

**A KNOWLEDGE BASED SYSTEM  
FOR TESTABLE DESIGN  
METHODOLOGY SELECTION**

by

**Xi-an Zhu**

**Technical Report CRI-86-23**

-----

A Dissertation Presented to the  
**FACULTY OF THE GRADUATE SCHOOL  
UNIVERSITY OF SOUTHERN CALIFORNIA**

In Partial Fulfillment of the  
Requirements for the Degree  
**DOCTOR OF PHILOSOPHY**  
(Electrical Engineering)

August 1986

Copyright 1986 Xi-an Zhu

UNIVERSITY OF SOUTHERN CALIFORNIA  
THE GRADUATE SCHOOL  
UNIVERSITY PARK  
LOS ANGELES, CALIFORNIA 90089

*This dissertation, written by*

.....  
*Xi-An Zhu*  
.....

*under the direction of h... Dissertation  
Committee, and approved by all its members,  
has been presented to and accepted by The  
Graduate School, in partial fulfillment of re-  
quirements for the degree of*

**DOCTOR OF PHILOSOPHY.**

.....  
*Barbara Solomon*  
.....

*Dean of Graduate Studies*

*Date August 1, 1986*  
.....

**DISSERTATION COMMITTEE**

.....  
*Mark Allen*  
.....

*Chairperson*

.....  
*Alisa C. Parker*  
.....

.....  
*James W. Potts Preser*  
.....

# DEDICATION

To

My mother and father,

and my dear friends

## ACKNOWLEDGEMENTS

I take this opportunity to express my sincere appreciation to Professor Melvin A. Breuer for his guidance and support during my graduate study in USC. I consider it my privilege to have worked with him. His ideas, encouragements and constructive criticisms made this thesis possible.

I thank Professor Alice Parker and Professor Francesco Parisi-Predicce for serving on my dissertation committee. Their valuable suggestions enhanced the presentation of this thesis. I also thank Dr. Alan Bond and Dr. Yigal Arens for many useful discussions during this research.

I thank my parents, sister and brother for their endless love, care and encouragement throughout my life.

Although it is too many to name them individually, I will never forget my advisors and friends in China who have given me a hand when I was pursuing my college and graduate education during a special period in the history. Without their help, I could not even be here. I thank my schoolmates in the Graduate School of Chinese Academy of Sciences, for their spirit and achievements have greatly inspired me.

I thank all my friends and colleagues in USC for their warm friendship and various help.

Finally, I gratefully acknowledge the financial support received from the National Science Foundation (Grant MCS-8203485) and the Defense Advanced Research Projects Agency (Contract No. N00014-84-K-0649). It was their support that enabled me to come to the United States and finish my doctorate.

# Table of Contents

<b>1. Introduction</b>	<b>1</b>
1.1. Background	1
1.1.1. VLSI testing problem	1
1.1.2. Design for testability	2
1.1.2.1. Ad hoc design for testability	3
1.1.2.2. Structured design for testability	4
1.1.2.3. Built-in-testing and self-testing designs	5
1.2. A knowledge based approach for DFT	6
1.2.1. Testable design methodologies	6
1.2.2. A divide-and-conquer strategy for DFT	9
1.2.3. A testable design expert system	10
1.3. Proposed task	11
1.4. Outline of dissertation	12
<b>2. Testable designs methodologies for PLAs</b>	<b>14</b>
2.1. PLA models	15
2.2. PLA testing problems	19
2.2.1. Fault models	19
2.2.2. Problems with traditional test generation methods	21
2.3. Test Generation Algorithms for PLAs	22
2.3.1. Deterministic test generation	23
2.3.2. Semi-random test generation	25
2.4. Testable PLA designs	27
2.4.1. Concurrent testable PLAs with special coding	28
2.4.1.1. PLA with concurrent error detection by a series of checkers	29
2.4.1.2. Concurrent testable PLAs using Modified Berger code	31
2.4.2. Parity testable PLAs	33
2.4.2.1. PLA with universal test set	34
2.4.2.2. Autonomously testable PLAs	39
2.4.2.3. A built-in self-testable PLA design with cumulative parity comparison	41
2.4.3. Signature testable PLAs	43
2.4.3.1. PLA with multiple signature analyzers	44
2.4.3.2. Self-testable PLAs with single signature analyzer	45
2.4.3.3. A self-testable PLA design based on crosspoint	47

	v
counting	
2.4.4. Partitioning and testing of PLAs	49
2.4.4.1. PLA with BILBOs	49
2.4.4.2. Parallel testable PLAs	51
2.4.4.3. Divide and conquer strategy for testable PLA design	53
2.4.5. Fully testable PLA designs	54
2.4.5.1. A design of testable PLAs by specialized layout	54
2.4.5.2. A PLA design for testing single bridging faults	55
2.4.5.3. Fully testable PLAs	57
2.4.5.4. A design for complete testability of PLAs	57
2.4.5.5. A PLA design for ease of test generation	58
2.4.5.6. Low overhead design of testable PLAs	60
2.5. Evaluation of PLA TDMs	62
2.5.1. Measures of TDMs	62
2.5.1.1. Testability characteristics	63
2.5.1.2. Resulting effect on the original design	66
2.5.1.3. Requirements on test environment	70
2.5.1.4. Design cost	77
2.5.2. Evaluation matrix	81
2.5.3. Performance statistics of PLA TDMs	83
2.5.3.1. Area overhead vs. PLA size	90
2.5.3.2. Test storage vs. PLA size	93
2.5.3.3. Test application time vs. PLA size	96
2.6. Conclusion	101
<b>3. A general approach for selection</b>	<b>103</b>
3.1. The general selection problem	103
3.1.1. Main factors affecting selection	104
3.1.1.1. The selector	104
3.1.1.2. The receiver	105
3.1.1.3. The domain of selection	106
3.1.1.4. Requirements	110
3.1.1.5. Priorities of attributes	111
3.1.2. The selection problem and selection process	111
3.1.3. Characteristics of a selection process	112
3.2. The evaluation function	115
3.2.1. Properties of attributes	115
3.2.2. The comparison function	119
3.2.3. The penalty-credit function	125
3.2.3.1. General format of PCFs for numeric attributes	127
3.2.3.2. General format of PCFs for logical attributes	130
3.2.3.3. General format of PCFs for complex attributes	130
3.2.4. The score function	132

	vi
3.3. Analysis of solution space	134
3.3.1. Structure of the solution space	134
3.3.2. Relationship among subdimensions	138
3.3.3. Regions in a solution space	139
3.3.3.1. The candidate solution region	141
3.3.3.2. The single failure region	143
3.3.3.3. The tradeoff region	144
3.3.3.4. The complete failure region	146
3.4. Ramification analysis in a selection processes	147
3.4.1. Ramification graphs	148
3.4.2. Static ramification bushes	151
3.4.3. Dynamic ramification trees	160
3.4.4. Ramification analysis	167
3.4.4.1. Consistency checking	167
3.4.4.2. Consequences of changing requirements	172
3.4.4.3. Incremental requirement specification	178
3.5. The dynamic selection process	182
3.5.1. Searching under fixed requirements	183
3.5.2. Reason analysis directed backtracking	186
3.5.3. Knowledge guided search and selection	189
3.6. Knowledge based expert consultant systems	190
3.6.1. General characteristics of the expert consultant system	192
3.6.2. The selector-consultant relationship	193
3.6.3. The knowledge base	194
3.6.3.1. Information content of the knowledge base	195
3.6.3.2. Knowledge representation	196
3.6.4. Control mechanism	199
3.7. Related research	199
3.7.1. Linear assignment method	201
3.7.2. Simple additive weighting method (SAW)	202
3.7.3. Hierarchical additive weighting method	203
3.7.4. ELECTRE method	204
3.7.5. Technique for ordering preference by similarity to ideal solution (TOPSIS)	206
<b>4. A knowledge based system for selecting PLA TDMs</b>	<b>210</b>
4.1. Overview of PLA-TSS	210
4.1.1. The task of PLA-TSS	211
4.1.2. The structure of PLA-TSS	212
4.2. The evaluation functions	216
4.2.1. The weight vector	216
4.2.2. Penalty-credit functions	217
4.2.2.1. Custom-defined PCFs	217
4.2.2.2. Standard PCFs	218

	vii
4.2.2.3. Default PCFs	220
4.2.3. The requirements	224
4.2.4. The score function	226
4.3. Searching in the three dimensional design space	231
4.3.1. Variations of TDMs	231
4.3.2. Characteristic Variations of TDMs	235
4.3.3. Environment consideration in selecting a TDM	237
4.3.4. The general search procedure	241
4.4. Reason Analysis	243
4.4.1. Reason analysis for search failures	243
4.4.1.1. Classification of failures	245
4.4.1.2. Ordering of failure classes	251
4.4.2. Reason analysis for confirmation failures	253
4.5. The control mechanism	256
4.5.1. Initialization of a selection process	259
4.5.2. Search for a solution	261
4.5.3. The main user's actions in a selection process	262
4.5.3.1. Designer selects a TDM	263
4.5.3.2. Selecting a TDM by the system	264
4.5.3.3. Making changes	265
4.5.3.4. Checking system information	266
4.5.3.5. Tracing design history	268
4.5.3.6. Explanation	269
<b>5. A case study</b>	<b>271</b>
<b>6. Conclusions and future work</b>	<b>303</b>
6.1. Conclusions	303
6.2. Future work	306
<b>References</b>	<b>311</b>



## List of Figures

<b>Figure 1-1:</b>	Some available DFT techniques	6
<b>Figure 2-1:</b>	A taxonomy of PLA TDMs	16
<b>Figure 2-2:</b>	A general PLA structure	17
<b>Figure 2-3:</b>	Input decoders	17
<b>Figure 2-4:</b>	A PLA example	18
<b>Figure 2-5:</b>	Embryonic test patterns for an AND gate	25
<b>Figure 2-6:</b>	A concurrent testable PLA design	29
<b>Figure 2-7:</b>	A PLA design for concurrent testing	32
<b>Figure 2-8:</b>	PLA with universal test set	35
<b>Figure 2-9:</b>	Augmented input decoder and its truth table	36
<b>Figure 2-10:</b>	Function-independent testable PLA	38
<b>Figure 2-11:</b>	An autonomously testable PLA	39
<b>Figure 2-12:</b>	A testable PLA with cumulative parity comparison	42
<b>Figure 2-13:</b>	Self-testable PLA using signature analysis	44
<b>Figure 2-14:</b>	Block diagram of a self-testing PLA	46
<b>Figure 2-15:</b>	A built-in self-testable PLA using crosspoint counting	47
<b>Figure 2-16:</b>	Testing a PLA using BILBOs	50
<b>Figure 2-17:</b>	A parallel testing PLA structure	52
<b>Figure 2-18:</b>	A testable PLA design with partitioning	53
<b>Figure 2-19:</b>	Testable design for eliminating untestable bridging faults	56
<b>Figure 2-20:</b>	Complete testable design (CTD)	59
<b>Figure 2-21:</b>	A testable PLA design	60
<b>Figure 2-22:</b>	Augmented PLA for multiple fault detection	68
<b>Figure 2-23:</b>	An easily testable PLA architecture	74
<b>Figure 2-24:</b>	A built-in test PLA	79
<b>Figure 2-25:</b>	Area overhead vs PLA size for PLA TDM FIST [30]	92
<b>Figure 2-26:</b>	Area overhead vs PLA size	93
<b>Figure 2-27:</b>	Test storage vs PLA size	96
<b>Figure 2-28:</b>	Test application time vs PLA size	99
<b>Figure 3-1:</b>	Illustration of upper-bound and lower-bound property	117
<b>Figure 3-2:</b>	The typical form of the PCF for numeric attributes	128
<b>Figure 3-3:</b>	A structure tree of the TDM domain	136

<b>Figure 3-4:</b>	Flattening a subtree	137
<b>Figure 3-5:</b>	A sample two-dimensional solution space	140
<b>Figure 3-6:</b>	A two-dimensional solution space divided by the requirements	140
<b>Figure 3-7:</b>	Region1: the solution region	141
<b>Figure 3-8:</b>	Region 2: the single critical failure region	144
<b>Figure 3-9:</b>	Region 3: the tradeoff region	145
<b>Figure 3-10:</b>	Region 4: the complete failure region	146
<b>Figure 3-11:</b>	The complete R-bush for self-testing	150
<b>Figure 3-12:</b>	The static R-bush for fault coverage	153
<b>Figure 3-13:</b>	The R-bush for $f_c$ when $t_g = no$	157
<b>Figure 3-14:</b>	A static R-tree	158
<b>Figure 3-15:</b>	A dynamic R-bush for $f_c$	163
<b>Figure 3-16:</b>	Two dynamic R-trees for a requirement vector	166
<b>Figure 3-17:</b>	Two non-conflict dynamic R-trees for a requirement vector	173
<b>Figure 3-18:</b>	A typical selection process	182
<b>Figure 3-19:</b>	A knowledge guided selection process	191
<b>Figure 3-20:</b>	Selection by the selector only	192
<b>Figure 3-21:</b>	Selection with help of an expert consultant	192
<b>Figure 3-22:</b>	Basic configuration of an expert consultant system	193
<b>Figure 3-23:</b>	A basic TDM frame	198
<b>Figure 4-1:</b>	Overview of PLA-TSS	212
<b>Figure 4-2:</b>	The PCF for area overhead for $r = 20$ and $r = 30$	220
<b>Figure 4-3:</b>	The default PCF for numeric attributes	221
<b>Figure 4-4:</b>	The PCF for extra connections	222
<b>Figure 4-5:</b>	Solution area indicated by requirements	225
<b>Figure 4-6:</b>	Control flow in PLA-TSS	258

## List of Tables

<b>Table 2-1:</b>	A universal test set for PLAs	36
<b>Table 2-2:</b>	Abbreviations of TDMs in the evaluation matrix	82
<b>Table 2-3:</b>	The evaluation matrix	84
<b>Table 2-4:</b>	52 PLAs used for statistics	90
<b>Table 3-1:</b>	TDM attributes and their units and value range	109
<b>Table 3-2:</b>	Properties of TDM attributes	120
<b>Table 3-3:</b>	General form of the PCF for logical attributes	130
<b>Table 3-4:</b>	A typical PCF for a lower-bound logical attribute	130
<b>Table 3-5:</b>	A small evaluation matrix	150
<b>Table 3-6:</b>	A small fully specified evaluation matrix	163
<b>Table 4-1:</b>	A PCF for test generation	220
<b>Table 4-2:</b>	The default PCF for self-testing	223
<b>Table 4-3:</b>	Calculation of scores [I]	229
<b>Table 4-4:</b>	Calculation of scores [II]	230
<b>Table 4-5:</b>	Calculation of scores [III]	230
<b>Table 4-6:</b>	Effect of the three variations on TDM attribute values	234
<b>Table 4-7:</b>	The FAIL lists	244
<b>Table 4-8:</b>	The critical fail lists for 15 TDMs	249

## Abstract

In designing a testable VLSI circuit, numerous testable design methodologies (TDMs) can be used. Designers thus face the problem of selecting an appropriate TDM to match their goals and constraints. TDMs usually have multiple incommensurable attribute values, some of which are functions of the circuit under consideration. Selection is based on a requirement vector which specifies multiple design goals and constraints. This problem is further complicated due to the facts that there are often no TDMs which completely satisfy the initial requirements, and the requirements may be changed during the selection process.

In this dissertation, the selection problem and the selection process are modeled and analyzed in depth. Key factors affecting a selection are identified. A knowledge based system approach to aid in selection is proposed, and the basic principles and framework of such a consultation system are established. A prototype of a knowledge based selection system, called PLA-TSS (Programmable Logic Array TDM Selection System), is described in detail which emulates a human expert in assisting a designer in selecting a suitable TDM for a given PLA and requirement vector. A *knowledge base* containing numerous TDMs for PLAs has been constructed, where TDMs are represented by frames. Based on this knowledge, *ramification analysis* is carried out, which reveals the relationship among candidate TDMs and their attribute values, so as to help a designer in specifying and changing requirements. A *score function* consisting of a distance function and a penalty-credit function is defined and used to evaluate and compare TDMs. The dynamic selection process is controlled by *reason directed backtracking*, in which when no solution can be found, the reasons for the failure are identified and the best way to resolve the failure is suggested. The principles used in PLA-TSS are also applicable to other selection problems of a similar nature.

# Chapter 1

## Introduction

The rapid growth of VLSI techniques and systems has been one of the most remarkable achievements in modern science. Advances in fabrication technology have led to tremendous increases in chip density, complexity and functionality. These changes present great challenges to design methodology, design tools, and testing.

### 1.1. Background

#### 1.1.1. VLSI testing problem

Before VLSI became a reality, basic building blocks of digital systems were relatively small functional units, each of which consisted of less than a few thousand transistors. These units were homogeneous in that they contained only a single function, such as a RAM or ROM. Numerous algorithms and heuristics have been developed to test such circuits [12, 63]. Most testing methods were directed toward a specific type of circuit, such as combinational logic, and the complexity of these techniques increased rapidly with the size of the circuit-under-test (CUT).

VLSI chips are often composed of several types of circuits and contain

up to a million transistors. Deeply embedded logic and limited I/O pins result in poor controllability and observability. Therefore generating test patterns and performing a test are both time-consuming and expensive tasks. As VLSI systems increase in size, testing VLSI circuits becomes a critical problem, which, if not solved, will be a bottleneck in developing even larger integrated systems.

It is important to realize that the VLSI testing problem is not only caused by the complexity of VLSI circuits, but also by the conventional design process which separates design and test into two independent phases. Thus the key to solving the testing problem is **design for testability** (DFT), i.e., designing a circuit in such a way that it is inherently testable. New design methodology and efficient CAD tools are needed to meet this challenge.

### **1.1.2. Design for testability**

Testability is a property of a circuit which allows for the efficient generation of tests, and for the circuit to be tested easily and/or cost-effectively. Design for testability embodies a collection of techniques which deal with how to modify a circuit to improve its testability. This is a relatively new field in computer engineering, and is becoming increasingly attractive due to the development of VLSI systems. In recent years, many DFT techniques have been proposed [82]. In this section we will briefly review some of these techniques.

### 1.1.2.1. Ad hoc design for testability

*Ad hoc* DFT techniques try to make minor changes or use some special features of a circuit to enhance testability. They are applicable to some circuit, but may not solve general problems. Examples of ad hoc methods include the following.

- Employing special design rules. For example, all registers must have the reset function, and sequential logic must be synchronous.
- Adding extra test points to increase controllability and/or observability.
- Using a special testable architecture. For example, a bit-sliced system can be tested by a constant number of patterns regardless of the length of the array [73], if it satisfies certain conditions.

Ad hoc techniques are often used in combination with testability analysis [32, 43, 18, 76, 48] which identifies the area of a circuit having poor testability so as to determine the locations for placing test points and other hardware modification.

### 1.1.2.2. Structured design for testability

The structured DFT approaches transform a circuit into a well defined testable structure. For example, it is well known that sequential circuits are hard to test because embedded states are difficult to control and observe. Therefore many structured design methods are aimed at obtaining complete controllability and observability of all internal memory elements in a sequential circuit by chaining them into a shift register in the test mode. By so doing the internal states can be explicitly controlled/observed by shifting the values of registers in/out. In this way, the circuit to be tested becomes combinational, for which testing is a well understood and tractable problem.

The most popular structured DFT techniques are scan designs such as LSSD [23], Scan Path [27], Scan/Set Logic [75] and the Random Access Scan [5]. They differ in the way that memory elements are designed, organized and accessed.

Structured approaches lend themselves more easily to design automation and are generally applicable to many digital circuits independent of their functions and design styles.



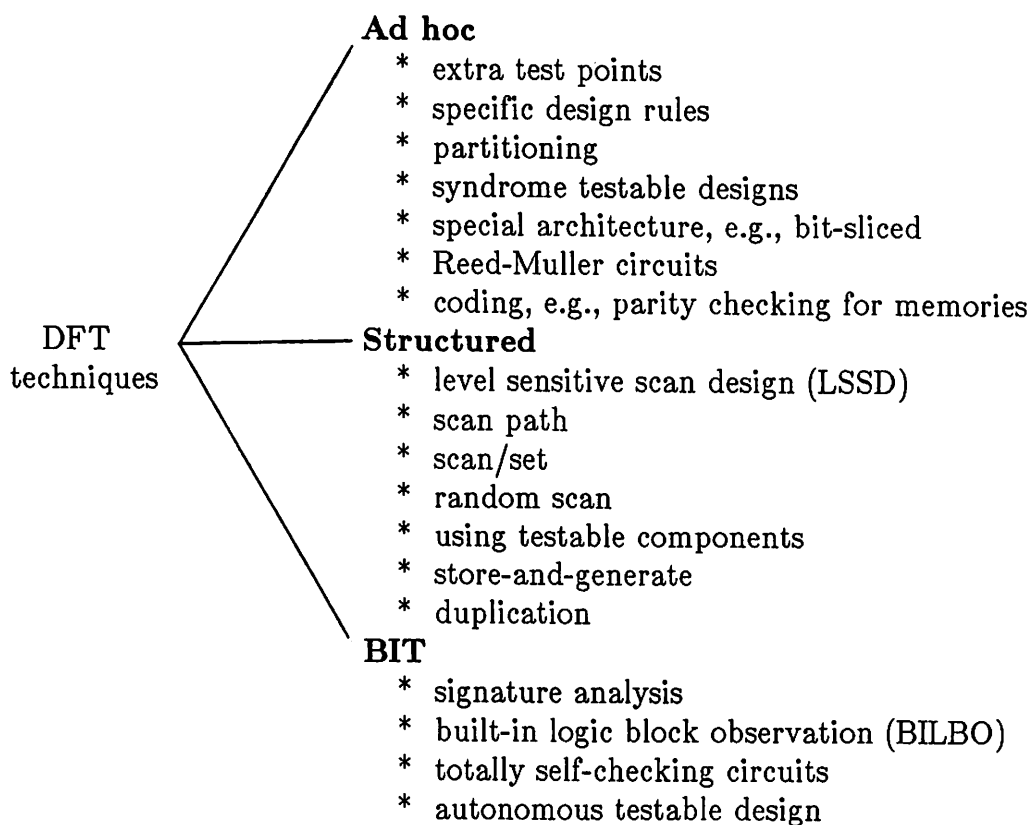
### 1.1.2.3. Built-in-testing and self-testing designs

Both the ad hoc techniques and structured approaches make testing easier, but they may still require external test equipment and software test generation which result in high testing cost and reduced field maintainability. Another class of methods for solving this problem is known as built-in-testing (BIT). The basic idea of BIT is to insert test hardware into a circuit such as to eliminate or greatly simplify external test generation, response evaluation and real time test control. BIT techniques may make a circuit semi-self-testable or fully self-testable, where the testing may be on-line or off-line. Compact testing techniques<sup>1</sup> and the decreasing cost of silicon area make BIT possible to implement with acceptable overhead.

Some examples of available DFT techniques are given in Figure 1-1. It can be seen that many approaches are available for design for testability. DFT techniques trade extra cost and possible performance degradation for testability. Therefore in trying to make a given circuit testable under certain constraints there are a number of choices and tradeoffs to make. How to utilize and integrate the existing techniques to make a custom-designed VLSI chip testable in a cost-efficient and effective way is the problem to be addressed next.

---

<sup>1</sup>Traditional testing methods evaluate every output pattern, while in compact testing, a sequence of outputs is compacted into a single "signature," and only this signature is evaluated. Typical compact testing techniques are signature analysis, syndrome testing / one's counting, and transition counting.



**Figure 1-1:** Some available DFT techniques

## 1.2. A knowledge based approach for DFT

### 1.2.1. Testable design methodologies

Although DFT techniques have quite different features, they can all be described using a common model, referred to as a testable design methodology (TDM) [1, 13]. Each TDM defines a unique process of designing and testing a testable circuit. A TDM has six main components.

1. **Applicable kernel structure.** Four fundamental units of logic used to implement digital systems are referred to as basic structures, namely busses, RAMs, registers and combinational logic. A structure is defined as either a basic structure or a

compound structure, which is an interconnection of two or more basic structures. Any digital circuit is a structure and may be partitioned into substructures. A structure to be tested is called a kernel. The applicable kernel structure of a TDM is the set of all types of kernels to which the TDM can be applied.

2. **Logic implementation.** Logic implementation of a TDM specifies what BIT hardware (either inside the kernel or in the circuit containing the kernel) is required for test generation, response evaluation, test information transfer, control, and testability improvement.
3. **A test set or a test creation method.** A test  $t^*$  for a fault  $f^*$  in a kernel  $K$  is a sequence of one or more input patterns which, when applied to  $K$ , will cause a different output from the fault free response if  $f^*$  is present. A TDM may specify a set of tests required to cover the faults to be detected. If the test set is not fixed, i.e. is kernel-dependent, there must be some test generation method(s) (either implemented in software or hardware) which can be used to create the desired set of tests. The test set influences fault coverage, test application time, and many other aspects of testing.
4. **Test environment.** The test process consists of applying a set

of test vectors to a kernel and observing and evaluating the kernel's responses. The test environment of a TDM specifies what hardware and software are required in the test process for

- preparing, storing or generating test vectors,
- applying the test set to the kernel,
- observing the kernel's outputs,
- evaluating the responses obtained,
- transferring information between the kernel and other hardware components involved in testing, and
- controlling the test process.

5. **A test plan.** A test plan of a TDM specifies the sequence of operations required to actually execute the test process defined by the TDM, and includes control and timing information associated with the test environment.

6. **A set of measures.** The characteristics of each TDM are represented by a set of common measures for TDMs. These measures are in terms of the attributes of concern in digital system design and test. TDMs are distinguished by their different values for these attributes.

A suitable combination of a TDM and a kernel structure creates a testable structure, in which case the kernel is considered testable. If every

kernel in a CUT is embedded in a testable structure, by definition the entire circuit is testable.

### **1.2.2. A divide-and-conquer strategy for DFT**

The state-of-the-art of DFT techniques shows that there are numerous TDMs for individual kernels in VLSI circuits, but there are very few good TDMs for an entire chip. To apply existing TDMs to a VLSI chip, a knowledge based divide-and-conquer strategy [1, 13, 87] can be employed which has the following main steps.

1. Partition a circuit into tractable structures (kernels). A kernel is tractable if there are known TDMs for it.
2. Select a suitable TDM for each kernel such that every kernel can be embedded in a testable structure.
3. Synthesize these testable structures into a global testable chip.

This step includes

- modifying the kernels and/or inserting BIT circuits if necessary,
- embedding the chosen TDMs into the circuit while sharing BIT hardware among testable structures as much as possible so as to minimize overhead,
- synthesizing an on-chip test controller if necessary, since

the inserted BIT hardware requires control signals which cannot always be provided by the original circuit, nor be applied externally due to limited pin count.

4. Generate a test plan which includes test schedules for each kernel and a global test plan for the entire chip [3].

In general, this procedure is not a simple divide-and-conquer process because there are interactions between the steps and between TDMs for different kernels. A number of choices exist at every step, and so the possible designs are numerous. How to reach an "optimal" testable design quickly is a complicated problem. Intelligent decisions at each step are desired, and backtracking may be necessary.

### **1.2.3. A testable design expert system**

It is generally agreed that the solution to the VLSI testing problem is to incorporate testability into the design process. Since most chip designers are not experts in testing and DFT, this new design philosophy requires new computer aided tools for helping designers create testable chips. Using knowledge based systems is an expedient way to solve this problem. Recently several such systems have been proposed. Bellon *et al.* [7] proposed an intelligent assistant for selecting automatic test pattern generation tools; Horstmann [34] described an expert system for DFT rule checking (mainly LSSD design rules) using PROLOG; and Mangir [49] suggested using expert system for tradeoff analysis in testable design.

An automatic DFT system for a specific silicon compiler is under development at the GTE labs [28].

At USC prototypes of a testable design expert system (TDES) have been built. As a subsystem in an Advanced Design AutoMation system (ADAM) [29], the task of TDES is to take a custom-designed circuit from a synthesis system, find an appropriate way of using known TDMs to enhance the testability of the chip using the testability goals and constraints specified by the user, and then modify the original design so that it becomes a testable design. TDES is a knowledge based system which integrates up-to-date TDMs and the divide-and-conquer design strategy. It works interactively with the user as an intelligent consultant and assistant. A prototype of TDES, TDES1, was developed by Magdy Abadir [2] in parallel to the research to be presented in this dissertation. TDES1 has the functions of partitioning a circuit into kernels, finding all possible ways to embed a given TDM into a circuit, and producing an optimal test schedule for an embedding. One open question is how to determine which TDM should be used for which kernel. This question is addressed here.

### 1.3. Proposed task

This thesis focuses on the problem of selecting a suitable TDM for a given kernel and the issues involved in developing an intelligent system to assist TDM selection. The problem can be generally stated as follows:

Given (1) a kernel  $K$  for which known TDMs exist,

- (2) the circuit containing K, and
- (3) a set of loosely defined design requirements,

help a designer to determine the most suitable TDM for K.

The purpose of studying this problem is twofold. First, if the circuit to be made testable can be viewed as a single kernel, such a system constitutes the main body of TDES. Secondly, when the circuit consists of multiple kernels, it can be made testable by handling each kernel one by one [2]. Since there may be a large number of TDMs for each type of kernel and these TDMs have different features, it is important to select a suitable TDM for each kernel in order to obtain a good overall design instead of randomly picking any TDM.

