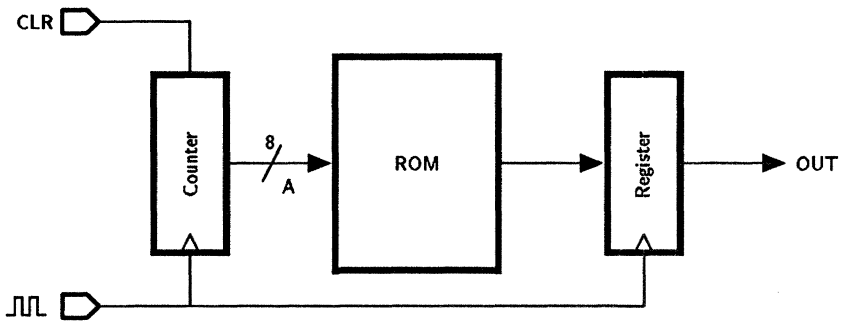
PF-TG propagates tokens representing typed streams of data to handle groups of similar goals during test generation. Many tests involve repeating a pattern of activity with changing data, and increasing efficiency by exploiting repetition is an important goal in test generator design. Classical test generators propagate tokens representing single (usually boolean) values, therefore they must solve an embedding problem repeatedly as the data changes or use complex caching schemes to exploit this kind of repetition. Hierarchical test generators and DB-TG improve on this by propagating symbolic variables, i.e., tokens representing unspecified values. A solution created in this way represents a group of similar embeddings that can be used repeatedly by substituting in the changing data.

Figure 7.3 shows an example where propagating a variable yields an awkward solution. Suppose the task is to test the ROM by cycling through its inputs and observing its outputs. The symbolic variable approach starts by putting a variable V at node A and propagating backwards. The test generator would have a rule for propagating back through a counter that says: "to set the output to V, clear the counter, then clock it V times." This yields a plan for controlling A that takes V clock cycles. Repeatedly instantiating this plan with values $1, 2, 3...n$ for V yields a test that takes $1 + 2 + 3...n = (n^2 + n)/2$ clock cycles.

Initializing the counter each time is unnecessary. An expert can easily see that the counter provides a simple way to cycle through the ROM inputs: initialize the counter once, then let it count. PF-TG finds this solution by taking these steps: (i) the ROM component test says to apply an *exhaustive stream* of values to A, i.e., a stream that contains all 256 possible values, (ii) propagating backwards, the counter is asked if it can output an exhaustive stream, (iii) it can because it can generate a *counting stream* which is a kind of exhaustive stream, (iv) to do this, the test generator must initialize the counter and then refrain from initializing it for a period of time. Figure 7.3 shows this solution.[3]

The key to this solution is the language for expressing the problem and the solution. We want to apply a *set* of values to A that have the *property* that they are

[3]Most of the syntax of this program should be self explanatory. Assignment statements (indicated by :=) tell the tester to set the circuit input on the left-hand side to the value on the right, and observation statements (indicated by =) tell the tester to compare the value of the output on the left with the expected value on the right.




```
ARRAY ROMOUTPUTS = ... contents of the ROM ...
CLR := 0
CLK := 0; CLK := 1;
CLR := 1
CLK := 0; CLK := 1;
FOR INDEX12 FROM 0 TO 255 DO
    BEGIN
        OUTPUT = ROMOUTPUTS[i];
        CLK := 0; CLK := 1;
    END
```

Figure 7.3: *Here is an example where propagating a variable yields an awkward solution. The circuit is a portion of the MAC-1. The goal is to test the ROM by running through the possible inputs at A. Propagating a symbolic value back from A yields this solution: clear Counter, then count up to the value of A. This solution takes time proportional to the value of A, and using it repeatedly for the different values of A takes $O(n^2)$ time, where n is the maximum value of A. Repeatedly initializing the counter makes this unnecessarily slow. PF-TG finds a solution that initializes the counter only once by propagating a typed stream back from A. This solution takes $O(n)$ time.*

exhaustive. There are two ideas here: the first is the idea of a stream, i.e., a set of values that are applied in order over time, and the second is the idea of characterizing the desired stream by a property, e.g., exhaustiveness. Classical algorithms (e.g., the D-algorithm) cannot solve the problem above as efficiently because they do not group desired values together as a set. Test generators that propagate variables (e.g., Saturn[singh85]) can tell you how to apply *any one* of a set of values interchangeably, but this yields a less efficient solution for the problem above because the method for applying one value is not combined with the method for applying the rest of the values.

PF-TG currently uses a vocabulary of four properties and twelve stream types some of which are shown in figure 7.4. Properties and stream types have parameters describing, for instance, the values that a counting stream counts *from* and *to*. The parameter *Width* in the figure refers to the width of a circuit node in bits. For example, an ArithmeticCountingStream counts from *Low* to *High* with values *Width* bits wide. A GreyCountingStream is similar except that the count proceeds in grey code order. A PseudoRandomStream stream is a sequence of pseudorandom numbers typically generated by a linear feedback shift register. A ClockStream is simply a single bit rising and falling for *Length* cycles.[4] Figure 7.4 describes the four remaining stream types which are used to test n-bit nodes and bit-parallel components.

Rules for testing a component generally say to supply a stream with certain properties to a component input and observe the component's responses. Rules for propagating through components or for controlling the tester generally say that they can supply certain kinds of streams. If one of the available streams has the desired property, then the test generator can succeed. For instance, a stream with the (Exhaustive 4) property must contain all possible values on a 4-bit node (in any order), i.e., the values 0-15 decimal: an (ArithmeticCountingStream 4 0 15) , for example, has this property. The PairwiseExhaustive property holds if, for any two bit positions, the stream has all four 00,01,10,11 pairwise combinations. DiamondStreams and LogPairwiseStreams have this property, as does any stream that has the exhaustive property. The BitwiseExhaustive property holds if each bit gets set to both 0 and 1, and ZeroOne and Checkerboard streams have this property among others. Finally, a stream has the CoveringSet property if it contains at least the values in *SetOfValues*.

[4]ClockStream assumes a 50% duty cycle. I have not needed a more general definition for the current set of examples, but it is easy to add a parameter for the duty cycle.

(Exhaustive *Width*)
(PairwiseExhaustive *Width*)
(BitwiseExhaustive *Width*)
(CoveringSet *SetOfValues*)

(a) Properties

(ArithmeticCountingStream *Width Low High*)
(GreyCountingStream *Width Low High*)
(PseudoRandomStream *Width Length*)
(ClockStream *Width Length*)
(ZeroOneStream *Width*)
(CheckerboardStream *Width*)
(DiamondStream *Width*)
(LogPairwiseStream *Width*)

(b) Stream Types

1	0 0 0 0 0 0 0 0
2	1 1 1 1 1 1 1 1

(c) (ZeroOneStream 8) covers SA-0 and SA-1 faults in each bit position.

1	0 1 0 1 0 1 0 1
2	1 0 1 0 1 0 1 0

(d) (CheckerboardStream 8) covers stuck bits and bridges between adjacent bit positions.

1	0 0 0 0 0 0 0 0
2	1 0 0 0 0 0 0 0
3	1 1 0 0 0 0 0 0
4	1 1 1 0 0 0 0 0
5	1 1 1 1 0 0 0 0
6	1 1 1 1 1 0 0 0
7	1 1 1 1 1 1 0 0
8	1 1 1 1 1 1 1 0
9	1 1 1 1 1 1 1 1
10	0 1 1 1 1 1 1 1
11	0 0 1 1 1 1 1 1
12	0 0 0 1 1 1 1 1
13	0 0 0 0 1 1 1 1
14	0 0 0 0 0 1 1 1
15	0 0 0 0 0 0 1 1
16	0 0 0 0 0 0 0 1

(e) (DiamondStream 8) covers stuck bits, bridges and all other faults between pairs of bit positions.

1	0 0 0 0 0 0 0 0
2	0 1 0 1 0 1 0 1
3	0 0 1 1 0 0 1 1
4	0 0 0 0 1 1 1 1
5	1 1 1 1 1 1 1 1

(f) (LogPairwiseStream 8) covers stuck bits and bridges between all bit positions

Figure 7.4: *Stream properties (a) and types (b) currently in PF-TG. An ArithmeticCountingStream counts from Low to High with values Width bits wide. A GreyCountingStream is similar except that the count proceeds in grey code order. A PseudoRandomStream stream is a sequence of pseudorandom numbers. A ClockStream is simply a single bit rising and falling for Length cycles. Figures (c) through (f) show the definitions of the less familier streams.*

7.1.5 Reader's Guide

The previous section described the new ideas exhibited in PF-TG. Section 7.2 describes how the test generator works by stepping through a simple example in detail. Unfortunately, to follow the example, we must first introduce some of the test generator's implementation. Most important are three languages for specifying how-to-test rules, constraints on timing and test program fragments. Section 7.2.4 contains the example, and section 7.3 describes the implementation in further detail.

7.2 How PF-TG Works

Figure 7.5 shows the system modules and how they communicate. These modules run as coroutines, but they can be viewed as performing the tasks in figure 7.6 in sequence. This section describes the modules sufficiently to work through a detailed example in section 7.2.4. We concentrate here on the rule engine and the languages for describing constraints and subgoal expansions and test program fragments. After the example, section 7.3 describes how constraints are solved during test generation.

7.2.1 The Rule Engine and Library

The first step, decomposing the test programming problem, is handled by a backward-chaining rule-based system. The system has rules for solving the following kinds of problems: (i) how to test a component, (ii) how to control component outputs, (iii) how to move data through a component, (iv) how to make the tester drive circuit inputs, (v) how to make the tester observe circuit outputs, (vi) how to make a component inactive, (vii) how to initialize a component, (viii) how to move a state machine from one state to another, and (ix) how to take a state machine through a cycle.

The rule engine and rule language underlying PF-TG is Prolog [sterling86], chosen primarily for its simplicity and availability. Figure 7.7 shows a sample propagation rule in Prolog that specifies two ways to control the output of a multiplexor. Each rule handles exactly one goal and lists all ways of decomposing it into subgoals. Rules start with `define-predicate` followed by the name of the goal they handle and a list of clauses. This rule handles `mux-2-control-scalar` goals, which are for controlling the output of a two input multiplexor. `scalar` indicates that the value of the output will be scalar, i.e., a single value rather than a stream. Each clause specifies one way to solve the goal. The first clause (lines 2-4) says "To make `out` be `?value` at `?time`, set the select input (`sel`) to `0` at `?time` and set `in0` to `?value` at `?time`." The

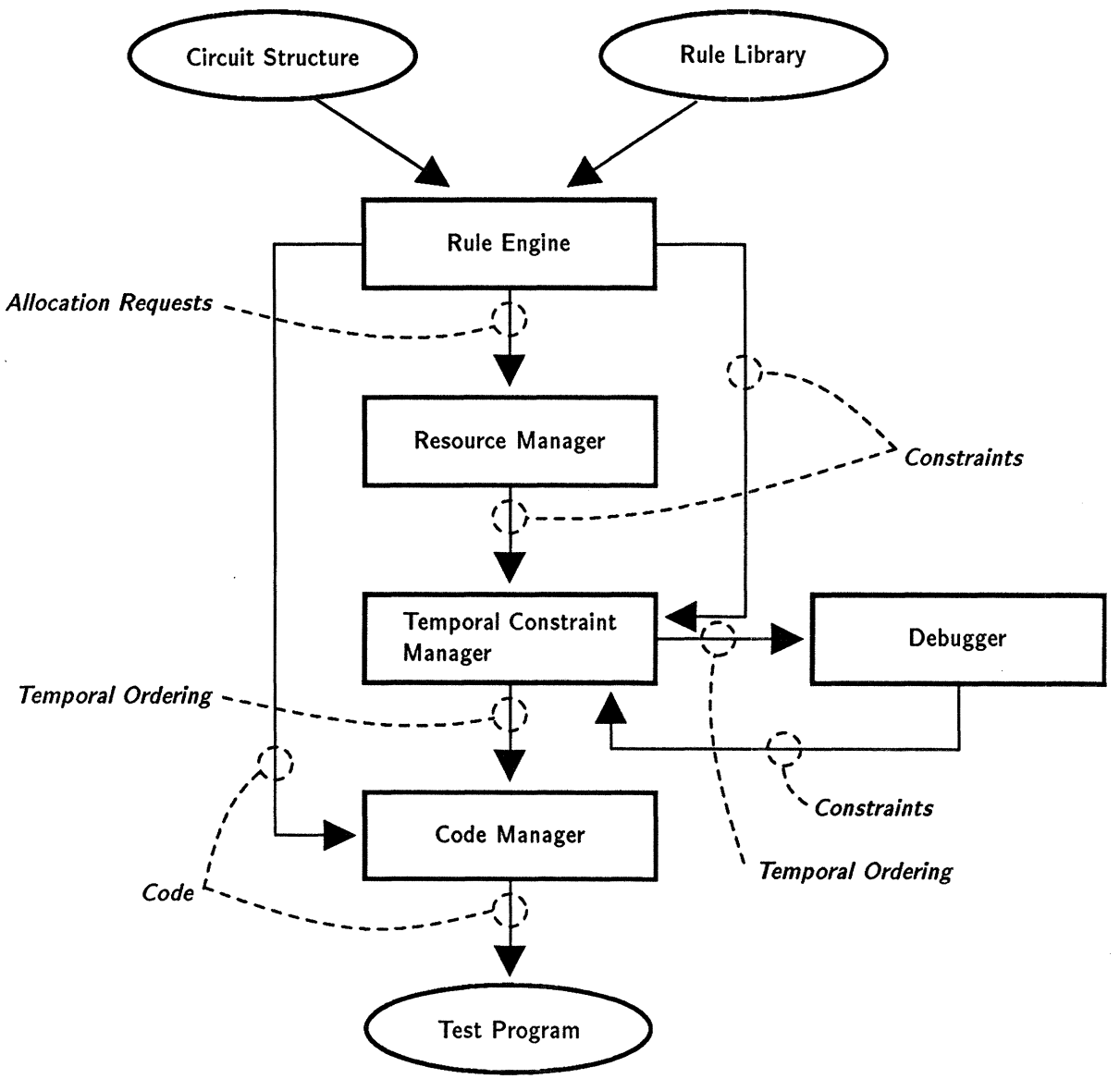


Figure 7.5: *The structure of PF-TG: ovals represent input and output databases, boxes represent active processes (the program modules) and arrows represent messages.*

1. **Problem Decomposition**: How-to-test rules decompose the problem of generating a test into subproblems. Decomposition continues until directly solvable subproblems are reached (e.g., controlling a circuit input or generating tests for a small combinational component) yielding a tree of rule invocations. Rules are stored in the **Rule Library** and are selected and executed by the **Rule Engine**.

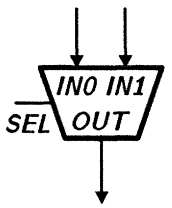
2. **Fragment Collection and Constraint Posting**: In addition to breaking up test generation problems, rules can put program fragments into the output test program. When the engine executes a rule, it copies any program fragments in the rule and passes them to the code manager. Rules also contain constraints controlling how the program fragments fit together. Constraints either control the execution times of program statements or the allocation of tester or circuit resources, and they are passed to the **Time Manager** and to the **Resource Manager** respectively.

3. **Constraint Satisfaction**: The Resource Manager reduces resource constraints to temporal constraints. These plus the temporal constraints sent directly to the Time Manager are reduced to a set of linear inequalities in two variables, where the variables represent execution times. The Time Manager solves these inequalities for integer values.

   The rule language contains constructs for protecting resource assignments made by one subgoal from interference by another subgoal. Protection constraints are maintained by the **Debugger**.

4. **Code Generation**: The Code Manager sorts the program fragments by execution time and assembles code for the tester.

Figure 7.6: *PF-TG performs these steps*




```
1. (define-predicate mux-2-control-scalar
2.   ((mux-2-control-scalar ?component out ?value ?time)
3.    (control-port-scalar  ?component sel 0      ?time)
4.    (control-port-scalar  ?component in0 ?value ?time))
5.   ((mux-2-control-scalar ?component out ?value ?time)
6.    (control-port-scalar  ?component sel 1      ?time)
7.    (control-port-scalar  ?component in1 ?value ?time)))
```

Figure 7.7: *A rule for controlling a MUX's output*

first element of the clause (line 2) specifies the goal to be solved and the remaining elements (lines 2-3) specify subgoals to work on to solve the main goal. As in Prolog, the rule engine tries clauses and subgoals in the order they appear in the text. The second clause (lines 5-7) specifies a similar method using `in1`. Symbols prefixed with "?" indicate variables. The line numbers simplify the explanation and are not a part of the rule.[5]

The particular version of Prolog used[6] has several characteristics that turned out to be important: (i) convenient access to the underlying implementation language (Lisp), (ii) efficient facilities for caching and retrieving solutions to previously solved subproblems, (iii) powerful trace and debugging facilities for examining how goals were solved, and (iv) a freeze goal mechanism. The interface between Prolog and Lisp simplified program development by allowing rules written in Prolog to interface cleanly with the other program modules (e.g., the temporal constraint manager) which were more conveniently implemented directly in Lisp. The caching facilities quicken test generation by recalling solutions to repeated subproblems, e.g., controlling a node. Currently, the user, not the system, decides which goals are cached. The debugging facilities aided program development. Finally the freeze goal mechanism, which is an escape from Prolog's strict depth-first search for deferring goals until sufficient information is available for solving them, simplified some rules. For instance, to move a value through a register file, one must load a register and then read it later. Using this facility, the rule that implements this defers the choice of which address to use until late in the problem where constraints to help choose that address might be available.

[5]Controlling the output of a multiplexor is an extremely simple task, and this Prolog notation is somewhat clumsy here. PG-TG provides a truth table notation to simplify expressing rules like this. For more complex tasks, especially those involving temporal constraints, the Prolog notation quite reasonable.

[6]PF-TG is written in a variant of LM-PROLOG [kahn83] written by Andrew Ressler and modified heavily for this use.

7.2.2 Constraints

PF-TG is a *constraint posting* test generator. Constraint posting is the process of defining an object, a test in this case, by incrementally specifying partial descriptions it must satisfy. For instance, PF-TG constrains the execution times of test program statements until the order in which they should occur becomes clear. The advantage of constraint posting is that choices can be deferred until reasoned decisions can be made. A temporal constraint in one rule can say, for instance, that a component must be initialized before it is used, yet leave for other rules the choice of exactly when these events occur. The reduction in arbitrary choices can increase search efficiency. This is a familiar idea in the AI Planning literature.

PF-TG uses three types of constraints to control how program fragments fit together. **Temporal constraints** control the execution times of program statements, e.g., statement A must execute before statement B. **Structural constraints** control the structure of the test program, e.g., assignment statement C must appear within loop D's body. **Resource constraints** control the allocation of scarce resources to different uses at different times, e.g., circuit node E has a certain value at time T and cannot have any other.

7.2.2.1 Temporal Constraints

Each planned event in the circuit or statement in the test program is associated with a temporal variable; PF-TG controls when these events and statements occur by constraining the temporal variables. A Time Manager [allen-cacm, joyce83, shirley85, valdes86, dean87] records and satisfies the constraints by finding a consistent set of integer variable assignments. The integers represent the clock cycle during which the event or statement will occur when the test is executed.[7]

The Time Manager handles `t`$\leq$`-offset`, $=$, $\neq$, $<$, $\leq$, $>$, $\geq$, and `t=-offset` constraints between pairs of variables plus `and` and `or` connectives between expressions. For convenience, it also provides a macro language for expressing more complex relations such as `disjoint-intervals` and `overlapping-intervals`. The `(t=-offset a b c)` constraint stands for `b = a + c`. This constraint is used in the example below. The other constraints are explained in section 7.3.1

[7]Modeling time at the level of granularity of clock cycles restricts the program to circuits with a single, continuously running clock. PF-TG has a second rule set that models time at a finer level of granularity, thus allowing asynchronous control of every circuit input. This document describes only the first rule set, which produces much simpler constraints if it is applicable to the circuit under test.

