Technical Report 1051

Test Generation Guided Design for Testability

Peng Wu

MIT Artificial Intelligence Laboratory

Test Generation Guided Design for Testability

by

Peng Wu

B.E., Tsinghua University

1982

Submitted to the

Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Science

at the

Massachusetts Institute of Technology

May, 1988

Signature of Author [illegible]

Department of Electrical Engineering and Computer Science
May 5, 1988

Certified by [illegible]

Randall Davis
Associate Professor of Management Science
Thesis Supervisor

Accepted by ________________________

Arthur C. Smith, Chairman
Committee on Graduate Students

Test Generation Guided Design for Testability

by

Peng Wu

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

Abstract

This thesis presents a new approach to building a design for testability (DFT) system. The system takes a digital circuit description, finds out the problems in testing it, and suggests circuit modifications to correct those problems.

The key contributions of the thesis research are: (1) setting design for testability in the context of test generation (TG), (2) using failures during TG to focus on testability problems, and (3) relating circuit modifications directly to the failures. A natural functionality set is used to represent the maximum functionalities that a component can have.

Although knowledge about TG is critical to a competent DFT system, this knowledge has not been used by previous DFT systems. Our approach introduces TG knowledge into a DFT system by following the principle of repairing TG failures. Using TG knowledge has two advantages. First, our DFT system focuses its efforts on areas where testability problems actually arise and introduces modifications only in those areas. Thus our system eliminates some false testability problems and the associated DFT overhead that other more structured, less focused techniques often introduce. Second, our approach integrates different DFT techniques according to the need for repairing TG failures, and thus obtains even lower DFT modification overhead. By comparison, previous DFT approaches combine DFT techniques only in the same categories due to lack of TG knowledge.

The current implementation has only primitive domain knowledge and needs other work as well. However, armed with the knowledge of TG, it has already demonstrated its ability and produced some interesting results on a simple microprocessor.

Keywords: Artificial Intelligence, Knowledge-based System, Knowledge Representation, VLSI, Design for Testability, Test Generation.

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

Acknowledgments

I would like to thank:

Randall Davis, my thesis supervisor, for his guidance, encouragement and many ideas on technical, presentation and research issues. Without him, this thesis would not have been possible.

Mark H. Shirley, my fellow student, for encouragement, many discussions and suggestions, on the content and the presentation of thesis-related work. Without his help, this thesis would not be the same.

Walter Hamscher, for generous help and keeping HTVAX alive.

Raul Valdes-Perez, for generous help, advice, and friendship.

Gordon Robinson, of GenRad, for discussions on testing.

Reid Simmons, Brian Williams, Choon P. Goh, and Jeff Van Baalen, for many suggestions and comments.

Cristina Ciro, Sandiway Fong, Guangrong Gao, William Lim, Yang Lee, and Lixia Zhang, for encouragement, advice, and friendship.

The MIT AI Lab and the Hardware Troubleshooting group, for providing an excellent research environment.

All the people at the MIT Writing Center, for reviewing the entire draft of the thesis and improving the presentation.

My parents, for constant encouragement, unconditional sacrifice and support.

This thesis describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the author's artificial intelligence research is provided by Chinese Government, the Digital Equipment Corporation, and Wang Corporation.

Contents

Chapter 1

Introduction

1.1 DFT: Modifying Circuits for Easy Testing

This thesis presents a new approach to building a design for testability (DFT) system, and an implemented system based on this approach. The system can be viewed as a DFT adviser to a function designer so that he can concentrate on more important design issues. It takes a digital circuit description, finds out the problems in testing it, and suggests circuit modifications to correct those problems. The key contributions of the thesis research are: (1) setting design for testability in the context of test generation (TG) , (2) using failures during TG to focus on testability problems, and (3) relating circuit modifications directly to the failures. A natural functionality set is used to represent the maximum functionalities that a component can have.

In addition to the primary goal of implementing a desired behavior, a digital circuit design must meet secondary goals relating to its physical realization. Testability, i.e., ensuring that improperly manufactured circuits can be identified by testing them, is such a secondary goal.

Testing plays a very important role in circuit design verification, quality control, reliability improvement, and maintenance. For instance, in VLSI chip manufacturing processes, testing prevents defective intermediate or final products from passing through the rest of the production steps or getting shipped out, and it helps improve yield by revealing process defects.

The purpose of testing is to verify the functionalities of a circuit. Testing has two integrated parts: test generation and test application. Test generation takes a description of a circuit, calculates the signals for exercising it in order to verify its functionality, and predicts the correct responses. Test application applies the generated test to the circuit and checks whether there is any discrepancy between the predicted and the actual behaviors.

As VLSI technology has advanced, the number of I/O pins has not kept pace with the increased number of transistors per chip. As a result, the chip's internal nodes are farther from the I/O pins and are less easily controlled and observed. Testing is hampered by the fact that it often takes a long time to plan a method to control or observe an internal node and testing methods often take a long time to execute. For some chips, testing can be as high as 60% of total chip production cost. Excessively high cost in test generation or test application for VLSI chips is the *testability problem.*

There are two responses to this problem. The first response is to develop more powerful test generation (TG) methods so that inexpensive tests can be generated efficiently. The second response is to employ design for testability (DFT) techniques so that circuits can be designed as easy to test in the first place. Our research is basically in the DFT camp.

DFT techniques incorporate some testing facilities into a chip design so that a satisfactory test can be derived easily and the test will be inexpensive to carry out. This is essentially a process of making trade-offs among cost, performance and quality of a VLSI chip. Due to the complexity of the circuits involved, design for testability is challenging.

1.2 An Example: What the System is Doing

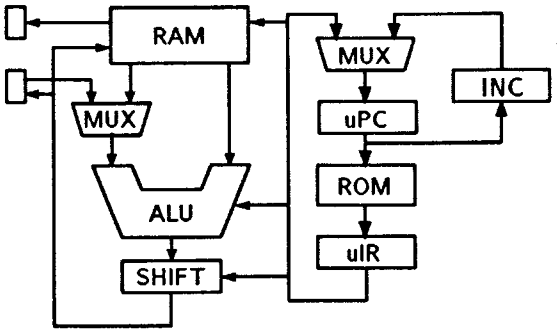


Figure 1.1: MAC-2: a microprocessor

Figure 1.1 shows a textbook microprocessor. The microprogram ROM is considered difficult to test because there is no direct access to this part of the circuit from the chip's pins. Our system suggests five ways to make the ROM testable; one of them is shown in Figure 1.2: (a) add a testing mode, (b) use the micro-PC (uPC), the MUX and the INCrementer together as a counter during testing to provide a counting sequence to the address of the ROM, and (c) make the micro instruction register uIR a shift register during testing so that the contents of the ROM can be shifted off-chip for verification.

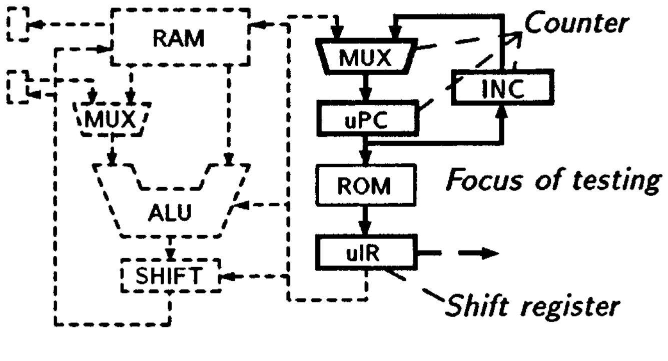


Figure 1.2: MAC-2: A way to make the ROM testable

The system's overall approach to finding and fixing testability problems in a circuit is a four-step process:

1. It runs a state-of-the-art test generator [19] on the circuit to identify the untestable components[1].
2. It further refines the testability problem by attempting to generate tests for the untestable components.
3. When it encounters a test generation failure, it selects a modification according to the nature of the failure.
4. Finally, the system modifies the circuit and repeats the process until all untestable components are processed.

This system is oriented towards fabrication-screen and static-logic testing for VLSI chips as complex as microprocessors. Fabrication-screen testing is intended to separate working chips from defective ones. Static-logic testing is concerned with digital logic behaviors represented at the gate and register transfer level, not analogue behaviors such as I/O voltage, or dynamic behaviors such as response time.

1.3 The Approach: Test Generation Guided DFT

A central concern for DFT is the chip performance and the production yield degradations resulting from DFT-modifications. The additional circuitry required by DFT modifications slows down a chip; the additional chip area lowers the production yield. In order to reduce these degradations, first a DFT system must locate testability problems precisely so that it will not introduce redundant modifications, or equivalently, will not introduce degradations without a sound reason. Second, the DFT system must be able to find cost-effective solutions to testability problems so that it will reduce the DFT-modification penalty.

[1]In the current implementation the interface between the test generator and the rest of the system is manual.

The second major concern is the size of the search space which a DFT process faces. There can be different kinds of testability problems in a given chip; there may be different design-modifications available to deal with them; and there are many subtle trade-offs among area overhead, performance penalty, redesign cost, test generation cost, test equipment cost, etc. Briefly, DFT identifies testability problems in a given chip design and looks for a cost-efficient global solution to them.

The approach developed in my thesis research

- pinpoints circuit testability problems in the process of test generation with a cooperating test generator
- selects design-modifications according to the testability problems they can fix
- reduces the search space by using high level circuit representation and some engineering heuristics.

One of the contributions of this research is to use a TG in a DFT system to find testability problems. This approach is inspired and justified by the following testability definition:

A circuit is testable if we can successfully generate a test for it.

This very simple definition is a radical departure from the definitions used in current testability measurements and other DFT systems. Its major advantage is that it helps *locate* and *repair* testability problems since it is much less ambiguous about whether a testability problem *exists*, and when there is a testability problem it shows *where* the problem is and why it happened.

In contrast, existing testability measuring methods calculate circuit statistics which are only highly *correlated to the testability* of the circuit. For example, one of these methods calculates the probability of a circuit node being driven to logic 1/0 and will signal a "testability problem" if the probability is lower than a certain threshold. However, the probability does not correspond well to the testability problem since no matter how low the probability is there is no testability problem as long as we can find a way to drive the circuit node to logic 1/0 and vice versa.

Disappointed by these uninformative testability measures, researchers have previously used *heuristic* testability definitions in their DFT systems. These heuristic definitions assume some specific predetermined test generation method and can be easily verified using mainly the knowledge of circuit connectivity. However, these definitions sometimes result in false testability problems. For instance, the LSSD (level sensitive scan design) design rule approach [10] defines testability problems as design rule violations. Since LSSD reduces sequential circuit testing to combinational circuit testing, this approach assumes that the test generator is only able to handle combinational circuits. As a result it finds testability "problems" that are not real problems. For example, according to the LSSD design rule approach, every register in the datapath part of the MAC-2 must be made into a shift register, despite the fact that existing test generators [19] can test these parts as is.

Using heuristic testability definitions may also leave a DFT system with fewer alternative solutions or may require rigid combination of modifications. For instance, the LSSD design rule approach would make the uPC a shift register to conform to the rule that every register in a circuit should be turned into a shift register. Though this enables the uPC to provide the counting sequence for the ROM, it will never find the less expensive solution suggested by our system[2].

Human experts, by comparison, are able to locate the hard-to-test spots in a circuit and then introduce "just enough" hardware to make the circuit testable. One reason that accounts for this phenomenon is that human experts understand circuit and testing much better than DFT systems do, which leads to more precise usage of DFT-modifications.

Our approach uses ideas from test generation to define and locate testability problems. This is an attempt to introduce TG knowledge into a DFT system and to

[2]Using the MUX, the uPC and the INC together as a counter requires special control signals, but is less expensive than making uPC a shift register, which requires both special control and component modification.

do sharply focused design for testability by having the DFT system cooperate with test generators. With the help of the TG knowledge the system is able to examine the circuit behaviors in testing. Although currently its design modification knowledge is relatively sparse, the system armed with TG knowledge has already demonstrated an ability to generate suggestions acceptable to a human expert.

Our approach has two advantages over previous approaches to DFT. First, it uses a test generator to locate *true* testability problems. This tends to minimize the area and performance overhead necessary to achieve testability because it avoids adding testability features to the portions of circuit that a test generator can already handle. Second, modifications can be combined in any way, subject only to the need to fix testability problems. This gives the system more alternatives to fix a given problem and tends to reduce the resulting overhead.

Ideally, testability should be considered while implementing the functional requirements of a circuit. However, due to the scale of VLSI circuits, designs are seldom bug-free at the beginning. It is a common practice in design that the functional goals are designed first to restrict the search space to a manageable size. Then secondary goals are implemented by debugging, perturbing the tentative design as little as possible to maintain the primary goals. Our DFT system accomplishes a variety of "minimum perturbation" because it works only on "real" testability problems (as defined by the test generation failure) and because it has a library of design modifications indexed by failure type.

1.4 Thesis Outline

Chapter 2 elaborates the theoretical aspects of the proposed approach. §2.1 refines the testability definition as stated in this chapter and analyzes its suitability in view of the available TG and DFT technologies. §2.2 establishes the relationship between test generation failures and their DFT repairs. §2.2.4 describes the desired properties of the test generator which can produce failures that are readily fixed. §2.3 situates the proposed system in the whole VLSI design process. §2.4 discusses our approach in relation with important testing issues such as fault modeling.

Chapter 3 presents the implementation of the proposed approach. §3.1 describes the system's structure. §3.2 describes the search process and the search control mechanism. §3.3 describes the knowledge and its representation in this system. §3.4 gives two example circuits used to test the system.

Chapter 4 discusses the related work and the limitations of our approach and possible directions of future efforts.

Chapter 2

TG Guided DFT

This chapter will describe the new approach to building a DFT system. §2.1 refines the testability definition as stated in the previous chapter and analyzes its suitability in view of the available TG and DFT technologies. §2.2 establishes the relationship between test generation failures and their DFT repairs. §2.2.4 describes the desired properties of the test generator which can produce failures that are readily fixed. §2.3 situates the proposed system in the whole VLSI design process. §2.4 discusses the new approach in relation with important testing issues such as fault modeling.

2.1 DFT: Establishing TG's Failing Goals

2.1.1 The Motivation: Cost Trade-off

Since every physical fault in a circuit which affects the circuit's behavior can be detected by exhaustively exercising the circuit, it is always possible in theory to guarantee complete coverage. The pragmatic question is cost. In order to detect faults in a circuit, people have to pay for test generation, test application, redesign for testability, yield and performance degradation due to "extra" testability features, etc. If manufacturers spend less on testing and let more faulty chips slip through, users have to pay for the damages these faulty chips could cause. The ratio of testing-related cost (including damage by undetected faulty circuits) to the total cost of the circuit is an appropriate testability indicator.

However, the trade-offs among various testing related costs are seldom made freely, due to the difficulty in estimating them before a design takes shape, and due to the costs in redesigning a complicated VLSI chip in order to accommodate the trade-offs. That is why designers usually make the trade-offs at the beginning of a design project based on their past experience, rather than based on the actual problems in the design, such as design rules or guidelines. Automated DFT may ease the redesign difficulty and enable the designers to explore more alternatives concerning testability.

2.1.2 A Definition of Testability

Definitions

The purpose of circuit testing is to verify circuit behaviors. The prevailing approach to circuit testing is to exercise a circuit in order to observe its behaviors.

Previously we have defined the testability of a circuit as successful test generation on the circuit. Here we will be more specific about how to generate tests. Thereafter, we will assume that circuits are tested by testing their components, a common practice of divide and conquer. Accordingly, the testability definition is refined as follows:

A component is testable if it has a satisfactory test; a circuit is testable if each of its components is testable.

In this thesis:

- a *focus* is the component being tested;
- a *test pattern* for a focus refers to the signals required to exercise it and the responses at its I/O ports which one must verify in order to test it satisfactorily;
- *primary I/O ports* are the chip I/O ports directly accessible during testing; and
- a *test* for a focus is the signals and responses at the primary I/O ports of the chip which cause the test pattern to happen.

Test pattern embedding is the process of finding the test given the test pattern, i.e., to propagate signals backward from the inputs of the focus to the primary inputs of the chip, called the *source* of the signal, and forward from the outputs of the focus to the primary outputs of the chip, called the *sink* of the response. The testability problems at the input side of the focus are the *controllability* problems; those at the output side are the *observability* problems. The *satisfactoriness* of a test can be decided by a threshold on a weighted sum about its fault coverage, fault resolution, length, the time needed to compute the test, the complexity of the test equipment needed to carry out the test, etc. This research is focused on test pattern embedding for the untestable components by modifying the circuit, but not on measuring the satisfactoriness of tests so developed.

About the Testability Definition

Though the testability definition seems straightforward, it captures the spirit of testing and works well. This definition has several advantages over the definitions used explicitly or implicitly in other DFT and testability measuring systems:

- Emphasizing the link between TG and DFT:

  TG technology and DFT technology have long been attacking the same testability problem. These two should be made allies to help each other towards testability. DFT has been making TG easier, but so far little help has been delivered from TG to DFT systems. It is obvious that the more powerful the TG methodology employed, the fewer the testability problems in a circuit. Furthermore, test generators can provide information about a circuit regarding testability issues to the DFT system, since it is the job of test generators to find a test for a circuit and since most testability problems can only be encountered in the process of TG.

  Our definition encourages the exchange of information between TG systems and the DFT system. It can be viewed as a specification for the interface from a TG to a DFT system. As long as it fits the interface, any TG can help the DFT system to locate test problems and reduce DFT computation and modifications. For instance, for our system, any TG can be of help if it can find testable components. Thus the DFT system will not waste energy on them and, better yet, will never introduce DFT overhead to make these components "more testable."

- Locating testability problems unambiguously:

  If there is a testability problem according to our definition, there is no reasonable way to test some components, so circuit design changes are needed. Otherwise we already know how to test the circuit so no design changes are needed. Thus unnecessary modifications are eliminated.

- Selecting suitable design modifications:

  We can further narrow in on the location of a testability problem by examining the reason for component untestability. The reason is called *test generation failure*, or *TG failure* for short, which is a failed goal in test generation. When we know exactly what kind of TG failure it is, we can select the most suitable design modification to correct it.

2.1.3 Achieving Controllability and Observability — the Fundamentals of TG and DFT

There is a variety of TG algorithms and methods, most of which try to achieve controllability and observability at various circuit description levels.

And there is a variety of DFT algorithms and methods, most of which can be viewed as providing controllability and observability at various circuit description levels.

Underlying both DFT and TG is the same fundamental task, that is, to achieve controllability and observability. The task is achieved in the case of TG by finding out how to run the given circuit *as is*; in the case of DFT it is achieved by finding out how a given circuit *can be changed* to run. These are the fundamentals, though there are many other extremely important issues to consider in real life design projects, such as the testing time, testing equipment, overhead on the circuit, etc. In building our system, we emphasize the fundamentals because they give much better guidance in *finding* the solution, not just in *evaluating* it.

This symmetry strongly suggests that controllability and observability are a natural hinge for TG and DFT working together synergistically to achieve low cost testability. This view of TG and DFT plays an important role in building our system and leads us to integrate a test generator into the DFT system.

The Fundamentals of TG

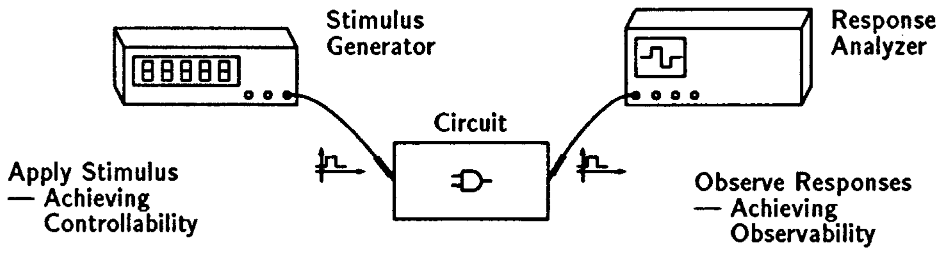

Stimulus Generator
Response Analyzer
Circuit
Apply Stimulus — Achieving Controllability
Observe Responses — Achieving Observability

Figure 2.1: Testing a Circuit

Test generation[1] can be divided into three steps: (1) figuring out how to test a circuit, i.e., figuring out the so-called test pattern — the stimulus and the expected responses which verify the behavior of the circuit; (2) figuring out how to apply the stimulus, i.e., how to control the circuit; (3) figuring out how to observe the responses.

When a circuit is tested as a black box, the challenging step is to figure out the test pattern; the other two steps are straightforward as illustrated in Figure 2.1.

When a circuit is complicated, information about the internal structure or the function of the circuit must be used to generate a short and effective test pattern.

[1]Here we are concerned only with logic testing, i.e., the test verifying the digital behavior of the circuits.

Stimulus		Expected Response
Input-1	Input-2	Output
0	0	0
0	1	0
1	1	1
1	0	0

Table 2.1: A Test Pattern for a Two-Input AND-Gate

For instance, in a chip containing four two-input AND-gates, there are eight inputs that need stimulating and four outputs that need observing. Using the knowledge that the chip contains four *unconnected* two-input AND-gates, the test pattern of the chip can be easily generated by quadrupling the test pattern shown in Table 2.1. Without such knowledge, one might have to use an exhaustive test pattern for the eight-input chip, which is much longer (2^8).

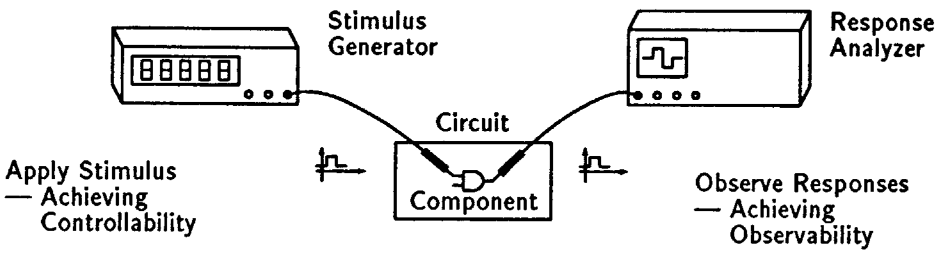

Stimulus Generator
Response Analyzer
Circuit
Component
Apply Stimulus — Achieving Controllability
Observe Responses — Achieving Observability

Figure 2.2: Test a Circuit by Testing its Components

The strategy of *assembling a test pattern for a circuit from component test patterns* is not possible if the internal components are *interconnected.* Instead, the concept of *executing component test patterns directly* is used in practice; this concept is usually referred to as *testing a circuit by testing its components*, as illustrated in Figure 2.2[2].

[2]It is the objective of *design test*, or, *design verification*, to check whether the circuit can behave correctly or not, given that all the components are functioning and all the interconnections are correct. The length of a design test is not critical since it is used only a few times **per chip design**, as compared with more than once **per chip** in the case of the manufacturing screen test.

Probing circuit nodes internal to a VLSI chip in order to execute component test patterns is very expensive and might cause circuit damages due to overriding circuit nodes during testing. Therefore, in practice, except on rare occasions, all component test patterns are executed *indirectly* through surrounding circuitry (Figure 2.3).

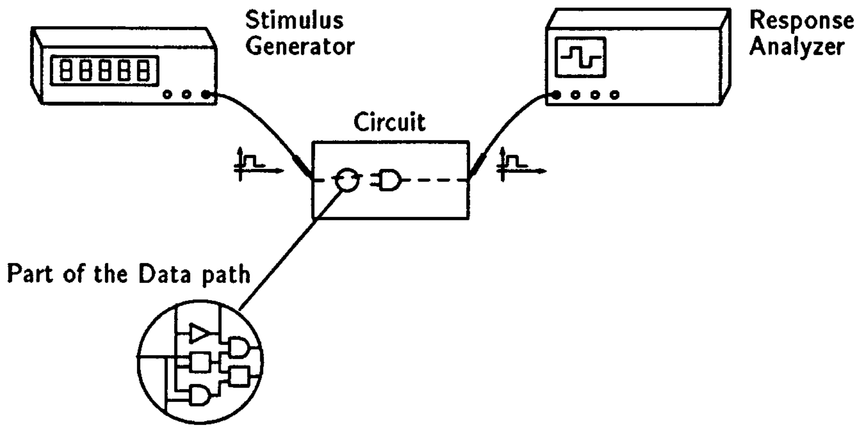


Figure 2.3: Testing Circuit Components Indirectly

Since component test patterns can be looked up in a library, the fundamental task for a test generator is to embed component test patterns[3]. As defined in §2.1.3, *component test pattern embedding* serves to *achieve controllability and observability* for the component. Since routing signals around a circuit involves satisfying conjunctive goals, the embedding is computationally intensive and in fact often *impossible.*

Many published test generation methods are computationally methods of signal propagation with testing as the ultimate goal. The famous D-algorithm, for example, is an algorithm for propagating signals around a circuit at the gate level, that is, propagating the D-token, which can be either logic 0 or 1 to indicate the presence or absence of a fault.

