# Design for Testability

**Peng Wu**
MIT AI Lab
545 Technology Sq., Rm833
Cambridge, MA 02139

## Abstract

This paper presents an implemented system for modifying digital circuit designs to enhance testability. The key contributions of the work are: (1) setting design for testability in the context of test generation, (2) using failures during test generation to focus on testability problems, (3) indexing from these failures to a set of suggested circuit modifications. This approach does not add testability features to the portions of the circuit that a test generator can already handle, therefore, it promises to reduce the area and performance overhead necessary to achieve testability. While the system currently has only a small body of domain knowledge, it has demonstrated its ability to integrate different DFT techniques and to introduce only sharply focused modifications on a textbook microprocessor, an ability that is missing in previous DFT systems.

## 1 Introduction

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.

As VLSI technology advances, 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.

There are two responses to the testability problem. The first is to develop more powerful test generation (TG) methods in order to generate inexpensive tests efficiently. The second is design for testability (DFT), that is, to design circuits that are easy to test.

This paper presents an implemented system for modifying digital circuit designs to ensure testability. This system can be viewed as a DFT adviser to a circuit designer so that he can concentrate on more important design issues. The key contributions of this work are: (1) setting design for testability in the context of test generation, (2) using failures during test generation to focus on testability problems, (3) indexing from these failures to a set of suggested circuit modifications.

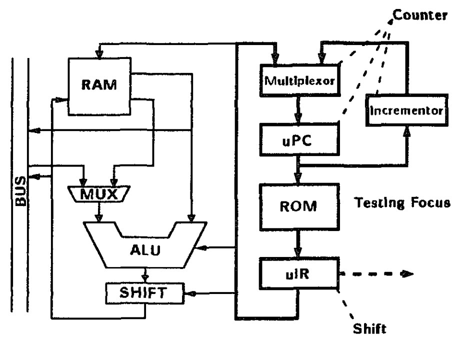

Figure 1: MAC-2: A textbook microprocessor

Figure 1 shows a textbook microprocessor. The left half is the datapath, the right half is the micro sequencer. The components in the sequencer have testability problems because they are internal to the circuit and are not easily accessible from outside.

To solve the testability problem for one of those components, the read-only memory (ROM), our system starts by consulting its library to see how a ROM can be tested. According to the library, a ROM can be tested by applying an exhaustive counting sequence on its address input, then verifying that its outputs are correct. When trying to apply a counting sequence to the ROM address, the system fails because it doesn't have direct access to the ROM input (hence it cannot directly input a counting sequence), and because even in normal use (i.e., getting addresses from the uPC), only a fraction of all the addresses might be applied. Our system then suggests that a register and an incrementor can be used as a counter (and hence provide a counting sequence) when connected in a loop, and indicates that the uPC, the Incrementor and the Multiplexor can do this if the Multiplexor always connects the uPC and the Incrementor during testing.

The output side of the ROM is more difficult — according to the system's TG algorithm, there is no way in the current design to observe the output, so the system encounters another test generation failure. A heuristic associated with this particular kind of failure indicates that the output can be observed by adding a shift function to

*This paper describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the author's research is provided by the Digital Equipment Corporation, Wang Corporation, and the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

a register. In this case, it suggests adding a shift function to the microinstruction register uIR, connecting the additional shift-out port to an output of the circuit, then using the uIR to shift the ROM contents out so that they can be verified.

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

1. It runs an external test generator [Shirley, 1986] on the circuit to identify the untestable components.
2. It further examines the testing problem by attempting to generate tests for the untestable components and analyze the reasons of failure. The test generator for this purpose has a simple algorithm and our system has access to its internal.
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.

Previous approaches to DFT have used *heuristic* testability definitions that assume a limited test generation capability, i.e., that of a classical, combinational test generator. This may result in false testability problems. For instance, the LSSD (level sensitive scan design) design rule approach [Horstmann, 1983] 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, and as a result would find "testability problems" that are not problems to some existing test generator. For example, according to the LSSD design rule approach, every register in the datapath part of the MAC-2 must be changed into a shift register, despite the fact that existing test generators (e.g., [Shirley, 1986]) can test these parts as they are.

