BETA: BEHAVIORAL TESTABILITY ANALYZER AND ITS APPLICATION TO HIGH-LEVEL TEST GENERATION AND SYNTHESIS FOR TESTABILITY

Chung-Hsing Chen

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Approved for Public Release. Distribution Unlimited.
BETA: BEHAVIORAL TESTABILITY ANALYZER
AND ITS APPLICATIONS TO
HIGH-LEVEL TEST GENERATION AND SYNTHESIS FOR TESTABILITY

BY
CHUNG-HSING CHEN
B.S., National Taiwan University, 1985
M.S., University of Massachusetts at Amherst, 1989

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1993

Urbana, Illinois
In this thesis, a behavioral-level testability analysis approach is presented. This approach is based on analyzing the circuit behavioral description (similar to a C program) to estimate its testability by identifying controllable and observable circuit nodes. This information can be used by a test generator to gain better access to internal circuit nodes and to reduce its search space. The results of the testability analyzer can also be used to select test points or partial scan flip-flops in the early design phase. Based on selection criteria, a novel Synthesis for Testability approach called Test Statement Insertion (TSI) is proposed, which modifies the circuit behavioral description directly. Test Statement Insertion can also be used to modify circuit structural description to improve its testability. As a result, Synthesis for Testability methodology can be combined with an existing behavioral synthesis tool to produce more testable circuits.
DEDICATION

To my father, my wife and the memory of my mother
ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my research advisor, Professor Daniel G. Saab, for his enthusiasm, encouragement and professional guidance throughout this thesis work. I also like to thank my committee, Professors Prith Banerjee, Kent Fuchs, Ibrahim N. Hajj, Wen-mei Hwu and Janak Patel for their time and effort in reviewing this thesis.

Last, but not least, I wish to thank my parents Chao-Chin Chen and Bee-Fang Yeh and my wife Emerald Chang for the love, encouragement and support they have provided for many years.

This research was supported in part by Semiconductor Research Corporation Contract 91-DP-109 and in part by NASA under contract NASA NAG 1-613.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION ................................</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview ..................................</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Previous Works on Testability Analysis</td>
<td>1</td>
</tr>
<tr>
<td>1.2.1 SCOAP ..................................</td>
<td>2</td>
</tr>
<tr>
<td>1.2.2 TMEAS ..................................</td>
<td>3</td>
</tr>
<tr>
<td>1.2.3 PREDICT ................................</td>
<td>4</td>
</tr>
<tr>
<td>1.2.4 I-path, F-path, S-path ..................</td>
<td>7</td>
</tr>
<tr>
<td>1.3 Previous Works on Synthesis for Testability</td>
<td>8</td>
</tr>
<tr>
<td>1.4 An Approach for Testability Analysis ....</td>
<td>10</td>
</tr>
<tr>
<td>1.5 An Approach for Synthesis for Testability</td>
<td>11</td>
</tr>
</tbody>
</table>

| 2 BEHAVIORAL TESTABILITY ANALYSIS ............ | 15 |
| 2.1 Behavioral Description ...................... | 15 |
| 2.2 Path Analysis ................................ | 18 |
| 2.3 Controllability ................................ | 21 |
| 2.3.1 Controllable type, $CC$ .................. | 21 |
| 2.3.2 Controllability calculation .............. | 38 |
| 2.3.3 NCC handling ................................ | 39 |
| 2.3.4 NC analysis ................................ | 41 |
| 2.3.5 Loop handling ................................ | 44 |
| 2.4 Observability ................................ | 47 |
| 2.4.1 Observability types ....................... | 47 |
| 2.4.2 Observability calculation ............... | 53 |
| 2.5 Results .................................... | 54 |

| 3 BEHAVIORAL SYNTHESIS FOR TESTABILITY ....... | 59 |
| 3.1 The Selection Process ....................... | 59 |
| 3.1.1 Complexity ................................ | 60 |
| 3.1.2 Heuristic approach ......................... | 61 |
| 3.2 Test Statement Insertion ...................... | 63 |
| 3.2.1 Methodology ................................ | 63 |
| 3.2.2 Comparison ................................ | 67 |
| 3.3 Results .................................... | 69 |
4 A PROBABILISTIC APPROACH FOR EVALUATION AND SYNTHESIS FOR TESTABILITY ............................................. 74
4.1 Introduction ........................................ 74
4.2 Probabilistic Controllability Evaluation .................... 75
  4.2.1 Derivation of PCI(N) ................................ 75
  4.2.2 Derivation of PPCI(p, N[b]) ........................ 79
  4.2.3 Derivation of SPCI(p, S_i, N[b]) ..................... 81
  4.2.4 PCF(AND, CPPCI(p, S_i, A[A_b]), CPPCI(p, S_i, B[B_b])) 82
  4.2.5 PCF(ADD, CPPCI(p, S_i, A), CPPCI(p, S_i, B)) ........ 84
  4.2.6 PCF of other functions .............................. 84
4.3 Probabilistic Observability Evaluation ........................ 85
4.4 Probabilistic Approach for Synthesis for Testability .......... 90
  4.4.1 Test point selection ............................... 90
  4.4.2 Testability modification ............................ 91
4.5 Results ............................................. 92

5 SUMMARY .............................................. 96

REFERENCES .......................................... 99

VITA ................................................. 102
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>56</td>
</tr>
<tr>
<td>2.2</td>
<td>57</td>
</tr>
<tr>
<td>2.3</td>
<td>58</td>
</tr>
<tr>
<td>2.4</td>
<td>58</td>
</tr>
<tr>
<td>3.1</td>
<td>69</td>
</tr>
<tr>
<td>3.2</td>
<td>70</td>
</tr>
<tr>
<td>3.3</td>
<td>70</td>
</tr>
<tr>
<td>3.4</td>
<td>71</td>
</tr>
<tr>
<td>3.5</td>
<td>72</td>
</tr>
<tr>
<td>3.6</td>
<td>73</td>
</tr>
<tr>
<td>4.1</td>
<td>77</td>
</tr>
<tr>
<td>4.2</td>
<td>85</td>
</tr>
<tr>
<td>4.3</td>
<td>92</td>
</tr>
<tr>
<td>4.4</td>
<td>93</td>
</tr>
<tr>
<td>4.5</td>
<td>94</td>
</tr>
<tr>
<td>4.6</td>
<td>94</td>
</tr>
<tr>
<td>4.7</td>
<td>95</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Sample circuit</td>
<td>6</td>
</tr>
<tr>
<td>1.2 1-controllability calculation in PREDICT when line 6 is 1</td>
<td>6</td>
</tr>
<tr>
<td>1.3 1-controllability calculation in PREDICT when line 6 is 0</td>
<td>6</td>
</tr>
<tr>
<td>1.4 I-mode example</td>
<td>7</td>
</tr>
<tr>
<td>1.5 I-path example</td>
<td>8</td>
</tr>
<tr>
<td>1.6 Synthesis for testability system outline</td>
<td>13</td>
</tr>
<tr>
<td>1.7 Testability Modifier outline</td>
<td>14</td>
</tr>
<tr>
<td>2.1 Sample circuit described in behavioral description</td>
<td>17</td>
</tr>
<tr>
<td>2.2 CFG example</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Check $\text{var} \Rightarrow {R_i}$</td>
<td>25</td>
</tr>
<tr>
<td>2.4 Check consistency C2</td>
<td>27</td>
</tr>
<tr>
<td>2.5 Example of reconvergent fanout</td>
<td>29</td>
</tr>
<tr>
<td>2.6 Verify Criterion C3</td>
<td>29</td>
</tr>
<tr>
<td>2.7 Branch example 1</td>
<td>32</td>
</tr>
<tr>
<td>2.8 Branch example 2</td>
<td>33</td>
</tr>
<tr>
<td>2.9 Check CC</td>
<td>37</td>
</tr>
<tr>
<td>2.10 A loop example</td>
<td>46</td>
</tr>
<tr>
<td>2.11 Algorithm 5. Check variables' observability</td>
<td>51</td>
</tr>
<tr>
<td>2.12 Microprocessor example</td>
<td>55</td>
</tr>
<tr>
<td>2.13 Microprocessor CFG</td>
<td>56</td>
</tr>
<tr>
<td>3.1 Test Statement Insertion</td>
<td>64</td>
</tr>
<tr>
<td>3.2 The effect of TSI on structure diagram</td>
<td>64</td>
</tr>
<tr>
<td>3.3 TSI algorithm</td>
<td>66</td>
</tr>
<tr>
<td>4.1 A branch example</td>
<td>90</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 Overview

Test generation involves searching all possible input combinations to find an input pattern or a sequence of input patterns which produce erroneous output response in the presence of a physical defect. Due to the increasing complexity of VLSI circuits, test generation has become a very time-consuming process. It is known to be an NP-complete problem [1]. Testability measures [2]-[8] have been used in a preprocessing stage to speed up test generation. Since test generation consists of controlling and observing the internal nodes of a circuit, testability measures usually use the concepts of controllability and observability to measure the difficulty of testing a design. Traditional testability measurement tools consider only the structural circuit description which consists of an interconnection of gates or relatively small functional modules. This limits their application and results in a high loss of insight about the control signals in a circuit.

1.2 Previous Works on Testability Analysis

Several different low-level testability analysis approaches [2, 3, 4, 5, 6] have been proposed in the literature. Also, Abadir [25, 26] and Freeman [27] proposed the concepts
of \textit{I-path}, \textit{F-path}, and \textit{S-path} for the high-level circuits. Here, we shall briefly discuss some of these methods.

1.2.1 SCOAP

\textit{SCOAP} [2] is the first popular testability measure. It is intended for use at the logic-level, and it is based on controllability and observability measures. \textit{SCOAP} considers circuits made up of two types of nodes, namely combinational nodes representing logic gates and sequential nodes representing elements with memory such as flip-flops. Two kinds of measures, combinational and sequential, are defined for the controllability and observability of each node. This leads to a total of six measures associated with each node \( N \) in the circuit.

- \textit{Combinational Controllability}, \( CC_1(N) \) or \( CC_0(N) \): the number of combinational nodes within the circuit whose values must be specified in order to set \( N \) to logic value 1 or 0.

- \textit{Sequential Controllability}, \( SC_1(N) \) or \( SC_0(N) \): the number of sequential nodes within the circuit whose values must be specified in order to set \( N \) to a logic value 1 or 0.

- \textit{Combinational and Sequential Observability}, \( CO(N) \) and \( SO(N) \): the number of combinational (or sequential) nodes whose values must be set in order to propagate a change in the value at \( N \) to a primary output.
Take a simple 2-input AND gate for example. Let the inputs be \( A \) and \( B \) and the output be \( C \). Then,

\[
CC_0(C) = MIN[CC_0(A), CC_0(B)] + 1
\]
\[
CC_1(C) = CC_1(A) + CC_1(B) + 1
\]
\[
SC_0(C) = MIN[SC_0(A), SC_0(B)]
\]
\[
SC_1(C) = SC_1(A) + SC_1(B)
\]

Initially, primary circuit inputs have a combinational controllability value equal to 1, and a sequential controllability value equal to 0. Primary outputs have both combinational and sequential observability values equal to 0. The controllability and observability measures of all other nodes are initially set to infinity. The controllability measure of each node is computed during a forward trace from the primary inputs to the primary outputs. The observability measure is computed by tracing backward from the primary outputs to the inputs.

### 1.2.2 TMEAS

TMEAS [3] is one of the earliest testability measures. The circuit is represented as a directed graph, with nodes representing functional modules and links representing signal paths. A node may represent a gate or a register transfer-level module, and each link may represent a number of connections between modules. Sequential components are modeled as nodes with implicit feedback links.
TMEAS derives controllability and observability values for each link. These values are between 0 and 1, where 1 indicates perfect controllability or observability. The input controllability of a node is defined as the average of the controllabilities of its input links, and the output controllability of a node is the average of the controllabilities of its output links. Similarly, the output observability of a node is defined as the average of the observabilities of its output links.

The Controllability Transfer Factor (CTF) and Observability Transfer Function (OTF) are associated with each node (a gate or a module) in the circuit. They are defined as follows.

- The CTF of a node specifies the ratio of its output controllability to its input controllability and is determined from its input/output mapping. It reflects the evenness of the input/output mapping, as measured by the number of 0's and 1's, with higher values indicating more even distribution. For example, for a single output node, its CTF equals 1 if exactly half of all input combinations make the output to be 1.

- The OTF of a node specifies the ratio of the input observability of the node to its output observability. The OTF attempts to estimate the extent to which the input values of a node can be determined by observing its outputs. An OTF of 1 indicates that a change of value on any input link of a node will change the signal value on an output link, regardless of the values of the other inputs.

1.2.3 PREDICT

As alternative to SCOAP-like testability analysis is to model controllability and observability as probability. Given a signal line $l$ in the circuit, the $I$-controllability ($0$-controllability) on $l$, denoted as $C1(l)$ ($C0(l)$), can be defined as the probability that $l$
is set to 1 (0) by a random vector. Similarly, *1-detectability* (*0-detectability*), denoted as $D_1(l)$ ($D_0(l)$) on $l$, is defined as the probability that line $l$ stuck-at 1 (0) can be detected by a random vector. There are several probabilistic measures of testability based upon the above definition. *PREDICT* [6] is one of the most popular probabilistic testability measures for combinational circuits.

In *PREDICT*, $D_1(l)$ and $D_0(l)$ are expressed as follows:

$$D_1(l) = C_0(l) \cdot B_0(l)$$
$$D_0(l) = C_1(l) \cdot B_1(l)$$

and $B_1(l)$ ($B_0(l)$) denotes the probability of observing $l$ when it is set to 1(0).

The signal dependency due to reconvergent fanout complicates the computation of controllability and detectability. In *PREDICT*, Seth used a *divide and conquer* approach by using the concept of a *supergate*, which is a subcircuit of the original circuit and completely includes reconvergent fanouts.

**Example:** Take the circuit in Figure 1.1 as an example to illustrate how to compute controllability in *PREDICT*. The number along each line denotes the line number of that line. Assume that all of the inputs have 0.5 probability to be 1 or 0. There are two maximal *supergates* in this circuit. It would be easy to find that $C_1(6)$ is equal to 3/4 and $C_0(6)$ is 1/4. Then, Figure 1.2 shows the 1-controllability for the lines in the bigger supergate given that line 6 is set to 1. Figure 1.3 shows the same probability given that line 6 is 0. Then, $C_1(11)$ can be determined as follows:

$$C_1(11) = \frac{3}{4} \cdot \frac{5}{8} + \frac{1}{4} \cdot \frac{1}{2}$$
Figure 1.1 Sample circuit.

Figure 1.2 1-controllability calculation in PREDICT when line 6 is 1.

Figure 1.3 1-controllability calculation in PREDICT when line 6 is 0.
1.2.4 I-path, F-path, S-path

Abadir and Breuer proposed the concepts of I-mode and I-path in [25, 26]. There exists an identity mode (I-mode) for a module $M$ if $M$ has a mode of operation so that the data on the input of $M$ can be transferred to the output without change. For example, the multiplexer shown in Figure 1.4 has two I-modes. One is from node $A$ to $OUT$ when $select$ is equal to 0. The other is from node $B$ to $OUT$ when $select$ is equal to 1. Latches, registers, ALUs and buses are the other examples of high-level modules with I-mode.

There is an Identity path (I-path) from a node $A$ to another node $B$ in the circuit, if the data on $A$ can be transferred to $B$ without any changes. An I-path consists of a chain of modules, and each module possesses I-mode. Figure 1.5 shows an I-path example. The output of module $A$ can be set to the input of module $B$ without change by properly setting $MSelect$ and $BSelect$. As a result, there is an I-path from the output of $A$ to the input of $B$, and this I-path consists of three I-modes.

The concepts of I-mode and I-path are expanded to S-mode and S-path. There is a sensitized-mode (S-mode) between an input port $A$ and output port $B$ of a module $M$ if $M$ has an mode of operation such that there is a one-to-one mapping between $A$ and $B$. 
As a result, any fault which appears on $A$ will be propagated to $B$ through $M$. Adder is one such example having $S$-mode. Then, $S$-path refers to a chain of modules having 0 or more $I$-modes and at least one $S$-mode. Similarly, $T$-mode refers to onto mapping between an input port and an output port. One such example is an array of inverters which maps input vector $A$ into $\overline{A}$. Also, $T$-path is defined as a chain of modules which consists of $I$-mode and $T$-mode.

1.3 Previous Works on Synthesis for Testability

Currently, the two most popular methods used to enhance circuit testability are Test Point Insertion [9, 10] and Partial Scan Design [16, 17, 18, 19]. Test Point Insertion increases controllability and observability by inserting controllability and observability
points in the circuit. Partial Scan Design is an alternative to Full Scan Design [20, 21], in which only part of the flip-flops are put into the scan chain.

Research on test point selection has concentrated only on combinational circuits [9, 10, 14, 15]. In [9, 10], methods are proposed for inserting test points to make a combinational circuit completely testable by a small number of test vectors. Krishnamurthy in [14] uses a dynamic programming approach for selecting test points for combinational fanout-free circuits to minimize the number of test vectors needed. In [15], Pomeranz and Kohavi propose a testing-module insertion approach for general combinational circuits. However, due to the advances of combinational test generation [11, 12], emphasis has shifted to testing sequential circuits, which were not considered in the existing test point selection techniques. Although Test Point Insertion still helps the sequential test generation [13], the optimal combinational test point selection problem is NP-hard [14], and the sequential test point selection problem is even harder.

As a result, Partial Scan drew a lot of attention in recent years as an appropriate alternative to full scan and to Test Point Insertion. Unlike Test Point Insertion, only the flip-flops are considered as candidates for test points. Once a flip-flop that has a large impact on the overall testability of the circuit is found, it is selected and placed into a scan chain. There are three main categories for Partial Scan selection methodologies [19]: testability measure-based [16], structural analysis-based [22] and test generation-based [17]. In [16], a testability measure-based approach is used. The usefulness of traditional testability measures [2, 3, 6, 7] for identifying hard-to-detect (HTD) areas is questionable, since it is difficult to acquire the global picture of the circuit's behavior from a low-level structural description. In a structural analysis-based approach [22], a minimum set of flip-flops is selected to break sequential cycles in the circuit. The drawback of this approach is
that the circuit's functionality is ignored during the selection process. A test generation-based approach [17] cannot be applied when a test generator is not available and is very time-consuming, especially in the early design phase. In addition, one common drawback for all of the existing Test Point Insertion or Partial Scan methods is that they fail to handle high-level modules.