[3]Component test patterns can be *shared* by different circuit designs. Therefore generating them is separable from circuit test generation and more resources can be devoted to them to build up a library of high-quality component test patterns. On the other hand, the component test pattern embedding has to be done for *every* circuit design.

The Fundamentals of DFT

As discussed in the previous section, the fundamental tasks and failures for a test generator are concerned with achieving controllability and observability. From this viewpoint, it is not a coincidence that almost every DFT technique is also concerned with achieving controllability and observability. Table 2.2 shows some of the typical DFT techniques [13,14,22] in this perspective.

Techniques	Improve Controllability by Providing	Improve Observability by Providing
Test Point		Direct Access Datapath
Resettable Flip-Flop	Known Init'l State	
LSSD	Shift Chain Datapath	Shift Chain Datapath
BILBO	Source Near Focus	Sink Near Focus

Table 2.2: Example DFT techniques Help with Controllability and Observability

The first two rows in Table 2.2 are examples of early DFT techniques, better known as the ad hoc DFT techniques. The "test point" technique reconfigures a circuit during testing so that the states of selected internal circuit nodes can be observed directly from the pins of the circuit. Using "resettable flip-flops," one can start testing by resetting all flip-flops in a circuit so that the correct states and responses of the circuit can be determined easily.

The last two rows in Table 2.2 are examples of later DFT techniques, better known as the structured DFT techniques. The strategy of the "LSSD" technique is to connect during testing all the registers in a circuit into shift chains which start and end at the primary I/O ports so that (1) the stimuli can be transported from primary inputs to a focus through these shift chains and (2) the responses of the focus can be observed at the primary outputs by loading them into registers and then shoveling them out via the shift chains. The "BILBO" technique uses linear-feedback shift registers to generate stimuli and analyze responses.

One contribution this research makes is to classify published DFT techniques into three categories according to whether they can be easily used to fix particular controllability and observability problems:

1. *Design Guidelines, or Sweeping DFT Techniques*: This category includes those DFT techniques that can be used to reduce the overall possibility of testability problems in a circuit, but are difficult or too expensive to be used to fix a single signal propagation failure. Examples in this category are the use of the bus oriented structure, of the LSSD standard, and of some special layout configurations.

2. *Testable Components, or Highly Integrated DFT Techniques*: This category includes those DFT techniques that are concerned with test patterns of components, not with signal propagation. Examples are the PLA with a universal test pattern and the Reed-Muller implementation of components. Looking inside a testable component, one may find controllability and observability enhancement. For instance, in a PLA with a universal test pattern, an extra register is used to control the individual product lines. However, since it is usually integrated with the component structure, such enhancement is difficult to scale up and apply to general signal propagation problems.

3. *High-Level Controllability and Observability Enhancement*: This category includes those DFT techniques that can be viewed as cures to high-level signal propagation failures. This is the category that our system is tuned to. DFT techniques as seen in the literature are decomposed according to the signal propagation problems that they can solve. For instance, although the LSSD standard is classified into another category, *Design Guidelines*, part of its essence, the modifying of registers to shift registers in order to provide controllability or observability, is in this category. It is the author's belief that this category captures most of the published DFT techniques that can be effectively automated.

Associating Circuit Modifications with TG Failures

A large part of the DFT knowledge in our DFT system is about circuit modifications in relation to the test generation failures they can repair. The ad hoc DFT techniques can be viewed as circuit modifications intended to correct single testing problems. The structured DFT techniques can be viewed as integrated circuit modifications that correct several testing problems. For instance, the LSSD design standard reduces testing sequential components to testing combinatorial ones, thus solving test pattern generation problems, and provides a uniform control and observation scheme, thus eliminating test pattern embedding problems. Since each of the structured DFT techniques covers a wide range of testing problems, they can be used preventively.

However, low-overhead DFT needs to integrate circuit modifications with an original circuit design more than it needs to integrate circuit modifications among themselves. This is exactly the problem that structured DFT techniques sometimes have — if not all testing problems that a structured DFT technique can fix are present, applying the DFT technique may result in superfluous overhead.

This research is an attempt to explore the relationship between circuit modifications and the testing problems they fix in order to achieve low-overhead DFT. The results show that the signal propagation process during test pattern embedding is a reasonable thread with which to string both ad hoc and structured DFT techniques together and to tie circuit modifications with testing problems.

2.2 TG Failures and Fixes

2.2.1 The Goal Tree of TG

In terms of problem solving, test generation is a process of breaking goals into conjunctive and disjunctive subgoals and satisfying them. The goals and subgoals form a tree structure. In this thesis we will use a simplified goal tree structure for test generation: goals have either conjunctive or disjunctive subgoals, not both. In order to accomplish a goal in such a tree, we need to accomplish either all its conjunctive or any one of its disjunctive subgoals.

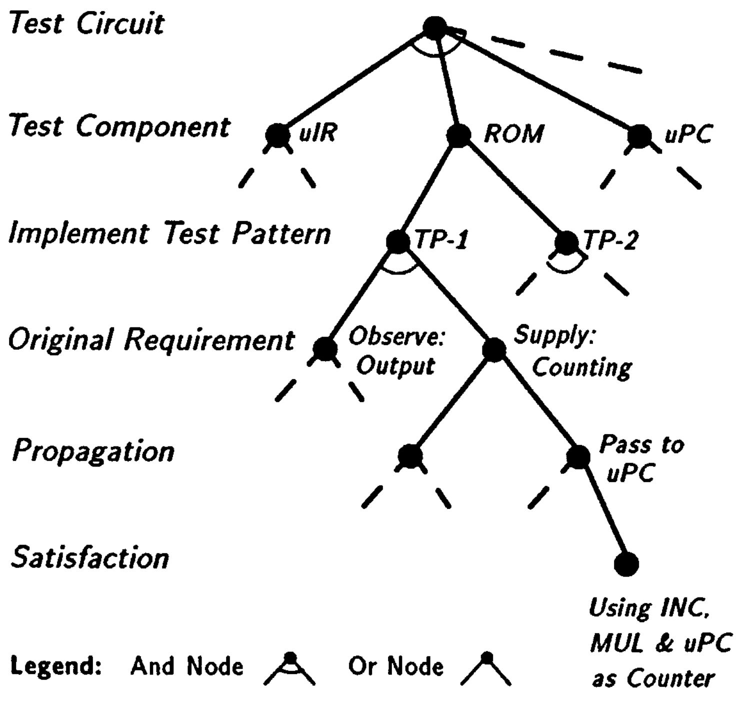


Figure 2.4: A goal tree of test generation

As an example, Figure 2.4 shows a goal tree of a path sensitization TG: The top goal is to test the whole circuit. This goal breaks down to conjunctive subgoals of testing each component of the circuit. For testing each component, different test patterns may be available, which form disjunctive subgoals of the "test component" level. To implement a test pattern, signals are needed to stimulate the component and responses of the component need to be observed, which together constitute the fourth layer of subgoals. The subtrees below the third level correspond to tasks of providing needed signals or observing responses, or the task of "path sensitization" and will be decomposed further according to the circuit structure. For instance the task of providing a counting sequence to the address of the ROM in the MAC-2 (Figure 1.1) can be accomplished through a component driving the ROM, the MUX.

Sometimes there can be multiple driving components, thus choosing one of them makes a disjunctive branch in the goal tree. The MUX can route in the counting sequence either from its input IN1 or IN2, i.e., working at different modes, making up another disjunctive branching in the goal tree. For the MUX working at either mode, different signals are needed for its input and this constitutes a conjunctive branching. As a requirement of the counting sequence "passed" through the MUX, we have another round of tasks to provide the counting signal, and the structure of the goal tree repeats.

2.2.2 Some General Properties of Goal Trees

Different test generators have different goal tree structures. But at the first few levels of their goal trees there are some similarities that are suitable traits for comparing and allying different test generators.

First the goal trees all have the same top goal — testing the whole circuit. What does this tell us? It means that every test generator is similarly capable of deciding whether a circuit is testable: when the top goal of a test generator is not achievable the circuit is not testable with respect to that test generator. Moreover this similarity leads to a testability definition and a measuring method of theoretical interest: a circuit is not testable if *every* test generator fails on it.

The second level of goals reveals how test generators first break down the top level goal. For instance, they could be testing a circuit by testing its components, its circuit nodes or its functions. If the top goal characterizes all test generators, the second level of goals characterizes different groups of test generators which directly complement each other. This is of practical interest: it can serve as a primitive interface between a DFT system and a large variety of test generators. For instance, "test component" is a general interface from test generators to our DFT system: each of the test generators that test components can mark the components testable to it; then our DFT system will take care of the rest.

The levels below the third are quite specific for each test generator; but occasionally there will be similar low level goals across test generators, and they can be used as alternative ways to achieve their common goals. More interesting is that for those test generators that work closely with the circuit structure, the low level goals correspond well to the circuit structure and provide guidance about which part of the circuit is supporting a goal.

2.2.3 Failures at Different Level of the Goal Tree

At different working stages the DFT system can be tuned at different levels of TG failure. At the stage of deciding whether a circuit is testable at all, the system is tuned at the top goal of TG. Tuned at this level, the system will not get any idea about particular testability problems, but it can in theory summon *all* test generators and therefore summon all the human wisdom on TG. Once a circuit is deemed not testable, to locate the testability problem the system has to look for the low level goals of TG that caused the higher level failure. As the system walks down the goal tree, more detailed information about the problem becomes available, but the number of test generators which are helpful with the particular problem decreases. For instance, when locating untestable components, the second level goals are examined and only those test generators that test components remain helpful. When locating controllability problems, even lower level goals are examined, and only the test generators that work on controllability problems may be helpful.

2.2.4 The Reference Test Generator

While all test generators fail on a untestable circuit, not all test generators produce goal trees that locate the testability problems to the extent that a fix is readily identified.

The reference test generator, the one integrated into the DFT system, must have the following properties to give the DFT system enough knowledge and computational advantages:

1. It must capture the signal flow in the circuit which reflects the fundamentals of testing and the essence of the circuit behaviors during testing.

2. It must locate testability problems to the extent that the problems can be fixed by circuit modifications.

3. It must be able to work with high-level circuit representations.

The first property is the most important one. This property allows a DFT system to use our testability definition naturally, since controllability and observability problems are equivalent to signal transferring problems. As analyzed previously, most DFT techniques solve signal transferring problems. Therefore, this property enables the DFT system to employ a larger variety of DFT techniques than the previous systems do which use only certain categories of DFT techniques, say, only structured ones. This property also gives a DFT system deeper knowledge about circuit behaviors than the previous DFT systems have employed. Deeper knowledge of circuit behaviors enables the DFT system to combine DFT techniques more flexibly and to use a circuit more wisely to repair its testing problems, since the DFT system knows whether merely a fraction of a DFT technique or a combination of DFT techniques suffices to repair a testability problem.

The second property will help the system identify the relevant fixings. The TG should produce failures that can easily index into a large variety of DFT techniques. If the failures cannot index DFT fixes easily, the computational cost for finding fixes will likely be high; if the failures can only index a small variety of DFT techniques, the circuit overhead will likely be high since the options are restricted and the system is less likely to hit the optimal solution.

The third property will enable the system to reduce the search space. If the TG is based on detailed circuit designs, it will drive up the computation complexity and make it difficult to work with circuits at early design stages. And it is also doubtful that such TG could have the second property, i.e., produce failures that meet the indexing criterion.

The implemented system uses a version of a path sensitization test generator working at the register transfer level. The TG works directly with controllability and observability; thus it has property 1. It propagates controllability and observability requirements through a circuit; therefore its failures can be directly matched up by known design modifications (as we analyzed in Section 2.1.3); thus it has property 2. The register transfer level is a high-level circuit representation; thus it has property 3.

2.2.5 Fixes for the Failures

A failed goal can be turned into a successful one either by finding a new way to achieve it or by fixing some of its failing subgoals. For instance, a needed signal can be injected directly into a circuit node, or achieved indirectly by passing it from other circuit nodes through existing data paths.

Repairs can be divided into three groups according to how they change the goal tree, or correspondingly, what kind of changes result in the circuit (Figure 2.5).

Fix I: Swapping Foci

This group contains all the DFT techniques in category 2 defined in §2.1.3.

Swapping a focus means substituting the component with a different implementation so that its test pattern becomes easy to compute or apply. For instance, a PLA with a universal test pattern can be substituted for an ordinary PLA. Substitutions of this kind often need some minor circuit structure changes as well, such as different control encodings.

The resulting changes to the TG goal tree are below the second level. The goal of testing the original component and its whole subtree are changed — that is the component and all its test patterns are new, and so are the test-pattern implementations. When the substitute has exactly the functionality of the original component,

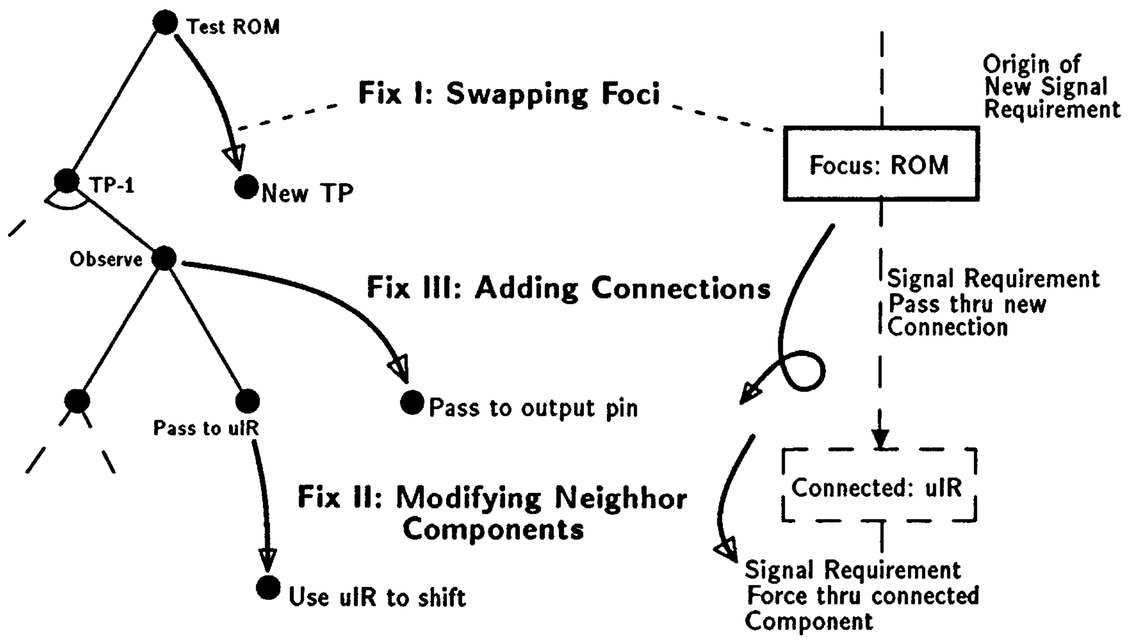

Test ROM
Fix I: Swapping Foci
Origin of New Signal Requirement
Focus: ROM
TP-1
New TP
Observe
Fix III: Adding Connections
Signal Requirement Pass thru new Connection
Pass to output pin
Pass to uIR
Fix II: Modifying Neighhor Components
Connected: uIR
Use uIR to shift
Signal Requirement Force thru connected Component

Figure 2.5: Different fixes

other parts of the goal tree remain the same even if they use the original component; otherwise the related parts of the goal tree need to be adjusted accordingly.

To be competent at such substitution, *evaluation* knowledge is very important. Such evaluation is usually local to the site of focus swapping. Since the implementation does not currently have enough evaluation knowledge and since researchers have built a system [23] specializing in swapping foci, we suggest calling such a system for help.

Fix II: Modifying Neighbor Components

This group of DFT-modifications includes part of those in category 3 defined in §2.1.3.

The purpose of Fix II is to improve the controllability or observability of the focus. In order to preserve the function of the original circuit, we consider only modifications that add functionality to existing components and invoke the new functionality only in test mode. Examples of such modifications are using an LSSD or BILBO register in place of an ordinary register; adding entries to a ROM; and adding new inputs to a MUX. To limit the search space, the functionalities to be added to a component are constrained by two criteria: they must conform to some well known DFT modifications and they must be cost-efficient.

Currently the system is focused on making suggestions in this group. To be good at this, the system needs to know how to *find* and *evaluate* relevant modifications. Our implementation emphasizes the former.

Fix III: Adding New Connections

This group of DFT-modifications contains the other part of those in category 3 defined in §2.1.3.

The basic difference between Fix III and Fix II is in the knowledge needed to limit the search space. Fix III needs layout information that is not explored in the current implementation.

2.2.6 Another Scenario: Bridge the Gap

Having chosen the path-sensitization test generator as the reference test generator, we can depict our approach as a bridge building process:

Once the test pattern for a component is known, the task of the TG is to route the signal in and out. In a failed TG attempt at a focus, the route starting from the primary I/O fails before getting to the focus, and the route starting from the focus fails before getting to the primary I/O, leaving the focus unconnected to primary I/O. There is a gap for the signal to cross, which is what Fix II and III are for.

If the test pattern for a component is not known Fix I may solve the problem. Or, the component can be treated as an untestable circuit, using our system recursively to examine its internal structure then modify it so that a test pattern can be found.

2.3 The DFT, TG and Designer Trio

Testability concerns were basically an afterthought for the VLSI chips of early days which worked fine for some time. As the complexity of VLSI chips grew, last-minute changes to achieve testability of the circuit design have become increasingly expensive. The need for integrating testing into the design process becomes ever more pressing.

Sections 2.3.1, 2.3.2 and 2.3.3 present the view developed in this research: DFT should be one of the three stages in VLSI design — functional design, test generation, and DFT. Circuit design cycles through those stages while the details of the design are worked out and testability blends into the design as early as is appropriate to approximate the desired integration of DFT and design. Based on this view, §2.3.4 analyzes some confusion about the role of the DFT process in previous DFT systems. §2.3.5 discusses what kinds of test generators are suitable for checking circuit testability. §2.3.6 reviews the trio view and discusses the relation of our DFT system with the trio. §2.3.7 summarizes the testability problems the DFT system repairs. §2.3.8 collects a set of heuristics which allow the DFT stage to be relatively self-contained, i.e., to work relatively independently in the absence of the other two stages of the design process.

2.3.1 Incremental Implementation

Automating the whole VLSI circuit design process is a long-term goal that is still beyond our reach. In order to move gradually toward this goal and to enjoy a partially automated design process along the way, a divide and conquer strategy can be employed. This strategy would involve dividing the whole design process into smaller pieces according to the knowledge involved and automating each of these pieces whenever it becomes feasible.

2.3.2 Three Parts of Knowledge

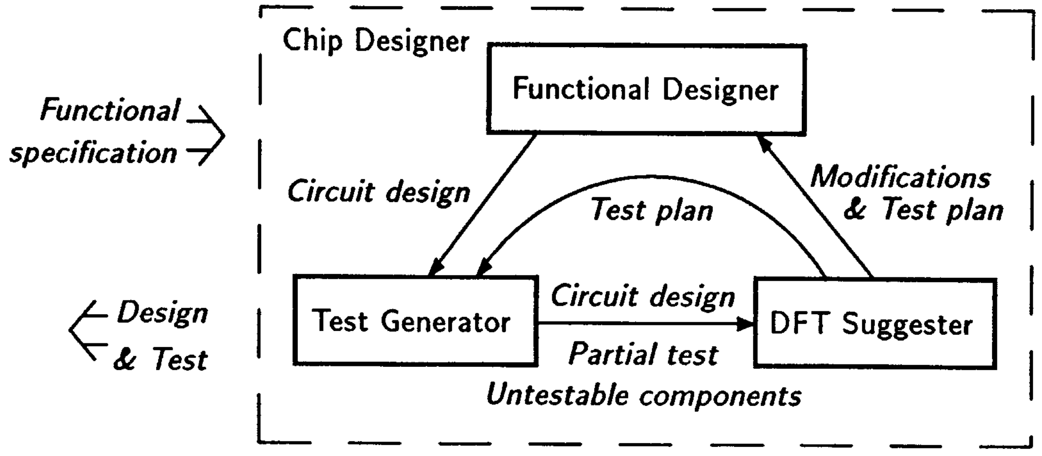


Figure 2.6: VLSI Design System: a DFT Perspective

The knowledge involved in VLSI circuit design can be divided into three relatively independent parts: design knowledge (concerning implementing functional specifications), test generation knowledge, and DFT knowledge (solving testability problems for digital circuits). The three parts of knowledge can be embodied in three relatively independent subsystems — namely a designer, a test generator and a DFT-suggestor — which cooperate as shown in Figure 2.6.

The circuit design cycles through the three components of the CAD system as the details of it are gradually worked out in a top down fashion. The Functional Designer deals with the implementation of functional requirements, without concern for testability. At every design stage, the designer passes the design to the Test Generator to check testability and to generate a test. Then the design along with testability problems and other relevant information is passed to the DFT-Suggestor. The DFT-Suggestor examines the testability problems in detail, looks for possible circuit modifications to solve them, and finally passes its findings back to the designer as additional functional specifications to implement.

2.3.3 Approximating an Integrated System with an Interwoven One

An approximation of an integrated DFT and design process is achieved by passing a circuit design around the three systems. On the surface the testability issue is still an afterthought for each design stage, but when early design is passed through the DFT system, test problems can be found when they are less expensive to fix. With the TG and DFT systems acting as a testability consultant, the designer can view the DFT modifications as new specifications to the circuit and integrate testability considerations into the circuit without explicitly worrying about testability. In this scenario the testability consultant looks over the shoulder of the designer and reminds him *as soon as* a testability problem surfaces.

2.3.4 Concentrating on DFT

There has been a common confusion in previous DFT systems about the central task of a DFT system. DFT systems inevitably have to deal with issues of design, test generation and DFT simultaneously. However, being unclear about the difference among issues of a different nature, these systems fail to focus their efforts on the central task of DFT: locating testability problems in circuits and discovering suitable solutions. These systems often vigorously tackle issues less essential to DFT systems, for instance, evaluating the impact of suggested modifications, while missing the more vital ones, for instance, deciding the necessity of suggested modifications. This confusion is the root of the following phenomena:

- Improperly defined tasks

  These systems derive circuit modifications to make components "testable" without knowing whether these components are originally testable or not. They do not have a proper definition of the testability problem, a key to defining the central task of a DFT system. A possible reason is that these systems fail to recognize the importance of test generation knowledge to DFT systems and use heuristics instead. Or, to put it differently, they only assume very primitive test generation technology and require the DFT system to make circuits excessively simple to test. For instance, the LSSD method only assumes test generation ability for combinatorial circuits and reduces sequential circuit testing to combination circuit testing.

- Improperly allocated resources

  On the one hand these DFT systems pretend to include some design and some test generation tasks, for instance, deciding the optimal modifications or working out detailed test timings. On the other hand, they do not seek help from existing systems and try to solve all related problems by themselves. The result is that these DFT systems have to spread their limited resources over all the different tasks and do not attack DFT issues effectively.

Tasks Better Done by the Designer

It is better to identify testability problems and fix them as early as possible in the design process. The later a modification is made, the more expensive it is, since many low-level, detailed design specifications are very expensive to work out and so specific to high level specification that they often do not survive the design changes. Therefore, the circuit design is given to the DFT system in its early stage, with details still to be worked out. The DFT process should be situated between stages of the design process, not at the end, i.e., the DFT system should not generate a finished circuit design but rather requirements for the designer to implement in later design stages.

Furthermore, except for the explicit circuit changes required by DFT modifications — such as adding a wire — there are implied circuit changes that are better viewed as additional specifications to be implemented: providing proper control signals in testing mode for example, or making a component faster. Implementing these additional specifications is actually a design task and involves checking whether they are consistent with the original specifications, resolving inconsistencies encountered, integrating the new specifications into the original design, and so on. This process depends significantly on information and knowledge available only to the designer and is therefore better handled by him.

Tasks Better Done by TG

A DFT system is not and should not be a powerful test generator. Since much test generation expertise is still difficult to capture in a program, a system competent for test generation alone is already challenging enough to build and to keep up to date with new developments in TG.