## 2 Test Generation

The purpose of testing a circuit is to verify its behavior. This is done by exercising the circuit, i.e., by applying inputs to it and comparing its outputs against the expected values. Each set of inputs and expected outputs is called a *test pattern*. Circuits as complex as microprocessors are tested using divide-and-conquer: first, partition the circuit into components, then test each component. This partitioning for testing usually follows the schematic or a partitioning suggested by the designer.

Testing a specific component (the *focus* of the test) involves three steps:

1. Work out test patterns for the focus.
2. Work out how to apply inputs to the focus via the surrounding components.
3. Work out how to observe the focus outputs, again via the surrounding components.

Working out how to test a component is a recursive subproblem that bottoms out at primitive logic gates, e.g.. AND, OR, and NOT, for which there are simple, well-known tests. To test components internal to a VLSI chip, the test patterns must be executed through the surrounding components. This typically involves routing signals through the circuit, which we refer to as *routing tasks*.

## 3 Test Generation Guided DFT

For the purposes of this paper, we define the testability of a circuit relative to a test generation algorithm. We say that a circuit is *testable* by an algorithm if the algorithm can generate a test for the circuit. If a circuit is not testable by that algorithm, then it is the job of our DFT system to suggest design changes which will enable the algorithm to succeed. Although DFT's goal has been understood in this way previously, only recently have developments in TG technology made it possible to do so in an automatic DFT system.

In our system, when a component has a testability problem, the problem must be in routing, since its test pattern is assumed known (we have built up a library of test patterns for components by consulting testing experts). For example, in Figure 1 the ROM cannot get a counting sequence since none of the five components in the sequencer are directly connected to primary inputs in the original circuit. Thus routing a counting sequence to the ROM fails and this is one of the reasons for the ROM being untestable.

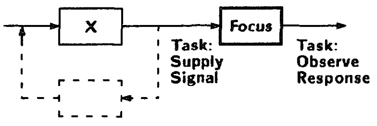

Figure 2: DFT in the context of TG failure

Figure 2 shows how our DFT system will fix routing problems for a specific test Focus. In order to figure out what kind of modifications are helpful, the system checks each of a focus' neighboring components to determine whether it can help to solve a routing task. A neighbor can help in any of several ways:

- The component may be able to complete the task. For example, assume that the Focus is a ROM that needs a *counting* sequence to exercise it. If the neighboring component driving it, X, happens to be a shift-register connected to a primary input, the task of providing a counting sequence can be accomplished by using X in its shifting mode (and a test equipment will drive the primary input with the counting sequence).
- The component may be able to pass the task along for other components to solve. If X has a mode in which its output can be made equal to its input, it can be made "transparent," and the task of providing a counting sequence can be passed to components further "upstream."
- The component may be able to solve part of the task and cooperate with others. For example, suppose X is an ordinary register that happens to be in a loop with an incrementor (the dashed lines). By using them *together* we can generate the counting sequence. Thus, X accomplishes part of the original task.

In each of the three ways of accomplishing a routing task, design modifications may be involved. For instance, if X is used to complete the task of providing a counting sequence for testing the ROM, i.e., used in the first way as shown above, but it is a register without shift function originally, it needs a modification.

Using a component to pass a signal differs from using it to cooperate with other components to route a signal. When passing a signal the component functions as no more than a wire; when cooperating with other components it plays a critical role, as for instance, when a register is used to hold the state when it is cooperating with an incrementor as a counter. These two usages of components produce different subgoals that are handled differently — cooperation also involves circuit structure matching tasks (described in the next section) in addition to routing tasks.

## 4 Domain Knowledge

The mechanism presented in the previous section offers a framework for our DFT system. The domain knowledge needed to complete the system consists of (i) test generation knowledge, such as test patterns for different components and the way of using components to accomplish routing tasks, and (ii) TG failure repair strategies, such as the component modifications that help routing tasks.