7.2.2.2 Structural Constraints

The time manager also handles precedence constraints on the structure of the test program. Some program statements do not take time to execute, but the order in which they appear in the program is important. For instance, variable declarations do not generally involve execution time, but, in some languages, they must appear before all executable code lying in the same begin-end block. Structural constraints control the textual order in which statements appear in the final test program.

Structural constraints are implemented using the same mechanisms used by temporal constraints. Each variable has two values: one along a temporal timeline and another along a structural timeline. We refer to a "temporal variable" when we are are interested in its temporal value and similarly with a "structural variable."

The "timeline" for structural constraints is separate from the temporal timeline, but the two are related because test programs execute in structural order: a statement that executes before another must have appeared earlier in the program, and a statement that appears earlier cannot execute later.[8] If t< is temporal precedence and s< is structural precedence, then

$$A\mathrm{t} < B \Rightarrow A\mathrm{s} < B \quad \text{and} \quad A\mathrm{s} < B \Rightarrow A\mathrm{t} \leq B$$

7.2.2.3 Resource Constraints

The Resource Manager detects resource assignments that might potentially conflict (e.g., distinct values assigned to a single node during propagation) and creates temporal constraints to make sure they do not. There are two kinds of resources: node resources involving activity inside the circuit and tester resources involving activity in the tester.

Node Resources

Circuit nodes are resources because they can only have one value at a time. Suppose, for example, that the rule engine assigns node A the value 1 at time T1 (**A=1@T1**) to achieve one goal and **A=0@T2** to achieve another goal. Since $0 \neq 1$, the times must be different, i.e., ($\neq$ **T1 T2**). In the general case, either the node values are the same or the times are different. The time manager enforces this by first trying to prove that the two values are equal using a simple equality theory (i.e., a

[8]Temporal order implies structural order except for loops and conditionals, which are handled specially. Loop bodies also execute in structural order.

set of Prolog rules for proving algebraic expressions equal). If this proof fails, then it asserts a $\neq$ constraint between the times.

Enforcing resource constraints with temporal disjunctions can be expensive, since it can yield $n(n-1)$ disjunctions for n assignments to the same node. However, for moderately sequential circuits, there are usually few assignments to a single node within a single test generation problem. For the examples in this chapter, the average number of assignments per node is roughly 1.5, and the maximum number is 5.

Two mechanisms tend to keep this number down. The first is partitioning the problem into independent subproblems called test phases.[9] If two times belong to different test phases, then they effectively lie on different timelines and no temporal disjunction is needed to keep them apart. The second mechanism is stream propagation: resource constraints are not needed nor are they enforced between the individual elements of the stream.

Tester Resources

PF-TG manages the following tester resources: (i) the special-purpose stream generators, (ii) the execution stack of the main processor and (iii) the driver-sensors. A special-purpose stream generator is allocated to tasks during non-overlapping intervals. The execution stack is used by program statements that create variable binding environments: `let`, `loop` and `multi-stream`. Proper use of the stack resource requires that binding environments be nested, that is, every binding statement that starts during the execution of binding statement A must also finish during binding statement A. Thus every pair of binding statements (A, B) must stand in this relationship:

(or (disjoint-intervals A B)
    (interval-contains A B) [10]
    (interval-contains B A))

By special dispensation, the code generator can merge two loops if they involve the same number of iterations. Consequently, pairs of `loop` and `multi-stream` statements stand in this relationship instead of the previous one:

[9] Test phases are similar to reference intervals [allen-cacm] except there is no hierarchy of them.

[10] The `interval-contains` constraint means one interval must be *completely* within the other, i.e., their start times are distinct and their finish times, distinct. The start and finish times are distinct because two statements cannot manipulate the stack at the same time, and we model binding statements as pushing a new binding onto the stack before doing anything else and popping the binding as their final action.

(or (equal-intervals *loop-a loop-b*)
(disjoint-intervals *loop-a loop-b*)
(interval-contains *loop-a loop-b*)
(interval-contains *loop-b loop-a*))

Finally, there are a limited number of driver-sensors, each of which can be allocated to only one pin (circuit input or output) within a test phase. A driver-sensor is the electronics package that alternately creates the voltages for controlling a circuit node and measures the voltage on the node. A pin is a wire that makes physical contact with the circuit. Pins are cheaper than driver-sensors, and there are often more of them. PF-TG assumes they are connected via a few-to-many crossbar switch. The driver-sensor resource is implemented by incrementing a counter every time a signal propagating to an input or output is handled by a normal control or observe test language statement (as opposed to one of the stream generators). If this counter hits a pre-specified maximum, that signal propagation fails and the rule engine backtracks, decrementing the counter as appropriate.

7.2.3 The Code Manager and The Test Language

The code manager has three jobs. First, it prepares code fragments given to it by the rule engine for inclusion in the final test program. A code fragment can imply additional subgoals and constraints that are sent to the rule engine and to the time manager respectively. For instance, writing a program statement that assigns a value to an internal circuit node is shorthand for the goal of propagating that value back to primary inputs: this goal can either be explicit as a subgoal in a rule or implicit as an assignment statement in a rule's program fragment. Second, when the rule engine and time manager quiesce, the code manager uses the total order on temporal variables to merge all fragments together to produce a program in PF-TG's test language. Finally, the code manager converts the test program into the language of the target tester.

7.2.3.1 The Test Programming Language

PF-TG's language for expressing test program fragments is a simple procedural language with two novel aspects: (i) it contains unexecuted, advisory declarations for controlling how the code manager combines program fragments and (ii) the language features partially overlap those of the rule language and provide an alternative and more convenient notation for specifying certain kinds of goal expansions and temporal constraints. This section focuses on the novel aspects of the language.

A test program fragment must specify two things:

1. The fragment must specify what to do. Fragments say what the combination of tester software and hardware must do to solve a particular testing goal (usually part of a larger problem).

2. The fragment must specify what *not* to do. Since fragments can interfere with each other when merged, fragments must tell PF-TG how they can be merged without being broken. These restrictions are expressed as temporal, structural and resource constraints.

To see the importance of specifying what not to do, consider the following example. Suppose that fragment A loads register R in one statement and reads it in the next, and fragment B assigns R a different value. If B is merged in between the statements in A, then A is unlikely to work as expected. Between the load and the read, the register's contents must be protected: the language includes statements for expressing this.

PF-TG's test programming language has a lisp-like syntax and includes constructs for controlling and observing circuit nodes, iteration, conditional branching several other advisory statements. Figure 7.8 contains a full list of these statements. In the text, I introduce statements as needed to describe the ideas behind the language.

Figure 7.10 shows a sample code fragment extracted from the rule for testing adders. The `declare-fragment` statement in the rule language sends a program fragment to the code manager for inclusion in the final test program. The `multi-stream` iteration construct accepts a set of (variable stream) pairs and executes its body with the variables bound to successive stream elements. The streams are stepped in parallel and must be the same length.

The `equal-instants` advisory declaration specifies that the statements in its body must be executed simultaneously. The first `control` statement says to assign node `?in1` the value `in1-data`; the remaining statements are analogous. All prolog variables appearing in a program fragment are bound to circuit nodes or constants by the rule engine before it passes the fragment to the code manager. For instance, `?in1` is bound to a circuit node earlier in the complete version of this rule.

7.2.3.2 Processing a Code Fragment: Implied Constraints and Goals

The code manager breaks up code fragments into individual statements, creates temporal variables to represent their execution times, and constrains these variables ac-

Statement	*Description*
Manipulating Nodes	
`(CONTROL-SCALAR node value time)`	Assign a single value to a node and justify it.
`(CONTROL-STREAM node value time)`	Assign and justify a stream.
`(OBSERVE-SCALAR node value time)`	Sensitize a path from a node to an output.
`(OBSERVE-STREAM node value time)`	Sensitize a path for a stream.
Iteration	
`(LOOP (variable from to) &body body)`	Repeat the body. All subgoals caused by body finish before the next iteration.
`(MULTI-STREAM stream-decls &body body)`	Declare streams to be used in the body.
Temporal Declarations	
`(GROUP &body body)`	Textually like a BEGIN-END block but places no time constraints on the inferior statements.
`(BLOCK &body body)`	Inferior statements must execute during the block, but no constraints between inferiors.
`(LOOSE-SEQUENCE &body body)`	Inferior statements execute in sequence but with arbitrary gaps (allowing statements from other fragments to come between).
`(TIGHT-SEQUENCE &body body)`	Inferiors execute in sequence with no gaps.
`(EQUAL-INSTANTS &body body)`	Inferiors have the same execution-time.
Other Declarations	
`(TEST-PHASE declarations &body body)`	The body is an independent subproblem.
`(LET declarations &body body)`	Bind some variables for use in the body.
`(DECLARATIONS code)`	An escape for including arbitrary code in the final test program. Used mainly to declare arrays.
`(WITH-SYNCH &body body)`	All streams within the body are synchronized.
Resource Protection	
`(PROTECT nodes &body body)`	Protect the nodes while body executes.
`(PROTECT-EXPLICIT nodes &body body)`	Only lexically visible statements within body can effect the nodes while the body is executing.
Sequencing	
`(IF condition then else)`	A conditional branch. Each branch must be a test-phase.
Syntactic Sugar (macros)	
`(CONTROL node value time)`	Expands into control-stream if inside a multi-stream, else expands into control-scalar.
`(OBSERVE node value time)`	The same thing for observe-stream and observe-scalar.

Figure 7.8: *All of the statements in PF-TG's test programming language.*

```
(declare-fragment
  (multi-stream ((in1-data ADDEND-STREAM-16-1)
                 (in2-data ADDEND-STREAM-16-2)
                 (out-data SUM-STREAM-16))
    (equal-instants
      (control ?in1 in1-data)
      (control ?in2 in2-data)
      (control ?op ALU-ADD)
      (observe ?out out-data))))
```

Figure 7.9: *A sample test program fragment (part of an adder test).*

```
(declare-stream 'ADDEND-STREAM-16-1 :value '(0 43690 1 1 65534 65535 21845 65535))
(declare-stream 'ADDEND-STREAM-16-2 :value '(0 43690 65534 65535 1 1 21845 65535))
(declare-stream 'SUM-STREAM-16      :value '(0 21844 65535 0 65535 0 43690 65534))
```

Figure 7.10: *Stream declarations for the fragment above.*

cording to the statement types and where they appeared in the original fragment. This network of constraints determines how the fragments combine to form the final test program.

Figure 7.11 shows the temporal variables and constraints associated with various statement types. Statements that execute once and over a single clock cycle (e.g., controlling a circuit node) are associated with a single temporal variable called `execution-time`. Statements that group other statements together, like begin-end blocks in Pascal, execute over an interval of time and are associated with two temporal variables called `start` and `finish`.

Iteration statements are associated with two intervals: one interval from the start of the first iteration to the end of the last iteration, and a second interval that spans one prototypical execution of the loop body.

Stream statements are a special kind of loop that apply (or observe) a stream of values to a node. Stream statements are associated with five variables: the four variables corresponding to a loop (the actions associated with one value must be repeated with each element of the stream) and a fifth one, `execution-time`, corresponding to a prototypical time within the loop. For streams that control a node, the stream must set up the node value before and hold it past `execution-time`. For streams that observe a node, the node value is sampled at `execution-time`.

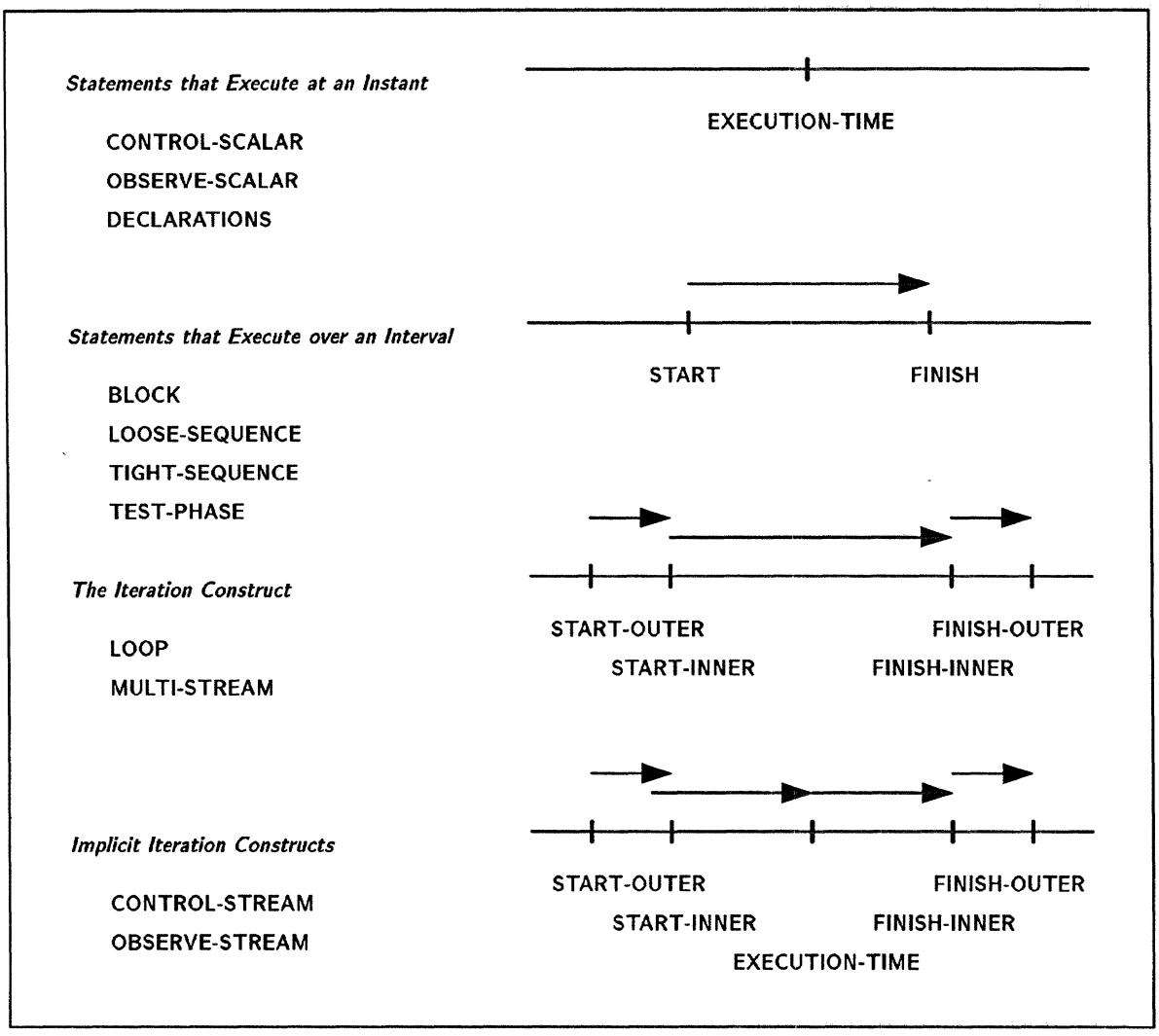


Figure 7.11: *Temporal variables and constraints associated with various statement types in PF-TG's test programming language.*

After breaking up the fragment into individual statements and creating and constraining temporal variables according to the statement types, the time manager constrains the variables according to the textual structure of the fragment and the semantics of any advisory statements. In establishing constraints between statements, textual order is shorthand for temporal order and textual inclusion is shorthand for temporal inclusion. For instance, if a `loose-sequence` statement surrounds several statements, then the `loose-sequence`'s start time must come before the first time of any contained statement and its end time must come after the last time of any contained statement.

Most of the statements that execute over an interval are advisory, e.g., `loose-sequence`, `tight-sequence` and `test-phase`. `loose-sequence` constrains its inferior statements to execute in order, but allows arbitrary intervals before, after and between the statements (see figure 7.12). `tight-sequence`, on the other hand, forces the execution times or intervals of the inferior statements to meet. `test-phase` is an implicit `loose-sequence` and has other advisory purposes (declaring isolatable subproblems), and `block` puts no constraints on its inferiors.

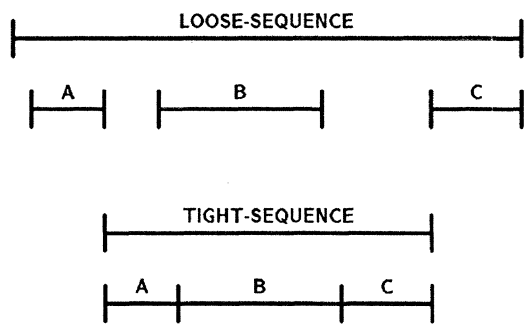


Figure 7.12: `loose-sequence` *allows arbitrary intervals between inferior statements (e.g., A, B and C), while* `tight-sequence` *does not.*

This shorthand notation for expressing temporal constraints based on the structure of the test program code has turned out to be quite convenient, and many component tests and propagation rules can be written with no explicit temporal constraints.

This concludes the description of the basic mechanisms in PF-TG. The next section shows how these mechanisms work together to solve test generation problems by stepping through a simple example in detail.

7.2.4 A Detailed Example

Figure 7.13 shows a very simple digital filter implemented by a delay line and an adder. Figures 7.14 through 7.20 work through the task of embedding a test for the adder.

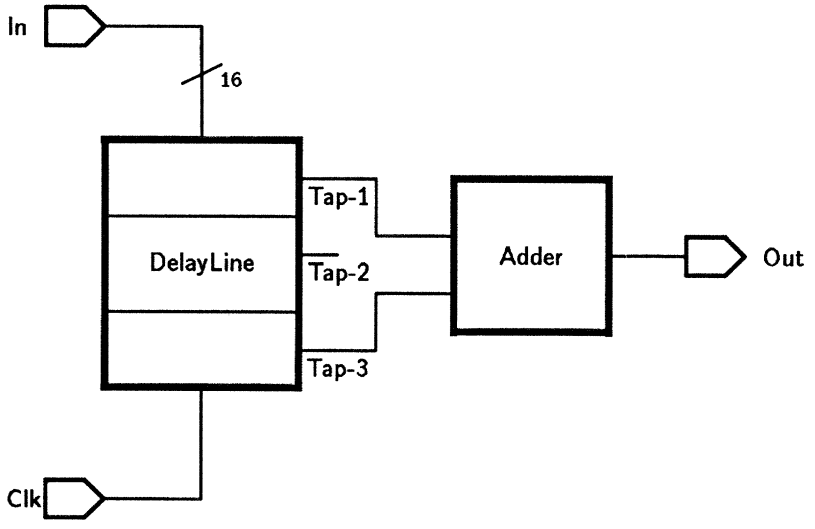


Figure 7.13: *A simple digital filter*

```
    ;;; The CARRY-CHAIN-ADDER-TEST component test
1.  (define-predicate CARRY-CHAIN-ADDER-TEST
2.    ((carry-chain-adder-test ?adder)
3.     (component-port ?adder in1 ?in1)
4.     (component-port ?adder in2 ?in2)
5.     (component-port ?adder out ?out)
6.     (declare-fragment
7.       (test-phase (:COMPONENT ?adder :FACILITY PLUS)
8.         (multi-stream ((in1-data ADDEND-STREAM-16-1)
9.                        (in2-data ADDEND-STREAM-16-2)
10.                       (out-data SUM-STREAM-16))
11.          (equal-instants
12.            (control ?in1 in1-data)
13.            (control ?in2 in2-data)
14.            (observe ?out out-data)))))))
```

Figure 7.14: *The ADDER component test.*

1 Start: type `(generate-test ADDER)` at the Prolog interpreter. The system looks up what type of component ADDER is and runs the appropriate how-to-test rule: `carry-chain-adder-test`.

2 `carry-chain-adder-test` first looks up the node names of its inputs and outputs using three `component-port` subgoals (lines 3-5). This binds `?in1`, `?in2` and `?out` to `Tap-1`, `Tap-3` and `Out` respectively. The rule then declares a program fragment containing the basic structure of the test. The `test-phase` form documents this fragment and produces a comment in the final test program. The `multi-stream` form declares three streams to step in parallel.

The system creates temporal variables for the statements in this fragment and posts constraints corresponding to the statement types and nesting structure. A temporal variable, here called `test-time`, is created to represent the execution time of the three statements inside the `equal-instants` declaration (lines 12-14). The times and constraints in figure 7.19.c, line 1 come from the `multi-stream` statement. Temporal constraint 2 comes from the nesting structure of the loop and the `equal-instants` statement. Structural constraints 1 and 2 arise similarly (figure 7.19.d).