Two major problems are associated with this task.

- **DFT problem:** To determine which TDM is suitable for which application, the TDMs must first be well understood and evaluated.
- **AI problem:** How to build a system which can emulate a human expert in using knowledge to make a good selection?

#### 1.4. Outline of dissertation

In this chapter, we have introduced the motivation for and background material relating to design for testability, as well as a knowledge based approach and a knowledge based system for DFT. The main problem



to be solved in this thesis is defined as selecting the most suitable TDM for a kernel, which is an important step in making a VLSI chip testable.

We will give a closer look of the TDM selection problem by studying one type of kernel--the programmable logic array (PLA). In Chapter 2, existing TDMs for PLAs are briefly surveyed. The measures for evaluating TDMs are defined and applied to the PLA TDMs. It will be shown that each TDM is an entity with multiple attributes. The performances of TDMs vary with the parameters of the kernel, and the suitability of TDMs is subject to application requirements.

In Chapter 3 we will define and analyze the multiple attribute selection problem and the selection process, propose a method for solving such a problem, and give a framework of a knowledge based selection system.

In Chapter 4 a prototype of a knowledge based TDM selection system, PLA-TSS, is discussed in detail. This system contains knowledge about PLA TDMs, and works as a consultant to a designer in selecting a TDM for a PLA. Working examples of PLA-TSS will be given in Chapter 5.

Finally in Chapter 6, we summarize the results presented in this thesis and discuss further research problems related to TDM selection.

## Chapter 2

# Testable designs methodologies for PLAs

In order to understand the problems involved in selecting a TDM for a kernel, in this chapter we will conduct a case study of one type of kernel, namely the programmable logic array (PLA). We will first survey the available TDMs for PLAs, then abstract and define common attributes of TDMs, and finally characterize the TDMs for PLAs in terms of these attributes. This study will lay the basis for TDM selection.

One of the reasons for focusing on PLAs is that PLAs play an important role in digital system design. Since a PLA can implement any Boolean function, it has become a very useful device in the realization of both combinational and sequential logic circuits. In addition, a PLA's "programmable" nature allows for flexibility in making engineering changes. Its regular array structures result in high circuit density and easy layout generation, thereby reducing design cost and time. These advantages make PLAs especially suitable for VLSI circuit and computer aided design. Being a popular design style, PLAs have been widely used to replace random logic circuits, to build controllers, and for self-checking checkers [72, 79], as well

as entire systems [44, 47]. A second reason for focusing on PLAs is that there exist numerous TDMs for PLAs, as can be seen from Figure 2-1. We will show that these TDMs have different characteristics, hence selecting a suitable one is a real problem.

### 2.1. PLA models

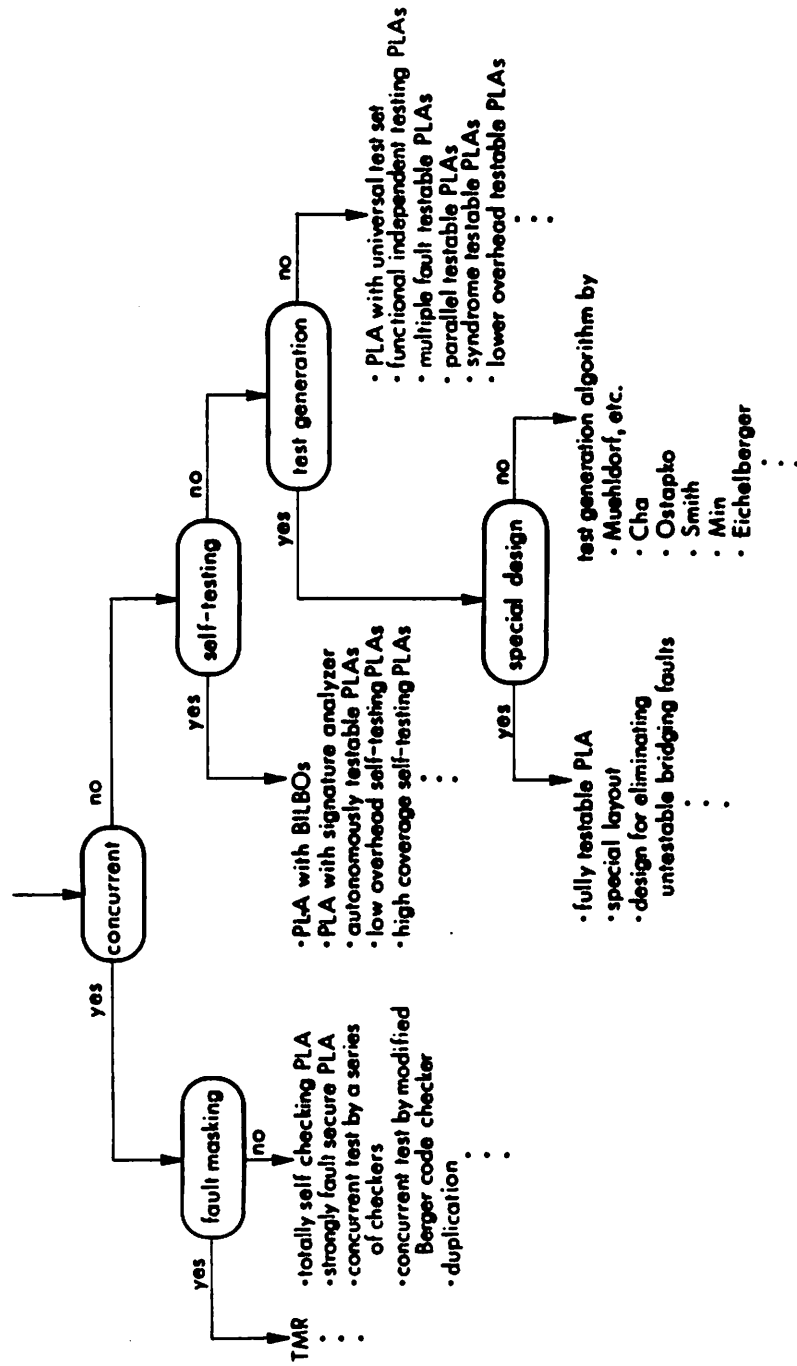
In general, a PLA consists of three parts: an input decoder, an AND array and an OR array. A basic PLA structure is shown in Figure 2-2.

The input decoder partitions the  $n$  primary inputs into  $2n$  bit lines. There are two common forms of input decoders: 1-bit decoders which transform each input  $x_i$  into  $x_i$  and its complement, and 2-bit decoders which transform a pair of inputs into the four minterms of the two variables, as shown in Figure 2-3.

PLAs are usually implemented in one of two ways, using either AND-OR or NOR-NOR logic, depending on the manufacturing technology. In case of a NOR-NOR implementation, e.g. in nMOS, an array of output inverters is also used. In the following, unless otherwise specified, we will assume that PLAs are implemented as NOR-NOR structures and use 1-bit decoders.

A PLA can be described by a matrix  $P = ( A, O )$ , where  $A$  is an  $m \times n$  matrix representing the AND array,  $O$  is an  $m \times k$  matrix representing

Figure 2-1: A taxonomy of PLA TDMs



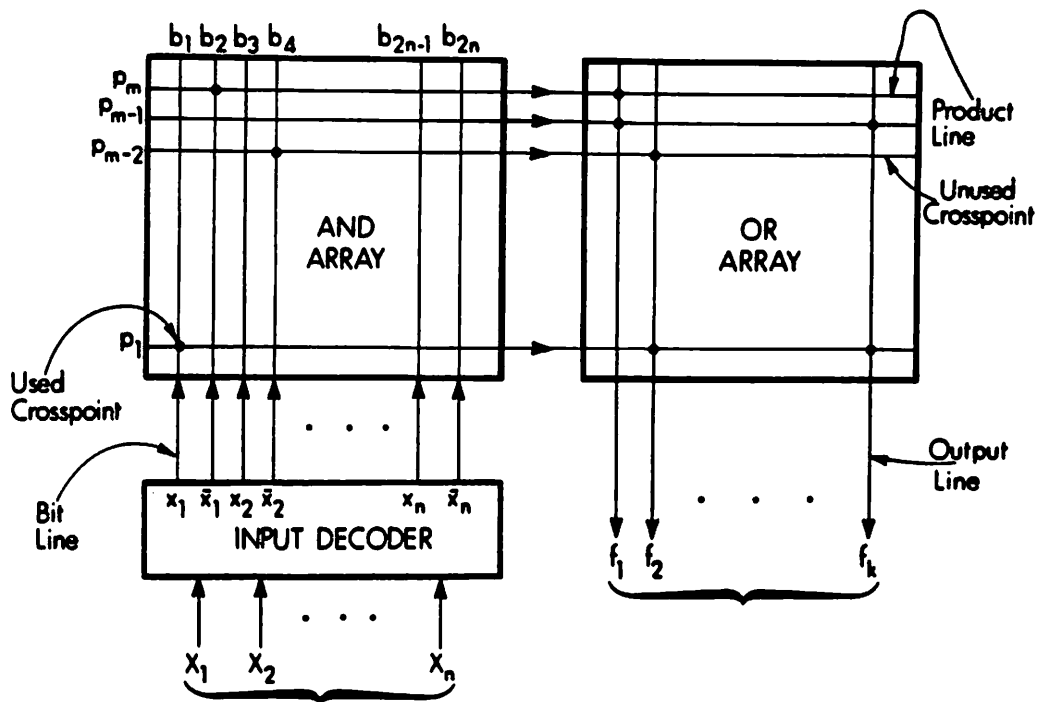


Figure 2-2: A general PLA structure

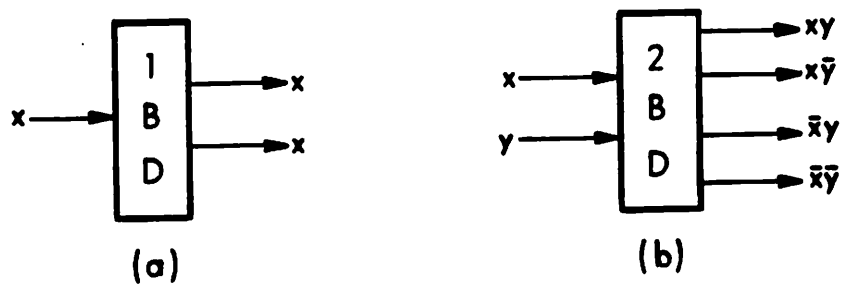


Figure 2-3: Input decoders

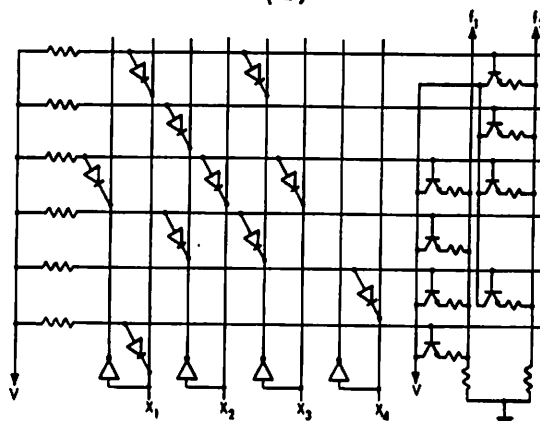
the OR array, and  $n$ ,  $m$ ,  $k$  are the number of inputs, product terms and outputs, respectively. This matrix is called the PLA's *personality*. The function realized by a PLA can also be represented by a set of *cubes*. Each cube corresponds to a row in the personality matrix. A small PLA example is given in Figure 2-4.

$$F1 = x1 + x4 + \bar{x}2\bar{x}3 + \bar{x}1x2x3$$

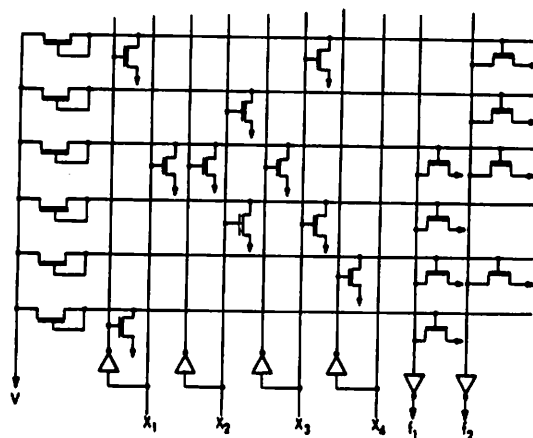
$$F2 = \bar{x}2 + x4 + x1\bar{x}3 + \bar{x}1x2x3$$

$$P = \begin{array}{c} \begin{array}{cccccc} & x1 & x2 & x3 & x4 & F1 & F2 \end{array} \\ \begin{array}{cccccc} 1 & x & 0 & x & 0 & 1 \end{array} \\ \begin{array}{cccccc} x & 0 & x & x & 0 & 1 \end{array} \\ \begin{array}{cccccc} 0 & 1 & 1 & x & 1 & 1 \end{array} \\ \begin{array}{cccccc} x & 0 & 0 & x & 1 & 0 \end{array} \\ \begin{array}{cccccc} x & x & x & 1 & 1 & 1 \end{array} \\ \begin{array}{cccccc} 1 & x & x & x & 1 & 0 \end{array} \end{array}$$

(a)



(b)



(c)

**Figure 2-4:** A PLA example

- (a) PLA personality matrix
- (b) The AND-OR implementation in DTL
- (c) The NOR-NOR implementation in nMOS

## 2.2. PLA testing problems

The widespread application of PLAs makes testing a PLA an important issue. Though PLAs offer many advantages, they also present new testing problems.

### 2.2.1. Fault models

In testing digital circuits, the most commonly considered fault model is the stuck-at fault. A stuck at 1 or stuck at 0 fault may occur on any wire within a PLA, including input/output lines, bit lines and product lines. Stuck-at faults alone cannot adequately model all physical defects in PLAs. Due to the PLAs' array structure, an additional class of faults, known as *crosspoint faults*, often occur.

A crosspoint fault is either an extra or a missing connection at a crosspoint in the AND or OR array of a PLA. There are four kinds of crosspoint faults [14, 56, 70]:

1. **Shrinkage fault:** an extra connection between a bit line and a product line in the AND array which causes the implicant to shrink because it is specified by one additional input variable.
2. **Growth fault:** a missing connection between a bit line and a product line in the AND array which causes the implicant to grow because it is independent of an input variable on which it should be dependent.

3. **Appearance fault:** an extra connection between a product line and an output line in the OR array which causes the corresponding implicant to appear in the output function although it is not part of that function.

4. **Disappearance fault:** a missing connection between a product line and an output line in the OR array which causes an implicant to disappear from the corresponding output function.

Note that missing crosspoints are equivalent to some stuck-at faults, but extra connections cannot be modeled by stuck-at faults, since the faulty signals at a crosspoint are not fixed at 1 or 0 but vary with the input values.

The compact array structure of PLAs makes shorts between two adjacent lines, the so-called bridging faults, more likely to occur. There is no efficient algorithm for generating tests for bridging faults in random combinational circuits.

Because of the way PLAs are fabricated, multiple faults are common. Multiple stuck faults, multiple crosspoint faults, or combinations of these faults may occur in newly manufactured circuits.

The various fault modes which may occur in a PLA can make test generation a complex process. However analysis of the relationships between different kinds of faults reduces the complexity of the problem. Smith [70], Cha [14] and Ostapko and Hong [56] proved that a complete test set for



single crosspoint faults also covers most single stuck faults in input decoders and output lines, as well as many shorts and a large portion of multiple faults. Min [51] presented a unified fault model and proved that any stuck-at fault or bridging fault (of AND type) is equivalent to a multiple crosspoint fault. Agarwal [4] considered the coverage of a single crosspoint test set for multiple crosspoint faults and verified that 98% of all multiple crosspoint faults of size 8 and less are inherently covered by every complete single crosspoint fault test set in a PLA. These results indicate that single crosspoint faults should be of primary concern in testing. In case other classes of faults are considered significant, special effort must be made to ensure for their high fault coverage.

### **2.2.2. Problems with traditional test generation methods**

A PLA corresponds to a two-level sum-of-product circuit with input inverters, although in nMOS technology it is often implemented by two levels of NOR-NOR gates with output inverters. One way to generate tests for a PLA is to first convert the PLA into a two-level gate circuit, and then find tests for stuck faults in the "equivalent" gate circuit. Many algorithms exist for generating tests for such circuits [12], e.g., the D-algorithm [12, 62]. However this method has two serious problems. First, although the two-level circuit is logically equivalent to the PLA, as far as fault behavior is concerned, they are not equivalent. Some faults in the PLA, such as an extra crosspoint fault in the AND array, cannot be modeled as a stuck fault

in the gate circuit. Therefore high fault coverage is not guaranteed. Secondly, traditional test generation algorithms are not always effective for PLAs because PLAs have high fan-in, fan-out, redundancy and special fault modes. Although exhaustive testing is not affected by these factors, it becomes less practical as the size of the PLA increases.

It has also been shown that PLAs are not efficiently tested by random test vectors [24, 81]. The main reason is that a PLA usually has a relatively large number of *used crosspoints* in the AND array. Suppose a product term  $P$  is connected to  $j$  bit lines ( $j \leq n$ ). To test a missing crosspoint fault at  $A(P, b_i)$ , one needs to place a 0 on the  $i$ th bit line and 1's at all the other  $(j - 1)$  connected bit lines. Since there are  $2^j$  possible patterns on these  $j$  bit lines and only one pattern can test this fault, the probability of detecting such a missing connection with a random pattern is approximately  $1/2^j$ . Since  $j$  is frequently 10 or larger, many random patterns may be needed in order to achieve high fault coverage.

### 2.3. Test Generation Algorithms for PLAs

Since conventional test generation methods are not suitable for PLAs, several *ad hoc* test generation approaches have been developed since the late 1970s [9, 14, 24, 52, 54, 56, 70, 80]. It has been shown that a PLA's regular structure leads to more efficient test generation and fault simulation algorithms than those for random logic.

### 2.3.1. Deterministic test generation

The basic idea behind most PLA test generation algorithms is path sensitization, namely, to select or deselect a product line and then sensitize the chosen product line through one of the output lines. In fact, a test must satisfy the condition that it matches or almost matches (only one bit differs from) a product term in the AND array. Knowing a PLA's personality, tests of this nature can be easily found.

The SHARP operation, denoted by "#",<sup>2</sup> is especially useful for finding such a test. For example, let  $c_i$  be the cube representation of product line  $P_i$ . For example, the cube representation for the term  $\bar{x}_1x_2x_3$  is 011x (see Figure 2-4). Suppose the connection between  $P_i$  and the  $j$ th input variable is missing thus resulting a growth fault. The cube  $c_i'$  representing the faulty  $P_i$  would be the same as  $c_i$  except the  $j$ th bit changes to x. To detect this missing crosspoint fault, a test  $t$  must be covered by  $c_i' \# c_i$ . Let  $z(i,k)$  be a set of cubes representing product lines which are connected to output line  $k$ , except  $c_i$ . To propagate the error through output  $k$ , the test  $t$  must also cause all product lines represented by  $z(i,k)$  to be 0. That is to say,

$$t \in (c_i' \# c_i) \# z(i, k).$$

If the result is empty, another  $k$  should be tried until a test  $t$  is found. If no test can be identified, the fault is undetectable. Formulas for generating tests for other crosspoint faults can be similarly defined [56, 70].

---

<sup>2</sup>By definition,  $a \# b = a \& \bar{b}$ .

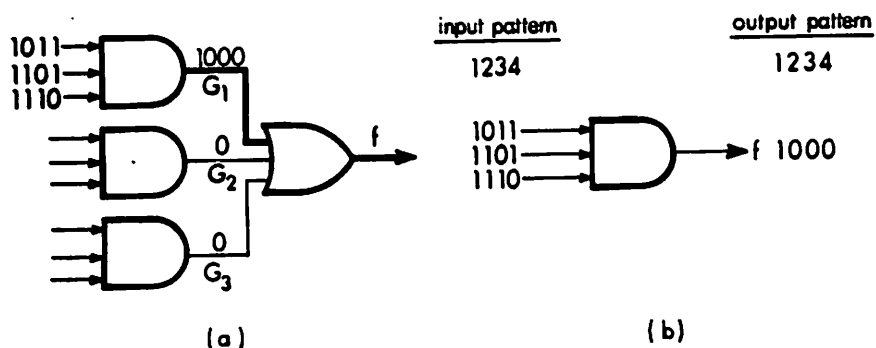
After a test is generated, to determine fault coverage test analysis is carried out instead of conventional fault simulation. A PLA's regularity makes it possible to directly determine what faults are detected by a given test. For example, if a test  $t$  results in only product line  $P_j$  being set to 1, and there is an output line  $q$  which does not connect to  $P_j$ , then an appearance fault at position  $O(j, q)$  can be detected by  $t$ , where  $O$  represents the OR array. Such rules of fault analysis can be formulated easily and work much faster than fault simulation for random logic. Details of test analysis for PLAs are given in [57, 70].

Muehldorf and Williams [55] developed a different approach for test generation. Only stuck faults are considered. They first generate an exhaustive list of tests for the PLA arrays and input decoders, and then reduce the list to a set of activity patterns by resolving conflicts and subsuming. Finally activity patterns are translated into input patterns.

Under the single fault assumption, most test generation algorithms can achieve high fault coverage for detectable stuck faults and crosspoint faults. AND-type shorts may also be covered by some algorithms [14, 57]. Min [52] proposed a test generation algorithm for irredundant PLAs which covers multiple crosspoint faults. The test generated automatically detect stuck-at faults and bridging faults. In general, the test set produced by these algorithms is bounded by the number of crosspoints in a PLA, e.g.,  $m * (2n + k)$  [70].

### 2.3.2. Semi-random test generation

Deterministic test generation for PLAs is feasible but laborious. Random patterns can be easily generated, but are often ineffective for PLAs. Eichelberger and Lindbloom [24] found that a PLA's regular structure allows deterministic and random test generation methods to be used together to form an effective and inexpensive test generation scheme. A PLA has a direct logic correspondence with a two level AND-OR combinational circuit with all inputs and their complements supplied. It is well known that the circuit shown in Figure 2-5 (a) can be tested, under the single stuck fault assumption, by applying a set of "embryonic" patterns to each AND gate (see Figure 2-5 (b)) and sensitizing the output of the AND gate through the OR gate. The embryonic patterns are hard to generate randomly. However it is very probable that a random pattern on G2 and G3 will result in a sensitized path for G1, since for an AND gate with  $i$  inputs,  $2^i - 1$  out of  $2^i$  possible input combinations lead to an output of 0.



**Figure 2-5:** Embryonic test patterns for an AND gate  
 (a) Embryonic patterns for an AND gate together with a sensitized path.  
 (b) Embryonic test patterns for an AND gate.

Based on these observations, a heuristic test pattern generation procedure can be specified which assigns embryonic test patterns deterministically, and assigns unspecified bits in the corresponding input vector arbitrarily. Such fully specified patterns are candidate tests. They are subject to *pattern evaluation* for determining if the random assignments provide for a sensitized path.

This heuristic test generation method is very fast. Compared with converting a PLA into a conventional gate circuit and then carrying out deterministic test generation, a speed-up of 1 : 6.5 is reported. If the conversion time involved is counted, this ratio becomes 1 : 20. A limitation of this heuristic is that it only deals with missing crosspoint faults. Fault coverage averaged 98.94% for 31 PLAs studied. All undetected crosspoint faults were found to be redundant.

Pseudo-random tests can be used for a PLA. Such tests are easy and fast to generate. Fault coverage for such tests cannot be easily determined and the resulting test sets are usually large. A newly developed PLA test generation package called PLATYPUS [80] uses a strategy which mixes biased random pattern generation and deterministic test generation. The biased random method is first used to generate tests for most of the faults. Then the uncovered faults are processed using a deterministic method. The test set is finally reduced by 20-30% using fault simulation. This mixed strategy efficiently produces a compact test set. Many powerful heuristics

and algorithms used in the PLA logic optimizer ESPRESSO-II [11] contribute to the efficiency of PLATYPUS.

The test generation methods discussed provide a software solution to the PLA testing problem. They do not require any hardware modification to the PLA. Since test generation and the application of large test sets are expensive, several alternative methods have been developed for testing PLAs.

#### 2.4. Testable PLA designs

As PLAs increase in size, a larger number of test patterns have to be generated and stored. Sophisticated automatic test equipment (ATE) is needed to execute the test process. Hence stored pattern testing becomes a time-consuming and expensive task. To alleviate this problem, more hardware oriented approaches have been developed which add extra built-in test (BIT) circuitry to the original PLA such that the modified PLA can be more easily tested. Most techniques reported to date fall into one of four categories, namely

- \* special coding,
- \* parity checking,
- \* divide and conquer, and
- \* signature analysis.

Some techniques are combinations of the above testable design philosophies. In the following, we will summarize basic principles of testable PLA design methodologies and give examples for each category.

### 2.4.1. Concurrent testable PLAs with special coding

Since PLAs have been used to implement important parts of digital systems, such as controllers in CPUs, concurrent checking is desirable in certain situations.

Error detection and correction codes are widely used in checking memories. A PLA's regular memory-like structure also suggests the application of special coding for either concurrent or off-line fault detection. The most useful code for PLA testing is the parity code. Since it has been the basis for a large number of testable PLA designs, we will discuss parity testing of PLAs in a separate section.

To concurrently test a PLA, it is required that the PLA should be non-concurrent [39, 71, 79], that is, during fault-free operation, only one product line can be activated by any input vector. Not every PLA has this property. However concurrencies in a PLA can be detected and removed using the procedure of Wang and Avizienis [79], which, unfortunately, increases the PLA's size. In this section we assume that a PLA has been made non-concurrent.



### 2.4.1.1. PLA with concurrent error detection by a series of checkers

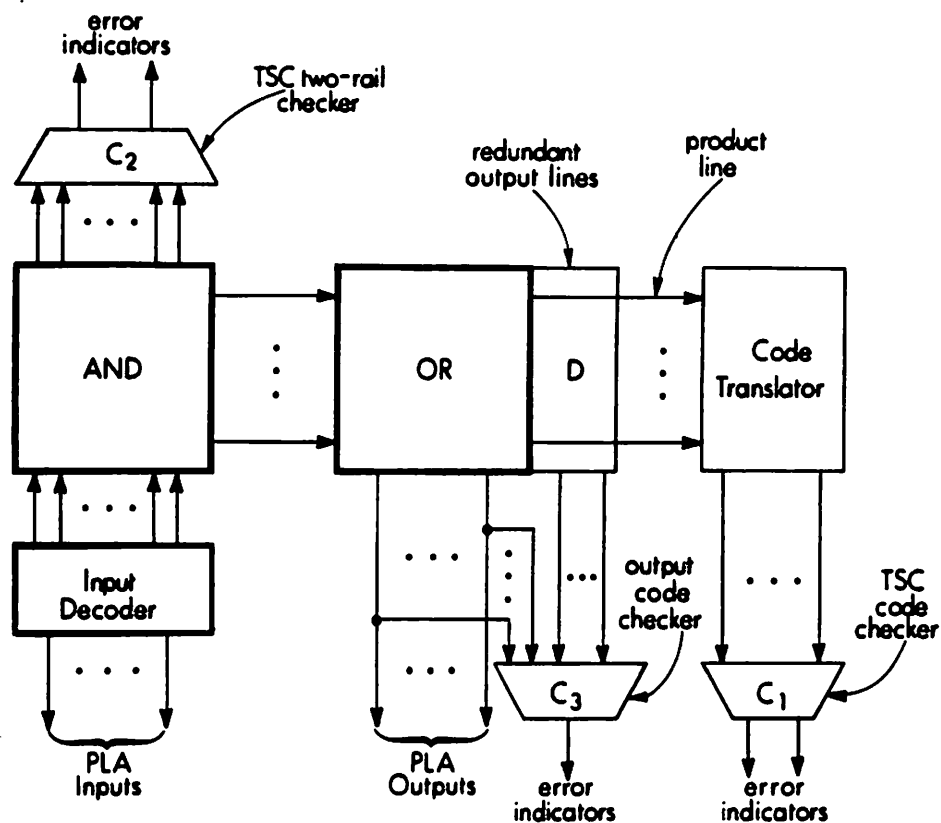


Figure 2-6: A concurrent testable PLA design

A technique proposed by Khakbaz and McCluskey [39] makes use of the following facts about a PLA.

- The bit lines in the AND array naturally form a two-rail code.
- For a non-concurrent PLA, during normal operation the signals on the  $m$  product lines form a 1-out-of- $m$  code.
- The fault-free output patterns are determined by the PLA's personality matrix. They can be coded into some error detection code by adding extra output lines to the OR array.

The proposed testable PLA has three checkers, as shown in Figure 2-6. C1 is a totally-self-checking (TSC) 1-out-of-m checker on all product lines, and detects any fault which destroys the non-concurrent property, such as a product line stuck at 1(0), or any missing and/or extra crosspoint in the AND array. C2 is a TSC two-rail checker which tests all single stuck at faults on the bit lines and input decoders. C3 is an output code checker. Its complexity depends on how the outputs are coded. The simplest code makes all output patterns have even (odd) parity. In this case, only one extra output line needs to be added, and C3 would be a parity checker. In general, C3 is not a TSC checker and may not be fully tested during normal operation, since the inputs to C3 are basically the PLA's outputs which are not arbitrarily controllable.