On the other hand, the DFT system needs the testing wisdom to eliminate false testability problems. The DFT system also needs the testing expertise to verify whether partially developed tests for original circuits are invalidated or new testability problems emerge due to the changes it introduces.

Our solution to this apparently paradoxical situation has two constituents. The first one is an interface between the DFT system and conventional test generators so that existing test generation expertise can be exploited to locate testability problems and help with test details. This can be viewed as off-line help from test generators. The second constituent is to work closely with a test generator — the reference test generator — which bears the test generation knowledge in the DFT process, i.e., gives the DFT system on-line help.

Concentrating on DFT: Summary

By acknowledging that there is both a designer and a test generator working with the DFT system, the confusion on the central DFT task is avoided; and the system is able to focus limited resources on its central task. Without full knowledge of design and test generation, a focused DFT system still can do substantial and meaningful work. For instance, in the absence of a standby designer, heuristics can be consulted instead and a few favorable solutions can be generated without "bothering" an off-line designer with trivial or premature questions.

2.3.5 Test Generators Locating Testability Problems

Locating real testing problems in a circuit is the key to a near-minimal DFT modification. Theoretically every and any test generator that fails on an untestable circuit can locate testability problems, but not all of them are suitable for this purpose. There are two important criteria:

- *Speed*: The test generator must succeed quickly on testable parts of a circuit and fail equally quickly on problematic parts. In a sense DFT can be viewed as trading circuit overhead for less test generation cost. Therefore, if it takes too much time for a test generator to figure out what the testing problems are, the DFT overhead reduction resulting from the locating accuracy so gained does not pay off.

- *Accuracy*: The test generator must have a suitable failure vocabulary. It will not help if a test generator locates testing problems which the DFT system cannot handle efficiently.

More work needs to be done to modify existing test generators and seek new test generation approaches to meet this requirement so that the TG part of the trio can be automated. We find that human experts are good at locating testability problems in circuits. Except for empirical knowledge such as which part of a circuit of a particular type is statistically difficult to test, knowledge about design intention, i.e., how the parts of a circuit are going to work together, helps a lot in figuring out which part of a circuit is problematic in testing. Recently some work has been done in this direction [19]. It is based on such work and human testing expertise that the TG part of the trio is formulated.

2.3.6 The DFT System in the Scene of the Trio

This section reviews the trio view of VLSI design and discusses our DFT system in relation with it (Figure 2.7).

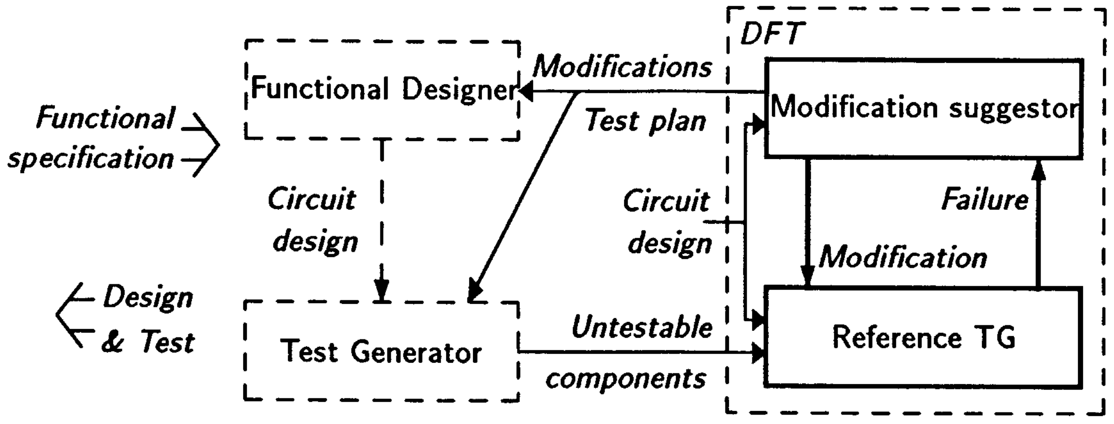


Figure 2.7: Our DFT System in Context

The first advantage of the trio view together with the new approach to building a DFT system is that it emphasizes TG knowledge in the DFT process. Test generation knowledge can play the following roles in DFT:

- Decide whether there is a testability problem in the circuit design,
- Refine the test plans which a DFT system suggests for the problematic parts of a circuit,
- Locate testability problems to the resolution for which a fixing circuit modification can be readily found, and
- Guide the DFT system to combine repairs wisely.

In Figure 2.7 the box labeled Test Generator performs the first two roles in addition to acting as conventional test generator. To play the first role, a test generator needs a sense about the difficulty in achieving its subgoals, i.e., it should detect the "hard" problems *efficiently* and leave them to the DFT system. The box labeled Reference TG basically plays the last two roles. The reference test generator is tuned to available DFT circuit modifications and high level circuit and signal representation. It serves as a framework for the proposed DFT system.

The second advantage in this view is that it cleans up the relationship between the three design stages:

- Functional designer versus DFT: Testability is one of the mandatory specifications that the final circuit design must meet. The design is not finished until the testability requirement is satisfied. Therefore DFT must be part of the design process. On the other hand, the designer can help with the DFT process. Besides serving as a consultant on circuit modification trade-offs and implementability, the designer can also share circuit design information with the DFT system, such as the purpose of a component.

- Functional designer versus TG: The ultimate criteria for testability is that there exists a test for the circuit, which is most accurately verified by real test generators. Therefore test generation should be a natural part of the design process to verify the testability, and a test should be included in the result of the design. On the other hand, as addressed in the work of others [19], the designer could give the TG more information about the design than is usually included in the blue prints, such as design intention, which will help the test generation for complex circuits.

- DFT versus TG: Their relationship should the most friendly in the trio. The TG can help the DFT to locate testability problems in a circuit and to assemble modifications properly. On the other hand, the DFT helps the TG by removing the hard testability problems from a circuit.

- "Don't fix it if it ain't broken": The testability specification should be specified *during* the design process as problems arise. In contrast, some existing DFT methods or systems predetermine a certain solution *before* the design process is started; for instance, some systems apply the LSSD method regardless of whether there is a problem or not.

The third advantage the trio view brings us is that it divides up the whole design process more properly for building DFT systems. Compared with the division in previous DFT systems, it explicitly addresses tasks for measuring testability and implementing modifications and suggests solving them with test generators and designers. As a result, it leads to a more practical DFT system which concentrates on the tasks the other two do not take care of, and makes more competent suggestions.

2.3.7 The Testability Problems Our DFT System Repairs

An untestable component may have one or more of the four categories of testability problems:

1. No test pattern
2. Lengthy embedding
3. Not enough controllability
4. Not enough observability

Our DFT system repairs the last two kinds of testability problems. This system does not have special knowledge for solving the second kind of problem. However, we speculate that by treating the lengthy embedding problem as a controllability or observability problem this system could suggest modifications which in fact shorten the embedding. As stated previously, our system is not capable of repairing the first kind of problem, though we have suggested calling a system specializing in swapping foci for help, or to applying our DFT system recursively on the untestable component (see §2.2.5 and §2.2.6).

2.3.8 Some Heuristics

This section presents some basic heuristics. These heuristics can be viewed as a supplement to the consulting services from the designer and test generator (Figure 2.6) so that they will not be bothered with trivial and premature queries.

As previously mentioned, our DFT system is only one of three closely related systems for designing a VLSI chip. This thesis research is dedicated only to the DFT part and the system does not yet have good access to designers or test generators. In this situation these heuristics become vital to the DFT system. These heuristics are necessary for estimating the feedback from the designer or the test generator in order to avoid having to consult the designer or test generator, and hence stopping the DFT system before any meaningful work has been accomplished.

"Functionality Reservation" Heuristic

According to the "functionality reservation" heuristic, if modifications are additive, the original functionalities will be preserved.

Additive modifications only add more logical functionality to a device, for instance, adding a shift function to a register or adding a disconnect function to a wire by installing a switch. The original functionalities of the system are preserved because we have full control over when the additional functionality is invoked, so that the modified device can still perform the function required by the original system.

This heuristic may not work when a modification introduces timing changes exceeding a certain limit. However, we consider that the effect usually can be balanced out by small adjustments to the overall timing, perhaps at the cost of some small performance degradation.

"Heavy Usage" Heuristic

According to the "heavy usage" heuristic, if a functionality of a component is used only in testing, it does not need a specific test since if the test is passed it is exhaustively tested regarding the duty assigned to it.

This heuristic implies that by introducing additional testing hardware into the circuit, we usually do not bring in a testability problem harder than the original one.

There are two situations we need to watch out for:

1. We will have problems when adding functionality to a component changes the test pattern for its original functionalities *and* it is not clear whether the new test pattern can be embedded when the old one can.

When this happens, we should run test generation on the component and run DFT on it if the test generation fails. If it is too expensive to test the changed component, an alternative to the change should be tried instead. This case should be rare since we expect that the data path for carrying out the test pattern is capable of carrying arbitrary patterns, thus the new test pattern can be easily substituted for the old one without working through the circuit again.

If the component is originally untestable due to an embedding problem, there is even less to be worried about: unless the additional functionality makes the test pattern for the component non-existent, we will embed the new test pattern instead the old one. Since most of the DFT modifications will introduce data paths general enough to carry arbitrary signals, different test patterns will make little difference.

If adding functionality results in there being no test pattern, the cost of re-implementing the component should be checked out before discarding the suggestions involving the functionality addition.

2. We will have problems when an added component can not be tested adequately without a specific test for it. The purpose of the added component is testing, but it must provide certain functionalities in the normal working mode so that it will not interfere with the normal operation. These functionalities are used in the normal operations of the chip but the testability of the added component has not been checked because it has just come into existence. If these functionalities are not tested adequately during the testing of other components, a specific test is needed. In general the newly added component should be handed to the TG to see if it is testable with

regard to these functionalities. Possible consequences are similar to the previous case.

"Used Function" Heuristic

The "used function" heuristic complements the previous heuristic. It says that only the component functionalities that are used in the normal operation need testing.

For instance, an incrementer in a circuit might have been implemented by an adder with one addend tied to 1. According to this heuristic, you do not need to test whether the adder can add two arbitrary numbers correctly (i.e., test the adding function in general); instead all you need to do is to test whether it adds 1 correctly. This means you do not need to untie the input of the adder and try to rout an arbitrary number to it at testing.

This heuristic reflects the opinion that one needs not worry about the redundant faults, i.e., the physical faults in a circuit that do not affect its function. The unused functionalities are usually associated with redundant faults. By ignoring unused functionalities in testing, the overhead for making effectless faults detectable can be eliminated.

This heuristic can fail if exercising only a subset of the functionalities costs more than testing all of its behavior. Should this happen we may have to introduce a test mode in order to provide the necessary control and I/O to exercise the redundant functionalities in testing.

This heuristic shows the iceberg tip of the vast amount of information that the designer possesses which can help the DFT system. More work needs to be done to exploit such design information.

"Must Modify" Heuristic

According to "must modify" heuristic it is always necessary to introduce a modification to solve a testability problem. In other words there is no solution to a testability problem which does not introduce modifications. This heuristic is a direct consequence of "DFT in the context of TG failure" (see discussions in the "focusing on circuit testing behavior" heuristic to be introduced next).

One caveat is in order: Currently the interface between our DFT system and the TG only specifies whether a component is testable. It is possible that the testability problem for an untestable component is only in controllability but the system will try to solve the non-existent observability problem as well (and vice versa). This wastes CPU time but has no theoretical importance since it will disappear when in the future a better interface is available to pass more information from the TG to the DFT system.

"Focusing on Circuit Testing Behavior" Heuristic

The "focusing on circuit testing behavior" heuristic is a direct consequence of the idea about dividing the circuit design task among the designer, the TG, and the DFT as presented in §2.3.4.

Circuits have two operation contexts: normal and testing. Since circuit behaviors are simpler in testing than in normal operation, it is beneficial to distinguish them. Based on this distinction, this heuristic is concerned with the knowledge and task distribution between the TG and our DFT, two components of the trio. This heuristic argues that in the context of TG failure, our DFT system can concentrate on circuit testing behavior, simplify the circuit representation, and focus its resources on modifying the circuit. The DFT system can get help on test related circuit normal behavior from conventional TG if necessary.

As implied by "DFT in the context of TG failure," our DFT system does not need to check whether a testing problem can be solved without modification, i.e., within the normal behavior of the circuit. But someone has to check this when a testing problem is first identified. Theoretically, any test generator is capable of locating testing problems. However, the complexity involved renders most off-the-shelf test generators unprofitable. Fortunately, recent development in test generation has brought us some good news: [19] has presented a concrete test generator that explores the normal behavior of a circuit, quickly succeeds on easy testing tasks and quickly fails on hard testing tasks, which is ideal for checking the normal circuit behavior and locating testing problems for our DFT system.

By ignoring the normal behavior of a circuit, the DFT system needs to know less about the circuit and thus can work with a higher level of abstraction than it could otherwise. Working with higher level of abstraction not only brings a computational advantage to this system, but also enables the system to work with circuits at early design stages when few details of the circuits have been worked out so that testability problems can be found and corrected early and expensive last minute design changes avoided.

Our experiment has shown that ignoring circuit normal behavior in a DFT system does not severely hinder its power.

From a broader point of view about the whole design process and considering that there is an ultimate need for checking circuit normal behavior, it is still computationally more advantageous to implement *two specialized systems* than to implement *one general system*; because the computational complexity for an ordinary TG system that must check circuit normal behavior is already at the NP-complete fringe, it will certainly make the matter worse to generalize the TG system in a non-trivial way.

"Ignoring Control Changes" Heuristic

The "ignoring control changes" heuristic is also a direct consequence of the idea about assigning the circuit design task among the designer, the TG, and the DFT as presented in §2.3.4. Ignoring circuit normal behavior is a tradeoff between the TG and the DFT; ignoring control circuitry changes is a tradeoff between the Functional Designer and the DFT.

Whenever a component is modified or a testing mode is introduced, that is, whenever an untestable component is turned into a testable one, the original control circuitry must accommodate the change. However, instead of figuring out the exact control circuitry changes, the system simply reports the needed control scheme as *specifications* for the Functional Designer to implement.

This treatment of control changes is supported by arguments about the computational complexity and the need for handling circuits at early design stages similar to those for ignoring circuit normal behavior. In addition, one can further argue that:

- The circuit control scheme is not finally decided at early design stages. Therefore it is unlikely to incur "re-implementation" costs by the additional specifications.

- It is expected that the control circuitry in modern VLSI chips is implemented with structured circuitry such as PLA or ROM. Therefore, if re-implementation does occur the impact of control specification changes is expected to be small, both in terms of re-implementation costs and resulting circuit overhead.

- Designing or redesigning the control circuitry needs global knowledge about the whole circuit. Therefore it should be better to collect all the specifications and then implement them globally rather than implement individual specifications as soon as it comes to light.

2.4 Testing Criteria

Concentrated on automating DFT process, so far we have suppressed some important issues in TG, such as test fault coverage and resolution, fault modeling, etc. However, if a DFT system cannot defend itself on these issues, although only remotely related, it will be of little practical value. This section discusses our approach in relation with these issues: §2.4.1, fault coverage and resolution; §2.4.2, fault simulation; §2.4.3, fault modeling.

2.4.1 Fault Coverage versus Resolution

Two important criteria about tests are *fault coverage* and *fault resolution*. *Fault coverage* is the ability of a test to detect physical faults in a circuit. It is usually measured by the ratio of the physical faults that the test can detect to the total possible physical faults in the circuit. *Fault resolution* is the ability of a test to distinguish different physical faults.

A manufacturing screen test emphasizes fault coverage because its purpose is to detect any physical fault in order to prevent faulty circuits from being shipped out. A quality control test emphasizes fault resolution because its purpose is to locate faults or to gather statistics about how often certain faults occur in certain areas of a circuit in order to debug the manufacturing process.

Our DFT system will not explicitly indicate the resulting fault coverage and resolution when it makes a circuit testable. Once an untestable component is made testable through circuit modifications, the fault coverage for that component will be whatever coverage the test pattern offers; the fault resolution, however, will be lower than the resolution that the test pattern carries since faults on data paths used in the test might be confused with the faults in the component. The degradation is expected to be small though, since faults in data paths may affect several foci and thus are likely to be recognized.

2.4.2 Fault Simulation

Fault simulation serves two purposes in testing: test grading and diagnosis. In test grading, a faulty circuit is simulated to see how many faults a test can detect. In diagnosis, the behavior of a simulated faulty circuit is recorded in a fault dictionary and compared with the behavior of an actual faulty circuit in order to locate the fault. If the behaviors of the actual and the simulated circuits match, the simulated fault explains the behavior of the actual faulty circuit.

Simulating a complicated circuit as a whole is expensive. Our approach encourages localized simulation, i.e., simulating individual components instead of the whole circuit. Test grading results for component test patterns are directly applicable[4]. As for fault dictionaries for each component, to a less degree they are also applicable but no general claim is made here. The situations for test grading and diagnosis are analogous to those for fault coverage and resolution.

Therefore, the conclusion is that if the fault simulation results are available for library test patterns, our DFT system introduces little extra fault simulation requirement after it makes a circuit testable.

2.4.3 Fault Modeling

Our DFT system does not specify how it models faulty circuits. Test pattern generation should take care of the fault modeling for individual components. In other words, the system models faulty circuits in the same way as the test patterns in its library do. At the structural level, however, the system assumes that there can be at most one faulty component in a circuit.

[4]Underestimation may occur since some faults on data paths will also be detected when the test pattern is executed but they may not be counted as covered faults.

Chapter 3

An Implementation

This chapter describes the functional components of the system, the process to construct the solutions, and the representation of circuit, test generation and DFT knowledge.

3.1 The System Structure

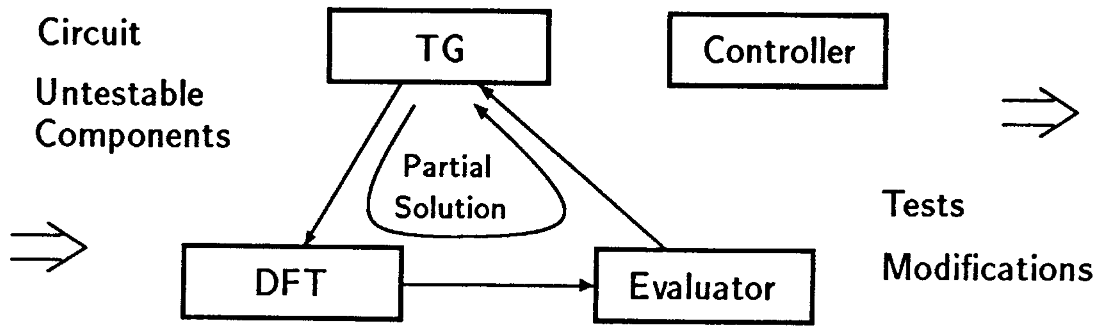


Figure 3.1: Structure of the DFT system

Figure 3.1 shows the conceptual structure of our DFT system. The input to the system is a circuit description and a list of untestable components. The reference test generator TG runs on every untestable component until it fails. The partially finished test along with failure information is passed to the DFT modification suggestor DFT to be fixed. The partial test with fixes, the output of the DFT, is evaluated in the Evaluator. The partial test with fixes considered worth continuing is given back to the TG to resume the TG process from the point where it failed. This process repeats until all the untestable components are made testable. Finally the tests for the untestable components and the modifications needed to make these components testable are reported to the user. For any component that remains untestable after this process, the system reports some partial tests to reveal the remaining problems.

Details about each of the system components are presented in the rest of this section. However, since the process of TG, DFT and Evaluator are highly interrelated there are no clean separations among them in the implementation.

3.1.1 The Reference Test Generator

The reference test generator uses the D algorithm except that it works with a high level circuit and signal representation — the circuit description is at the block diagram level while the signal description vocabulary includes "counting sequence", for example, instead of using a gate level circuit description and D token vocabulary. This test generator is also able to use a group of components together as a single component.

Test generation for an untestable component starts from a test pattern retrieved from the library. The test pattern is represented as input required to exercise the component and signals on the outputs to be observed in order to verify the behavior of the component (see §3.3); both are called the *signal-requirements.*

The test generator then propagates the signal-requirements through the surrounding circuitry as far as it can go. The propagation is guided by the circuit topology and component behavior model. When a signal-requirement reaches a component port through a circuit connection, the system determines the signal-requirements to appear on other ports of the component according to its behavior model.

For instance, assume there is a signal-requirement for a four-bit binary signal sequence 0001, 0010, 0100 and 1000 (called walking 1 sequence, used to test registers). When a ROM receives this signal-requirement at its output, and it has the signal stored in sequence, it can propagate the signal-requirement through itself to become a signal-requirement for providing the appropriate counting signal on its address. Section 3.3 will detail the representation of component behavior.

The propagation terminates successfully when a signal-requirement is propagated to a source or a sink (see §3.3.1, *Component Port*). The propagation fails if the test generator cannot satisfy a signal-requirement at the input or output of the focus. The unsatisfied signal-requirements of the focus and the subgoal tree generated from the unsuccessful propagation will be given to the DFT modification suggestor to be fixed.

3.1.2 The DFT Modification Suggestor

In this implementation, the DFT modification suggestor makes suggestions only for test generation failures involving signal requirements on data paths, not on control lines[1]. These failures are used to index DFT modifications in the library. Details are presented in §3.2.1.

3.1.3 The Evaluator

The test generator and DFT suggestor as described above do not check global consistency; for instance, they do not check whether a component has to work in different modes at the same time. This is left to the evaluator. In one sense, the system works in a generate and test fashion: the TG and DFT is the generator, the evaluator is the tester.

The evaluator does the consistency check incrementally: it evaluates incomplete solutions as well as completed ones. When the TG propagates a signal-requirement one step further, the new partial solution is evaluated again. If a partial solution

[1]See §3.3.1, *Component Port* and §4.2 for the reason and other modifications

become unfavorable, it will be suspended immediately until it becomes clear that there are no better solutions.

The evaluation is based on the consistency of the solution, the complexity of the data path used in testing the foci, the cost of modifications, and other factors. Details are presented in §3.3.

3.1.4 The Search Controller

DFT processing can be further divided into two stages: constructing the local solutions for individual signal-requirements and constructing the global solutions from the local solutions.

In the first stage, all the signal-requirements from the test patterns are locally propagated in a non-monotonic forward chaining rule based subsystem (see [7]). Here *local propagation* means propagating the signal-requirements according to the property of the signal requirements and directly involved components, that is, ignoring the history of the signal-requirements and resource contention (see §3.2.1). The rule-based system uses a modified depth-first search. The controller has a last-in-first-out *task-group* stack; it completes all the tasks in a task-group before popping off another task-group on the stack. The advantage of this control scheme over a simple depth-first scheme is that several tasks can be executed as a group without interruption. Used properly this can bring some computational savings. For instance, in a non-monotonic system several facts can be derived in a task-group to stop the firing of some rules which would otherwise be fired by a subset of these facts.

After the rule-based subsystem propagates the signal-requirements to an end, the system organizes the local solutions into an explicit tree structure by tracing the reasoning dependencies of the rule system. The purpose of this transformation is to eliminate some redundant dependency information in order to facilitate the subsequent processing.

In the second stage of constructing global solutions, the system uses a mixed best-first and depth-first search in the local solution tree. At the grand level a set of criteria (§3.2.3) is posed on global solutions. If there are not enough solutions the criteria are deemed too strict and will be relaxed in order to explore more alternatives. This is a kind of best-first search. However, between two relaxations of the criteria, all the active searches are processed in a depth-first fashion.

3.2 Constructing the Solution

The system proceeds in two stages: in the first stage the reference test generator and DFT modification suggestor are involved, working together to achieve controllability and observability without concerning their interactions. This is done by local signal-requirement propagation. At this stage all relevant circuit modifications are tried. A solution tree is produced for the second stage to process. In the second stage, the reference test generator works with the evaluator to form complete tests from independently developed controllability and observability for each untestable component and then the whole circuit. In this stage, the modifications tried in the first stage are selected for the whole circuit. Details about the two stages are presented in §3.2.1 and §3.2.2.

3.2.1 Constructing Local Solutions

In the local solution construction stage, the system tries to build controllability and observability for the circuit nodes that are relevant to circuit testing. It does so by converting the top goal of making a circuit testable into subgoals without considering interactions among those subgoals and their solutions. The result will be a goal reduction tree with solutions for leaf goals attached. The process will be presented using the example of Figure 3.2.