The test patterns specify the input stimulus to exercise components and the predicted responses of the components. Each type of component has its own test pattern specifications; we get these from experts.

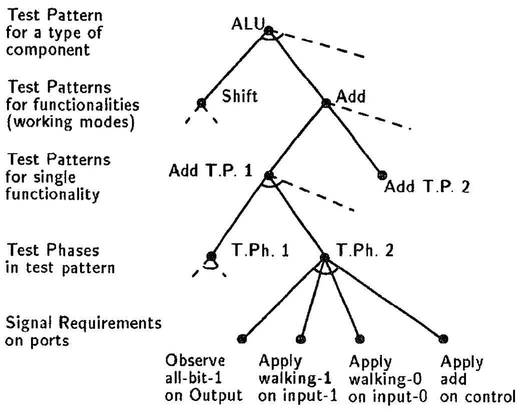
Test Pattern for a type of component
Test Patterns for functionalities (working modes)
Test Patterns for single functionality
Test Phases in test pattern
Signal Requirements on ports
ALU
Shift
Add
Add T.P. 1
Add T.P. 2
T.Ph. 1
T.Ph. 2
Observe all-bit-1 on Output
Apply walking-1 on input-1
Apply walking-0 on input-0
Apply add on control

Figure 3: Test pattern for ALU

A test pattern for a specific type of component is represented in our system as a tree (Figure 3). The root node indicates the component type, an ALU in this case. The second level nodes represent test patterns for each function that this type of component can perform. For example, an ALU can Add, And, Shift, etc. The third level nodes are alternative test patterns for testing the same function. Each of the functions has its own test patterns because a component may not have all the functions mentioned in the generic test pattern definition and its test pattern should vary accordingly. For instance, an ALU without logic operations should be tested differently from one with logic operations. Each test pattern consists of several test phases that are performed in sequence. At the bottom are the signals that need to be routed to and from the I/O ports of a focus. The test pattern for a particular component can be selected by choosing one test pattern for each of its functions and merging them.

Circuits are represented at the register transfer level as in Figure 1. Component behaviors are represented as I/O mapping functions; this predicts how a component will respond to a signal such as a counting sequence. The mapping functions are used to determine subgoals for handling routing tasks, for example, to determine what signals must appear on each I/O port of a component when passing a signal.

The system uses *compound component* templates to specify how components can work together to accomplish routing tasks that none of the individual components can accomplish alone. A *compound component* template specifies the required components, the connections between them, the kinds of routing tasks the compound component can handle, the I/O ports at which the compound component handles the routing tasks, and the routing tasks for the system to accomplish further.

```
(1)        Type: Counter
(2)     Purpose: Generate
(3) Handled RTs: Counting, Exhaustive
(4) Required
    Components
    - Incrementor
    - Register
(5) Required
    Connections
(6) Further RTs
(7) Handling
    Port
```

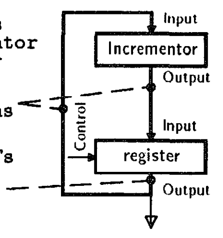

Figure 4: A compound component

Figure 4, for example, shows a compound counter that can generate a counting sequence. Its required components are a register and an incrementor, connected in a loop feeding each other.

To match compound component templates to a circuit, each of the required components must match a component, and each of the required connections must match a signal path. The template of Figure 4 matches the circuit in Figure 1, with the register matched to the uPC, the incrementor matched to the Incrementor, the connection from the output of the incrementor to the input of register matched to the Multiplexor working at the proper mode, etc.

So far we have discussed the system's knowledge about test generation. Now we introduce how components can be modified to repair test generation failures.[1] Our component modifications are all additive, that is, components are modified to perform *more* functions, never fewer. Con-

[1] There are other strategies, such as swapping components to get different test patterns as in [Zhu and Breuer, 1985], that are not currently included.

sequently, the circuit can always perform its original functions after the modification.