Processing the fragment spawns three subgoals: one to set `Tap-1 = in1-data @ test-time`, another to set `Tap-3 = in2-data @ test-time`, and a third to observe `out-data = out-data @ test-time`. The system starts with the first subgoal.

Figure 7.15: *A Detailed Example*

```
   ;;; The propagation rule for controlling DELAY-LINE output taps
1. (define-propagation-rule DELAY-LINE control-stream
2.   ((delay-line-control-stream ?component tap-1 ?value ?output-time)
3.    (new-time ?input-time)
4.    (t=-offset ?input-time ?output-time 1)
5.    (control-port-stream ?component in ?value ?input-time))
6.   ((delay-line-control-stream ?component tap-2 ?value ?output-time)
7.    (new-time ?input-time)
8.    (t=-offset ?input-time ?output-time 2)
9.    (control-port-stream ?component in ?value ?input-time))
10.  ((delay-line-control-stream ?component tap-3 ?value ?output-time)
11.   (new-time ?input-time)
12.   (t=-offset ?input-time ?output-time 3)
13.   (control-port-stream ?component in ?value ?input-time)))
```

Figure 7.16: *The DELAY-LINE propagation rule.*

3 The system responds to the `Tap-1 = in1-data @ test-time` goal by recording the time and value, then looking up the component that drives this node (`DelayLine`) in the schematic. Since the value involved is a stream and the goal involves control, the system calls the `DELAY-LINE control-stream` goal, passing it the component, the port being controlled (`Tap-1`), the value and the time.

4 The DELAY-LINE propagation rule has three clauses, one for each output tap; these clauses are identical except for the constants in the `t=-offset` subgoals. The first clause (lines 2-5) responds to the goal of controlling `Tap-1`. (The other clauses cannot respond because the third element of the goal – the port name – does not match). The first clause binds `?input-time` to the new temporal variable `time-1` and constrains `time-1 + 1 = test-time`. It then spawns the subgoal of controlling its input port. The system responds to this goal by looking up the node driving this port, recording the value (`in1-data`) and time (`time-1`) of the assignment and looking for a component to drive this node.

5 The `tester-control-stream` rule, part of the tester model, responds to this goal. It declares a fragment that contains one assignment statement: `In := in1-data` to occur at `time-1` (see fragment 1 in figure 7.19 – `in1-data` is replaced with the array reference during code generation).

6 The goal of controlling `Tap-2` is handled similarly. `DelayLine` is again the component that drives the node, so the system calls `DELAY-LINE control-stream` with the appropriate arguments. The third clause (lines 10-13) responds this time, creates the new temporal variable `time-2`, and constrains it to occur 3 clock cycles earlier than `test-time`. As before, it uses a `control-port-stream` subgoal to control the input.

Figure 7.17: *A Detailed Example (continued)*

7 When the system responds this time, it notices that the node `In` has been assigned a new value (`in2-data`) that is distinct from a previously assigned value (`in1-data`) so it posts a temporal constraint to keep the assignment times distinct too: `time-1` $\neq$ `time-2`. Again, the system passes the goal of controlling the circuit input to the tester model which responds with another program fragment (fragment 2 in figure 7.19.a).

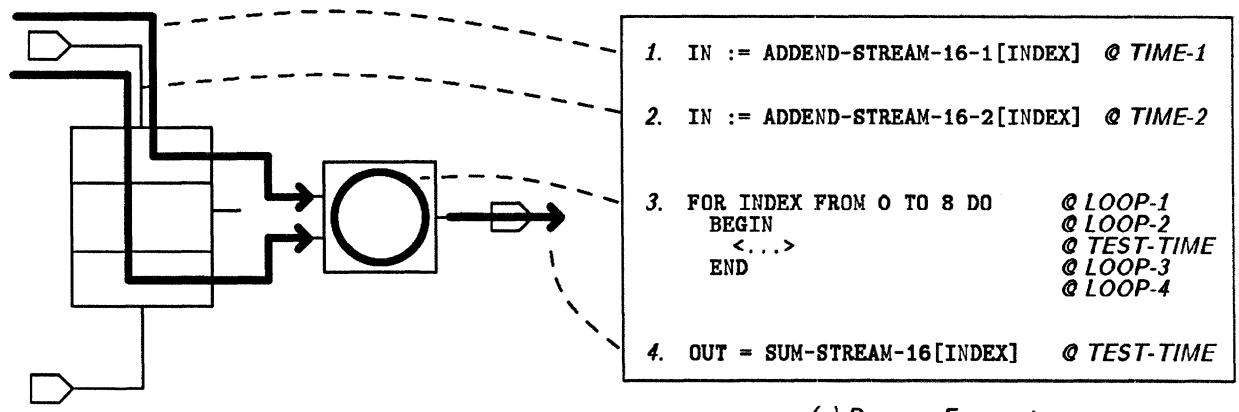
8 The system responds to the goal of observing the adder's output by recording the value and time and passing the problem to the tester model. The `tester-observe-stream` rule responds by declaring a program fragment that observes the stream of values (fragment 4 in figure 7.19.a).

9 At this point, all subgoals have been solved. The Time Manager collects and solves all temporal and structural constraints. All of the constraints are explained above except those connecting `time-1` and `time-2` to the loop, i.e., temporal constraints 5 and 6 and structural constraints 3 and 4. These constraints are generated by `tester-control-stream` by special dispensation. In the general case, `tester-control-stream` creates a new loop to supply an input stream, and the new loop is related to other loops in the test because they all use the program stack in the tester's main processor. In this example, sharing the program stack places sufficient constraint on the new loop that it must be merged with the `multi-stream` loop in the original fragment (figure 7.14). This situation is common, can be recognized early to optimize away the cost of using the stack resource: the system passes the original loop times along during propagation as annotations to the stream variables (e.g., `in1-data`). Using this information, `tester-control-stream` posts the temporal and structural constraints shown.

10 One solution for the constraints is shown in figure 7.19.f: temporal values for each variables appear to the left of the commas and structural values appear to the right. They are used together in dictionary order to provide a total order on the statements in the program. The Code Manager sorts the program statements together and produces the solution of figure 7.20.

11 This description is simplified in two ways: it suggests that the constraints are solved after the rule engine is finished, while this is actually interleaved with rule firing. This description also ignores the times associated with the `test-phase` statement (these times bracket the whole solution) and the structural variables associated with the array declarations.

Figure 7.18: *A Detailed Example (continued)*



(a) Program Fragments

```
1. ARRAY ADDEND-STREAM-16-1 = [0, 43690, 1, 1, 65534, 65535, 21845, 65535];
2. ARRAY ADDEND-STREAM-16-2 = [0, 43690, 65534, 65535, 1, 1, 21845, 65535];
3. ARRAY SUM-STREAM-16 = [0, 21844, 65535, 0, 65535, 0, 43690, 65534];
```

(b) Declarations

```
1. LOOP-1 <= LOOP-2 <= LOOP-3 <= LOOP-4
2. LOOP-2 <= TEST-TIME <= LOOP-3
3. TIME-1 + 1 = TEST-TIME
4. TIME-2 + 3 = TEST-TIME
5. LOOP-2 <= TIME-1 <= LOOP-3
6. LOOP-2 <= TIME-2 <= LOOP-3
```

(c) Temporal Constraints

```
1. LOOP-1 < LOOP-2 < LOOP-3 < LOOP-4
2. LOOP-2 < TEST-TIME < LOOP-3
3. LOOP-2 < TIME-1 < LOOP-3
4. LOOP-2 < TIME-2 < LOOP-3
```

(d) Structural Constraints

```
1. TIME-1 ≠ TIME-2
```

(e) Resource Constraints

```
LOOP-1 = 0,0      TIME-2 = 0,2
LOOP-2 = 0,1      TIME-1 = 2,2
LOOP-3 = 3,3      TEST-TIME = 3,2
LOOP-4 = 3,4
```

(f) One Solution (format is TemporalOrder, StructuralOrder)

Figure 7.19: *All pieces of the solution. The solution itself appears in figure 7.20.*

```
BEGIN
    ** TEST-PHASE (:COMPONENT ADDER :FACILITY PLUS)
    CLK := 0; CLK := 1;
    ** DECLARE-STREAMS (ADDEND-STREAM-16-1 ADDEND-STREAM-16-2 SUM-STREAM-16)
    BEGIN
        ARRAY ADDEND-STREAM-16-1 = [0, 43690, 1, 1, 65534, 65535, 21845, 65535];
        ARRAY ADDEND-STREAM-16-2 = [0, 43690, 65534, 65535, 1, 1, 21845, 65535];
        ARRAY SUM-STREAM-16 = [0, 21844, 65535, 0, 65535, 0, 43690, 65534];
        FOR INDEX1 FROM 0 TO 8 DO
            BEGIN
                IN := ADDEND-STREAM-16-2[INDEX1]
                CLK := 0; CLK := 1; CLK := 0; CLK := 1;
                IN := ADDEND-STREAM-16-1[INDEX1]
                CLK := 0; CLK := 1;
                OUT = SUM-STREAM-16[INDEX1]
                CLK := 0; CLK := 1;
            END
    END
END
```

Figure 7.20: *The solution as a program: the adder test embedded in the digital filter*

7.3 Further Details of the Mechanism

This section covers the time manager implementation, protection constraints and the debugger.

7.3.1 The Time Manager

The Time Manager maintains and incrementally solves temporal and structural constraints. It handles `t≤-offset`, $=$, $\neq$, $<$, $\leq$, $>$, $\geq$ constraints, plus conjunctive constraints (e.g., a is before b and b is before c) and disjunctive constraints (e.g., either a occurs before b or b occurs before a). The algorithm is both sound (all solutions satisfy the posted constraints) and complete in one sense (all ways to choose the disjuncts are eventually tried), however it is not complete in the sense of enumerating all solutions. This distinction is important and is discussed in the section 7.3.3 describing the debugger.

The Time Manager is incremental. As the Rule Engine creates a test program, it posts constraints connecting sets of temporal and structural variables. The Time Manager maintains a consistent set of variable assignments, and when the rule engine changes the constraints, the Time Manager updates the assignments. If it cannot do so because the constraints are unsatisfiable, then it forces the Rule Engine to backtrack.

The Time Manager accepts three kinds of messages from the other modules: (i) requests for the creation of a new temporal variables, (ii) specifications of new constraints on the values of existing variables, (iii) finish messages requesting integer values for the temporal variables. The Time Manager responds to these messages by updating its database and ensuring that the new database is globally consistent. If the database is consistent, the time manager accepts the message and returns success to the sending module. If the database is no longer consistent, the time manager rejects the message and restores the database to its previous state. The Time Manager therefore maintains the following invariant: the database is consistent between messages.

Most messages come from the Rule Engine and are generated directly by subgoal statements in the rules. If the Time Manager accepts the message, then the subgoal succeeds and the rule engine continues. If it rejects the message, the subgoal fails and the Rule Engine backtracks. The other modules send messages to the Time Manager in response to messages sent to them from the Rule Engine. In the event of failure, these modules pass the failure back to the Rule Engine.

7.3.1.1 List of Messages

The time manager accepts seven primitive messages and a large and extensible set of messages defined in terms of the primitives. A NEW-TIME message causes the time manager to create and return a new temporal variable. This message always succeeds. A (new-time ?time) subgoal causes the rule engine to send this message and binds ?time to the new temporal variable.

The TEMPORAL-OFFSET message, e.g., (t≤-offset ?time1 ?time2 ?offset) constrains the values of two temporal variables such that ?time2 ≥ ?time1 + ?offset. ?offset must be an integer. This message succeeds if the new constraint is consistent with the constraints previously given to the time manager.

The AND message allows the time manager to accept a set of messages at once and process them serially. The OR message allows the time manager to accept disjunctions, i.e., a set of alternative constraints only one of which need be enforced. All other constraints are defined in terms of AND, OR and t≤-offset. See, for instance, the definition of (≠ ?a ?b) in figure 7.21. Using these three primitives, the Time Manager provides a vocabulary for describing relations between intervals, e.g., t-disjoint-intervals and t-overlapping-intervals, and between points and intervals, e.g., t-point-outside-interval.

The VALUE message accepts a temporal variable and returns its current integer assignment. The (temporal-value ?variable ?value) subgoal sends this message. The LISTVARS message returns a list of all temporal variables so that other modules need not maintain their own.

The DEALLOCATE-VARIABLE message and the DEALLOCATE-CONSTRAINT message remove variables and constraints respectively from the temporal database. These messages are called internally when the rule engine backtracks and do not appear explicitly in any rules. Deallocating constraints is described with the time manager implementation below.

7.3.1.2 Time Manager Implementation

This section gives a brief description of the time manager implementation. The algorithm was selected for good average-case performance in medium-sized problems (i.e., up to 100 temporal variables) for systems of constraints that are not tightly connected. The most important goal was to reduce the cost of adding a new constraint and checking whether the augmented set of constraints was still consistent. Reducing this cost makes it practical to run the time manager incrementally as the rule engine

Constraint	*Definition*
(t≤-offset ?a ?b ?o)	?b ≥ ?a + ?o
(≥ ?a ?b)	(t≤-offset ?a ?b 0)
(≤ ?a ?b)	(t≤-offset ?b ?a 0)
(= ?a ?b)	(and (≥ ?a ?b) (≤ ?a ?b))
(> ?a ?b)	(t≤-offset ?a ?b 1)
(< ?a ?b)	(t≤-offset ?b ?a 1)
(≠ ?a ?b)	(or (> ?a ?b) (< ?a ?b))
(interval (?a ?b))	(≤ ?a ?b)
(disjoint-intervals (?a ?b) (?c ?d))	(and (interval (?a ?b)) (interval (?c ?d)) (or (< ?b ?c) (< ?d ?a)))
⋮	⋮

Figure 7.21: *Definitions of representative temporal constraints in terms of AND, OR and* t≤-offset

expands goals, using Time Manager failures to prune inconsistent branches of the search tree.

To meet these goals, I chose an algorithm with particularly simple procedures for adding and removing constraints. The algorithm does not have the best known complexity measure for solving a single set of constraints, but it does have low overhead when changing constraints. Low overhead is important here because the constraints are changed repeatedly by the rule engine as it searches for paths through the circuit. Moreover, further tuning of the algorithm is currently unwarranted. On the examples shown in this chapter, less than 20% of PF-TG's time is spent in the time manager (almost 80% is spent in the rule engine with negligible time spent in the code manager).

Time management is in essence a simple linear programming problem: (`t≤-offset` x_i x_j a_{ij}) constraints are linear inequalities in two variables:

$$x_j - x_i \leq a_{ij}$$

where $x_i, x_j \in$ TemporalVariables and $a_{ij} \in$ Integers. Sets of `t≤-offset` constraints are solvable in numerous ways; the time manager uses a simple graph traversal algorithm. Nodes in the graph represent temporal variables and are associated with an integer "time" along the timeline. Directed arcs represent `t≤-offset` temporal constraints and have an integer weight corresponding to the a_{ij} term in the inequality above. Conjunctions of `t≤-offset` constraints are simply represented as sets of arcs. We discuss disjunctions later.

Next, I describe how the time manager's primitive messages are implemented by updating the graph representation. In this discussion, I refer to temporal variables and nodes interchangeably and temporal constraints and arcs interchangeably. Before each message arrives, the graph representing the previously specified constraints is satisfiable and that the set of integers assigned to the graph nodes constitutes a solution. This is the situation between messages to the time manager.

There are four primitive operations: adding and deleting a node (a temporal variable) and adding and deleting a weighted arc (a constraint). Adding a new node is trivial. Since it is new, it must be unconstrained, so give it the value 0. This operation happens in constant time.

Deleting a node is trivial if it is unconnected and not allowed if it is connected. Thus modules that use the time manager must obey a stack convention for allocating nodes and arcs, i.e., they must allocate a node before connecting it and remove the arcs before removing the node. The rule engine obeys this convention since it uses depth-first search.

Deleting an arc is also trivial. The variable assignments are already consistent and removing a constraint cannot make them inconsistent, so the time manager simply removes the arc from the graph and leaves the node values as they are.

Adding a new arc is the difficult case, since the set of constraints may become inconsistent or the variable assignments may have to be changed. A set of constraints is inconsistent if it implies that a variable is greater than itself, i.e., if there is a positive weight cycle in the graph. After adding a new arc, the time manager searches for a positive weight cycle involving the new arc. It does this using depth-first search from the head of the new arc, while, at the same time, updating the variable assignments. Figure 7.22 shows the update procedure.

```
1.   A is the set of arcs.
2.   If arc (ij) ∈ A then a_{i,j} is the weight of that arc.
3.   The . operator pushes a new value onto the front of a list
     and nil is the empty list

To add a new arc (ab) with weight w
1.   A ← A ∪ (ab)
2.   update(b, value(a)+w, nil)

1.   PROCEDURE update (node, newvalue, stack)
2.   IF node ϵ stack THEN restore old values and fail;   /* found a cycle */
3.   IF value(node) < new-value THEN                    /* update outgoing arcs */
4.        FOREACH (node next) ϵ e DO
5.             update (next, value(node) + a_{node,next}, node.stack);
```

Figure 7.22: *The procedure for adding an arc*

Depth-first search takes $O(A)$ time in the worst case if arcs are marked so they are traversed at most once. The update procedure above behaves similarly with the partial solution acting as the mark: when value(node) $\geq$ new-value, the program need not search beyond the node. If this condition never occurs, the worst case time for the procedure above is $O(2^A)$. In practice, however, the update algorithm runs quite quickly: the average number of arc traversals per new arc is 2.5 for the examples in this chapter and in appendix A. This indicates that, for these examples, the constraints are sparse.[11]

[11]Implementing the update algorithm using best-first search can potentially reduce arc traversals,

The time manager handles a new disjunctive constraint using several steps. First, it reduces the new disjunct to conjunctive normal form and adds it to the set of disjuncts that must be enforced. Disjuncts are maintained in the order they were presented to the time manager. The time manager then makes a choice from each of these disjuncts in order, starting from the first disjunct and backtracking when it makes a choice that is inconsistent with earlier choices.

Maintaining a partial solution (i.e., the consistent node assignments) is important to holding down the cost of making these choices. The key idea is this heuristic: first try choices that are already satisfied in the current partial solution. Using the partial solution, choices can be checked for consistency in time proportional to the number of conjuncts in the choice. Adding a set of arcs for a choice (a conjunction) that is already satisfied in the partial solution requires no propagation.

The time manager maintains a **current choice** for each disjunct, which is always reflected in the current partial solution. When a new disjunct is added, it is useful to think of the time manager making a choice for each disjunct in order. The program, however, has already made consistent choices for all but the new disjunct. Thus old disjuncts can be skipped, since by the heuristic above, the program would simply make the same choices again. If a new choice can be made that is already satisfied, then the program marks that as the new disjunct's current choice, adds the new arcs to the graph (to make sure the choice continues to be reflected in the partial solution) and returns success. If none of the new choices is already satisfied, then they are tried one at a time to see if any are consistent. If one is, the program marks the current choice, adds the arcs and returns success. If none are, then the program backtracks to the most recent disjunct and tries another choice for it. If it cannot find another consistent choice for that disjunct, then the program continues backtracking. The program returns failure (rejecting the new disjunct) if it backtracks past the first choice.

Backtracking occurs when the set of disjunctive choices is inconsistent or when all sets of disjunctive choices have been tried. The time manager detects when the choices have been exhausted by checking choices against a list of nogoods, i.e., groups of choices that have previously been found inconsistent. The search for consistent choices uses a simple version of selective backtracking, but it does not use resolution to deduce smaller nogood sets. All nogoods involving an OR constraint are removed if the rule engine retracts the constraint.[12]

---

but appears to be unnecessary for these examples. I tried implementing a best-first version of this algorithm and discovered that the overhead of maintaining the best-first queue outweighs the advantages.