The key factor to the success of Test Point Insertion and Partial Scan is the selection of test points.¹

1.4 An Approach for Testability Analysis

In this thesis, first, an approach for computing testability is proposed. This approach, called BETA (Behavioral Testability Analyzer) [23], can provide guidance to test generators and synthesis tools. It is based on analyzing the circuit's behavioral description in the form of a Control Flow Graph (CFG). The CFG is provided either by the designer or by high-level synthesis tools [29, 30]. In BETA, a path analysis on the CFG is performed first. Then, variable classification is used to explore the intrinsic controllability and observability of the circuit. This procedure examines the controllability and observability of every variable and classifies them into different groups. Unlike other testability measures that compute only measures of difficulty, BETA derives the exact sequence for justifying and propagating certain variables. For the most controllable and observable registers, an algorithm is developed to derive the shortest sequence of paths to be traversed along the CFG in order to control and observe these variables. This alleviates blind searching

¹In Partial Scan, the only possible test points are the outputs of flip-flops. Throughout this thesis, test point refers to either regular test points or the outputs of flip-flops.
during test generation. Based upon this classification, guidance can be provided to a high-level test generator [28].

*BETA* is applicable only when the structure of the given control flow graph remains the same under faulty conditions. For faults which change the CFG structure (denoted as *control faults*, as opposed to *data faults*), no direct guidance is provided. However, the guidance for data faults is still useful in testing control faults, since the activation and propagation of control faults require variable justification and propagation.

*Variable classification* is also useful in pointing out hard-to-control and hard-to-observe areas of the circuit. This information can be fed back to the high-level synthesis or to the designer. Based on this feedback, alternative designs can be explored. This motivates our research on Synthesis for Testability (SFT).

1.5 An Approach for Synthesis for Testability

The proposed Synthesis for Testability (SFT) approach is based on *BETA* [23]. A key factor crucial to SFT is the identification of HTD areas. *BETA* introduces the concept of *Completely Controllable (CC)* and *Completely Observable (CO)* to distinguish easy-to-test and hard-to-test areas. Nodes that are not identified as *CC* or *CO* are more likely to be HTD nodes, and are good candidates for test points. Due to hardware restrictions, not all of the HTD areas can be modified by inserting test points. This requires a test point selection process among these HTD nodes. In this thesis, two test point selection algorithms are presented. This method can be used for both combinational and sequential test points. In addition, a novel SFT technique called *Test Statement Insertion (TSI)* is also presented. This is used to enhance the testability of a circuit by directly modifying
its behavioral description rather than its structure. *Test Statement Insertion* requires less area overhead and less test application time than do scan techniques (full scan or partial scan) and traditional *Test Point Insertion*. This technique has been integrated with a behavioral-level synthesis tool, as shown in Figure 1.6. The middle part of Figure 1.6 shows that *BETA* plus this *SFT* approach form a bridge between synthesis and test. During the synthesis process, the intermediate product CFG is first taken out of the synthesis tool as input to *BETA*. After the *Testability Modifier*, a modified and testable CFG can be fed back to the synthesis tool to resynthesize the circuit. As a result, behavioral synthesis for testability can be achieved in the early design phase. Figure 1.7 shows the detailed operations performed in the *Testability Modifier*. First, a testability analyzer identifies the HTD areas and diagnoses causes. Second, this information is used in a test point selection process. The designer can then decide to use a traditional approach (test point insertion or scan design) or *TSI*.

The remainder of this thesis is organized as follows. The behavioral testability analysis tool *BETA* is presented in Chapter 2, which also describes the behavioral information used in *BETA*. The proposed synthesis for testability approach is presented in Chapter 3. The first part of Chapter 3 shows the test point selection procedure; the second part is the proposed *Test Statement Insertion*. In Chapter 4, an approach for evaluation and synthesis for random testability is presented. A summary of this thesis is given in Chapter 5.
Behavioal-Level Synthesis Tool

**Figure 1.6** Synthesis for testability system outline.
Testable nodes.
- How to access.

Selection Process
- Test points.

Method?

TPI (or SCAN)  TSI

Testable circuits

Figure 1.7 Testability Modifier outline.
CHAPTER 2

BEHAVIORAL TESTABILITY ANALYSIS

2.1 Behavioral Description

The circuit's behavioral description is provided either by the designer or by behavioral-level synthesis tool. If it is provided by a behavioral-level synthesis, it is in an intermediate format for the circuit's final structural description. In BETA, the behavioral description consists of a symbol table and a statement list. The symbol table describes the variables defined in the circuit, including primary inputs (denoted as INPUT), outputs (OUTPUT), constants (CONSTANT), compiler-generated intermediate variables (VARIABLE) and states (outputs of memory elements, denoted as REGISTER). It also specifies the bit range of each variable. For variable var, the bit range is denoted by Range(var). The statement list is a list of control, assignment, logic, arithmetic and user-defined statements. Control statements consist of IF, SWITCH and CLOCK. There are seven types of assignment statements, including variable split and merge. In addition, there are 19 logic operations and 10 arithmetic operations. Variables appearing on the left-hand side of statements are called results, while those appearing on the right-hand side are called operands. In every statement, each result or operand is specified with a variable name and a range. For example, a symbol A[0 : 2] denotes a variable named A,
with a range bit 0 to bit 2. In the remainder of this proposal, for simplicity, if a symbol
with no bit range is specified, every bit defined in the symbol table of that variable is
used (full range is assumed).

The behavioral description is represented by a directed graph \( G(V,E) \), called the
Control Flow Graph (CFG). A vertex \( v \) in \( V \) corresponds to a statement in the behavioral
description. The root of \( G(V,E) \) is the vertex which corresponds to the first statement
of the behavioral description. Let \( s_i \) represent the statement associated with vertex \( v_i \).
There is an edge \( (v_i, v_j) \) in \( E \) from vertex \( v_i \) to \( v_j \) if and only if

- \( s_i \) is not a control statement and \( s_j \) is the statement following \( s_i \) in the behavioral
description.

- \( s_i \) is a control statement and \( s_j \) is one of the branch destinations.

Throughout this proposal, “vertex” and “statement” will be used interchangeably.

Figure 2.1 shows a sample circuit’s behavioral description, which is similar to the
input format accepted by BETA. The first part of Figure 2.1 is a symbol table, which
describes all of the variables (including inputs and outputs) defined in this circuit. The
second part of Figure 2.1 shows the statement list which can be represented as a graph
(CFG). Figure 2.2 shows this circuit’s CFG. As in Figure 2.2, more insight into the circuit
functionality can be derived from CFG than from the circuit’s structural diagram. For
example, given different values on \( Cin \), we can identify different operations executed in
this circuit.
SymbolTableStart

Cin   INPUT [0:0]
CNT   REGISTER [0:2]
T1    VARIABLE [0:2]
T4    VARIABLE [0:2]
T7    VARIABLE [0:2]

SymbolTableEnd

StatementListStart

SWITCH Cin
CASE 0
ASG 0    T1
ENDCASE
CASE 1
ADD CNT 1 T4
ASG T4    T1
ENDCASE
CASE 2
SUB CNT 1 T7
ASG T7    T1
ENDCASE
CASE 3
ASG 7    T1
ENDCASE
ENDSWITCH
ASG T1    CNT

StatementListEnd

Figure 2.1 Sample circuit described in behavioral description.
2.2 Path Analysis

Definition: A path is a sequence of edges \( \{(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)\} \), where \( v_0 \) is the root of \( G \) and \( v_n \) has no outgoing edge.

Graph \( G \) can be partitioned into a set of paths and analysis is done on these paths. After forming the paths, variable renaming and constant folding are applied to each path to simplify statements by removing intermediate variables. As an example, statement \( A = 1 \) followed by \( B = A + 1 \) is simplified to \( A = 1 \) and \( B = 2 \). Since every result or operand is specified with a bus range, this complicates the implication process. Currently, only symbols with the same variable name and range are implied.

Definition: For a statement \( St_k \) of the form \( z = y \ op \ z \), where \( \op \) is an operator, then this statement defines \( x \), and uses \( y \) and \( z \) (\( x, y \) and \( z \) are symbols).
The process of justification and propagation constitutes the major work of test generation. Therefore, the information on how each node can be justified and propagated in the circuit is very important. To derive this information from CFG leads to the following definitions.

**Definition:** \( JPath(var) \) is the set of paths which can be used to justify at least some portion of \( Range(var) \). They are paths which define \( var[a:b] \), where \( [a:b] \in Range(var) \).

One simple way to derive \( JPath(var) \) is to find the set of paths which define \( var \). However, one variable may be defined more than once with a different range in a path. This complicates the derivation of \( JPath(var) \). Let a path, \( p_i \), define a symbol \( A[R_1] \). If \( p_i \) redefines \( A \) with a different range \( R_2 \) (\( R_1 \) and \( R_2 \) are both in \( Range(A) \)), \( p_i \) can no longer be a \( JPath \) for \( A[R_1] \) if \( R_1 \cap R_2 \neq NULL \). This is because after executing \( p_i \), the value of \( A[R_1 \cap R_2] \) is determined by the second definition. Nevertheless, \( p_i \) can still serve as a \( JPath \) for symbol \( A[R_2] \). As a matter of fact, \( p_i \) can be used to justify \( A[R_1 \cup R_2] \).

A path \( p_i \) is in the \( JPath \) of variable \( var \) if there exists a statement, \( s_j \), in \( p_i \) such that \( s_j \) defines \( var[R] \) (\( R \subseteq Range(var) \)) and no subsequent definition of \( var \) in \( p_i \) redefines any bit in \( var[R] \).

**Definition:** \( PPath(var) \) is the set of paths which can be used to propagate at least some portion of the contents of \( var \).

A path \( p_i \) is in \( PPath(var) \) if there exists a statement in \( p_i \) that uses \( var[R] \) (\( R \subseteq Range(var) \)). Path \( p_i \) cannot be guaranteed to propagate the contents of \( var \) to the primary outputs. This is investigated during the observability derivation.

We need the following definition to determine \( PPath \).
Definition: \( PropVar(p_i, var) \) is a set of symbols \( (V_d[R]) \) which can propagate at least some portion of the contents of \( var \), where path \( p_i \) is used to propagate.

To determine \( PropVar(p_i, var) \), the following definition is needed:

Definition: A variable \( V_d[R] \) is **reachable** from \( var[R_k] \) by path \( p_i \), denoted as \( p_i : var[R_k] \rightarrow V_d[R] \), if there exists a sequence of statements, \( St_1, St_2...St_n \), in \( p_i \) such that the defined symbol in \( St_i \) is used or partially used in \( St_{i+1} \) for \( 1 < l \leq n \), \( var[R_k] \) is used in \( St_1 \) and \( V_d[R] \) is **defined** in \( St_n \).

Each variable \( V_d[R] \) in \( PropVar(p_i, var[R_k]) \) should satisfy

- \( V_d[R] \) is reachable from \( var[R_k] \) in \( p_i \).
- \( var[R_k] \) is not redefined before it reaches \( V_d[R] \). \(^1\)
- \( V_d[R] \) is not redefined afterward in \( p_i \).

Then, \( p_i \) is in \( PPath \) of \( var \) if \( PropVar(p_i, var) \) is not empty.

Definition: \( JContVar(p_i, var) \) is the set of variables needed to be justified (controlled) if path \( p_i \) from \( JPath(var) \) is used to justify \( var \).

To derive \( JContVar(p_i, var) \), the following definition is needed:

Each \( V_j[R_k] \) in \( JContVar(p_i, var) \) satisfies \(^2\)

- \( p_i : V_j[R_k] \rightarrow var \).
- \( V_j[R_k] \) is not defined before \( var \) is defined.

According to the above criteria, \( JContVar \) can be treated as the inputs to path \( p_i \), while justifying \( var \). To use \( p_i \) to justify \( var \), all of the variables in \( JContVar \) have to be

\(^1\)As in \( JPath \), as long as part of \( var[R_k] \) is redefined, this condition fails.
\(^2\)If \( var \) is defined more than once in \( p_i \), the following criteria are based on its last definition.
set properly. Therefore, the size of $JContVar$ is somewhat related to the effort required while executing $p_i$ to justify $var$.

**Definition:** $PContVar(p_i, var, V_d[R])$ is the set of variables needed to be justified if $p_i$ in $PPath(var)$ is used as a propagation path for $var$ and variable $V_d[R] \in PropVar(p_i, var)$ is used to propagate the contents of $var$.

To ensure that the data in $var$ is properly propagated to $V_d[R]$, every variable in $JContVar(p_i, V_d[R])$ has to be justified. Therefore, $PContVar(p_i, var, V_d[R])$ is exactly the same as $JContVar(p_i, V_d[R])$. No extra effort is needed for deriving $PContVar(p_i, var, V_d[R])$.

### 2.3 Controllability

#### 2.3.1 Controllable type, $CC$

Unlike traditional testability measures, BETA first performs variable classification according to each variable's intrinsic controllability and observability. Then, different testability derivations are applied to different types of variables. In this chapter, controllability classification is first addressed.

**Definition:** A *writing sequence* is a sequence of executable paths that can be used to set the contents of a variable to any possible value. These values can be supplied from primary inputs.

**Definition:** A variable is of type *Completely Controllable (CC)* if that variable has a writing sequence.
**Definition:** A nonconstant variable if it is not of type CC is said to be *Non-Completely Controllable* (NCC).

For a variable, var, to be of type CC, the following criteria need to be satisfied for a path \( p_i \) in \( JPath(var) \) \(^3\)

- **C1:** All of the variables in \( JContVar(p_i, var) \) should be of type CC.
- **C2:** All of the variables in \( JContVar(p_i, var) \) can be justified to any possible values simultaneously \(^4\).
- **C3:** If both C1 and C2 are satisfied, after the execution of \( p_i \), var can be any possible value.

In the following sections, each criterion will be examined carefully.

### 2.3.1.1 Criterion 1

According to the definition of \( JContVar(p_i, var) \), the variables in \( JContVar(p_i, var) \) can be treated as the input cone that has to be set up before var can be justified using \( p_i \). Criterion C1 is used to ensure that all of the inputs are controllable.

### 2.3.1.2 Criterion 2

Criterion C2 ensures that a sequence of paths, denoted as \( PrePath(p_i, var) \), can be found such that after the execution of \( PrePath \), all of the variables in \( JContVar(p_i, var) \) can be set to any possible values simultaneously. As a result, if \( p_i \) is executed after

---

\(^3\)These conditions are only sufficient.

\(^4\)To meet this criterion, a sequence of paths is required to be executed before \( p_i \). This issue will be addressed later.
PrePath(p, var), every possible value needed in \( JContVar(p, var) \) in order to make var 
CC can be justified by PrePath.

The algorithm to examine C2 depends on whether the sequential elements of the 
circuit have the HOLD property or not. Two different algorithms are presented in the 
following sections.

Without HOLD Property: Sequential elements in most circuits do not have the 
HOLD property. In this case, the following consideration is needed. Let \( PrePath(p_i, 
var) \) consist of \( p_1, p_2 \ldots p_n \), where \( p_k \) is executed before \( p_{k-1} \) and \( p_1 \) is the path executed 
right before \( p_i \). Assume that \( p_i \) is used to make var CC. Finding the appropriate \( p_i \) is 
exactly the same as finding the \( p_i \) for var which satisfies all three CC criteria, except 
that instead of making one variable var, all of the variables in \( JContVar(p_i, var) \) have 
to be CC after the execution of \( p_i \). If such a \( p_i \) is found, the following set of variables 
needs to be justified:

\[
\{v_i | v_i \in JContVar(p_1, v_j), \forall v_j \in JContVar(p_1, var)\}
\]

Then, it is necessary to find a \( p_2 \) which can justify \( \{v_i\} \). This process is continued 
until \( \{v_i\} \) consists of only primary inputs.

With HOLD Property: If all of the sequential elements have the hold property, there 
is no need to set all \( JContVar(p_i, var) \) simultaneously. Instead, variables in \( JContVar(p_i, 
var) \) can be justified one after the other. Whenever one variable has been justified, 
its content is held and the next variable in \( JContVar(p_i, var) \) is justified. During the 
justification of a variable, the contents of other previously justified variables may be 
destroyed (redefined). This situation can be avoided if \( JContVar(p_i, var) \) is consistent. 
This leads to the following definition:
Definition: A set of variables is consistent if there exists an ordering of these variables, \( \{v_1, v_2...v_n\} \), such that while justifying \( v_i, 1 < i \leq n \), the contents of \( v_j \) are not destroyed, for all \( j, j < i \).

Then, Criterion C2 requires checking the consistency among \( JContVar(p_i, var) \). Note that even though \( JContVar(p_i, var) \) are not consistent, it does not mean that they cannot be justified to any possible values simultaneously. For example, \( v_1 \) and \( v_2 \) are variables in \( JContVar(p_i, var) \). Assume that \( v_1 \) has been justified, and \( p \) is used to justify \( v_2 \). If \( p \) defines \( v_1 \), the original content of \( v_1 \) is destroyed by \( p \). If \( p \) is the only path in \( JPath(v_2) \), \( \{v_1, v_2\} \) are not consistent unless we can justify \( v_2 \) first and the justification of \( v_1 \) does not destroy \( v_1 \). However, as in the case of circuits without the hold property, the execution of \( p \) still satisfies C2 as long as after the execution of \( p \), both \( v_1 \) and \( v_2 \) can be set to any possible values. As a result, consistency among \( JContVar(p_i, var) \) is a sufficient condition for C2. To reduce the complexity, BETA checks only consistency for C2.

To determine whether a set of variables is consistent, we have to know how the justification of each variable in this set affects the other variables.

Definition: Let \( var \Rightarrow \{R_i\} \) denote that no matter how \( var \) is justified, at least one variable in the set of variables \( \{R_i\} \) is destroyed.

Let \( Def(p_i) \) be the set of variables defined in path \( p_i \) and \( JPathC(var) \) denote the set of \( JPath(var) \) which can make \( var \) CC. The algorithm that checks \( var \Rightarrow \{R_i\} \) is shown in Figure 2.3.