The system uses *maximum function sets* to represent what functions can be efficiently added to a particular component. Each type of component has its maximum function set, which includes all the functions commonly associated with it. For instance, the maximum function set for register includes *load, shift, linear-feedback-shift*, etc. When a function is needed for a component to handle a routing task, and the function is in the maximum function set of the component but is not currently implemented, it can be added to the component through modification.

## 5 Constraint Relaxation

This system uses a constraint relaxation mechanism for three purposes: to control the search, to represent preferences between solutions, and to represent criteria for solution validity. For example, without a constraint explicitly ruling out solutions that loop (i.e., a signal going in a circle), the program would produce many such low-quality solutions.

| Category | Concern |
|---|---|
| sharing controls | Incompatible control for test segments |
| sharing modifications | Incompatible modifications to the same component |
| control-observe intersection | Control path crosses observation path |
| signal intersection | Two control paths intersect or two observation paths intersect |
| loop | A signal path intersects itself |
| protect focus | A focus is used to test itself |

Table 1: Constraint Categories

| |
|---|
| use-component-once (strictest) |
| use-component-once-except-for-focus |
| loop-signal-changing |
| loop-signal-stable-with-purpose |
| loop-signal-stable (validity boundary) |
| pass (weakest) |

Table 2: The "loop" constraint category

In our system, most of the constraints on the solutions are organized in a two-level hierarchical structure. Constraints are first divided into 6 categories according to the parts of the solutions they are concerned about (Table 1). For instance, the "loop" constraint category concerns whether a signal path intersects with itself, i.e., whether it forms a loop.

Constraints in each category are then organized by a strictness ordering, that is, if a constraint is violated, all the constraints in the same category that are stricter are also violated. Therefore the violation of constraints in one category can be characterized by the weakest constraint violated. Table 2 shows the constraints in the "loop" category.

**Search Control**

To reduce the search space, the system first generates solutions incrementally in order to take advantage of the fact that whenever a constraint is violated in a partial solution, it is violated in any solution built from the partial solution. Whenever the system adds a building block to a partial solution, it checks whether the resulting partial solution is violating any constraint; if so, the resulting partial solution will be *suspended*.

Second, the system starts with the strictest constraints – stricter than needed to guarantee the validity of solutions – and it relaxes the constraints gradually when there are not enough solutions under the enforced constraints. Since the stricter the constraints, the smaller the search space, and heuristically, the higher quality the solutions, the system is likely to be searching in the smallest search space that contains the best solutions.

**Solution Validity and Preference**

In addition to search control, the constraint relaxation mechanism also captures knowledge about solution validity and preferences. Preference is represented as a relaxable constraint as explained above. Validity is represented as a validity boundary in each constraint category. A validity boundary is the weakest constraint in a category that still guarantees the validity of a solution. An example of the validity boundary is the "loop-signal-stable" category in Table 2, which checks resource contention within a signal path. Usually the system will not relax the constraints beyond the validity boundary. However, if the system cannot find *any* valid solution, it will relax the constraint further, producing a partial solution for examination by the designer, to help him fix the remaining testability problems.

**Constraint Relaxation**

When all partial solutions are suspended before the system finds a given number of solutions, the constraints are relaxed so that some of the suspended solutions may be completed. Each time, the system relaxes the constraints *minimally*, that is, *just enough* to re-invoke at least one suspended partial solution. This is done in three steps:

1. For each of the constraint categories, collect the weakest violated constraints from each of the suspended partial solutions into a Weakest-violation set. If a constraint in this set is relaxed, at least one of the suspended partial solutions will be re-invoked.
2. Collect the strictest constraint in each of the categories from the Weakest-violation set to form a Strictest-Weakest-violation set. This set contains the candidates for a *minimal* relaxation.
3. Relax the constraints in the Strictest-Weakest-violation set one at a time according to the category order in Table 1 until one suspended partial solution is re-invoked.