As shown in Figure 2.4, after the first few steps of goal reduction, the system comes down to the layer of implementing test segments in test patterns for different components. Figure 3.2 shows one of those subgoals, the implementation TS-1 for the ROM. That subgoal is further divided into two subgoals: applying a counting sequence for the address of the ROM and observing its content at the output.

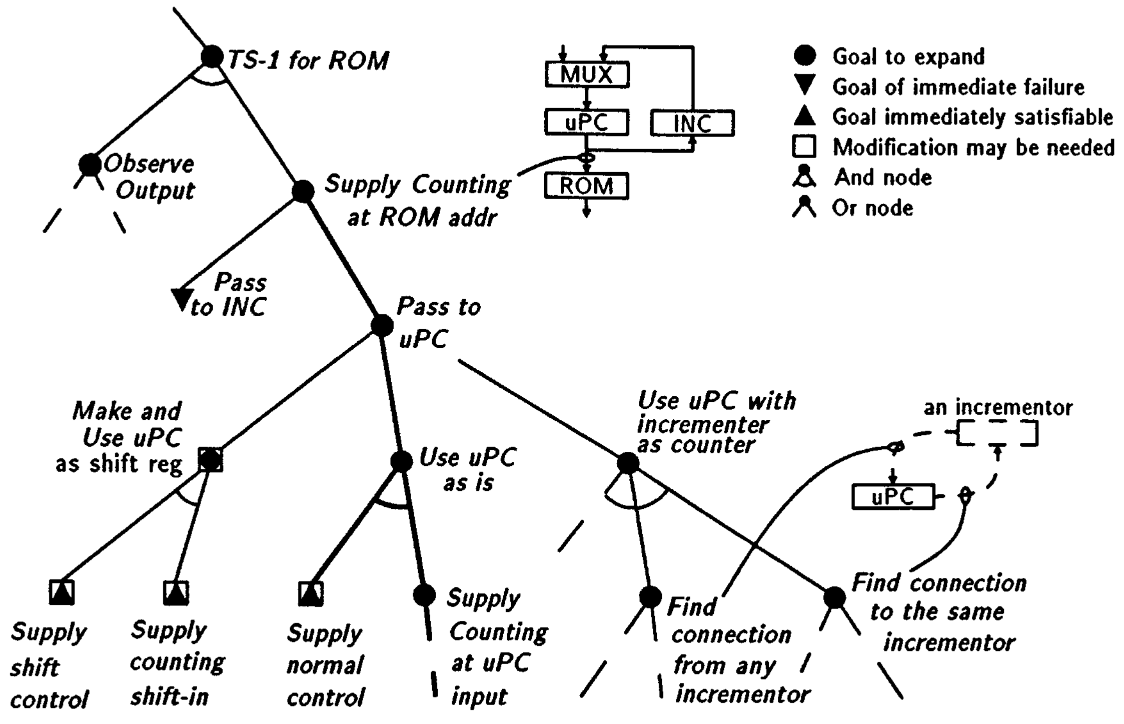

TS-1 for ROM
Observe Output
Supply Counting at ROM addr
Pass to INC
Pass to uPC
Make and Use uPC as shift reg
Use uPC as is
Use uPC with incrementer as counter
Supply shift control
Supply counting shift-in
Supply normal control
Supply Counting at uPC input
Find connection from any incrementor
Find connection to the same incrementor
MUX
uPC
INC
ROM
an incrementor
uPC
● Goal to expand
▼ Goal of immediate failure
▲ Goal immediately satisfiable
□ Modification may be needed
And node
Or node

Figure 3.2: Local goal expansion tree

These two signal-requirements must be satisfied simultaneously to implement TS-1. Further processes for these two signal-requirements are symmetrical so the picture only goes down along the input side, which is more interesting in this example.

Recall that there are two components in MAC-2 that are connected to the ROM address, namely the INC and the uPC, thus there are two ways we might get the needed signal.

Trying to get the needed counting sequence from the INC is an *immediate failure* since it is the input of the INC which is connected to the ROM address. The immediate failure is one of the two conditions for terminating a signal-requirement propagation; the other is *immediately satisfiable* which will be described later. The only alternative for providing a counting sequence to the ROM is from the output of the uPC.

In general a component can satisfy a signal-requirement in three different ways, as outlined below.

- *Pass the signal-requirement further.* For instance, one way to provide the counting sequence to the address of the ROM is to get it to the input of the uPC then pass it through the uPC, as depicted by the central part of Figure 3.2 with bolder lines.

- *Satisfy it without passing any further.* For instance, modify the the uPC to a shift register, connect the new shift input to some pin of the chip and shift in the needed counting sequence from off chip, as represented by the left part of Figure 3.2.

- *Satisfy it jointly with other components.* For instance, the uPC can work with the INC as a counter to provide the counting sequence, provided that the MUX always connects the input of uPC with the output of the INC, as in the right part of the Figure 3.2.

The first method is just to propagate the signal-requirement, sometimes called *line justification* in the testing community. Note that the direction of signal-requirement passing for a "supplying" type of signal-requirements is different from the direction of signal flow, while the directions agree for an "observing" type of signal-requirements.

Possible changes may happen to the signal-requirement to account for the behavior of the component it passes through. An inverter for instance passes a signal-requirement inverted. The signal-requirement might also be split into several new signal-requirements if the component behavior so demands. For instance, an adder may pass a signal-requirement from its output to its two inputs split in the following manner: one input receives a signal-requirement equal to the original signal-requirement and the other input receives a signal-requirement of constant 0; these two add up to the needed signal-requirement.

The second method defines the successful termination of a signal-requirement propagation. A signal-requirement propagation ends successfully if the new signal-requirement in the propagation step is considered as *immediately satisfiable* by the system.

There are two categories of immediately satisfiable signal-requirements. The first includes signal-requirements that are already known to be satisfiable, either from the results of a previous propagation or from the knowledge the system has. The second includes all signal-requirements on control inputs or control outputs of any component since propagating signal-requirements into control circuitry needs design knowledge which our system does not have (see §2.3.8, the "ignoring control changes" heuristic). For the first category of signal-requirements information about how they are implemented is recoded for later retrieval.

The third method searches for groups of components working together as a *compound component* to provide the signal-requirement in question. For instance, a counting sequence can be satisfied by a compound component counter consisting of a register and incrementer connected in certain way. The difference between this method and the first one is that while the first specifies a signal-requirement as the goal of further search, this method specifies as the goal the subcomponents in a compound component and all the interconnections.

As it constructs the local solution, two kinds of circuit modifications are identified by the system but left for the Designer (§2.3) to implement: function augmentations to components (§2.3.8) and control signals needed in testing mode (§2.3.8). Implementing the first kind of modification can be viewed as substituting more powerful components for the original ones. The substitutes may need customization in order to satisfy requirements of speed, load, etc. These requirements may be known only to the Designer and may need trade-off globally. The second kind of modification needs even more global information than the first to implement or estimate its impact.

Since, the system is not supplied with this information, it only records the required control signals and waits for the Designer (currently assumed to be human, or some specialized CAD subsystem).

All three methods for handling signal-requirements may result in either or both kind of modifications. When passing the signal-requirement requires the component to perform a function which is not originally implemented, the first kind of modification is identified. In Figure 3.2, an example for this kind of modification is the node marked "Make and Use uPC as shift reg" since the uPC is originally only a latch. When a signal-requirement propagation ends at a control signal-requirement, a modification of the second kind is identified as depicted in Figure 3.2 by "triangle" nodes in a square frame.

The tree produced by the first stage is in fact a directed graph, allowing us to deal with signal path loops. For instance, the counting sequence signal-requirement received at the output of the uPC may pass through the uPC, the MUX and the INC to become another counting sequence signal-requirement at the input of the uPC again. The system terminates such propagation by (1) identifying a signal-requirement as a triple of direction, signal name, and location, e.g. {*Supply, Counting signal, at ROM's address*}, and (2) maintaining only one copy of a signal-requirement for the whole circuit and pointing all subsequent copies to the same node in the graph. As illustrated in Figure 3.3, the signal-requirement S_1 comes to circuit $Node_1$, is passed around in the circuit as S_5 and S_4, and finally comes back to $Node_1$ as S_2. When S_2 is equal to S_1, the propagation is terminated and any solution to S_1 is a solution S_2, subjected only to the looping constraints in §3.2.3. This not only prevents endless propagation, but also shares the results of signal-requirement propagation: if later on $Node_3$ receives S_3, it can be passed to $Node_2$ via $Node_1$ to become S_6 without new propagation.

The resulting graph can be viewed as a signal flow representation of the original circuit, with all the signals rooted from the goal of making the circuit testable.

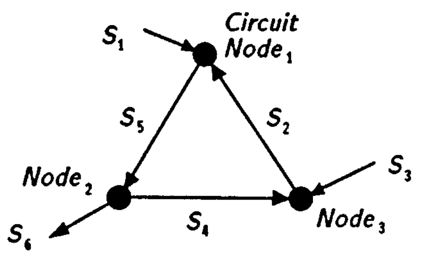


Figure 3.3: Loops and sharing in signal-requirement propagation

However we will keep calling this graph a tree in later discussion so that we can conveniently talk about the "level" of nodes to explain certain ideas.

To summarize, constructing local solutions is a process of breaking goals into subgoals until solutions for the subgoals are readily found. Most of the subgoaling is concerned with signal-requirement propagation, i.e., with controllability and observability. The result of this stage is a directed And-Or graph which records (1) the relationship among signal-requirements and higher level goals of implementing component test patterns and the circuit test, (2) the relationship among the signal-requirements themselves and the solutions for them. A component can handle a received signal-requirement in three different ways, namely pass it, satisfy it or satisfy it jointly with other components. Two kinds of modifications are identified at this stage: function augmentation and control signals for testing mode. Signal-requirement propagation is the responsibility of the reference test generator, while identifying modifications to keep the propagation going is the responsibility of the DFT modification suggestor.

3.2.2 Constructing Global Solutions

At this stage, local solutions are assembled into global solutions and global consistency is checked. In terms of the solution tree from the previous stage, the system tries to find subtrees that are complete solutions to the testing problem of the whole circuit. To put it more formally, the system tries to find alternative subtrees, each of which meets three conditions: (1) the subtree includes the top node of the solution tree, (2) each *and-node* in the subtree has all its subgoals achieved, and (3) each *or-node* in the subtree has one of its subgoals achieved. This section will focus on the process and leave the consistency issue to §3.2.3. Figure 3.4 is an abridged version of the tree produced in the previous stage, in which only some top level nodes are shown to illustrate three different phases of this stage, that is, satisfying signal-requirements at the I/O of foci, testing components, and testing circuits.

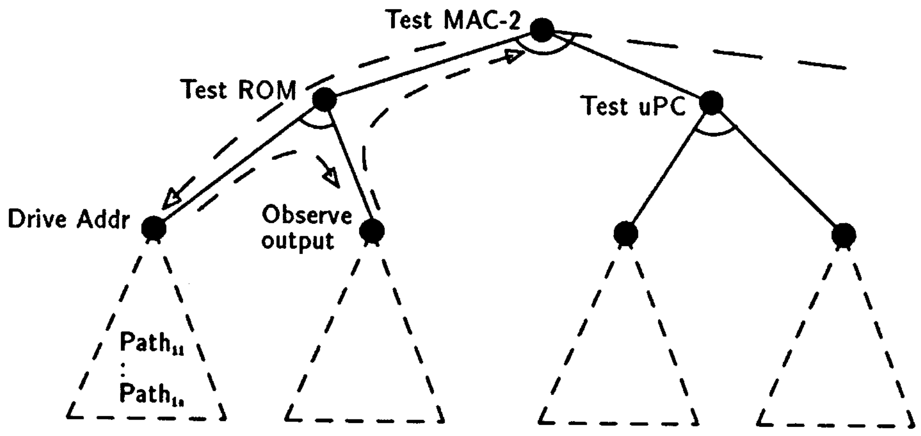


Figure 3.4: Assembling global solution

The first phase is denoted by the signal-requirement nodes directly derived from component test patterns, i.e., the first level of signal-requirement nodes from the top of the tree. When in this phase, the system is searching under that level in the tree to identify subtrees with all leave nodes being immediately satisfiable signal-requirements. These subtrees correspond to data paths and the sources or sinks to which the data paths are connected. Figure 3.2 contains three such subtrees for providing a counting sequence to the ROM address, namely (1) connecting to the compound component counter of uPC, MUX and INC, (2) connecting to uPC working as shift register, and (3) connecting to the new input of MUX via uPC and MUX. Figure 3.14 shows the circuit for these three data paths.

In this stage, the knowledge about satisfying constraints among conjunctive goals and grading alternative solutions is involved. This knowledge is critical to generating meaningful global solutions and is distributed among the three phases. The constraints associated with the first phase are concerned with resource contention within a single data path.

The second phase is denoted by the goals at the level of testing a component, the second level in the tree. In this phase, with the data paths for fragments of a test pattern available, the system tries to put these data paths together to form a complete test for a component. The constraints associated with this phase are concerned with resource contention between different data paths.

The third phase is denoted by the top goal, testing the circuit. In this phase tests for individual untestable components are put together for the whole circuit. This phase is only concerned with the cost of the solution or preferences between solutions rather than validity. To simplify the terminology and the evaluation function described in §3.2.3, both cost and preferences are implemented as constraints on the validity; the constraints are made flexible in order to capture preferences.

We have presented the three phases of global solution construction in a bottom-up order. In the real process, the system proceeds top-down. As shown in Figure 3.4 with the MAC-2, the system traverses the tree in a depth-first fashion, so the three phases overlap. The system first tries to find a test for ROM. Since the test is not available at the beginning, the system starts to work on it. Then for a similar reason the system turns to implement the signal-requirement of providing a signal at the ROM address. After working out the path and source to provide the signal-requirement, the system goes back to test the ROM and is dispatched to look for a path and sink in order to observe the output of the ROM, another signal-requirement as specified by the test pattern for ROM. The system thus goes back and forth until it gets all the paths and sources or sinks needed by the test pattern. After that the system returns with one solution for testing the ROM.

At this point, the system determines whether any remaining untestable components turn out to be testable in view of the new modifications. If so those components can be eliminated as test foci. Then it selects another untestable component waiting to be processed. When a test for the whole circuit is found, the system will try to find another solution until enough alternatives are found.

The order in which the system processes the foci plays some role in reducing the search space, since some of the foci may get eliminated in the process. Our speculation is that the best order depends on the location and nature of each focus, for instance, a focus not connected to other foci may not be eliminated or eliminate others, therefore, it can be processed at any time, while a focus on the data path used to test others is more likely to be eliminated and therefore should be processed later. The current implementation has a primitive ordering function based on the type, presence of control port, and size (see *Selecting the Focus to Work on*, pp 95).

3.2.3 Constraint Relaxation: An Evaluation Function

The system uses a constraint relaxation mechanism to prune that part of the search space that yields obviously unfavorable or invalid tests. This thesis research emphasizes the constraint organization and the relaxation mechanism; the domain specific constraints themselves need further work. This section tries to make the main ideas stand out by omitting many details. A more detailed description can be found in §3.3.

Constraints: Validity and Preference

Constraints in this system capture two kinds of knowledge: the validity of tests and the degree of preference among valid tests. Constraints are divided into different categories according to their applicable test construction phases and their logical relationship. Table 3.1 is a list of the constraint categories implemented.

The "phase" column in Table 3.1 shows the conceptual applicable phase and then the actual application phase in the implementation if they do not agree. This discrepancy is only an implementation phenomenon. The system simulates an "early

Category	Phase	Concern
propagation	1	Signal propagation on one data path
protect focus	1	Using focus to test itself
loop	1	Intersection on one data path, i.e., looping
same side intersection	2, 1	Intersection of control or observe paths
both sides intersection	2, 1	Intersection of control and observe paths
sharing modifications 1	2, 1	Compatibility of component modifications
sharing modifications 2	3, 1	Compatibility of c. m. with other foci
sharing controls	2	Compatibility of control for test segments

Table 3.1: Constraint Categories

commitment" strategy demonstrated by human experts: deciding on a solution for a focus before working on the rest of the foci (see *Selecting the Focus to Work on*, pp 95). This strategy requires that the constraints are checked as early as possible to stop unfavorable searches.

Each category of constraints is organized as a scale ordered according to relative strictness. Table 3.2 shows an example.

	"Loop" Constraints
Category:	Same Path
Applicable time:	During first phase of second stage
Concern:	Data path from focus to one sink or source
Prefer:	Data path with no loop or shorter loop
Prevent:	Contention from reusing components
Scale:	use-component-once (most strict)
	use-component-once-except-for-focus
	loop-signal-changing&resolvable
	loop-signal-stable-with-purpose&resolvable
	loop-resolvable (validity boundary)
	pass (least strict)

Table 3.2: Constraint scale: same path

The constraints in a scale have a total order according to their strictness: If a solution fails a constraint, it fails all the constraints in the same scale which are stricter. Therefore the violation of constraints in one category can be characterized by the least strict constraint violated.

The constraint hierarchy was constructed in three steps:

- Collect all the constraints which can be applied to the solutions.

- Sort the constraints into categories according to their concern, applicable time. There should be a rough strictness order among the constraints collected into the same category to make the redefinition easy.

- Adjust the constraints so that the constraints in each category have a perfect strictness order. This can be done by re-defining each constraint. For instance, if the constraints are represented as predicates of and-ed clauses, we can twist the clauses and the constraints so that the constraints in the same category have a strict subset order in terms of their clause sets. For instance, the "use-component-once" constraint is stricter than the "use-component-once-except-for-focus" constraint. These two constraints can be defined respectively as follows:

$$UseNoneFocusOnce \wedge UseFocusOnce$$
$$UseNoneFocusOnce$$

There are three constraints on each scale worth special notice: the strictest constraint on a scale, corresponding to most favorable solutions; the least strict one, equivalent to no constraint most of the time; and the validity boundary somewhere in between which is the least strict one that still guarantees solution validity.

Maintaining Constraints

All the constraints are concerned with solution characteristics. A natural way to manage the constraints is to associate a constraint set with each of the goals in the DFT oriented TG goal tree. To associate a constraint set with a goal is to keep an individual record for the goal regarding the constraint set, the constraint relaxation history, the solutions, and the constraints that each solution satisfies. The constraint set for a goal includes all the constraints applicable to assembling its solutions from the solutions of its subgoals. For instance, the constraint set associated with the goal of testing a component includes every category in Table 3.1 while the constraint set associated with the goal of satisfying a signal-requirement includes all except the category "sharing controls." The constraints for a goal are relaxed for individual solutions. For example, the solutions *ROM-1* and *ROM-2* in Figure 3.16 do not have data path intersection while the solution *ROM-3* does, that is, the three solutions satisfy different constraints.

Maintaining a constraint set for each goal is one extreme; maintaining only one constraint set for the whole circuit, i.e., only one for the top goal, is the other extreme. There are two conflicting issues about constraint set maintenance: sharing the constraint checking results and simplifying the maintenance. If constraint sets are maintained only for high level goals, sharing is difficult for low level goals since they do not have the records for their solutions and constraint checking results hence constraint re-checking is needed when their solutions are referred to more than once. However, when solution reuse is made possible, the system has to keep different constraint sets for a solution since the constraints vary with different reusing contexts. For instance, providing an exhaustive sequence at the input of the uIR in Figure 1.1 may help test both the uIR and the ROM; however, constraints such as "protect focus" have complete meanings for the two foci. The system also has to keep track of consistent constraint versions for different solutions even if the goal-subgoal relationship does not change. For example, when the solution for testing a component is changed, the semantics of "sharing modification 2" changes accordingly. Another issue concerned with associating constraint sets for each goal is that the solutions and the constraints for adjacent goals in the goal tree are dependent. For instance, the solutions for a signal-requirement before and after propagating through a component are similar and the constraints for them are often identical. This could result in either redundant information or a complicated mechanism to eliminate the redundancy.

The current implementation emphasizes the simplicity of maintenance; the constraint sets are maintained only for the goals of testing components. These constraint sets are initiated according to the types of foci, e.g., whether a focus can be used to test itself. Each constraint set is divided into two parts: one part is for satisfying signal-requirements, the first phase of the global solution construction; the other for testing components, the second phase. An ordinary evaluation function (§3.2.3) is employed for combining component-level solutions (the third phase, a trade-off oriented task).

Relaxing Constraints

As described in §3.2.2 tests for the whole circuit are accumulated gradually. With respect to constraint checking, the building block of tests is *component usage*. A component usage corresponds to the test generation step of passing a signal-requirement through a component and deciding what the working mode for this component should be. For instance a building block in one ROM test of the MAC-2 is to use the micro instruction register in shifting mode, shifting out the ROM's response. Whenever such a building block is added onto a partially constructed test, the system checks whether together they break any of the current constraints. If so, the development of this partial test is suspended.

When there are less alternative tests than desired under the current constraints, the constraints are too restrictive and will be relaxed one at a time unless every category is already stretched to its validity boundary. Usually, the system relaxes each constraint only up to its validity boundary. However, if the system can not generate any solution within the validity boundaries, it will relax the constraints further to generate some partially valid solutions, i.e., ones that are incomplete or violating some critical constraints, in order to show remaining testing problems.

Relaxing Order

When the time to relax constraints comes, the system gathers all the violated constraints from suspended partial tests and picks one of them to relax. The constraints are divided into two groups, one applicable during phase one, the other applicable during phase two, as shown in Table 3.1. The system defines a total order for each of the two constraint groups: It relaxes constraints according to the constraint category order from the bottom up as in Table 3.1, and within a category relaxes the most restrictive one first.

The reason for the order within a category is simple: Since the more restrictive the constraints are, the more favorable the solution will be, we want to keep the constraints as strict as possible. And in a category, the constraints have the logical relationship as described in §3.2.3; relaxing a constraint automatically relaxes all the constraints that are less restrictive. Therefore by relaxing only the most restrictive violated constraint we can avoid relaxing the constraints more than necessary.

The principle for the category order within each of the two groups is to relax the category corresponding to the latest stage of the test generation first. The reason for doing so needs some analysis: The constraint relaxation mechanism is used to cut off invalid or obviously inferior solutions when the system tries to generate enough alternative solutions for the designer to consider. So we only need to relax constraints when there are not enough solutions, or none at all. Getting any solution is the primary purpose of the constraint relaxation, whereas controlling the quality of the solution is secondary. Actually this system itself does not have enough knowledge to judge the quality of solutions, i.e., to trade-off among conflicting requirements. It considers this the task of the designer. On the other hand the system is implemented in such a way that different constraints apply to different search stages and later stages depend on the results of earlier stages. Therefore, violations of constraints at late stages imply that there probably are enough partial results for the late stage to digest; thus the constraint violations at later stages are probably responsible for holding things up. In spirit this is similar to the depth first search, i.e., alternatives are tried after getting a complete solution. Improper ordering could result in an unwelcome situation: some constraints might be unnecessarily relaxed before the key constraint; then, unchecked by the constraints which are unnecessarily relaxed, the system would produce many inferior solutions. The justifications for the ordering in Table 3.1 can be found in *Constraint Ordering*, §3.3.5.

The relaxing order between the two groups is not fixed. The system uses the following procedure to decide which group to relax first, the constraints on embedding test pattern (TPC), i.e., the phase two group, or, the constraints on embedding test segments (TSC), i.e., the phase one group:

- Initially, no category will not be relaxed beyond the validity boundaries.
- If there are enough solutions for test segments but not enough for test patterns, and there are TPC which are violated but are relaxable, one of the TPC will be relaxed.
- If the above is not the case, it is possible that there are not enough solutions for test segments or that there is no relaxable TPC. In either of the situations, if there are relaxable, violated TSC one of them will be relaxed.
- If the above condition is not met either, that is, there is no relaxable, violated TSC, the TPC will be relaxed if possible.
- If the system cannot generate enough valid solutions, partially valid solutions will be sought, and the constraints will be relaxed in a similar order.

Future research could usefully focus on improving the principle used in ordering constraints by analyzing which constraint is really responsible, in the same spirit as the ordering between the two groups, not just by relying on a fixed order based on the above general observations.

Condition: Centralized versus Distributed

The constraint relaxation mechanism controls two aspects of a solution: validity and preferability. This and the next sections compare this mechanism with ordinary condition checking and preferability evaluation, respectively, with regard to these two aspects.