Definition: Let \( \{JO_j(p_i, var)\} \) denote a justification ordering for all the variables in \( JContVar(p_i, var) \). If \( R_k \) is the \( j \)-th variable to be justified in \( JContVar(p_i, var) \), \( JO_j(p_i, var) \) is \( R_k \).
input: a set of variables (\{R_i\}) and a variable var.
output: TRUE if var \Rightarrow \{R_i\}. FALSE, otherwise.

Destroy (\{R_i\}, var) {
    for every pi in JPath(var) {
        if ( exists Ri in Def(pi) \&\& Ri in JContVar(pi, var) ) next pi;
        for every R in JContVar(pi, var) {
            for every pj in JPath(var) {
                if ( Destroy (\{R_i\}, R) ) break;
                next pj;
            };
            if ( pj is empty) break;
            next R;
        };
        return (FALSE);
    };
}

Figure 2.3 Check \(\text{var} \Rightarrow \{R_i\}\).
Lemma 1: The set of variables $J_{ContVar}(p_i, var)$ is consistent if and only if there exists an ordering $\{JO_j(p_i, var)\}$ such that $JO_k(p_i, var) \Rightarrow \{JO_1(p_i, var) \ldots JO_{k-1}(p_i, var)\}$ is FALSE for $0 \leq k \leq |JO_j(p_i, var)|$, where $\{JO_1(p_i, var) \ldots JO_{k-1}(p_i, var)\}$ denotes the set of variables which starts from the first element in $\{JO_j(p_i, var)\}$ to the $(k - 1)$-th element.

Proof: Obvious. □

Now, we can direct our attention on how to determine whether $J_{ContVar}(p_i, var)$ is consistent.

Lemma 2: If $var \Rightarrow \{R_i\}$ is FALSE, then $\{R_i\}$ and $var$ are consistent if and only if $\{R_i\}$ is consistent.

Proof: Obvious. □

By Lemma 2, the algorithm shown in Figure 2.4 can be used to determine whether $J_{ContVar}(p_i, var)$ are consistent, i.e., Criterion C2. If an ordering exists, the algorithm outputs a consistent ordering. Otherwise, it returns FALSE. Given a set of variables $\{R_i\}$ to check consistency, Figure 2.4 first finds a variable $R_j$ in $\{R_i\}$ such that $R_j \Rightarrow (\{R_i\} - R_j)$ is FALSE. As a result, while justifying $\{R_i\}$, $R_j$ can be the last one to justify without destroying others. Then, consistency checking is continued for the remaining $\{R_i\} - R_j$ variables. If $\{R_i\}$ is found to be consistent, a justifying ordering for the corresponding consistent $J_{ContVar}$ is also found.

2.3.1.3 Criterion 3

Given a $J_{Path}(var)$ $p_i$, Criteria C1 and C2 check if $J_{ContVar}(p_i, var)$ can be set to any possible combinations. This does not guarantee that $var$ can be justified to any
input : JContReg(pi, var).
output : A consistent ordering, if exists. Return FALSE, otherwise.

Consistent({Ri}) {
    if ( ({Ri}) consists of only one variable) return(TRUE);
    find= FALSE;
    for every Rj in {Ri} {
        if ( Destroy( {Ri}-Rj, Rj) ) next Rj;
        push (Rj, Order);
        find= TRUE;
        break;
    };
    if ( !find) return(FALSE);
    Consistent ( {Ri} - Rj);
};

Figure 2.4 Check consistency C2.
possible values after executing $p_i$. This leads to the examination of Criterion C3. The following issues affect the examination of C3:

- Reconvergent fanout.
- Operation characteristic.
- Branches.
- Multiple define/use.

Each is discussed below.

**Reconvergent Fanout Issue:** Figure 2.5 shows the effect of reconvergent fanout on the determination of $CC$ type variables. In this figure, $A$, $B$ and $D$ are of type $CC$ and we want to determine if $F$ is also of type $CC$. It is possible to produce any possible value on $E$ (or $C$) by adjusting the value on $A$. However, after $A$ has been used for $E$ (or $C$), its value is fixed and cannot make $C$ (or $E$) to be any possible value. In this case, $A$ becomes constrained\(^5\) in statement $s1$ (or $s2$). As a result, one of $C$ and $E$ may not be of type $CC$\(^6\) and $F$ may not be of type $CC$.

To handle reconvergent fanout, during the examination of Criterion C3, *temporary controllability* types are determined and stored in *ConstrainList*. Initially, the temporary controllability of every $CC$ variable is set to $FREE$, and that of the $NCC$ variable is set to $NCC$. Once a $FREE$ variable becomes constrained, its temporary controllability becomes $CONSTRAIN$. If an $NCC$ variable is set to $CC$, its temporary controllability becomes $FREE$.

---

\(^5\)This is because it can no longer be whatever value we want and acts as a constant.

\(^6\)The reason we use “may not” is whether it is of type $CC$ also depends on the operation performed at that statement. This will be explained later.
\[ s_1 : C = A \text{ op} 1 B; \]
\[ s_2 : E = A \text{ op} 2 D; \]
\[ s_3 : F = C \text{ op} 3 E; \]

**Figure 2.5** Example of reconvergent fanout.

- **input**: one variable, \( \text{var} \), and one of its JPath, \( p \).
- **output**: TRUE if \( p \) can make \( \text{var} \) to be CC.

```c
CheckRegFree(var, p) {
    ResetConstrainList();
    for every statement, \( s \), in \( p \) {
        type = ResultConstrainType(s, p);
        /** possible type : FREE, CONSTRAIN, NCC. **/
        if ( \( s \) is branch) {
            if ( type != FREE) return(FALSE);
            if ( BranchVar(s) == RBranchVar(p, var) ) return(TRUE);
        }
        if ( CheckLastDefine(s, p))
            if ( type != FREE) return(FALSE);
        EnterConstrainType(Result(s), type);
    }
    return(TRUE);
}
```

**Figure 2.6** Verify Criterion C3.
Figure 2.6 shows the algorithm to verify Criterion C3. Starting from the first statement in a path, each statement’s operands and operation used are examined to determine the temporary controllability type of that statement’s output. In the next section, we show how to derive the temporary controllability type. Also, in a later section, we describe how to handle a branch statement.

**Operation Characteristic Issue:** Another possibility for a variable var violating C3 is when the operation performed to produce var cannot generate all possible values on var. Consider the statement var = A * 2 and assume that A is of type CC. In this case, the least significant bit of var is always 0 after executing this statement, and it cannot be used for setting var to any possible value. If this is the only statement that defines var in this path, then var is not of type CC. This issue is handled by defining, *Controllability Degree of Freedom* for each operation.

**Definition:** The *Controllability Degree of Freedom*, CDF, of an operation is the number of CONSTRAIN-type operands that it can tolerate and still produce a FREE result, given that all of the operands are not of type NCC.

For example, the CDF of multiplication is 0, since as long as one operand is of type CONSTRAIN, it is possible that not all possible values can be produced at the output.

When a statement is executed, the CDF of that statement’s operation along with the temporary controllability of the operands are used to determine the temporary controllability type of the outcome. This is performed in the procedure ResultConstrainingType() as shown in Figure 2.6.

**Example:** Take Figure 2.5, for example. Assume that we want to determine the controllability type of F, under the condition that A, B and D are all FREE variables and that
the CDF of op1, op2 and op3 are 0, 1 and 0, respectively. First, we have to determine the temporary controllability type of C and E. Since op1 cannot tolerate any CONSTRAIN variable (its CDF is 0), in order to make C FREE, both A and B become CONSTRAIN. On the other hand, op2 can tolerate one CONSTRAIN variable. Even though A is no longer FREE, E becomes FREE since D is FREE. As a result, F is FREE, because both of its operands are of type FREE, even though CDF of op3 is 0.

**Branch Issue:** BETA makes no distinction between control and data signals. Thus, there is no need to assume that the data part and the control part are identified and completely separated. However, from the viewpoint of CFG, control signals can be considered as those variables which affect the control flow of CFG. Thus, the input cone of all of the branch variables (including themselves) is a control signals. To resolve the conflicts between control signals or between control and data signals, we have to consider the branch issue.

Conditional branch statements such as “if” and “switch” in the CFG require special handling. Every conditional branch in the CFG produces at least two paths that share some common statements. To traverse one of these paths, every branch variable has to be properly justified. This effect should be taken into account when deriving JContVar. Let Branch(pi) denote the set of branch variables in path pi. One straightforward way to modify the JContVar of each variable, var, defined in pi, is to include all JContVar(pi, Rb), where Rb \( \in \) Branch(pi).

\[
J\text{ContVar}(pi, \text{var}) = \left[ \bigcup_{R_b} J\text{ContVar}(pi, R_b) \right] \cup J\text{ContVar}(pi, \text{var})
\]

This equation is too pessimistic. Consider Figure 2.7, which is part of a CFG. Assume that no further branches appear after R_b. If statement St_i is the last definition
of \texttt{var} for both paths \texttt{p1} and \texttt{p2}, then there is no need to include \texttt{JContVar}(R_b) into \texttt{JContVar(var)}, because \texttt{var} can be properly defined whether \texttt{R_b} is TRUE or FALSE. Thus, \texttt{JContVar(p1, var)} should be the same as \texttt{JContVar(p2, var)}. Therefore, only those branches above \texttt{R_b} and along \texttt{p1} (\texttt{p2}) should be added to \texttt{JContVar(var)}. This result is not always TRUE. Consider another portion of CFG, shown in Figure 2.8. Although the \texttt{var} in statement \texttt{St_i} is the last definition for \texttt{p1}, it is not the last definition for \texttt{p2}. To use \texttt{St_i} to define \texttt{var}, \texttt{R_b} has to be 0. Therefore, \texttt{JContVar(R_b)} should be put into \texttt{JContVar(var, p1)}. This leads to the following definition:

\textbf{Definition} : \texttt{RBranch(p_i, var)} is the branch variable in \texttt{p_i} such that all of the statements below \texttt{RBranch(p_i, var)} form the largest define-free region in \texttt{p_i} for \texttt{var}.

As an example, \texttt{R_b} is the \texttt{RBranch} for \texttt{var} on both \texttt{p1} and \texttt{p2} in Figure 2.7. In Figure 2.8, however, \texttt{RBranch(p1, var)} is \texttt{R_c}, and \texttt{RBranch(p2, var)} is \texttt{R_d}. This leads to the fact that all of the branch variables in \texttt{p1} below (including) \texttt{RBranch(p1, var)}, do not have to be taken into account while deriving \texttt{JContVar(p1, var[R1])}. 

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{branch_example.png}
\caption{Branch example 1.}
\end{figure}
Figure 2.8 Branch example 2.
Not only are the branch variables affecting the derivation of \( J\text{ContVar} \), but they are also affecting the examination of C3. In Figure 2.6, let statement \( s_p \) be a branch statement with branch variable \( v_b \) in \( p_i \). Assume that \( p_i \) is executed while \( v_b \) equals 0. If \( v_b \) is not of type \( FREE \), we have no control on \( v_b \). As a result, \( p_i \) cannot be properly traversed. In Figure 2.6, if \( v_b \) is not of type \( FREE \), Criterion C3 fails as Figure 2.6 returns FALSE. If all of the branches above \( R\text{BranchVar}(p_i, \text{var}) \) are of type \( FREE \) and the last definition of \( \text{var} \) in \( p_i \) is of type \( FREE \), Figure 2.6 returns TRUE. Note that as in deriving \( J\text{ContVar} \), the branch variables below (including) \( R\text{Branch} \) have no effect on \( \text{var} \). Only the branch variables above \( R\text{Branch} \) count in Figure 2.6.

**Multiple Define/Use Issue:** Another factor which affects the checking of C3 is the multiple define and use issue. It is possible that a variable \( \text{var} \) is defined and/or used more than once in a path \( p \) of CFG. This will affect the identification of \( CC \) because each \( define \) on \( \text{var} \) results in a different temporary controllability type (\( FREE, CONSTRAIN, NCC \)); for each \( use \) of \( \text{var} \), the \( current \) temporary controllability type is used. In addition, this \( current \) controllability type may not be the same as the final controllability type of \( \text{var} \) because the last definition in \( p \) on \( \text{var} \) dominates all of the previous definitions.

Another issue which needs consideration is that each define/use on \( \text{var} \) may not be in \textit{Full Range}. \(^7\) This leads to the following two situations:

1. Every define/use of \( \text{var} \) is in \textit{Full Range}:
   Three more different cases to be considered:
   
   - Multiple uses only:
     
     For each usage of \( \text{var} \), its controllability type is used to determine the output

\(^7\)This means that every bit of \( \text{var} \) is involved in each define/use.
controllability (as explained in the previous sections). At each usage of \textit{var}, its current controllability type has to be identified. If \textit{var} is not defined in path \textit{p}, \textit{var} must be either a primary input or defined in some other path. In the latter case, the final controllability type defined in some other path is used as its current controllability type. If there is more than one path which defines \textit{var}, the more controllable one is used as \textit{var} current controllability type, since it is better to use this path to justify \textit{var}.

- Multiple defines only:
  After each definition of \textit{var}, its current controllability type is changed, depending on the operands and operation used in each definition. After execution path \textit{p}, only the last definition counts. If \textit{p} is used to control \textit{var}, the controllability type of \textit{var} is completely determined by the last definition.

- Multiple defines and uses:
  This is a combination of the above two cases and is the same as the above, only the last definition counts in determining the controllability type on \textit{var} if this path is used. Also, if \textit{var} is used in \textit{p}, the current controllability type on \textit{var} is used as the operand's controllability type. However, this current controllability type is determined by the previous definition of \textit{var} before this usage. If \textit{var} is not yet defined in \textit{p}, \textit{var}'s controllability type is determined by some other path, as in the multiple uses case.

(2) Not all of the define/use of \textit{var} is in Full Range: All of the above situations assume that for every define/use of \textit{var}, every bit of \textit{var} is involved. This is not true for the general circuits. In general, for each define/use of \textit{var} in \textit{p}, denoted as
\(DU(p, var)\), there is a specific range \(R(DU(p, var))\) associated with this define/use, where \(R(DU(p, var))\) is in \(Range(var)\).

If \(var\) is used in \(DU(p, var)\), the current controllability type for each bit in \(R(DU(p, var))\) should be found. It is possible that part of but not all \(R(DU(p, var))\) is defined by another definition on \(var\), denoted as \(DU_1(p, var)\). As a result, the controllability type of that portion of \(R(DU(p, var))\) is determined by \(DU_1(p, var)\). The remaining part of \(R(DU(p, var))\) is then determined by the definition prior to \(DU_1(p, var)\). This process is continued until the current controllability of every bit in \(R(DU(p, var))\) is determined.

Since it is possible that every bit of \(R(DU(p, var))\) is determined by a different definition, not every bit has the same current controllability type. To be conservative, in BETA, the current controllability type of \(R(DU(p, var))\) as a whole is determined by the least controllability type. For example, as long as one bit in \(R(DU(p, var))\) is of type NCC, the current controllability type of \(R(DU(p, var))\) becomes NCC.

The algorithm shown in Figure 2.9 determines the controllability type for each variable in the circuit. This is accomplished by checking that a variable satisfies all of the criteria stated above.

As a byproduct of the algorithm shown in Figure 2.9, all type CC variables and their corresponding writing sequences are found. All of these variables can then be treated as pseudo-primary inputs while testing the circuit.
input: every variable in the circuit.

output: controllability type (ContType) for each variable.

DetermineCC() {
    for every variable, var, {
        if var is PI, ContType(var) = CC;
        else ContType(var) = NCC;
    }

    change = TRUE;
    while ( change ) {
        change = FALSE;
        for every NCC variable, var,
            for every pi in JPath(var)
                if ( pi satisfies all the criteria ) {
                    ContType(var) = CC;
                    change = TRUE;
                }
    }
}

{;

Figure 2.9 Check CC.
2.3.2 Controllability calculation

Let $\text{var}$ be a type $CC$ variable. Define $WS(\text{var})$ as the set of $JPath(\text{var})$ which can make $\text{var}$ a $CC$ variable. These paths are more controllable than others when justifying $\text{var}$. To justify $\text{var}$, only this type of path is considered. To justify $\text{var}$ through path $p_i$, we need to justify every variable in $JContVar(p_i, \text{var})$. Under the assumption that variables in $JContVar$ can be justified one after the other, the total number of paths needed to justify $\text{var}$ depends on the number of paths needed to justify each variable in $JContVar(p_i, \text{var})$. Therefore, the total number of paths needed to justify a type $CC$ variable, $\text{var}$, will be different if a different $p_i$ is used to justify $\text{var}$. In this section, an algorithm is developed to find the shortest number of path sequences needed to justify $\text{var}$ using only paths from $WS(\text{var})$. This algorithm is not valid for type $NCC$ variables, since they do not have writing sequences.

**Definition:** $CC(\text{var})$ is a measure of the minimum number of paths needed to justify a type $CC$ variable $\text{var}$.

$CC(\text{var})$ is calculated as follows:

\[
\begin{align*}
\text{if (var is a PI) then } CC(\text{var}) &= 0; \\
\text{else} \\
CC(\text{var}) &= \min_{p_i} \{ \sum_{r_j \in JContVar(p_i, \text{var})} [CC(r_j[R_j])] \} + 1,
\end{align*}
\]

where $p_i \in WS(\text{var})$ and $r_j[R_j] \in JContVar(p_i, \text{var})$

The above equation can be interpreted as follows:

- Primary inputs can be controlled directly without traversing any paths. Therefore, their controllability is set to 0.
• If \( p_i \) is used to justify \( \text{var} \), we have to justify all of the \( r_j[R_j] \in J\text{ContVar}(p_i, \text{var}) \) first. Assume that variables are justified in a sequence one after the other. Then, the total number of paths needed to justify all \( r_j[R_j] \) is \( \sum_{r_j}[CC(r_j[R_j])] \).

• After traversing \( \sum_{r_j}[CC(r_j)] \) paths, all inputs in \( p_j \) have been set. But, \( p_j \) has not yet been traversed. Thus, a "+1" is needed to account for this.

• Any path \( p_i \) in \( WS(\text{var}) \) can be used to justify \( \text{var} \). To find the path \( p_j \in WS(\text{var}) \) which requires the minimum number of paths, the \( \text{Min} \) of all \( p_j \) is used.

One such equation is associated with every variable in the circuit. An iterative relaxation algorithm is then used to solve this set of equations.

As long as \( CC(\text{var}) \) is found, the path which makes \( CC(\text{var}) \) minimum is recorded. This is defined to be the best path to justify \( \text{var} \), denoted by \( BJ\text{Path}(\text{var}) \), which is the best candidate for the writing sequence for \( \text{var} \).