[12]This method can potentially yield an exponential number of nogoods. Consider, for example,

7.3.2 Protection Constraints

PF-TG's test programming and rule languages contain declarations for locking circuit nodes to prevent them from being modified during an interval of time. For example, to load and hold a value in a register that has an enable input, first enable the register, then load the value, disable the register and protect the disable node until the value is used. Figure 7.23 shows the full rule for controlling the output of a register, and lines 7-16 implement this method. The key step is line 15, which protects `?load-node` during the interval `(?disable-time ?output-time)`.

The Resource Manager cooperates with the Debugger (described in the next section) to handle protection constraints. Each protection interval is associated with a set of nodes, and whenever one of those nodes is assigned a value, the time is constrained to lie outside the protection interval. The `protect` test language statement provides a convenient syntax for declaring a protection interval (the duration of the `protect` statement) and a set of nodes to be protected.

The `protect-explicit` test language statement is similar. It protects a set of nodes for the duration of its body, except against statements that are contained lexically within the body. This statement allows two fragments to be merged while ensuring that one has exclusive control of a portion of the circuit. This feature is critical for preserving the functionality of complex component tests (e.g., the register file test of figure 7.25) while allowing program fragments that solve subgoals of the component test to be merged in.

7.3.3 The Debugger

This section describes the simulator and debugger that PF-TG uses to avoid protection violations involving unplanned side effects of the test. First, I describe what the debugger does and how it does it. Then I argue that the features in the test programming language that make this debugging step necessary are worth the cost.

7.3.3.1 What the debugger does

The set of node=value@time assignments chosen by the rule engine and the total order on the times chosen by the time manager forms a skeletal solution for a test. If

constraints that try to uniquely assign 9 values to 10 variables. The smallest nogood generated by the program in this case is 9 and there are lots of them. Space for the nogoods has not been a problem in practice.

```
  (define-propagation-rule REGISTER-L control-scalar
    ;; One clause for loading right before the output is needed
1.    ((register-l-control-scalar ?register out ?value ?output-time)
2.     (component-port ?register LOAD ?load-node)
3.     (new-time ?input-time)
4.     (t=-offset ?input-time ?output-time 1)
5.     (control-port-scalar ?register LOAD 1 ?input-time)
6.     (control-port-scalar ?register IN ?value ?input-time))
    ;; Another clause for loading and holding until the value is needed
7.    ((register-l-control-scalar ?register out ?value ?output-time)
8.     (component-port ?register LOAD ?load-node)
9.     (new-time ?input-time)
10.    (new-time ?disable-time)
11.    (t=-offset ?input-time ?disable-time 1)
12.    (t< ?input-time ?output-time)
13.    (control-port-scalar ?register LOAD 1 ?input-time)
14.    (control-port-scalar ?register LOAD 0 ?disable-time)
15.    (handle-protection-constraints ?load-node (?disable-time ?output-time))
16.    (control-port-scalar ?register IN ?value ?input-time)))
```

Figure 7.23: *Here is the library rule for controlling the output of an enabled register. The first clause implements a method that needs no protection because the value is used as soon as it is loaded. The second clause uses protection to load and hold a value: first enable the register (line 13), then load the value (line 16), disable the register (line 14) and protect the disable node until the value is used (line 15). Note that the order of subgoals in the text determines when the rule engine will work on them. These subgoals are ordered following the heuristic that control inputs are more difficult to control and should be handled first.*

PF-TG reaches this stage, all node assignments in the skeletal solution are justified back to primary inputs and observations out to primary outputs, and all conflicts between pairs of node assignments have been avoided. However, the solution may still be wrong because one or more protection interval constraints has not been met. These constraints come only from `protect` and `protect-explicit` constructs in test program fragments.

To check these constraints, an event-driven simulator "fleshes out the solution" using the total order on time variables and the persistence assumption[13] to predict values for all circuit nodes. The simulator starts with unassigned node values at the earliest time in the skeletal solution and stops at the latest. As the simulator moves forward, it checks the simulation against the protection intervals. If a protection interval has been violated, then it tries to patch the test by adding a temporal constraint to prevent the violation, calls the time manager to update the temporal variables, and starts the simulation over. If no bugs are found, then PF-TG calls the code generator to produce the final test program.

The debugging actions are limited (in the current system) to moving the time of the conflicting node or resource assignment out of the protection interval. The debugger does this by identifying the set of skeletal node assignments that cause the offending assignment and moving them as a rigid group either forward or backward (using a disjunctive constraint) so that the offending assignment is outside the protection interval. Currently, the debugger stops after it has found the first bug and passes the appropriate disjunctive constraint to the time manager. The next time the simulator is called to check the test, it saves work by resimulating only the portions of the test that have changed.

The debugging process always terminates. At each stage, a new temporal constraint is added that reorders a pair of times in the skeletal solution. This constraint is never retracted unless the test generator backs up into the rule engine to change the skeletal solution. The skeletal solution is finite, so there is a finite number of orderings of its variables. Therefore only a finite number of debugging steps can occur before a solution is found or the temporal constraints become unsatisfiable.[14]

The debugging process is not complete. In particular, the debugger cannot remove a node assignment that violates a protection interval, although by failing it can force

[13]The persistence assumption says that a node value persists until it is explicitly changed.

[14]There are clearly cases where debugging can take many iterations to terminate. However, in practice, few debugging steps seem to be necessary. Either the skeletal solution is almost right and one or two steps suffice to fix it, or it is very wrong and either never gets simulated or fails after a small number of debugging steps. This area merits further study.

the rule engine to backtrack and try again.

7.3.3.2 Why this seems to be effective

This heuristic debugging solution is not needed often, because PF-TG's propagation rules (and the rule engine) take positive action to prevent unwanted events. For example, the rule for loading and holding a register protects the register contents by explicitly setting the enable input to "disable" for the duration of the hold interval. This puts the disabling node assignment into the skeletal solution so that it will be justified back to primary inputs and definitely protected.

However, protection constraints resulting from `protect` and `protect-explicit` language constructs are not enforced when they associate protection intervals with nodes and times not involved in signal propagation. These nodes and times are not in the skeletal solution and are not handled by the normal conflict resolution mechanism.

These constructs are so useful, however, that they have been included in the language. They are useful because they allow the rule writer to say "prevent these nodes from being touched, but I will not give you a way to actively do this." Hence these constructs should be viewed as a filter on the possible solutions.

7.3.4 An Example Using More Complex Language Features

This example illustrates using a protection constraint to control how PF-TG embeds a complex component test. One method for testing the address lines of a register file appears in figure 7.24. Figure 7.25 shows this method expressed in PF-TG's test language. Lines 1 and 2 document the test. The `protect-explicit` form creates a protection constraint that prevents the test generator from manipulating the register file except as specified explicitly by statements in the component test. Without this protection, the test generator could, for instance, try to propagate signals through the file, thereby changing its contents and breaking the test. The `loose-sequence` form says that its three subforms (lines 5-9, 10-13 and 14-18) must be executed in sequence, but that time delays and other statements may be inserted between them. The loop on lines 6-9 initializes all cells to 1111, the all-ones-value for a 4-bit register file. Lines 12 and 13 write 0000 into address 1111, and the loop on lines 15-18 read back the contents of four cells. Symbols starting with ? are variables whose values are set by the rule that contains this fragment; their values are suggested by their names, e.g., `?data-in` references the data input. `address-stuck-at-stream` contains the values 1110, 1101, 1011, 0111 as specified by the method.

To detect stuck at faults in the address lines:

1. Initialize all cells to 1111
2. Select working address A = 1111
3. Write 0000 into A
4. Read contents of addresses 1110, 1101, 1011, 0111

To interpret the results: The output should be 1111. Reading 0000 from any of the four addresses indicates: (a) an address line stuck-at-1, i.e., data 0000 went to the correct 1111 address, or (b) an address line stuck-at-0, i.e., data 0000 went to the wrong location.

To merge this method with others: initialization must happen before writting 0000 which must happen before reading the four cells. Other test program code can be placed between these steps as long as they do not disturb the register file.

Figure 7.24: *How to test the address lines of a 4-bit, 16-cell register file (from [bennetts82])*

```
1.  (test-phase (:component ?rf
2.                :comment "Detect address line stuck-ats")
3.    (protect-explicit (:protected (?rf))
4.      (loose-sequence
5.        (comment ''Fill the register file with ones''
6.          (loop ((address (iota 0 15)))
7.            (equal-instants
8.              (control ?data-in ?all-ones-value)
9.              (control ?ca address))))
10.       (comment ''Write a 0 at one address''
11.         (equal-instants
12.           (control ?ca \#b1111)
13.           (control ?data-in \#b0000)))
14.       (comment ''Read other addresses to see if they were affected''
15.         (loop ((address ADDRESS-STUCK-AT-STREAM))
16.           (equal-instants
17.             (control ?address-line address)
18.             (observe ?data-output ?data-in-all-ones)))))))
```

Figure 7.25: *The register file test expressed in PF-TG's test programming language.*

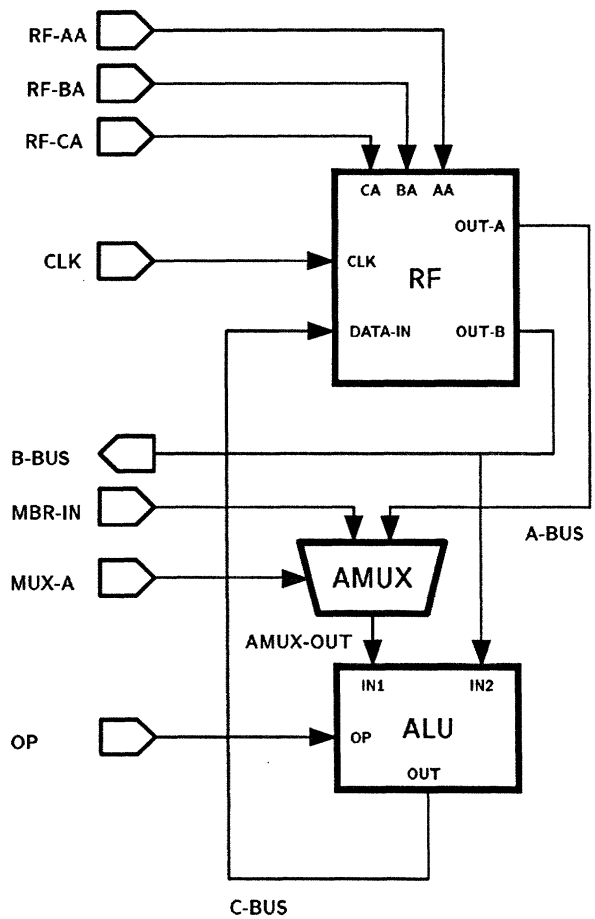


Figure 7.26: *A simple datapath*

Figure 7.26 shows a simple datapath containing a register file (RF), and figure 7.27 shows a test program for the register file produced by PF-TG. The structure of the test program reflects the structure of the component test: The first loop fills the register file with a background pattern of 1111's (lines 6-13). Then the program writes 0000 to one cell (lines 16-10) and reads four other cells to see whether they were affected (lines 25-30). Each of these steps has, however, been rewritten to reflect that fact that the file is inside the datapath and therefore not directly accessible. For example, PF-TG replaced lines 11-13 in the component test with lines 17-19 in the test program. These program lines have the same effect on the register file, but reference only primary inputs and outputs and can be carried out directly by the tester. A more complete test for this register file is shown in appendix A.

7.4 Discussion

What is the knowledge embedded in PF-TG and where is it? In the current system, all of the testing knowledge is embedded in the rules. Some of these rules implement line justification by describing how to achieve a single value or a stream of values on the output of a component. Other rules implement path sensitization in an analogous fashion. Still other rules contain program fragments describing how to drive the tester. PF-TG also has access to a schematic. Since the propagation rules are centered on individual components, the schematic is needed to match the names in one rule with the names another. Finally, some simple programming knowledge is embedded in the code generator for merging loops.

What knowledge and mechanisms are critical to solving the problem? Most important is the way rules specify how to break up the top-level goal into subproblems. The knowledge of good ways to decompose test programming problems comes partially from experienced test programmers (e.g., the register file test is composed of a subproblem to check the address lines and another to check the memory cells) and partially from propagation through the circuit structure. Posting temporal constraints allows PF-TG to commit very late to the order events happen in the test and the actual delays (in clock cycles) between them. Avoiding premature choices often reduces backtracking.

What kinds of circuits is this technique good for? Planning tests via goal decomposition and temporal constraint posting appears to work best for circuits where this yields independent or weakly-interacting subgoals. This appears to translate into datapath-like circuits, but subgoal independence depends upon how the circuit is modeled.

What kinds of circuits is this technique not *good for?* Goal decomposition and temporal constraint posting do not work well in situations where either (i) subgoals have strong interactions or (ii) where the ordering decisions strongly dictate the kinds of interactions between events. By contrast, the Designed Behavior Test Generator is most effective in precisely this situation, at least, it is when the circuit also provides few operations at its interface.

PF-TG is incomplete in two senses. First, PF-TG bounds the amount of work it will do to solve a problem. It currently does this by detecting and limiting cycles in the line justification and path sensitization parts of the program. For example, if the sensitive path loops through the same components twice, then the program cuts off propagation and backtracks. The number of cycles allowed before propagation is cut off is a parameter to the program.

Second, PF-TG does not necessarily search the full space within the above cutoff: while it does consider all orderings of events in the skeletal solution, it does not consider all orderings of all side effects that may conflict with a protection interval. This is a consequence of the debugger not trying all possible ways of fixing a skeletal plan.

7.5 Summary

This chapter introduced the Program Fragment Test Generator to illustrate several new ideas about test generation. These ideas are:

- Generate test programs rather than test vectors.
- This can be done by merging test program fragments.
- Represent the capabilities of the tester explicitly.
- Propagate Typed Streams.

The strategy of generating programs rather than vectors is used for three reasons. First, programs are often more compact than vectors, because looping constructs can efficiently encode repetitive tests. Second, programs are much more readable than vectors. Readability promotes accountability, and accountability is important in an environment where no single tool can solve the entire problem as is the case for testing complex sequential circuits. Third, modern tester architectures offer many special-purpose features, and test programming languages are a convenient and accepted way to access these features.

```
1.   ** TEST-PHASE (:COMPONENT RF)
2.   ** Detect address line stuck-ats
3.   BEGIN
4.       ** Fill the register file with ones
5.       BEGIN
6.           FOR ADDRESS FROM 0 TO 15 DO
7.               BEGIN
8.                   MBR-IN := 65535
9.                   MUX-A := 0
10.                  OP := ALU-NOOP
11.                  RF-CA := ADDRESS
12.                  CLK := 0; CLK := 1;
13.              END
14.      END
15.      ** Write a 0 at one address
16.      RF-CA := 15
17.      MBR-IN := 0
18.      MUX-A := 0
19.      OP := ALU-NOOP
20.      CLK := 0; CLK := 1;
21.      ** Read other addresses to see if they were affected
22.      ** DECLARE-STREAMS (ADDRESS-STUCK-AT-STREAM)
23.      BEGIN
24.          ARRAY ADDRESS-STUCK-AT-STREAM = [14, 13, 11, 7];
25.          FOR INDEX10 FROM 0 TO 3 DO
26.              BEGIN
27.                  RF-BA := ADDRESS-STUCK-AT-STREAM[INDEX10]
28.                  B-BUS = 65535
29.                  CLK := 0; CLK := 1;
30.              END
31.      END
32.  END
```

Figure 7.27: *A tester can directly execute this test for the address lines of the register file in figure 7.26. (Line numbers have been added to aid the discussion and are not part of the test.) Note that the looping structure of this test reflects the expert-supplied method (figure 7.24) but the test generator has replaced statements that reference internal nodes with other statements that manipulate the circuitry surrounding the register file that have the desired effect.*

PF-TG, the Program Fragment Test Generator, produces programs by combining program fragments in much the same way that a conventional test generator combines component behaviors or D-cubes. Test program fragments are selected and combined using both expert-supplied component tests and conventional signal propagation techniques. PF-TG rewrites component tests written in terms of component I/O pins into equivalent tests written in terms of circuit I/O pins. The resulting test programs reflect the sequence and looping structure of the original methods, but references to internal circuit nodes are replaced with equivalent statements for controlling tester driver-sensor pins or other tester features.

Explicitly representing the capabilities of the tester allows the test generator to recognize when special features are useful and to generate test program statements to apply them.

One of the key ideas in this chapter is choosing representations that are rich and expressive so that the structure of the test can be more easily identified and exploited. For example, we chose programs rather than vectors to make the structure more apparent to the test engineer. The sequences of test data used by experts have structure too, and this structure can be used to identify efficient methods of generating or observing it. To exploit this structure, PF-TG represents test data as typed streams. One task is to identify a useful vocabulary of stream types (e.g., counting streams) and properties (e.g., exhaustivity), and this chapter has provided a start on this vocabulary.

Chapter 8
Related and Future Work

Summary: The problem of generating tests for digital circuits has been addressed at length in the testing literature. This chapter describes several approaches that have influenced this work or are otherwise connected to it in interesting ways. Test generation has also been addressed in AI as a part of problem of circuit diagnosis; some of the central issues arising in testing have also been addressed in the robot planning literature. Finally, the capabilities and especially the limitations of this thesis suggest several opportunities for continuing work.

8.1 Related Work in Testing

8.1.1 Podem

While the Podem algorithm (described in section 2.3.4, page 64) was not a strong initial influence on this thesis, we have since realized that some of the strengths of Podem and DB-TG come from the same source: focusing search on achievable circuit behavior. Podem generates tests by searching the space of boolean assignments to circuit inputs. Since all input assignments are achievable by definition, Podem focuses closely on achievable circuit behavior and expends little [1] time considering unachievable behaviors. However, Podem's potentially exhaustive search at a low level makes the algorithm ineffective for complex sequential circuits. [2] DB-TG handles sequential circuits by searching a much higher-level space of "assignments," i.e., the operations that are valid at the interface.

[1] Podem does expend some time searching outside achievable behavior as it decides what input assignment to make next. This shows up as time spent backtracking from failures caused by circuit structure, i.e., not by previously made choices. In DB-TG's case, this time shows up as failed matches between component tests and simulated operations.

[2] In fairness to Podem's author, the algorithm was not designed to handle sequential circuits. However, one can easily apply it to them where it fails for the reason described above.

8.1.2 SCIRTSS

SCIRTSS (A Sequential CIRcuit Test Search System) [hill77] uses a two-part circuit representation: datapaths are represented as networks of boolean gates and control circuitry is represented as state transition graphs. The algorithm uses different propagation methods for these two distinct representations. Interaction between the two propagation methods increases search efficiency, especially since propagating through the state transition diagram is an effective mechanism for controlling circuit state.

Using component representations appropriate for its role within a larger system is a powerful idea. However, SCIRTSS' implementation of this idea has several drawbacks. First, role classification must be done by hand because we do not have programs capable of automatically recognizing component roles. For example, are the μPC and μIR registers in the MAC-1 data registers or part of a state machine? DB-TG can provide a reasonable answer to this question by searching the behavior graphs to see whether the registers are ever loaded with simple variables. However, more work must be done.

Second, it is not clear whether the roles of data and control are exhaustive. Perhaps additional or more finely-grained classifications and associated propagation methods would help.

Third, functional classification is static in SCIRTSS, i.e., a component's classification does not change during test generation. Clearly a component's role *can* change depending on what operation the circuit is performing. Consider, for example, the MAC-1 with DFT modifications. During normal circuit operations, the μIR functions as part of a state machine. During the test mode operations, the μIR functions as a register for scanning data in and out. SCIRTSS does not account for this kind of role change and presumably would require two models – one for the normal mode and one for the test mode – and changes to the algorithm to prevent redundant effort.

8.1.3 An Automatic Programming Approach to Testing

PF-TG is an extension of work described in preliminary form in [shirley85]. The earlier system propagated stream values and generated test programs as output by solving temporal constraints to combine program fragments. PF-TG extended this by using a larger vocabulary of stream types, solving temporal constraints incrementally rather than all at once when propagation finished, and handling a broader range of protection constraints. Solving temporal constraints incrementally increases speed roughly an order of magnitude on the examples in this thesis. PF-TG also gains

roughly another factor of 5 speed increase over the earlier system because its rule language is compiled Prolog rather than a homebrew interpreted language.