From the solution validity aspect, the constraint relaxation mechanism can be viewed as a predicate with each constraint category serving as an and-ed clause. The system generates solutions incrementally in order to take the advantage of the fact that when a clause is violated in a partial solution, it is violated in any solutions built on top of the partial solution (monotonically necessary). Each clause is applied to a partial solution as soon as it becomes applicable and the partial solution is suspended as soon as a clause is violated. Therefore, no efforts will be wasted to grow any invalid partial solution into multiple invalid solutions.

In the generate-and-test (GAT) paradigm, by comparison, *all* the clauses are applied *after* a candidate solution is generated. When the number of candidates is large and when most of the candidates are not valid solutions, the test-after-generate paradigm generates more invalid candidates. And it also repeats some unnecessary tests only to catch all the invalid solutions grown out of a single invalid partial solution. Our mechanism is a test-during-generation variation of the GAT paradigm; it gains efficiency by reducing the granularity of generation.

In formulating the constraint relaxation mechanism, we have touched some issues relevant to incremental condition checking in general: how often the condition should be checked (the building-block size problem); how the condition should be organized (the and-ed clause concept and the constraint category and scale); and some requirements on the domain (the monotonically necessary condition).

Evaluation: Symbolic versus Numerical

Given the total order over all the constraints, the constraint relaxation mechanism is similar to any other ordinary numerical evaluation function. However, there are different mind sets and applications behind them. An ordinary numerical evaluation function gives back only a number which does not preserve the information about what the grading is based on. Usually a single score corresponds to a number of different situations. This is advantageous in trade-off oriented applications. When different situations are considered as equally desirable but not directly comparable, we can map them onto the same score and solve the problem in comparison.

For the DFT system, however, constraint relaxation has its merits. A DFT system not only needs to make subtle trade-offs between incomparable or conflicting requirements, but also needs to guarantee certain properties of the solutions. For the purpose of guaranteeing certain solution properties, the constraint relaxation mechanism is more suitable than ordinary evaluation functions.

Moreover, the paradigm of "DFT in the context of TG failure" can be naturally viewed as a constraint relaxation process in a large scale: if a test can be generated without circuit modifications, there is no need for the DFT process, i.e., the TG can be restricted to have no circuit modifications. When a circuit is handed over to the DFT system, this constraint no longer holds for the whole circuit, but it might hold locally, i.e., it might hold for a subtask of a testing objective which does not meet with this constraint as a whole. The constraint relaxation concept is viewed by testing experts as the key to low-overhead DFT and is naturally expressed with our constraint relaxation mechanism.

In general, the constraint relaxation mechanism is a kind of symbolic evaluation function which represents its grading explicitly as violated constraints instead of a numerical score. It is better suited for applications that demand certain characteristics from their solutions. On the other hand, an ordinary evaluation function is better suited for applications that emphasize trade-offs among considerations that are neither demanding nor directly comparable.

Summary: The Constraint Relaxation Mechanism

The constraint relaxation mechanism is a kind of symbolic evaluation function, which is suitable for control-oriented evaluation. On the other hand numerical evaluation functions are better for trade-off oriented evaluation. This system uses the former mechanism to reduce the search space in some parts of the process, and it captures two kinds of knowledge: validity and preference of solutions. The system divides constraints into categories of scales according to the knowledge involved and the application phase in the DFT process. The validity is represented as validity boundaries while the preference is represented as relaxable scales. The purpose of relaxing constraints is to find a given number of solutions. The constraints related to later DFT process stages or those that are more restrictive are relaxed first. Invalid solutions may be presented to provide hints about remaining problems.

3.3 The Representations

3.3.1 Circuit: Block Diagram

Circuits are specified at the block diagram level. Figure 1.1 is an example of such a representation level. The blocks in a circuit diagram are called *components*. Between components there are *connections*. When the circuit is running, there are *signals* on connections. The rest of this section will describe these circuit elements.

Connection

The representation for connections is straightforward. The system uses assertions to specify existing connections between I/O ports of components. All connections are considered bidirectional, but I/O ports in general are not. The syntax of the connection assertions is:

```
(CONNECTION (<component-1> <port-1>) (<component-2> <port-2>))
```

As an example, the connection between the address of the ROM and the output of the uPC is represented as[2]:

```
(CONNECTION (ROM Addr) (uPC Output))
```

Signal: Signal-Requirement

Conceptually this system deals with signal sequences which span a continuous time period. We have copied the signal sequence vocabulary of DFT and test generation experts. However, to reduce unnecessary computation we want a sequence to cover as long a time period as possible. The length of a sequence should allow a component to process it in a single working mode, i.e., under the same control signal. If a sequence cannot be so handled by any type of components we will split it into small pieces.

[2]Technical detail: Assertions are not symmetric in the sense that assertion `(CONNECTION A B)` is not equivalent to `(CONNECTION B A)`. To eliminate the trivial reasoning of `(CONNECTION A B)`$\rightleftharpoons$`(CONNECTION B A)`, the system actually uses two assertions to specify one connection, for instance, there is also a assertion `(CONNECTION (uPC Output) (ROM Addr))`.

There is no formal definition for "same control signal," here too we follow human experts' practice.

The collection of signal sequences the system uses comes from two sources: (1) those in test patterns, and (2) those that can be transformed from the signal sequences in test patterns by some components. Or, to put it formally, it is a set of signal sequences which (1) includes all the signal sequences in all the test patterns, and (2) it is closed under all the mapping functions of circuit components.

In a DFT system, every signal sequence should be relevant to a test. Some signal sequences are to be generated and others are to be observed for testing a circuit. Since those two kinds of requirements are treated symmetrically in our system, both are called *signal-requirements.*

The current representation of signal-requirements can be viewed as a triple: location, direction and signal specification. Location is denoted by a component port; direction specifies whether the signal is to be generated or observed; and the signal specification carries the signal's characteristics. To shorten the assertions of the signal-requirements, the direction is implied in the current implementation by the direction of the component port: if it is an input, the signal is some response to be observed, otherwise it is to be generated. The syntax of signal requirements is:

```
(SSR <component-name> <port-name> <signal-name>)
```

In the current implementation a signal specification is a name that is associated with the information needed to propagate signal-requirements and to specify sources or sinks. As to a fine-tuned conflict resolution and cost analysis, a signal name does not provide enough information, such as the length or the boundary condition of a signal sequence. Since this system does not emphasize these tasks, the signal specification refinement is left as future work. It could be done by using a hierarchical specification to work at different abstraction levels, and designing necessary interpretation functions for it. We believe the frame work used in current implementation will be compatible with the refined signal specification, for instance the new specification could have signal name as one of the abstraction levels.

Another experiment that could have been done is to use a signal specification of even higher abstraction level — every signal is treated the same, i.e., the differences among different signal sequences are ignored. We speculate that at this signal specification level, although it uses only the most primitive signal propagation knowledge, i.e., does not use signal-requirement specific knowledge such as a counting sequence can be generated by a counter, the system will still preserve most of the power it has now. The reason lays in the characteristics of available DFT techniques. Most DFT techniques reduce foci to combinatorial components and provide sources and sinks capable of testing arbitrary combinatorial foci; and most data paths between source, focus and sink, carry signal sequences unchanged. Of course without details about signal sequences, the system has to be more conservative. This might lead to higher DFT fixing overhead.

In general, using what level of data abstraction depends on the purpose of a system, and also depends on the knowledge available to the system for manipulating the data. For identifying relevant modifications in a DFT system, the signal name appears to be a quite satisfactory level of signal abstraction.

Component

Preferences for Components

There is no restriction on what can be a component as long as the component behavior can be specified. However we prefer the component to be *small enough* to expose testability problems that are readily fixed and also to be *large enough* to simplify the circuits given to the system in order to reduce computation complexity. To be specific, we prefer to see all the registers or other state components as individual components since they usually cause testing problems and most DFT methods modify them to solve testability problems; otherwise components should include as many transistors as possible. The number of I/O ports should be restricted, however. Larger components mean fewer propagation steps and fewer ports per component mean fewer conjunctive goals in the signal propagation, both simplify the signal-requirement propagation. Here *I/O port* refers to a group of I/O wires which are conceptually considered as one port; for instance, a 32-bit register has a data input port which consists of 32 wires.

Three Attributes of Components

A component in this system is identified by its *type*, *functionality sets*, and *ports*. *Type* specifies a template for a class of components, which defines the testing-oriented behaviors of components. *Functionality sets* and *ports* specify a particular component, i.e., an instance of a type. For example, the uPC in the MAC-2 is specified by the following facts:

```
(TYPE uPC REGISTER)
(PORT uPC input INPUT DATA DATA-INPUT 8)
...
(USER-FUNCTION uPC REGISTER LOAD)
(EXTENSIBLE-FUNCTION uPC REGISTER LFSR)
...
```

These fact declarations say that the `uPC` is a component of type `REGISTER`; one of its ports is named `input` which is an `INPUT` port, of port type `DATA-INPUT`, of port category `DATA`, and `8` bits wide; its function for normal circuit operations, specified by `USER-FUNCTION`, is {`REGISTER LOAD`}; and one of its extensible functions, specified by `EXTENSIBLE-FUNCTION`, is {`REGISTER LFSR`}. The details concerning these attributes will follow.

Component Port

The syntax of component port declaration is:

```
(PORT <component-name> <port-name>
      <port-direction> <port-category>
      <port-type>      <port-size>)
```

Each port in a circuit is identified by the name pair of the component and the port. The functional aspects of a port can be specified by the type pair of the component and the port. For instance, the name pair for the input port of the uPC is {`uPC input`} and the type pair is {`REGISTER DATA-INPUT`}. The name pair is used for describing connection, while the type pair is used for signal-requirement mapping.

Other attributes associated with a port are size and category. The size attribute is used for checking the signal path capacity and analyzing the modification cost. The category attribute specifies whether the port is considered a control or data port. These two port categories are treated differently in this system: signal propagation ends successfully on control ports but not on data ports since we assume that control signals can be implemented by the Designer (see §3.2.1 and §2.3). The classification of the port category usually follows the human experts' convention, but some types of ports are classified as control ports despite the fact that human experts might consider them data ports. For instance, the shift-in and -out ports of registers are considered control ports by this system, though human experts would consider them data ports. These ports are only a few bits wide, often result from DFT modifications, and are considered to be difficult for the DFT system to trace them to the end. From the trio view about the chip design process, it is better for the Designer to implement the requirements concerned with those ports than for the DFT system to implement the requirements or to verify whether they can be accommodated by the circuit without change.

Our decision on which signal-requirements to propagate reflects the way to divide the whole circuit design task among the Designer, the Test Generator and the DFT suggestor according to the trio concept. Accordingly, the *source* and *sink* are redefined as the components where signal-requirement propagations end (compare with definitions in §2.1).

Component Type

The component type specifies a set of I/O signal-requirement mappings over the I/O ports for each of the signal-requirements the type of components might handle. The specification is written in declaration format. The syntax is:

```
(TESTING-COMPONENT-USAGE     <component-type>
   <receiving-port-type>     <received-signal-requirement>
   <other-port-type>         <passed-signal-requirement>
   <component-working-mode> <component-purpose-in-testing>)
```

Since one signal-requirement may be mapped to different signal-requirements on different ports simultaneously, a complete mapping is specified by a group of declarations that only differ on the `other-port-type` and `passed-signal-requirement` fields.

There is a group of such declarations for every quadruple of

```
{<component-type>
 <receiving-port-type>
 <received-signal-requirement>
 <component-working-mode>}
```

when the mapping exists. As an illustration, the following are two groups of signal-requirement mappings for a register.

Example 1. For {register data-input ?any load}, i.e., for observing `?any` kind of signal on the data-input port of a register working at loading mode, the register needs a control signal `(constant load)`[3] on the control port as specified by the first declaration below, and it needs a signal-requirement of observing `?any` at its data-output port specified by the second declaration. Since the register is passing the signal-requirement `?any` from the data-input to the data-output, the register is "passing-response":

```
(TESTING-COMPONENT-USAGE register
         data-input      ?any
         control         (constant load)
         load            passing-response)

(TESTING-COMPONENT-USAGE register
         data-input      ?any
         data-output     ?any
         load            passing-response)
```

[3]`(constant load)` specifies the requirement for a steady control signal appropriate for the "load" mode.

Example 2. A register can generate `?any` signal on its data-output port if its control signal sets it at `shift in` mode and its `shift in` port is supplied with the `?any` signal. This knowledge is encoded as the following two declarations corresponding to the quadruple {register data-output ?any (shift in ?left-or-right)}[4]:

```
(TESTING-COMPONENT-USAGE      register
        data-output           ?any
        control               (constant (shift in ?left-or-right))
  (shift in ?left-or-right)   generate)

(TESTING-COMPONENT-USAGE      register
        data-output           ?any
  (shift in ?left-or-right)   (serialize ?any)
  (shift in ?left-or-right)   generate)
```

In these examples, an identifier beginning with a question mark denotes a variable. All the variables in the same group of `TESTING-COMPONENT-USAGE` with the same name must unify in the same instantiation.

`Port-type` can be defined by several attributes. For example, a "shift left input" port is represented as `(SHIFT INPUT LEFT)`. This structure can be used hierarchically to overlook some attributes when appropriate. For instance, when using the existing shift-in function of a register to provide a testing signal, it will cost less to follow the direction already implemented, i.e., we do not have to add "shift left input" port and corresponding working mode when we have a "shift right input" register; but when the shift function is new to the register, whether it is shift left or shift right does not make much difference. Therefore, some fields like the `right-or-left` field in the shift I/O port type are left unspecified in the mappings. The underlying need for doing so is that we want a systematic way of overlooking some details in order to prevent the system from generating trivial alternatives like "shift left" and "shift right."

The `component-working-mode` field has a similar structure to that of `port-type` for the same reason.

[4]The expression *(serialize ?any)* demands a serialization of the parallel signal *?any*. The expression *(shift in ?left-or-right)* means a port or a mode, depending on the context, of shift-in with the direction unspecified.

The last field `<component-purpose-in-testing>` can be one of four: passing stimulus, passing response, generating, observing. This field specifies whether a signal-requirement propagation comes to a completion, i.e., whether the needed source or sink is encountered.

Component Functionality Sets

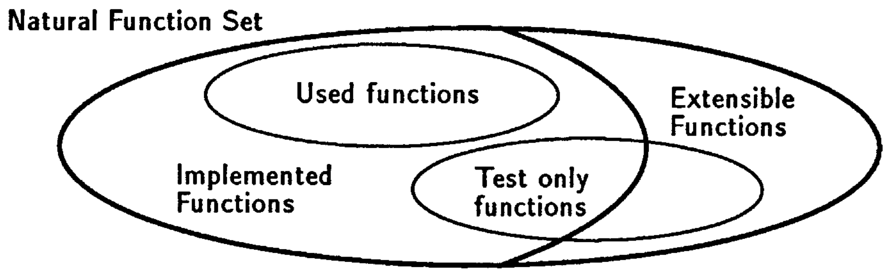


Figure 3.5: Component functional sets

Component functionalities, or working modes, are divided into four non-exclusive categories: (1) those used in normal operations, (2) those implemented, (3) those which are extensible and (4) those used only in testing. The first category includes the functionalities which are used by the designer to meet the circuit specifications. These are the component functions that must be tested. The second category, i.e., "implemented functionalities" includes all those in the first category and redundant ones. Functions of the second category can be used in testing other components without modifying the component. The third category includes functionalities which are not implemented in the particular component but could be built into it cost-efficiently with some modification. The fourth category is the set of functionalities which are used in testing the circuit but not involved in normal operations. The relationship between those sets is illustrated in Figure 3.5. The union of the implemented function set and the extensible function set represents a very important concept to this system: the *natural functionality set.*

The Natural Functionality Set

Intuitively the natural functionality set is the maximum set of functionalities that a particular type of component can have. The natural function set for a component type is defined[5] as the union of all the implemented function sets of components commonly classified as that type.

The concept of natural function set is inspired by the small number of different kinds of component modifications that DFT techniques introduce: they add certain functionalities to certain types of components, but never to others. For instance a shift function can be added to registers as in the LSSD method, but it is never added to a ROM. The speculation is that some functionalities can be implemented more efficiently together rather than separately, i.e., modifying an existing component is more efficient than adding a new component.

The natural function set in our system is DFT oriented: it only cares about those functionalities that help solve testing problems. In other words, the system uses the natural function set to represent those component modifications which are useful for solving testing problems. The natural function set together with the paradigm of "fixing test generation failure" (problem finder) constitutes most of the DFT knowledge in our system.

[5]Performance and economic efficiency should be the ultimate deciding factor about whether a function should be in the set. Suppose we have a new function to be implemented at the place of an existing component of a certain type. If it is more favorable to add the function into the existing component than to add a new component, the function is in the natural function set of that type of components. The precise definition must answer two questions: what is the core function set which defines a type? how can one evaluate the favorableness? Unfortunately except for the empirical method presented here we currently do not have sufficient data to answer these questions. One can avoid the core function problem by using a component library to directly compare whether adding a function is more efficient than adding a component.

Compound Component

A compound component is defined as a group of components configured in a certain way. These components function as a single component in testing. In a test for a focus, there are four kinds of ingredients: the focus, the sources, the sinks and the data paths. In our system, except for the focus (which is a single component by definition), all the others can be compound components. However, for reasons to be analyzed later, we consider that it is beneficial if only sources and sinks are compound components. In the current implementation, we have tested only one compound source — a compound counter.

The Trade-off between being Told and Looking for Oneself

One can go to two extremes in handling the compound component problem. One extreme is to "tell the system everything"; that is, to mark every component configuration in a circuit for the system as a part of the input. The other extreme is to "let the system figure out everything"; that is, not to tell the system anything about the existing compound components in a circuit but let the system figure them out by itself. The trade-off between the two extremes concerns how much energy the system should spend on looking for compound components. There are arguments supporting each of the two sides. From the "telling the system everything" point of view, not every possible compound component in the circuit is useful. The most useful ones are those intended by the designer, so it is easy for the designer to point them out to the system. Making a general definition for compound components could be a formidable problem by itself. For the "letting the system figure out everything" side, one can argue that it is tedious to indicate every compound component and could be a waste of time and energy if most of the compound components are not going to be used. Contrary to the argument that the designer knows every useful compound component, one could easily find compound components unintended for normal operation but useful for testing.

All these arguments have their merits and we have to settle on something in between. The implemented system considers three things in dealing with the compound component problem:

- Define the compound components as only those that help solve the testing problem. Such compound components can be further restricted to those that can be used as sources or sinks since data paths are quite simple and do not warrant the use of compound components. "Whatever you can do with a compound component for a data path you can do with the components of the compound component" is the heuristic. Since the types of sources and sinks are limited in number — there are only a handful in all built-in testing techniques — the types of compound components are limited in number too. Therefore we only have a limited and specific version of the compound component problem to begin with.

- Require using a suitable component level to represent the circuits. "Suitable component level" refers to the highest possible representation level, that is, when a group of components can be represented either as a single component or as a compound component, the single component representation will be chosen. This can reduce the need for recognizing compound components.

- Give the system some ability to recognize compound components but only start looking for a compound component if it could help solve the testing problem at hand. Thus the system only searches for compound components in part of the circuit.

The rest of this section will elaborate on how compound components are represented in and recognized by the system.

Compound Component Specification

```
(1)         Type: Counter
(2)      Purpose: Generate
(3) Handled SRs: Counting, Exhaustive
```

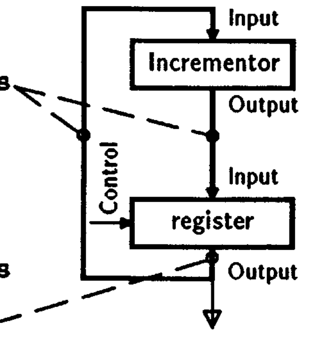
Compound Counter

```
(4) Required
    Components

(5) Required
    Connections

(6) Further SRs

(7) Receiving
    Port
```



Required Component 1

```
Reference name: Incrementor
Component type: Incrementor
Working   mode: Increment
Required ports: Input, Ouput
                (compulsory)
```

Required Component 2

```
Reference name: Register
Component type: Register
Working   mode: Load
Required ports: Input, Ouput
                (compulsory)
                Control
                (optional)
```

Figure 3.6: A compound counter

Compound component definitions specify how to connect element components and coordinate them to perform the desired function.

As an example, Figure 3.6 shows the definition for a compound counter. The compound counter consists of an incrementer and a register connected in a loop. In each counting step, the incrementer reads the content of the register and adds one to it. Then the register stores the new value for next step.

A compound component specification has seven fields:

1. *Compound component type*: This serves as the identifier of the compound component template.

2. *The purpose of the compound component*: This is a tag which indicates one of the following: `generating` or `observing`. Its function is the same as the purpose field in an ordinary component definition, i.e., signaling the termination of a

signal-requirement propagation.

3. *Handled signal-requirements*: They are the signal-requirements that can be handled by this compound component.

4. *Required components*: A required component performs a function which is critical to the compound component. It has compulsory ports that are critical to the compound component, optional ports which are not critical though the compound component cares what signals should be on them if these ports present, and unspecified ports which the compound component does not care about.

   The critical function is specified as a working mode. The critical and optional ports are specified in the same way as the ports for ordinary components except for the component and port name fields which are internal identifiers to be unified with real component and port names.

   For example, in the compound counter shown in Figure 3.6, the `register` is a required component. It must be in `load` mode when working as a part of the compound counter. It has two compulsory ports — input and output — and an optional control port. The control port is optional because it can be absent for a clocked register; the clock signal then acts as the control but is usually suppressed at high level representation. A better example for the concept of optional ports is the carry-in port in an adder: in normal addition, it must be set to 0 if the port is provided. Therefore it is an optional port to the normal addition function.

5. *Required connections*: A required connection is one which is between required components and which is critical for the compound component. It can be viewed as a data path which carries a signal unchanged and it may be made up of components other than the required components.

   The signal carried by a required connection is also specified. If the carried signal is known, then the connection can be specified as "a connection which passes *the* signal unchanged." If the carried signal is unknown, the connection must satisfy a more conservative and stronger condition: "a connection which passes *any* signal unchanged." As a result, the system might miss some valid connections and is left with less options which may result in higher DFT overhead.

Take the compound counter as an example again. The connection between the incrementer's output and the register's input is a required connection. In the circuit MAC-2, this connection is matched to a data path from the INC's output to the uPC's input through the MUX. The signal sequence carried by the connection is a "counting" sequence when the compound counter is activated.

6. *Further signal-requirements for handling the received signal-requirement*: These are the diffused signal-requirements needed for manipulating the handled signal-requirement just as an ordinary component does. For example, when present, the register's optional control port wants a signal of `(constant load)` to work in the compound counter. This is a fan-out signal-requirement of the compound counter when handling the received signal-requirement.

7. *Signal-requirement receiving port*: This is the I/O port to receive the signal-requirement for the compound component as a whole which is a port of a particular required component. For instance, the output port of the register is the receiving port for the compound counter.

In a compound component specification, each of the required components will match a specific component in a circuit; each required connection will match a data path which might contain zero or more components. The components on required connections are part of the "required connections" and should not be confused with the "required components."

Compound Component Matching

The search for a compound component starts when two conditions are satis-

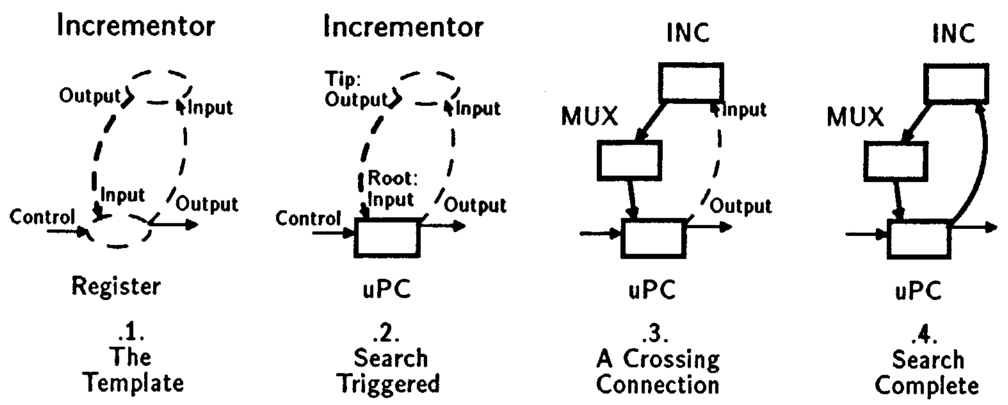


Figure 3.7: Search for the compound counter

fied: a port in the circuit receives a signal-requirement which is in the handled signal-requirement set of a compound component, and the port matches the signal-requirement receiving port of the same compound component. For example, when the system tries to get an exhaustive sequence to the ROM's address port in the MAC-2, a signal-requirement for an exhaustive sequence will be passed to the uPC's output port. This triggers a process searching for a compound counter since (1) the exhaustive sequence is in the handled signal-requirement set of the compound counter (see Figure 3.6) and (2) the uPC output port, the signal-requirement receiving port, is a "register output" port that matches the receiving port in the compound counter template. These two conditions are similar to the conditions for an ordinary component, except that more template matching is needed for a compound component.