Testing occurs concurrently with normal operation. Most errors are caught by one of the three checkers. However the circuit is not totally self-checking. Therefore off-line testing is still needed to ensure high fault coverage. This technique combines concurrent error detection with off-line testing by using the same added circuits for both modes of testing. A simple test generation procedure which produces a complete test set has been developed, which is partially function-dependent, and covers faults in both the original PLA and the additional hardware [39].

### 2.4.1.2. Concurrent testable PLAs using Modified Berger code

Dong and McCluskey [19] proved an interesting property concerning PLAs, namely that the output errors of a PLA caused by single stuck-at and crosspoint faults and some multiple faults inside a PLA, except input stuck-at faults, are unidirectional<sup>3</sup>. Consequently, any codes that detect unidirectional errors, such as n-out-of-m [78] codes and Berger codes [8], may be used for the concurrent testing of PLAs. On this bases, a concurrent testable PLA can be designed which uses a parity checker and a two-rail code checker [39, 78] to detect input stuck-at faults, and a modified Berger code checker to detect other faults (see Figure 2-7).

Modified Berger (MB) codes [21] are separable codes in which each code word consists of a data part D and a check symbol C. Let  $O(D)$  be the number of 1's in D. Then a check symbol C for a MB codeword [D C] can be obtained by first setting  $C' = O(D) \bmod M$ , and then encoding  $C'$  into a codeword C that detects unidirectional errors. Here M is a predetermined integer and all unidirectional errors of size not equal to multiples of M can be detected by MB codes. By proper selection of M, all single faults in the AND and OR array can be detected.

Figure 2-7 shows the PLA design for this concurrent testing scheme. Suppose the number of inputs is odd, and the inputs are associated with a

---

<sup>3</sup>Errors are said to be unidirectional if they change a vector  $P = (p_1, \dots, p_n)$  to  $Q = (q_1, \dots, q_n)$  such that  $p_i \geq q_i$  or  $q_i \geq p_i$  for all i.

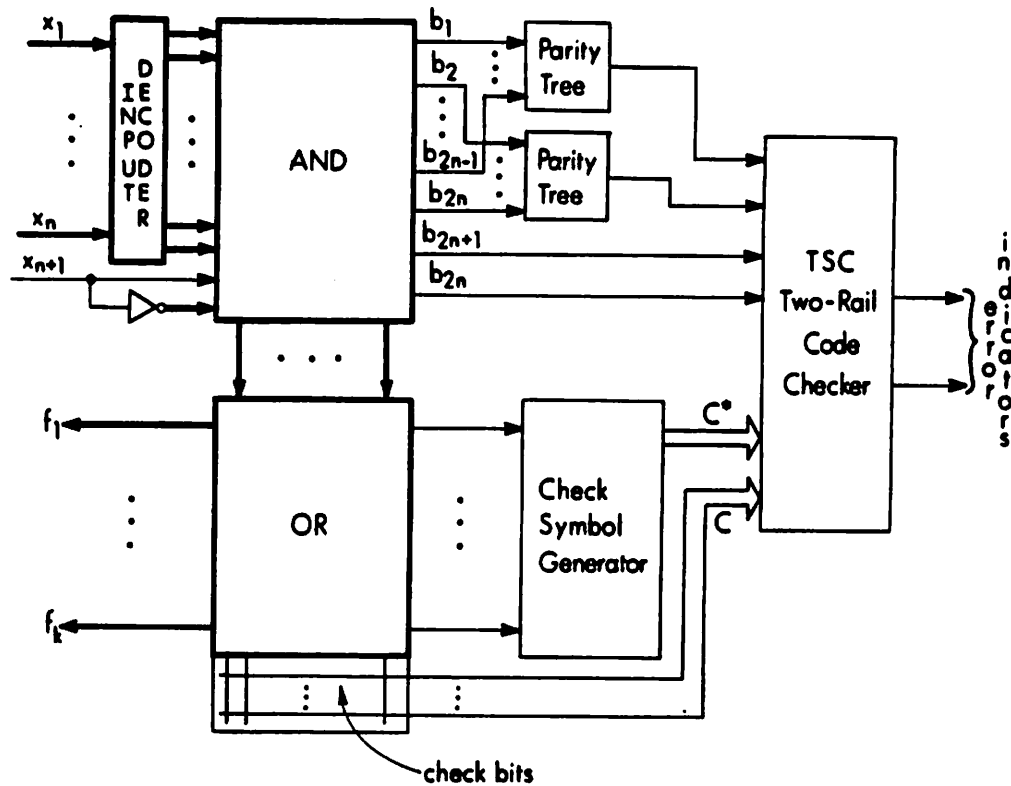


Figure 2-7: A PLA design for concurrent testing

parity bit. Faults on input lines, bit lines and input decoders are detected by a duplicated parity checker, in which one XOR tree generates the parity of all  $x_i$ 's and another generates the parity of all  $\bar{x}_i$ 's. During normal operation, the outputs of the two XOR trees are always complement values, which, together with the parity bit, are monitored by a TSC two-rail checker.

Since all other faults cause unidirectional errors on the outputs, the PLA's outputs are checked using a MB code. A check symbol is attached to every output pattern. During normal operation, an on-line generator [21] produces a check symbol  $C^*$  for each output pattern, which is

compared with the attached check symbol  $C$  using a TSC two-rail checker. Any disagreement between these two signals indicates some fault in either the PLA or the checking circuits.

This scheme is suitable for PLAs with a large number of outputs and product lines where each product line is shared by a small number of outputs, since the number of check bits required only depends on  $M$ . If  $O(D)$  is small, a small  $M$  can be chosen and hence area overhead is not too large. Also with a small number of code bits, the likelihood of every codeword being applied to the PLA increases, so that the totally self-checking property of the two-rail checker can be ensured.

#### 2.4.2. Parity testable PLAs

Among traditional testing techniques, exhaustive and random tests can be applied to any circuit, independently of the function realized, resulting in a large number of test patterns. Deterministic test generation methods reduce the number of tests, but tests need to be generated for each specific circuit. Since PLAs have a regular array structure, it is possible to design a PLA such that it can be tested by a small set of deterministic tests which are function-independent, i.e. independent of the personality matrix. This is possible because of two important observations:

1. Let  $N_i$  be the number of used crosspoints on bit line  $b_i$ . One can add an extra product line and make connections to it such that every bit line has an even(odd) number of connections with

product lines. Then any single crosspoint fault on  $b_i$  changes the parity of  $N_i$ . The same is true for output lines. Therefore single crosspoint faults can be detected by parity checking on the product and output lines.

2. To easily test a PLA, it must be possible to individually control each bit line and product line, and sensitize each product line through the OR array.

In order to do parity checking and line selection, some built-in hardware must be added to a PLA.

#### **2.4.2.1. PLA with universal test set**

Fujiwara and Kinoshita [25] developed one of the first easily testable PLA designs in which a PLA can be tested using a universal test set (UTS) which is independent of the PLA's personality. The augmented PLA is shown in Figure 2-8.

One column and one row are appended to the AND array and the OR array, respectively, so that each bit line in the AND array can have an odd number of connections and the portion of each product line in the OR array can have an even number of connections. Two parity checkers consisting of cascades of XOR gates are used to examine the parities of the two arrays during testing.

A selection circuit is used to activate the columns and rows in the PLA

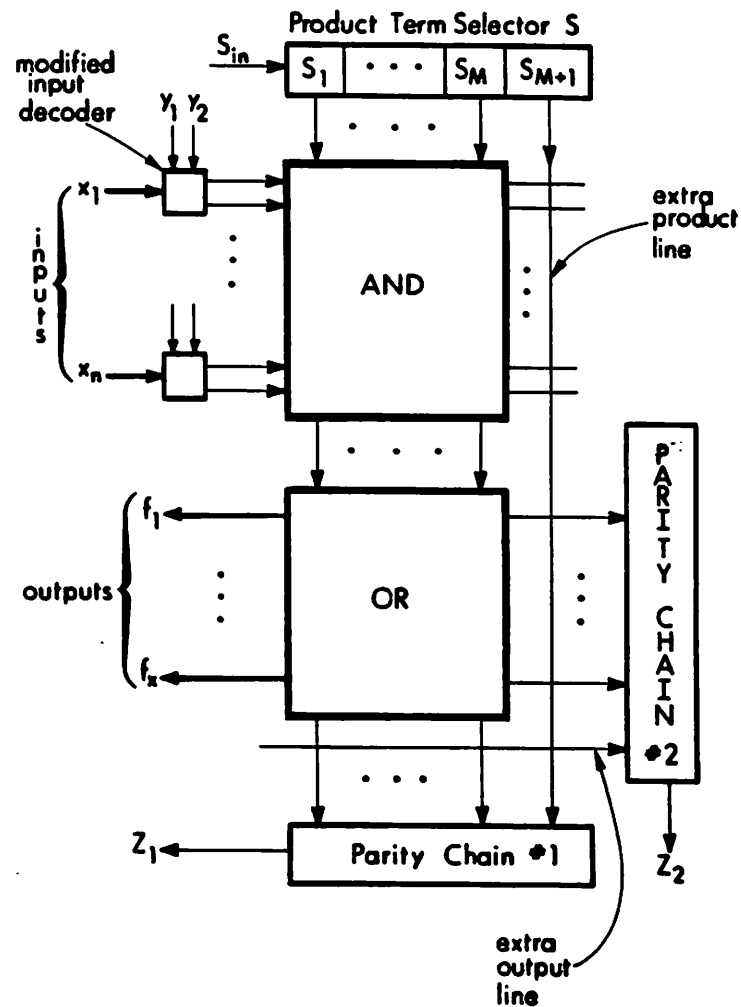


Figure 2-8: PLA with universal test set

so that every crosspoint can be uniquely selected and tested. In UTS, product line selection is accomplished by adding a shift register  $S$  as shown in Figure 2-8. Each bit  $S_i$  of  $S$  is connected to a product line  $P_i$  in such a way that the value of  $P_i$  is determined by  $P_i \times S_i$ . Thus if  $S_i = 1$  and  $S_j = 0$  for all  $j \neq i$ , product line  $P_i$  is selected and sensitized to the output, since all other product lines are 0.

Bit lines are selected by modifying the input decoder as shown in

Figure 2-9. By properly assigning values to  $y_1, y_2$  and the  $x_i$ 's, all but one input line can be set to 0. Thus that line is selected and tested.

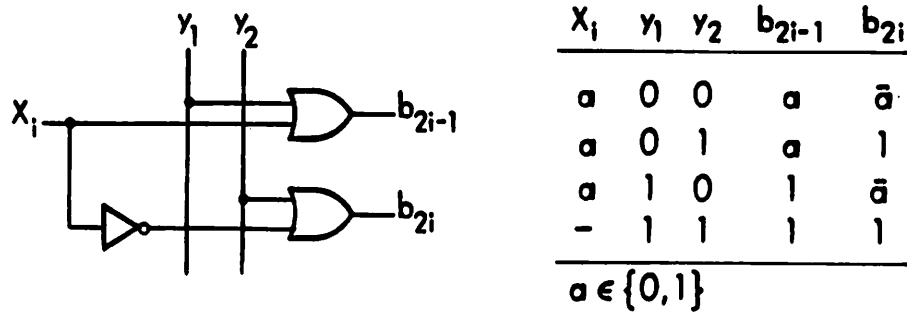


Figure 2-9: Augmented input decoder and its truth table

		$x_1 \cdots x_i \cdots x_n$	$y_1$	$y_2$	$s_1 \cdots s_j \cdots s_m$		$z_1$	$z_2$
$I_1$	-	-	-	-	0	0	0	0
For $j=1, \dots, m$								
$I_{j0}$	0	0	0	1	0	0	0	1
$I_{j1}$	1	1	1	0	1	0	0	1
For $i=1, \dots, n$								
$I_{i0}$	1	0	1	0	1	1	1	$e_m$
$I_{i1}$	0	0	1	0	1	1	1	$e_m$

$$e_m = \begin{cases} 0, & \text{if } m \text{ is odd} \\ 1, & \text{if } m \text{ is even} \end{cases}$$

"-" represents a don't care condition

Table 2-1: A universal test set for PLAs

Based on this augmented PLA design, a universal test set can be derived (see Table 2-1), which is applicable to all PLAs regardless of what function is being implemented [25]. This universal test set is easy to



generate, since it either applies all 1's or 0's to all bit lines and/or product lines, or shifts a 1 or 0 through the input lines or product lines. Basically this test set attempts to sensitize each product line and bit line, one at a time. For example, when the test  $I_{j0}$  or  $I_{j1}$  is applied, all bit lines and the  $j$ th product line are set to 1 and all other product lines are set to 0. Then the crosspoints along the  $j$ th product line in the OR array are tested by the parity chain connected to the output lines, and the result can be observed from  $z_2$ . Similarly, the AND array is tested by applying the  $I_{i0}$ 's and  $I_{i1}$ 's and observing the output  $z_1$ . All single stuck-at and crosspoint faults are detected by the parity checkers. This test set can also detect stuck-at faults in the input decoder and the added hardware. The size of the universal test set is linear with  $n$  and  $m$ .

Hong and Ostapko [33] independently proposed another design for function-independent testable PLAs (FIT) which employs the same philosophy as UTS. The test set for this design is essentially similar to that of UTS. As seen in Figure 2-10, the main differences between FIT and UTS are the following.

1. A control decoder rather than control gates is used to select and activate bit lines. Suppose 2-bit input decoders (IDs) are used. A control line is attached to each ID which either enables an ID or set all outputs of the ID to 0. To select each individual bit line,  $\lceil \log_2(n/2+1) \rceil$  extra inputs are needed to address one of the  $n/2$  2-bit input decoders, and certain

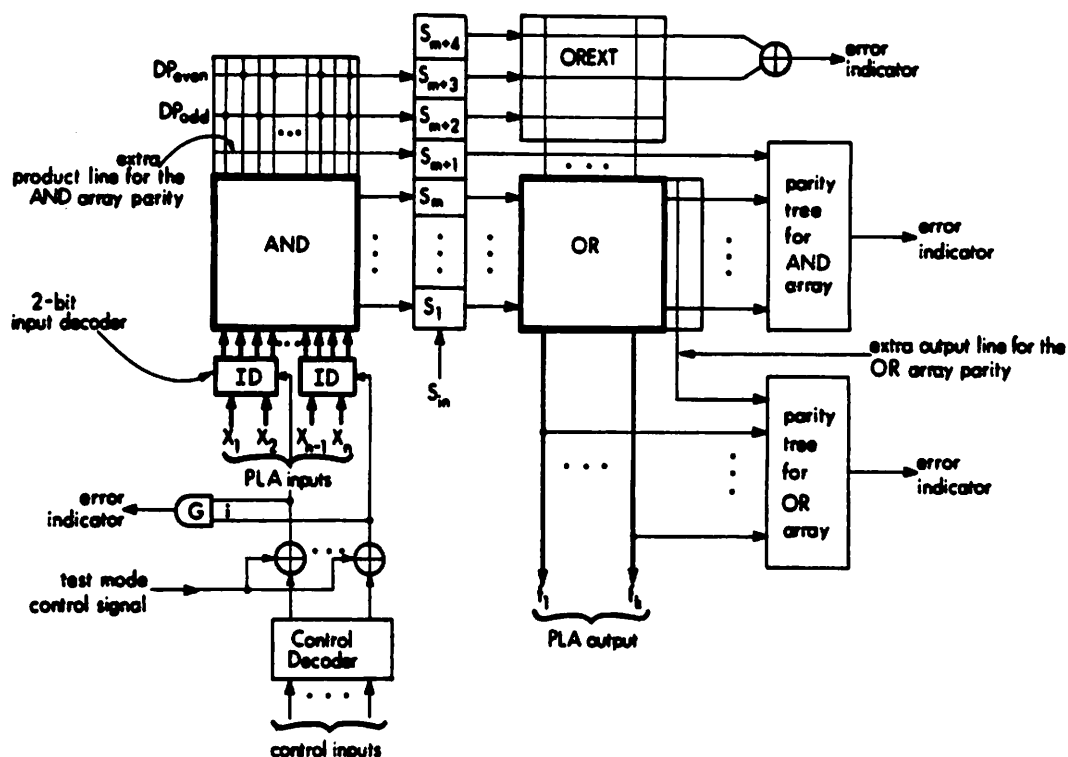


Figure 2-10: Function-independent testable PLA

primary input patterns need be applied. The control decoder is disabled during normal operation.

2. The parity checkers are made up of XOR trees rather than XOR chains in order to reduce logic delay in the test mode and thus decrease the test time.

3. It is well known that only four test patterns are necessary to test a parity tree consisting of 2-input XOR gates. Since there is no guarantee that these four patterns can be generated from the OR array of the PLA, more subsidiary circuits, such as the OREXT array, have to be added to generate tests for the OR parity tree.

4. The gate G is added in order to detect some faults in the control decoder. In addition, two product lines DP1 and DP2 are introduced which produce the OR functions of all even bit lines and odd bit lines, respectively. The valid value of the two lines are 01 or 10 during the AND array test mode. These signals can be checked from the EOR output of the OREXT array.

#### 2.4.2.2. Autonomously testable PLAs

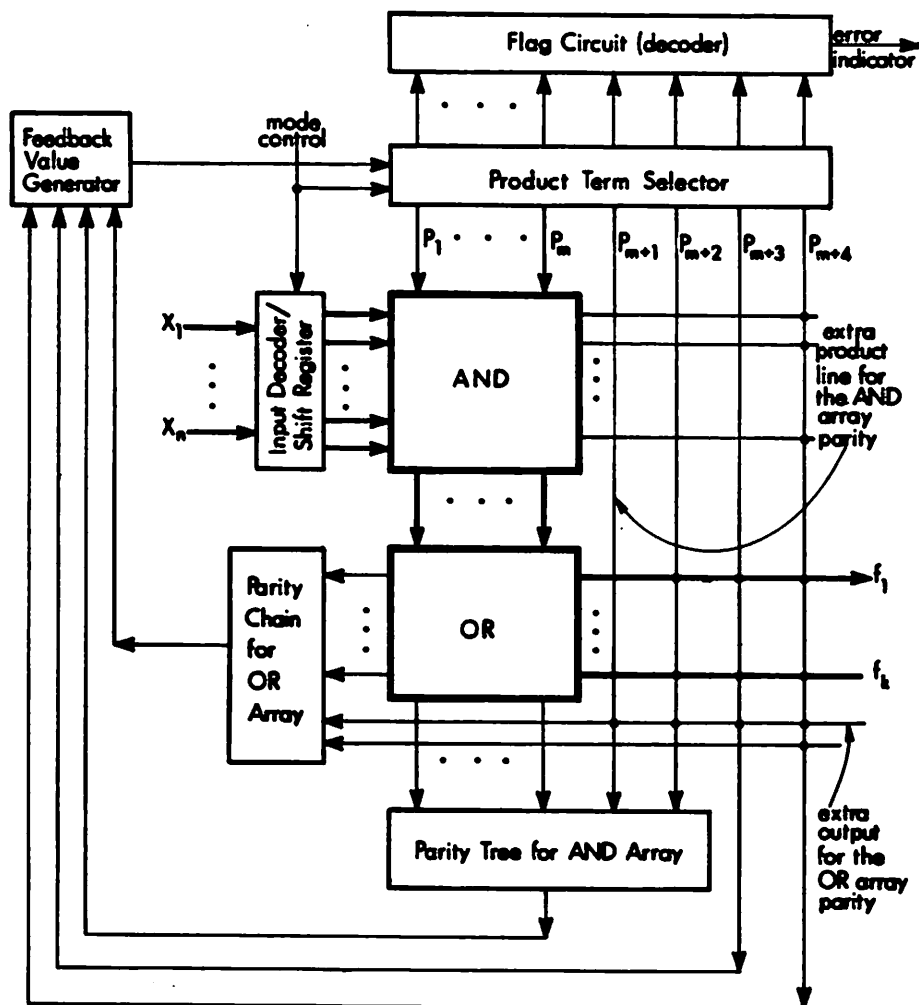


Figure 2-11: An autonomously testable PLA

For UTS or FIT type PLAs, the test vectors must be either stored or generated externally, and the test results have to be constantly monitored during the test process. Yajima and Aramaki [83] have augmented these schemes so that the resulting PLA circuits can generate the universal tests using a feedback value generator, a product term selector and an input shift register (see Figure 2-11). The test results are compacted using a feedback shift register which is also the product term selector, and evaluated by a built-in flag circuit, which is simply a decoder for a specific value, when the test operation is completed. Any single fault in the original PLA, and in most of the additional circuits, changes the final flag and therefore is detected. However some faults in the feedback value generator and the flag circuit are not covered, hence these circuits need to be duplicated in order to achieve a high fault coverage.

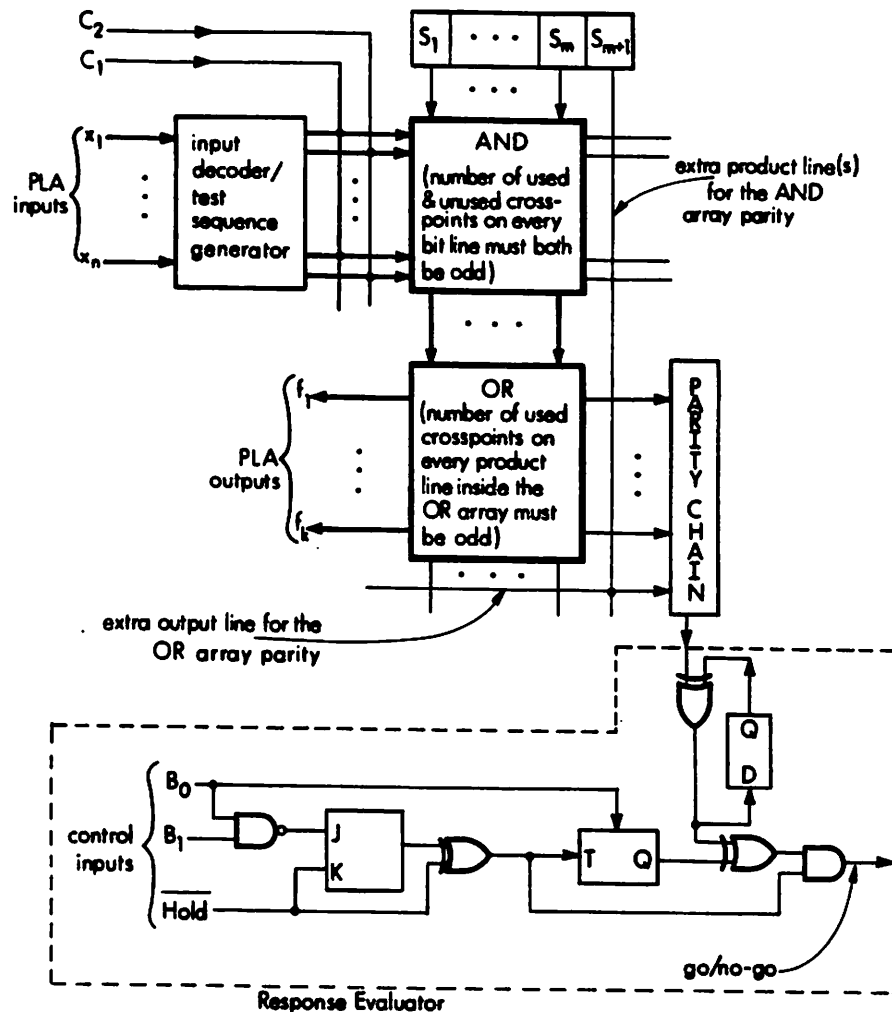
The advantages of an autonomously testable PLA are that test patterns need not be generated *a priori* nor stored, hence field testing becomes very easy. Since the tests are function-independent, any PLA can be modified in a uniform way.

### 2.4.2.3. A built-in self-testable PLA design with cumulative parity comparison

To obtain more efficient testable PLA designs, recent research has been focused on reducing area overhead and/or increasing fault coverage. Treuer et al. [77] proposed a design which employed the idea of parity compression discussed in [26].

In the testable designs discussed previously [25, 33], two parity checkers are used to monitor the parity of the two arrays. These checkers can be replaced by a *cumulative parity comparison* method, i.e., by accumulating parity signals in a flip-flop and comparing its value with expected values only at specific times.

This scheme is illustrated in Figure 2-12. Two control lines C1 and C2 are added to the AND array to disable all  $x_i$ 's and  $\bar{x}_i$ 's, respectively. C1, C2 and primary inputs can be used together to uniquely select each bit line. As before, a shift register is added to select product lines. One or two product lines are appended to the AND array so that every bit line has (1) an odd number of used crosspoints, and (2) an odd number of unused crosspoints. The same is done for the OR array. Area is saved by eliminating the parity checking circuit for the product lines. Only one parity chain is employed at the PLA's outputs, and cumulative parity comparison is used to detect faulty information.



**Figure 2-12:** A testable PLA with cumulative parity comparison

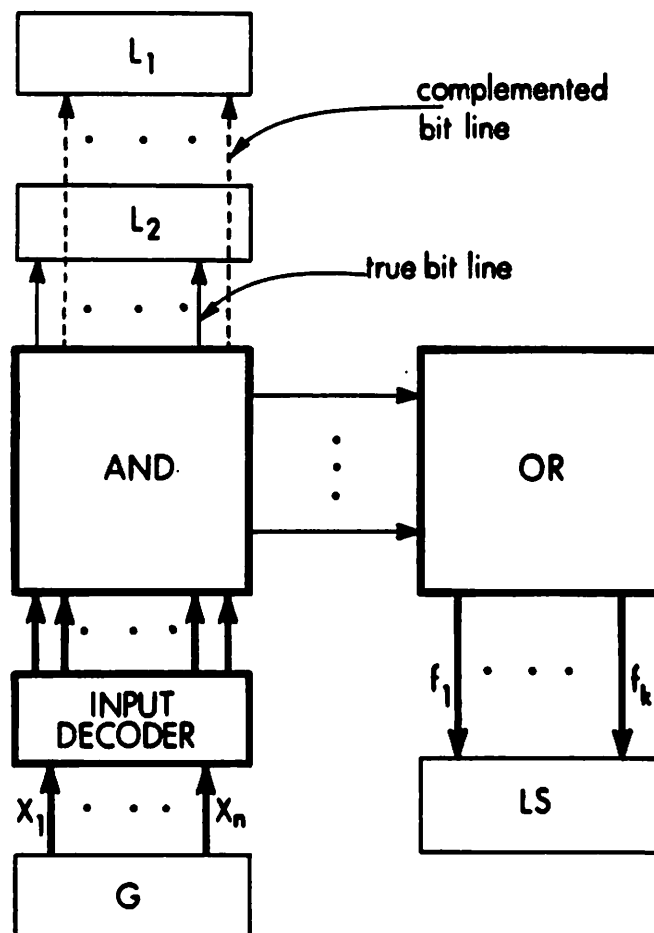
In the test mode, a universal test set of length  $2m(1 + n) + 1$  is applied to the inputs of the PLA. It is obvious that faults in the OR array can be detected by the parity checker on the outputs. The AND array is tested as follows. Suppose we select one bit line  $b_i$  and activate product lines one at a time. When  $P_j$  is activated, if there is a device connecting  $b_i$  and  $P_j$ ,  $P_j$  is forced to 0 and the output pattern should be all 0's, thus the parity value of the output is 0. If there is no device connecting  $b_i$  and  $P_j$ ,  $P_j$

is one and the output should have an odd parity. Since there is an odd number of used and unused crosspoints on each bit line, an odd number of 1's and 0's is produced from the output of the parity tree. This information is cumulated in a parity counter which produces the parity of the parity sequence. An interesting property of this scheme is that the sequence of cumulated parity bits at  $2n+2m+1$  selected check points is simply a sequence of alternative 0's and 1's. Hence it is very easy to generate the expected value on-line. The comparator is shown at the bottom of Figure 2-12. The proven fault coverage of this testable design is very high; all single and  $(1 - 2^{-(m+2n)})\%$  of all multiple crosspoint, stuck-at and bridging faults are covered.

#### **2.4.3. Signature testable PLAs**

Signature analysis has proven to be a simple and effective way for testing digital systems, and several self-testing PLA designs using this concept have been proposed. In these approaches, a set of input patterns is applied and the results are compressed to generate a "signature" which, when compared with a known correct value, determines whether or not the PLA is faulty.

## 2.4.3.1. PLA with multiple signature analyzers



**G:** maximum length sequence generator (LFSR)

**L1, L2:** n-input parallel signature analyzer with the same characteristic polynomial as G

**LS:**  $\lceil k/n \rceil$  independent n-input parallel signature analyzer each having the same characteristic polynomial as G

**Figure 2-13:** Self-testable PLA using signature analysis

As proposed by Hassan [31], at least four linear feedback shift registers (LFSR)  $L_1$ ,  $L_2$ ,  $G$  and  $LS$  having the same length and characteristic polynomial are added to the original PLA, as shown in Figure 2-13, to produce a testable PLA.  $G$  is used as a maximum length sequence generator for exhaustive input pattern generation, and the others for signature



analysis. The complemented bit lines are fed into L1 and the true bit-lines into L2. If the number of outputs of the PLA is greater than the number of inputs, more than one LFSR has to be used to obtain the output signature.