8.1.4 Knowledge Based Test Generation for VLSI Circuits

Brahme and Abraham[brahme87] describe a test generator that uses a hierarchical circuit model and shifts to the most abstract component descriptions available. Like SCIRTSS, this system uses separate representations and propagation heuristics for datapaths and control circuitry. Like PF-TG, it can search first for a path from a component input (output) to a circuit input (output), then work out the details of how to propagate signals across the path. In Brahme's system, this path serves to guide for propagation, taking the place of the controllability and observability metrics in other systems (e.g., Saturn[singh85]). PF-TG, on the other hand, finds groups of paths and uses them as abstract plans for controlling and observing the component under test. These abstract plans are then sorted to find the plans with the smallest number of subgoal conflicts (as estimated by path crossings).

Brahme's system apparently generates and solves temporal constraints on its propagation subgoals, as does PF-TG. The paper does not describe the types and representations of these temporal constraints nor how they are solved.

The system does, however, use an interesting heuristic: when testing datapath components, it defers propagating through the controller. By collecting all of the pertinent subgoals first, the system limits its search for ways to make the controller supply the set of needed signals. In general, I believe this is a useful heuristic, but the system might propose and work out the details of an unachievable pattern of movement through the datapath, then discover that the plan is unachievable only at the last minute when it examines the controller. Thus, this heuristic assumes that most patterns of movement in the datapath will be achievable. Whether this is so in general is not at all clear.

Perhaps this system could benefit from the notion of designed behavior to help propose patterns of data movement that are known or likely to be achievable. If the pattern is known, it should be straightforward to store the appropriate control actions with the pattern and to bypass propagating back through the controller. Alternatively – to reduce storage or precomputation costs – pieces of common patterns of activity could be stored. In this case, the test generator would need to propagate through the controller to verify a solution made up of a combination of known patterns of activity.

8.1.5 HITEST[robinson83]

The HITEST System and the ideas of Gordon Robinson, one of its architects, were an important influence on the design of both DB-TG and PF-TG. HITEST is a semi-automatic test generator for sequential circuits: the system handles the combinational bits of the circuit with a gate-level test generator and leaves it up to the test expert to specify how to handle the sequential bits. HITEST encorporates (i) a gate-level test generator, (ii) a fault simulator, (iii) a knowledge base of how-to-test rules called **test frames** and program fragments called **waveforms** and (iv) a mechanism for using the frames and waveforms. The gate-level test generator is a variant on Podem, and the fault simulator predicts circuit behavior and serendipitously detected faults during test generation. The most interesting parts of the system are the test frames and the waveforms.

Waveforms and test frames together express pre-written plans for accomplishing testing goals. For example, if the test programmer deems it important to be able to reset a particular counter within a circuit, he will give HITEST waveforms and test frames for doing that. HITEST combines these plans together with results from the gate-level test generator to form a complete test.

Test frames are written in a simple frame language and are primarily used to associate goals with subgoals. Waveforms are written in a test programming language and define specific patterns of activity at the circuit inputs that will cause something interesting to occur inside the circuit, e.g., load a particular register. Waveforms also contain points where the HITEST system can insert other waveforms or can insert data values generated by its gate-level test generator.

The subgoal mechanism in HITEST has several important restrictions. First, the system does not search through the space of subgoal expansions. If an expansion fails, the system abandons the goal, takes a new look at the predicted circuit state, finds another goal and continues. There is no facility for automatically patching plans that fail in minor ways. As a consequence, designing and tuning waveforms and test frames for a particular circuit is a difficult task and one that is deliberately left to the user. Second, HITEST was designed for situations where its testing goals would be independent. This design decision is reflected in several ways. For instance, waveforms must[3] be written in terms of circuit inputs. Thus, it is awkward to cause HITEST to combine several waveforms together to propagate a signal into or out of the circuit.

[3]Gordon Robinson points out that it is possible to violate this restriction, but only experts who are extremely skilled with HITEST succeed. They do so by cleverly partitioning the problem in advance so that all propagation subgoals are independent.

In PF-TG, this is straightforward. HITEST can concatenate two waveforms but it cannot merge them together by interleaving their statements as can PF-TG.

HITEST uses two powerful ideas. First, HITEST allows the expert and the system to share the work of generating tests, each concentrating on the jobs they do best. In the case of HITEST, the system handles the details of gate level test generation and fault simulation while the expert describes to the system how to manipulate the sequential parts of the circuit. For simple sequential circuits, HITEST's gate-level test generator and heuristics for capturing faults in registers can do most of the work. Lack of search in HITEST's subgoal mechanism forces the expert to do most of the work (in advance) for complex sequential circuits. Second (and expressed partially with the benefit of hindsight), HITEST waveforms are a way of describing known-achievable circuit behaviors to the system. In HITEST, a known-achievable behavior is tied to a single testing goal, i.e., the goal that will invoke it. In DB-TG, known-achievable behaviors are searched to find many ways they can be used to test the circuit.

8.1.6 Marlett[marlett86]

Marlett [marlett86] describes a test generation algorithm for sequential circuits that uses (roughly) gate-level models, attempts to detect conflicts as soon as possible, backtracks to the most recent relevant choice, uses a variation on the reconvergence heuristic, and relies partially on human guidance. This test generator appears to be quite effective on small, complex sequential circuits. However, it is not clear how well it works on larger sequential circuits such as the MAC-1.

8.1.7 Functional Testing of Digital Systems

Lai [lai81, lai83] describes a functional test generator that uses a manually created circuit representation. The representation is quite abstract, and in environments where it is available from the design or where it can be written by the test engineer, Lai's approach appears to be quite effective.

It is difficult to compare the performance of Lai's method with DB-TG because the performance measures are different. Lai reports 98.5% coverage of stuck-at faults in an ISPS model of a microprocessor. One can imagine how stuck-ats could have been mapped onto ISPS statement, e.g., assignment statements could modify one bit of the value being assigned and conditional branches could be stuck taking one branch, but these papers do not say.

DB-TG achieved 97% coverage of the stuck-at faults in a gate-level circuit model[4], which is the standard in the literature. The 97% figure is a percentage of the *detectable* faults in the circuit, as I removed stuck-ats in the ALU that are impossible to detect (as determined by an exhaustive gate-level test generator). These stuck-ats are associated with unused ALU functionality.

Lai's method ignores reconvergent fanout in the physical signal paths. Its coverage figures provide additional evidence that reconvergent fanout is not significant for complex sequential circuits.

8.1.8 Functional Testing of Microprocessors

In [brahme85], Brahme and Abraham describe a method that is tuned for testing microprocessors and is as a consequence quite effective. This is an example of a specialized testing method that gains its power by tackling a limited problem domain with techniques designed specifically for that domain. This method is currently limited to microprocessors, while neither DB-TG nor PF-TG is limited to a specific circuit type this closely. Whether Brahme's method is suitable for circuit can be determined by where the circuit sits along an axis of circuit type. The axis that characterizes the suitability of DB-TG and PF-TG appears to be only loosely related to circuit type. Whether DB-TG and PF-TG are suitable depends on the complexity of subgoal interactions during test generation.

8.2 Related Work in AI

8.2.1 Saturn [singh85, singh86]

The Saturn test generator is a particularly clean and comprehensive example of the hierarchical test generation methods described in section 2.3.5. First I describe Saturn's key features, then I focus on several points of comparison with DB-TG.

Saturn is similar to the D-algorithm: it uses a circuit model partitioned into components and embeds component tests using D-drive, justification and implication steps. Saturn makes three contributions to the state-of-the-art:

1. Saturn uses a uniform and expressive language for describing circuit structure and behavior at multiple levels of abstraction.

[4]except for the register file and microcode ROM which used functional models.

2. Saturn uses an effective strategy for switching between models during test generation.

3. Singh carefully identifies several ways that using high-level models can reduce search during test generation.

Saturn can either hypothesize specific faults, e.g., stuck-ats, or embed pre-written component tests. Saturn also uses heuristics to select the most promising choices in the search space and caches tests and solutions to subgoals, two useful features that have been explored in the testing literature.

The top-level goal behind Saturn and DB-TG is the same: reduce search. Saturn does this by using the abstract levels of a hierarchical circuit description whenever possible. DB-TG does this by focusing on the behavior the circuit will actually execute during normal operations rather than on other behavior. These two techniques are complementary, and a hierarchical version of DB-TG could be built and would be a useful extension.

Beyond this goal, there are numerous differences between the two methods. Some of these differences are good ideas that I did not encorporate into DB-TG because they are orthogonal to the notion of designed behavior, e.g., the meta-level reasoning features of MRS [russell85] that implement search heuristics and level shifting in Saturn. The differences listed below are more substantive.

Saturn requires that all abstract circuit descriptions be provided as part of the circuit model. The abstract descriptions used by DB-TG, i.e., operation relations and effects summaries, are generated automatically from the schematic and behavioral descriptions of the components. DB-TG combines low-level component behavior into higher-level descriptions for itself, using its simplification rules to identify useful abstractions.

Note that DB-TG does not try to solve the problem of recognizing high-level structure in a low-level circuit model in its full generality. The program tackles instead the more restricted task of recognizing structure in specific examples of a circuit's designed behavior. When it is successful, i.e., the simplification rules actually simplify, the operation relations reflect the high-level structure. When it is not successful, the operation relations are more complex, but no more complex than the structure of the circuit.

Saturn's circuit model is a strict hierarchy, while operation relations can cut across module boundaries and connect internal circuit nodes directly with circuit inputs and outputs.

Saturn suffers from the level-shift problem described in section 6.4.4. When shifting up one level, for example, Saturn guesses the values of unspecified bits when converting a set of bits into an integer. These guesses can easily be wrong and lead to unnecessary backtracking. Assuming Saturn were to use dependency directed backtracking, Saturn's strategy for shifting between levels could benefit from estimates of the likelihood of backtracking. A more sophisticated strategy would be to stay at a lower level to increase subgoal independence when backtracking is likely.

One key difference between Saturn and DB-TG lies in the area of handling reconvergence. Singh did not consider this issue in detail in his thesis, but Saturn could, with trivial changes, handle reconvergence soundly. It could do this by only using behavioral rules that *do not depend* upon the fault hypothesis. The hierarchical model, in effect, allows Saturn to "zoom in" on the circuit around the fault site to predict in detail how signals flow in the presence of a fault.[5]

Figure 8.1 shows an example. The circuit contains a four-bit carry-chain adder, a multiplier and other components that allow the adder's outputs to feed back into itself. The task is to generate a test for S0. Saturn cannot use the normal behavior rule for S0 because it is the component under test, i.e., it is potentially faulty. Saturn also cannot use the normal behavior rule for ADDER because it depends upon S0 to work correctly. Saturn is, however, free to use the normal behavior rules for S1, S2, S3 and MULT because they do not contain the component under test. Note that the multiplier has substructure whose rules could also be used, but Saturn will use the higher-level MULT rule instead.

Reconvergence causes additional difficulty with embedding component tests. In the example, S0 is implemented by several boolean gates, which Saturn would test by embedding a pre-written component test for S0. This strategy is unlikely to work: consider what happens as Saturn tries to embed one vector from the component test. If reconvergence causes propagation out from S0 to wrap back to S0, what behavior rule can Saturn use for S0? It cannot use the normal behavior rule because the component is potentially faulty, and the remaining possibilities are problematic:

1. Use the test vector itself. If the values being propagated do not match the test vector exactly, then the vector cannot be used. Thus this method is unlikely to work.

2. Replace S0 with its substructure and propagate through that. This does not

[5]This aspect of the algorithm appeared in [genesereth81], which used hierarchical models in general, and in [shirley83b], which used hierarchical models for the specific purpose of avoiding potentially interfering fault effects.

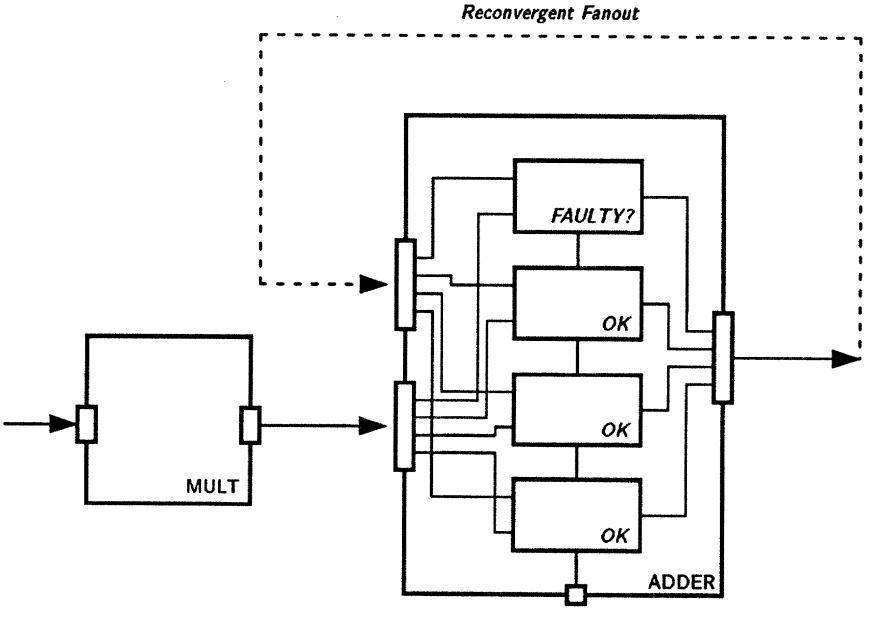


Figure 8.1: *Saturn could handle reconvergence soundly, at the cost of dropping to the gate level around the fault site.*

work because Saturn cannot use rules for S0's substructure when S0 (or any of S0's substructure) is potentially faulty.

Guaranteeing soundness in the face of reconvergence forces Saturn to propose faults at a level where very specific predictions can be made about misbehavior, e.g., the gate level. At this low level, option #1 works because there are so few behaviors that either the test vector matches exactly or there is not solution and propagation must fail. Note that these problems only occur in the presence of reconvergence, and even then Saturn gains somewhat by using abstract descriptions in areas of the circuit away from the fault.

Saturn's strategy relinquishes abstraction to guarantee soundness. DB-TG could also use this strategy (as suggested in section 5.3.3), however I have proposed a different approach in this thesis. Using abstract descriptions is unlikely to yield unsound tests, therefore we can save effort by using abstract descriptions and fault simulating the resulting tests. Saturn could use this strategy too.

8.2.2 Choosing Models Based on the Cost of Reasoning

Kramer [kramer84, kramer85] describes a system that solves line justification problems in sequential circuits. This system chooses among multiple models of circuit behavior by minimizing the expected cost of solving the problem over the different models. This idea could be used to decide between propagating through operation relations or propagating through circuit structure, based on the complexity of these models and an estimate of the likelihood of reconvergent fanout causing a problem.

8.2.3 Case-Based Reasoning

DB-TG has several themes in common with the Case-Based Reasoning (CBR) literature and in particular Derivational Analogy [carbonell85]. CBR is a method of solving problems by transferring past experience to new problem situations. This process is illustrated in figure 8.2. Given a problem to solve, CBR selects a similar problem from a library of previously solved problems and solutions. If the new and old problems are exactly the same, then the old solution can be used without change. If, however, the new and old problems are somewhat different, i.e., there is a *partial* mapping between them, then the old solution must be modified to solve the new problem. This approach is useful when the work needed to modify the old solution is less than the work of solving the new problem from scratch.

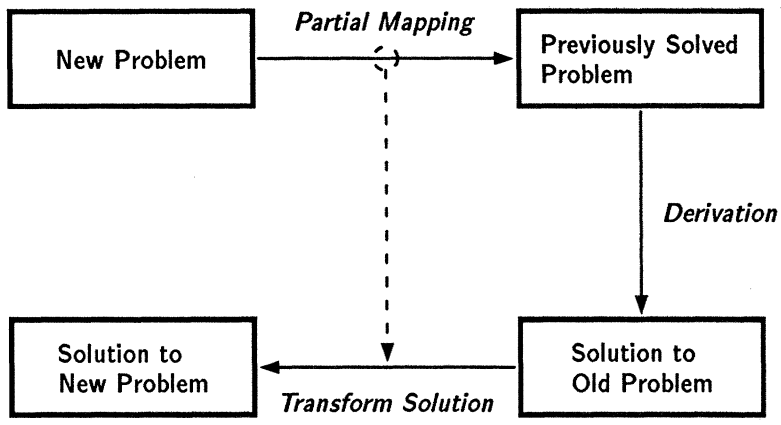


Figure 8.2: *Cased-Based Reasoning*

DB-TG can be viewed as a case-based problem solver in the following way: (i) DB-TG's set of behavior graphs is the library of previously solved problems; (ii) behavioral subsumption is DB-TG's similarity metric, i.e., if a simulated pattern of activity subsumes a desired pattern of activity, then the simulated pattern is considered similar; and (iii) solving for parameters of a circuit operation is DB-TG's method of modifying the old "solution". Thus DB-TG can be viewed as a case-based problem solver that creates its own library of solutions by simulating circuit behavior. The library comes from simulation rather than past problem-solving behavior because test generation happens infrequently per circuit and detailed solutions involving one circuit are unlikely to be helpful on the next.

Derivational Analogy (DA) is a particular case-based reasoning method that suggests some useful extensions to DB-TG. In DA, the library describes how solutions were derived as well as the solutions themselves. DA then gauges similarity by comparing the first few steps of an agent trying to solve the problem against the first few steps of the stored derivation. If these match, then DA modifies the old derivation by "replaying" as much of it as is appropriate in the new context and then reinvoking the problem solver to finish the solution.

One of the proposed extensions to DB-TG resembles this strategy. DB-TG can increase its performance when run again after circuit modifications by taking advantage of the previously generated test. It can do this by using the previous test or pieces of the test as "virtual circuit operations," simulating their effects and searching for ways to embed component tests. In effect, DB-TG could mine earlier versions of a

test program for good ideas and for the things it did right serendipitously. This may help reduce the cost of updating a test program due to ECO's or other kinds of design changes.[6]

8.2.4 Bumpers[miller85]

D. Miller [miller85] describes a planner called **Bumpers** that solves robot navigation tasks by searching totally ordered plans. Miller's argument begins with a strong and appropriate criticism of least-commitment planning:

> The whole least-commitment philosophy is designed for domains where interactions dictate ordering decisions - not the other way around. When insufficient commitment is made during the planning process, either in plan choice or in plan ordering, the planner can spend a considerable amount of time exploring unprofitable or infeasible plan avenues - plan possibilities that would be obviously unwise if only it had made and stuck to an earlier decision.
>
> Domains that involve efficiency constraints, loops and continuous resources can have plan interactions that are dependent on how the pieces of the plan are ordered. Least commitment planners are designed to make ordering decisions only when adverse interactions are detected. They therefore have difficulty in the domains described above because no interactions appear until an ordering is enforced and the planners will not enforce an ordering until an interaction appears.
>
> – [miller85], p.16

In response to this criticism, Bumpers explores the space of total orderings of plan actions. The planner does this by enumerating plan prefixes and using temporal, spatial and resource-based constraints to prune inconsistent orderings. Predicting the consequences of a plan prefix in order to check constraints is simulation, since the prefix is a total order. Bumpers also has heuristics for controlling how the space is searched. It can use, for example, estimates of total plan efficiency, e.g., how long the plan will take to execute, to decide which plan prefixes to expand further.

[6]The idea is somewhat more involved than this. In particular, a very simple way to reuse old work is to annotate tests with the components they depend on. If none of those components have changed, nor have the circuit operations used by the old test, then that test can be reused directly. After handling as much of the circuit as possible with this simple technique, DB-TG would then try to mine the remaining old tests by simulation and matching.

Bumpers and DB-TG were developed independently, but they share some underlying ideas. Both systems search known-achievable behavior. In the circuit testing domain, much of the interesting activity occurs within single primitive actions, i.e., circuit operations. Hence DB-TG searches simulations of single circuit operations. Since the interactions between circuit operations are fairly limited, DB-TG uses conventional goal-directed planning technology to search sequences of operations and does so using abstract descriptions and as something of a last resort. In Miller's formalization of the robot planning domain, most of the interesting activity occurs in the interaction between primitive actions, hence Bumpers searches simulations of sequences of actions.