When a search for a compound component starts, the required component with the receiving port is already known. In the compound counter example as shown in .1. and .2. of Figure 3.7, the register is bound to the uPC when the search starts. Subsequently, the system repeats the following two steps until all the required components are found:

1. Select a "crossing" connection, place its "seed" signal-requirement on its "root" port and specify its "tip" port. A "crossing" connection is a required connection

between a known and an unknown required component; its "seed" is the carried signal-requirement on it; its "root" is its end-port on the known component; and its "tip" is another end of it on the unknown component.

As shown in .2. of Figure 3.7, the first "crossing" connection of the compound counter is from the input of the register uPC to the output of an incrementer which is not yet determined. The "seed" is "generate counting sequence," the "root" is the input port of the uPC, and the "tip" is the output port of the incrementer to be determined.

2. Instantiate the "crossing" connection by propagating the "seed" through the circuit. When the "seed" emerges at a component port which matches the "tip" port, an instance of the "crossing" connection is found and so is that of the required component at its "tip." The phrase "the seed emerges" means that the emerged signal-requirement must be the same as the seed when it is sown; it does not matter how it changes during the propagation.

   As shown in .3. of Figure 3.7, the "seed" of "generate counting sequence" is propagated unchanged through the MUX to the INC output. A "crossing" connection is completed, and the "tip" component is instantiated to the INC.

   Note that if a "tip" component can also pass the "seed" other than acting as the "tip," the system will also try to use it as an ordinary component and propagate the "seed" through it in order to find alternative "tips" and "crossing" connections.

After the above is accomplished, all the required components are found, as are the required connections on a spanning tree of the compound component as exemplified by .3. of Figure 3.7. The system then switches to a similar repeated two-step process to find the rest of the required connections. This time however, the system is no longer looking for required components, and the "tip" and the "root" of a required connection are both known physical ports.

3.3.2 Test Pattern

Test pattern definitions in this system have a four-level tree structure as shown in Figure 3.8. The test pattern for a component type consists of test patterns for each function, or working mode, of this type of component. A test pattern for a function consists of some *test segments.* A test segment is a building block of a test pattern which has one signal-requirement at most for each component I/O port. Accordingly a test pattern for a component includes at least one test pattern for each of its functions used in the normal operation[6]. The component test pattern can be constructed by (1) picking a test pattern for each function used in normal operations, and (2) taking the test segment union[7] over the selected function test patterns.

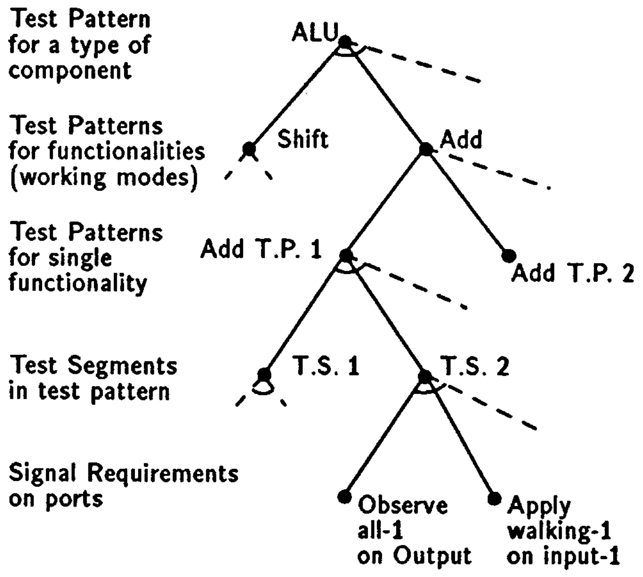

Test Pattern for a type of component
Test Patterns for functionalities (working modes)
Test Patterns for single functionality
Test Segments in test pattern
Signal Requirements on ports
ALU
Shift
Add
Add T.P. 1
Add T.P. 2
T.S. 1
T.S. 2
Observe all-1 on Output
Apply walking-1 on input-1

Figure 3.8: Test pattern structure

[6]See §3.3.1 for definition of functions used in normal operations and see §2.3.8 for the validity of the test pattern so defined.

[7]Viewing each functionality test pattern as a set of test segments.

Test patterns are specified as assertions in the following format:

```
(TEST-PATTERN <focus-type>   <tested-working-mode>
              <test-id>      <segment-id>
              <port-type>    <signal-sequence>
              <working-mode> STIMULUS/RESPONSE)
```

A `TEST-PATTERN` assertion specifies one signal-requirement leaf node in the test pattern tree shown in Figure 3.8. The first four fields indicate the path from the top node to the leaf node in the tree: the `focus-type` specifies the top level node, the `tested-working-mode` a second level node on the path, the `test-id` a third level node, and the `segment-id` a fourth level node. The next two fields, `port-type` and `signal-sequence` specify the signal-requirement: on which port what kind of signal sequence is to appear. Since sometimes a focus has to be driven outside the working mode under testing in order to be tested adequately, the seventh field `working-mode` is introduced. The last field `STIMULUS/RESPONSE` is redundant information for readability, which indicates the direction of the signal-requirement, i.e., whether it is driving the focus or it is the output of the focus to be observed.

The following is a test segment specification for multiplexors. This group of declarations only differs at the fields `port-type`, `signal-sequence` and `STIMULUS/RESPONSE`, so only these three fields are given for the last three declarations:

```
(TEST-PATTERN multiplexor                   select0
     (test-id multiplexor select0 2)        (segment-id 1)
     data-input-0                           walking-0
     select0                                STIMULUS)

     data-input-1 (constant all-bit-1)      STIMULUS
     control      (constant select0)        STIMULUS
     data-output  walking-0                 RESPONSE
```

This segment tests the `select0` functionality, i.e., directing the `data-input-0` to the `data-output`. The stimulus on `data-input-0` is `walking-0` which is to be passed to the `data-output` and to be observed. The `walking-0` sequence is a shifting 0 in a background of 1; an example of this is a set of four bit binary signal `1110`, `1101`, `1011` and `0111`. The `walking` sequence is useful in detecting bridge faults between adjacent lines. While testing, the other input `data-input-1` is driven by an `all-bit-1` constant to see if inputs on this port interfere with the normal operation.

When all the test segments in a test pattern can have the *same control scheme* the details of the test segment are not significant. A test with the *same control scheme* meets two conditions[8]: (a) all the test segments in a component test pattern refer to the same group of ports, and (b) all the signal-requirements on the same port in the test pattern have the same solutions exclusive of the sources and sinks. For example, when an ALU has all its I/O buffers modified to LSSD registers, its test pattern can be modified to meet condition (a), and its test can be done by keeping these LSSD registers shifting to meet (b).

This means that it is a useful abstraction level to specify only what I/O ports of a focus need control or observation, since this can be viewed as a "super test segment" and the system only needs to embed the single super test segment for a focus when the "same control scheme" condition holds.

[8]This is a constraint in the "segments" constraint category, see §3.1.

3.3.3 Generating Tests: Propagation Rules and Test Assembly Procedure

Previous sections on representation have covered the *static* part of the system's test generation knowledge; this includes test patterns, component behaviors and circuit descriptions. This part of the knowledge is treated as facts and represented as assertions in the system. The *dynamic* part — about the test generation process — is represented as inference rules or procedures.

TG and DFT Processes in the Inference Rule Format

Inference rules in an if-then format are used to represent the part of the test generation process that needs only local circuit and signal properties. Here *local* properties refer to those which concern only one or two connected components. All the rules and facts are interpreted by a forward-chaining rule-based system.

The role of the rules in this system is to construct the local solution.

Rules to be shown in this section are rephrased to suppress details and improve readability. Tokens of format `?variable-name` denote variables, and all the variables with the same name, in the same rule, denote the same thing.

Signal-Requirement Propagation Rules

Signal-requirement propagation tasks are encoded as rules:

- *Instantiating test pattern:*

```
IF   (1) ?Component is a Focus,
     (2) ?Component is of ?Type,
     (3) Working ?Mode is in the User Function set of ?Component,
     (5) A ?Port of Port Type ?PT is present,
     (4) There is a Test Pattern ?TP for ?Type and ?Mode, and
     (6) The ?TP specifies ?Signal-Requirement on ?PT
THEN ?Port needs a ?Signal-Requirement
```

- *Passing signal-requirements over connections*:

```
IF   (1) ?Port0 needs a ?Signal-Requirement,
     (2) ?Port0 is Connected to ?Port1, and
     (3) The Directions of ?Port0 and ?Port1 are different
THEN ?Port1 needs a ?Signal-Requirement too
```

- *Deciding how to handle signal-requirements*: This rule determines both the component working mode and the consequent signal-requirements for handling a signal-requirement.

```
IF   (1) ?Port of ?Component needs ?Signal-requirement
     (2) ?Port is of ?Port-type of ?Component-type
     (3) There is a
          (TESTING-COMPONENT-USAGE ?Component-type
             ?Port-type            ?Signal-requirement
             ?Other-port-type      ?Passed-signal-requirement
             ?Working-mode         ?Component-purpose-in-testing)
THEN (1) The ?Component should be set to ?Working-mode
     (2) The ?Other-port of ?Other-port-type
          needs a ?Passed-signal-requirement
```

When it places the `?Passed-signal-requirement` on the `?Other-port`, the rule checks the category of the `?Other-port`. If it is a control port, the `?Passed-signal-requirement` will be considered as granted and prohibited from further propagation:

```
IF   The ?Category of the ?Other-port is DATA
THEN The ?Other-port needs an ordinary ?Passed-signal-requirement
ELSE The ?Other-port is Granted the ?Passed-signal-requirement
```

This rule is also used for DFT purposes: it decides whether modifications are needed, and if so whether new I/O ports are also needed. This can be viewed as second level rules for the action parts (1) and (2), respectively:

```
IF   ?Working-mode is not in ?Component's IMPLEMENTED-FUNCTION set
     but in EXTENSIBLE-FUNCTION set
THEN Request ?Working-mode be added to ?Component

IF   ?Component does not have a port of ?Other-port-type
THEN Request an ?Other-port of ?Other-port-type be created
```

Compound Component Matching Rules

The compound component templates are also translated into rules in order to match them against circuit structures. Each compound component specification is translated into a rule that will start the match process. This rule will define a relay rule and sow the first *seed* for the first required tree connection to start a compound component search process whenever the conditions for doing so are met (see §3.3.1 for more detail):

```
The Compound Component Search Triggering Rule

  IF   (1) ?Port needs a ?Signal-requirement
       (2) ?Signal-requirement is Handled by ?Compound-component
       (3) ?Port matches the Receiving-port of ?Compound-component
  THEN (1) Define the first Relay Rule
       (2) Sow the first ''seed''
```

A relay rule will notice the completion of a `Tree-connection`, then it will act differently according to whether there are more `Tree-connections`, or equivalently `Required-components`, to be found or not:

```
The Relay Rule

  IF   The newly found tree connection is the last one
  THEN (1) Define the Compound component completion rule
       (2) Sow all the ''seeds'' for non-tree-connections
  ELSE (1) Define a new Relay Rule
       (2) Sow a new ''seed'' for a new tree-connection
```

The Compound component completion rule signals that an instance of the compound component is found upon the completion of all the non-tree connections. This rule is defined after all the tree connections and required components have been found:

```
The Compound Component Completion Rule

  IF   All the Non-tree connections are found
  THEN An instance of the Compound Component has been found
```

The *seeds*, sown by the triggering rule and the relay rule to start search for required connections, are represented in the following format:

```
(?Front-port ?Start-port ?End-port-specification ?Required-connection)
```

When just sown the `?front-port` and the `?Start-port` are the same; both are equal to the corresponding port of the required-component just instantiated. The `?End-port-specification` is a list of `Component-type, Port-type, Port-size,` etc., abstracted from the corresponding `?Required-connection`. Note that the `?Required-connection` carries the `Appeared-signal-requirement` on the connection. A `Current-signal-requirement` should have been added to allow the signal-requirement to change on the connection. The current implementation requires that every component on the connection pass the `Appeared-signal-requirement` unchanged. This is weaker than requiring it to pass an *arbitrary* signal-requirement unchanged, but stronger than requiring the signal-requirement at the `?Start-port` and that at the `?End-port` to be the same (see Figure 3.6 and its explanation).

The frontier port of a connection is pushed forward by signal-requirement propagation rules designed for this purpose:

```
The Connection Growing Rule

  IF   (1) ?Front-port is connected to ?C'd-port of ?C'd-Component
       (2) The ?C'd-Component can pass the Appeared-signal-requirement
           from ?C'd-port to ?New-front-port
  THEN  Update the ?Front-port in the seed to ?New-front-port
```

The completion of a required connection is also detected by rules:

```
The Connection Completion Signaling Rule

  IF   (1) ?Front-port meets ?End-port-specification in the seed
       (2) The Appeared-SR on ?Front-port = the Carried-SR
  THEN  A connection is complete
```

The seeds and their growing rules for non-tree-connections are identical to those for tree-connections except that the `?End-port-specifications` of the former are physical ports instead of port type specifications.

According to the description above, the rules for matching compound component templates are *procedural*; that is, they strongly suggest the accumulation process. All of the rules except `Triggering Rules` are generated during the process. Generating rules on the fly can improve execution efficiency because (1) doing so makes the rule antecedent parts as *specific* as possible so that the applicable circuit area is restricted to a minimum and (2) doing so withholds the rules as long as possible so that the burden of managing the rules is reduced. In accordance with this effort to improve the execution efficiency, different signal-requirement representations are used for different purposes, e.g., signal-requirements versus *seeds* in order to localize inferences for different domains.

When implementing the compound component searching process, I was torn between the execution *efficiency* of the process and its data base *consistency* with simple components. The former is an issue because the compound component search is essentially a graph match process which is computationally hard; the latter was an issue because (1) we had built part of the system handling ordinary components and (2) the compound component search shares knowledge of signal propagation and circuit description with other parts of the system. This implementation was also attempted by the rule system I used. The AMORD-like system allows rule definitions and an arbitrary lisp code in the rule action parts; this makes it possible to accumulate results through inheriting the calling environment although the environment itself is somewhat implicit to the rules.

TG and DFT Processes in the Procedure Format

Ordinary procedures are used to present the part of the test generation process that needs global circuit and signal properties — those properties which concern the connectivities of the circuit and usually involve many connected components. The tasks of these procedures include assembling data paths from signal-requirement propagation results and assembling tests from data paths.

Procedures versus Rules

Why are ordinary procedures better than predicate calculus-based rules for such tasks? One reason is that a procedure call has a well-defined environment which indicates the satisfying of many conditions; thus the called procedure can focus on the most critical issues. Without this environment, as in the case of a rule triggering, one has to specify all the relevant conditions explicitly. This is distracting when writing the program, and inefficient when executing it, if the number of relevant conditions is large, as it is in our later solution construction stages.

Sometimes a rule-based system can get by using hierarchical condition structures, e.g., using a condition to summarize a group of conditions. Such hierarchical conditioning provides the necessary decomposition which helps both problem solving (knowledge engineering and program writing) and running efficiency.

There are situations, however, when the relationship between the higher-level conditions and the lower-level conditions forms a tangled graph instead of a neat tree. In these situations, it is necessary to write *many* different summaries of the lower-level conditions and there is little sharing of the summaries among the higher-level conditions. This is exactly the situation in test generation: there are conditions specifying data paths which divide components into groups according to the data paths, and there are also conditions specifying relationships between the paths which simply lump together all the components on all the relevant data paths. For such a situation, the use of flexible data structures to represent facts and procedures to manipulate the data structure is more suitable.

Why Any Rules?

Using both rules and procedures for important tasks in a system is not convenient. This problem in the current implementation is rooted in a decision to use a rule based system at the beginning of the implementation when the nature of the problem was not as clear to me as it is today. The major rationales for the decision were: (1) rule based systems are suitable for prototyping, and (2) the candidate system does not restrict what can be the rule action part and provides easy interface with LISP programming language. As the implementation went on and harder issues were dealt with, the inadequacy of the rule-based system was felt. At the time I wrote the compound component search, I was still preoccupied with how to express the process in a rule format, and failed to consider the suitability of the rule-based system for our task. After the compound component search was finished, it became clear that the rule format was not sufficient for the task. However, I was reluctant to re-implement the system at that time. So the parts of the system in the rule format survived, and later parts of the system treated them as a data base.

The System Flow Chart

Figure 3.9 is the system flow chart. The inputs to the system are a circuit description and a list of its untestable components. First the rule based subsystem is called to instantiate test patterns for these untestable components and propagate the signal-requirements until quiescence. Then the procedures are called to perform all the remaining tasks.

Extracting the Local Solution Tree

To facilitate later processes, at the end of local solution construction the local solutions are extracted from the the rule subsystem and re-formatted into a graph.

Constructing Global Solution

Once the local solution tree is extracted, the system enters the second stage to construct global solutions by traversing the local solution tree.

Selecting the Focus to Work on

Currently the system processes the untestable components in an order determined

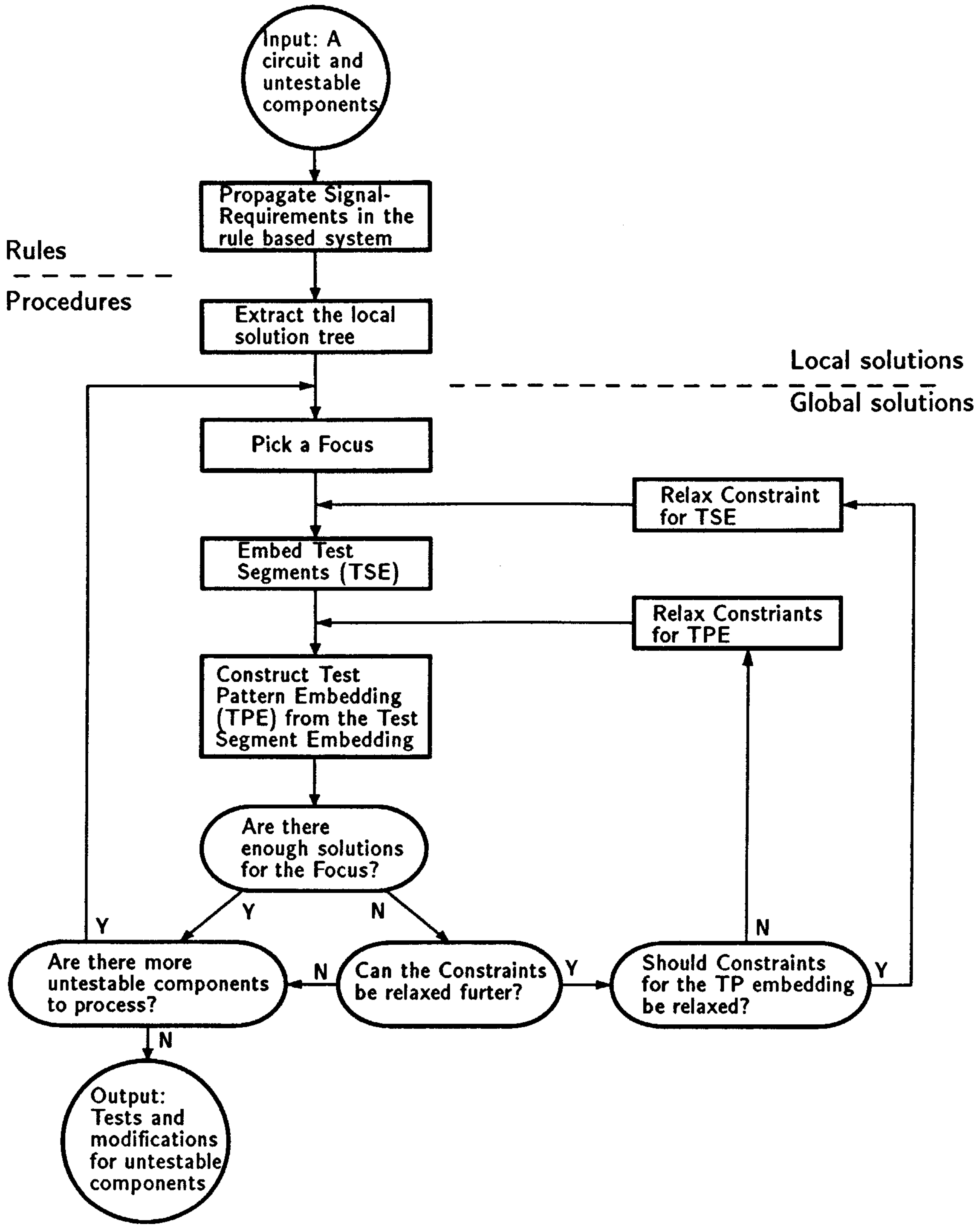

Input: A circuit and untestable components
Propagate Signal-Requirements in the rule based system
Rules
Procedures
Extract the local solution tree
Local solutions
Global solutions
Pick a Focus
Relax Constraint for TSE
Embed Test Segments (TSE)
Relax Constraints for TPE
Construct Test Pattern Embedding (TPE) from the Test Segment Embedding
Are there enough solutions for the Focus?
Y
N
Y
Are there more untestable components to process?
N
Can the Constraints be relaxed furter?
Y
Should Constraints for the TP embedding be relaxed?
N
Y
N
Output: Tests and modifications for untestable components

Figure 3.9: The system flow chart

by the types and sizes of the untestable components.

The motivation is to improve the computation efficiency without missing favorable solutions. The order in which the components are tested is relevant because the components processed first set up a context for the rest and that may affect the overall efficiency of the system.

Currently, when a set of modifications are chosen for testing a focus the system modifies the circuit accordingly and processes the remaining components in the changed circuit. Before picking the next component on the list to process, the system checks if there is any component free-tested or normally-tested in view of the new modifications and the test for the just processed component. Finding free-tested or normally-tested components are considered less computationally expensive than DFT-testing components with new modifications because the differences in tasks or working environment: finding free-tested components needs only *check the records*, finding normally-tested components needs *test generation in a fixed circuit*, whereas DFT-testing components needs *test generation in a changing circuit* (see §3.3.5).

Deciding on certain modifications early and limiting the search space may cause the system to miss the optimal solution. Human experts have the same problem. However they have strong *sense* about which untestable components could introduce critical modifications and which modifications are critical to (or sharable among) testing problems and often get favorable results without massive searching. The system has implemented some of the heuristics for selecting modifications in the evaluation functions (described in §3.3.5), but much work along this direction is waiting for future research.

Some heuristics help decide the processing order of components. Two criteria are often used to justify our heuristics:

- *The dependency criterion*: This criterion says that if there is one-way dependency between two tasks, the dependent should be processed later. Otherwise the computation for the dependent may be wasted.

- *The branch factor criterion*: This criterion is used for ordering chronically dependent tasks. "Chronically dependent" means that the solutions of a task do not depend on the solutions of the tasks processed later, no matter what the order is. For instance, we treat component test embedding as chronically dependent tasks with respect to free-tested component elimination, that is, only unprocessed components are elimination candidates.

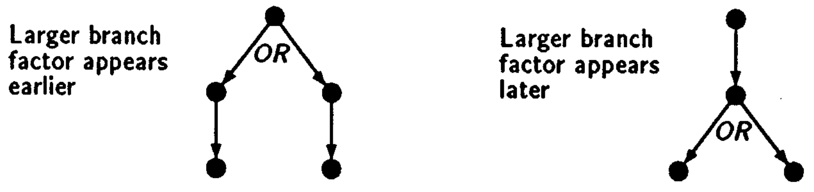

Larger branch factor appears earlier
OR
Larger branch factor appears later
OR

Figure 3.10: The branch factor and the search space

When this computation model is applicable, the search space is smaller if tasks with fewer alternative solutions are processed earlier, as illustrated in Figure 3.10. In the figure there are two problems, one has two solutions and the other has one. If the former is processed first, the latter has to be processed twice, as shown in the left part of the figure. If the order is reversed, as shown in the right part, every one needs to be processed only once.

The following are some heuristics analyzed with regard to these two criteria and whether they can be automated easily.