In the test mode, all possible input patterns are generated by G. Responses from the PLA are compacted in the three signature analyzers. The final signatures are shifted out for inspection. It has been shown that using this scheme, all multiple bit-line stuck-at faults, all output faults and most of the product line stuck-at faults are detected. This method is only practical for PLAs with a small number of inputs/outputs and a large number of product lines.

#### **2.4.3.2. Self-testable PLAs with single signature analyzer**

Grassl and Pfleiderer [30] proposed a simple signature test design for PLAs which attempts to reduce the number of test patterns and additional hardware, and also takes into account the impact on the PLA layout. It appears to be applicable to large PLAs.

Figure 2-14 shows the block diagram for this design. The input register is modified to be a test pattern generator during test mode. It shifts a single "1" across the bit lines. The selection of product lines is done by the shift register SHR and an additional selector SEL. The SHR selects a pair of adjacent product lines at a time, so its length is only half of the number of product terms. The SEL will in turn connect product lines 1, 3,

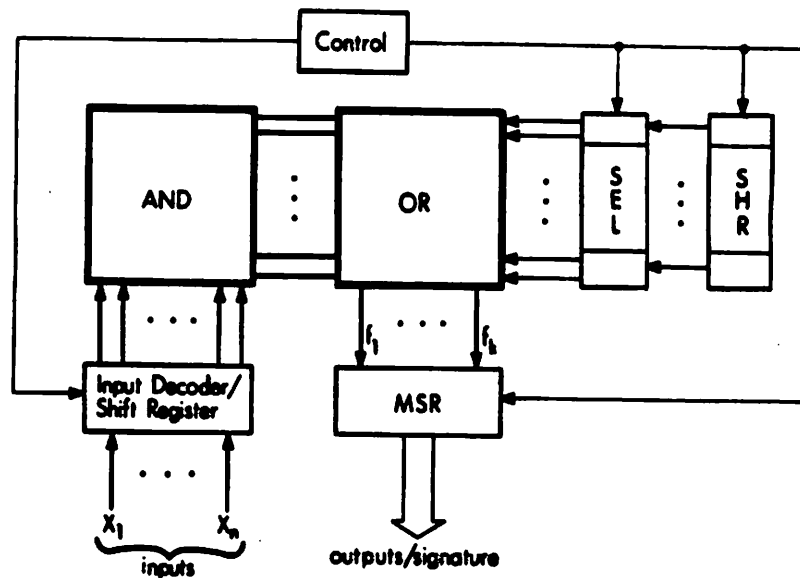


Figure 2-14: Block diagram of a self-testing PLA

5, ... or 2, 4, 6, ... to ground, thus resulting in a unique selection of each product line. Splitting the product line selector into SHR and SEL allows the added circuit to physically fit into the narrow pitch of the PLA arrays, substantially reducing the area overhead.

During testing, every crosspoint in the AND array is addressed using the two shift registers. The results are analyzed in the multiple input signature register MSR which acts as a conventional latch register during normal operation.

### 2.4.3.3. A self-testable PLA design based on crosspoint counting

In most built-in testable PLA designs, product line and bit line selection circuitry are required. One way to reduce area overhead is to simplify the response evaluation mechanism. Cumulative parity comparison [77] provides one solution. Saluja and Upadhyaya [66] proposed another technique based on counting the number of crosspoints on product lines.

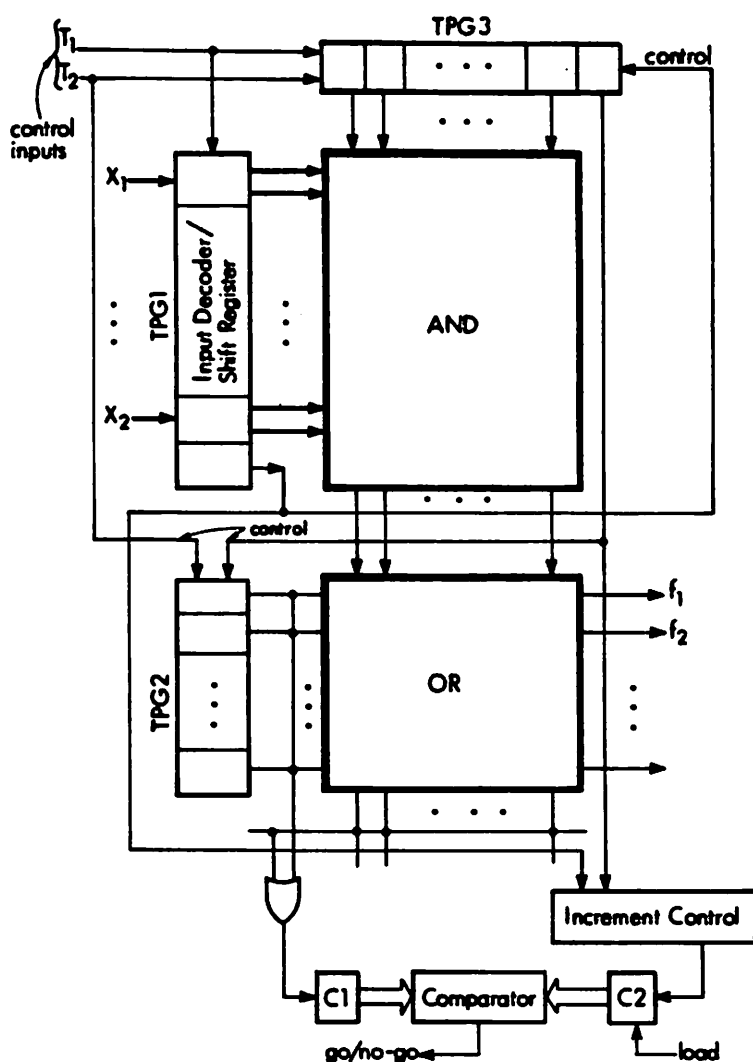


Figure 2-15: A built-in self-testable PLA using crosspoint counting

In the design shown in Figure 2-15, three shift registers are connected to bit lines, output lines and product lines, denoted by TPG1, TPG2, and TPG3, respectively. Two extra output lines are added, as well as a response evaluator consisting of two counters C1 and C2 and a comparator. Let  $g_i$  be the number of crosspoints on product line  $P_i$  in the AND array. The main concept in this scheme is to arrange the product lines such that the number of crosspoints on the lines form an ascending sequence. That is,

$$g_1 \leq g_2 \leq \dots \leq g_m \quad \text{and} \quad g_{j+1} - g_j \leq 1 \quad \text{for} \quad 1 \leq j \leq m-1 .$$

If this condition cannot be satisfied, for example,  $g_{h+1} - g_h = 2$ , a dummy product lines with  $g_h + 1$  crosspoints has to be added.

When testing the AND array, the counter C2 is initially loaded with the value  $g_1-1$ . The product line P1 is selected first, and the  $2n$  bit lines are activated in turn. The number of times P1 is 1 is counted using C1. At the  $(2n+1)$ th clock cycle, a 1 is shifted into the last stage of TPG1 and added to C2. Thus C2 equals  $g_1$  and comparison with C1 takes place at this moment. Any difference indicates a fault in the AND array or other logic. Then C1 is cleared, and the same process is repeated for the next product line. The additional output line z2 connects to a product line  $P_j$  if and only if  $P_j$  has the same crosspoint count as  $P_{j-1}$ . Signals on z2 are used to control the increment of C2 such that C2 always holds the correct reference value. The OR array is tested in the same way. The length of the counters is equal to  $\max\{\log_2 n, \log_2 m\}$ . The faults detected by this design include all single stuck and crosspoint faults and most multiple faults.

#### 2.4.4. Partitioning and testing of PLAs

Testing becomes more complex as the size of the PLA increases. A common strategy for dealing with large problems is that of divide and conquer. This principle has also been applied to the design of testable PLAs.

##### 2.4.4.1. PLA with BILBOs

Daehn and Mucha [17] suggested a partitioning approach for self-testable PLAs which partitions a PLA into three parts, an input decoder, an AND array and an OR array, and then inserts three BILBOs<sup>4</sup> between these parts as shown in Figure 2-16. Since BILBOs can be used for both test pattern generation and response evaluation, in the test mode the partitioned blocks can be tested separately by properly controlling the BILBOs. Testing of each block is done by letting the BILBO at its input operate as a test generator and the BILBO at its output operate as a signature analyzer. The final signature is shifted out for inspection. The three parts of the PLA are tested one by one.

After partitioning, the AND array and the OR array are just arrays of NOR gates. All inputs are controllable and outputs observable. Testing becomes a very simple task. It is well known that a NOR gate can be fully tested by a simple sequence such as

---

<sup>4</sup>BILBO stands for Built-In Logic Block Observer. A BILBO is a multi-function register which can behave as a regular register, a shift register, a pseudo-random generator or a signature analyzer. For details, see [42].

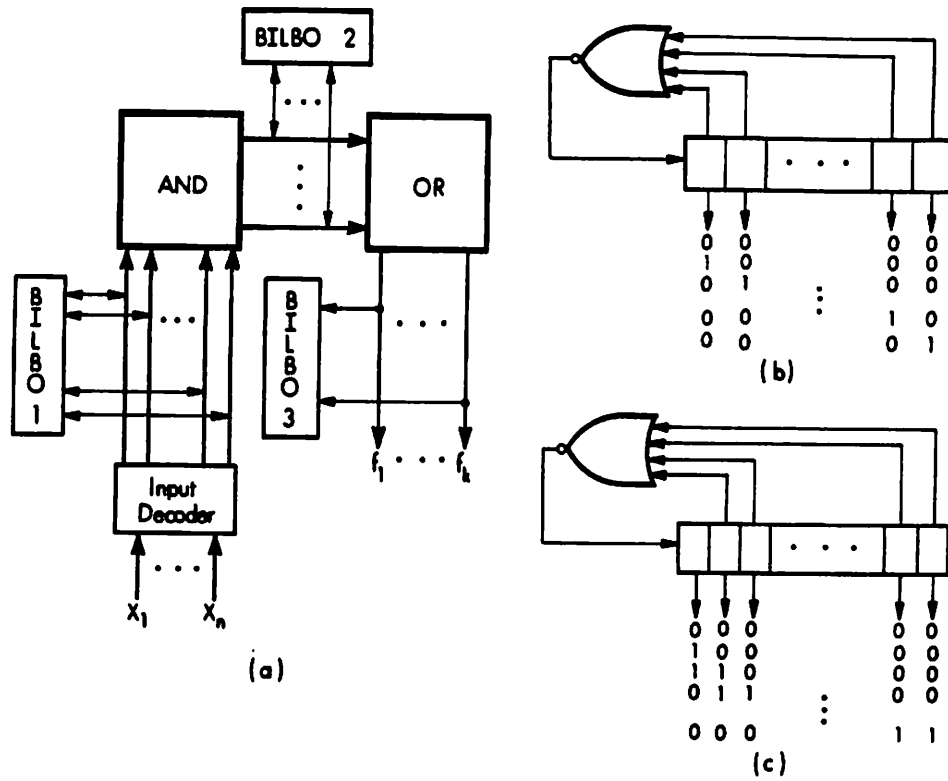


Figure 2-16: Testing a PLA using BILBOs

```

0 0 0 ... 0
1 0 0 ... 0
0 1 0 ... 0
0 0 1 ... 0
...
0 0 0 ... 1

```

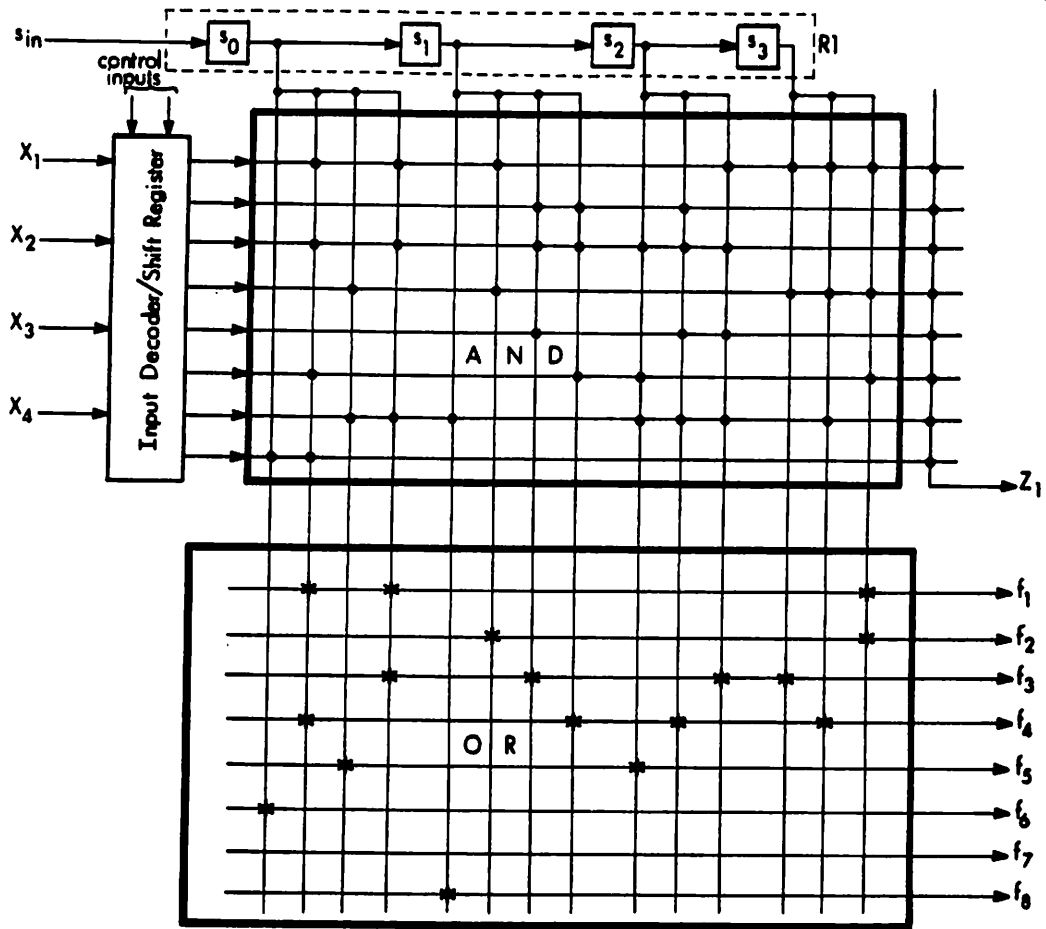
A NOR gate array can be tested by the same patterns. Hence the test generator need not be a pseudo-random generator producing all input combinations, but rather a non-linear feedback shift register as shown in Figure 2-16 (b) which produces the patterns given above. In this way, the number of test patterns is greatly reduced. This sequence can detect all single stuck-at faults, crosspoint faults and bridge faults in the AND or OR array. The faults in the input decoder are tested by a similar sequence generated by the non-linear feedback shift register shown in Figure 2-16 (c).

#### 2.4.4.2. Parallel testable PLAs

In most testable designs discussed so far, the product lines are tested individually one at a time. This simplifies testing and leads to high fault coverage. However an  $m$ -bit shift register is required and a long test sequence of the order of  $O(nm)$  may be necessary. Saluja et al. [69] has shown that by exploiting possible parallelism in testing product lines, high fault coverage can be maintained while area overhead and the number of test vectors are reduced.

The conditions for an entire PLA to be testable in the way referred to as parallel testable are too stringent to be satisfied by general PLAs. However it is possible to partition the product lines into groups such that each group is parallel testable. A shift register  $R_1$  is added to the circuit, each of its cells controlling one partitioned block of product lines. An output  $z_1$  is added which connects to all bit lines. Finally a  $2n$ -bit shift register and input control circuit  $R$  are inserted for controlling each bit line individually. An example of a parallel testable PLA is shown in Figure 2-17.

In the test mode, some patterns are first applied to the entire PLA, then the groups of product lines are tested in sequence. Within each group, product lines are tested in parallel. The length of the shift register used for selecting product lines is reduced from  $m$  to the number of groups, which is usually less than  $m/2$ . The number of test patterns is reduced as well because the tests applied to each group are basically the same as those



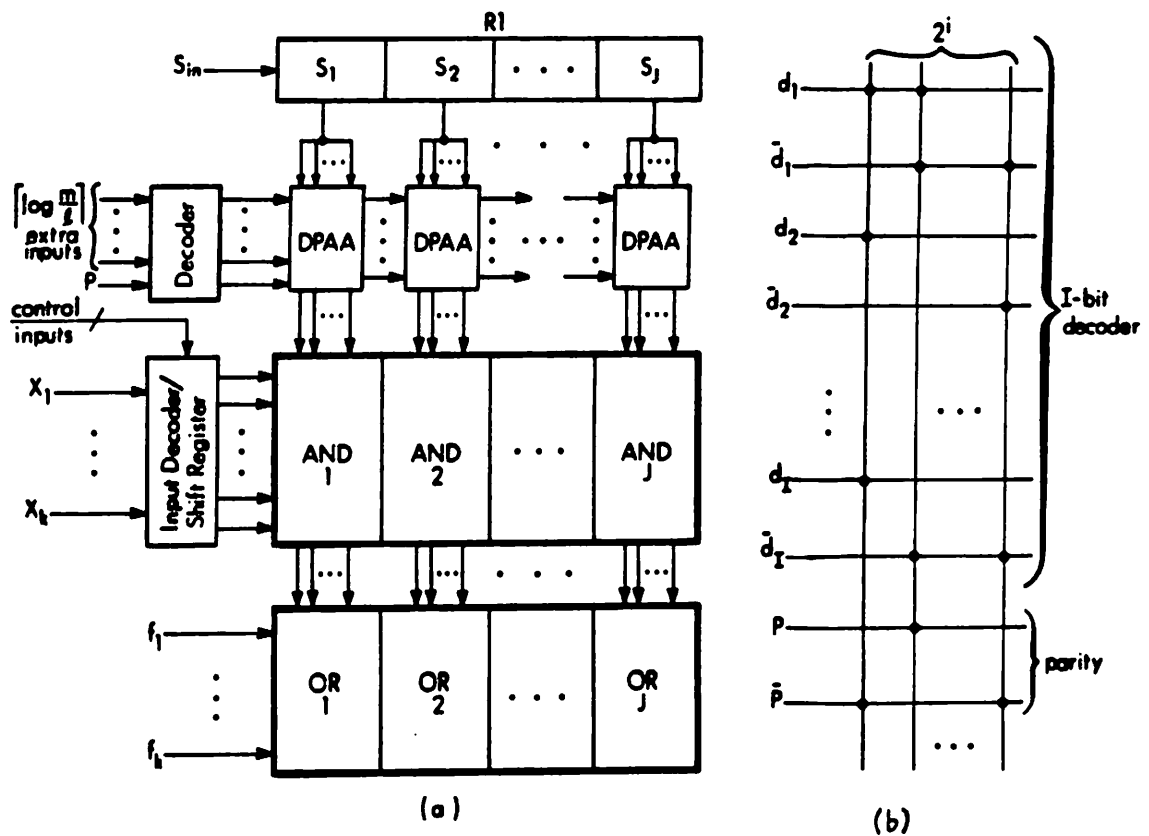
**Figure 2-17:** A parallel testing PLA structure

applied previously to each individual line. The test sequence is simple and universal, but the response is function dependent. It has been shown that all single and multiple stuck-at faults and crosspoint faults are detected using this technique.



**2.4.4.3. Divide and conquer strategy for testable PLA design**

The optimum partitioning of a PLA into parallel testable blocks is a difficult problem. Partitioning a PLA structurally into three parts and inserting BILBOs is easy but requires a substantial amount of overhead. Saluja and Upadhyaya [67] proposed a divide-and-conquer design which has lower overhead and is easily applicable to any PLA.



**Figure 2-18:** A testable PLA design with partitioning  
 (a) A testable PLA by divide-and-conquer  
 (b) Decoder-Parity AND array (DPAA)

Figure 2-18(a) illustrates this partitioning technique. The product lines are partitioned into J groups, and a J-bit shift register R1 is used to

control each group. Within each group, individual line selection is done by a decoder in the decoder-parity AND array (DPAA) [68], shown in the Figure 2-18(b). During testing, groups of product lines are tested one by one. Within a group, product lines are also tested sequentially. This design does not reduce the number of test patterns, but reduces the number of shift register cells for selecting product lines. This leads to a simpler layout. It has been proven that for this scheme, a PLA can be tested for all faults by a universal test set of length  $m(3+2n+\lceil\log(m/J)\rceil)+c$ , where  $c$  is a constant.

#### **2.4.5. Fully testable PLA designs**

Due to redundancy and concurrency in PLAs and the diversity of fault models, it is difficult to test a PLA and achieve high fault coverage without modifying the PLA design. The testable design methods (TDMs) discussed so far solve this problem by inserting a considerable amount of built-in test hardware. To design inherently testable PLAs, several methods have been proposed which aim at simplifying the test generation process or improving the fault coverage of existing test generation algorithms by employing little or no BIT circuits.

##### **2.4.5.1. A design of testable PLAs by specialized layout**

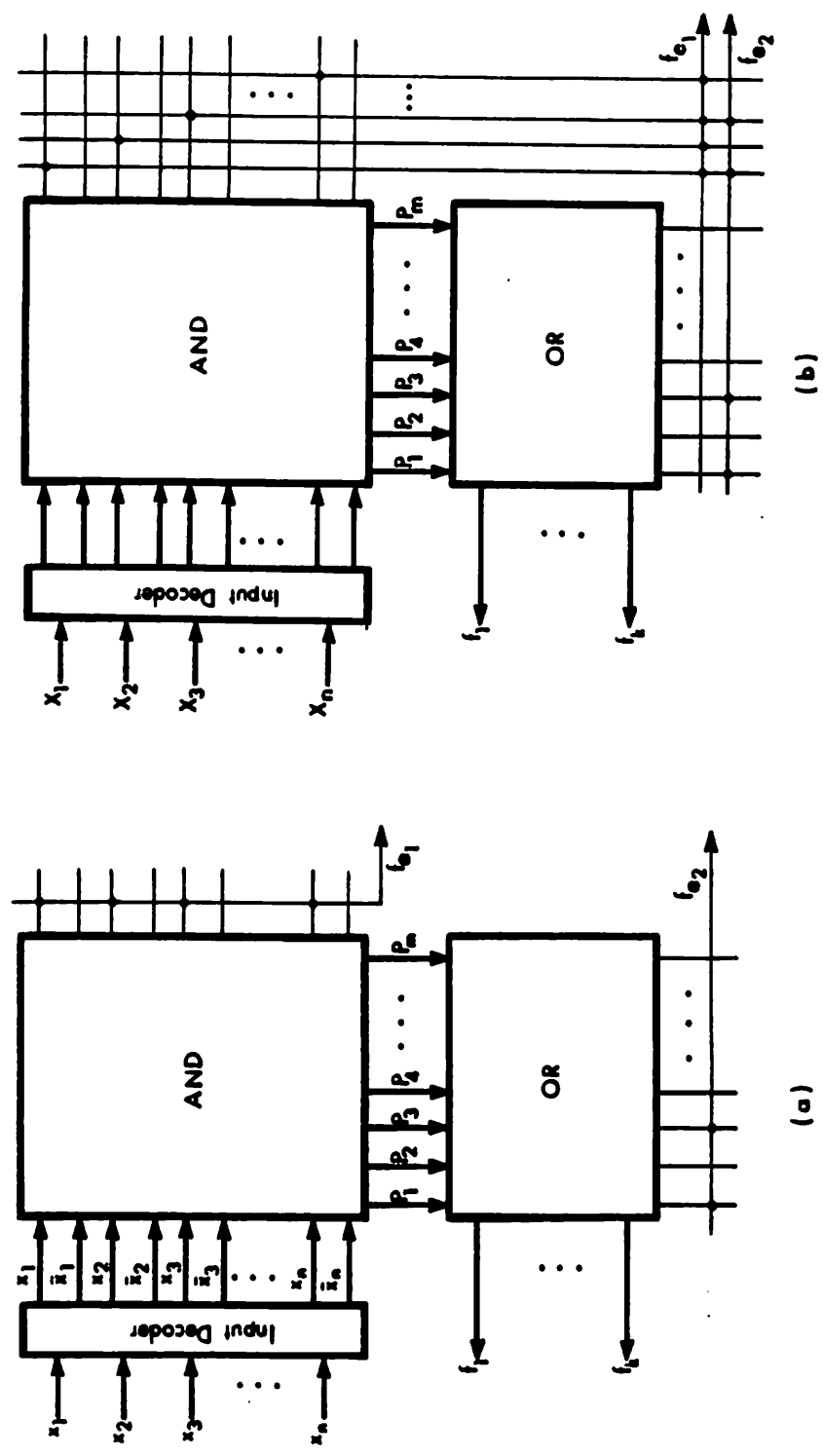
An example of this type of scheme has been proposed by Son and Pradhan [71]. The PLA is modified to be non-concurrent in the sense that the AND array only consists of mutually disjoint implicants of the function being implemented. The product lines of the PLA are arranged in such a

way that the connection patterns on adjacent output lines are distinct. If this cannot be done, an extra output (test point) should be added to make each pair of neighboring output lines differ from each other. It has been shown that PLAs designed in such a way have the nice property that the test set for crosspoint faults also covers stuck-at faults and bridging faults. A simple algebraic test generation algorithm exists which calculates tests for all crosspoint faults.

#### **2.4.5.2. A PLA design for testing single bridging faults**

Pradhan and Son [58] have also shown that there are some undetectable bridging faults in PLAs which can *invalidate* a test set for crosspoint faults. Thus removal of undetectable faults should be considered to ensure high fault coverage. Such undetectable bridging faults only occur between adjacent bit lines and adjacent product lines. They have developed a simple testable design for eliminating undetectable bridging faults. The designs shown in Figure 2-19 (a) and (b) are for the cases when the bridging faults produce AND functions and OR functions, respectively.

In the design of Figure 2-19 (a), all single bridging faults can be detected with  $m+1$  tests, as explained next. First apply all 1's to the inputs. The added test point  $f_{e1}$  should be 1, but any bridging fault between adjacent bit lines will change  $f_{e1}$  to 0. Then apply  $m$  tests  $t_1, t_2, \dots, t_m$ , where  $t_i$  sets product line  $P_i$  to a 1 and all other  $P_j$  ( $j \neq i$ ) to 0. The test point  $f_{e2}$  should be 1 for  $i$  odd. A bridging fault between  $P_i$  and  $P_{i+1}$  will



**Figure 2-19:** Testable design for eliminating untestable bridging faults  
 (a) Bridgings produce AND functions  
 (b) Bridgings produce OR functions

make  $f_{e2} = 0$ . A similar principle applies to bridging faults resulting in OR type functions. In that case,  $n$  extra product lines and two output check points must be added, as shown in Figure 2-19(b).

#### 2.4.5.3. Fully testable PLAs

Dong [20] has concluded that the existence of undetectable faults is basically due to redundancies in a circuit. Therefore, one way to make a PLA fully testable is to remove redundancies and add some check points. He has developed an algorithm which combines logic minimization and redundancy elimination to produce a minimized fully testable PLA. The resultant PLA is not necessarily non-concurrent. The fault models assumed are single stuck faults and crosspoint faults. Any test generation procedure for PLAs can be used to produce a test set.

#### 2.4.5.4. A design for complete testability of PLAs

The design in Figure 2-19 makes all single bridging faults testable. However it does not eliminate undetectable crosspoint faults and stuck faults. Ramanatha and Biswas [59, 60] studied this problem and proposed a design for complete testability of PLAs, that is, a design in which all crosspoint faults, stuck faults and bridging faults are testable.