Unlike the constraints within a category, the constraint categories do not have a clean logical relationship, i.e., given that a partial solution violates some constraints in one category, little can be said about whether the partial solution violates any constraint in other categories. Therefore the order of relaxation among the categories is based on a heuristic: relax the constraint first that is related to the latest stage of solution construction. For instance, given that our system constructs one signal path at a time, the *control-observe intersection* category (involving *two* signal paths) is related to a later construction stage than that of the *loop* category (involving only *one* signal path). Hence the former is relaxed earlier. This heuristic can be justified by noting that, among all the suspended partial solutions, those at the latest stage of construction are closest to completion, therefore re-invoking them first is likely to yield complete solutions with least constraint relaxation.

## 6 Related Work

This research has been inspired by the flexibility and precision demonstrated by human DFT experts. For instance, multiplexors are used to partition the MC68020 and on-chip signature analysis is used only where the accessibility is poor [Kuban and Salick, 1984]. As one test expert remarked, the strategy is to "introduce just enough hardware to make the circuit testable." Our research is an effort to automate some of the techniques used by human experts.

The work on test generation in [Shirley, 1986; Shirley *et al.*, 1987] has had a strong impact on this research. Shirley's work recognizes that test generation effort can be traded off against DFT effort. Therefore, it may be appropriate for a test generator to give up quickly on the hardest portions of a circuit, when DFT techniques can solve the problem more inexpensively. This is the kind of test generator needed to identify testability problems. The point at which the test generator gives up can be chosen based on the relative costs of generating tests vs modifying the circuit.

Horstmann's DFT system [Horstmann, 1983] takes a design rule approach, using rules from LSSD design standards. Abadir's DFT system [Abadir and Breuer, 1985] uses a "testable structure embedding" approach, employing general circuit structure models, similar to our compound components, to represent structured DFT methods. Our approach differs from these DFT systems in the following aspects.

- These systems tend to prevent testability problems from arising while our system solves testability problems as they arise. Previous DFT systems define a testability problem to be either a design rule violation [Horstmann, 1983] or a testable structure mismatch [Abadir and Breuer, 1985]. Rule violations and structure mismatches are only heuristically related to real testability problems in a circuit. This uncertainty forces a conservative strategy that can result in unnecessary modifications.
- Our approach examines more of a circuit's potential behavior than previous systems and, therefore, can use existing components in a larger variety of ways. For example, our system can use a register as part of a counter but previous systems do not.
- Our approach can employ a larger variety of DFT techniques, both structured and ad hoc DFT techniques, flexibly. In comparison, Horstmann's system specializes only in LSSD; Abadir's system bundles components that accomplish control/observation tasks with the focus, and thus has a coarser granularity of testable structure than our system does if both are viewed in terms of testable structure matching.

Zhu's system [Zhu and Breuer, 1985] is more an optimization system than a DFT system. This system specializes in replacing components, i.e., selecting from candidate replacements according to trade-offs among incompatible requirements. In a sense, we are solving a different problem; Zhu's system repairs TG failures caused by components that have no test patterns by swapping in components with *known* test patterns; our system repairs TG failures in signal routing.

This research has been inspired in part by [Horstmann, 1983; Abadir and Breuer, 1985]. However, in our view these DFT systems fail to answer adequately the critical question about what a testability problem is, an issue that has been central to this research. We define a testability problem as a test generation failure, use a test generator to locate testability problems, and organize circuit modifications according to TG failures they repair.

## 7 Current Implementation

### Status

This research is still at the prototype stage, demonstrating the plausibility of our approach. We think that test generation knowledge accounts for large part of the flexibility and precision of human DFT experts. The example shown in the introduction of this paper is interesting because it shows that our system does not introduce DFT hardware on portions of the circuit which we already know how to test; and where real testing problems exist, the system introduces DFT hardware to solve the actual problems, e.g., it introduces only a modification to the output side of the ROM. Additional solutions for all the five components in the sequencer part of MAC-2 and for a circuit from [Abadir and Breuer, 1985] can be found in [Wu, 1988].