- Purpose of the untestable components: When components are used for a special purpose, their control schemes and controllers are simplified. For instance, a register may be used for buffering purposes, loading new contents every clock circle, so its control may be tied to a clock signal and *omitted* at a certain representation level. This makes them less helpful for testing purposes since changing their control scheme to a more general one is expensive. In other words, such components are less dependent on the test embedding of others. Thus according to the dependency criterion they should be processed *earlier*.

The specialty can be estimated automatically by checking whether components have explicit control ports.

- Type of the untestable components: Some types of components like registers are often used in datapath and modified to perform important testing tasks. These components have a larger chance of getting involved in testing others, that is, a larger chance of getting changed than other components. These components should be processed latter.

  We have implemented a type table to compare the *helpfulness* of different types.

- Size of the untestable components: The same kind of DFT modifications may cost more for larger components. For instance, a larger combinatorial component may need a larger LSSD register to drive. Therefore, the potential savings are larger to consider more alternative solutions for larger components in order to find the optimal solutions. According to the branch factor criterion, larger components, i.e., those with a larger number of solutions should be processed later.

  The component size could have many definitions, such as absolute or relative chip area, number of logic nodes, etc. Current implementation defines the size of a component as the sum of its I/O wires[9].

- Relative position of components: The components on heavily traveled datapath have larger chance of getting free-tested or partially tested, so it is preferable to work on them later. On the other hand, the untestable components that are isolated from others could be worked on any time since there is no sharing of modification for them.

[9]We only compare components of the same type since we use component type as the primary order as described soon later.

Heuristics concerning component position are harder to implement than others since they are based on an understanding of how a circuit works.

Testing experts also use the position information to estimate the testability of components. For instance, the control sequencer is usually difficult to test. This testability estimation is unnecessary for this system since untestable components are known to the system.

Currently the system has an ordering function based on the component type, specialty and size. The heuristics concerned with the relative positions of components are unimplemented to date due to the difficulties in quantifying them.

Untestable components are first ordered according to their type. For the components of the same type, those without explicit control ports will be processed first. When components are regarded the same by the above two criteria, they are further sorted according to their size. If even their sizes cannot distinguished them, the components will be processed in an arbitrary order.

The type order is given by the list below. A component will be processed before any component to the right of it in the list:

```
Combinatorial ROM Incrementer Multiplexor ALU Shifter RAM Register Bus Clock
```

This list includes all the component types encountered in the example circuits. This ordering has taken into account the following considerations:

- *Combinatorial*: Since the functionalities of combinatorial components vary widely, it is difficult to figure out how to use combinatorial components to test other components. So they have little chance to be free-tested or changed and can be processed early.

- *ROM*: ROMs can be augmented with new entries to help test other components. However, doing so is expensive since a special addressing sequence is needed in addition to the new entries; and addressing the special entries does not test the original ROM contents. So it is right after combinatorial.

- *Incrementer* versus *Multiplexor*: Compared with multiplexors, more reasoning power is needed to use incrementers properly since they *change* the signals passed through them. So incrementers have less chances to be modified or to be free- or partially-tested than multiplexors and should be worked on earlier.
- *ALU* versus *Multiplexor*: ALUs functionally subsume multiplexors therefore have larger chances to be used in helping test others.
- *Registers* versus *RAM*: RAMs can be viewed as register arrays, that is, RAMs subsume registers functionally. However, since it is difficult to modify one entry in a RAM as we do with registers, RAMs are less useful than registers in testing.
- *Shifter*: There is no particular reason for putting it where it is.
- *Clock*: Clocks are very challenging to test automatically due to their functions and locations. They are on the list to make the list complete for the example circuits.

Regularity in Test Segment Embedding Subtree

The part of the local solution tree corresponding to test segments has a patterned structure, which alternates three layers of single type nodes: a layer of and-nodes representing component usages, a layer of or-nodes representing alternative connections to pass signal-requirements, and a layer of or-nodes representing alternative component usages to process signal-requirements (see Figure 3.2). The other part of the local solution tree above test segment embeddings does not have similar regularity. The difference in the tree structure is the main reason for us to treat them differently. The two stages of test pattern embedding also have different applicable constraints.

Desired Number of Embeddings

In general, the more accurately the evaluation functions direct the search, the fewer embeddings the system needs to generate. For the examples shown below, the minimum number N_g of solutions for the whole circuit is set to 4. The minimum numbers for other low-level solutions are calculated from N_g:

- N_f: For a focus, $N_f = N_g$
- N_{tp}: for a test-pattern, $N_{tp} = N_g$
- N_{ts}: For a test-segment, $\sum_{tp_i \in the\ focus} \min_{ts_j \in tp_i} N_{ts_j} = 2N_g$

  Here N_{ts_j} is the minimum number of solutions for the test-segment ts_j of test-pattern tp_i; the term

  $$\min_{ts_j \in tp_i} N_{ts_j}$$

  is the estimated number of solutions for tp_i, which is not conservative considering the *same control scheme* constraint which only allows one embedding of a test segment to combine with a single embedding of another test segment; and the factor of 2 is to leave some room for over-estimates and later choices.

Deciding what Constraint to Relax

When there are not enough solutions for the circuit under the existing constraints, the system will try to relax some of the violated constraints unless none can be relaxed. A constraint cannot be relaxed if it is the last in the category or if the user specifies that it cannot be relaxed. To select a constraint to relax, the system first decides whether to relax the constraints on embedding test segments or to relax the constraints on combining test segment embeddings. Once the system has decided which group of constraint categories to relax, it will relax the most restrictive constraint as described in §3.2.3.

Relaxing Constraints

Relaxed constraints will be removed. The system will then check each of the suspended searches to see if all the constraints violated are removed. If so, the suspended search will be resumed.

3.3.4 DFT: the Idea Behind the Whole Implementation

There are four major ideas behind the whole system: the testability definition; the notions of DFT within the context of TG failure and TG while modifying the circuit; the notion of the Designer, TG and DFT trio; and the emphasis on the importance of TG knowledge to the DFT system.

Defining testability as successful test generation on circuits leads to a powerful and practical method of locating real testing problems. This in turn enables the system to focus its computational efforts on circuit portions with testing problems, to modify the circuit selectively, and consequently to introduce no redundant modifications.

DFT within the context of TG failure and *TG allowed to modify the circuit* are important extensions of the testability definition. The former justifies both the system's generation of only skeleton tests and its focus on modifying circuits. The latter points to a concrete method of pinpointing testing problems and of relating them to suitable circuit modifications.

The natural function set catches a large part of the DFT modifications that are related to improving the circuit's internal controllability and observability. Though this set does not catch all modifications, the system has already demonstrated a significant ability on several example circuits.

The trio scenario (§2.3) clears up some confusion in previous DFT systems. It points out from the DFT standpoint that there are three kinds of knowledge involved in the VLSI design process, and that a practical way to build a DFT system is to concentrate on DFT and to get help from the designer and the test generator. This scenario also points out the importance of test generation knowledge to the DFT system — in fact the whole DFT process is *test generation guided*. The test generator locates testing problems, assembles modifications, and determines the termination of the process. The final products of the system are tests with circuit modifications.

3.3.5 Evaluation Functions

The system has two types of evaluation functions, namely a constraint relaxation mechanism and a solution grading function. The former is used for the purpose of pruning the search space, the latter for the purpose of grading alternative solutions.

Relaxable Constraints

The constraint relaxation mechanism has three elements: constraint ordering, constraint maintenance, and constraint definitions. Section 3.2.3 has covered constraint maintenance and a major part of constraint ordering. This section justifies the constraint category order as shown in Table 3.1 and shows how the constraints are defined.

Constraint Ordering

The constraint order has a two level hierarchy: the order of constraint categories and the order within categories, with the former dominating. The category order is shown in Table 3.1. The following are the justifications for the order in the form of comparisons between *simultaneous* adjacent category violations.

- *propagation* versus *protect focus*: Note that the *protect focus* category is a special case of the *loop* category, i.e., a loop involving the focus. The relaxation of the constraints in the loop category before that in the *protect focus* category is based on the principle of "relaxing the holding constraint first." The justification is that if there are partial solutions stopped by protect focus constraints, the propagation constraints are less likely to be responsible, since at least on the loop the propagation can proceed forever.

- *protect focus* versus *loop*: The *protect focus* is a special case of *loop*. Relaxing the general category first is consistent with the principle of "relaxing the holding constraint first." When a constraint in the *protect focus* category is violated, the corresponding constraint in the loop category is also violated but the opposite is not necessarily true. Therefore, relaxing the general version constraint alone is likely to advance the partial solution.

The validity boundary of the *protect focus* category changes with the focus type to give state bearing foci more protection. All other constraint categories are independent of focus type.

- *loop* versus *same side intersection*: The *loop* category is concerned with only one data path, but the *same side intersection* category is concerned with all data paths on either input side or output side of the focus. Since several data paths must exist in order to apply the constraints in the latter category, the loop category associated with the earlier search stages is not likely to hold the development of the partial solution — this is the derived principle of "relaxing the later stage constraint first" which we will use repeatedly.

- *same side intersection* versus *both sides intersection*: The system builds data paths on only one side (input or output side) first; therefore, *both sides intersection* is associated with the later stages of solution construction.

- *both sides intersection* versus *sharing modifications 1*: Here *sharing modifications* means that two modifications made to one component needs less overhead than the sum of the overhead for individual modifications. This order of relaxing the *both sides intersection* category before the sharing modifications category is more arbitrary than others. The supporting heuristic is that component modifications usually appear at sources or sinks. A particular modification can be made to create sinks or sources but can not be used to create both. Modifications to the same component in sources and sinks, however, can share some hardware. Therefore, when a test is stopped by a second modification to a component, that is, when the sharing modifications category gets violated, it is likely that one of the two modifications involved is for a source and the other is for a sink. Since sources and sinks are at different sides of the focus, and sources and sinks are likely at the ends of data paths, the violation of sharing modifications is at the very end of the both sides stage. In other words, the sharing modifications constraint violation occurs later than the both sides intersection constraint violation and should be relaxed first.

Starting from the categories of sharing modifications, the constraints are concerned with the preference, not the validity, of solutions.

- *sharing modifications 1* versus *sharing modifications 2*: The *sharing modifications 2* constraint category is concerned with modification sharing between different foci; therefore, its violation is considered to occur at a stage later than its counterpart for a single focus. Also, the sharing modification 2 category is not activated for the first focus in a circuit.

- *sharing controls*: All the constraints above are concerned with embedding a single test pattern segment. This category is concerned with combining the embedding of test segments and is applied at a different time. Therefore, it is treated independently (see §3.2.3).

Constraint Definitions

Each constraint is identified by its category and its name. The functional part of a constraint is written in LISP code that extracts information from a partial solution and the new building block to be added and checks the condition. The following is an example:

```
(def-consistency-checker
   protect-focus          ; The category name
   intact                 ; The constraint name
   (building-block partial-result)   ; The inputs to be checked
   ;; The code:
   (let* ((all-component-names-but-focus
          ;; Extract all components not functioning as the focus
          ;; in the partial test segment embedding including the
          ;; building block
          (get-all-component-names-but-focus
             building-block partial-result))
      (focus-name (get-focus-name-from-ts-implementation
          ;; Extract the focus
                   partial-result)))
  ;; The violation condition: the "focus-name" is included in
  ;; "all-component-names-but-focus", i.e., the focus is used
  ;; to test itself
  (member focus-name all-component-names-but-focus :test #'equal)))
```

Component Test Grading

The evaluation function for phase 3 (see §3.2.2) is a numerical one. It is used to grade component tests when individual component tests are combined into circuit tests.

A component test is graded according to its beneficial side effects: it may free-test, normally-test, or partially-test other untestable components. The system evaluates these side effects separately, then weight-sums them as the final score.

A test *free-tests* some untestable components other than the focus when they are on data paths used by the test and when the signal sequences passing through them during the test are adequate for testing them. A test *normally-tests* other components that are not free-tested but can be satisfactorily tested with the help only of the modifications already introduced. A test *partially-tests* other components when they are *not* free-tested or normally-tested, but the test can help improve their controllability or observability, or more specifically, there are data paths between the sources or sinks of the test and these components.

The free-testing or normally-testing grade of a test is calculated as following:

$$\begin{aligned} FreeTestingGrade &= NumberofUntestableComponentsItFreeTests \\ NormalTestingGrade &= NumberofUntestableComponentsItNormalTests \end{aligned}$$

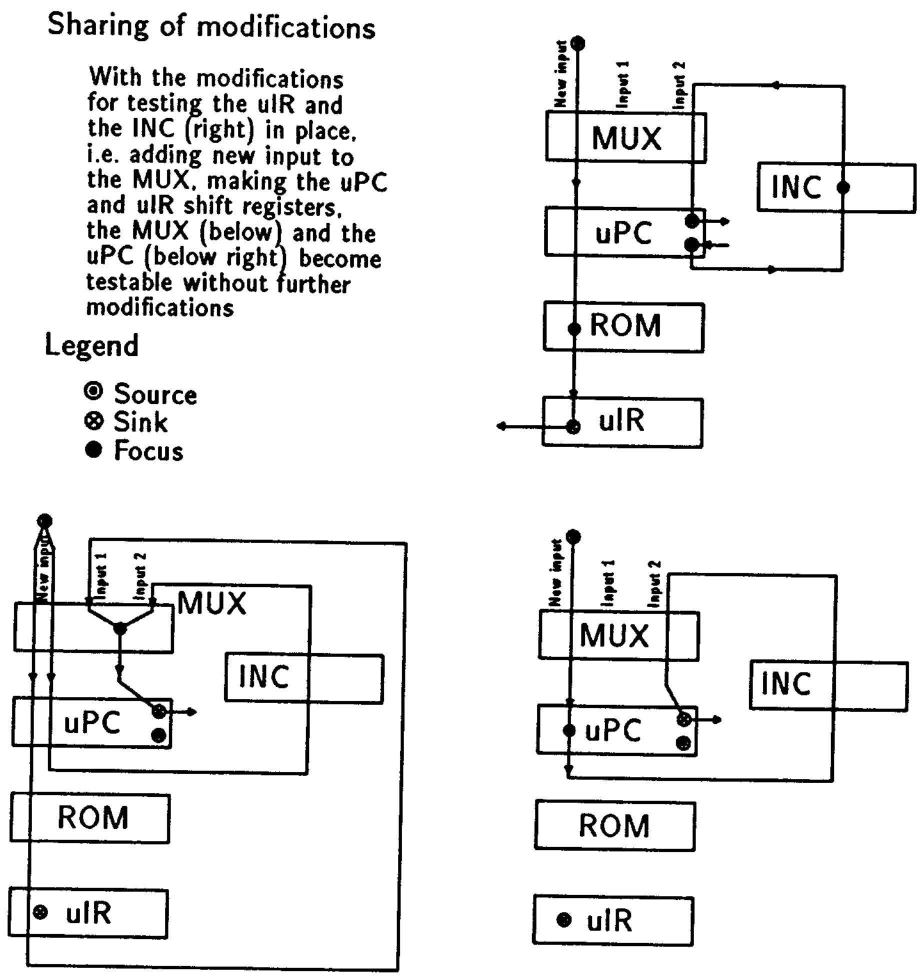
In these two formulas the *UntestableComponents* include only those that are unprocessed.

The partial-testing grade of a test is the sum of its credit on each data port of the partially-tested untestable components. Control ports do not contribute to the grade because we take the controllability or observability of control ports for granted. When there is a data path from a source (or sink) of the test to an input (or output) port of an untestable component,

$$\begin{aligned}
PortCredit &= CoverageRate \times UniquenessRate \\
CoverageRate &= \frac{1}{NumberofOriginalDataPortsofTheComponent} \\
UniquenessRate &= e^{-NumberofOtherPartialTestsHelpingThisPort}
\end{aligned}$$

The *coverage rate* normalizes the maximum credit from a single component. The uniqueness rate emphasizes the first achievement of controllability (or observability) as well as considers the benefit of alternatives. The first achievement gets a full credit for solving a controllability or observability problem; the second one gets some credit for providing an alternative in case that the first achievement is not consistent with a global solution. But as the number of alternatives increases, the value of an additional option diminishes. Therefore, later achievements get exponentially smaller credits. In the uniqueness rate formula, *Other Partial Tests* mean the selected tests for other untestable components that also partially-test the same untestable component.

Figure 3.11 shows an example of those gradings. Suppose that, in the MAC-2 circuit, the first untestable focus is the ROM, and the test for it is to use the MUX with an additional input to route the stimulus in. Also suppose that the stimulus would be passed through the uPC to the ROM, and that the uIR would be used to shift the response out. In this context, the test for the INC — using uPC to shift the stimulus in and using the MUX to pass the response back to the uPC to shift it out — will get the following gradings. Its *free-test* grade is 0 because it does not *free-test* any of the remaining untestable components, that is, the uPC, the MUX or the uIR; its *normally-test* grade is 2 because the MUX and the uPC can be tested


Sharing of modifications

With the modifications for testing the uIR and the INC (right) in place, i.e. adding new input to the MUX, making the uPC and uIR shift registers, the MUX (below) and the uPC (below right) become testable without further modifications

Figure 3.11: An example of gradings

with the help of the sink uPC introduced by this test[10]; and the *partially-test* grade is 0.11 because

- The uIR has 13 data ports, so the *coverage-rate* for each data port is 1/13,
- The input port of the uIR can be controlled by the source uPC of the INC test via the ROM. Since it can also be controlled by the source connected to the created third input of the MUX for testing the ROM, its uniqueness-rate is $e^{-1} = 0.38$ and the credit is $0.38 \times 1/13 = 0.03$.
- The uIR output port connected to the MUX can be observed by the sink uPC through the MUX. Since this is the only way to observe this port, the uniqueness-rate is $e^{-0} = 1$ and the credit is $1 \times 1/13 = 0.08$.
- The remaining 11 uIR output ports, which are suppressed from the picture for simplicity, can not be observed by any of the two sinks. Therefore, none of them get any credit and the total *partially-test* grade is 0.11.

Currently all the weights for the three side-effects are set to 1, so the total score is 2.11 for the INC test.

3.3.6 Solution: Test Scheme

The system produces several solutions, each solution is a *test scheme* consisting of a test for each of the originally untestable components. A component test has an implementation for each test segment in a test pattern. A test segment implementation records how the components involved should work and what modifications are needed. The next section will illustrate the recorded information with two examples.

[10]The sink uIR for testing the ROM can not be of help because the ROM can not pass their responses. A ROM can pass a stimulus easily by storing it in additional locations and then reading it out. But in order to pass a response, a ROM has to store special values at locations dictated by the response, which may interfere with its normal operation.

3.4 Running Examples

3.4.1 Mark-2

A Detailed Example of a Test for the ROM

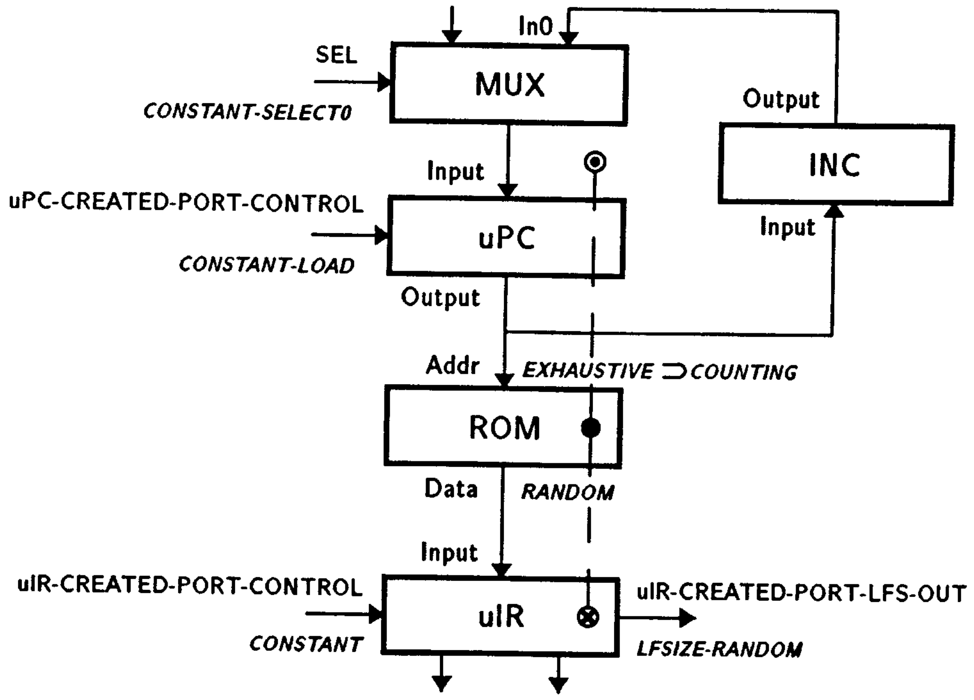


Figure 3.12: A test-pattern embedding of the MAC-2 ROM

Figure 3.13 below shows the program's print-out of a test for the ROM of the MAC-2 illustrated by Figure 3.12. The first two lines describe the test pattern — applying an `EXHAUSTIVE` signal sequence on the `ADDR` port of the `ROM`, and observing a `RANDOM` signal sequence on the `DATA` port of the `ROM`. The `RANDOM` signal sequence depends on the contents of the ROM and the signal `EXHAUSTIVE`.

The third line outlines the data path used in the test pattern embedding. The three components `INC`, `uPC` and `MUX` are used together as a compound counter to provide an instance of the required exhaustive sequence. The uIR is used to observe the response of the ROM. In the print-out, the "*" denotes either *source* or *sink*; the

```
ROM's Test Pattern #1
 Apply EXHAUSTIVE on port ADDR and Observe RANDOM on port DATA of Focus ROM

  COUNTER=INC+uPC+MUX*->[ROM]->*!uIR

 This is a realization for SEGMENT-1 of the ROM ROM-READ-TEST-1
  Use Components INC and uPC as Compound COUNTER to GENERATE
  Use ROM at Mode READ to FOCUS
  Use uIR at Mode LFS-COMPRESS to OBSERVE

  And we should
   In Compound COUNTER,  as the Critical Components
    Use INC at Mode INCREMENTER
    Provide CONSTANT-LOAD on uPC-CREATED-PORT-CONTROL (optional) of uPC (LOAD)
   And as the components used as Wire
    Connect uPC's INPUT to INC's OUTPUT through MUX
     Provide CONSTANT-SELECTO on SEL of MUX (doing SELECTO)
    Connect uPC's OUTPUT to INC's INPUT directly
   Extend uIR with LFS-COMPRESS and port uIR-CREATED-PORT-CONTROL,
                                        uIR-CREATED-PORT-LFS-OUT
    Provide CONSTANT on uIR-CREATED-PORT-CONTROL
    Observe LFSIZE-RANDOM on uIR-CREATED-PORT-LFS-OUT of uIR (LFS-COMPRESS)
```

Figure 3.13: A Test Scheme for the MAC-2 ROM

"`!`" denotes some modifications are needed; and the "`[]`" denotes the focus in a test.

The rest of the output gives the details about the test pattern embedding. It describes

- What the components are used for in the test. For instance, the compound component is used as the source and the MUX is used as a wire in the compound component to connect the `uPC's` input to the INC's output;
- What working modes the components should be in. For instance, the `ROM` is working in READ mode to be tested;
- What kinds of signal-requirements are needed on the ports of the components (except those on data paths) in order to coordinate these components. For instance, the uIR needs a `CONSTANT` control signal to set it up in the Linear-Feedback-Shift mode; and

- What kind of modifications to the components are needed. For instance, the uIR needs an additional working mode and two new I/O ports.

Other Examples

Figure 3.14 shows more test segment embeddings for other untestable components in the MAC-2 micro sequencer part. The *ROM-1* (the first one on the third row) has been described in Figure 3.12.

In Figure 3.14, the figures in the same row have the same focus. Each of the small figures depicts at least one embedding, since different test segments may have the same data paths and different component usages may have the same representation. For instance, LFSRs are represented in the same way as shift registers when both are used as sources.

These test segment embeddings are part of the intermediate results when the system is asked to generate at least 4 global solutions for the MAC-2. All the five components are originally untestable. According to the ordering function, they are processed in the order of ROM, INC, MUX, uPC, and uIR.