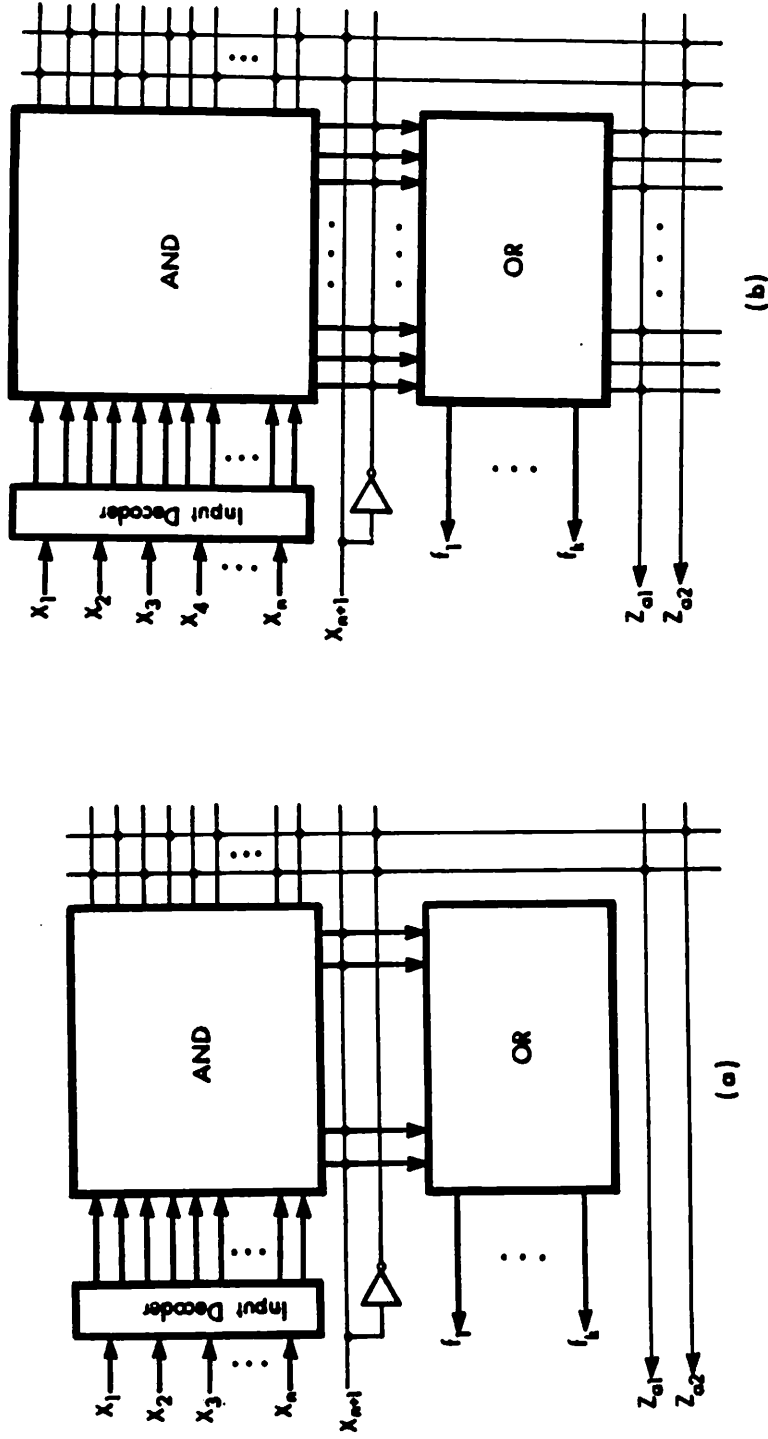
To achieve complete testability, a PLA is first converted into a *crosspoint-irredundant PLA*, namely, into a PLA in which all the crosspoint faults are detectable. Any PLA can be made crosspoint-irredundant by adding control inputs. The completely testable design for crosspoint

irredundant PLAs is given in Figures 2-20 (a) and (b), for AND-type bridging faults and OR-type bridging faults, respectively.

This completely testable design also has the advantage that a test set for single crosspoint faults is sufficient to test all single stuck-at faults and bridging faults as well. The complexity of test generation in order to achieve high fault coverage is reduced.

#### **2.4.5.5. A PLA design for ease of test generation**

Min has identified sufficient conditions for a PLA to be fully testable [53]. If a PLA meets those strict conditions, it can be completely tested by  $2n + C$  tests, where  $C$  is the number of used crosspoints in the AND array. The test set is function-dependent but can be easily generated from the PLA's personality. Fault coverage of this test set is 100% for all permanent stuck faults, crosspoint faults, bridging faults and combinations of these faults. However considerable effort and overhead are required to modify a given PLA so that it satisfies the required conditions. For the example given in [53], a PLA with 6 inputs, 6 product terms and 6 outputs is enlarged to 10 product terms and 10 outputs. That results in an area overhead of 54%.



**Figure 2-20:** Complete testable design (CTD)  
 (a) CTD when the bridging faults are of AND-type  
 (b) CTD when the bridging faults are of OR-type

### 2.4.5.6. Low overhead design of testable PLAs

Most testable designs of PLAs use a shift register to select individual product lines. An  $m$ -bit shift register takes up a significant amount of silicon area. The need for a shift register is due to concurrency inherent in most PLAs, i.e., an input pattern may activate two or more product lines. Bozorgui-Nesbat and McCluskey [10] proposed a novel way to deal with this problem: unique selection of each product term can be done by increasing the Hamming distance among product terms. The Hamming distance between product terms  $P1=(p11, p12, \dots, p1n)$  and  $P2=(p21, p22, \dots, p2n)$  is the number of bits in which  $p1i=1$  and  $p2i=0$  or  $p1i=0$  and  $p2i=1$ . For example, if  $P1 = 111x$  and  $P2 = 1x0x$ , the Hamming distance between  $P1$  and  $P2$  is 1. It is easy to see that if the Hamming distance between any pair of product terms is at least 1, any input patterns which activate product term  $P_j$  will not activate any other  $P_i$  for  $i \neq j$ .

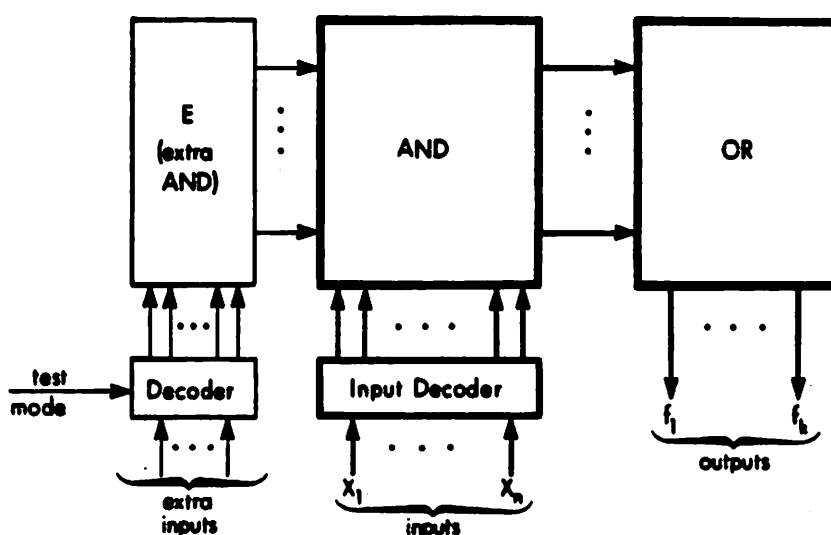


Figure 2-21: A testable PLA design



In the proposed TDM shown in Figure 2-21, extra inputs are added such that the Hamming distance between any two product terms is at least 2. Then a main test pattern and  $n$  auxiliary test patterns are applied to each product line. For  $P_i$ , the main pattern is a completely specified input that activates  $P_i$ , and the  $n$  auxiliary patterns are those inputs which have exactly one bit different from the main pattern. All main test patterns are found directly from the PLA's personality. Auxiliary patterns can be easily generated. By construction, the Hamming distance between any two main patterns for different product lines is at least 2. Since an auxiliary pattern has only one bit different from a main pattern, the Hamming distance between any auxiliary pattern for  $P_i$  and a main pattern for other  $p_j$  is at least 1. Therefore when testing  $P_i$ , no fault-free  $P_j$  ( $j \neq i$ ) can be activated, i.e., each product line can be uniquely selected and tested in the test mode. In this TDM any number of single mode stuck-at faults, missing crosspoint faults or extra crosspoint faults can be detected. Since no extra circuit except control input lines are added, area overhead is relatively low. However the problem of finding the minimal number of extra inputs and assigning connections to product lines is NP complete. A heuristic procedure for this problem is described in [10], which also requires extensive computation.

Fully testable (FT) PLAs are alternatives to BIT PLAs. They offer high testability by changing the logic design of the PLA and adding check

points. Since no or little change is made to the PLA's physical structure, they do not create any layout problem. However considerable effort has to be extended to convert a given PLA into a FT PLA. Software tools for logic modification and test generation are required. The amount of overhead involved in constructing a fully testable PLA is function dependent, and may be quite high.

## 2.5. Evaluation of PLA TDMs

### 2.5.1. Measures of TDMs

As we have seen there exist numerous TDMs for PLAs. Each TDM has its advantages and disadvantages. There is a number of common measures dealing with various performance criteria of DFT techniques which help in their characterization and evaluation. These measures can be classified into four categories, namely (1) testability characteristics, (2) resulting effect on the original design, (3) requirements on application environment and (4) design costs. Testability characteristics specify what degree of testability a TDM can achieve, for example, what type of faults are detected and the fault coverage, as well as if it supports fault masking, concurrent testing or self-testing. Making a circuit easily testable is usually accomplished by modifying the circuit design. This may affect several factors in the original design, such as the delay under normal operation, area overhead, and extra I/O pins. Most TDMs cannot operate alone without additional support and control. Requirements on the application

environment specify the necessary and alternative hardware required to support a TDM in a real-time test process. Design costs indicate how difficult it is to implement a TDM, e.g. the extent of difficulty in logic modification and layout. In the following, we will discuss each of these TDM measures in detail and illustrate how to obtain specific values for these measures.

#### 2.5.1.1. Testability characteristics

1. **Fault model:** The commonly considered fault models for PLAs are single and multiple stuck-at faults, single and multiple crosspoint faults, as well as single and multiple bridging faults between adjacent bit lines, product lines and output lines. Each TDM deals with some specific type of faults. For example, parity schemes detect all single crosspoint faults but are not designed for testing multiple crosspoint faults, since multiple faults which occur on an even number of crosspoints of the same line cannot be detected. Fault models for a TDM specify what kinds of faults are covered by the TDM.

2. **Fault coverage:** Fault coverage is closely related to fault models. If only one fault model FM1 is used, fault coverage is the probability of a fault in FM1 being detected. If several fault models FM1, FM2, ..., FMj are considered and their fault coverage is fc1, fc2, ..., fcj, respectively, the total fault coverage is given by the formula

$$FC = \frac{\sum_i \text{# of faults in FM}_i \times fc_i}{\sum_i \text{# of faults in FM}_i}$$

Note that we only take into account the *proven* fault coverage for each TDM although a TDM may also detect other classes of faults. For some TDMs, there are no theoretical or experimental results dealing with fault coverage. For such cases an estimate of fault coverage must be used. For example, for signature analysis we will assume a coverage of 99% of single stuck-at faults. For simplicity, we have not weighted the fault classes based upon their probability of occurrence.

**3. Fault masking:** A TDM is fault masking if it has the ability to mask certain types of faults, that is, if it ensures correct outputs even when such a fault occurs. Examples of fault-masking TDMs are those which use error-correcting codes, triple module redundancy (TMR), and totally self-checking designs. The TDMs for PLAs discussed in this thesis are not fault masking.

**4. Concurrent testing:** Testing can be carried out on-line or off-line. A TDM uses concurrent testing if errors can be detected on-line, i.e. during normal operation. Non-concurrent testing TDMs require testing to occur in a test mode when the circuit under test (CUT) is not being used for normal operation.

**5. Error latency:** Physical failures in a piece of hardware cause errors in signals generated by that hardware device. For a concurrent testable TDM, error latency refers to the time between the first occurrence

of an error within the circuit and the detection of the error at an error indication signal line. The latent period is mainly caused by delay in the checking circuit and is measured in terms of gate delays. For non-concurrent testing TDMs, error latency is not defined.

**6. Self testing:** A TDM which is self-testing can, while off-line, test a CUT without requiring an external source of test vectors or response analysis. The test process is carried out by built-in test (BIT) hardware. For example, in Figure 2-15, during the test mode, test patterns are generated internally by three shift registers, and responses are evaluated by the built-in counters and the comparator. The only outside aid required is to control signals  $t_1$  and  $t_2$ , load the contents of counter 2, and check the output of the comparator at proper points in time.

**7. Function independence:** A TDM is function-independent if it works on any applicable CUT in a uniform way regardless of the function realized by the circuit. A function-independent test set results in the same fault coverage for all CUTs of the same type. A sequence of responses to a test set is function-independent if it is the same for all CUT under fault-free condition. An example is the autonomously testable PLA shown in Figure 2-11. The built-in flag circuit decodes the contents of the product line selector. At the end of a testing process, if the PLA is fault-free, the final signature contained in the product line selector should consist of 0's only, and the flag circuit should produce a 1. Otherwise it generates a 0. The

correct responses given by the output of the flag circuit should be 00...001, no matter what function is implemented by the PLA. A TDM may be totally function-independent or partially function-independent. For example, in exhaustive testing, the test set is function-independent because it consists of all input combinations and produces 100% fault coverage for permanent faults. However, its responses are function dependent. TDMs which are not function-independent are said to be function dependent. TDMs which rely on test generation are all function dependent.

#### **2.5.1.2. Resulting effect on the original design**

**1. Logic overhead:** Logic overhead refers to the amount of extra logic added for testability. It is often measured in terms of the number of extra transistors used in the BIT circuitry. The basic method of calculating this measure is by counting how many building blocks of BIT circuit are used, and the number of transistors in each building block. Logic overhead affects power consumption and reliability. It also increases the number of possible faults. For PLAs, there is another kind of logic overhead: extra bit lines, product lines or output lines. Each extra product line creates sites for at least  $2n+k$  single crosspoint faults and numerous multiple faults.

**2. Extra I/O connections:** Higher testability generally requires more accessibility to the CUT. Additional I/O connections are often required for observability and controllability. Depending on whether or not the testable PLA interfaces with other chips or with control circuits on the

chip itself, these I/O connections may translate into extra I/O pins on the chip itself. If several testable PLAs exist on a chip, then it is often possible for these PLAs to share common test hardware and/or I/O pins.

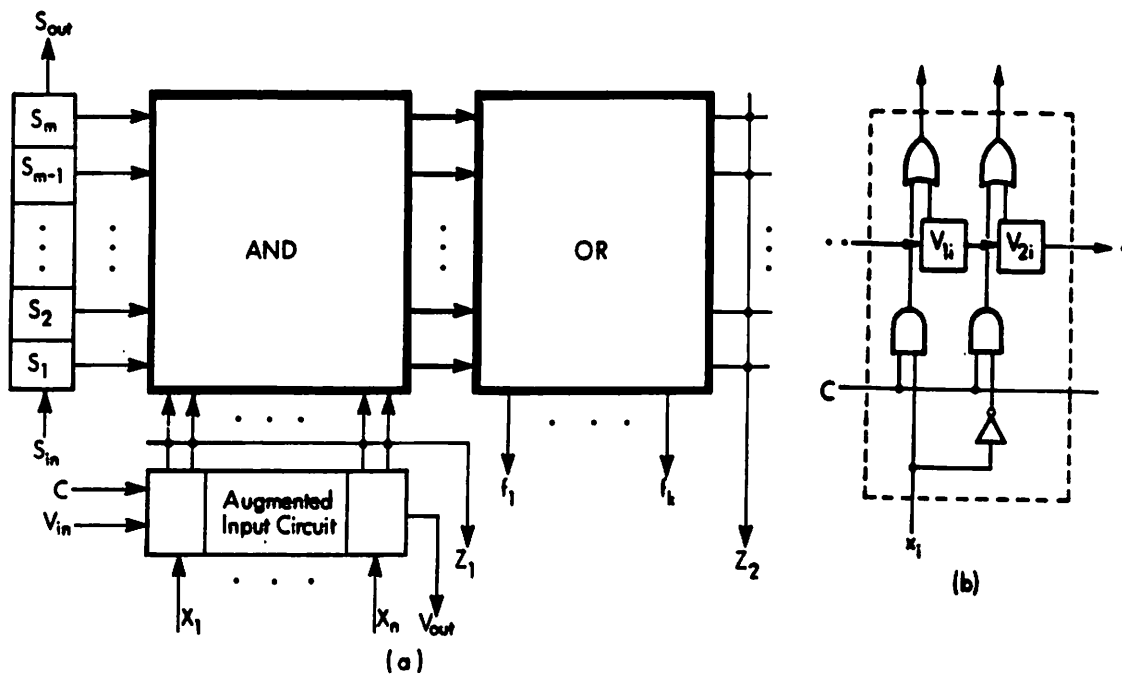
**Example.** Saluja, Kinoshita and Fujiwara [64, 65] presented a design, shown in Figure 2-22, which guarantees detection of all multiple stuck faults and/or crosspoint faults [65]. Two shift registers are used to select an arbitrary bit and product line of a PLA. All multiple faults, except those in the two shift registers, can be detected by applying tests through  $n$  primary inputs, two shift registers and 2 additional outputs and  $k$  primary outputs. Thus the extra I/O is 5 ( $C$ ,  $V_{in}$ ,  $S_{in}$ ,  $Z1$  and  $Z2$ ). However, if the final stage of the two shift registers ( $S_{out}$  and  $V_{out}$ ) can be observed, all faults in the shift registers can also be detected. Therefore to achieve 100% multiple fault coverage, the number of extra I/O connections is 7.

**3. Area overhead:** BIT circuitry requires extra area, referred to as BIT area. We will use the following formula to measure the area overhead for BIT.

$$\text{area overhead} = \frac{\text{BIT area}}{\text{original area}} \times 100\%$$

Different CAD systems employ different design rules and generate different PLA layouts. In estimating area overhead, we use the floor plan of a PLA given in [77], from which it follows that

$$\text{Area of original PLA} = 130nm + 65mk + 900n + 300m + 550k + 2200 (\lambda^2),$$



**Figure 2-22:** Augmented PLA for multiple fault detection  
 (a) The augmented PLA  
 (b) One augmented input circuit cell

where  $n$ ,  $m$ ,  $k$  are the number of inputs, product lines and outputs of the PLA, respectively. Since BIT circuits mainly consist of standard cells, such as shift register cells and parity checker cells, we will measure the additional area as follows:

$$\text{BIT area} = \left[ \sum_i \# \text{ of type}_i \text{ cells} \times \text{area of a type}_i \text{ cell} \right] \times \text{connection cost.}$$

Here connection cost is 1 if a BIT cell can be connected to the original PLA without requiring extra area for wiring. Otherwise, the connection cost is greater than 1. The dominant factor in the area overhead is the part of BIT area which grows with PLA size. As no accurate figures of area can be obtained without carrying out an actual chip layout, our estimates emphasize the major part of area overhead, although constant components are also



considered. The estimates tend to give a lower bound on actual area overhead.

**4. Extra delay:** Adding BIT hardware may have some side effect on system performance, such as additional delay encountered during normal circuit operation. This delay may affect the system's normal operation or change the system's critical path. Extra delay is measured in terms of gate delays. For example, in the TDM shown in Figure 2-8, when an input vector passes through the augmented input decoder, the signals must propagate through one additional logic gate. Hence the extra delay is 1.

**5. Delay per test:** Delay per test refers to the maximal number of gate delays in each test cycle. It is the maximal time for the effect of one test pattern to propagate through the CUT. For built-in testing, delay per test includes the time for generating a test and/or processing the response internally. Delay per test is different from extra delay plus normal delay because signals may go through different paths in the test mode. It is assumed that the original PLA has three gate delays which are attributed to the input decoder, the AND array and the OR array. Thus delay per test is always greater than or equal to 3. Delay per test partially determines the test application rate and may affect the system clock cycle.

**Example.** In the autonomously testable PLA design shown in Figure 2-11, in the test mode, after a test pattern is generated internally (this takes

two gate delays), it propagates through the AND array and the OR array (two gate delays), the parity circuit and the feedback value generator (which is a two level circuit and takes two gate delays). Delay per test is mainly determined by the parity circuits. Since the methodology specifies that the AND parity logic can consist of any circuitry which generates the correct parity, but the OR parity logic must be a parity chain, we assume that the AND parity circuit is a parity tree with minimal delay. Suppose a parity chain with  $i$  inputs consists of a cascade of  $i-1$  exclusive-or (XOR) gates and each XOR has two gate delays, the delay of the OR parity circuit is  $2(k+1)$ , and the delay of the AND parity circuit is  $2[\log(m+2)]$ , where  $[x]$  is the smallest integer greater than or equal to  $x$ . Thus the delay per test for this TDM is

$$6 + 2 \times \max\{ k+1, [\log(m+2)] \}.$$

If we assume that, in general,  $k+1 > \log(m+2)$ , this formula can be simplified to  $6 + 2 \times (k+1)$ .

### 2.5.1.3. Requirements on test environment

1. **Number of test patterns:** The number of test patterns is the size of the minimal test set which must be applied to the CUT to achieve a desired fault coverage. For some TDMs, the test set is well defined, and the number of tests is only a function of the CUT's parameters. For function-dependent TDMs, the number of tests may not be known until the test set is generated.

**Example.** In the testable PLA design with low overhead presented in Section 2.4.5.6, the test set consists of one main test pattern and  $n$  auxiliary patterns which have exactly one bit different from the main pattern for each product line. Therefore for a PLA with  $n$  inputs and  $m$  product lines, the total number of tests is  $m \times (n+1)$ .

**2. Test storage:** Test storage refers to the number of bits needed to store test patterns and reference responses.

Test storage = # of tests stored  $\times$  # of information bits in each test +  
# of response vectors stored  $\times$  # of information bits in each vector.

Since regular patterns can be either stored or generated or store-and-generated,<sup>5</sup> actual test storage depends on the test strategy implementation. Function-dependent patterns must be pre-generated and stored, while simple variations of known patterns or regular function-independent patterns may be generated on-line or stored.

**Example.** For the test set used in preceding example, all patterns are function-dependent. However for each main pattern,  $n$  other patterns can be easily derived. The  $m$  main patterns must be stored, and the others may be either stored or generated on-line. If all tests are stored, then

$$\text{test storage} = m \times (n + 1) \times (n + k + E_i),$$

where  $E_i$  is the number of extra inputs. If only the main patterns are stored, then

---

<sup>5</sup>Store-and-generate refers to a technique where some test vectors are stored, and these vectors are used as the kernels in circuits which generate additional test vectors.

$$\text{test storage} = m \times (n + k + E_i).$$

For this case more test hardware and control is required.

**3. Test application time:** Test application time specifies a lower bound for the time required to complete the entire test process. Suppose (1) there is no overlapping of tests between successive test cycles, (2) each test cycle finishes in one system clock cycle, and (3) all clock cycles are of the same length. Then

$$\text{test application time} = \text{delay per test} \times \text{length of the shortest test sequence which contains the entire test set in the right order.}$$

Note that a test sequence may be different from a sequence of patterns in a test set, because some transition sequences may be necessary in order to apply a given set of tests.

**Example.** Consider a BILBO self-testable PLA, and assume that only the AND array and the OR array are to be tested. The number of test patterns is  $2n+m+2$ . Suppose the contents of the BILBO register cannot be observed in parallel (otherwise the number of extra I/O connections would be  $m+k$ ), so that  $m+k$  clock cycles are needed to serially shift the signatures out for observation. As each BILBO register takes two gate delays, each test cycle is at least five gate delays long. Thus the test application time is  $5(2n+m+2+m+k) = 5(2n+2m+k+2)$ .

**4. Means for test generation:** The manner in which tests are

generated partially determines the applicability of a TDM to a specific user. To produce the required test set prior to testing, some software tools may be necessary. There are several cases.

Case 1. No tools are required. For example, for the self-testable design shown in Figure 2-12, the tests [77] are generated by the BIT hardware and responses are evaluated by the BIT circuit. The logic modification procedure is function-independent, and there is no need for *a priori* test generation.

Case 2. The test patterns are function-independent, but the responses are function-dependent, so simulation is required in order to determine the correct response. For example, in the TDM using signature analysis [31], the tests are pseudo-random sequences and generated on-line. However the expected signature is determined by the function realized by the CUT and has to be determined prior to testing via simulation or physical measurements.

Case 3. The test patterns are function-dependent but can be generated by a simple program.

**Example.** Khakbaz [41] proposed a simple fully testable design which uses an  $m$ -bit shift register to control product lines and an extra output  $z^*$  to help observing outputs and separating the AND array and the OR array, as shown in Figure 2-23.

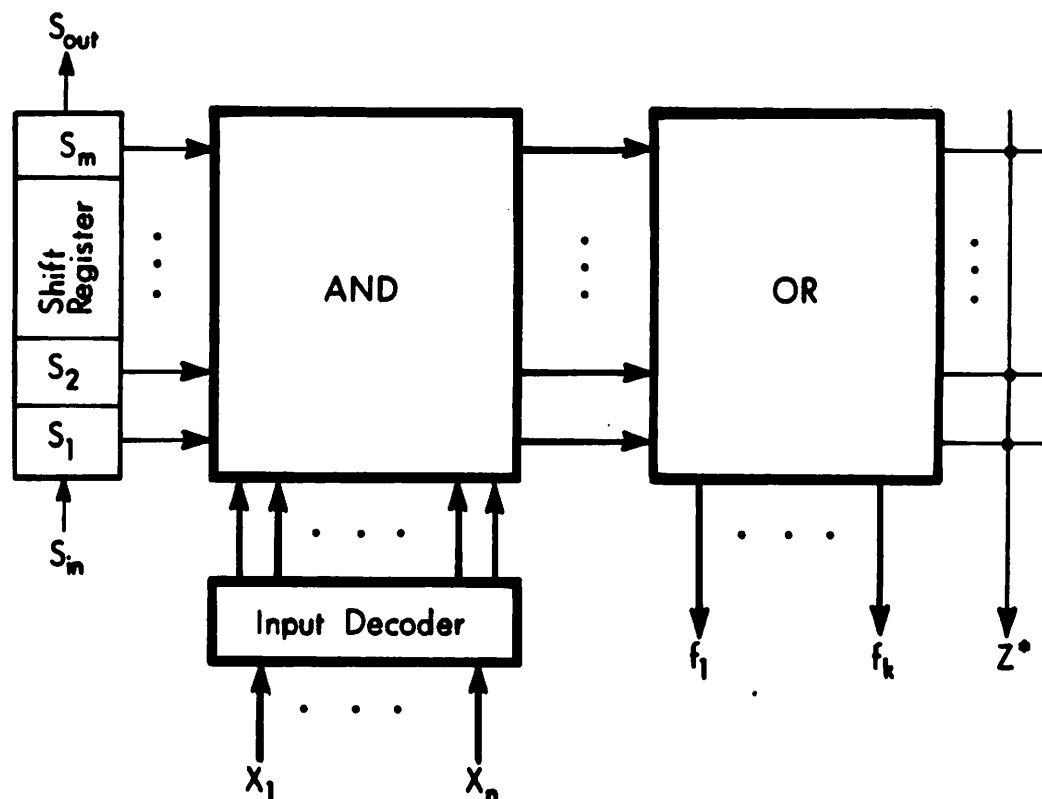


Figure 2-23: An easily testable PLA architecture

A simple test procedure for the modified PLA exists. The first step is to test the added circuitry. A string of  $m$  0's followed by a string of  $m$  1's are shifted through the shift register to make sure it functions properly. Then the shift register along with primary inputs are used to test for  $z^*$  stuck-at 0 and 1.

The second step is to test the AND array. Each product line  $P_i$  is selected in turn by setting  $S_i$  to 0 and all other  $S_j$  to 1. In an NOR-NOR implementation of a PLA, this will make all product lines  $P_j$  equal to 0, for  $j \neq i$ . The value of  $P_i$  is determined by the primary inputs. A selected

pattern  $c_i$  is applied which sets  $P_i$  to 1.  $P_i$  is then observed from  $z^*$ . Next, the faults associated with bit lines and  $P_i$  are tested by patterns that have exactly one bit different from  $c_i$ .

The last step tests the OR array by activating product lines one by one and comparing the output with the expected patterns. This can also be done in the process of step two.

Although test patterns and responses are all function-dependent, this scheme makes test generation very easy. Its fault coverage includes all single and multiple stuck faults, crosspoint faults and all combinations of these faults [40].

Case 4. The test patterns are function-dependent and can only be generated by a special program which may not be commercially available. As we have seen, several test generation algorithms have been developed for PLAs [9, 14, 24, 52, 54, 56, 70, 80]. Considerable effort is required to implement these algorithms.

**5. Means for test application:** Means for test application refers to the hardware required to apply the defined tests to a CUT during real time testing. For a given test set, different ways of test application may exist. There are four typical situations.

**a. No extra hardware necessary.** In the cases of self-testing or concurrent testing, no extra hardware is required to test a testable PLA.

**b. On-line generator.** If a test set is function-independent and very regular, it can be generated by an on-line pattern generator. For example, most TDMs with a universal test set apply the following patterns to a PLA's inputs:

```

0 0 0 ... 0 0
1 0 0 ... 0 0
0 1 0 ... 0 0
...
0 0 0   1 0
0 0 0   0 1

```

These patterns can be generated on-line by a shift register. Using an on-line pattern generator saves test storage but is subject to hardware availability.

**c. Stored test.** If test patterns are not regular, they have to be pre-generated and stored. During real time testing, tests are read from an on-chip or off-chip memory and sent to the CUT through some sensitized path. If the required test storage is small, it may be possible to store test vectors on the chip itself. Otherwise ATE or other off-chip storage is needed. It is assumed that ATE can store large amounts of data and can control the entire test process if both the CUT's inputs and outputs are accessible.

**d. Store-and-generate.** For test sequences consisting of several regular sequences with different initial patterns, a store-and-generate strategy can be used such that a set of initial patterns are stored in a memory and each of them in turn is loaded into an on-line pattern generator which produces a sequence of patterns from that initial vector.



**6. Means for response evaluation:** The test process normally consists of applying test patterns and evaluating the response of the CUT to these patterns. Means for response evaluation of a TDM specifies what extra hardware is required in order to determine the correctness of test responses.