### 2.3.3 NCC handling

Not all of the variables in a circuit are of type \( CC \). Thus, it is important to derive some more testability information for \( NCC \) variables. In this section, several approaches are used for this purpose. The first one is to subdivide \( NCC \) into several more types. Another approach is to derive some heuristic for exploring the relative controllability among the \( NCC \) variables. The other approach is to identify the relative controllability among those \( NC \) variables by the reason which makes them \( NC \).
2.3.3.1 More controllability types

Variables of type $NCC$ can be subdivided into the following types: $PCC$, $VCC$ and $NC$.

**Definition:** A variable $var$ is of type $PCC$ *Partially Completely Controllable* if $var$ is not of type $CC$ and a writing sequence exists for symbol $var[R]$, where $R$ is in $Range(var)$.

The multiple define/use issues mentioned in the previous section allow us to identify $PCC$ variables. In $BETA$, each controllable range of a $PCC$ variable is identified, along with the corresponding path sequence which makes this range controllable. This information is helpful for the test generator to control at least part of a variable.

By using $PCC$, the controllable range of a variable is found. On the other hand, a variable may be controllable in the sense that some, but not all, possible values are completely controllable. This leads to the following controllability type:

**Definition:** An $NCC$ variable, $var$, becomes $VCC$ if there exists a $JPath(var)$ such that some, but not all, values on $var$ are completely controllable.

**Definition:** An $NCC$ variable becomes $NC$ if it is not of type $PCC$ or $VCC$.

There are several possible reasons for the existence of $VCC$ variables. One is due to constant assignment. The other case is due to reconvergent fanout. Consider the statement $S_i$: $A = B \ op \ C$. Assume that the temporary controllability of $B$ is $CONSTRAIN$ and that of $C$ is $FREE$. If the $CDF$ of $op$ is 0, $A$ becomes $CONSTRAIN$. This means that $A$ can be set to some, but not all, values freely.

By using the concept of $VCC$ variables, more controllability information can be derived. If the controllable values of $var$ can be determined (for example, due to constant assignment), $BETA$ will keep track of this information. Then, this controllable value
can be treated as the reset value for var. Sometimes, the controllable values is hard to derive (for example, var is of type CONSTRAIN). The path sequence which makes var VCC is very likely to be able to produce the value needed by the test generator, and this sequence is relatively more controllable. Thus, the test generator should try this sequence to justify var first.

### 2.3.4 NC analysis

In BETA, for every p in a variable n’s JPath(n), the reason that p fails to make n CC is also determined. Based on this determination, p is associated to one of the following PathTypes:

- **Rule-1-violated**: There exists at least one NC node in JContVar(p,n).
- **Rule-2-violated**: Violates Criterion 2.
- **Branch-violated**: Branches on p cannot be set up properly.
- **Rule-3-violated**: Violates Criterion 3.

Assume that every NC variable has only one JPath. Then, NC variables can be classified into the following four categories, denoted as NCType according to the PathType of its JPath.

- **Type1**: PathType is Rule-1-violated.
- **Type2**: PathType is Rule-2-violated.
- **Type3**: PathType is Branch-violated.
- **Type4**: PathType is Rule-3-violated.
As a result, \textit{NCType} shows the reason why a variable is \textit{NC}. This is helpful in classifying the relative controllability among these \textit{NC} variables. This issue would be explained later.

In general, however, an \textit{NC} variable may have more than one \textit{JPath}, and each \textit{JPath} may have a different \textit{PathType}. Then, the previous way of defining the \textit{NCType} becomes ambiguous. This leads to the following modified definitions of \textit{NCType}:

- \textbf{Type1}: All PathTypes are Rule-1-violated.
- \textbf{Type2}: All PathTypes are either Rule-1-violated or Rule-2-violated, and at least one PathType is Rule-2-violated.
- \textbf{Type3}: All PathTypes are Rule-1-violated, Rule-2-violated or Branch-violated, and at least one PathType is Branch-violated.
- \textbf{Type4}: There exists at least one Rule-3-violated PathType.

This definition implies that a Rule-3-violated \textit{JPath} has higher priority in determining \textit{NCType} than Branch-violated, Branch-violated is higher than Rule-2-violated, and Rule-2-violated is higher than Rule-1-violated. The following example is used to explain the reason for such priorities. Let an \textit{NC} variable \textit{n} have two \textit{JPaths}, \textit{p}_1 and \textit{p}_2. Path \textit{p}_1 is Rule-1-violated and \textit{p}_2 is Rule-3-violated. It is possibly harder to use \textit{p}_1 to control \textit{n} than to use \textit{p}_2, since a Rule-3-violated path satisfies all \textit{C1}, \textit{C2} and all branches can be properly set up. Thus, \textit{p}_2 should be a better choice for justifying \textit{n}. This motivates that Rule-1-violated paths, which are the least testable, have the lowest priority in determining \textit{NCType}. 

42
As a result, Type\_4 NC variables are relatively more controllable than the other NCType variables, and Type1 is the least controllable one. This information is useful for a high-level test generator when it has to make a decision as to which NC variables to justify.

2.3.4.1 NCC heuristic

Variables of type PCC and VCC are considered more controllable than NC variables. In this section, a heuristic called \( NCCDepth \) is presented to derive the relative controllability among NC variables.

**Definition:** \( NCCDepth \) is a measure of the difficulty in justifying an NC variable.

For an NC variable \( N \), \( NCCDepth(N) \) is defined as follows.

- If \( N \) is of type PCC or VCC, \( NCCDepth(N) \) is 0.

- If there exists a \( JPath(N) \) which makes \( N \) to be NC and violates Criterion C3 only, \( NCCDepth(N) \) is 1.

- If there exists a \( JPath(N) \) which make \( N \) to be NC and violates Criterion C2 but not C1, \( NCCDepth(N) \) is 2.

- Otherwise, set \( NCCDepth(N) \) to be infinite. Then, iteratively solve the following equation:

\[
NCCDepth(N) = \min_p \left\{ \sum_M [NCCDepth(M)] + 1 \right\}
\]

where \( p \) is in \( JPath(N) \) and \( M \) is in \( JContVar(p, N) \) and is of type NCC.

The reason that \( NCCDepth(PCC) \) and \( NCCDepth(VCC) \) are equal to 0 is that PCC and VCC variables are relatively more controllable than other NC variables. Similarly,
among all of the NC variables, those variables having a JPath which violates only Criterion C3 are more controllable than others. Thus, their NCCDepths values are assigned to 1. Variables violating C2 are assigned in a similar fashion. For the other variables, if there are more NCC variables in their JContVar, they are less controllable. Thus, their NCCDepth is computed by the above equation.

It is possible that some NCC variables' NCCDepth remain infinite. For example, if N itself appears in its JContVar for all of its JPaths, NCCDepth(N) would be infinite. In this case, these variables are the least controllable ones.

Note that the path p which satisfies the above equation should be recorded as the best candidate to be used to justify var.

2.3.5 Loop handling

BETA derives testability information out of paths formed from CFG. The existence of a loop complicates the path decomposition process, since we do not know exactly how many times the loop iterates. A typical loop is shown in Figure 2.10. Currently, we consider only the single natural loop.  

Definition: A natural loop [31] in a flow graph is a set of nodes in that graph such that

- All nodes in that set are strongly connected.
- There is only one entry, called header to this set of nodes. The entry is the node through which the outside nodes can reach that set of nodes.

In BETA, CFG is decomposed into paths. For a path with a natural loop, it may be decomposed into an infinite number of paths, which is impractical. To deal with

---

8“Single” means no nested loops.
this problem, note that one basic concept behind CC is that it is value-independent. 

BETA will only identify those CC variables which are independent of the number of loops extended. For example, if a variable can be set to any possible value only after the loop body has been executed exactly 10 times, BETA will fail to identify it as a CC variable. We use the idea of Loop Reduction to decompose the loop into two paths. One does not traverse the loop; the other traverses the loop exactly once. In BETA, only the paths which do not traverse the loop or traverse it exactly once are examined to identify CC variables. This leads to the following definition:

**Definition**: The E(exit) path of a loop is a path whose branch variable is selected so that it exits the loop. The L(loop) path is to select the branch variable such that the loop is traversed exactly once.

Consider Figure 2.10 as an example. The E path in that loop consists of statements \( \{A, B = 10, Out\} \), whereas the L path consists of \( \{A, B \neq 10, C, A, B = 10, Out\} \). In the L path, the branch variable (B in this example) appears exactly twice. Let \( BV_1 \) denote its first appearance and \( BV_2 \) denote the second one.

The E path behaves just like a normal path, except that the branch variable in B (\( BV_1 \), in this case) has to be properly set to avoid the traversal of the possibly uncontrollable loop body. Thus, while deriving the RBranch of every variable in E path, \( BV_1 \) has to be taken into account. To deal with L path, take Figure 2.10 as an example. Assume that node C consists of only one statement \( S_c \) which defines var. Let \( BV_2 \) be var's RBranch in the L path. According to the previous discussion, there is no need to examine the controllability type of \( BV_2 \) while evaluating Criterion C3 of var. However, if \( BV_2 \) is not properly set, the loop body will be traversed one more time, which is not specified in the L path. As a result, the controllability on var derived by the L path is
no longer valid. Thus, while using an \( L \) path to evaluate \( \text{var}'s \) controllability, both \( BV_1 \) and \( BV_2 \) have to be taken into account, no matter whether they are above or below the \( R\text{Branch} \). As a result, more variables have to be satisfied in \( \text{JContVar} \), and both \( BV_1 \) and \( BV_2 \) have to be \( \text{FREE} \) to make \( \text{var CC} \) using the \( L \) path.

The above discussion imposes a stronger criterion on \( \text{var} \) if an \( L \) path is used. One way to alleviate this criterion is to use \textit{backward implication}. For example, in Figure 2.10, let node \( C \) consist of only one statement \( B = B + 1 \) and node \( A \) be null. The \( L \) path is not executable unless \( B \) is initially set to 9. To ensure that the \( L \) path is executable, an extra constraint has to be imposed on \( BV_1 \). Therefore, a backward implication (backtrace) is done on the \( L \) path from \( BV_2 \) back to \( BV_1 \) to find the assignment on \( BV_1 \) such that after executing the loop exactly once, \( BV_2 \) is set to the value which exits the loop. Let this assignment on \( BV_1 \) be \( BV_{\text{exit}} \). If the branch variable \( B \) is \( \text{FREE} \) before the loop and is assigned to \( BV_{\text{exit}} \), then the \( L \) path will be executed. As a result, if \( BV_{\text{exit}} \) can be successfully identified, there is no need to consider \( BV_2 \) anymore. This reduces the size of \( \text{JContVar} \) and the algorithm in Figure 2.5 does not have to check whether \( BV_2 \) is \( \text{FREE} \). If the backtrace fails to find the proper \( BV_{\text{exit}} \), both \( BV_1 \) and \( BV_2 \) have to be \( \text{FREE} \) in order to use \( L \) path as a \( J\text{Path} \) or \( P\text{Path} \) to produce \( \text{CC} \) or \( \text{CO} \) variable.
2.4 Observability

2.4.1 Observability types

Similar to controllability, variables are classified into several types according to their observability.

A reading sequence is a sequence of executable paths that can be used to propagate the contents of a variable to primary outputs.

**Definition:** A variable is of type *Completely Observable (CO)* if that variable has a reading sequence.

Similar to type *PC*, type *PO* is defined as follows:

**Definition:** A variable, *var*, is of type *Partially Observable (PO)* if *var* is not of type *CO* and there exists a reading sequence for symbol *var[R]*, where *R* is in *Range(var)*.

Then, symbol *var[R]* is named observable.

If a variable is not of type *CO* or *PO*, it is denoted as *NCO, Not Completely Observable*.

Assume that path *p* is in *PPath(var)* and is used to propagate the content of *var*.

Consider the following two cases:

- There exists a primary output *out* in *Prop Var(p, var)* such that the content of *var* is observable at *out*:

  If all of the variables in *PCont Var(p, var, out)*\(^9\) are consistent, then a path sequence can be executed to set up all of the variables in *PCont Var(p, var, out)* before the execution of *p* to propagate *var* to *out*. Let this path sequence be denoted as

---

\(^9\) *PCont Var(p, var, out)* is the set of variables needed to be justified if *out* is used to observe *var*.

47
PathSeq_p(p). As a result, the examination of CO on var consists of two phases in this case. The first phase is to find out, a variable which is capable of observing var. Then, PContVar(p, var, out) is checked for consistency. If both phases are satisfied, var is of type CO.

• No such primary output exists:

In this case, the content of var has to be propagated to some register first. Let this register be reg. To use reg to observe var, the following three conditions have to be satisfied:

- All of the input variables needed are consistent. Then, a path sequence, denoted as PathSeq_p(p), is executed before p to set up all of these variables as in the previous case.

- After the execution of p, the content of var will be propagated to reg.

- There exists another path sequence, denoted as PathSeq_a(p), such that PathSeq_a(p) is executed right after p and is able to propagate the content of reg to the primary output.

If all of these conditions are satisfied, the content of var can be observed by some primary outputs by cascading PathSeq_p(p), p and PathSeq_a(p).

In the remainder of this section, we shall show how to determine CO by identifying PathSeq_p(p), p and PathSeq_a(p).

Assume that the content of var has been propagated to reg. Then, another set of variables needs to be justified to propagate the content of reg to the primary outputs. This leads to the following definition:
Definition: $ObContVar(var)$ be the set of variables to be justified such that the content of $var$ can be propagated to the primary outputs.

Several notes for $ObContVar(var)$:

- It is different from $PContVar$ which is associated with a specific path $p$ and an output on that path. This output may be a primary output or some other register. Thus, $PContVar$ can be treated as an input cone if only path $p$ is used. However, the destination of $ObContVar(var)$ must be a primary output. Thus, $ObContVar(var)$ is an input cone for the combined path sequence $p$ and $PathSeq_a(p)$. As a result, $PContVar$ is a subset of $ObContVar$.

- There exists more than one set of $ObContVar(var)$, since $ObContVar(var)$ depends on $p$ and $reg$.

Variables $ObContVar(var)$ can be derived as follows. For simplicity, assume that path $p$ and $reg$ are used to propagate the content of $var$ to the primary output. We assume that the registers do not have the HOLD property. After the execution of $p$, the content of $var$ has been propagated to $reg$. Since the registers do not have the HOLD property, it is not allowed to hold the content $reg$ while justifying $ObContVar(reg)$. As a result, after the execution of $p$, it must be able to propagate the content of $var$ to $reg$ and set up all $ObContVar(reg)$ simultaneously. To propagate $var$ to $reg$, $PContVar(p, var, reg)$ have to be set up. Thus, they are part of $ObContVar(var)$. To justify all $ObContVar(reg)$, the $JContVar(p, ob)$ of each $ob$ where $ob$ is a variable in $ObContVar(reg)$ also have to be included in $ObContVar(var)$. This leads to the following derivation of $ObContVar$:

$$ObContVar(var) = \{\cup_{ob} JContVar(p, ob)\} \cup PContVar(p, var, reg)$$
Then, the criteria for examining CO can be formulated as follows. Variable \( var \) is of type \( CO \) if there is a \( PPath(var), p_i \), and a \( reg \) in \( PropVar(var) \) such that the following criteria are satisfied:

- **O1**: \( reg \) is of type \( CO \).
- **O2**: \( reg \) can observe \( var \) and all \( ObContVar(reg) \) are \( FREE \) after \( p_i \).
- **O3**: All of the variables in the \( ObContVar(var) \) are of type \( CC \).
- **O4**: All of the variables in the \( ObContVar(var) \) are consistent.

Criterion **O1** is to ensure that the content of \( var \) can be propagated to the primary output through \( reg \). Criteria **O3** and **O4** are similar to Criteria **C1** and **C2**, respectively.

Therefore, we examine Criterion **O2**. According to the definition of \( PropVar(p_i, var) \), the original contents of \( var \) will reach \( reg \). However, this does not guarantee that \( var \) can be propagated all the way to \( reg \). For example, let \( St_j, A[0:3] = B[0:3] \& 0001 \) be a statement in \( p_i \) (\& represents bit-wise \( AND \) operation). After executing this statement, only the least significant bit of \( B \) can be “observed” by \( A \), i.e., the least significant bit of \( B \) can be uniquely determined by examining the value on \( A \). Therefore, an algorithm is needed to determine whether \( var \) can be sensitized to any variable in \( PropVar(p_i, var) \).

This leads to the examination of **O2**.

The algorithm (see Figure 2.11) to check Criterion **O2** is similar to the procedure used to check Criterion **C2** of controllability (see \( CheckRegFree() \) in Figure 2.6).

The following is the explanation for the algorithm shown in Figure 2.11:

- To observe one variable, several variables have to be properly justified. Thus, as in procedure \( CheckRegFree() \), a \( ConstrainList \) is used to keep track of the controllability status, \( FREE \), \( CONSTRAIN \) or \( NCC \), of each variable.
input: one variable, var, and one of its PPath, pi.
output: TRUE if pi can make var to be CO.

CheckRegObser(var, pi) {
    ResetConstrinaList();
    ResetObserList();
    use = FALSE;
    define = FALSE;
    for every statement s in pi {
        if (operand is var) {
            if (use) EnterConstrainList(NCC, var);
            else {
                use = TRUE;
                UpdateObserRange(s);
            }
        }
        if (result(s) is var) EnterConstrainList(NCC, var);
        O_type = ResultObserType(s);
        EnterObserType(O_type, result(s));
        C_type = ResultConstrainType(s);
        EnterConstrainType(C_type, result(s));
        next s;
    }
    UpdatePropReg();
    if (exists one observable symbol in PropReg) return(TRUE);
    else return(FALSE);
};

Figure 2.11 Algorithm 5. Check variables' observability.
• There is another list, ObserList, in procedure CheckRegObser() to maintain the information about the ability of each variable to observe var. Every bit of each variable takes two possible values in ObserList, O (observable) or NO (not observable), to indicate the ability to observe the contents of var.

• If var appears in the operand and this is its first appearance, procedure UpdateObserverRange() will enter the range of this operand into ObserList to indicate the observable part of var. However, if var is reused, to simplify the whole procedure, we simply disregard the second usage by assigning NCC to var in the ConstrainList. Then, any further propagation from this second usage of var will be prohibited. Similarly, if var is defined, var no longer stores the original value. Then all of the subsequent uses of var cannot be used to propagate var. Therefore, we also assign NCC to var.