After the modifications for testing ROM and INC are made to the circuit, the MUX and the uPC become normal-tested, i.e., tested without new modifications, and are removed from the untestable component queue. However, the system can not find complete solutions for uIR. The question marks on one of the uIR outputs indicate the failed testing goals: the system failed to observe these outputs (which go to the control ports of other circuit components shown in Figure 1.1). The system does not have the knowledge for propagating signal-requirements into a control port. More knowledge is needed in order to solve this kind of testing problem. For instance, a new component may be inserted at the output of the uIR to observe these signals directly, opening up whole new territory of modifications. However, teaching the system to propagate signal-requirements through control ports is considered inappropriate for a DFT system based on the arguments in Chapter 2. For the time being, the system

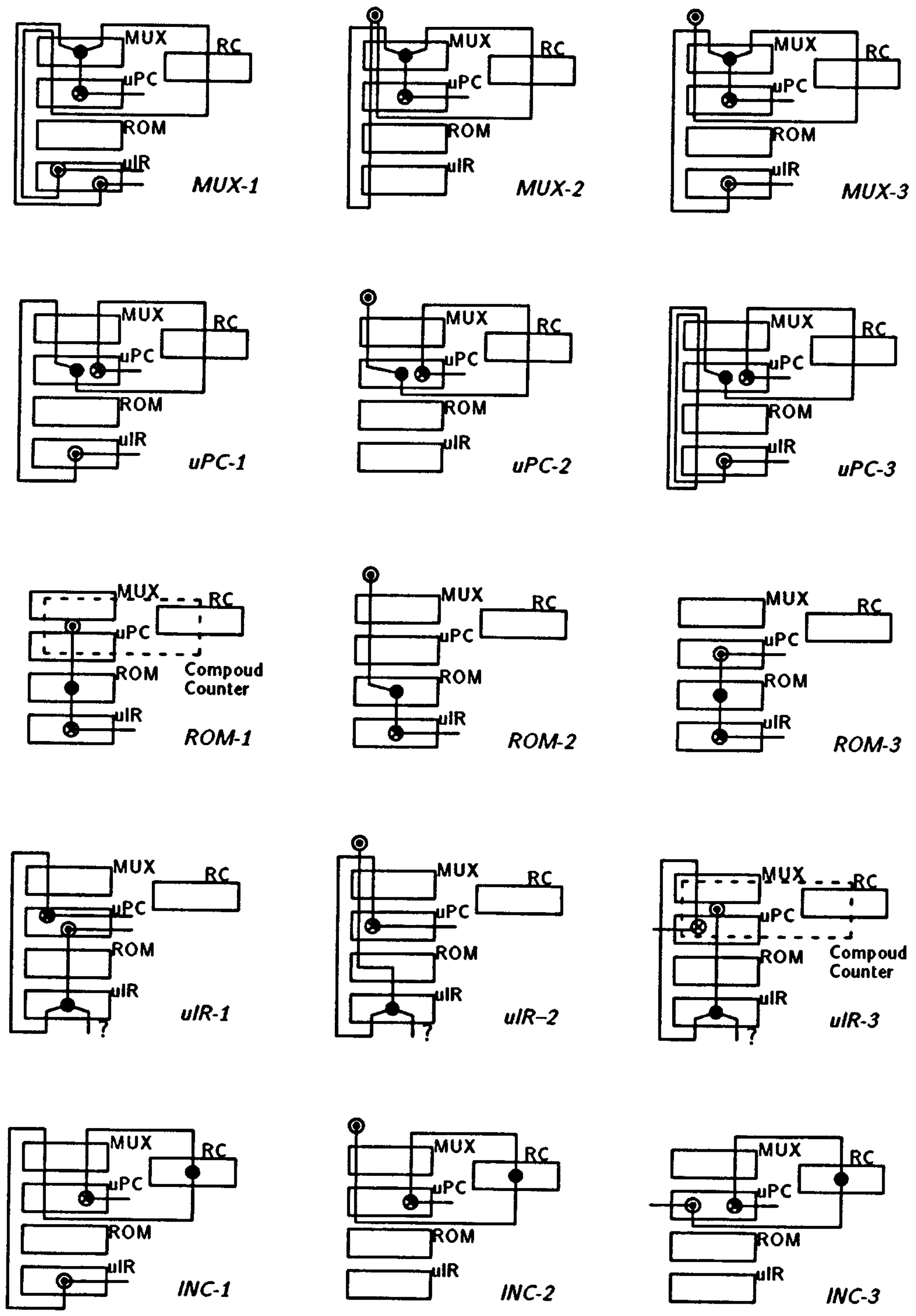


Figure 3.14: Test-segment embeddings of the MAC-2

returns partial solutions when it can not find complete ones.

Since the system choose only 2 of the 4 solutions for each of the three DFT-tested or partially-DFT-tested foci, there is a total of $2^3 = 8$ global solutions. Table 3.3 shows the data paths used by those global solutions:

Solution-1	*MUX-1*	*uPC-1*	*ROM-1*	*uIR-1*	*INC-1*
Solution-2	*MUX-2*	*uPC-2*	*ROM-2*	*uIR-2*	*INC-2*
Solution-3	*MUX-2*	*uPC-2*	*ROM-2*	*uIR-1*	*INC-3*
Solution-4	*MUX-2*	*uPC-2*	*ROM-2*	*uIR-2*	*INC-3*
Solution-5	*MUX-2*	*uPC-2*	*ROM-2*	*uIR-1*	*INC-2*
Solution-6	*MUX-2*	*uPC-2*	*ROM-1*	*uIR-1*	*INC-2*
Solution-7	*MUX-2*	*uPC-2*	*ROM-1*	*uIR-2*	*INC-2*
Solution-8	*MUX-1*	*uPC-1*	*ROM-1*	*uIR-2*	*INC-1*

Table 3.3: The global solutions for the MAC

By examining the test segment embedding groups for each component, we can see the trace of constraint relaxation. Among the three embeddings for the ROM, none has the data path providing controllability intersected with the data path providing observability. The reason is that enough embeddings are found before the constraints are relaxed to such an extent that data path intersections are allowed. The *uPC-3* is another example. When restricted to using only the modifications made for the ROM and the MUX as the system checks whether the uPC is normal-tested, the system is forced to allow looping datapath such as the one in the *uPC-3* in order to meet the $N_f = 4$ requirements[11]. A loop in the *uPC-3* has the effect of changing the signal-requirement passed through it, although the change is not necessary in view of *uPC-1*.

The global Solution-1 modifies two components, namely the two registers uPC and uIR, as sources and sinks. In Solution-1 the uPC is modified to a shift register and the uIR to an LFSR. If all are modified to shift registers, these modifications

[11]In this implementation, when checking whether a component is normally-tested, the system works the same way as it DFT-tests the component except making no new modifications to the circuit. In other words, the system works the same way as a ordinary test generator.

look like what would have been required by the LSSD method. However, despite the apparent similarity with one of the solutions of this system, the LSSD method differs in two fundamental aspects. First, the LSSD method would introduce modifications to *every* register in a circuit, regardless of whether there is a testing problem or not; our system, by contrast, introduces modifications *only* in the control sequential part. Second, the LSSD method uses only shift registers, whereas this system uses a larger variety of modifications, including built-in test techniques and ad hoc DFT techniques.

3.4.2 Another Example

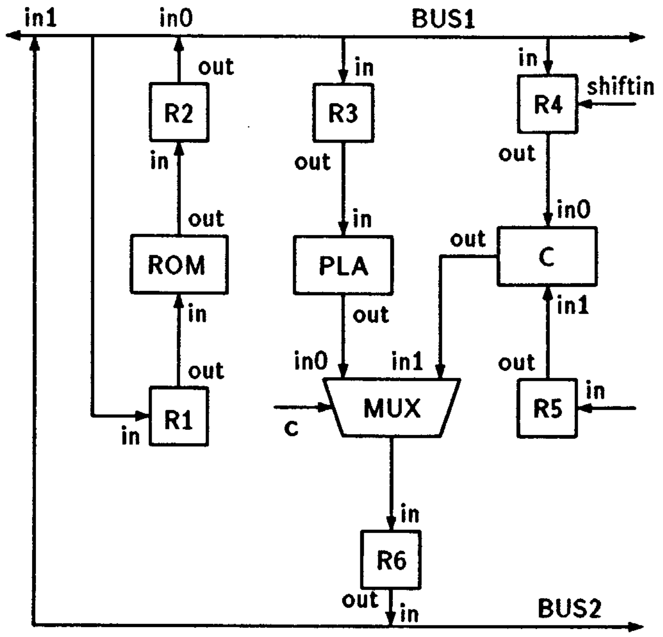


Figure 3.15: Another example circuit

The system has also been tested on other circuits. Figure 3.15 shows one of them, from an example used in [1]. In the circuit, all the R's are buffering registers, C is a combinatorial component, the PLA is a Programmable Logic Array (treated as a combinatorial component in our test runs), and the ROM and the MUX are similar

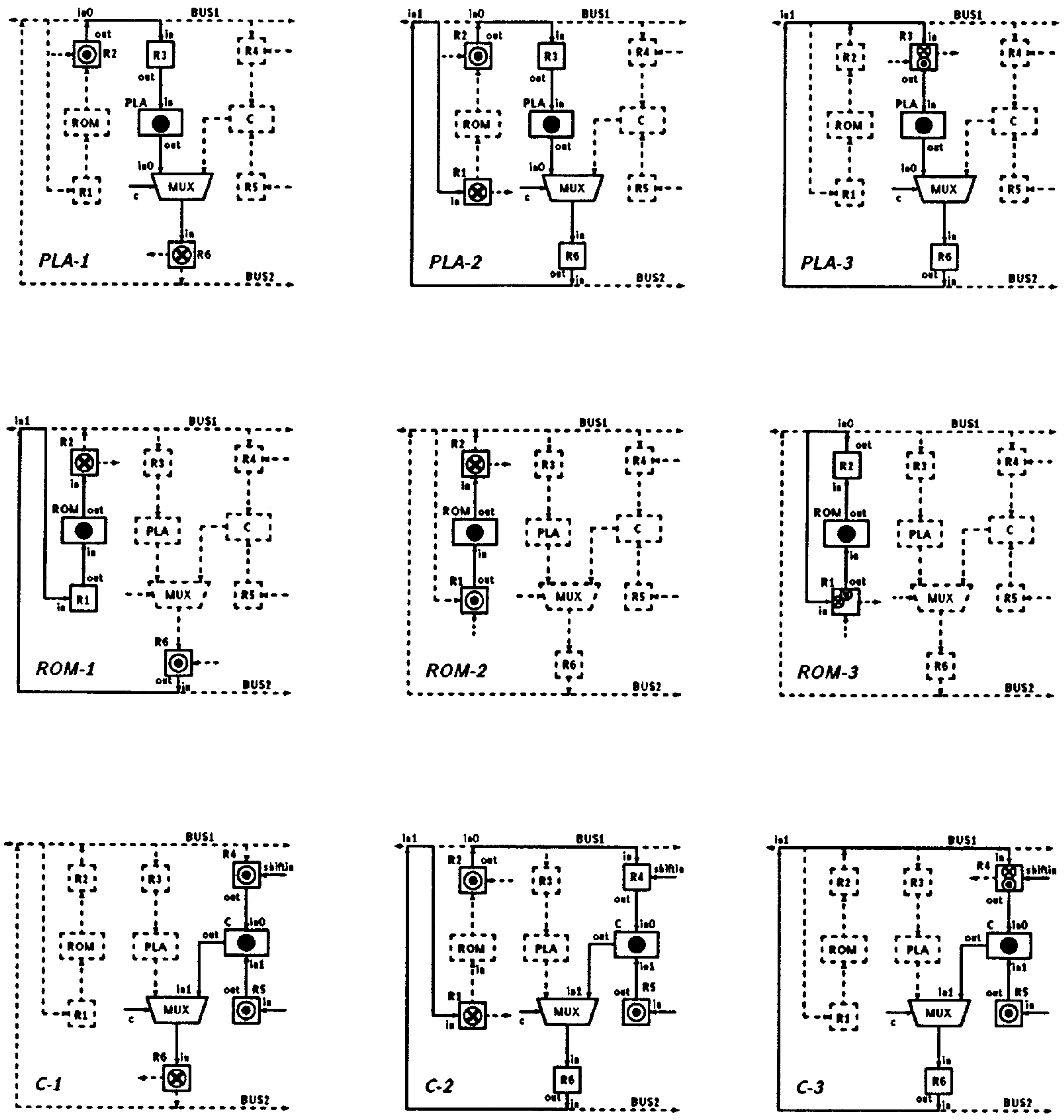


Figure 3.16: Test-Segment Embeddings

to those in the MAC-2. Figure 3.16 shows the solutions. The notations are similar to those in Figure 3.14 except that the data paths are directly superimposed on circuit connections since the solutions shown here do not have loops.

In the test runs, we assume that there is no direct access to any circuit node except the SHIFTIN port of the R4, and the untestable components are the PLA, the C and the ROM. Figure 3.16 shows some solutions under these conditions.

In all the solutions shown here the sources and sinks are registers, mostly modified either as LFSRs or scan shift registers during testing. The MUX is also being used by some solutions (not shown here) as sources with additional inputs. The three solutions at the left column, i.e., PLA-1, ROM-1 and C-1, are combined in one "global solution," which is similar to the example solutions in [1].

3.4.3 The Complexity and the Representation Abstraction Level

Once the example shown in Figure 3.15 was run under conditions similar to that in the MAC-2 example (where $N_g = 2$, all ten components are untestable, the system's most complete natural function sets and its most complete knowledge on test patterns are used), it took more than a day to finish[12]. The solutions using less knowledge are similar to the solutions using more knowledge but need much less computation time. When run a second time, the example shown in Figure 3.16 required one hour on a Symbolics 3600 under conditions indicating that (1) $N_g = 5$, (2) there are three untestable components, (3) only *one test pattern* is used for each component type, and (4) only *one source or sink component usage* is used for each component type and signal requirement.

Using *one test pattern* per component type and *one source or sink component*

[12]The system needs approximately one and a half hours to run the MAC-2 example. The absolute running time is less informative than the relative running time under different conditions because the current implementation is more concerned with what knowledge is needed to get the results than with implementation efficiency.

usage per component type and signal requirement is similar to ignoring the differences among different signal sequences. Therefore, the experiment with reduced knowledge can be viewed as an experiment with a higher level of signal sequence abstraction. The results are consistent with the speculation in §3.3 about the system power if a higher abstraction level is used. The computation time savings come from the fact that with the reduced knowledge the search space is much smaller because the number of alternatives (i.e., alternative test patterns) and the number of simultaneous tasks (i.e., the number of test segments per test pattern) are both reduced to a minimum.

Chapter 4

Discussion

4.1 Related Work

The work on test generation in [19] has had a strong impact on this research. In [19], Shirley presents a test generator that succeeds quickly on parts of a circuit that human experts consider easy to test, and fails quickly on parts human experts find difficult to test. He then suggests that DFT modifications should be considered to achieve low-cost testing.

Horstmann's DFT system [10] takes a design-rule approach, using rules from LSSD design standards. The design-rule approach seems difficult to generalize in order to accommodate other DFT techniques.

Abadir's DFT system [1] uses a "testable structure embedding" approach, employing a general circuit-structure model, similar to our compound-component representation, to represent structured DFT methods, such as the LSSD design standard, Built-in Logic Block Observation, etc. A "testable structure" in [1] bundles focus, its test pattern, source and sink together. A testable structure either matches for a focus or fails *completely*; i.e., any partial match is discarded. The match is done through a process similar to our signal propagation except for the signals passed on the data path. In [1] the signals are implicitly defined as arbitrary signal sequences independent of the test pattern to be implemented — as it only considers data paths that pass signals *unchanged*. Although Abadir's processes are similar to ours, his system did not explore the concepts of signal propagation, *propagation failure*, or evaluation functions than ours and it interfaces to a test-schedule optimizer that essentially serves as a data-path-contention resolver and as an evaluation function for the DFT system. To perform the sophisticated test-schedule optimization, the system keeps time tags during signal propagation (or, in his original terminology, forming kernel subgraph using I-path concept).

Our research has been inspired in part by Abadir's work. However, all previous DFT systems fail to answer the critical question of what a testability problem is, an issue that has been central to this research.

4.2 Merits, Limitations and Future Work

Our approach to DFT has four advantages distinct from those of previous DFT systems:

1. The system focuses its efforts on untestable components in circuits and thus has certain computational advantages.

2. It only suggests modifications necessary to make untestable components testable, thus eliminating the chip area and the performance penalties for originally testable components. Advantages 1 and 2 may be transportable to some of the previous DFT systems.

3. It further focuses DFT modifications for untestable components on signal-requirement propagation failures, thus eliminating DFT modifications irrelevant to any signal-requirement propagation failure.

4. It combines DFT techniques according to the need for fixing signal-requirement propagation failures, and thus it has more alternatives in order to make untestable

components testable, and further reduces the resulting DFT penalties. For example, Figure 3.12 shows how a counter assembled from on-chip components can be used with a shift register to make the ROM testable. This is a mixture of built-in test and scan-path techniques. Systems using more structured techniques would not produce such a hybrid solution and would not obtain the resulting lower DFT modification overhead.

The main drawback in this system currently is its primitive domain knowledge. Except for the constraint-relaxation mechanism, the system does not have a very sophisticated evaluation function. The system completely breaks the bundling between foci, sources, and sinks in structured DFT techniques, and thus it loses the knowledge about the preferences for certain combinations of foci, sources and sinks. We speculate that these preferences can be encoded as evaluation functions. Also related to solution evaluation is the lack of timing information that prevents the system from analyzing resource contention more competently. However, armed with the knowledge of test generation, the system has already demonstrated its ability and has produced interesting results. The question for future research is how to organize and use the vast amount of knowledge needed for the final touch to the solutions without getting trapped in the complexity pitfall. Using a hierarchical representation might be a promising direction which could make the detailed knowledge available on demand.

In contrast to the too-coarse representation of timing, the representation of test patterns is felt to be too detailed. Test patterns should be used in a hierarchical fashion in future research in order to suppress details about the actual signal-requirements when they are not needed.

The current implementation does not recognize compound foci. This fact could cause a problem if a circuit is presented to the system at an overly low a level, that is, components are presented as combinations of their subcomponents. In the current implementation, circuits are required to be presented at an appropriate structure level.

However, more experiments are needed to assess how often circuits are presented at an unsuitable structure level and how severe the problem is.

The system can modify circuits by adding functions to components, but not by adding connections between internal circuit nodes nor by inserting components in connections. Adding connections or components is consistent with our paradigm, i.e., creating new ways of handling signal-requirements, but doing so will increase the search space. Knowledge about the spatial relationship between components, e.g., chip layout, may help to reduce the search space. This will open another research regimen, that is, employing spatial and logical circuit models and knowledge simultaneously.

Issues also need to be raised about the trio view of the VLSI design process:

- The DFT modification suggestor needs the help of *multiple* test generators to locate the untestable components[1] because state-of-the-art test generation is collectively captured by many test generators. These test generators are required to be efficient in detecting untestable components. However, except for [19], no other test generators have been evaluated or modified in this regard.

- The purpose of the interface between test generators and our DFT system is to pass untestable components. This interface seems to have the potential to communicate with human testing experts and a wide range of test generators about circuit testability problems, but it has not been studied with respect to test generators other than [19]. Other kinds of interface might work equally well. Moreover, there is no particular reason for a uniform interface. Different test generators may interface with a DFT system differently.

- The signal-requirement propagation failure seems to be a suitable index for a large variety of DFT modifications. However, this index should be further evaluated since our collection of DFT modifications is not exhaustive. Besides

[1]Or to be general, the TG failures at the coarser level.

the implementation related aspects, there are also some profound strategies which do not seem to have a straightforward way to index with a particular TG failure: for instance, the strategy of task reduction as seen in LSSD or cutting feedback loops which reduces sequential circuits to combinatorial ones.

- The implementation has used a D-algorithm-like reference test generator to capture knowledge of test generation and to refine the TG failures. It remains to be determined whether other types of test generators are also suitable for these purposes and whether multiple test generators can be used.

4.3 Conclusion

Knowledge about test generation is critical to constructing a competent DFT system, yet it has not been used previously. This research proposes that test generation knowledge can be introduced into a DFT system by following the principle of repairing test generation failures.

According to this principle, the DFT system focuses its efforts on areas where testing problems actually arise and introduces modifications only in those areas. This means lower area and performance penalties are required to achieve testability than with other more structured, less focused techniques.

According to this principle, different DFT techniques can be integrated together in order to solve testing problems. For example, in Figure 3.12 we have seen how a counter assembled from on-chip components can be used with a shift register to make the ROM testable. This is a mixture of built-in test and scan-path techniques similar to the results seen in human practice [11] with real world circuits. Systems using more structured techniques would not produce such a hybrid solution and would not obtain the resulting lower DFT modification overhead.

The implemented system currently has only primitive domain knowledge and needs additional work. However, armed with the knowledge of test generation, it has already demonstrated its ability and produced interesting results on a simple microprocessor.

Bibliography

[1] Magdy S. Abadir and Melvin A. Breuer. A Knowledge-Based System for Designing Testable VLSI Chips. *IEEE Design & Test of Computers*, 56–68, August 1985.

[2] R. G. Bennetts. *Design of Testable Logic Circuits*, chapter Foreword, pages v–v. Addison-Wesley Publishing Company, 1984.

[3] Melvin A. Breuer. *Knowledge Based System for the Design of Testable VLSI Circuit Chips.* Technical Report Semi-Annual Technical Report, ARPA Order No.: 5088, University of Southern California, November 1986.

[4] Randall Davis. Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence*, 24(3):347–410, December 1984.

[5] Randall Davis. *Expert Systems: Where are we? and where do we go from here?* Technical Report A.I. Memo N. 665, MIT, Artificial Intelligence Laboratory, June 1982.

[6] Randall Davis. Reasoning from First Principles in Electronic Troubleshooting. *Int J. Man-Machine Studies*, (19):403–423, 1983.

[7] Johan de Kleer, Jon Doyle, Charles Rich, Guy L. Steele, Jr., and Gerald J. Sussman. AMORD — A Deductive Procedure System. August 1977. MIT AI Lab Working Paper 151.

[8] L. H . Goldstein et al. SCOAP: Sandia Controllability/Observability Analysis Program. In *Proceedings of 17th design Automation Conference*, pages 190–196, 1980.

[9] Walter Hamscher. *Using Structure and Functional Information in Diagnostic Design.* Master's thesis, Massachusetts Institute of Technology, May 1983. also available as MIT AI Lab TR No. 707.

[10] Paul W. Horstmann. Design for Testability Using Logic Programming. In *Proceedings of 1983 International Test Conference*, pages 706–713, 1983.

[11] John R. Kuban and John E. Salick. Testing Approached in the MC68020. *VLSI Design*, 22–30, November 1984.

[12] Kwok-Woon Lai. *Functional Testing for Digital Systems.* Technical Report CMU-CS-148, Carnegie-Mellow University, 1981.

[13] Edward J. McCluskey. Built-In Self-Test Structures. *IEEE Design and Test of computers*, 2(2):29–36, April 1985.

[14] Edward J. McCluskey. Built-In Self-Test Techniques. *IEEE Design and Test of computers*, 2(2):21–28, April 1985.

[15] Eugen I. Muehldorf and Anil D. Savkar. LSI Logic Testing — An Overview. *IEEE Transaction on Computer*, Vol. C-30(1):1–17, January 1981.

[16] Gordon D. Robinson. What is Testability? Feb 1985. (Author works with GenRad).

[17] 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.

[18] Mark H. Shirley. *Digital Test Generation from Hierarchical Models and Failure Symptoms.* Master's thesis, Massachusetts Institute of Technology, May 1983.

[19] Mark Harper Shirley. Generating Test by Exploiting Designed Behavior. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, pages 884–890, AAAI, August 1986.

[20] Mark Everett Sorrells. Design Rules to Enhance Testability in Digital Systems. B.S. Thesis, EECS of MIT, December 1982.

[21] P. Varma, A. P. Ambler, and K. Baker. An Analysis of the Economics of Self-Test. In *Proceedings of 1984 International Test Conference*, IEEE, 1984.

[22] M. J. Y. Williams et al. Enhancing Testability of Large-Scale Integrated Circuits via Test Points and Additional Logic. *IEEE trans on Computers*, C-22(1):46–60, January 1973.

[23] Xi-an Zhu and Melvin A. Breuer. A Knowledge Based System for Selecting Test Methodologies for VLSI Circuits. In *Knowledge Based System for the Design of Testable VLSI Circuit Chips*, chapter Appendix A, University of Southern California, 1986. Semi-Annual Technical Report, ARPA Order No.: 5088.