For some TDMs, such as in signature analysis and BILBO technique, a vast amount of output data is compacted into one or few values to be evaluated. For such cases, the response evaluation hardware consists of a small ROM (or RAM) which stores the correct signature, a shift register which receives the generated signature and an on-chip comparator. A signature can also be decoded directly using one or more gates.

Some TDMs require a real time comparison between response data from the CUT and known correct data. If the amount of information to be compared is small, response evaluation may be done on-chip, otherwise this comparison has to be done using ATE.

#### **2.5.1.4. Design cost**

**1. Logic design cost:** Logic design cost of a TDM refers to the degree of difficulty for logically modifying the original design of the CUT to suit the TDM. To measure and compare the work involved in logic modification for various TDMs, design difficulty is quantified into four categories.

Category 1: No logic change required. For example, to apply an exhaustive test, no logic modifications are required.

Category 2: Function-independent changes which apply to any CUT in the same way. For example, any PLA can be easily modified to have a universal test set [25].

Category 3: Function-dependent changes that can be carried out by simple procedures. An example is the self-testable PLA design by crosspoint counting [66]. The number of crosspoints on the product lines and output lines need to be counted and the two arrays rearranged.

Category 4: Function-dependent changes which requires special programs and extensive computation, for instance, the parallel testable PLA design [69] and the low overhead design [10]. It has been shown that finding an optimal solution for these designs is an NP complete problem. Even the heuristic procedure presented in [10] may require several hours on a VAX when applied to a large PLA.

**2 Layout cost:** The layout cost associated with a TDM is a measure of the difficulty in laying out the modified CUT in an area-efficient way. There are again four categories of results.

Category 1: The required additional circuit layout is carried out by a PLA generation macro. For example, to design a fully testable PLA [20, 53],

a great amount of logic modification is required. However since no extra BIT hardware is added, the layout of the testable PLA is as easy as that of the original PLA.

Category 2: Standard BIT units, such as shift registers and parity checkers, are added which can fit into a PLA's narrow pitch with almost zero connection cost.

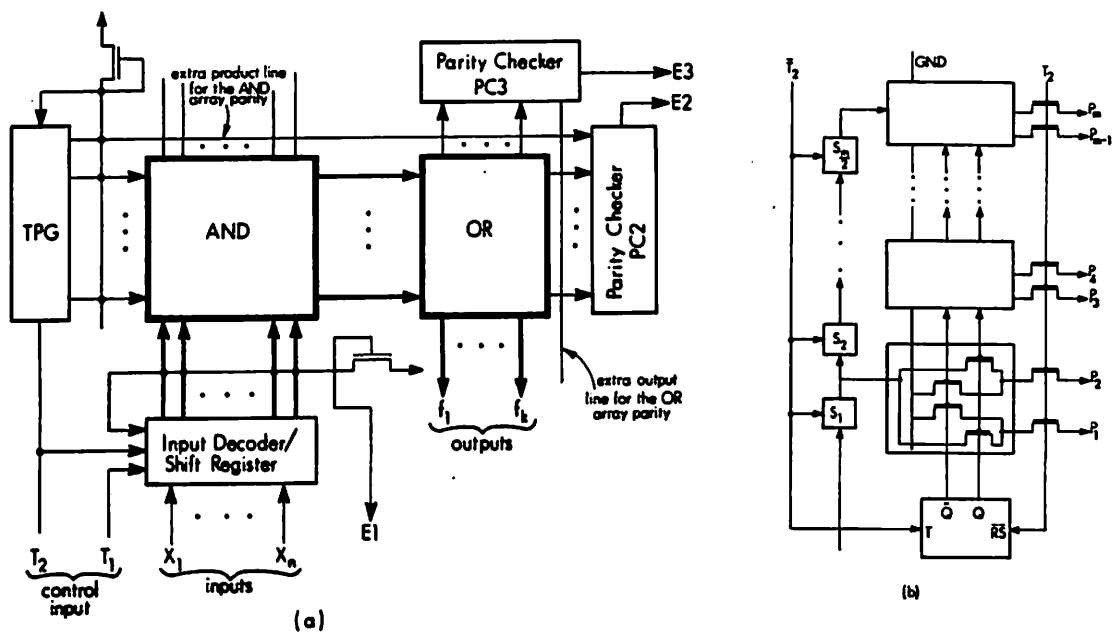


Figure 2-24: A built-in test PLA

- (a) The built-in test architecture  
 (b) The two-stage test pattern generator (TPG)

**Example.** The design of a testable finite-state machine using PLAs is described in [35], and the results illustrated in Figure 2-24. Here the layout is carried out in a way similar to that described in [30]. In Figure 2-24(a), the TPG is basically a shift register for controlling the  $m$  product lines. It

consists of a  $\lceil m/2 \rceil$ -bit shift register and  $\lceil m/2 \rceil$  2-to-2 switch boxes, as shown in Figure 2-24 (b). Similarly, an  $n$ -bit shift register is used in the augmented decoder AD to control the  $2n$  bit lines. A compact parity checker design (for PC2 and PC3) is also used. The layout for all the BIT hardware can fit into the PLA's normal pitch. The connection cost between BIT hardware and the original PLA is almost 0. This design and layout technique is adopted by many designers.

Category 3: Standard BIT units are added which cannot fit into a PLA's narrow pitch. For example, there is no known layout of a one bit signature analyzer cell such that one dimension of its bounding box is  $8\lambda$ , which is the pitch between two output lines in a PLA.

Category 4: BIT hardware is often hard to lay out and causes high connection cost. A typical example is the testable PLA with BILBO methodology [17]. BILBO cells are too big to be inserted into PLA arrays. In addition, BILBO 2 in Figure 2-16 has to receive and send data from/to  $m$  product lines. This makes the layout even more difficult and connection cost very high.

### 2.5.2. Evaluation matrix

The criteria defined in the previous section can be used to compare different TDMs and to determine their applicability to various PLAs. An evaluation matrix has been constructed which contains attribute values for most known PLA TDMs. This matrix is given in Table 2-3, where each row corresponds to a TDM and each column corresponds to an attribute. The TDM's values in respect to the attributes are based on either theoretical proof or careful estimation. Some attribute values are calculated for particular implementations of TDMs, and thus may be different for other implementations. For layout related measures, such as area overhead, nMOS technology and Mead-Conway design rules [50] are used. These measures can only be estimated if actual placement and routing are not carried out. The abbreviations for the TDMs used in Table 2-3 are listed in Table 2-2. None of these TDMs masks faults, and only CONC1 & CONC2 support concurrent testing. Therefore these two attributes are not listed in the table in order to save some space.

Some notes for the table entries are given below.

1. Blank entries represent unknown data.
2. NA stands for not applicable.
3. F.D. stands for function-dependent.
4. ASR = area of 1-bit shift register.
5. APC = area of 1-bit parity checker.

---

EXH	Exhaustive test
CONC1	PLA with concurrent error detection by a series of checkers [39]
CONC2	Concurrent testable PLA using error detection code [22]
UTS	PLA with universal test set [25]
FIT	Function-independent testable PLA [33]
AUTO	Autonomously testable PLA [83]
BILBO	PLA with BILBOs [17]
MFT	Multiple fault testable design of PLA [65]
SIG	PLA with signature analyzer [31]
FIST	Function-independent self-testable PLA [30]
TLO	Testable PLA with low overhead and high fault coverage [41]
BIT	Built-in tests for VLSI finite-state machines [35]
UT2	PLA design for universal testability [26]
LOD	Low overhead design of testable PLA [10]
PTPLA	Parallel testable PLA [69]
BIST2	Built-in self-testable PLA [77]
HFC	High fault coverage built-in self testable PLA [66]
DAC	Divide and conquer strategy for testable PLA [67]
MIN	PLA design for ease of test generation [53]
FTPLA	Fully testable PLA [20]
TSBF	PLA design for testing single bridging faults [58]
TBSL	Design of testable PLA by specialized layout [71]
DCT	Design for complete testability of PLA [59]
CHA	Test generation algorithm by Cha [14]
EL	Test generation algorithm by Eichelberger & Lindbloom [24]
OH	Test generation algorithm by Ostapko & Hong [56]
SMITH	Test generation algorithm by Smith [70]

---

**Table 2-2:** Abbreviations of TDMs in the evaluation matrix

6. AMC = area of a 1-out-of-m code checker.
7. ASG = area of a signature analyzer cell.

8. AB = area of a BILBO register cell.
9. AC = area of 1-bit counter.
10. AF = area of a flip-flop.

As can be seen from the evaluation matrix, many TDM measures are functions of a PLA's parameters. For different PLAs, a TDM may appear quite different. Given a PLA, all the entries in the evaluation matrix can be calculated and become constants, and the matrix becomes a *decision table*. This data is then used in the TDM selection process.

### 2.5.3. Performance statistics of PLA TDMs

It is interesting to look at some performance statistics of PLA TDMs. Experiments have been carried out on 52 PLAs given in [11] using the results shown in Table 2-3. For this analysis, the size of a PLA is defined to be  $m(2n+k)$  which is proportional to a PLA's layout area. A TDM's attributes can be roughly divided into two groups, i.e., those having values which are constant (or almost constant) for any kernel, and those having values which are a function of a kernel's parameters. Among the attributes with variable values, the most important ones are area overhead, test application time and test storage. We calculated the values of these three attributes for 15 TDMs and 52 PLAs whose sizes range from small (4 inputs, 7 outputs, 10 product lines) to very large (47 inputs, 72 outputs, 241 product lines). The PLA data are given Table 2-4. In the following, we will summarize the results and draw some conclusions.

## FAULT COVERAGE ( % )

	single stuck	single crosspoint	single bridge	multiple stuck	multiple unidirectional	crosspoint others	multiple bridge	
EXH	100	100	100	100	100	100	100	
CONC1	99	99	99					
CONC2	100	100			100			
UTS	100	100		90				
FIT	100	100						
AUTO	99	100						
BILBO	99	99	99					
MFT	100	100		100	100	100		
SIG	98	98		98				
FIST	99	99	99	99	99	99	99	
TLO	100	100		100	100	100		
BIT	100	100	100	100	100	100		
UT2	100	100	100	100	100	100	100	
LOD	100	100	100	100	100	100	100	
PTPLA	100	100		100	100			
BIST2	100	100	100	$100(1-2^{-(m+2n)})$	$100(1-2^{-(m+2n)})$		$100(1-2^{-m})$	
HFC	100	100			100	99.9		
DAC	100	100		100	100	100		
MIN	100	100	100	100	100	100	100	
FTPLA	100	100						
TSBF			100					
TBSL	100	100	100					
DCT	100	100	100					
CHA	100	100	100					
EL		~97(This figure is for missing crosspoint faults only.)						
OH	100	100						
SMITH	100	100			100			

Table 2-3. The evaluation matrix



TDM	ERROR LATENCY	SELF TESTING	FUNCTION TESTS	DEPENDENCY RESPONSES	TEST GENERATION	DESIGN LOGIC	COST LAYOUT	EXTRA IO PINS	EXTRA DELAY
EXH	NA	no	no	yes	no	1	1	0	0
CONC1	9	no	yes	no	easy	4	4	5	0
CONC2	8	no	no	no	no	4	4	2	0
UTS	NA	no	no	no	no	2	4	5	1
FIT	NA	no	no	no	no	2	4	$6+\log(n+1)$	1
AUTO	NA	yes	no	no	no	2	4	2	1
BILBO	NA	yes	no	yes	no	2	4	3 - 9	0
MFT	NA	no	no	yes	no	2	3	5	1
SIG	NA	yes	no	yes	no	2	4	4	0
FIST	NA	yes	no	yes	no	2	3	1	3
TLO	NA	no	yes	yes	easy	2	2	3	0
BIT	NA	yes	no	no	no	2	2	5	1
UT2	NA	no	yes	no	no	2	3	4	0
LOD	NA	no	yes	yes	easy	4	1	F.D.	0
PTPLA	NA	no	no	yes	no	4	3	4	1
BIST2	NA	yes	no	no	no	2	2	4	1
HFC	NA	no	no	yes	no	3	2	3	1
DAC	NA	no	no	yes	no	2	2	4	1
MIN	NA	no	yes	yes	easy	4	1	F.D.	0
FTPLA	NA	no	yes	yes	hard	4	1	F.D.	0
TSBF	NA	no	yes	no	med.	2	1	2	0
TBSL	NA	no	yes	yes	hard	3	1	1	0
DCT	NA	no	yes	yes	hard	4	1	F.D.	0
CHA	NA	no	yes	yes	hard	1	1	0	0
EL	NA	no	yes	yes	med.	1	1	0	0
OH	NA	no	yes	yes	hard	1	1	0	0
SMITH	NA	no	yes	yes	hard	1	1	0	0

Table 2-3. The evaluation matrix (cont'd)

	DELAY PER TEST	# OF TESTS	TEST APPLI- CATION TIME	TEST STORAGE
EXH	3	$2^n$	$3 \cdot 2^n$	0
CONC1	12	$2m+D+8$	$12(2m+D+8)$	$(2m+D+8)(3+n)$
CONC2	3	0	0	0
UTS	$3+2\log m$	$2(n+m)+1$	$(3+2\log m)(2n+3m)$	$5(5+n)$
FIT	$4+2\log m$	$5+2.5n+2m$	$(4+2\log m)(5+2.5n+2m)$	$8(5+2.5n+2m)$
AUTO	$6+2(k+1)$	$2(n+m)+9$	$(8+2k)(2n+2m+9)$	0
BILBO	4	$2n+m+k+3$	$4(2n+2m+\max(n,k)+1)$	$2n+2m+\max(n,k)$
MFT	4	$2nm+m+4n+4$	$4(2nm+m+4n+4)$	$(k+2)(2nm+m+4n+4)$
SIG	6	$2^n$	$6 \cdot 2^n$	$n(2+[k/n])$
FIST	7	$2nm$	$14nm$	$k$
TLO	3	$m(7+n)+2$	$3m(7+n)+6$	$(7m+nm+2)(n+k+2)$
BIT	$m+4$	$2n+m+2$	$(m+4)(2n+m+2)$	0
UT2	3	$2n(m+1)+m+3$	$6n(m+1)+3m+9$	$(k+1)(2n(m+1)+m+3)$
LOD	3	$(n+1)m$	$3m(n+1)$	$m(n+nk+k)$
PTPLA	4	$2n+1+4n+4$	$8n+1+24n+12$	F.D.
BIST2	$1.25k+3$	$2m(n+1)+1$	$(1.25k+3)(2nm+2m+1)$	0
HFC	6	$m(2n+k+3)+k+2$	$6m(2n+k+1)+6k$	0
DAC	4	$m(2+2n+\log m)+4$	$4m(2+2n+\log m)+16$	0
MIN	3	$2n+D$	$3(2n+D)$	F.D.
FTPLA	3	F.D.	F.D.	F.D.
TSBF	3	$m+1$	$3m+3$	$(m+1)(n+2)$
TBSL	3	F.D.	F.D.	F.D.
DCT	3	F.D.	F.D.	F.D.
CHA	3	F.D.	F.D.	F.D.
EL	3	random	random	random
OH	3	F.D.	F.D.	F.D.
SMITH	3	F.D.	F.D.	F.D.

Table 2-3. The evaluation matrix (cont'd)

EXTRA TEST SUPPORT MEANS REQUIRED			
TDM	TEST SET GENERATION	TEST APPLICATION	RESPONSE EVALUATION
EXH	response simulation	counter   LFSR	ATE
CONC1	simple TG program	ATE   ROM	monitor error indication
CONC2	none	none	none
UTS	none	ATE   on-line PG	monitor two extra outputs
FIT	none	ATE   on-line PG	ATE   ROM & comparator
AUTO	none	none	none
BILBO	obtain signature	control BILBO function	S.R. & comparator
MFT	response simulation	ATE   ROM   on-line PG	ATE
SIG	obtain signature	none	ROM & comparator
FIST	obtain signature	none	comparator   flag circuit
TLO	simple TG program	ATE   ROM & on-line PG	ATE
BIT	none	none	monitor 3 extra outputs
UT2	response simulation	ATE   on-line PG	ATE
LOD	simple TG program	ATE   ROM & on-line PG	ATE
PTPLA	response simulation	ROM & simple control	ATE
BIST2	none	none	none
HFC	count crosspoints	ROM & simple control	none
DAC	response simulation	none	ATE
MIN	simple TG program	ATE   ROM	ATE   ROM & comparator
FTPLA	special TG program	ATE	ATE
TSBF	special TG program	ATE   ROM	ATE   ROM & comparator
TBSL	special TG program	ATE	ATE
DCT	special TG program	ATE	ATE
CHA	special TG program	ATE	ATE
EL	special TG program	ATE	ATE
.OH	special TG program	ATE	ATE
SMITH	special TG program	ATE	ATE

Table 2-3. The evaluation matrix (cont'd)

	EXTRA LINES			EXTRA	AREA OVERHEAD
	BIT	PRODUCT	OUTPUT	TRANSISTORS	
EXH	0	0	0	0	0
CONC1	0	0	1	$10n+5k+\log m(m+5)$	$15680(n/2-1)+APC(k+1)+AMC$
CONC2	2	0	C	F.D.	F.D.
UTS	0	1	1	$4n+11m+5k+6$	$(m+1)ASR+(m+k)APC+64(6n+m+k+15)$
FIT	0	3	1	$11m+5k+0.5n\log n+29$	$5397+977n+2113m+1021k$
AUTO	0	4	2	$12n+12m+5k+50$	$2215m+1764n+1420k+11752$
BILBO	0	0	0	$16(2n+m+k)$	$1.3(2n+m+\max(n,k))A$
MFT	0	1	1	$6m+16n$	$(m+2n)ASR+128n+64m+752$
SIG	0	0	0	$6(3n+[k/n])$	$1.3(3n+[k/n])ASG$
FIST	0	0	0	$12n+6m+8k$	$(2n+m)ASR+1.1k*ASG$
TLO	0	0	1	$6m$	$m(64+ASR)+400$
BIT	0	2	2	$12n+11m+5k$	$(2n+m+1)ASR+(m+k+2)APC+2AF+128(m+2n+k)+896$
UT2	0	3	1	$6(m+1)$	$(m+1)ASR+384n+64m+64k+944$
LOD	2C	0	0	0	$C*(128m+640)$
PTPLA	$2\log(m/J)$	0	0	$6J+2\log(m/J)$	$64m+128n+696+ASR(2n+J)$
BIST2	0	3	1	$11n+6m+3.5k+30$	$1360m+2160n+760k$
HFC	0	0	2	$12n+6m+6k+19\log m$	$(2n+m+k+2)ASR+(2AC+5)\log m+128m+1040$
DAC	$2+2J$	0	0	$12n+6J+10\log(m/J)$	$(1+2n)ASR+(1+[\log(m/J)])AC$
MIN	F.D.	F.D.	F.D.	0	F.D.
FTPLA	0	F.D.	F.D.	0	F.D.
TSBF	0	1	1	0	$64(2n+m+k)+816$
TBSL	0	0	1	0	F.D.
DCT	2	2	2	0	$16n+8k+256m+1989$
CHA	0	0	0	0	0
EL	0	0	0	0	0
OH	0	0	0	0	0
SMITH	0	0	0	0	0

Table 2-3. The evaluation matrix (cont'd)

## ASSUMPTIONS &amp; REMARKS

EXH	Correct response for all input combinations must be stored.
CONC1	PLA must be non-concurrent. The two-rail checker is composed of two EOR trees.
CONC2	PLA must be non-concurrent. C = the length of check symbols.
UTS	The fault coverage is 100% if the column rank of the OR array = k.
FIT	
AUTO	Assume the feedback value generator is not duplicated.
BILBO	The number of extra I/O pins depends on implementation, but is between 3 to 9.
MFT	The last stage of the shift registers(SRs) should be observed for testing SRs.
SIG	The BIT circuit may be placed outside of the PLA.
FIST	The fault coverage is estimated. No solid proof is given.
TLO	
BIT	PLA can be a sequential machine.
UT2	
LOD	C needs to be calculated by a special program and is usually a small integer.
PTPLA	J is the number of functionally dependent product line groups.
BIST2	Assume n is odd. If n is even, one more product line may be needed.
HFC	
DAC	J is the number of partitioned product line groups.
MIN	D is the number of used crosspoints in the AND array.
FTPLA	PLA's personality needs to be modified by a special program.
TSBF	Bridging faults are of AND type.
TBSL	AND array consists of mutually disjoint product terms.
DCT	PLA is crosspoint-irredundant Bridging faults are of AND type.
CHA	
EL	Only missing crosspoint faults are considered.
OH	Test set is generated for crosspoint faults only.
SMITH	The size of the test set is bounded by $n(2m+k)$ , but may be smaller.

Table 2-3. The evaluation matrix (cont'd)

PLA name	# of inputs (n)	# of outputs (k)	# of products (m)	# of used crosspoints (d)	PLA size (2n+k)*m	aspect ratio m/(n+k)
WIM	4	7	10	91	150	0.9091
DC1	4	7	15	69	225	1.3636
RD53	5	3	31	197	403	3.875
CO14	14	1	14	210	406	0.9333
DK27	9	9	20	200	540	1.1111
ALU1	12	8	19	60	608	0.95
DC2	8	7	58	455	1334	3.8667
SQN	7	3	84	732	1428	8.4
SQR6	6	12	63	637	1512	3.5
DK17	10	11	57	631	1767	2.7143
ALU3	10	8	68	352	1904	3.7778
DK48	15	17	42	672	1974	1.3125
Z4	7	4	127	1145	2286	11.545
ALU2	10	8	87	593	2436	4.8333

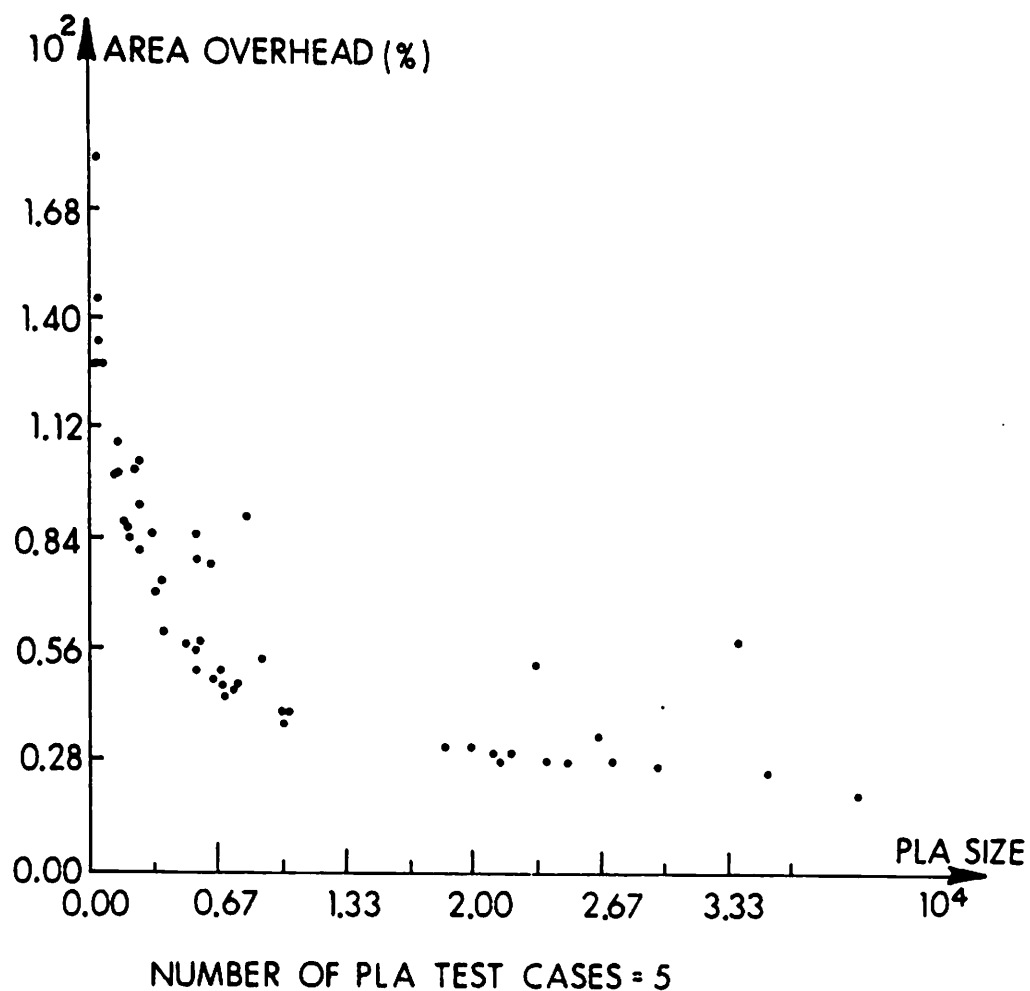
Table 2-4: 52 PLAs used for statistics

### 2.5.3.1. Area overhead vs. PLA size

Figure 2-25 shows a typical plot of a TDM's area overhead vs PLA size. Additional results are given in Figure 2-26, where in order to compare different TDMs, we have drawn a continuous line through the discrete dots for each TDM. Most curves have a similar shape. Note that the area overhead tends to decrease as the PLA size increases. However, there are some exceptions which correspond to PLAs with a large number of product lines. For example, in Figure 2-26 (a), three of the peak points A, B and C correspond to PLAs SYM, TIAL and ADD6 described in Table 2-4, and which have the aspect ratios of 42, 29 and 57, respectively. These values are far above the average aspect ratio for the rest of the PLAs, which is 4.65. But a counterexample can be seen in Figure 2-26 (b), where the area

PLA	n	k	m	d	(2n+k)*m	m/(n+k)
RD73	7	3	147	1023	2499	14.7
RADD	8	5	120	768	2520	9.2308
XP1	7	10	128	1472	3072	7.5294
RISC	8	31	74	407	3478	1.8974
APLA	10	12	112	1277	3584	5.0909
IN5	24	14	62	752	3844	1.6316
IN6	33	23	54	552	4806	0.9643
ROOT	8	5	255	2655	5355	19.615
IN7	27	10	84	563	5376	2.2703
IN1	16	17	110	2100	5390	3.3333
MLP4	8	8	225	2478	5400	14.063
INO	15	11	135	1825	5535	5.1923
F51M	8	8	255	3064	6120	15.938
VG2	25	8	110	914	6380	3.3333
IN2	19	10	137	1527	6576	4.7241
X1DN	27	6	112	1090	6720	3.3939
RCKL	32	7	96	1238	6816	2.4615
X9DN	27	7	120	1258	7320	3.5294
IN3	35	29	75	852	7425	1.1719
SYM	9	1	420	4200	7980	42.0
GARY	15	11	214	2240	8774	8.2308
CHKN	29	7	153	1865	9945	4.25
X6DN	39	5	121	1400	10043	2.75
MISG	56	23	75	255	10125	0.9494
EXEP	30	63	149	1944	18327	1.6022
IN4	32	20	234	3291	19656	4.5
MISH	94	43	91	255	21021	0.6642
JBP	36	57	166	1252	21414	1.7849
BCD	26	38	243	4483	21870	3.7969
TIAL	14	8	640	5627	23040	29.091
BCC	26	45	245	4903	23765	3.4507
X2DN	82	56	112	578	24640	0.8116
BCO	26	11	419	6673	26397	11.324
BCB	26	39	299	5684	27209	4.6
BCA	26	46	301	5905	29498	4.1806
ADD6	12	7	1092	9840	33852	57.474
OPA	17	69	342	2461	35226	3.9767
TI	47	72	241	3171	40006	2.0252

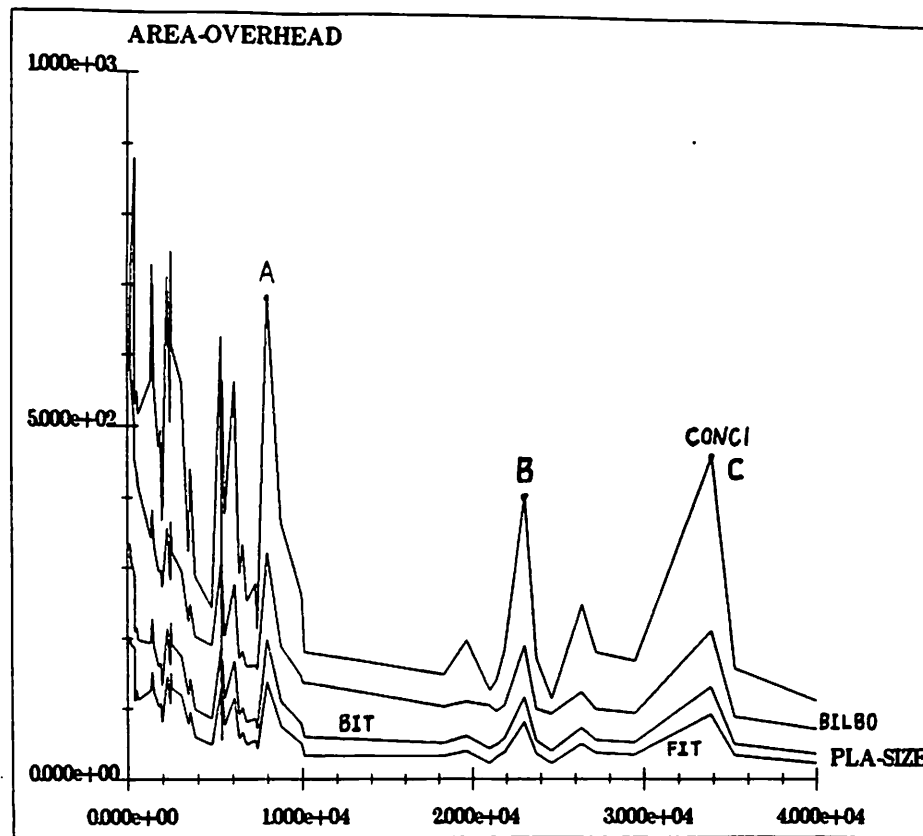
Table 2-4: 52 PLAs used for statistics (cont'd)



**Figure 2-25:** Area overhead vs PLA size for PLA TDM FIST [30]

overhead of TDM SIG changes almost in the opposite direction as that of the others. In general, the area overhead of PLA TDMs is a function of both the PLA size and the aspect ratio. Some TDMs are not plotted here because their area overhead is constant for all PLAs, e.g., testing a PLA using external ATE results in no area overhead.