• Procedure ResultObserType() is used here to determine the observability type of each statement result. If there exists any operand whose constraint type is NCC, the result would be NO. Then, according to the ObserList as well as the Observability Degree of Freedom of each operator (defined later), the observability type of the result is determined.

Definition: The Observability Degree of Freedom, ODF, of each operation is defined as the number of constrained operands that it can tolerate and still produce type O results, given that at least one of the operands is of type O and none of the operands are of type NCC.

\(^{10}\text{Note that this } O \text{ means the observability of the contents of var, which is different from } CO.\)
For example, the \textit{ODF} of addition is 1, whereas the \textit{ODF} of multiplication is 0. Then, the observability type of the statement result can be determined similarly to the determination of the controllability type in procedure \textit{ResultConstrainType()}. 

- After examining all statements in the path, those variables in \textit{PropVar}(\textit{pi}, \textit{var}) and of type \textit{O} are marked. They are the variables which are not just \textit{reachable} from \textit{var}, but also can be used to \textit{observe} the contents in \textit{var}. 

Then, for a variable, \textit{var}, if there exists a \textit{PPath}, \textit{pi}, which satisfies all four criteria, \textit{var} is of type \textit{CO}, and all such paths are defined as \textit{PPathCO} (\textit{var}).

\subsection*{2.4.2 Observability calculation}

\textbf{Definition:} For a type \textit{CO} variable \textit{var}, \textit{CO} (\textit{var}) is a measure of the number of paths needed to propagate the contents of \textit{var} to a primary output.

Similar to the \textit{CC} (\textit{var}) computation, the \textit{CO} (\textit{var}) calculation can be formulated as follows:

\begin{align*}
\text{if (var is a PO) then CO(var)=0;} \\
\text{else} \\
\text{CO(var)} &= \text{Min}_{p_i} \{ \text{Min}_{r_d} \{ CO(r_d) + \sum_{r_i} CC(r_i) \} \} + 1, \\
\text{where } p_i &\in \text{PPathCO}(\textit{var}), \text{ } r_d \text{ is of type } \textit{O} \in \text{PropVar}(p_i, \textit{var}) \text{ and } r_j \in \text{ObContVar}(p_i, \textit{var}, r_d)
\end{align*}

The differences between this equation and the one for controllability are the following:

(1) A path, \textit{pi}, is picked from \textit{PPathCO}(\textit{var}) instead of from \textit{JPathC}.

(2) There may exist more than one \textit{var} in \textit{PropVar}(\textit{pi}, \textit{var}). The contents of \textit{var} can be transferred to a primary output through any variables, \textit{r_d}, of type \textit{O} and in
\textit{PropVar}(p_i, var). Since the observability is defined to be the minimum number of paths used, there is a \textit{Min} in the above equation before each \textit{CO}(r_d).

(3) In this equation, \textit{ObContVar}(p_i, var, r_d) is used instead of \textit{JContVar}(p_i, var).

As long as \textit{CO}(var) is found, the path and the \textit{r}_d which makes \textit{CO}(var) minimum are recorded. This can be used when justifying var.

2.5 Results

\textit{BETA} was written in the C programming language. It consists of about 16,000 lines of code. \textit{BETA} was run on several circuits. One of these circuits is shown in Figure 2.12, called \textit{micro}. Its corresponding CFG is shown in Figure 2.13. It is a microprocessor with several instructions, where \textit{DI (Data In)} is the primary input, and \textit{MAR} and \textit{MBR} are primary outputs. The first row of Table 2.1 shows the general information of the CFG of \textit{micro}. According to Table 2.1, it has 13 variables, total 96 bits among these variables, 45 equations, 3 constant variables and 64 flip-flops. This circuit has been synthesized by a behavioral synthesis tool to flatten the circuit into gates. Table 2.2 shows the total number of transistors, gates, D flip-flops, inputs, outputs, stuck-at faults of the circuit and gate-level test generation result using test generator CRIS [32]. To illustrate the effectiveness of the testability guidance provided by \textit{BETA}, a high-level test generator developed by Wu [28] is used to generate tests for this circuit. \textit{BETA} is first executed as a preprocess. There are six data registers in this circuit, \textit{MAR}, \textit{MBR}, \textit{A}, \textit{B}, \textit{HAB} and \textit{PC}. \textit{BETA} identifies all of these registers as being of type \textit{CC} and \textit{CO}, along with their reading and writing sequences. Then, the high-level test generator takes the high-level
structural description (Figure 2.12), CFG (Figure 2.13) and the results generated by BETA to perform test generation.

Table 2.3 shows the high-level test generation run times on a SUN 3/50 workstation with and without BETA where those data registers and ALU are the modules under test. The run time for each module refers to the testing of all of the faults inside that module. The run time for BETA is 8.7 seconds. ¹¹ If BETA was not used, the test generator would randomly pick a path from the JPath (or PPath, the paths which can be used to propagate the content of a variable) to justify (propagate) that variable. As shown in Table 2.3, the test generator obtains a large speedup when BETA is used, because BETA identifies the best justification and propagation sequence for each register. There is no speedup in testing HAB because the random path happened to be the best one.

¹¹If SUN SPARC workstation is used to run BETA, its run time information is shown in Table 2.4.
Figure 2.13 Microprocessor CFG.

Table 2.1 Sample circuit information.

<table>
<thead>
<tr>
<th>name</th>
<th>variables</th>
<th>bits</th>
<th>equations</th>
<th>const</th>
<th>FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>micro</td>
<td>13</td>
<td>96</td>
<td>45</td>
<td>3</td>
<td>64</td>
</tr>
<tr>
<td>circuit1</td>
<td>28</td>
<td>47</td>
<td>28</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>circuit2</td>
<td>46</td>
<td>224</td>
<td>61</td>
<td>13</td>
<td>36</td>
</tr>
<tr>
<td>circuit3</td>
<td>67</td>
<td>336</td>
<td>90</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>circuit4</td>
<td>16</td>
<td>41</td>
<td>23</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>circuit5</td>
<td>8</td>
<td>66</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>circuit6</td>
<td>149</td>
<td>442</td>
<td>194</td>
<td>55</td>
<td>24</td>
</tr>
<tr>
<td>circuit7</td>
<td>29</td>
<td>134</td>
<td>36</td>
<td>11</td>
<td>14</td>
</tr>
</tbody>
</table>
Table 2.2 Gate-level description of the microprocessor example.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of transistor</td>
<td>10396</td>
</tr>
<tr>
<td>Number of gates</td>
<td>1580</td>
</tr>
<tr>
<td>Number of dff</td>
<td>64</td>
</tr>
<tr>
<td>Number of inputs</td>
<td>8</td>
</tr>
<tr>
<td>Number of outputs</td>
<td>24</td>
</tr>
<tr>
<td>Number of faults</td>
<td>4066</td>
</tr>
<tr>
<td>Untestable faults</td>
<td>638</td>
</tr>
<tr>
<td>Detected faults</td>
<td>3232</td>
</tr>
<tr>
<td>Fault Coverage</td>
<td>79.488 %</td>
</tr>
<tr>
<td>Fault Efficiency</td>
<td>95.180 %</td>
</tr>
</tbody>
</table>

On the other hand, even though the ALU does not appear on the CFG, BETA is also helpful for testing the ALU. This is because testing the ALU involves the justification and propagation of variables A and B. Control faults have not been tested by Wu’s test generator. We believe that speedup is also achievable since to activate the control fault and propagate the effects require variable justification and propagation.

We have also applied BETA to seven circuits, named circuit1, circuit2...circuit7, obtained from a behavioral synthesis tool. Information about the sample circuits is shown in Table 2.1. The first column is the name of the circuit. This is followed by the total number of variables and bits in the symbol table of this circuit. The fourth column shows the number of statements performed in this circuit. Some of the variables in the circuit are simply constants. This is shown in the fifth column. The last column in Table 2.1 shows the number of flip-flops in the circuit.

The results of running BETA are shown in Table 2.4. It shows the total number of type CC, PCC, VCC, NC, CO, PCO and NCO variables found in each circuit. The last column shows the run times (in seconds) of BETA if SUN SPACR workstation is used.
Table 2.3 Test generation time for microprocessor example.

<table>
<thead>
<tr>
<th>fault site</th>
<th>total gates</th>
<th>total faults</th>
<th>red. faults</th>
<th>fault cov. (%)</th>
<th>fault efficiency (%)</th>
<th>time (sec) w/o BETA</th>
<th>time (sec) w/ BETA</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAR</td>
<td>112</td>
<td>243</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>15.1</td>
<td>4.2</td>
</tr>
<tr>
<td>MBR</td>
<td>120</td>
<td>260</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>10.0</td>
<td>2.6</td>
</tr>
<tr>
<td>A</td>
<td>120</td>
<td>260</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>112.9</td>
<td>2.9</td>
</tr>
<tr>
<td>B</td>
<td>120</td>
<td>260</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>116.0</td>
<td>2.6</td>
</tr>
<tr>
<td>HAB</td>
<td>40</td>
<td>87</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>2.3</td>
<td>2.2</td>
</tr>
<tr>
<td>PC</td>
<td>256</td>
<td>556</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>13.5</td>
<td>2.8</td>
</tr>
<tr>
<td>ALU</td>
<td>491</td>
<td>1065</td>
<td>149</td>
<td>86.0</td>
<td>100</td>
<td>80.1</td>
<td>7.2</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>349.9</td>
<td>24.5</td>
</tr>
</tbody>
</table>

Table 2.4 The results of running BETA on the sample circuits.

<table>
<thead>
<tr>
<th>name</th>
<th>nodes</th>
<th>const</th>
<th>CC</th>
<th>PCC</th>
<th>VCC</th>
<th>NC</th>
<th>CO</th>
<th>PCO</th>
<th>NCO</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>micro</td>
<td>13</td>
<td>3</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>3</td>
<td>0.7</td>
</tr>
<tr>
<td>circuit1</td>
<td>28</td>
<td>8</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>0</td>
<td>11</td>
<td>0.6</td>
</tr>
<tr>
<td>circuit2</td>
<td>46</td>
<td>13</td>
<td>21</td>
<td>0</td>
<td>8</td>
<td>4</td>
<td>22</td>
<td>2</td>
<td>9</td>
<td>4.8</td>
</tr>
<tr>
<td>circuit3</td>
<td>67</td>
<td>11</td>
<td>56</td>
<td>0</td>
<td>0</td>
<td>21</td>
<td>2</td>
<td>33</td>
<td>24.7</td>
<td></td>
</tr>
<tr>
<td>circuit4</td>
<td>16</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0.1</td>
</tr>
<tr>
<td>circuit5</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>circuit6</td>
<td>149</td>
<td>55</td>
<td>18</td>
<td>0</td>
<td>61</td>
<td>15</td>
<td>26</td>
<td>0</td>
<td>68</td>
<td>233.7</td>
</tr>
<tr>
<td>circuit7</td>
<td>29</td>
<td>11</td>
<td>11</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>13</td>
<td>1.1</td>
</tr>
</tbody>
</table>

58
CHAPTER 3

BEHAVIORAL SYNTHESIS FOR TESTABILITY

As shown in Figure 1.6, our synthesis for testability approach is modeled as the Testability Modifier. Its major operation is shown in Figure 1.7. First, BETA identifies the HTD areas and diagnoses causes. Second, this information is used in a test point selection process. The designer can then decide to use a traditional approach (test point insertion or scan design) or our proposed Test Statement Insertion (TSI). In this chapter, our selection process is first presented followed by TSI.

3.1 The Selection Process

If the measure of the difficulty of testing each HTD area can be found, then the most difficult HTD area seems to be the best candidate for a test point. However, this is not generally true, because it is possible that by selecting a less difficult HTD area, other more difficult HTD areas become testable.

Example: A is a Type4 NCC variable, and A is in JContVar of B which makes B a Type1 NCC variable. In general, B is less controllable than A, since in order to control B, A has to be controlled first. If only one test point is allowed in this circuit to modify the controllability, then inserting it at A would be a better choice than at B because B can be controlled through a more controllable A. □
This implies that the best test point candidate may not be the least testable one, but the one with the most influence on other NCC variables. As a result, NCCDepth, the heuristic used to differentiate the relative controllability among NCC variables, is not a good heuristic for selecting test points. Therefore, another heuristic needs to be developed to measure the influence of selecting one NCC variable on the other NCC variables. The heuristic that we use to measure the influence is based on the total number of NCC variables reduced after one NCC variable is selected. As a result, if the number of test points selected does not reach the limit on hardware overhead or the modified circuit does not fulfill the testing (fault coverage or fault efficiency) requirement, the selection process continues until all of the NCC variables are removed.

3.1.1 Complexity

The goal of our selection procedure is to select the minimum number of test points such that no NCC remains in the circuit, unless the hardware constrain or testability requirement is reached. To explore the complexity of this problem, consider the following special case. Let the NCCType of all NCC variables be Type1. Let there be n NCC nodes denoted by N₁, N₂...Nₙ. For simplicity and without loss of generality, each NCC node, Nᵢ, has only one JPath, denoted as Pᵢ. Let the set of NCC nodes in JContVar(Pᵢ, Nᵢ) be NCC(Pᵢ, Nᵢ). Then, a directed graph, DG(N,E) can be formed. The nodes in N are the n NCC nodes in the circuit. There is an edge e(i,j) in E if node Nᵢ is in NCC(Pⱼ, Nⱼ).

Claim 1: There is no source in DG(N,E).
**Proof:** If there is a *source*, $N_s$, in $DG(N,E)$, there are no incoming edges to $N_s$, according to the definition of *source*. However, this violates the fact that $N_s$ is a Rule-1-violated NCC node. Therefore, there is no *source* in $DG(N,E)$. □

As a result of Claim 1, there must exist *cycles* [31] in $DG(N,E)$. On the other hand, in this example, if we can modify the circuit such that the graph is acyclic, then there will be no NCC nodes left in the circuit.

**Claim 2:** The minimum selection problem on the circuit with only Rule-1-violated nodes is equivalent to the feedback vertex set problem [24], which is an NP-complete problem.

**Proof:** The feedback vertex set problem is to find a subset $N' \subseteq N$ in a directed graph $G(N, E)$ such that $|N'| \leq K$, where $K$ is a positive integer, and every directed *cycle* in $G$ includes at least one vertex from $N'$. Given the previously defined directed graph, $DG(N,E)$, if a minimum number of nodes $N'$ can be found out of $N$ by the feedback vertex set problem, then by making these $N'$ nodes controllable the final $DG(N,E)$ will consist of no *cycles*, and there will be no NCC nodes. Therefore, these two problems are equivalent. □

**Theorem:** The minimum selection problem is an NP-complete problem.

**Proof:** It can be shown that the minimum selection problem is an NP problem. Then, according to Claims 1 and 2, we showed that a special case of the minimum selection problem, i.e., the one with only Rule-1-violated nodes, is equivalent to an NP-complete problem. Therefore, the general selection problem is NP-complete. □

### 3.1.2 Heuristic approach

Since the optimal selection among those NCC nodes is at least NP-complete, we propose a heuristic method to solve this problem.
Our heuristic approach can be derived as follows:

Associate with each NCC variable $N$ an effectiveness value, denoted as $EFF(N)$. $EFF(N)$ is defined as follows:

$$EFF(N) = \left( \sum_{n_t \in NCC_t} \text{BitSize}(n_t) - \sum_{n_m \in NCC_m(N)} \text{BitSize}(n_m) \right) / \text{BitSize}(N)$$

where $NCC_t$ denotes the set of original NCC variables, $NCC_m(N)$ denotes the set of NCC variables after variable $N$ has been inserted as a test point, and BitSize is the number of bits in variable $N$.

Heuristic $EFF(N)$ is a measure of the effectiveness and cost ratio if $N$ is selected. The nominator part of $EFF(N)$ measures the testability improvement associated with $N$. If $N$ is selected as a test point, every bit of $N$ has to be modified. As a result, the cost to select $N$ can be modeled by $\text{BitSize}(N)$. Then, the best candidate to be selected is the one with the largest $EFF$ value. The selection process is repeated until the limit on the number of variables selected is reached. \(^1\)

As can be seen from the above equation, the effect of modifying one variable is modeled by the number of NCC bits decreased. Therefore, $EFF$ is a prediction of testability improvement by inserting each variable into a scan chain (or test point).

\(^1\)Note that after each selection, total NCC is changed. This should be taken into account while deriving the next best candidate.
3.2 Test Statement Insertion

3.2.1 Methodology

Test Statement Insertion (TSI) is a technique that modifies the circuit to make it more testable. It modifies the circuit's CFG instead of its structure diagram. The basic idea of TSI is to bypass the original statement, which produces an NCC result, during the test mode by inserting a test statement which defines the same result as that of the original statement under normal mode of operation, but is able to make it CC under test mode. An extra control input, denoted as $T_{in}$, is needed to distinguish between normal mode and test mode. During the normal operation, $T_{in}$ is OFF and the original statement is executed. In the test mode, $T_{in}$ is set ON, the original statement is bypassed and the test statement is executed. Consider the example shown in Figure 3.1. The left-hand side of Figure 3.1 shows that the original CFG, where there is a path $p_i$ with a statement $s_k$ on it. Let $s_k$ produce an NCC variable $n_j$, and assume that $n_j$ has been selected as a test point. The right-hand side of Figure 3.1 shows the CFG modified by TSI. In the modified CFG, a branch is inserted with $T_{in}$ as the branch variable, and a new statement $s_{new}$ is inserted. The variable $n_j$ is NCC in the original CFG because the operation performed by $s_k$, i.e., $A \text{ op } B$, fails to produce a CC result. To make $n_j$ CC, a CC variable $n_{in}$ is assigned to $n_j$ in $s_{new}$. The corresponding change in the structure diagram is shown in Figure 3.2. In the original circuit, $n_j$ is produced from variables $A$ and $B$ through operation $\text{op}$. Another type CC variable $n_{in}$ exists somewhere else in the circuit. In the modified circuit, an extra fanout from $n_{in}$ along with the output of operation $A \text{ op } B$ are connected to a multiplexer $MUX$, which is controlled by an extra primary input $T_{in}$. The output of $MUX$ is assigned to $n_j$. 