8.2.5 Joyce's Extensions to DART[joyce83]

R. Joyce describes several extensions to DART [genesereth84] that use temporal least-commitment to improve search efficiency when generating tests for simple sequential circuits. The design of Joyce's system was an important influence on the design of PF-TG. Both systems have a backward chaining rule engine (Joyce used MRS rather than Prolog) and a separate time manager. However, PF-TG's Time Manager and rule engine run as co-routines while they execute serially in Joyce's system. PF-TG runs the time manager concurrently with the rule engine so that it can detect conflicts earlier and cause the system to backtrack. Both systems handle disjunctive constraints, but PF-TG takes advantage maintaining a partial solution to heuristically pick disjuncts that are likely to be satisfiable (i.e., disjuncts that are already true in the partial solution). In practice, this increases the efficiency of handling disjuncts, which both systems rely on heavily to represent resource constraints.

8.2.6 Automatic Programming

Of the large automatic programming literature, PF-TG most closely resembles Barstow's PECOS system [Barstow79] in its use of a library of refinement rules. However, PF-TG is much simpler than PECOS. Its simplicity reflects fundamental differences between the symbolic programming domain of PECOS (e.g., sorting and graph algorithms) and test programming. For example, Barstow's refinement rules map between general programs, from the abstract to the slightly more detailed. PF-TG's rules map between testing goals expressed in a fairly limited vocabulary to solutions expressed as code fragments and constraints. This strategy will works well if broadly useful pieces of test programs can be written to be almost independent, which appears to be possible for the simple sequential circuits that PF-TG is targeted

at. PF-TG relies on an equation solver to fit pieces of solutions together because of the primacy of timing relationships in testing.

8.2.7 Wu's DFT Advisor[wu88]

The relationship between Wu's DFT Advisor and DB-TG is described in detail in section 6.5. There are also relationships between the DFT Advisor and the program fragment test generator. Both systems can propagate streams of values, for example, although PF-TG has a somewhat richer vocabulary of stream types and properties. The DFT Advisor does not reason about time, i.e., it assumes that circuits behave as if they combinational, even if they are in fact sequential. In the context of redesign for testability, the test generator can handle many situations despite this simplification because the timing of the circuit is going to be changed anyway. However, augmenting the DFT Advisor's test generator to reason about time as does PF-TG would be a useful enhancement.

In its turn, the DFT Advisor has a capability that may be useful to add to PF-TG: the DFT Advisor can in some cases recognize when several components can work together to achieve a testing goal. One example given in [wu88] involves testing the ROM in the MAC-1 by exhaustively enumerating its addresses and checking each data value as it emerges. Driving the ROM's data input is a collection of three components: a multiplexor, an incrementer and a register holding the μPC. The DFT Advisor can recognize that these three components can be used together as a counter to enumerate the ROM addresses.

The DFT Advisor recognizes that groups of low-level components form high-level components. It does this when called for and guided by specific testing goals rather than trying to solve the problem in the general case. Recognizing groups of components rather than being told (e.g., by a hierarchical circuit model) is useful because it can identify unanticipated uses of components. Whether this technique is worth its cost depends on how often unanticipated uses can be found.

8.3 Future Work

The capabilities and especially the limitations of the ideas in this thesis suggest several opportunities for continuing work.

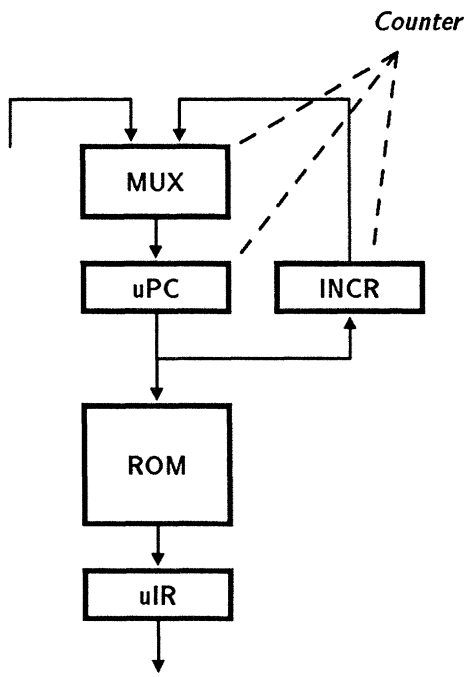


Figure 8.3: *The multiplexor, incrementer and μPC register together form a counter. The DFT Advisor can recognize this without being told and use them together to generate a counting-stream to help test the ROM.*

8.3.1 DB-TG: Improving the Implementation

Careful indexing can speed up DB-TG's search for operation relations. DB-TG already uses one indexing technique for this purpose: the component activation summary lists the kinds of component operations that occur within each behavior graph. If an operation type does not appear in the summary for a graph, then the graph need not be searched. The ALU/INVERT operation, for instance, does not appear in the component activation summary for the MAC-1/SUM instruction, therefore DB-TG can skip searching the MAC-1/SUM behavior graph for simulated instances of the ALU/INVERT operation.

This idea can be extended in several ways. For instance, the component activation summary could be reindexed by component operation type rather than by behavior graph, implemented as a discrimination net with behavior graphs in the buckets. This indexing trick would move work from the equation solver to a cheaper discrimination mechanism.

The component activation summary could also be extended to include the types of the operation parameters. For instance, instead of mapping all ALU/ADD operations together like so:

$$\texttt{(ALU/ADD ?pc 1)} \mapsto \texttt{ALU/ADD}$$

DB-TG could map them to a larger set of buckets which would discriminate more during lookup. For instance,

$$\texttt{(ALU/ADD ?pc 1)} \mapsto \texttt{(ALU/ADD controllable-value constant-value)}$$

The key problem here is identifying a set of properties of operation arguments (e.g., controllable and constant) that discriminates well and lets the program avoid unnecessary search without being too expensive to check.

A second technique can reduce the space needed to store behavior graphs. Rather than record everything that occurs during a simulation run in the behavior graphs, DB-TG could record only what happens to certain landmark components, e.g., registers. Then to generate a test, DB-TG would use conventional techniques to propagate signals from the component under test out to landmarks. The assignments to landmarks could then be used as a search key for the behavior graphs. One interesting issue this raises is which components would make useful landmarks. A criterion based solely on component type would probably not perform as well as one based on a component's role in the circuit. In the MAC-1, for example, the accumulator, the program counter and the microinstruction register would be useful landmarks, but the latches in the datapath would not. All of these components are registers, but they play different roles in the microprocessor.

8.3.2 DB-TG: Extending the Representation

One goal for extending DB-TG's representation of tests and circuit behavior is to take more advantage of repetitive behavior and tests. Many tests involve repeating a sequence of steps: working with these sequences as aggregates is a powerful technique for speeding up test generation. Sometimes sequences are regular in a way that can be captured nicely by our test representations. For instance, the two level test representation in DB-TG consisting of a Component Test Operation and Component Test Data nicely captures repetition where a single operation is executed repeatedly with changing data. PF-TG's stream vocabulary gives it a somewhat more powerful method for exploiting repetition without reinitializing the circuit as often. DB-TG uses the simpler test representation because it can not take advantage of the more powerful one currently. The difficulty lies with its representation of circuit behavior, i.e., the behavior graphs.

DB-TG's behavioral representation is less successful at describing repetitive behavior. Within the behavior graphs, a signal is represented as a data value and a time. A data value is represented by an algebraic expression that can contain variables. This captures repetition only in the operation/data sense, i.e., the values of variables within an expression can change, but the expression remains the same. The timestamps are real numbers representing the beginning of the simulated interval of time when the circuit node held the value denoted by the data value. In order to describe repetitive behavior, DB-TG must create behavior graphs that contain many data/time pairs, one for each repetition. Spreading out the description into many timestamped values makes matching the behavior against repetitive component tests difficult.

There are three problems that must be solved: (i) representing repetitive circuit behavior, (ii) generating descriptions of repetitive circuit behavior from schematics and component behavior and (iii) matching repetitive circuit behavior against component tests. The literature contains suggestions for several representations for streams of signals that might be adapted and extended for use in DB-TG. In [breuer79], Breuer proposes an interesting representation for bit-level streams based on regular expressions. CRITTER [kelly82] uses an algebraic signal representation that has variables in the timing fields.

A second change, orthogonal to representing signals as streams, involves representing cyclic behaviors explicitly by cycles in the behavior graphs. Weld's ideas about aggregation [weld86] could potentially be adapted for use here. Weld's simulator, working in the domain of Molecular Biology, recognizes cycles by comparing the current state of the simulator against the simulator history. When a pair of states are

found that are similar in a particular way, then the simulator recognizes a cycle. If the two states have identical state variables, for instance, then there is clearly a cycle. If all but one of the state variables are identical and that one is different in one of a small set of ways, e.g., it has been incremented, then the simulator also identifies a cycle. Complex digital circuits have a lot of state that would have to be compared, however a person may be able to focus its comparison by telling the program which state variables pertain to important cycles.

A third kind of repetition involves similar behaviors spread out over space rather than time. Figure 8.4 shows an example. The task is to check whether this communications multiplexor can transmit a character from the bus on the left out to a terminal on the right. The behavior needed to transmit a character from the bus to Terminal 1 is very similar to the behavior needed to transmit a character to Terminal 2. Current test generators and DB-TG do not take advantage of this similarity. They would either solve the problem 4 times or, if they cache subproblem solutions, would reuse some pieces of the solutions. Human experts appear to know this similarity of behavior is a characteristic of this kind of circuit and set out to exploit it from the start. If so, then this similarity is built into their circuit representation and they do not have to discover it by remembering subproblems. One intriguing idea is the possibility of using explanation based generalization techniques [mitchell86] to identify similarities between behavior graphs. When a similarity is found, the behavior graphs could be combined to form a more general behavior graph which would then be used for test generation.

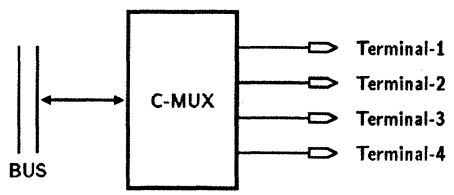


Figure 8.4: *Here is an embedding problem that involves similar behaviors spread out over space rather than time. The behavior of transmitting a character from the bus to Terminal 1 is very similar to the behavior of transmitting a character to Terminal 2. This similarity is easily exploited by human test experts, but current test generators and DB-TG do not take advantage of it.*

Another extension involves simulating and searching the effects of sequences of circuit operations rather than of individual operations. For instance, I have experimented with considering the MAC-1/LOAD/SUM/STORE sequence to be a single

"compound" circuit operation and having DB-TG search the corresponding behavior graphs, to find a way to embed the ALU/ADD operation without using the State Planner. By searching the behavior graph for a compound operation, DB-TG could match and successfully embed component tests that could not be matched within a single circuit operation. It would likely be too expensive to simulate many compound circuit operations, but this method offers a promising way for a human test programmer to guide DB-TG by supplying good operation sequences to try. One interesting extreme case of an interesting operation sequence is a previously written test program, which may no longer work due to design changes. Following this idea, DB-TG may be able to mine a test program for useful operation sequences.

8.3.3 DB-TG: Continuing the Empirical Work

Two of the heuristics used in DB-TG – the designed behavior heuristic and the reconvergence heuristic – should be tested further on a wide spectrum of circuits. These heuristics are currently based on (i) our test experts' experience, (ii) the partial success of functional testing techniques and (iii) a small amount of empirical evidence in this thesis. Complex circuits are difficult to model, so experimenting with a large set of them requires both the ability to import circuit descriptions in an industry standard format such as VDIF or VHDL and a suitable symbolic simulator.

8.3.4 DB-TG: Forging a Stronger Link with Design

One of the most interesting directions for extending DB-TG is to look for ways of extracting operation relations or similar abstract relationships between components from the design process, thereby bypassing the current simulation stage. Both the design synthesis and design verification tasks are potential sources of operation relations.

Consider a hypothetical synthesis tool that designs circuits by (1) breaking up the desired circuit behavior into pieces, (ii) selecting (perhaps modifying) landmark components to directly implement important pieces of the circuit behavior and (iii) implementing the rest of the circuit around those landmarks. For instance, think of the ALU and the register file in the MAC-1 datapath as landmark components and the rest of the datapath as glue for connecting them together. This is a plausible approach that manages the complexity of design by first selecting landmark components from a limited vocabulary, then fits them together.

In a design process like this, relationships between the operations of landmark

components and operations of the circuit (or containing module) are known early, and it is the purpose of the glue circuitry to implement those relationships. Other than for design verification, there is no point in having the test generator propagate through that glue circuitry or having it reconstruct the operation relations via simulation.

Design verification is another potential source of operation relations. Some systems (e.g., [kelly82, barrow83, weise86]) effectively compute symbolic simulations of circuit behavior in order to compare them against circuit specifications. Given such a system, it may be a small amount of extra work to construct operation relations from the output of such a verification system.[7] If so, then the test generator could share work with the design verification system

8.3.5 PF-TG: Improving the Implementation

PF-TG could stand improvements to make writing propagation rules easier. Writing propagation rules for PF-TG is difficult for two reasons: (i) rules for some components can be overly verbose and (ii) the propagation rules mix descriptions of component behavior with how-to-test knowledge. The first problem can be addressed by adding syntactic sugar to the language. For example, the behavior of some components can be described most easily using truth tables, consequently, truth tables should be provided in the language or as macros.

The second problem occurs in the parts of propagation rules that specify protection intervals and temporal least-commitment. These concepts involve how-to-test knowledge rather than intrinsic properties of the circuit, hence these rules cannot be generated from component structure and behavior without some additional knowledge. The rules used by PF-TG were straightforward to write by hand. However, determining how to generate this sort of propagation rule automatically is an open problem.

The amount of work involved in describing complex circuits has been a significant barrier to experimental test generators that use nonstandard circuit descriptions. This problem can be addressed by importing circuit descriptions from one of the industry standard formats. One promising approach for doing this is described by Kramer in [kramer84]. This approach has not yet been tried on a full-featured circuit description language and would be an interesting development project.

A second improvement involves a technique called Qualitative Fault Simulation. Often a test generated for one component will also fully or partially test another

[7] Some of these systems are hierarchical, so creating operation relations would involve collecting and composing functions output by the verifier.

component. A test generator can exploit this fact to save effort if it costs less to recognize serendipitous tests than it does to generate new tests. Doing this can also yield shorter tests, since each test vector detects more faults on average.

One well-known technique is to simulate each vector as it is created and to remove all detected faults from the fault list. Another technique is to actively try to generate tests that exercise more than one component at a time. This is done by, for example, the LASAR algorithm [bowden75, breuer76]. A third technique is a hybrid of the first two: simulate each vector [8], then generate a test for another fault by adding to the test for the first fault. This technique is used by PODEM-X [goel81b].

These three techniques are applicable to PF-TG if the fault simulation step is modified so that it can determine how well streams of values cover faults in a component. One would have to develop a language for describing how a component has been partially tested and then determine how the streams in PF-TG's vocabulary partially cover the components in PF-TG's library. Consider the task of testing the ROM in the MAC-1 plus testability modifications. One way to test the ROM is to shift an address into the μIR and load that into the μPC (DB-TG uses this method). Then capture the ROM output in the μIR and shift back out to the tester. As the ROM is tested, the μPC is fully exercised (note that each bit is exercised) and the μIR is exercised except for its outputs that go over to control the datapath. A trivial language for describing how a component has been partially tested is the fault list. A more interesting language would allow more abstract descriptions, which need not be completely accurate as long as they are conservative.

8.3.6 PF-TG: Extending the Representation

PF-TG could use more comprehensive descriptions of tester capabilities and restrictions. Pin Formatting, for example, is one important area of human expertise that has not been addressed well by test generators. Modern testers provide facilities for accurately controlling the timing of edges, i.e., voltage changes, as they apply test vectors. For instance, a hand-written test program might set up a timing phase signal like the one in figure 8.5 before applying a sequence of test vectors. This signal is described in terms of three parameters: the *cycle time*, i.e., the amount of time per test vector, $e1$, i.e., the delay from the beginning of a cycle until test data is applied to the circuit inputs, and $e2$, the delay until the test data is (optionally) removed from the circuit inputs. Figure 8.5 also shows two pin formats. The first format is

[8] In some test generators, simulation is unnecessary as the node assignments produced during test generation are sufficient.

called surround by complement where a value D from the test vector is applied to the circuit surrounded by its inverse $\overline{D}$. The second format, Return to Zero, surrounds the data with logical 0's. Similar features are provided to control when to sample voltages coming out of the circuit.

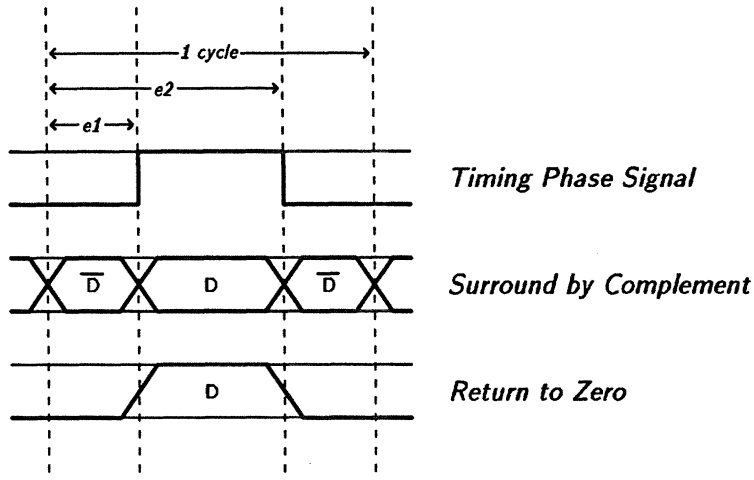


Figure 8.5: *A cycle description and two pin formats*

In some cases, experts can use the Surround by Complement pin format to effectively double the speed of a tester. Pin formats can also be used to test whether a circuit meets its timing specifications by bringing the edge delays $e1$ and $e2$ closer together or reducing the delay before the tester samples the circuit outputs. These tasks are commonly performed by test experts, but are beyond the scope of traditional test generators.

8.3.7 PF-TG: A Test Generation Apprentice

An important disadvantage of PF-TG relative to an expert and DB-TG is PF-TG's lack of a global understanding of circuit behavior. However, PF-TG is good at fitting together test program code. One interesting idea is to use PF-TG as the basis of a test generation apprentice somewhat along the lines of HITEST and the decision steering ideas of [marlett86]. PF-TG could be used as an interactive system that accepts abstract plans for testing a component in the form of a collection of datapaths to use. DB-TG would then combine its data movement plans for the components along the paths to produce a test program. In this view, the human would be doing the

key parts of the test generation task, i.e., selecting groups of paths that are likely to work. PF-TG would do the limited search necessary to put together the pieces or to determine that the plan would not work.

Chapter 9
Conclusions

This thesis makes several contributions to the field of circuit testing:

- It introduces operation relations, a representation of circuit behavior that often makes embedding problems easy.
- It describes a way to compute operation relations by symbolic simulation. This method is efficient for circuits that offer a small number of operations at their interface.
- It introduces the designed behavior heuristic, i.e., test a circuit without going outside its normal operations, elucidates the issues surrounding this heuristic and provides empirical evidence that the heuristic is useful.
- It describes an automated method of creating test programs by combining test program fragments.
- It demonstrates how propagating typed streams of values can produce more efficient tests and introduces a vocabulary of stream types.
- It extends the goals of test generation to include using the capabilities of the tester well. To achieve this goal, PF-TG uses an explicit description of tester capabilities and resource limitations.

DB-TG and PF-TG are two novel test generators that extend the range of techniques available to test engineers. DB-TG and PF-TG are complimentary: the first is effective on complex sequential circuits that display tightly interacting component behavior. Search in DB-TG is indexed and guided primarily by what is possible for the circuit to do rather what is desired to test a component. However, the cost of simulating circuit operations renders DB-TG inefficient for circuits where many operations are possible. PF-TG uses conventional goal-directed planning techniques and is targeted at simpler sequential circuits. In PF-TG, search is indexed and guided

primarily by specific testing goals, i.e., how to test a component, and is not limited by the number of circuit operations.