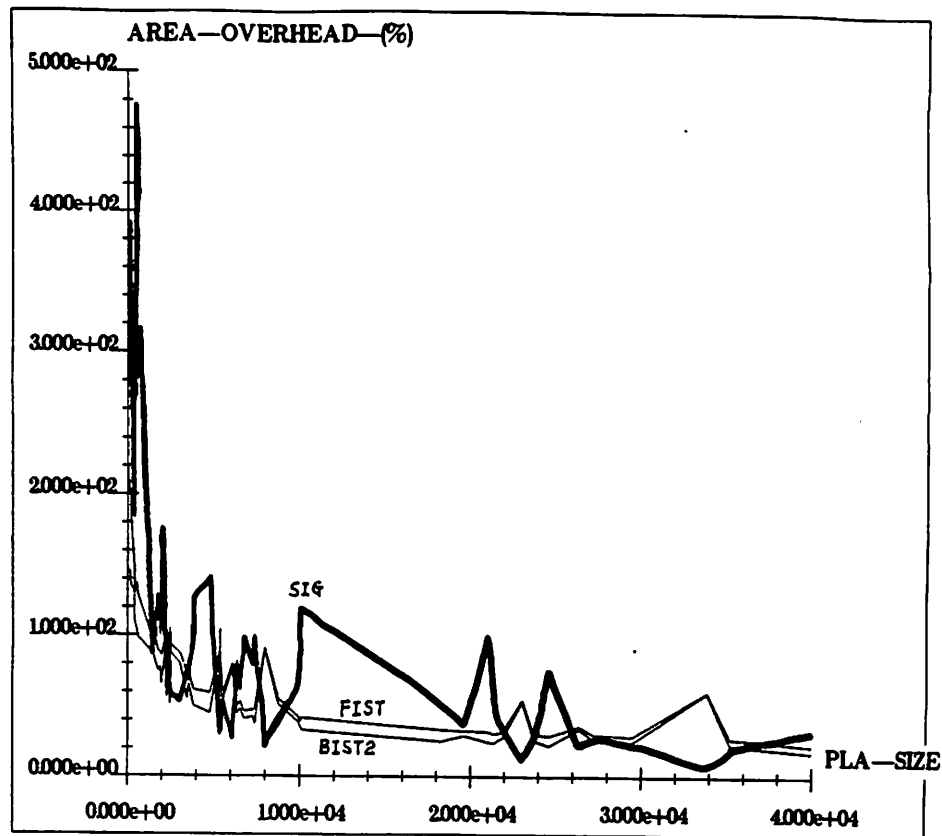
(a)

Figure 2-26: Area overhead vs PLA size

### 2.5.3.2. Test storage vs. PLA size

Figure 2-27 illustrates the relationship between the test storage and the PLA size. Unlike the area overhead, the test storage required by TDMs generally increases as the PLA size increases. There are three cases.

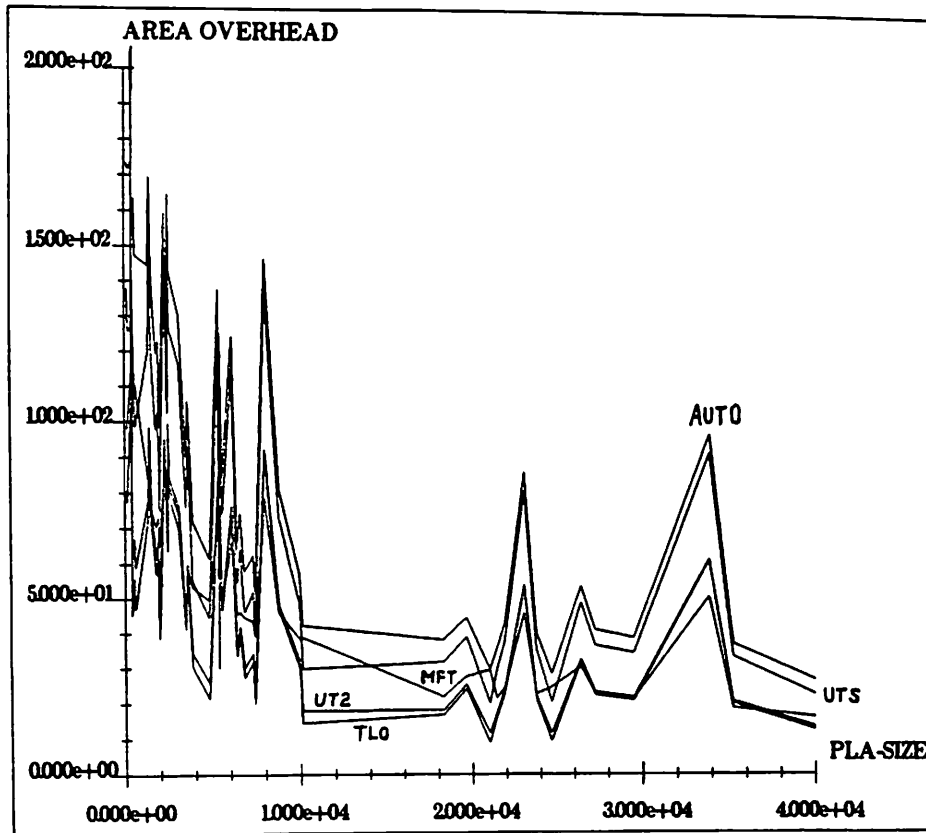
- Case 1. For self-testable or semi-self-testable TDMs, test storage increases very little, if any, with PLA size. In fact, it may only increase with the number of outputs, since only few signatures need to be stored [see TDM SIG and FIST in Figure 2-27 (b)].



(b)

Figure 2-26: Area overhead vs PLA size (cont'd)

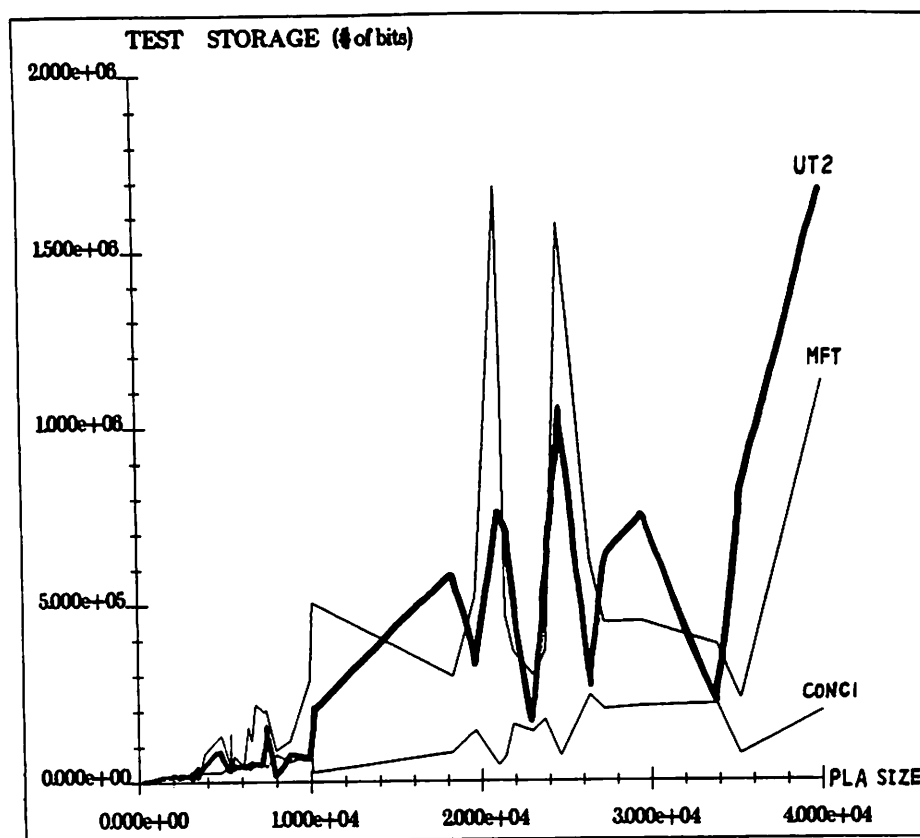
- Case 2. For more than half of the TDMs, test storage increases almost linearly with PLA size, although their rates of increase are different [see TDMs in Figure 2-27 (a) and (c) as well as BILBO in (b)].
- Case 3. For a few TDMs, such as EXH, test storage increases exponentially with the number of inputs (which are not plotted because of their large values). These TDMs are not applicable to big PLAs.



(c)

FIGURE 2-26: Area overhead vs PLA size (concluded)

A PLA's aspect ratio also influences test storage. For example, in Figure 2-27 (b), if we draw a line approximating the linear increase of storage vs. PLA size for the BILBO TDM, most points above the line correspond to PLAs with a large aspect ratio. As for area overhead, the three peak points A, B and C correspond to PLA SYM, TIAL and ADD6, respectively. In Figure 2-27 (c), besides A, B and C, there are other peak points, such as D, E and F, which correspond to PLAs with a large number of inputs. Figure 2-27 (a) shows that the test storage for different TDMs changes radically with the parameters of a PLA.

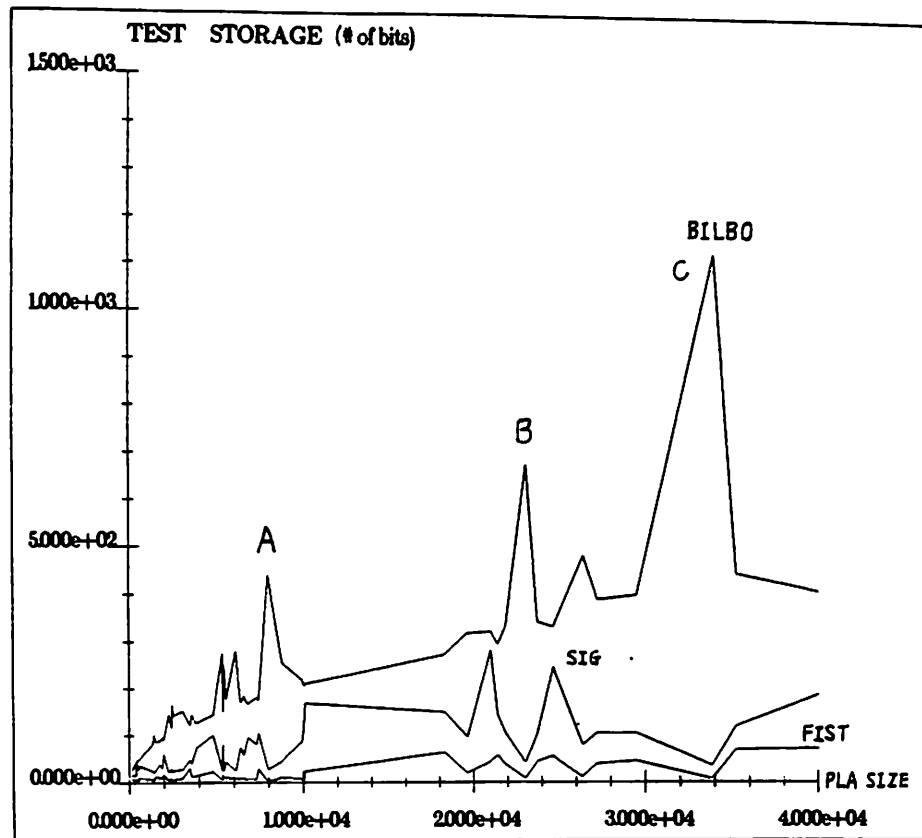


(a)

Figure 2-27: Test storage vs PLA size

### 2.5.3.3. Test application time vs. PLA size

Figure 2-28 illustrates how the test application time and the PLA size are related. Similar to test storage, for most TDMs the test application time increases linearly with the PLA size and varies with the PLA's aspect ratio. However, for a few TDMs, the test application time increases dramatically for some PLAs, e.g., see UTS shown in Figure 2-28 (c). For UTS, considerably longer test application time would be required for PLAs with a large aspect ratio. Much worse examples occur for EXH and SIG, which



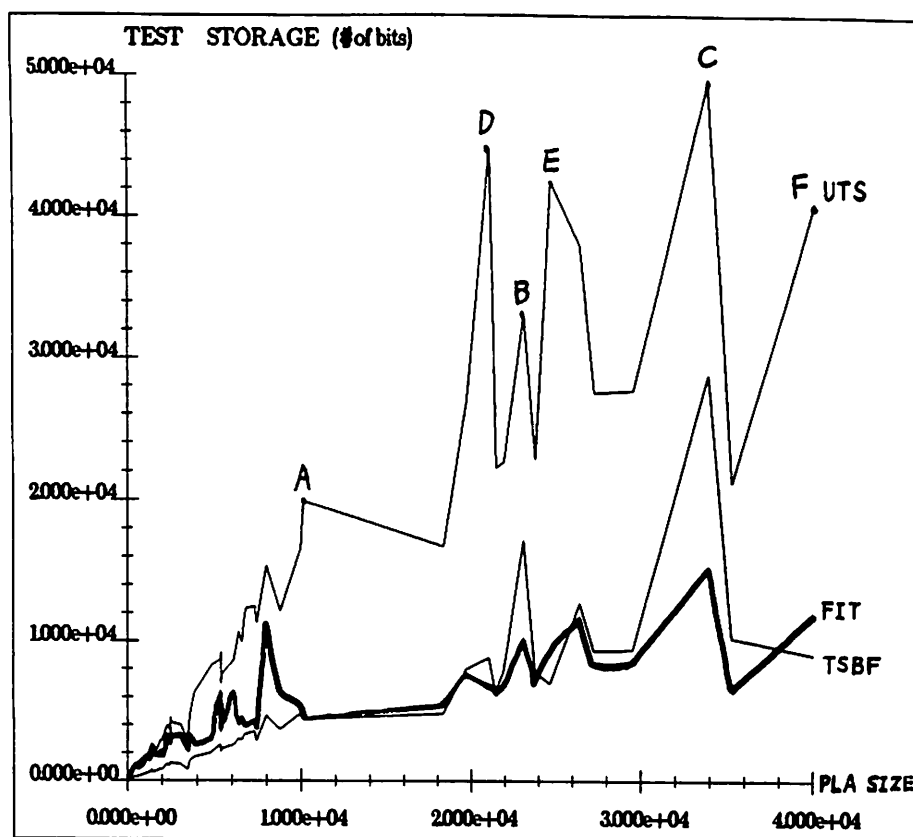
(b)

Figure 2-27. Test storage vs PLA size (Cont'd)

apply all input combinations as tests. Thus testing time grows exponentially with the number of inputs. One PLA would require 55 minutes to test.

Since TDMs for PLAs have diverse features, many different plots exist which are not shown here because of limited space. Generally speaking, statistics lead to the following conclusions.

- Testable designs for PLAs become more practical as PLA size increases. For very small PLAs, all TDMs requiring BIT hardware result in large area overhead. For large PLAs,

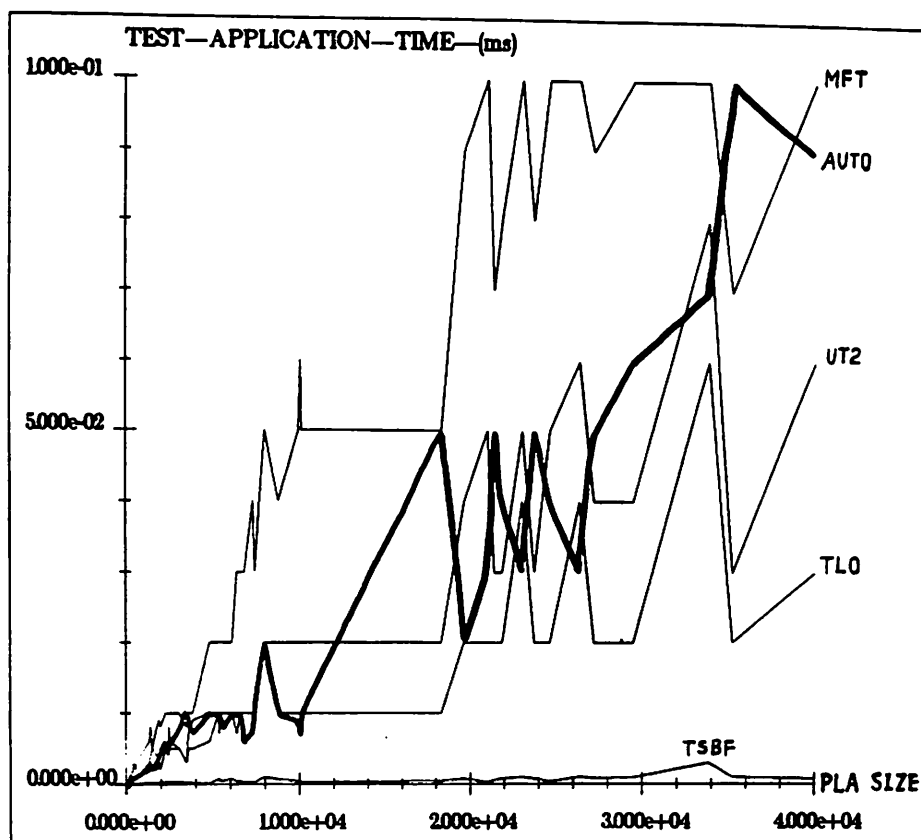


(c)

Figure 2-27. Test storage vs PLA size (concluded)

however, area overhead can drop to or below 20%. In addition, test time and storage usually increase linearly with PLA size.

- Different TDMs are suitable for different PLAs and applications. Since many measures are complex functions of a PLA's parameters, a TDM's attribute values may be good for some PLAs and bad for others. For example, Figure 2-26 (b) shows that the area overhead for SIG is much higher than BIST2 for most PLAs but lower than BIST2 for some PLAs which have a

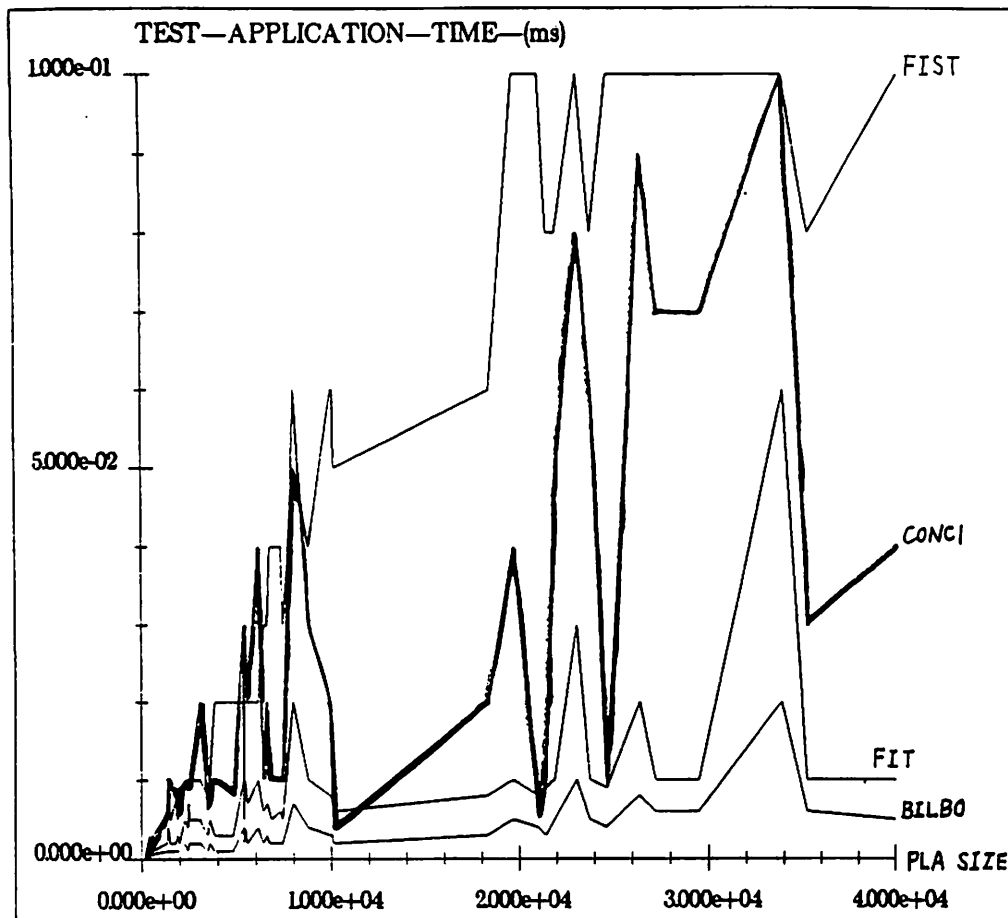


(a)

**Figure 2-28:** Test application time vs PLA size

large aspect ratio. A similar contrast can also be seen in Figure 2-28 (c). This indicates that a simple ranking of TDMs cannot be determined, but rather needs to be examined for each specific PLA.

- Each TDM has distinguished advantages and disadvantages. The most apparent example is EXH which requires no area overhead but a great deal of test storage and long test application time for large PLAs. Among many other examples, Figure 2-26 (a) shows

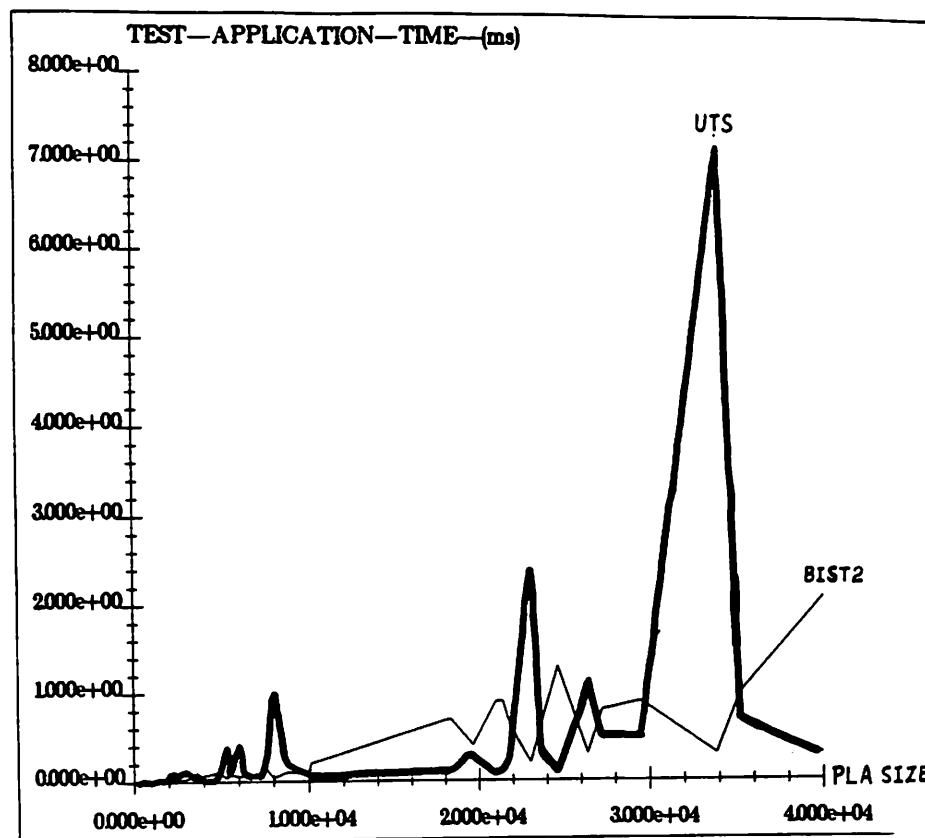


(b)

**Figure 2-28.** Test application time vs PLA size (Cont'd)

that BILBO has larger area overhead than FIT, while Figure 2-28 (b) illustrates that FIT uses more testing time than does BILBO. Considering 20 attributes, it is not obvious which TDM is better, even for a specific PLA.





(c)

Figure 2-28. Test application time vs PLA size (concluded)

## 2.6. Conclusion

The design of testable PLAs is a relatively new and dynamic research area. Early work dealt with identifying fault models and special test problems. Recognizing that traditional test methods, like the D-algorithm, could not be efficiently applied to PLAs, special test generation algorithms were developed. Due to the PLA's regular structure, many built-in test schemes and function-independent test methods have evolved. Although several early TDMs required a significant amount of area overhead, these

techniques demonstrated the basic principles which later were incorporated into more efficient designs. Another branch of research focused on the design of fully testable PLAs via logic modification. Several new TDMs for PLAs are proposed each year. The primary current thrust is to reduce area overhead and increase fault coverage so as to make testable designs more practical. There is little doubt that research in PLA testing will continue to produce better solutions.

The ever-growing number of TDMs for PLAs introduces a new problem to PLA designers and users who want to make a PLA testable: how to select the most suitable TDM for a given PLA and application? This selection problem will soon confront most digital system designers who must deal with testing and design for testability. We have shown that TDMs have multiple attributes. No TDM is absolutely better than any other in every aspect for all PLAs and requirements. Simply applying one TDM to all PLAs will, in many cases, result in poor designs. Therefore, dynamic selection of TDMs for specific circuits and applications is necessary. Selection of a suitable TDM for a given PLA involves searching in a multi-dimensional space under some given requirements, and making tradeoffs between different attribute values. To solve this problem, a knowledge based system for selecting TDMs for PLAs has been built (PLA-TSS). It will be discussed in detail in the following chapters.

## Chapter 3

### A general approach for selection

We have seen that there are numerous design for testability methods for PLAs. The selection of the most suitable one for a particular application is a non-trivial problem. This problem is not limited to just the selection of PLA TDMs; it is a much more general problem. In this section we present a general model for selection problems and discuss an approach for solving this class of problems. We will also describe a framework for a knowledge based consultant system for aiding a selection. Examples will usually be in terms of TDMs for PLAs.

#### 3.1. The general selection problem

Life is full of choices. We face selection problems almost every day.

**Example 1.** Consider the problem of buying a car. There are many different cars on the market. They have distinct features, performance and price. Selecting a car which is most suitable to a buyer is a non-trivial problem. A buyer needs to investigate many cars and shop for the best price before he can decide which one to buy.

**Example 2.** Consider the problem of making a PLA testable under

given design constraints. Often a designer will employ some well known technique for designing a testable PLA. Since there are many TDMs for PLAs, the problem becomes one of selecting the most suitable TDM which meets the designer's requirements.

Although these two examples deal with different problems, they do have several features in common. They both involve selecting an object which satisfies some constraints and goals, i.e., they are both selection problems. In the following we will define the general selection problem, analyze the problem, and present a knowledge based system approach for solving such problems.

### **3.1.1. Main factors affecting selection**

We first consider what factors affect a selection and what make the selection problem complex. Three main elements involved in a selection are a selector, a receiver and a domain of alternatives (objects) to be selected.

#### **3.1.1.1. The selector**

A selector is an element which initiates and controls a selection, and thus plays an important role in a selection process. A selector may be a person or a system. For instance, TDM selection can be performed by a human designer or a synthesis program. The main functions of the selector are twofold.

1. Specify a set of requirements for the alternative to be selected.
2. Decide which item to select, or end the selection process with no

solution.

A selector's personal preference and knowledge about the solution space greatly affects the result of a selection.

#### **3.1.1.2. The receiver**

A receiver is the element for which the selection is made. Each candidate in a solution space has a set of applicable receivers, e.g., TDMs are for digital circuits. The receiver need not be the selector itself. For example, a car buyer may buy a car for himself or for somebody else. In TDM selection, the receiver is a circuit under test but not the designer.

A receiver influences selection in two ways. First it may affect the selector's requirements. For instance, many young drivers like race cars, but most elderly drivers do not. Secondly, it may partially determine the goodness of the candidates. From Table 2-3 we see that the values of many PLA TDM measures depend on a PLA's size or personality, hence for different PLAs, these values become quite different, e.g. exhaustive testing is fine for small PLAs but impractical for large ones. In general, the goal of selection may not be choosing the alternative with the best "quality," but rather choosing the one which is most suitable for a particular application.

### 3.1.1.3. The domain of selection

Each selection problem deals with a set of alternatives from which a choice can be made. The collection of such alternatives is called the domain  $D$  of the selection. In general, since the availability of alternatives may be limited and alternatives may be a function of the receiver, the solution space  $S$  for a particular selection problem is a subset of the domain  $D$  in which each alternative takes its specific form with respect to the receiver. An alternative is a candidate for selection if it is in the solution space.