To date the system has not been tested on real circuits. What remains to be seen is how this approach scales up with real circuits and whether precise DFT modifications actually yield lower total DFT overhead than would result with a more structured approach.

### Limitations

The test generation process underlying our DFT process is computationally intensive since it involves satisfying conjunctive goals. When more capability is added to the system in order to make wider changes to circuits (e.g., adding connections between internal circuit nodes), the complexity problem will become more acute. Using more abstract or hierarchical circuit representations might help, but more experiments with real circuits are needed.

Our approach is TG-failure-driven. Therefore the system can employ only DFT techniques that can be viewed as repairs to TG-failures. Other techniques, such as partitioning a circuit, or using bus structure to reduce the TG complexity, fall outside our framework.

Our approach provides a framework for suggesting precise DFT modifications. However, due to a lack of related knowledge, the system is currently incapable of directly evaluating the resulting chip area overhead (requiring layout knowledge), test time (requiring details of signal sequences), fault coverage (requiring quality of test patterns), etc. The work in [Wu, 1988] considers this issue in more detail.

Finally, our approach is not intended to deal with gate-level circuits. That level of detail is, first of all, computationally impractical for microprocessor-scale circuits. In addition, in order to avoid expensive late design changes, our system is intended to work at early design stages when only high-level circuit descriptions are available. This seems appropriate since many DFT techniques are concerned with only high-level circuit structures, e.g., built-in self testing.

**Future Directions**

Ideally, testability should be considered while designing a circuit. However, due to the scale of VLSI circuits, simply providing the device functionality correct is very difficult. As a result, it is common practice to pay attention only to functionality at first, then deal with secondary goals like testability by debugging, minimally perturbing the design while maintaining the primary goals. Our DFT system accomplishes a variety of "minimum perturbation" because it works only on true testing problems (as defined by test generation failures) and because it has a library of minimal design modifications indexed by failure type.

The general idea employed in our approach is to use a simulation process to find defects in given design, then use the defects to guide the redesign process. Possible additional applications of this idea are design for manufacturability and design for diagnosability. For instance, simulating how parts of a machine can be put together might reveal assembly problems in the design and the solutions to them.

## 8 Conclusion

Knowledge about test generation is critical to constructing a competent DFT system, yet this knowledge 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.

The implemented system currently has only primitive domain knowledge and needs more work. It can modify a circuit by adding functions to components, but cannot add connections due to a lack of circuit layout knowledge. Except for the constraint relaxation mechanism, it does not have a sophisticated evaluation function. However, armed with the knowledge of circuit testing behavior and test generation, it has already demonstrated its ability to integrate different DFT techniques and to introduce only sharply focused modifications on a textbook microprocessor, an ability that is missing in previous DFT systems.

## Acknowledgments

The author wishes to thank Prof. Randall Davis for supervising this work and Mark Shirley for many discussions. Both have made great contributions to the ideas presented here and to the presentation of the paper. Gordon Robinson, of GenRad, provided many discussions on testing, Walter Hamscher and Reid Simmons carefully read the draft and offered many useful comments, while Choon P. Goh supplied encouragement, a careful reading of the draft and many useful comments.

## References

[Abadir and Breuer, 1985] 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.

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

[Davis, 1982] 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.

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

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

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

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

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

[Robinson, 1985] Gordon D. Robinson. What is Testability? Feb 1985. Available from the author.

[Shirley *et al.*, 1987] 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.

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

[Shirley, 1986] 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.

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

[Wu, 1988] Peng Wu. *Test Generation Guided Design for Testability.* Master's thesis, Massachusetts Institute of Technology, May 1988.

[Zhu and Breuer, 1985] Xi-an Zhu and Melvin A. Breuer. A knowledge based system for selecting a test methodology for a PLA. In *Proc. 22nd Design Automation Conference*, pages 259–265, June 1985.