PF-TG's use of conventional planning techniques provides a testbed for experimenting with several other aspects of test generation. For instance, PF-TG produces test programs rather than test vectors to raise the level of the language between the test generator and the agent that will carry out the test. Using this richer language, together with using a simple model of the tester capabilities, helps PF-TG to design more efficient ways to test a circuit.

These new methods plus the existing combinational, functional and special-purpose test generators (e.g., for memories) form a collection of tools that test engineers can draw upon as appropriate. Our larger goal, of which this thesis is a part, is to build a collection of specialized testing tools that share circuit descriptions and work together autonomously or partially under human guidance to solve testing problems.

Appendix A
PF-TG Examples

This appendix contains four sample tests generated by PF-TG. The ALU test in section A.1 forces PF-TG to manage circuit nodes as resources, since the ALU is used twice in the test: once to help set up one of its inputs and again to actually perform the test. The register file test in section A.2 is considerably more complex and uses a protection constraint to prevent PF-TG from breaking the component test as it embeds it. The ROM test in section A.3 illustrates typed streams by solving a problem once by propagating a scalar quantity and again using a stream. Finally, another ROM test in section A.4 shows how PF-TG can generate tests involving DFT components like a BILBO register.

A.1 An ALU Test

Figure A.1 shows the usual method of testing an ALU's addition operation by applying pairs of addends and observing the outputs. Lines 3-6 bind Prolog variables to the circuit nodes around the ALU into (e.g., ?op is the ALU's operation control input). Lines 8-16 contain the test program fragment, and the data for the three streams is declared elsewhere. Lines 13-16 form the core of this component test. They apply three streams to the ALU inputs and observe one stream coming out of its output. The `with-synch` declaration forces the loop portions of the streams to step together. The `equal-instants` declaration forces the three streams deliver their values and the one stream to observe the output simultaneously (within each repetition).

Figure A.2 shows a version of the MAC-1 datapath, and figure A.3 shows an embedding found by PF-TG. This test program begins with several lines of comments to the test engineer, then there are three array declarations containing the expert-supplied streams. The bulk of this fragment is a FOR loop that steps through the arrays, writing values from the addend-streams to the circuit and checking that values in the sum-stream come back out. Activity in the loop body occurs over three clock cycles: (i) load one input through AMUX and ALU into the register file RF at address 0; (ii) read the value from RF and apply it to the ALU via the B-BUS, apply the

```
1.  (define-predicate SIMPLE-ALU-PLUS-TEST
2.    ((simple-alu-plus-test ?alu)
3.     (component-port ?alu in1 ?in1)
4.     (component-port ?alu in2 ?in2)
5.     (component-port ?alu out ?out)
6.     (component-port ?alu op  ?op)
7.     (declare-fragment
8.       (test-phase (:component ?alu :facility plus
9.                    :comment "test the ALU's ability to add"
10.                   :mnemonics (ALU-ADD 0))
11.        (with-synch
12.          (equal-instants
13.            (control-stream ?in1 (STREAM ADDEND-STREAM-16-1))
14.            (control-stream ?in2 (STREAM ADDEND-STREAM-16-2))
15.            (control-stream ?op  (REPEAT ALU-ADD))
16.            (observe-stream ?out (STREAM SUM-STREAM-16))))))))
```

Figure A.1: *A component test for the ALU's addition operation: version 1*

other value from MBR-IN through AMUX, and load the sum into RF at address 0; and (iii) read the value from RF and check it via B-BUS. The fragment repeats these steps 8 times, once for each group of data in the streams.[1] Note that the ALU's operation input (OP) is handled differently from the data inputs, because a single value (ALU-ADD) is being applied repeatedly rather than a stream of values. The assignment `OP := ALU-ADD` cannot be moved out of the loop because the operation input is changed in order to apply addend-stream-16-2.

[1] ALU-NOOP and ALU-ADD are mnemonic constants that I have used here to enhance readability. They are declared when used in the rules, and PF-TG collects the declarations used within a single problem, i.e., test-phase, and includes them at the beginning of that test phase.

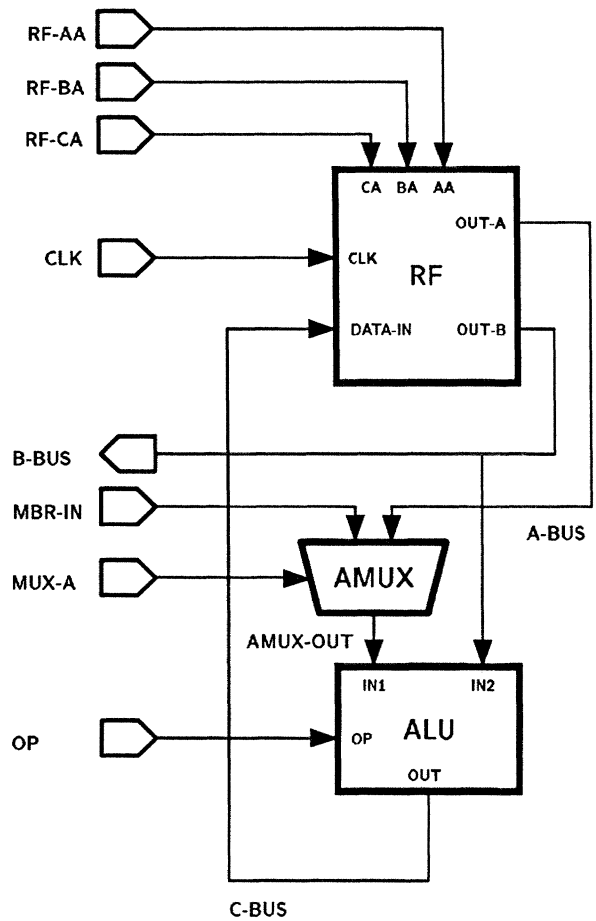


Figure A.2: *A simple datapath*

```
** TEST-PHASE (:COMPONENT ALU :FACILITY PLUS)
BEGIN
    VAR ALU-ADD = 0, ALU-NOOP = 2;
    ** test the ALU's ability to add
    CLK := 0; CLK := 1;
    ** DECLARE-STREAMS (ADDEND-STREAM-16-1 ADDEND-STREAM-16-2 SUM-STREAM-16)
    BEGIN
        ARRAY ADDEND-STREAM-16-1 = [0, 43690, 1, 1, 65534, 65535, 21845, 65535];
        ARRAY ADDEND-STREAM-16-2 = [0, 43690, 65534, 65535, 1, 1, 21845, 65535];
        ARRAY SUM-STREAM-16 = [0, 21844, 65535, 0, 65535, 0, 43690, 65534];
        FOR INDEX11333 FROM 0 TO 7 DO
            BEGIN
                RF-CA := 0
                MBR-IN := ADDEND-STREAM-16-2[INDEX11333]
                MUX-A := 0
                OP := ALU-NOOP
                CLK := 0; CLK := 1;
                MBR-IN := ADDEND-STREAM-16-1]
                MUX-A := 0
                RF-BA := 0
                RF-CA := 0
                OP := ALU-ADD
                CLK := 0; CLK := 1;
                B-BUS = SUM-STREAM-16[INDEX11333]
                RF-BA := 0
                CLK := 0; CLK := 1;
            END
    END
END
```

Figure A.3: *The embedding for the ALU addition test*

A.2 A Register File Test

This section contains a complete version of the register file test shown in chapter 7 (page 233). Figure A.2 shows the circuit, and figures A.4 and A.5 show the complete component test. Lines 3-6 fetch global names from the circuit nodes surrounding the register file (?rf). Lines 8-10 create several "constants" whose value depends on the bit-width of the data input. Line 7 creates two temporal variables that will be exceptions to the `no-implicit-assignments` statement on line 15. That is, manipulations of the register file at these two times will be allowed, even if the statements doing it are not lexically contained within the `no-implicit-assignments` statement. The exceptions are needed because this test uses several subroutines (e.g. `write-at-1111-read-back`) that manipulate the register file but whose internal statements are not contained.

The main body of the test is a loose sequence of three test phases. The first test phase fills the register file with ones, writes zeros into one cell and reads back several other cells looking for stuck-at faults in the addressing logic. This phase was shown in chapter 7. The second test phase checks for bridge faults instead. It does almost the same thing, but writes zeros to a different location and reads back different cells. The final test phase writes and reads a checkerboard pattern (an alternating 0-1 pattern) using each register. Figures A.6, A.7 and A.8 show this component test embedded for the register file.

```
1.  (define-predicate REGISTER-FILE-TEST
2.    ((register-file-test ?rf)
3.     (component-port ?rf data-in ?data-in)
4.     (component-port ?rf out-a ?out-a)
5.     (component-port ?rf aa ?aa)
6.     (component-port ?rf ca ?ca)
7.     (rf-create-ok-times ?ok-times1 ?ok-times2)
8.     (node-all-ones-value ?data-in ?all-ones-value)
9.     (node-checkerboard-value ?data-in 0 ?checkerboard0)
10.    (node-checkerboard-value ?data-in 1 ?checkerboard1)
11.    (declare-fragment
12.   (loose-sequence
13.     (test-phase (:component ?rf
14.                  :comment "Detect address line stuck-ats")
15.       (no-implicit-assignments (:protected (?rf) :ok-times ?ok-times1)
16.         (tight-sequence
17.           (comment "Fill the register file with ones"
18.             (multi-stream ((address (iota 0 15)))
19.               (equal-instants
20.                 (control ?data-in ?all-ones-value)
21.                 (control ?ca address))))
22.           (bounded-subgoal
23.             (rf-observe-cells-1 ?rf ?ok-times1 :START :FINISH)))))
24.     (test-phase (:component ?rf
25.                  :comment "Detect address line bridges (assuming wired-AND faults)")
26.       (no-implicit-assignments (:protected (?rf) :ok-times ?ok-times2)
27.         (tight-sequence
28.           (comment "Fill the register file with ones"
29.             (multi-stream ((address (iota 0 15)))
30.               (equal-instants
31.                 (control ?data-in ?all-ones-value)
32.                 (control ?ca address))))
33.           (bounded-subgoal
34.             (rf-observe-cells-2 ?rf ?ok-times2 :START :FINISH)))))
35.     (test-phase (:component ?rf
36.                  :comment "Check that each register cell can hold a checkerboard")
37.       (multi-stream ((address (iota 0 15)))
38.         (tight-sequence
39.           (equal-instants
40.             (control ?ca address) (control ?data-in ?checkerboard0))
41.           (equal-instants
42.             (control ?aa address) (observe ?out-a ?checkerboard0)))
43.         (tight-sequence
44.           (equal-instants
45.             (control ?ca address) (control ?data-in ?checkerboard1))
46.           (equal-instants (control ?aa address) (observe ?out-a ?checkerboard1)))))))))
```

Figure A.4: *The full component test for the register file: part 1*

```
47. (declare-stream 'ADDRESS-STUCK-AT-STREAM :value '(#b1110 #b1101 #b1011 #b0111))
48. (declare-stream 'ADDRESS-BRIDGING-STREAM :value '(#b1000 #b0100 #b0010 #b0001))

49. (define-predicate WRITE-AT-1111-READ-BACK
50.   ((write-at-1111-read-back ?rf (?write-time ?observation-time) ?s ?f)
51.    (node-all-ones-value ?ca ?address)
52.    (write-read-back ?rf ?address ADDRESS-STUCK-AT-STREAM (?wtime ?otime) ?s ?f)))

53. (define-predicate WRITE-AT-0000-READ-BACK
54.   ((write-at-0000-read-back ?rf (?write-time ?observation-time) ?s ?f)
55.    (write-read-back ?rf 0 ADDRESS-BRIDGING-STREAM (?wtime ?otime) ?s ?f)))

56. (define-predicate WRITE-READ-BACK
57.   ((write-read-back ?rf ?ca-addr ?read-stream (?wtime ?otime) ?s ?f)
58.    (component-port ?rf ca ?ca)
59.    (component-port ?rf data-in ?data-in)
60.    (node-all-ones-value ?data-in ?data-in-all-ones)
61.    (or (and (component-port ?rf aa ?address-line)
62.             (component-port ?rf out-a ?data-output))
63.        (and (component-port ?rf ba ?address-line)
64.             (component-port ?rf out-b ?data-output)))
65.    (declare-bounded-fragment
66.      ?s ?f
67.      (tight-sequence
68.        (comment "Write a 0 at one address"
69.          (equal-instants
70.            (control ?ca ?ca-addr ?wtime)
71.            (control ?data-in 0 ?wtime)))
72.        (comment "Read other addresses to see if they were affected"
73.          (multi-stream ((address ?read-stream))
74.            (equal-instants
75.              (control ?address-line address ?otime)
76.              (observe ?data-output ?data-in-all-ones ?otime)))))))
```

Figure A.5: *The component test for the register file: part 2*

```
** TEST-PHASE (:COMPONENT RF)
** Detect address line stuck-ats
BEGIN
    ** Fill the register file with ones
    BEGIN
        FOR ADDRESS FROM 0 TO 15 DO
            BEGIN
                MBR-IN := 65535  MUX-A := 0  OP := ALU-NOOP  RF-CA := ADDRESS
                CLK := 0; CLK := 1;
            END
    END
    ** Write a 0 at one address
    RF-CA := 15  MBR-IN := 0  MUX-A := 0  OP := ALU-NOOP
    CLK := 0; CLK := 1;
    ** Read other addresses to see if they were affected
    ** DECLARE-STREAMS (ADDRESS-STUCK-AT-STREAM)
    BEGIN
        ARRAY ADDRESS-STUCK-AT-STREAM = [14, 13, 11, 7];
        FOR INDEX10 FROM 0 TO 3 DO
            BEGIN
                RF-BA := ADDRESS-STUCK-AT-STREAM[INDEX10]  B-BUS = 65535
                CLK := 0; CLK := 1;
            END
    END
END
```

Figure A.6: *The RF Test Embedding: Part 1. This portion of the test detects stuck ats on the register file address inputs. I have moved node assignments that occur at the same time onto the same line to save space.*

```
** TEST-PHASE (:COMPONENT RF)
** Detect address line bridges (assuming wired-AND faults)
BEGIN
    ** Fill the register file with ones
    BEGIN
        FOR ADDRESS FROM 0 TO 15 DO
            BEGIN
                MBR-IN := 65535  MUX-A := 0  OP := ALU-NOOP  RF-CA := ADDRESS
                CLK := 0; CLK := 1;
            END
    END
    ** Write a 0 at one address
    RF-CA := 0  MBR-IN := 0  MUX-A := 0  OP := ALU-NOOP
    CLK := 0; CLK := 1;
    ** Read other addresses to see if they were affected
    ** DECLARE-STREAMS (ADDRESS-BRIDGING-STREAM)
    BEGIN
        ARRAY ADDRESS-BRIDGING-STREAM = [8, 4, 2, 1];
        FOR INDEX11 FROM 0 TO 3 DO
            BEGIN
                RF-BA := ADDRESS-BRIDGING-STREAM[INDEX11]  B-BUS = 65535
                CLK := 0; CLK := 1;
            END
    END
END
```

Figure A.7: *The RF Test Embedding: Part 2. This portion of the test detects bridge faults on the register file address inputs.*

```
** TEST-PHASE (:COMPONENT RF)
** Check that each register cell can hold a checkerboard
BEGIN
    CLK := 0; CLK := 1;
    FOR ADDRESS FROM 0 TO 15 DO
        BEGIN
            RF-CA := ADDRESS  MBR-IN := 43690  MUX-A := 0  OP := ALU-NOOP
            CLK := 0; CLK := 1;
            RF-AA := ADDRESS
            CLK := 0; CLK := 1;
            B-BUS = 43690  RF-BA := 0  RF-CA := 0  OP := ALU-NOOP  MUX-A := 1
            CLK := 0; CLK := 1;
            RF-CA := ADDRESS  MBR-IN := 21845  MUX-A := 0  OP := ALU-NOOP
            CLK := 0; CLK := 1;
            RF-AA := ADDRESS
            CLK := 0; CLK := 1;
            B-BUS = 21845  RF-BA := 0  RF-CA := 0  OP := ALU-NOOP  MUX-A := 1
            CLK := 0; CLK := 1;
        END
END
```

Figure A.8: *The RF Test Embedding: Part 3. This portion of the test applies a checkerboard pattern to each register. That is, it checks that 1010101010101010 and 0101010101010101 can be loaded and read from each register.*

A.3 The ROM Test

This example contrasts propagating a scalar value with propagating a stream. The circuit in figure A.9 is part of the MAC-1 microsequencer. The task is to test the ROM by reading the contents of all address locations.

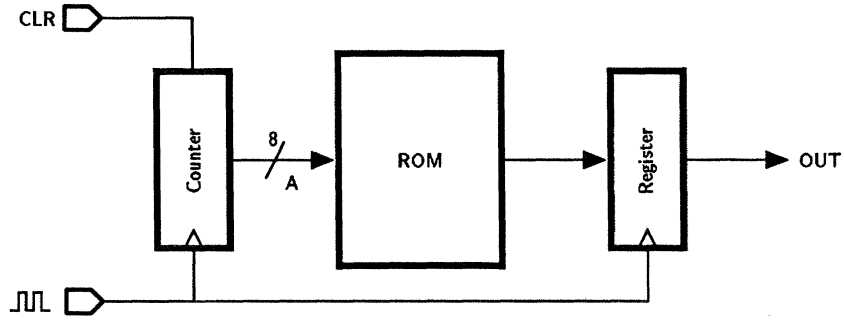


Figure A.9: *A portion of the MAC-1 microsequencer*

A.3.1 Solution #1

Propagating a scalar value backwards from A yields this solution: clear the counter, then count up to the value of A. This solution takes time proportional to the value of A, and using it repeatedly for a range of different values of A from 0 to n takes $O(n^2)$ time. Repeatedly initializing the counter makes this solution unnecessarily slow. Rules for this example are in figure A.10 and the solution is in figure A.11.