63
Figure 3.1 Test Statement Insertion.

Figure 3.2 The effect of TSI on structure diagram.
To successfully apply TSI, the following issues have to be considered in selecting a variable $n_{in}$:

- Primary inputs are good candidates. However, a lot of primary input fanouts should be avoided.

- Try to reduce the usage of the same CC variable as $n_{in}$ for several NCC variables.

- It is better not to use $n_{in}$ again in subsequent statements along $p_i$ to avoid bus reuse which may produce extra NCC variables.

- Each variable has its bit range. The bit range of $n_{in}$ should be greater than or equal to that of $n_j$. Otherwise, more than one $n_{in}$ should be used to fill the range of $n_j$.

- Variable $n_{in}$ should be physically close to $n_j^2$ to reduce extra routing.

Given a test point NCC to be modified, Figure 3.3 shows the TSI algorithm. Procedure DetermineCandidate finds the possible candidates for this test point and returns a CandidateList. According to the previous discussion, every candidate must be of type CC and should create no loops after TSI. Also, the range in this candidate should be greater than that of NCC. As a result, several CC variables may need to be combined to modify NCC. Before examining each candidate, the original CFG is saved. Then, the modified CFG is generated using the method shown in Figure 3.1. Then, for each candidate in the CandidateList, its ability on overall testability improvement is examined. This heuristic is similar to the EFF heuristic used in the test point selection algorithm. The only difference is that in EFF modification extra primary inputs are assumed, whereas, in

\[\text{From the CFG, we have no idea how physically close between two variables. However, some simple heuristic can be used to measure this distance.}\]
input: CFG and NCC, the selected test point.
output: the best candidate to modify NCC using TSI.

TSI()
{
    CandidateList = DetermineCandidate();
    for each Candidate in CandidateList
    {
        SaveOriginal CFG();
        ModifyCFG(Candidate);
        heuristic = TSI.DetermineCC(Candidate);
        if (heuristic is the best ever)
            BestCandidate = Candidate;
        RestoreCFG();
        next Candidate();
    }
    report BestCandidate for NCC;
}

Figure 3.3 TSI algorithm.
Figure 3.3, a CandidateList (a list of internal variables) is used to modify NCC. As in deriving an EFF value, the heuristic value in Figure 3.3 requires the use of deriving a CC procedure similar to the one used in BETA. Finally, the best candidate is reported.

3.2.2 Comparison

TSI offers the following advantages:

- **It can be adopted in behavioral or high-level synthesis:** TSI can be applied directly on the intermediate format of behavioral-level synthesis. Therefore, this type of testability enhancement technique can be done in the early design phase before the circuit's structural diagram is generated. It can also be used to modify the structural diagram directly, as shown in Figure 3.2.

- **Fewer extra primary inputs are needed:** In the regular test point insertion procedure, many extra primary input pins are required. Even for the scan design, at least three pins, TCK, ScanIn, ScanOut, are required. In TSI, however, only one extra pin, Tm, is needed, since controllability enhancement is done by internal variables.

- **Lower test application time compared to a scan-based design:** One major drawback for a scan design is the longer test application time associated with it. This makes at-speed testing impossible. In TSI, controlling and observing an internal variable can be done under the normal clock rate of the circuit.

- **Lower area overhead than a scan-based design:** Let variable nj in Figure 3.2 have m bits. Then, in the modified circuit, both of the inputs to the MUX have m bits. Thus, the MUX can be decomposed into m multiplexers, each with two
single-bit inputs. Each multiplexer can be implemented by two pass transistors with \( T_{in} \) as the control, and, as a result, only \( 2m \) transistors are needed to modify variable \( n_j \) by TSI. This is much less than for the traditional LSSD implementation. Test Statement Insertion is used to enhance controllability. It can also be used to enhance the observability in a similar fashion. If variable \( n_j \) is also NO (non-observable), an extra \( 2m \) transistors are needed to make it observable. In this case, an extra pin, Observability Test Mode Selector, is needed for observability enhancement. Note that a variable may be CC but not CO, or CO but not CC. Therefore, two kinds of test mode selector, controllability and observability, can be used independently to select different variables in the circuit depending on the characteristics of that variable. As a result, even though a variable is both NCC and NO, a total of \( 4m \) transistors is required to make it both controllable and observable. The area overhead is still less than that of an LSSD design. As shown in Section 3.3, low area overhead was achieved by running experiments on several sample circuits from the synthesis tool.

The following issues have to be addressed in using TSI:

- For a large circuit, \( T_{in} \) may drive many pass transistors in the circuit depending on the original controllability of the circuit, and may need higher driving capability than other regular inputs.

- Due to signal degradation caused by a pass transistor, a careful examination of each gate’s input should be done to ensure that it has enough driving capability.
Table 3.1 NCC selection result.

<table>
<thead>
<tr>
<th>name</th>
<th>NCC</th>
<th>Select</th>
<th>NCC remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuit1</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>circuit2</td>
<td>12</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>circuit3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>circuit4</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>circuit5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>circuit6</td>
<td>76</td>
<td>6 (31)</td>
<td>29 (0)</td>
</tr>
<tr>
<td>circuit7</td>
<td>7</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

• There would be some testability penalty associated with TSI if \( n_{in} \) is not from extra primary inputs, as in test point insertion. Some experiments have been run to evaluate this penalty and the results are shown in the next chapter.

3.3 Results

While deriving CC variables, BETA diagnoses the reason for a variable being of type NCC. Based on this analysis, the proposed selection procedure selects a set of NCC variables as test points such that all NCC variables are eliminated. The total number of test points selected is shown in Table 3.1.

The second column in Table 3.1 gives the total number of NCC variables in the circuit, while the third column gives the number of NCC variables that were selected.\(^3\)

Gate-level test generator CRIS [32] was then run on these circuits. The result is shown in Table 3.2. The fourth column of Table 3.2 shows the untestable faults. \(^4\) The fifth

---

\(^3\) A note about circuit circuit6. It has more NCC variables than the other circuits, and, by selecting 6 out of 76 variables, the remaining NCC variables in circuit6 were reduced to 29. To make the remaining 29 variables CC, 25 variables had to be selected.

\(^4\) The untestable faults are due to the sequential behavior of each circuit. The combinational parts of the circuits are 100% testable.
and sixth columns of Table 3.2 show the fault coverage and fault efficiency, respectively. Note that according to Table 2.4, circuit3 and circuit5 are more controllable than the others in the sense that they have more CC variables. This observation matches the test generation result in Table 3.2 in which both circuit3 and circuit5 are 100% testable. The least controllable circuit predicted by BETA, as shown in Table 2.4, is circuit6. This circuit is also found to be least testable as in Table 3.2.

According to Table 3.2, micro, circuit2, circuit3, circuit4 and circuit5 are already testable. Thus, the selection process is applied to the remaining circuits to select test points. After the selection, test generation is run. This gives the results shown in Table 3.3.

Table 3.2 Test generation result for the original circuits.

<table>
<thead>
<tr>
<th>circuit</th>
<th>injected</th>
<th>detected</th>
<th>untestable</th>
<th>f cov. (%)</th>
<th>f eff. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>micro</td>
<td>4066</td>
<td>3232</td>
<td>638</td>
<td>79.488</td>
<td>95.180</td>
</tr>
<tr>
<td>circuit1</td>
<td>60</td>
<td>36</td>
<td>0</td>
<td>60.000</td>
<td>60.000</td>
</tr>
<tr>
<td>circuit2</td>
<td>737</td>
<td>593</td>
<td>144</td>
<td>80.461</td>
<td>100</td>
</tr>
<tr>
<td>circuit3</td>
<td>839</td>
<td>468</td>
<td>362</td>
<td>56.386</td>
<td>100</td>
</tr>
<tr>
<td>circuit4</td>
<td>105</td>
<td>76</td>
<td>25</td>
<td>72.381</td>
<td>96.190</td>
</tr>
<tr>
<td>circuit5</td>
<td>795</td>
<td>741</td>
<td>54</td>
<td>93.208</td>
<td>100</td>
</tr>
<tr>
<td>circuit6</td>
<td>1480</td>
<td>537</td>
<td>45</td>
<td>36.284</td>
<td>39.324</td>
</tr>
<tr>
<td>circuit7</td>
<td>462</td>
<td>281</td>
<td>150</td>
<td>60.823</td>
<td>93.2</td>
</tr>
</tbody>
</table>

Table 3.3 Test generation result for the modified circuits.

<table>
<thead>
<tr>
<th>circuit</th>
<th>total</th>
<th>mod.</th>
<th>time</th>
<th>f cov.(%)</th>
<th>f eff. (%)</th>
<th>TSI f eff. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuit1</td>
<td>28</td>
<td>1</td>
<td>0.3 sec</td>
<td>100.000</td>
<td>100.000</td>
<td>100.000</td>
</tr>
<tr>
<td>circuit6</td>
<td>149</td>
<td>4</td>
<td>7.89 hr</td>
<td>77.363</td>
<td>95.055</td>
<td>89.728</td>
</tr>
<tr>
<td>circuit7</td>
<td>29</td>
<td>1</td>
<td>1.7 sec</td>
<td>68.872</td>
<td>98.054</td>
<td>97.531</td>
</tr>
</tbody>
</table>
Table 3.4 Comparison of 5 test point candidates in circuit circuit6.

<table>
<thead>
<tr>
<th>modify</th>
<th>EFF values</th>
<th>f cov. (%)</th>
<th>f eff. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>7</td>
<td>35.676</td>
<td>47.082</td>
</tr>
<tr>
<td>node2</td>
<td>7.5</td>
<td>46.410</td>
<td>57.637</td>
</tr>
<tr>
<td>node3</td>
<td>1</td>
<td>38.055</td>
<td>47.785</td>
</tr>
<tr>
<td>node4</td>
<td>8</td>
<td>41.646</td>
<td>52.532</td>
</tr>
<tr>
<td>node5</td>
<td>5</td>
<td>35.027</td>
<td>47.727</td>
</tr>
</tbody>
</table>

The second column of Table 3.3 shows the total number of variables in each circuit. The third column shows the number of controllability test points inserted in this experiment. This is followed by the run time (on SUN SPARC workstation) to select these test points. The fifth and sixth columns show the fault efficiency and fault coverage if these test points are modified using Test Point Insertion. The last column shows the fault efficiency if Test Statement Insertion is used to modify these test points.

To show that the selection process selects good test points, we performed the following experiment on the least testable circuit, circuit6. The NCC variables in circuit6 with the 5 highest EFF values were found first. The test generation was then run on 5 copies of the modified circuit6, one copy for each selected NCC variable as a test point. The results are shown in Table 3.4. The best candidate variable for test point predicted by the selection process is node4 followed by node2, whereas the test generation shows that node2 is the best followed by node4. Both nodes are better than the other nodes in the sense of testability improvement.

We also run another circuit, circuit8, with 556 statements, approximately 10000 transistors and 71 flip-flops. There are 145 NCC nodes in that circuit. The 6 most EFF value NCC nodes are selected, and test generation is run on the 6 different copies of circuit8. The result is shown in Table 3.5, where the first row shows the test generation
Table 3.5 Comparison for 6 test point candidates in circuit circuit8.

<table>
<thead>
<tr>
<th>modify</th>
<th>EFF values</th>
<th>f cov. (%)</th>
<th>f eff. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>x</td>
<td>60.049</td>
<td>92.334</td>
</tr>
<tr>
<td>node1</td>
<td>2.833</td>
<td>68.072</td>
<td>100.000</td>
</tr>
<tr>
<td>node2</td>
<td>1</td>
<td>60.456</td>
<td>91.298</td>
</tr>
<tr>
<td>node3</td>
<td>1</td>
<td>60.957</td>
<td>91.800</td>
</tr>
<tr>
<td>node4</td>
<td>1</td>
<td>61.948</td>
<td>93.025</td>
</tr>
<tr>
<td>node5</td>
<td>1</td>
<td>62.404</td>
<td>93.479</td>
</tr>
<tr>
<td>node6</td>
<td>1</td>
<td>60.891</td>
<td>92.266</td>
</tr>
</tbody>
</table>

result for the original circuit. As shown in Table 3.5, the best test point candidate predicted by the selection procedure is node1, which is the node that gives the best test generation result.

Finally, we show the amount of area overhead due to TSI. Assume that in the worst case, no test generation is available (this is possible if we evaluate the circuit testability at high level), and all the NCC variables found in Table 3.1 are modified using TSI. The area overhead was computed by first counting the total number of transistors used in the original circuit using the cell library of the synthesis tool. Then, if one node with bit width $m$ is selected from the circuit, $2m$ transistors are added to the modified circuit. Table 3.6 shows the area overhead according to the transistor count.

The second column of Table 3.6 shows the total number of transistors in the original circuit. The third column is the number of NCC nodes selected. Note that each node has a different bit range. The fourth column gives the total number of transistors included by TSI to modify those selected nodes. The overhead is given in the fifth column. Only a low percentage of overhead was needed to enhance the overall controllability. As indicated in the last row, the average area overhead for these seven circuits was around 2.599 %. Note that this result assumes that all of the NCC variables found in Table 3.1 were modified.
Table 3.6 Area overhead analysis in transistor count.

<table>
<thead>
<tr>
<th>name</th>
<th>total Trans.</th>
<th>NCC selected</th>
<th>extra Trans.</th>
<th>overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuit1</td>
<td>1126</td>
<td>2</td>
<td>18</td>
<td>1.599</td>
</tr>
<tr>
<td>circuit2</td>
<td>2110</td>
<td>5</td>
<td>94</td>
<td>4.455</td>
</tr>
<tr>
<td>circuit3</td>
<td>2228</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>circuit4</td>
<td>284</td>
<td>1</td>
<td>6</td>
<td>2.113</td>
</tr>
<tr>
<td>circuit5</td>
<td>1108</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>circuit6</td>
<td>3868</td>
<td>31</td>
<td>160</td>
<td>4.137</td>
</tr>
<tr>
<td>circuit7</td>
<td>1126</td>
<td>2</td>
<td>30</td>
<td>2.664</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
<td></td>
<td>2.599</td>
</tr>
</tbody>
</table>

As shown in Table 3.3, the actual number of test points needed may be far less than this, and, as a result, the actual area overhead will be even lower.
CHAPTER 4

A PROBABILISTIC APPROACH FOR EVALUATION AND SYNTHESIS FOR TESTABILITY

4.1 Introduction

As the complexity of VLSI circuits increases, automatic test pattern generation (ATPG) becomes a very time-consuming process. Design for testability by the use of built-in self-test [33] becomes an attractive alternative. This motivates research on testing through random or pseudorandom test patterns [34]. However, due to the fact that the test patterns are not exhaustive, random test generation requires test quality verification. Some research has been done on random testability analysis [35] and signal/detection probability calculation [6, 36, 37, 38, 39].

In [35], a random testability algorithm called the Cutting Algorithm is presented. It computes signal probability for combinational circuit and models detection probability by signal probability. STAFAN [38] uses sampling techniques to compute the detection probability of combinational and sequential circuits. PREDICT [6] uses the idea of a super gate to compute exact signal probability. Chakravarty extended the concept of a super gate to compute the exact detection probability for combinational circuits in [40].
In this chapter, a probabilistic testability measure and its corresponding synthesis for testability approach are presented. First, a behavioral-level probabilistic testability analyzer is presented, called BEPTA, which computes controllability and observability for sequential and for combinational circuits based on probabilistic analysis of the circuits. The inputs to this analyzer are in the form of CFG. The second part of this chapter describes a synthesis for testability approach based on the analyzer’s result. In this approach, first, a set of test point candidates are selected out of the less testable variables identified by the testability analyzer. Then, the designer can choose either to use Test Point Insertion [9, 10, 14, 15], Partial Scan [16, 17, 18] or Test Statement Insertion (TSI) to modify the test points.

This chapter is organized as follows. The probabilistic controllability analysis is presented in Section 4.2. Section 4.3 presents the observability analysis. In Section 4.4, an approach for testability improvement is presented. Some results are shown in Section 4.5.

4.2 Probabilistic Controllability Evaluation

4.2.1 Derivation of PCI(N)

The concept of CC is based on the deterministic analysis of the circuit testability, whereas, the probabilistic analysis leads to the definition of Completely Probabilistic Controllability (CPC).

Definition: Given that every primary input pattern has an equal probability to occur, a variable \( N \) is said to be Completely Probabilistically Controllable (CPC) if every possible value on \( N \) has equal probability to occur.
Under the random testing environment, if a variable is \( CPC \), it would be easier to activate any faulty effect on this variable. Thus, it would be important to find out whether a variable in the circuit is \( CPC \) or not, and if not, how close it is to \( CPC \). This leads to the following definition of \textit{Probabilistic Controllability}.

**Definition:** The \textit{Probabilistic Controllability} of a variable \( N \) is defined as the number of input patterns which are capable of making \( N \) \( CPC \), divided by the total number of input patterns, given that every input pattern has an equal probability to occur.

The following example is used to illustrate how to compute \textit{Probabilistic Controllability}.

**Example:** Let a CFG have only two statements, \( C[0:1]= A[0:1] \ ADD \ B[0:1] \) followed by \( D[0:0]= C[0:1] \ GE \ 3 \). Variables \( A \) and \( B \) are primary inputs, and all of the variables are two bits wide, except \( D \). Operation \( ADD \) refers to addition without carry and \( GE \) means \textit{greater than or equal to}. Variable \( D \) becomes 1 if \( C \) is greater than or equal to 3. Otherwise, \( D \) becomes 0. Table 4.1 shows the value of each variable under sixteen different input patterns. According to Table 4.1, \( C \) is \( CPC \), since every possible value of \( C \) is equally likely to occur. However, only four input patterns make \( D \) to be 1. Thus, only eight out of sixteen input patterns are capable of making \( D \) \( CPC \) (four patterns make it 1 and the other four make it 0). Then, the \textit{Probabilistic Controllability} of \( D \) becomes 0.5. □

However, the above approach of computing each variable's \textit{Probabilistic Controllability} is impractical, since it relies on the exhaustive examination of input combinations. Thus, in this section, a heuristic approach called \textit{PCI} is proposed to measure how \textit{probabilistically controllable} a variable is.
Table 4.1  The value on each variable in the example under all possible input combinations.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Definition: The Probabilistic Controllability Index, \( PCI(N) \), is a measure of the Probabilistic Controllability of \( N \).