A.3.2 Solution #2

PF-TG finds a solution that initializes the counter only once by propagating a typed stream back from A. The component test in figure A.12 creates a subgoal to supply an exhaustive stream to the ROM address input. The propagation rule in figure A.13 responds to the supply subgoal by generating a counting stream which exhaustively covers the addresses. The component test also sets up another subgoal to observe the data coming out of the ROM and forces the input and output streams to move concurrently. This solution takes $O(n)$ time to run on a tester.

```
1.  (define-predicate ROM-TEST
2.    ((rom-test ?rom)
3.     (component-port ?rom addr ?addr)
4.     (component-port ?rom data ?data)
5.     (declare-fragment
6.       (test-phase (:component ?rom
7.                    :comment "This version uses scalar assignments")
8.         (loop (value 0 255)
9.           (equal-instants
10.            (control ?addr VALUE)
11.            (observe ?data (ROM VALUE))))))))

    ;;; Part of the rule for controlling a counter's output.
12. (define-propagation-rule COUNTER control-stream
13.   ((counter-control-stream ?counter out (REPEAT ?value ?synch) ?output-time)
14.    (component-port ?counter clr ?clr)
15.    (extract-synch-loop-times ?synch ?loop-s ?body-s ?body-f ?loop-f)
16.    (declare-bounded-fragment
17.      ?body-s ?body-f
18.      (tight-sequence
19.        (control ?clr CLEAR)
20.        (control ?clr COUNT)
21.        (protect (?clr)
22.           (multi-stream ((count (iota 1 ?value)))))
23.        (bounded-subgoal
24.          (t= ?output-time :start))))))
```

Figure A.10: *The first rule describes one way to test a ROM. This method counts through the addresses and asks for each address to be supplied as a scalar. The second rule shows a third of the real rule for controlling a counter's output. This part supplies a scalar value that is being asked for repeatedly. The part that supplies a stream appears in figure A.13, and the final part which supplies a single scalar is not shown.*

```
1.    ** TEST-PHASE (:COMPONENT ROM)
2.    ** This version uses scalar assignments
3.    BEGIN
4.        FOR VALUE FROM 0 TO 255 DO
5.            BEGIN
6.                CLK := 0; CLK := 1;
7.                CLR := CLEAR
8.                CLK := 0; CLK := 1;
9.                CLR := COUNT
10.               CLK := 0; CLK := 1;
11.               FOR COUNT FROM 1 TO VALUE DO
12.                   BEGIN
13.                       CLK := 0; CLK := 1;
14.                   END
15.               CLK := 0; CLK := 1;
16.               OUT = (ROM VALUE)
17.               CLK := 0; CLK := 1;
18.           END
19.   END
```

Figure A.11: *An inefficient solution to the ROM test. Note that this solution has one loop nested inside another. The outer loop steps through the ROM addresses. The body of this loop clears the counter on line 7, lets it count up to the address (*`value`*) on lines 9-14 and observes the output at line 16. I did not give PF-TG the contents of this ROM, so the value observed (*`(ROM VALUE)`*) is an unevaluated function.*

```
(define-predicate ROM-TEST
  ((rom-test ?rom)
   (component-port ?rom addr ?addr)
   (component-port ?rom data ?data)
   (node-width ?addr ?addr-width)
   (declare-fragment
     (test-phase (:component ?rom
                  :comment "exhaustively check the contents of each address")
       (with-synch nil
          (equal-instants
            (control-stream ?addr (stream (EXHAUSTIVE ?addr-width)))
            (observe-stream ?data (stream ROM-CONTENTS-STREAM)))))))))
```

Figure A.12: *A ROM component test using streams*

```
;;; This is the part of a more complex rule that is relevant to this example.
;;; It says that a ?num bit wide counting stream exhaustively covers all
;;; patterns of ?num bits.

(define-predicate COMPATIBLE-STREAM-PROPERTY
  ((compatible-stream-property (EXHAUSTIVE ?num) (COUNTING-STREAM ?num))))

;;; This is the backward propagation rule for a COUNTER (for generating streams).

(define-propagation-rule COUNTER control-stream
  ((counter-control-stream ?counter out (?stream ?stream-property ?synch) ?output-time)
   (component-port ?counter out ?out)
   (node-width ?out ?out-width)
   (compatible-stream-property ?stream-property (COUNTING-STREAM ?out-width))
   (extract-synch-loop-times ?synch ?loop-s ?body-s ?body-f ?loop-f)
   (new-time ?clr-time)
   (new-time ?start-counting)
   (t=-offset ?clr-time ?start-counting 1)
   (t< ?start-counting ?loop-s)
   (t=-offset ?body-s ?body-f 1)      ;Constrain the loop body = 1 clock cycle
   (t=-offset ?loop-s ?loop-f 256)
   (control-port-scalar ?counter CLR CLEAR ?clr-time)  ;CLEAR and COUNT are mnemonics
   (control-port-scalar ?counter CLR COUNT ?start-counting)))
```

Figure A.13: *Component rules for the ROM example. The first rule is a simplified version of the rule that relates stream properties to stream types. In this case, it says that a counting stream that is n-bits wide exhaustively covers all values on an n-bit node. The propagation rule first clears the counter, and then lets it count.*

```
1.   ** TEST-PHASE (:COMPONENT ROM)
2.   ** exhaustively check the contents of each address
3.   BEGIN
4.       CLR := CLEAR
5.       CLK := 0; CLK := 1;
6.       CLR := COUNT
7.       CLK := 0; CLK := 1;
8.       ** DECLARE-STREAMS (ROM-CONTENTS-STREAM)
9.       BEGIN
10.          ARRAY ROM-CONTENTS-STREAM = [];
11.          FOR INDEX8 FROM 0 TO 255 DO
12.              BEGIN
13.                  OUT = ROM-CONTENTS-STREAM[INDEX8]
14.                  CLK := 0; CLK := 1;
15.              END
16.      END
17.  END
```

Figure A.14: *An efficient ROM test. This version clears the counter in line 4 and then lets it count, observing the ROM outputs on line 13.*

A.4 The ROM Test with a BILBO Register

This example is similar to the last except that the normal register has been replaced by a BILBO register, i.e. a linear feedback shift register that can generate a single signature value summarizing a long stream of values. The signature has the property that single bit errors in the value stream will yield a "wrong" signature with high probability. Using a BILBO register, long streams of values can be compressed inside the circuit and the summary checked easily by the tester. The propagation rule for the BILBO register is in figure A.15 and resulting solution is in figure A.16.

```
1.  (define-propagation-rule BILBO-REGISTER observe-stream
2.    ((bilbo-register-observe-stream ?bilbo IN (?stream ?stream-property ?synch) ?output-time
3.     (extract-synch-loop-times ?synch ?loop-s ?body-s ?body-f ?loop-f)
4.     (compute-bilbo-signature ?stream ?stream-property ?signature)
5.     (new-time ?clr-time)
6.     (new-time ?start-recording)
7.     (new-time ?observe-signature)
8.     (t=-offset ?clr-time ?start-recording 1)
9.     (t=-offset ?start-recording ?loop-s 1)
10.    (t=-offset ?body-s ?body-f 1)          ;Constrain the loop body to 1 clock cycle
11.    (t=-offset ?loop-f ?observe-signature 1)
12.    (control-port-scalar ?bilbo CLR CLEAR ?clr-time)
13.    (control-port-scalar ?bilbo CLR RECORD ?start-recording)
14.    (observe-port-scalar ?bilbo OUT ?signature ?observe-signature)
15.    ))
```

Figure A.15: *This rule describes how to observe a stream of values using a BILBO register. The* `compute-bilbo-signature` *subgoal on line 4 tries to pre-compute the value of the signature. In this case, because I did not give PF-TG the contents of the ROM, this subgoal returns an function rather than an actual value.*

```
1.   ** TEST-PHASE (:COMPONENT ROM)
2.   ** exhaustively check the contents of each address
3.   BEGIN
4.       CLR1 := CLEAR
5.       CLR2 := CLEAR
6.       CLK := 0; CLK := 1;
7.       CLR1 := COUNT
8.       CLR2 := RECORD
9.       CLK := 0; CLK := 1;
10.      WAIT 257;
11.      OUT = (SIGNATURE (ROM VALUE))
12.  END
```

Figure A.16: *A solution for the ROM test using a BILBO register: this test program uses the counter to apply an exhaustive stream to the ROM address input and observes the data output stream using the BILBO register. Lines 4-8 initialize the clock and the BILBO register and start them going. Line 10 is equivalent lines 11-14 in figure A.11. That was an explicit loop that did nothing but drive the clock, and* `wait` *is a shorthand way of doing the same thing. The output value in line 11 is an unevaluated function of the ROM contents. PF-TG would replace this function with the actual signature if it the ROM contents were available.*

Bibliography

[abadir85] M. Abadir and M. Breuer. A Knowledge-Based System for Designing Testable VLSI Chips. *IEEE Design & Test of Computers*, pages 56–68, 1985.

[allen-cacm] J. Allen. Maintaining Knowledge About Temporal Intervals. *CACM*, 26(11):832–843, 1983.

[barrow83] H. G. Barrow. Proving the Correctness of Digital Hardware Designs. In *Proceedings of the National Conference on Artificial Intelligence*, pages 17–21. AAAI, August 1983.

[Barstow79] D. R. Barstow. *Knowledge Based Program Construction.* Elsevier North Holland, Inc., New York, NY, 1979.

[benmehrez83] C. Benmehrez and J. F. McDonald. The Subscripted D-Algorithm – ATPG With Multiple Independent Control Paths. In *Workshop on Simulation and Test Generation Environments*, pages 71–80. IEEE Computer Society and Test Technology Committee, March 1983.

[bennetts82] R. G. Bennetts. *Introduction to Digital Board Testing.* Crane Russak, New York, NY, 1982.

[bowden75] K. Bowden. A Technique for Automatic Test Generation for Digital Circuits. In *Proc. IEEE Intercon*, pages 1–5, 1975.

[brahme85] D. Brahme. Functional Testing of Microprocessors. Master's thesis, Coordinated Science Laboratory, College of Engineering, University of Illinois, January 1985.

[brahme87] D. Brahme and J. Abraham. Knowledge Based Test Generation for VLSI Circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 192–295, 1987.

[breuer79] M. Breuer. New Concepts in Automated Testing of Digital Circuits. In G. Musgrave, editor, *Computer Aided Design of Digital Electronic Circuits and Systems*, pages 57–80. North-Holland Publishing Company, Brussels and Luxembourg, 1979.

[breuer76] M. Breuer and A. Friedman. *Diagnosis and Reliable Design of Digital Systems.* Computer Science Press, Inc., Rockville, Maryland, 1976.

[bryant83] R. Bryant and M. Schuster. Fault Simulation of MOS Digital Circuits. *VLSI Design*, pages 14–71, October 1983.

[carbonell85] J. Carbonell. Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. Technical Report CMU-CS-85-115, Carnagie-Mellon University Department of Computer Science, March 1985.

[chandra87] S Chandra and J. Patel. A Hierarchical Approach to Test Vector Generation. In *Proceedings of the 24th ACM/IEEE Design Automation Conference.* ACM, June 1987.

[chapman85] D. Chapman. Planning for Conjunctive Goals. Technical Report AI-TR-802, MIT Artificial Intelligence Laboratory, November 1985.

[daniels85] R. Daniels and W. Bruce. Built-in self-test trends in Motorola Microprocessors. *IEEE Design & Test of Computers*, pages 14–71, April 1985.

[davis82a] R. Davis. Diagnosis Based on Description of Structure and Function. In *Proceedings of the National Conference on Artificial Intelligence.* AAAI, August 1982.

[dekleer-ATMS86a] J. de Kleer. An Assumption-based Truth Maintenance System. *Artificial Intelligence*, 28, 1986.

[dean87] T. Dean and D. McDermott. Temporal Data Base Management. *Artificial Intelligence*, 32(1):1–56, April 1987.

[ferguson87] F.J. Ferguson and J. Shen. Extraction and Simulation of Realistic CMOS Faults using Inductive Fault Analysis. *x*, pages x–x, 1987.

[fikes71] R. Fikes and N. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:198–208, 1971.

[foyster84] G. Foyster. Helios: User's Manual. Technical Report HPP-84-34, Stanford University Heuristic Programming Project, July 1984.

[fujiwara85] H. Fujiwara. *Logic Testing and Design for Testability.* The MIT Press, Cambridge, MA, 1985.

[garey79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, NY, 1979.

[genesereth81] M. Genesereth. The Use of Hierarchical Design Models in the Automated Diagnosis of Computer Hardware Faults. Technical Report HPP-81-20, Stanford University Heuristic Programming Project, December 1981.

[genesereth84] M. Genesereth. The Use of Design Descriptions in Automated Diagnosis. *Artificial Intelligence*, 24(3):411–436, December 1984.

[goel81a] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Trans. Comput.*, C-30 (3):215–222, 1981.

[goel81b] P. Goel. PODEM-X: An Automatic Test Generation System for VLSI Logic Structures. In *Proceedings of the 18th Design Automation Conference*, pages 260–268, 1981.

[gray69] P. Gray and C. Searle. *Electronic Principles Physics, Models, and Circuits.* John Wiley and Sons, Inc., New York, NY, 1969.

[hamscher83] W. Hamscher. Using Structural and Functional Information in Diagnostic Design. Technical Report 707, MIT Artificial Intelligence Lab, June 1983.

[hill77] F. Hill and B. Huey. SCIRTSS: A Search System for Sequential Circuit Test Sequences. *IEEE Transactions on Computers*, pages 490–502, May 1977.

[ibarra75] O.H. Ibarra and S.K. Sahni. Polynomially Complete Fault Detection Problems. *IEEE Transactions on Computers*, Vol. C-24:242–249, 1975.

[joyce83] R. Joyce. Reasoning about Time-dependent Behavior in a System for Diagnosing Digital Hardware Faults. Technical Report HPP-83-37, Stanford University Heuristic Programming Project, August 1983.

[kahn83] K. Kahn and M. Carlsson. *LM-Prolog User Manual: Release 1.0.* Uppsala Programming Methodology and AI Laboratory, Dept. of Computer Science, Uppsala University, October 1983.

[kelly82] V. Kelly and L. Steinberg. The CRITTER System– Analyzing Digital Circuits by Propagating Behaviors and Specifications. In *Proceedings of AAAI-82*, pages 284–289. American Association for Artificial Intelligence, August 1982.

[khorram84] R. Khorram. Functional Test Pattern Generation for Integrated Circuits. Master's thesis, MIT Department of Electrical Engineering and Computer Science, January 1984.

[kramer83] G. Kramer. Test Generation Algorithms: Overcoming the Von Neumann Bottleneck. In *Workshop on Simulation and Test Generation Environments*, pages 86–95. IEEE Computer Society and Test Technology Committee, March 1983.

[kramer84] G. Kramer. Brute Force and Complexity Management: Two Approaches to Digital Test Generation. Master's thesis, MIT Department of Electrical Engineering and Computer Science, May 1984.

[kramer85] G. Kramer. Representing and reasoning about designs. In H. Yoshikawa and E. A. Warman, editors, *Proceedings of IFIP W.G.5.2 Working Conference on Design Theory for CAD*, pages 59–94, Tokyo, October 1985. IFIP.

[krishnamurthy87] B. Krishnamurthy. Hierarchical Test Generation: Can AI Help? In *International Testing Conference 1987 Proceedings*, pages 694–700. The Computer Society of the IEEE, 1987.

[lai81] K. Lai. *Functional Testing of Digital Systems.* PhD thesis, Carnegie-Mellon University, December 1981.

[lai83] K. Lai. Functional Testing of Digital Systems. In *Proceedings of the 20th Design Automation Conference*, pages 207–213. ACM, 1983.

[leiserson84] C. E. Leiserson and J. B. Saxe. A Mixed-Integer Linear Programming Problem Which is Efficiently Solvable. Technical Report VLSI-84-216, Department of EE and CS, Massachusetts Institute of Technology, 1984.

[lindsay80] R. Lindsay, B. Buchanan, E. Feigenbaum, and J. Lederberg. *DENDRAL.* McGraw-Hill, New York, 1980.

[maly86] A. Strojwas Maly, W. and S. Director. VLSI Yield Prediction and Estimation: A Unified Framework. *IEEE Transactions on Computer-Aided Design*, CAD-5(1):114–120, January 1986.

[marlett86] R. Marlett. An Effective Test Generation System for Sequential Circuits. In *Proceedings of the 23rd Design Automation Conference*, pages 250–256. IEEE, 1986.

[mead80] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Welsey Publishing Company, Inc., Philippines, 1980.

[miller85] D. Miller. *Planning by Search Through Simulations.* PhD thesis, Yake University Department of Computer Science, October 1985.

[mitchell86] T. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-Based Generalization: A Unifying View. *Machine Learning*, 1(1), 1986.

[rawat87] S. Rawat and M. Irwin. C-Testability of Unilateral and Bilateral Sequential Arrays. In *International Testing Conference 1987 Proceedings*, pages 181–188. The Computer Society of the IEEE, 1987.

[robinson83] G. Robinson. HITEST - Intelligent Test Generation. In *Proceedings, 1983 International Test Conference*, pages 311–323. IEEE, October 1983.

[roth66] J. P. Roth. Diagnosis of Automata Failures: A Calculus and a Method. *IBM Journal of Research and Development*, 10:278–291, July 1966.

[roth80] J. P. Roth. *Computer Logic, Testing, and Verification.* Computer Science Press, Inc., Rockville, Maryland, 1980.

[russell85] S. Russell. The Compleat Guide to MRS. Technical Report KSL-85-12, Stanford Knowledge Systems Laboratory, 1985.

[sacerdoti77] E. Sacerdoti. *A Structure for Plans and Behavior.* Elsevier North-Holland, Inc., New York, NY, 1977.

[sargent83] B. Sargent. Implementation of a Memory-Emulation Diagnostic Technique. In *International Testing Conference 1983 Proceedings.* The Computer Society of the IEEE, 1983.

[sarkany87] E. Sarkany and W. Hart. Minimal Set of Patterns to Test RAM Components. In *International Testing Conference 1987 Proceedings*, pages 759–764. The Computer Society of the IEEE, 1987.

[shen85a] W. Maly Shen, J. and F. Ferguson. Inductive Fault Analysis of nMOS and CMOS Integrated Circuits. Research Report CMUCAD-85-51, Carnegie-Mellon University, August 1985.

[shirley85] M. Shirley. An Automatic Programming Approach to Testing. In *Workshop on Simulation and Test Generation Environments.* IEEE Computer Society and Test Technology Committee, September 1985.

[shirley86] M. Shirley. Generating Tests by Exploiting Designed Behavior. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, pages 884–890. American Association for Artificial Intelligence, August 1986.

[shirley83b] M. Shirley and R. Davis. Generating Distinguishing Tests Based on Hierarchical Models and Symptom Information. In *IEEE International Conference on Computer Design*, pages –. IEEE, October 1983.

[shirley87] M. Shirley, P. Wu, R. Davis, and G. Robinson. A Synergistic Combination of Test Generation and Design for Testability. In *International Testing Conference 1987 Proceedings*, pages 701–711. The Computer Society of the IEEE, 1987.

[singh83] N. Singh. MARS: A Multiple Abstraction Rule-Based Simulator. Technical Report HPP-83-43, Stanford University Heuristic Programming Project, December 1983.

[singh85] N. Singh. *Exploiting Design Morphology to Manage Complexity.* PhD thesis, Stanford Department of Electrical Engineering, April 1985.

[singh86] N. Singh. Saturn: An Automatic Test Generation System for Digital Circuits. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, pages 778–783. AAAI, August 1986.

[steele80] G. Steele, Jr. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* PhD thesis, MIT, August 1980.

[steele87] R. Steele. An Expert System Application in Semicustom VLSI Design. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 679–686. ACM, 1987.

[stefik80] M. Stefik. *Planning with Constraints.* PhD thesis, Stanford University Department of Computer Science, January 1980.

[sterling86] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* The MIT Press, Cambridge, Massachusetts and London, England, 1986.

[sussman75] G. J. Sussman. *A Computer Model of Skill Acquisition.* American Elsevier Publishing Company, Inc., New York, NY, 1975.

[tanenbaum84] A. S. Tanenbaum. *Structured Computer Organization.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

[tate75] A. Tate. Interacting Goals and Their Use. In *Proceedings of IJCAI-75*, pages 215–218, 1975.

[thatte80] S. Thatte and J. Abraham. Test Generation for Microprocessors. *IEEE Transactions on Computers*, Vol. C-29, No. 6:429–441, June 1980.

[valdes86] R. Valdés-Pérez. The Satisfiability of Temporal Constraint Networks. In *Proceedings of the National Conference on Artificial Intelligence*, pages 256–260, 1987.

[vere83] S. Vere. Planning in Time: Windows and Durations for Activities and Goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5, No. 3:246–267, May 1988.

[spice80] A. Vladimirescu and S. Liu. The Simulation of MOS Integrated Circuits using SPICE2. Technical Report UCB/ERL M80/7, Electronics Research Laboratory, Univ. of California, Berkeley, February 1980.

[warren74] D. Warren. WARPLAN: A System for Generating Plans. Technical Report Memo 76, University of Edinburgh, Department of Computational Logic, 1974.

[weise86] D. Weise. Formal Multilevel Hierarchical Verification of Synchronous MOS VLSI Circuits. Technical Report 978, MIT Artificial Intelligence Laboratory, August 1986.

[weld86] D. S. Weld. The Use of Aggregation in Qualitative Simulation. *Artificial Intelligence*, 30(1):1–34, October 1986.

[williams79] T. W. Williams and K. P. Parker. Testing Logic Networks and Design for Testability. *IEEE Computer*, 9(21), October 1979.

[williams83] T. W. Williams and K. P. Parker. Design For Testability – A Survey. *Proceedings of the IEEE*, 71(1), January 1983.

[williams87] W. Williams. An Automatic Test Generator for Programmable Logic Devices. In *International Testing Conference 1987 Proceedings*, pages 658–667. The Computer Society of the IEEE, 1987.

[wu88] Peng Wu. Test Generation Guided Design for Testability. Master's thesis, Massachusetts Institute of Technology, May 1988.

[zhu86] Xi-an Zhu. *A Knowledge Based System for Testable Design Methodology Selection.* PhD thesis, University of Southern California, August 1986.