While deriving \( PCI(N) \), according to the definition of \( PCI \), we do not treat every bit of \( N \) separately. However, due to the frequent variable merge and split, it is possible that not every bit of \( N \) has the same Probabilistic Controllability. As a result, \( PCI(N[b]) \) is actually computed, where \( b \) is a bit in \( Range(N) \). In the remaining section, \( PCI(N) \) refers to the average value of \( PCI(N[b]) \) among all bits \( b \) in \( Range(N) \).

The circuit’s CFG can be decomposed into a set of paths. Different sets of statements may be executed by different paths. As a result, \( PCI(N) \) depends on whether each path can make \( N \) probabilistic controllable. The general approach of deriving \( PCI(N) \) is outlined below. We first determine the Probabilistic Controllability of \( N \) in each path. This leads to the derivation of \( PPCI(N) \), defined later and derived in Section 4.2.2. A path is composed of statements. Thus, how probabilistically controllable a statement’s result is becomes the next topic to examine. This leads to the definition of \( SPCI \), which is defined in Section 4.2.3 and derived in Section 4.2.4.

Definition: Under the random assumption, Path Probabilistic Controllability Index, \( PPCI(p, N) \), is a measure of how probabilistically controllable \( N \) is if path \( p \) is executed.

As in \( PCI \), \( PPCI(p, N[b]) \) is computed, and \( PPCI(p, N) \) refers to the average value among \( PPCI(p, N[b]) \).

Let \( PathSet \) be the set of paths of the given CFG. Then, \( PCI(N[b]) \) should be determined by examining all \( PPCI(p, N[b]) \), where \( p \) is in \( PathSet \). In the ideal case, \( PCI(N[b]) \) is the weighted sum of all \( PPCI(p, N[b]) \), weighted by path probability, the probability that a path is executed under the assumption that inputs are purely random. Due to the fact that the content of the registers is initially unknown, it is hard to derive
the exact path probability. As a result, we assume that each path is equally likely to be executed as an approximation. Thus, \( PCI(N[b]) \) can be determined as follows.

\[
PCI(N[b]) = \frac{\sum_{p} PPCI(p, N[b])}{\#PathSet}
\]

where \( p \) is in the \( PathSet \), and \( \#PathSet \) denotes the total number of paths in the \( PathSet \).

### 4.2.2 Derivation of \( PPCI(p, N[b]) \)

To determine \( PPCI(p, N[b]) \), every statement in path \( p \) should be examined one after the other according to its order in \( p \). The Probabilistic Controllability of the output of each statement depends on the operands and the operation performed in this statement. This motivates the following definition.

**Definition:** Given a statement \( S_i : N = A \, op \, B \) in path \( p \), \( SPCI(p, S_i, N) \) is the measure of how probabilistically controllable \( N \) is if statement \( S_i \) is executed.

Similar to \( PCI \) and \( PPCI \), every \( SPCI(p, S_i, N[b]) \), where \( N[b] \) is defined in \( S_i \), is computed, if possible. Let \( DefineSet(p, N) \) denote the set of statements in path \( p \) which defines \( N \).\(^1\) As in deriving \( PCI(N[b]) \), \( PPCI(p, N[b]) \) can be computed using \( SPCI(p, S_i, N[b]) \) where \( S_i \) is in \( DefineSet(p, N) \).

There are the following three cases to consider.

- If \( DefineSet(p, N) \) is empty, \( p \) has no ability to produce \( N \) at all. In this case, \( PPCI(p, N) = 0 \).\(^2\)

\(^1\)It is not necessary that the definition of \( N \) is in full range.
\(^2\)This derivation makes sense only if \( N \) is either a combinational variable or a sequential variable without the \( HOLD \) property. This implicit assumption is applicable to the remaining sections. On the other hand, if \( N \) is a sequential variable with the \( HOLD \) property, this path should not be considered when computing \( PCI(N) \).
If \( \text{DefineSet}(p, N) \) has exactly one statement, e.g., \( S_i: N = A \text{ op } B \), consider the following two cases:

1. \text{\textit{N is defined in Full Range:}} This is the simpler case in which every bit of \( N \) is covered by \( S_i \). Then, \( \text{PPCI}(p, N[b]) \) would be equal to \( \text{SPCI}(p, S_i, N[b]) \).

2. \text{\textit{N is defined in Partial Range:}} This means that only part of \( N \) is defined in \( S_i \). As a result, \( \text{PPCI}(p, N[b]) \) becomes \( \text{SPCI}(p, S_i, N[b]) \), if \( N[b] \) is defined in \( S_i \). Otherwise, \( \text{PPCI}(p, N[b]) \) is set to 0.

If \( \text{DefineSet}(p, N) \) has more than one statements, e.g., \( S_1, S_2...S_k \) according to the order of appearance in \( p \), and \( S_k \) refers to the last one, then consider the following situations:

1. \text{\textit{If \( S_k \) defines Full Range on \( N \), then for every \( b \in \text{RangeN} \) \( \text{PPCI}(p, N[b]) \) equals \( \text{SPCI}(p, S_k, N[b]) \). This is because the last definition erases all of the previous definitions on \( N[b] \). Thus, \( \text{PPCI}(p, N[b]) \) is completely determined by \( \text{SPCI}(p, S_k, N[b]) \).}}

2. \text{\textit{If \( S_k \) defines only Partial Range on \( N \), then previous statements, \( S_{k-1}, S_{k-2}...S_1 \) begin to influence \( \text{PPCI}(p, N) \). The way we handle this condition is similar to the case in which there is only one statement with partial range. We can associate a \text{BITSPCI} value to each bit. Initially, all \text{BITSPCI} are set to 0. Starting from \( S_k \), for every bit \( b \) covered by \( S_k \), its \text{BITSPCI} value equal to \( \text{SPCI}(p, S_k, N[b]) \). Then, check \( S_{k-1} \). For every bit \( b \) covered by \( S_{k-1} \) and not covered by \( S_k \), assign \( \text{SPCI}(p, S_{k-1}, N[b]) \) to its \text{BITSPCI}. This process is continued until all bits are covered or no \text{DefineSet}(p, N) \) is left.}
4.2.3 Derivation of $SPCI(p, S_i, N[b])$

We have shown how to compute $PPCI(p, N[b])$ out of $SPCI(p, S_i, N[b])$. In this section, we shall illustrate the derivation of $SPCI(p, S_i, N[b])$.

Assume that $S_i$ is $N[N_i : N_h] = A[A_i : A_h] \text{ op } B[B_i : B_h]$, where $\text{op}$ is an operation, and $[N_i, N_h], [A_i, A_h]$ and $[B_i, B_h]$ specify the bit ranges used in $N, A$ and $B$, respectively.

There are the following three factors that determine $SPCI(p, S_i, N)$:

- $CPPCI(p, S_i, A)$ and $CPPCI(p, S_i, B)$.
- Data dependency between $A$ and $B$.
- Operation used in $S_i$, i.e. $\text{op}$.

**Definition:** Current $PPCI$, $CPPCI(p, S_i, K[b])$, is $PPCI(p, K[b])$ with only statements above $S_i$ are counted while deriving $CPPCI(p, S_i, K[b])$. If $K[b]$ is not yet defined, $PCI(K[b])$ is assumed to be $CPPCI(p, S_i, K[b])$.

Current $PPCI$ ($CPPCI$) are used to model the Probabilistic Controllability of the input operands of $S_i$, i.e., $A$ and $B$. Generally speaking, if the input operands are probabilistic controllable, the output will be more controllable. The detailed relationship between output controllability and input controllability will be shown later. The reason that if $A$ (or $B$) is not defined before $S_i$, $PCI(A)$ (or $PCI(B)$) is used. This is because $A$ (or $B$) must be defined in a previous time frame by some other path. Thus, $PCI(A)$ ($PCI(B)$) should be used as a general controllability index for $A$ (or $B$).

The other factor is data dependency between $A$ and $B$ due to reconvergent fanout. While computing $SPCI$, we are assuming that $A$ and $B$ are independent. The reasons are the following. First, it is hard to resolve the exact dependency. Second, assume that $A$ is a function of $C$ and $D$, e.g., $A = f(C, D)$, and $B$ is a function of $C$ and $E$, e.g., $B=$

81
g(C, E). Due to the difference between (f, g) and (D, E), A and B can still be considered as relatively random to each other while deriving a value-independent SPCI. Thus, this is a good approximation if f and g are different and there exist many different inputs such as D and E.

The last factor which affects SPCI is the operation used in $S_i$. Some operations have the ability to produce more probabilistically controllable results than the others. For example, consider the following two statements, $S_1: X = Y + 1$ and $S_2: X = Y \text{ AND } 0001$, where AND denotes a bit-wise logical AND operation. Assuming that Y is probabilistic controllable, X will be more random in $S_1$ than in $S_2$.

To derive SPCI by considering the different operation used in each statement, the concept of a Probabilistic Controllability Function (PCF) is used.

$$SPCI(p, S_i, N) = PCF(op, CPPCI(p, S_i, A), CPPCI(p, S_i, B))$$

We use the AND and ADD operations as an illustration for computing PCF given every CPPCI($p, S_i, A[A_b]$) and CPPCI($p, S_i, B[B_b]$), where $A_b$ is in $A[A_i : A_h]$ and $B_b$ is in $B[B_i : B_h]$.

4.2.4 PCF(AND, CPPCI($p, S_i, A[A_b]$), CPPCI($p, S_i, B[B_b]$))

The value of $N[N_i]$ depends only on the values of $A[A_i]$ and $B[B_i]$ as does the value of $N[N_{i+1}], N[N_{i+2}]...N[N_h]$. As a result, $SPCI(p, S_i, N[N_b])$ can be completely determined by CPPCI($p, S_i, A[A_b]$) and CPPCI($p, S_i, B[B_b]$), where $[N_b]$ is in $N[N_i : N_h]$, and $A[A_b]$ and $B[B_b]$ are the corresponding bits in $A[A_i : A_h]$ and $B[B_i : B_h]$, respectively. Three cases are examined:
(1) If both $A[A_b]$ and $B[B_b]$ are known constants, $N[N_b]$ is a known constant. Thus, $SPCI(p, S_i, N[N_b])$ becomes 0.

(2) If one of $A[A_b]$ and $B[B_b]$ is a known constant, e.g., $A$ is a constant, consider the following cases:

- If $A[A_b]$ is 0, $SPCI(p, S_i, N[N_b]) = 0$.
  This is because no matter what value is on $B[B_b]$, $N$ is 0.
- If $A[A_b]$ is 1, $SPCI(p, S_i, N[N_b]) = CPPCI(p, S_i, B[B_b])$.
  In this case, $N[N_b]$ is always equal to $B[B_b]$. Thus, $CPPCI(p, S_i, B[B_b])$ will be assigned to $SPCI(p, S_i, N[N_b])$.

(3) Otherwise, we can view $A[A_b]$ (or $B[B_b]$) as a combination of its probabilistically controllable part, $CPPCI(p, S_i, A[A_b])$ (or $CPPCI(p, S_i, B[B_b])$) and the uncontrollable part, $1 - CPPCI(p, S_i, A[A_b])$ (or $1 - CPPCI(p, S_i, B[B_b])$). Then, this leads to the following cases:

- Both operands are not probabilistically controllable (this probability is $(1 - CPPCI(p, S_i, A[A_b])) \cdot (1 - CPPCI(p, S_i, B[B_b]))$):
  $N[N_b]$ will not be controllable at all.

- $A$ is not probabilistically controllable and $B$ is probabilistically controllable (with probability $(1 - CPPCI(p, S_i, A[A_b])) \cdot CPPCI(p, S_i, B[B_b])$):
  The Probabilistic Controllability on $B[B_b]$ can only be transmitted to $N[N_b]$ if $A[A_b]$ is 1. Due to the fact that the whole analysis is value-independent, we can assume the probability that $A[A_b]$ equals 1 is 0.5. Thus, under this condition, $1/2 \cdot (1 - CPPCI(p, S_i, A[A_b])) \cdot CPPCI(p, S_i, B[B_b])$ will contribute to the Probabilistic Controllability on $N[N_b]$. 83
• The other conditions can be derived in a similar fashion.

As a result,

\[
PCF(\text{AND}, \text{CPPCI}(p, S_i, A[A_b]), \text{CPPCI}(p, S_i, B[B_b]))
\]

\[
= \frac{1}{2} \cdot \text{CPPCI}(p, S_i, A[A_b]) \cdot \text{CPPCI}(p, S_i, B[B_b])
\]

\[
+ \frac{1}{2} \cdot \text{CPPCI}(p, S_i, A[A_b]) \cdot (1 - \text{CPPCI}(p, S_i, B[B_b]))
\]

\[
+ \frac{1}{2} \cdot (1 - \text{CPPCI}(p, S_i, A[A_b])) \cdot \text{CPPCI}(p, S_i, B[B_b])
\]

4.2.5 PCF(ADD, CPPCI(p, S_i, A), CPPCI(p, S_i, B))

Unlike a logic AND operation, an ADD operation is not bitwise. As a result, we have to treat \(A[A_i : A_h]\) (and \(B[B_i : B_h]\)) as a whole. Let \(\text{CPPCI}(p, S_i, A)\) denote the average of \(\text{CPPCI}(p, S_i, A[A_b])\), where \(A_b\) is in \(A[A_i : A_h]\), as is \(\text{CPPCI}(p, S_i, B)\). Then, as we handled the AND operation, we can view \(A\) as a combination of its Probabilistically Controllable part, \(\text{CPPCI}(p, S_i, A)\), and its uncontrollable part, \(1 - \text{CPPCI}(p, S_i, A)\), as is \(B\). Then, as long as one of the operands is probabilistically controllable, \(N\) becomes probabilistically controllable, since the ADD operation will uniformly distribute the value on the probabilistically controllable operand to \(N\). Thus,

\[
PCF(op, \text{CPPCI}(p, S_i, A), \text{CPPCI}(p, S_i, B))
\]

\[
= 1 - (1 - \text{CPPCI}(p, S_i, A))(1 - \text{CPPCI}(p, S_i, B))
\]

4.2.6 PCF of other functions

The PCFs of the other functions are shown in Table 4.2. The way in which the PCFs are defined in Table 4.2 is based on the ability of each function to make the output
Table 4.2: PCF of other functions.

<table>
<thead>
<tr>
<th>type</th>
<th>RCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR*, NOR*, NAND*</td>
<td>same as AND</td>
</tr>
<tr>
<td>CAT</td>
<td>weighted sum (by bit width) of both operands’ CPRCI</td>
</tr>
<tr>
<td>EQ*, NE*</td>
<td>2/(total possible combinations on operand)</td>
</tr>
<tr>
<td>GE*, LT*</td>
<td>if one operand is constant: ( \frac{2}{2^{\text{min}(\text{MAX}-\text{constant}, \text{constant}+1)}} \cdot \text{CPPCI} ) ( (\text{MAX}+1) ) otherwise: 1</td>
</tr>
<tr>
<td>GT*, LE*</td>
<td>if one operand is constant: ( \frac{2}{2^{\text{min}(\text{MAX}-\text{constant}+1, \text{constant})}} \cdot \text{CPPCI} ) ( (\text{MAX}+1) ) otherwise: 1</td>
</tr>
<tr>
<td>XOR*, XNOR*</td>
<td>if both operands are variables: 1 otherwise: the variable’s CPRCI</td>
</tr>
<tr>
<td>SUB*, ADDC*, SUBC*</td>
<td>same as ADD</td>
</tr>
<tr>
<td>MUL*, MULC*</td>
<td>if one operand is constant: ( \text{variable’s CPRCI} \cdot \frac{\text{Range(\text{constant})} - # \text{ of 0 in the lsb}}{\text{Range(\text{constant})}} ) otherwise: same as ADD</td>
</tr>
<tr>
<td>SHR, SHL</td>
<td>( \frac{\text{Range(\text{variable})} - \text{total bits shifted}}{\text{Range(\text{variable})}} )</td>
</tr>
<tr>
<td>ROR, ROL</td>
<td>equal to variable’s CPRCI</td>
</tr>
<tr>
<td>NOT, ASG</td>
<td>equal to variable’s CPRCI</td>
</tr>
</tbody>
</table>

uniformly distributed among all of its possible values. For the operations with a “∗” in Table 4.2, their operands have equal bit width. In Table 4.2, MAX refers to the maximum possible value of that specific variable and type CAT refers to the concatenation operation. In addition, the CPPCI in operations GT, LE, GE and LT refers to the CPPCI of the nonconstant variable.

4.3 Probabilistic Observability Evaluation

Similar to the controllability analysis, the Probabilistic Observability Index (POI) of each variable is used as a measure of Probabilistic Observability.
Definition: The Probabilistic Observability Index, $POI(N)$, is defined as a measure of the probability that the content of a variable $N$ can be observed at a primary output \(^3\) under the random inputs assumption.

Initially, only the primary outputs’ POIs are 1. All other variables’ POIs are 0. Then, we recursively update each variable’s POI by computing $PPOI$. Unlike the PCI calculation, for simplicity, not every $POI(N[b])$ ($b$ is in $Range(N)$) is calculated. Instead, only the average observability value $POI(N)$ is derived.

Definition: The Path Probabilistic Observability Index, $PPOI(p, N, M)$, is defined as the probability that after executing one path, $p$, the content of $N$ can be observed at $M$ if $N$ is used in $p$, and $M$ is either of type OUTPUT or REGISTER, and $PPOI(p, N)$ is defined as the probability that when $p$ is used to observe $N$, the content of $N$ will be propagated to at least one primary output.

As in computing $RCI(N)$, $POI(N)$ is determined by

$$POI(N) = \frac{\sum_p PPOI(p, N)}{\#PathSet}$$

where $PathSet$ denotes the set of paths in the given CFG and $p$ is a path in $PathSet$.

Every statement in path $p$ has to be examined to derive $PPOI(p, N)$ and $PPOI(p, N, M)$. This leads to the derivation of $SPOI$.

Definition: Given a statement $S_i$: $C = A$ op $B$, $SPOI(p, S_i, N)$ denotes the probability that the content of $N$ can be propagated to $C$, the output of $S_i$, if $N$ is used in $p$.

Similar to deriving $SRCI$, the Probabilistic Observability of variables $A$ and $B$ over $N$ and the characteristic of op are critical in determining $SPOI$. Then, $CPPOI(p, S_i, \ldots)$

\(^3\)This observation may not be done in one path, but a set of paths are needed to propagate the content of $N$ to primary outputs.
N, A) and CPPOI(p, S_i, N, A) can be derived in the same way as CPRCI. Under the assumption of data independence between A and B, a Probabilistic Observability Function (POF) can also be derived. As a result,

\[ SPOI(p, S_i, N) = POF(op, CPPOI(p, S_i, A, N), CPPOI(p, S_i, B, N)) \]

Let us take the operation addition without carry (ADD) as an example to illustrate how to determine POF. Under the data-independent assumption, if either A or B can observe N (this is indicated by CPPOI(p, S_i, A, N) or CPPOI(p, S_i, B, N) greater than 0), C is able to observe N. As a result,

\[ POF(ADD, CPPOI(p, S_i, A, N), CPPOI(p, S_i, B, N)) \]

\[ = 1 - [1 - CPPOI(p, S_i, A, N)] \cdot [1 - CPPOI(p, S_i, B, N)] \]

The POF of the logic operation AND is determined as follows:

- If one of the operands, e.g., A, is of type CONSTANT:
  
  Find the the number of 1s in this constant divided by the total number of 1s. Let this number be OneRatio. Then,

  \[ POF(AND, CPPOI(p, S_i, A, N), CPPOI(p, S_i, B, N)) \]

  \[ = OneRatio \cdot CPPOI(p, S_i, B, N) \]

- Otherwise:

  As in the controllability case, assume that one half probability that 1 will occur.

  \[ POF(AND, CPPOI(p, S_i, A), CPPOI(p, S_i, B)) \]

  \[ = 1/2 \cdot CPPOI(p, S_i, A) \cdot CPPOI(p, S_i, B) \]
After execution of path \( p \), the content of \( N \) may be observable by one of primary outputs or registers. Then, \( PPOI(p,N,M) \) can be determined similarly to \( PRCI(p,N) \). Note that while deriving \( SPOI \) or \( PPOI \), whether each variable is defined or used by a full range or a partial range should be carefully examined as in the controllability evaluation. For example, let statement \( S \) use 8 out of 16-bit of \( var \), the variable under observation. Then, \( CPPOI(p, S, var, var) \) is 0.5, even though \( var \) is the variable to observe.

Another important note for \( CPPOI \) is the bus split issue. This is illustrated by the following example.

**Example:** Let \( A \) be a 16-bit variable. Statement \( S_1 \) defines \( A[0:10] \) with \( SPOI \) equal to 0.8, and statement \( S_2 \) following \( S_1 \) defines \( A[8:16] \) with \( SPOI \) equal to 1. According to the *multiple define/use* issue mentioned before, the \( SPOI \) of bits 8, 9 and 10 of \( A \) are determined by \( S_2 \). Assume that \( A[5:9] \) is used in statement \( S_3 \). The \( CPPOI \) of \( A[5:9] \) should be

$$ \text{CRPOI}(A[5:9]) = \frac{3}{11} \cdot 0.8 + \frac{2}{9} \cdot 1 $$

Since \( POI(M) \) is the probability that the content on \( M \) will be observed by the primary outputs, \( PPOI(p, N, M) \cdot POI(M) \) denotes the probability that if \( p \) is used to observe \( N \), the content of \( N \) will be observed by the primary outputs in the subsequent paths. Because only the registers are able to hold the content till the execution of the next path, if \( M \) is an internal variable (the output of a combinational module), it is not capable of propagating \( N \) to the primary outputs. As a result,
\[ PPOI(p, N) = 1 - \prod_{M}(1 - PPOI(p, N, M) \cdot POI(M)) \]

where \( M \) is either a register or a primary output other than \( N \).

One other issue that complicates \textit{Probabilistic Observability} derivation is the branch issue. Take the branch variable \( R_b \) in Figure 2.7 for example. The content of \( R_b \) may not reach any registers or primary outputs. As a result, \( R_b \) is completely not sensitizable. However, \( R_b \)'s content is still observable by comparing the difference between two branches taken. Take Figure 4.1 as an example. It shows a branch variable \( R_b \), and a primary output \( OUT \). In this example, the content of \( R_b \) is observable by \( OUT \). In other words, the value of \( R_b \) can be uniquely determined by examining the values of \( OUT \). However, for some other circuits, if the values of one primary output \( OUT \) are the same no matter which branch is taken, then branch variable \( R_b \) cannot be observed by \( OUT \). Under the random inputs' assumption, we can assume that the probability that \( OUT \) takes the same value no matter which branch \( R_b \) is taking is

\[ D(OUT) = 1 - \frac{1}{\text{total combinations on } OUT} \]

Then, \( PPOI(p, R_b) \) should be modified as follows, if \( R_b \) is a branch variable:

\[ PPOI(p, R_B) = 1 - \prod_{OUT} (1 - D(OUT) \cdot POI(OUT) \cdot \frac{\text{Partial}(OUT)}{\text{Full}(OUT)}) \]

where \( OUT \) is either a register or a primary output that is defined in \( p \) and after the branch on \( R_b \) is taken, and \( \text{Partial}(OUT) \) and \( \text{Full}(OUT) \) refer to the range of \( OUT \) defined beneath \( R_b \) and the total range of \( OUT \), respectively. Note that this equation assumes that \( OUT \) is not reachable from \( R_b \). If \( R_b \) reaches \( OUT \), the original method of computing \( PPOI \) is used.
4.4 Probabilistic approach for Synthesis for Testability

In this section, a Synthesis for Testability approach based on the previous probabilistic testability analysis is presented. It consists of two major steps, test point selection and testability modification.

4.4.1 Test point selection

After the Probabilistic Controllability evaluation, we can identify the variables in the circuit which have low Probabilistic Controllability. In this section, a systematic method is used to select test points.

Due to hardware constraints, the number of test points should be limited. We think that the most valuable test points are those having the most influence on other variables. The basic procedure is characterized as follows:
• Use a threshold value on PCI to identify the set of low controllable variables, called TPCandidate.

• Assume that we use extra primary inputs to modify each variable in TPCandidate and rerun BERTA.

• Find the variable with the best profit function (PF). The profit function is defined as

\[
P F(N) = \frac{\sum_{n} PCI_{new}(n) - \sum_{n} PCI(n)}{\text{bit width of } N}
\]

where \( n \) is a variable in the circuit, \( PCI_{new}(n) \) is \( PCI(n) \) after variable \( N \) has been modified and \( PCI(n) \) denotes the original PCI value on \( n \).

• The variable with the best PF value is selected as the test point. If the number of test points selected does not reach the hardware limit and the testability requirement has not been fulfilled, this selection process can be continued to find the subsequent test points.

4.4.2 Testability modification

Several different approaches can be used to modify the circuit controllability given the test points selected in the last section, for example Test Point Insertion [9, 10, 14, 15], Partial Scan Design [16, 17, 18] or Test Statement Insertion [41].
Table 4.3  RCI and ROI results for several sample circuits.

<table>
<thead>
<tr>
<th>name</th>
<th>avg. RCI (org.)</th>
<th>avg. ROI (org.)</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>micro</td>
<td>0.496224</td>
<td>0.546443</td>
<td>1.1</td>
</tr>
<tr>
<td>circuit1</td>
<td>0.775000</td>
<td>0.338889</td>
<td>0.3</td>
</tr>
<tr>
<td>circuit2</td>
<td>0.547732</td>
<td>0.670803</td>
<td>5.1</td>
</tr>
<tr>
<td>circuit3</td>
<td>0.662888</td>
<td>0.396255</td>
<td>33.3</td>
</tr>
<tr>
<td>circuit4</td>
<td>0.428571</td>
<td>0.675000</td>
<td>0.1</td>
</tr>
<tr>
<td>circuit5</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>circuit6</td>
<td>0.186490</td>
<td>0.401312</td>
<td>280.9</td>
</tr>
<tr>
<td>circuit7</td>
<td>0.533081</td>
<td>0.438194</td>
<td>0.9</td>
</tr>
</tbody>
</table>

4.5 Results

Our algorithm has been implemented on a SUN SPARC workstation in the C programming language. It was applied to several sample circuits synthesized from a behavioral synthesis tool.

We applied this tool to the circuits shown in Table 2.1. The results are shown in Table 4.3. The first column shows the name of the circuit. The second column is the average the RCI values in the original circuit among all of the variables in the circuit. The third column shows the average ROI value in the original circuit. The last column shows the run time for generating these results on a SUN SPARC workstation.

According to Table 4.3, micro, circuit2, circuit4, circuit6 and circuit7 have lower RCI values than the others. We then use the heuristic profit function to find the most appropriate test points in each circuit. After modifying those test points, those circuit's average RCI values increase, as shown in Table 4.4. The second and the fourth columns of Table 4.4 show the original and modified RCI values, respectively. The third column shows the number of controllability test points modified in each circuit.
Table 4.4 Average RCI before and after circuit modification.

<table>
<thead>
<tr>
<th>name</th>
<th>avg. RCI (org.)</th>
<th># T.P.</th>
<th>avg. RCI (mod.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>micro</td>
<td>0.496224</td>
<td>1</td>
<td>0.644661</td>
</tr>
<tr>
<td>circuit2</td>
<td>0.547732</td>
<td>1</td>
<td>0.707198</td>
</tr>
<tr>
<td>circuit4</td>
<td>0.428571</td>
<td>1</td>
<td>0.771429</td>
</tr>
<tr>
<td>circuit6</td>
<td>0.186490</td>
<td>5</td>
<td>0.420081</td>
</tr>
<tr>
<td>circuit7</td>
<td>0.533081</td>
<td>1</td>
<td>0.798706</td>
</tr>
</tbody>
</table>

Take circuit7 for example. Table 4.5 shows the RCI value of each variable in circuit7 before and after node1 is selected as a test point.

The test generation results on those sample circuits have been shown in Table 3.2. According to Table 3.2, micro, circuit2, circuit3, circuit4 and circuit5 are already testable. Thus, the test points identified in Table 4.4 are applied only to circuit1, circuit6 and circuit7. Table 4.6 shows the test generation results on these modified circuits. All of those originally less testable circuits become testable after the insertion of test points. The last column of Table 4.6 shows the run time for finding those test points. Compared to Table 3.3, the synthesis for testability approach based on BEPTA is much faster than that based on BETA, especially for larger circuit, for example circuit6. However, BETA and its corresponding synthesis for testability approach are more useful if a deterministic test generator is used, whereas BEPTA is more useful in a random testing environment. Take circuit micro for example. Variable PC is identified as CC in BETA. However, its RCI value is 0.031250, which is very low, and is suggested as a test point according to Table 4.4. This is because among the twenty paths in micro's CFG (Figure 2.13), only two paths can make PC more probabilistically controllable. Thus, PC is relatively less controllable if random patterns are used in test generation. As a result, the controllability of a variable may depend on whether test generation is deterministic or probabilistic.
Table 4.5  RCI results for circuit7.

<table>
<thead>
<tr>
<th>name</th>
<th>old RCI</th>
<th>new RCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>0.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node2</td>
<td>0.779297</td>
<td>0.779297</td>
</tr>
<tr>
<td>node3</td>
<td>0.500000</td>
<td>0.500000</td>
</tr>
<tr>
<td>node4</td>
<td>0.500000</td>
<td>0.500000</td>
</tr>
<tr>
<td>node5</td>
<td>0.500000</td>
<td>0.500000</td>
</tr>
<tr>
<td>node6</td>
<td>0.000000</td>
<td>0.250000</td>
</tr>
<tr>
<td>node7</td>
<td>0.750000</td>
<td>0.750000</td>
</tr>
<tr>
<td>node8</td>
<td>0.750000</td>
<td>0.750000</td>
</tr>
<tr>
<td>node9</td>
<td>0.750000</td>
<td>0.750000</td>
</tr>
<tr>
<td>node10</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node11</td>
<td>0.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node12</td>
<td>0.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node13</td>
<td>0.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node14</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node15</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node16</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
</tbody>
</table>

Table 4.6  Test generation results after circuit modification.

<table>
<thead>
<tr>
<th>name</th>
<th>test points</th>
<th>f cov. (%)</th>
<th>f eff. (%)</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuit1</td>
<td>1</td>
<td>100.000</td>
<td>100.000</td>
<td>0.1 sec</td>
</tr>
<tr>
<td>circuit6</td>
<td>5</td>
<td>77.363</td>
<td>95.055</td>
<td>78.52 min</td>
</tr>
<tr>
<td>circuit7</td>
<td>1</td>
<td>68.872</td>
<td>98.054</td>
<td>1.5 sec</td>
</tr>
</tbody>
</table>
Table 4.7 RCI results for circuit1.

<table>
<thead>
<tr>
<th>name</th>
<th>old RCI</th>
<th>new RCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>0.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node2</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node3</td>
<td>0.666667</td>
<td>0.666667</td>
</tr>
<tr>
<td>node4</td>
<td>0.666667</td>
<td>0.666667</td>
</tr>
<tr>
<td>node5</td>
<td>0.666667</td>
<td>0.666667</td>
</tr>
<tr>
<td>node6</td>
<td>0.666667</td>
<td>0.666667</td>
</tr>
<tr>
<td>node7</td>
<td>0.666667</td>
<td>0.666667</td>
</tr>
<tr>
<td>node8</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node9</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node10</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node11</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node12</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node13</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node14</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node15</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>node16</td>
<td>0.500000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node17</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node18</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node19</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>node20</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
</tbody>
</table>

The average RCI value shown in Table 4.3 may be misleading. High average RCI values do not guarantee that all of the variables have a high RCI. For example, the circuit1 in Table 4.3 does have high average RCI value. However, some internal variables have very low RCI values. Table 4.7 shows the RCI of each variable in circuit1. Most variables in circuit1 have high RCI values, except node1, node15 and node16, where node16 is 8 bits wide and 4 out of 8 bits have an RCI equal to 1 while the other 4 bits' RCI are 0. This explains the reason why circuit1 has low fault coverage, as shown in Table 3.2. After selecting node1 of circuit1, its fault coverage becomes 100% as shown in Table 4.6.
CHAPTER 5

SUMMARY

An approach for testability analysis, called \textit{BETA}, is first presented in this thesis. Unlike a traditional testability analysis tool, \textit{BETA} evaluates testability at the behavioral level using the \textit{Control Flow Graph (CFG)}. By using CFG, more of a circuit's functionality can be explored. Also, this allows testability evaluation to be done in the early design phase of the circuit design when no detailed structural description of the circuit exists.

The first procedure used in \textit{BETA} is path analysis. Each path in CFG can be considered as a macro instruction executed by the circuit. Thus, a path in CFG can be considered as a basic unit of operation performed by the circuit. Then, justification and propagation can be transformed to be a path traversal problem. This motivates the path analysis. After the analysis, all of the paths can be used to justify or propagate each variable are first identified. Also, all of the variables that need justification if a path is used to justify or propagate a specific variable are identified and associated to each path.

One issue that complicates the whole procedure is that each variable may be defined or used more than once in a path, and not every bit of that variable is involved in each definition or usage. Careful examination of the bits involved in each definition or usage is necessary.

After identifying all of the justification and propagation paths for each variable, \textit{BETA} tries to find the most controllable and observable ones. Before identifying such paths, \textit{variable classification} is first done to classify all of the variables into several groups.
according to their intrinsic controllability and observability. This leads to the concept of Completely Controllable (CC) and Completely Observable (CO). Unlike other testability analysis tools give only heuristic values on controllability and observability, BETA derives the exact writing and reading sequence for CC and CO variables. This information helps the test generator in controlling and observing those variables.

For the variables which are not CC or CO, further analysis is done by subdividing it into Partial Completely Controllable (PCC), Partial Completely Observable (PCO) and Value Completely Controllable (VCC). For PCC (PCO) variables, the controllable (observable) subrange is identified. For VCC variables, the controllable values are identified. For the remaining not controllable variables, some heuristics are derived to distinguish the relative controllability among these less controllable variables.

The second part of this thesis describes a behavioral-level synthesis for testability approach. Based on the controllability and observability information derived by BETA, the less controllable and observable variables are identified. They are the candidates for the test points. Two test point selection methods is presented in this thesis. The common objective of these two methods are to reduce the total number of less controllable (or observable) variables by making one uncontrollable (or unobservable) variable controllable (or observable).

Traditionally, test point insertion or partial scan design are used to modify the test points. In this thesis, another method called Test Statement Insertion (TSI), which modifies CFG directly is presented. As a result, it can be applied to a high-level design environment, such as a high or behavioral-level synthesis tool. Also, it is less expensive than both test point insertion and scan design in terms of extra pins and area space consumption. In addition, unlike partial scan design, no extra test clock is required. This makes
at-speed testing possible. The only drawback is the testability penalty. By combining both BETA and the synthesis for testability approach with an existing behavioral-level synthesis tool, a complete behavioral level synthesis for testability cycle can be formed such that the circuits generated by this synthesis tool would be automatically testable.

The last part of this thesis presents an approach for evaluation and synthesis for random testability. The importance of a built-in self-test motivates the research on random testability. In this thesis, an approach for random testability analysis is presented. As in BETA, CFG is analyzed instead of the circuit's structural diagram. Under the assumption that every primary input pattern has equal probability, random controllability and observability for each variable are derived. Random controllability of each variable is defined as the probability that a specific variable can be uniformly set to all of its possible values, whereas random observability is defined as the probability that one variable's content can be propagated to at least one primary output.

Based on this random testability analysis, less controllable and observable variables can be identified. Then, as in the synthesis for testability case, the most critical test points can be identified. The designer can then use test point insertion, partial scan design or the proposed TSI to make those test point testable.

All of these approaches have been implemented as a computer program using C programming language. Several sample circuits generated by a behavioral-level synthesis tool are used to evaluate the effectiveness of these approaches, and the results are encouraging.
REFERENCES


Chung-Hsing Chen was born in [redacted]. He received the B.S. degree in Electrical Engineering from the National Taiwan University, Taipei, Taiwan, in 1985. From 1985 to 1987, he served in the Noncommissioned Officer School, Chinese Army, as a platoon leader and research officer. He obtained his M.S. degree in Electrical and Computer Engineering from the University of Massachusetts at Amherst in 1989. From 1989 to 1992, he was employed as a research assistant with the Center for Reliable and High-performance Computing at the Coordinated Science Laboratory.

After completing his doctoral degree, he will join Hewlett-Packard, Santa Clara, California. His research interests include testing, computed-aided design, fault-tolerant computing, synthesis and computer architecture.