The size of a solution space is the number of candidates in  $S$ , and is also referred to as the size of the selection problem. For a non-trivial selection problem, the size must be greater than one.

In addition to the large number of candidates, the size of a solution space is usually not fixed. As shown in Chapter 2, the size of the TDM selection problem for a PLA is about 30 and increasing each year. Without being known by every selector, new elements may be added to the domain as they are created or made available. Therefore the selector often does not have complete knowledge of the domain.

#### **Multiple dimensions of a selection domain**

Each domain is characterized by a set of attributes. Different domains are distinguished by their attributes. For example, in the domain of automobiles, typical attributes are brand, year made, color, type of the engine, number of cylinders in the engine, size, MPG, price, and options.

For the domain of TDMs, the set of attributes includes fault masking, concurrent testing, self-testing, test application time, and area overhead. The attributes of a domain are defined based on knowledge about the domain.

Let  $ATT(D) = (att_1, att_2, \dots, att_N)$  be a set of attributes of a domain  $D$ . The dimension of the solution space refers to the number of attributes of the domain. Among the multiple attributes of a domain, there are some principle attributes and some subordinate attributes. The principle attributes characterize the main feature of the domain, for example, the testability characteristics in TDM selection. The subordinate attributes are features caused by the principle attributes, for example, in TDM selection, they include the effect on the original design, requirements on test environment, and design cost.

Each attribute  $att_j$  has a unit of measurement, and all values for  $att_j$  are measured in the same unit. For TDM selection, some attributes and their units and value ranges are listed in Table 3-1. It is easy to see that most attributes have incommensurable units, hence their values are not directly comparable. Translating these attribute values into a comparable scale is an important and inevitable step in making a selection.

In an  $N$ -dimensional solution space, if the individual dimensions are mutually independent, a selection problem can be divided into  $N$  sub-

selection problems in  $N$  unidimensional subspaces. Unfortunately, the multiple attributes often interact and conflict with each other. Thus finding a globally optimal solution by making tradeoffs between attribute values becomes a hard problem.

### Candidates in a domain

Every candidate in a domain can be characterized in terms of the  $N$  attributes of the domain. Given a domain  $D$ , an applicable receiver type (e.g., a PLA with  $n$  inputs,  $m$  product terms and  $k$  outputs), and an attribute array  $ATT(D) = (att_1, att_2, \dots, att_N)$ , for every candidate  $c_i$  in the domain, an evaluation vector

$$(v_{i1}, v_{i2}, \dots, v_{iN})$$

can be derived from knowledge about the domain attributes and the candidates, where  $v_{ij}$  is the value of attribute  $att_j$  using candidate  $c_i$ , and each  $v_{ij}$  is either a constant or a function of the receiver's parameters. For a specific receiver, the value of each  $v_{ij}$  can be determined. Therefore every candidate is uniquely described by a collection of its attribute values. In the following, we will often denote a candidate by its evaluation vector with respect to a given receiver, i.e.

$$c_i = (v_{i1}, v_{i2}, \dots, v_{iN}).$$

Selection is partially based upon a comparison among multiple attribute values of candidates with a given requirement vector.



Attribute	Unit	value range
Fault models	N.A.	fault models discussed in Sec. 2.2.1
Fault coverage	percent	$0 \leq \text{reals} \leq 100$
Fault masking	N.A.	{ yes, no }
Concurrent testing	N.A.	{ yes, no }
Error latency	# of gate delays	non-negative integers
Self testing	N.A.	{ yes, no }
Function dependence	N.A.	{ yes, no }
Logic overhead	# of transistors	non-negative integers
Extra I/O connections	# of I/O lines	non-negative integers
Area overhead	percent	non-negative reals
Extra delay	# of gate delays	non-negative integers
Delay per test	# of gate delays	non-negative integers
# of test patterns	# of input stimuli	non-negative integers
Test storage	# of bits	non-negative integers
Test application time	$(\# \text{ of gate delays})10^{-6}$	positive reals
Means of test generation	N.A.	set of test creation methods
Means of test application	N.A.	set of BIT blocks & ATE
Means of response evaluation	N.A.	set of BIT blocks & ATE
Logic design cost	level of difficulty	{ 0, 1, 2, 3 }
Layout cost	level of difficulty	{ 0, 1, 2, 3 }

Note: "N.A." stands for "not applicable."

Table 3-1: TDM attributes and their units and value range

#### 3.1.1.4. Requirements

Requirements represent the criteria for selection in terms of the desired attribute values of the selection domain. Since the domain is multi-dimensional, the global requirements consists of multiple subrequirements, each for a subdimension. A set of requirements can be represented by a requirement vector

$$R = (r_1, r_2, \dots, r_N),$$

where  $r_i$  is the subrequirement for the  $i$ th attribute. Note that  $r_i$  can be either a single constant value, a constant value range, a function, or "don't care." A requirement vector  $R$  is incompletely specified if it contains "don't care" entries. Otherwise  $R$  is completely specified.

A requirement vector contains both the goals and constraints of selection. Goals are the requirements for principle attributes. Constraints are requirements for subordinary attributes. Sometimes there is no clear cut distinction between goals and constraints, and they are handled in the same way. Therefore in the following, unless otherwise specified, we will refer to requirements rather than goals and constraints.

The selection process is requirement-driven. There is no general reason for a candidate to be good or bad without a requirement. As evidence is the fact that cars having vastly different features are purchased by different people because they have different needs and preferences.

### 3.1.1.5. Priorities of attributes

In general, all attributes are not equally important. To specify the relative importances of attributes in making a selection, a weight vector

$$W = (w_1, w_2, \dots, w_N)$$

is used, where  $w_i$  is a numeric value representing the relative importance of  $att_i$ . We use the convention that if  $att_i$  is more important than  $att_j$ , then  $w_i > w_j$ . The weight vector is defined by a selector based on application, environment and other subjective and/or objective factors.

### 3.1.2. The selection problem and selection process

The general selection problem can be described by a 3-tuple

$$S = \langle \text{domain, requirements, receiver} \rangle,$$

where domain specifies the set of candidates, requirements are specified by the selector and define goals and constraints associated with the selection process, and receiver is the element for which the selection is made. For example, in buying a car, domain = {automobiles}, requirements may be (price < \$20,000), (MPG > 28), and receiver can be the buyer himself. In this chapter, we only consider the problem of selecting one alternative from a given domain for single applicable receiver.

A candidate  $c_i$  is called an ideal solution (we will just call it a solution) of  $S$  if  $c_i$  satisfies  $R$ . A good solution of  $S$  is a  $c_i \in D$  such that  $c_i$  satisfies  $R$ , and for any other  $c_j \in D$  such that  $c_j$  satisfies  $R$ ,  $c_i$  is no worse than  $c_j$ . The precise definitions of "satisfy" and "worse" require a deep discussion, and will be given in the next section.

The selection problem S can be stated as follows.

Given a domain D, an initial requirement vector R, and a receiver RC, select a candidate  $c_i \in D$  such that  $c_i$  satisfies R and  $c_i$  is a good solution for R and RC. If no good solution exist, adjust R until a good solution is found and as long as R remains acceptable to the selector. If no such R can be found, no solution exists.

Selection is usually not a simple straightforward process, but rather a sequence of actions including specifying a set of requirements, searching for available candidates, comparing candidates and requirements, making tradeoffs between options such as changing requirements and weights, selecting a new candidate, and applying the selected alternative to the receiver for verification. We call this a selection process. Backtracking may happen at various stages of a selection process.

### **3.1.3. Characteristics of a selection process**

The general characteristics of a selection process are as follows.

- The selection problem is to choose the most suitable candidate for a given set of requirements and receiver. The result of selection is dependent on the requirements specified by the selector, the characteristics of the receiver, and the candidates in the domain of selection. Because of the diversity of goals and constraints, there is usually no universal or absolutely optimal solution.

- Requirements are changeable. The initial requirements may often be too difficult to satisfy. This is partly because the selector tends to overspecify a problem, e.g. to expect maximal results with minimal cost. Also, the selector may not be aware of the candidates in the domain of selection. However as we frequently experience in shopping, failing to satisfy initial requirements does not mean no selection can be made. A selector can modify the requirements until a satisfactory result is obtained. This is quite different from a constraint satisfaction problem [61] where constraints are strict and the problem is to find a solution which satisfies all constraints. It is also different from diagnosis problems solved by programs like MYCIN, where a set of syndromes exist and the problem is to select the most probable disease which manifests those syndromes.
- Tradeoffs between different requirements are often necessary. In a multi-dimensional solution space, the selection problem is complicated due to conflicts among attribute requirements, i.e., satisfying one requirement may invalidate another. For example, a high performance car is usually also expensive.
- Complete domain knowledge is required to make the best choice. However most selectors are not experts in the domain of selection. Therefore advice from expert consultants concerning the domain is often needed.

- Searching in the solution space is relatively simple. Once complete domain knowledge is obtained, an evaluation matrix can be built which consists of an evaluation vector for every element in the solution space. Once a receiver and a set of requirements are specified, the elements of the matrix can be evaluated, resulting in numeric and/or logical values. Then the task of searching for a solution becomes the task of comparing the requirement vector with evaluation vectors and identifying a candidate which satisfies all requirements.
- Human factors are important in selection. The selector has the ability to control the selection process. Advice from domain experts may help a selector to make better tradeoffs.

These characteristics make the selection problem unique. People usually solve such problems with the help of domain experts. Unfortunately, such experts are not always available. Our goal is to build a knowledge based expert consultant system which helps to solve selection problems. A prototype system has been built which helps a designer to select a TDM for a PLA. The design principles of this system will be described in this chapter.

### 3.2. The evaluation function

The general goal of selection is to choose a "good" candidate for a set of given requirements and a given receiver. What is a "good" candidate? Given two candidates with multiple attribute values, how does one judge which is better? To answer these questions, a function for evaluating candidates in a solution space must be defined. In this section we first analyze some properties of attributes which affect comparison and evaluation of attribute values, and then define the comparison operation in each subdimension. Next we introduce an important function in making a selection--the penalty-credit function, and finally give a score function which evaluates candidates globally.

#### 3.2.1. Properties of attributes

Each attribute of a domain corresponds to one subdimension in a multi-dimensional space. To compare different values of an attribute, we need to define an ordering relation for every subdimension. The ordering relation cannot be defined in the same way for all subdimensions. For example, for some attributes larger values are preferable, e.g., as far as fault coverage is concerned, 95% is better than 80%. But for other attributes, such as area overhead, smaller values are preferred, e.g. 80% is better than 95%. The ordering relation in a subdimension is dependent on the properties of the corresponding attribute.

### **Complex attributes vs simple attributes**

An attribute is a simple attribute if it only takes single values. Non-simple attributes are called complex attributes, whose values are sets which may be a singleton. For example, the fault model is a complex attribute whose values may be any subset of all possible fault models. Test storage is a simple attribute whose value is a single number.

### **Logical attributes vs numeric attributes**

A simple attribute may be either numeric or symbolic. A simple attribute is numeric if its values are in the real domain  $\mathcal{R}$ . Otherwise it is called a symbolic attribute. Symbolic attributes which only take binary values are also called logical attributes. For example, self-testing is a logical attribute whose value is either YES or NO. Area overhead is a numeric attribute whose value is a real number. Since most symbolic attributes are logical, we will mainly deal with logic symbolic attributes.

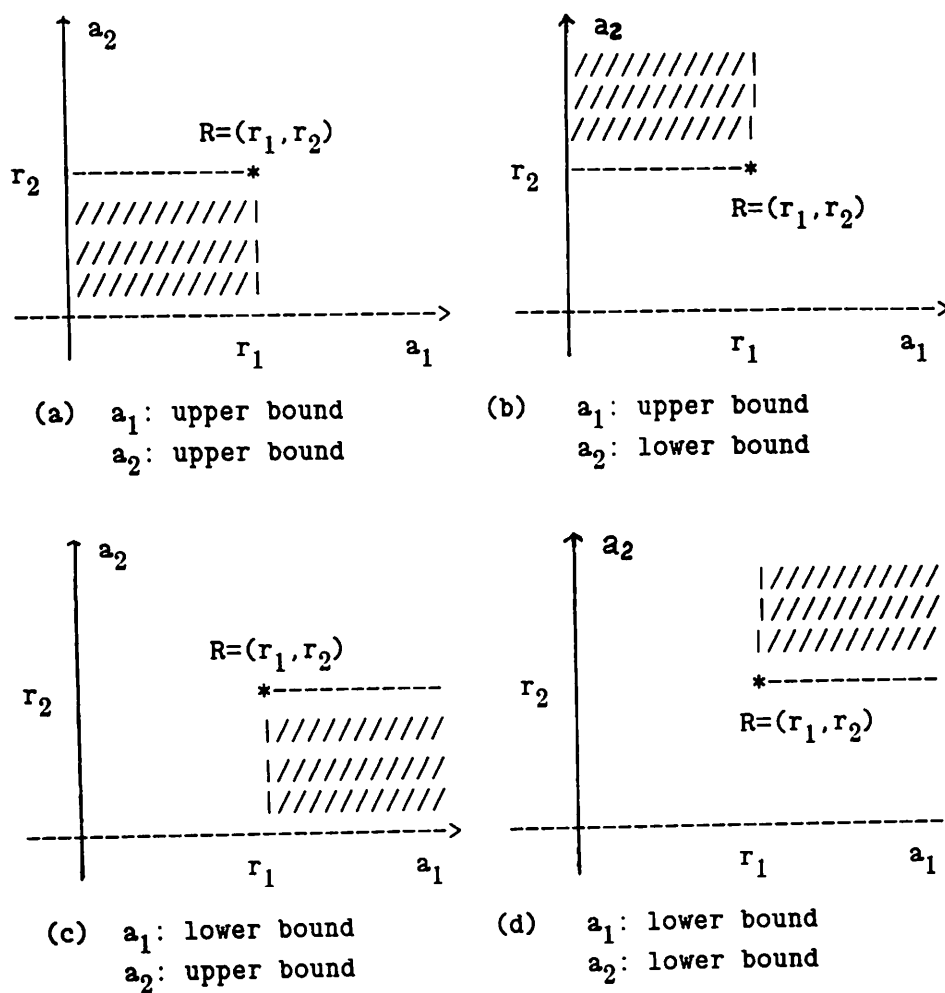
### **Upper-bound attributes vs lower-bound attributes**

A numeric attribute  $att_i$  is an upper-bound attribute if the requirement  $r_i$  specifies an upper bound for its value. For example, area overhead is an upper bound attribute, since by requiring 20% area overhead, it is implied that the actual area overhead should be no more than 20%. For an upper-bound attribute, a value  $x_1$  is better than  $x_2$  if  $x_1 < x_2$ . A value satisfies a requirement if it is less than or equal to the requirement.

A numeric attribute  $att_j$  is a lower-bound attribute if the requirement



$r_j$  specifies a lower bound for  $att_j$ 's values. For example, fault coverage is a lower bound attribute, because by requesting 90% fault coverage, it is implied that the actual fault coverage should be no less than 90%. For a lower-bound attribute, a value  $x_1$  is better than  $x_2$  if  $x_1 > x_2$ . An attribute's value satisfies a lower bound requirement if it is greater than or equal to the requirement.



**Figure 3-1:** Illustration of upper-bound and lower-bound property

A graphical illustration of the upper-bound and lower-bound property for numeric attributes is given in Figure 3-1. Suppose  $a_1$  and  $a_2$  are two

numeric attributes which define a two dimensional solution space. A requirement vector  $R = (r_1, r_2)$  corresponds to a point in this solution space. In Figure 3-1 (a),  $a_1$  and  $a_2$  are both upper bound attributes. The region in which solutions can possibly exist is the shaded area in the lower left corner, since if there is a candidate  $c_i = (v_{i1}, v_{i2})$  which satisfies  $R$ , then  $v_{i1} \leq r_1$  and  $v_{i2} \leq r_2$ . If  $a_1$  is an upper-bound requirement and  $a_2$  is a lower-bound requirement, the region which contains a solution moves to the upper left corner, as shown in Figure 3-1 (b). Similarly, Figure 3-1 (c) and (d) indicate two other situations.

The value of a logical attribute  $att_i$  may be either YES or NO. One value is stronger than the another if the former is more favorable. A logical attribute  $att_i$  is an upper-bound attribute, if for  $att_i$  YES is stronger than NO. Similarly, a logical attribute  $att_j$  is a lower-bound attribute, if for  $att_j$  NO is stronger than YES. For example, self-testing is an upper-bound attribute since it is usually more desirable to make a circuit self-testing. Test generation is a lower-bound attribute because it is better to have a TDM which does not need test generation.

A complex attribute  $att_i$  is an upper-bound attribute, if, for any requirement  $r_i$ , acceptable values of  $att_i$  are all subsets of  $r_i$ . For example, means for test application is a complex attribute. A requirement for test application means specifies a set of hardware available for real time test application. Any TDM which requires more than what can be provided

would not be used. For an upper-bound complex attribute,  $v_1$  is better than  $v_2$  if  $v_1$  is a subset of  $v_2$ .

A complex attribute  $att_j$  is a lower-bound attribute, if, for any requirement  $r_j$ , acceptable values of  $att_j$  contain  $r_j$ . For example, fault model is a lower-bound attribute. A requirement for fault model specifies a minimal set of fault classes that must be detected. Any TDM whose fault model contains these fault classes is acceptable for selection, and TDMs which cannot handle these fault classes would not be selected. For a lower-bound complex attribute,  $v_1$  is better than  $v_2$  if  $v_1$  contains  $v_2$ .

Properties of attributes can be determined from knowledge about the domain, and may be changed by a selector. We have analyzed these three properties of attributes for the TDM domain. A summary is given in Table 3-2.

### 3.2.2. The comparison function

Knowing the properties of attributes, a partial ordering relation " $\Leftarrow$ " can be defined for each subdimension of a given domain, with the definition that  $x \Leftarrow y$  iff  $y$  is better than or equal to  $x$ .

#### Definition 3.1:

1. Let  $x$  and  $y$  be two values of an upper bound numeric attribute.

Then  $x \Leftarrow y$  iff  $x \geq y$ .

attribute	simple/ complex	numeric/ logic/set	upper-bound/ lower-bound
Fault model	complex	set	lower-bound
Fault coverage	single	numeric	lower-bound
Fault masking	single	logic	upper-bound
Concurrent testing	single	logic	upper-bound
Error latency	single	numeric	upper-bound
Self testing	single	logic	upper-bound
Function dependence	single	logic	upper-bound
Logic overhead	single	numeric	upper-bound
Extra I/O pins	single	numeric	upper-bound
Area overhead	single	numeric	upper-bound
Extra delay	single	numeric	upper-bound
Delay per test	single	numeric	upper-bound
# of test patterns	single	numeric	upper-bound
Test storage	single	numeric	upper-bound
Test application time	single	numeric	upper-bound
Test generation means	complex	set	upper-bound
Test application means	complex	set	upper-bound
Response evaluation means	complex	set	upper-bound
Logic design cost	single	numeric	upper-bound
Layout cost	single	numeric	upper-bound

Table 3-2: Properties of TDM attributes

2. Let  $x$  and  $y$  be two values of a lower bound numeric attribute.

Then  $x \Leftarrow y$  iff  $x \leq y$ .

3. Let  $x$  and  $y$  be two values of an upper bound logical attribute.

Then  $x \Leftarrow y$  iff  $x = \text{NO}$  and  $y = \text{YES}$  or  $x = y$ .

4. Let  $x$  and  $y$  be two values of a lower bound logical attribute.

Then  $x \Leftarrow y$  iff  $x = \text{YES}$  and  $y = \text{NO}$  or  $x = y$ .

5. Let  $x$  and  $y$  be two values of an upper bound complex attribute.

Then  $x \Leftarrow y$  if  $x \subseteq y$ .

6. Let  $x$  and  $y$  be two values of a lower bound complex attribute.

Then  $x \Leftarrow y$  if  $x \supseteq y$ .

7. Let  $x$  and  $y$  be two values of the same attribute.  $x$  is incompatible with  $y$  if  $y \not\Leftarrow x$ .

Let  $V_j$  be a collection of all possible values of  $\text{att}_j$ .

**Lemma 3.2:**  $(V_j, \Leftarrow)$  is a chain if  $\text{att}_j$  is numeric.

**Definition 3.3:**  $\text{worst}(\text{att}_j) \in V_j$  is the worst value of  $\text{att}_j$  if for any  $v \in V_j$ ,  $\text{worst}(\text{att}_j) \Leftarrow v$ .  $\text{best}(\text{att}_j) \in V_j$  is the best value of  $\text{att}_j$  if for any  $v \in V_j$ ,  $v \Leftarrow \text{best}(\text{att}_j)$ .

**Lemma 3.4:** For any numeric attribute  $\text{att}_j$  and any nonempty subset  $V_s$  of  $V_j$ , there exist a best value  $\text{best}(\text{att}_j)$  and a worst value  $\text{worst}(\text{att}_j)$  in  $V_s$ . Specifically, if  $\text{att}_j$  is an upper-bound numeric attribute,

$$\begin{aligned}\text{best}(\text{att}_j) &= \min\{v_{ij} \mid v_{ij} \in V_s\} \\ \text{worst}(\text{att}_j) &= \max\{v_{ij} \mid v_{ij} \in V_s\}.\end{aligned}$$

If  $\text{att}_j$  is a lower-bound numeric attribute,

$$\begin{aligned}\text{best}(\text{att}_j) &= \max\{v_{ij} \mid v_{ij} \in V_s\} \\ \text{worst}(\text{att}_j) &= \min\{v_{ij} \mid v_{ij} \in V_s\}.\end{aligned}$$

Proof of the lemmas follow directly from the definition of the  $\Leftarrow$  relation and the fact that a finite chain is a totally ordered set which always contains a minimum and a maximum element [46].

**Lemma 3.5:** For an upper-bound logical attribute  $\text{att}_j$ ,  
 $\text{worst}(\text{att}_j) = \text{NO}$  and  $\text{best}(\text{att}_j) = \text{YES}$ .

For a lower-bound logical attribute  $att_j$ ,  
 $worst(att_j) = YES$  and  $best(att_j) = NO$  .

For an upper-bound complex attribute  $att_j$  and a set of values  $V_j$ ,  
 $worst(att_j) = \cap V_j$  and  $best(att_j) = \cup V_j$ .

For a lower-bound complex attribute  $att_j$  and a set of values  $V_j$ ,  
 $worst(att_j) = \cup V_j$  and  $best(att_j) = \cap V_j$ .

The above definitions and lemmas indicate how to compare any two values of the same attribute. In a selection process, another type of comparison is important, namely the comparison between a value and a requirement for an attribute, because the real goodness of a value is relative to the requirement. A function *compare* is defined which returns the distance between a value  $v_{ij}$  and a requirement  $r_j$ . *Compare* is a complex function due to the fact that attributes of a domain have various properties.

**Definition 3.6:** Let  $R_j$  be a set of possible requirement values for attribute  $att_j$ , and  $V_j$  be a set of possible values of  $att_j$  which the candidates may possess. **Compare** is a function from  $R_j \times V_j \rightarrow \mathcal{R}$  defined as follows, where  $\mathcal{R}$  is the domain of real numbers.

For any  $r \in R_j$ ,  $v \in V_j$  and any  $j$ ,  
 $compare(r, v) = 0$ , if  $r = \text{don't care}$ .

Otherwise

$compare(r, v) = r - v$ , if  $att_j$  is upper bound numeric;

$compare(r, v) = v - r$ , if  $att_j$  is lower bound numeric;

$-1$ , if  $att_j$  is upper bound logical and  $r = YES$ ,  $v = NO$ ;

$compare(r, v) = 0$ , if  $att_j$  is upper bound logical and  $r = v$ ;

$$\begin{aligned}
 & 1, \text{ if } \text{att}_j \text{ is upper bound logical and } r=\text{NO}, v=\text{YES}; \\
 \text{compare}(r, v) &= \begin{cases} -1, \text{ if } \text{att}_j \text{ is lower bound logical and } r=\text{NO}, v=\text{YES}; \\ 0, \text{ if } \text{att}_j \text{ is lower bound logical and } r = v; \\ 1, \text{ if } \text{att}_j \text{ is lower bound logical and } r=\text{YES}, v=\text{NO}; \end{cases} \\
 \text{compare}(r, v) &= \begin{cases} -1, \text{ if } \text{att}_j \text{ is upper bound complex and } r \cap v = \emptyset; \\ 0, \text{ if } \text{att}_j \text{ is upper bound complex and } r \cap v \subset v; \\ 1, \text{ if } \text{att}_j \text{ is upper bound complex and } r \cap v = v. \end{cases} \\
 \text{compare}(r, v) &= \begin{cases} -1, \text{ if } \text{att}_j \text{ is lower bound complex and } r \cap v \neq r; \\ 0, \text{ if } \text{att}_j \text{ is lower bound complex and } r = v; \\ 1, \text{ if } \text{att}_j \text{ is lower bound complex and } r \subset v. \end{cases}
 \end{aligned}$$

By this definition,  $\text{compare}(r_j, v_{ij})$  returns a numeric value  $x$  representing the "distance" between the requirement for  $\text{att}_j$  and the value of  $\text{att}_j$  for candidate  $c_i$ . If  $x \geq 0$ , we say  $v_{ij}$  satisfies  $r_j$ . If  $x < 0$ ,  $v_{ij}$  does not satisfy  $r_j$ . The bigger  $x$  is, the better.

### Examples:

Case 1:  $\text{att}_j$  is an upper-bound numeric attribute.

$$\text{att}_j = \text{extra I/O pins}$$

$$r_j = 4, \quad v_{ij} = 6$$

$$\text{compare}(r_j, v_{ij}) = 4 - 6 = -2.$$

Case 2:  $\text{att}_j$  is a lower-bound numeric attribute.

$$\text{att}_j = \text{fault coverage}$$

$$r_j = 90\%, \quad v_{ij} = 95\%$$

$$\text{compare}(r_j, v_{ij}) = 95\% - 90\% = 5\%$$

Case 3:  $att_j$  is an upper-bound logical attribute.

$att_j = \text{fault masking}$

$r_j = \text{yes}, v_{ij} = \text{no}$

$compare(r_j, v_{ij}) = -1$

Case 4:  $att_j$  is a lower-bound logical attribute.

$att_j = \text{test generation}$

$r_j = \text{yes}, v_{ij} = \text{no}$

$compare(r_j, v_{ij}) = 1$

Case 5:  $att_j$  is an upper-bound complex attribute.

$att_j = \text{test application means}$

$r_j = \{\text{ATE, (shift register, ROM)}\}$

$v_{ij} = \{\text{ATE, (counter, ROM)}\}$

$compare(r_j, v_{ij}) = 0$

Case 6:  $att_j$  is a lower-bound complex attribute.

$att_j = \text{fault model}$

$r_j = \{\text{single-stuck, single-bridging}\}$

$v_{ij} = \{\text{single-crosspoint}\}$

$compare(r_j, v_{ij}) = -1$

**Definition 3.7:** A candidate  $c_i = (v_{i1}, v_{i2}, \dots, v_{iN})$  satisfies a set of requirements  $(r_1, r_2, \dots, r_N)$  if for all  $1 \leq j \leq N$ ,  $r_j \Leftarrow v_{ij}$ , or equivalently,  $compare(r_j, v_{ij}) \geq 0$ .



### 3.2.3. The penalty-credit function

The *compare* function determines the distance between a requirement and a value in a subdimension of the solution space. In many cases, however, not the magnitude of the distance but the effect of the difference dictates the selection. The actual effect may not be proportional to the distance.

**Example.** Suppose a designer wants to design a testable chip with  $q$  primary inputs and outputs. A TDM satisfying all other requirements needs three extra I/O pins ( $v=3$ ), while the requirement for extra I/O pins is two ( $r=2$ ).  $compare(r,v) = -1$ . If the maximum number of allowable pins to be added to the chip is two, the TDM is unacceptable. Whereas if it is possible to add another I/O pad on the chip, the TDM can still be considered. On the other hand, assume the chip has four spare pins, i.e.,  $r = 4$ .  $compare(r,v) = 1$ . If three pins are used for implementing the TDM and the 4th one still remains unused, the designer will not benefit from this positive distance. However if the designer can use the spare pin to enhance some functionality, he would give more credit to this positive distance.

In general, there are several typical cases.

- Some values are absolutely unacceptable. Normally for a numeric attribute  $att_j$ , there is a break point  $b_j \in V_j$  such that  $\forall v \in V_j$  and  $v \leq b_j$ ,  $v$  is unacceptable. For an upper-bound

logical attribute, the break point, if one exists, is NO. For a lower bound logical attribute, the break point is YES, if one exists.

- Some values may be accepted with penalty. Usually a requirement, either an upper-bound or a lower-bound, is not a strict bound, but has a partial acceptability range, analogous to a "fuzzy" bound, which is called the penalty range. This means that although values in this range do not meet the requirement, they may still be considered with some degree of penalty. The amount of penalty may depend on the negative distance between the requirement and the value.
- Some values are accepted without penalty. If a value just meets a requirement, it is accepted without any penalty. So far we only considered the case that a requirement is a single value. In general, a requirement may take on a value range  $[r_1, r_2]$ , called the acceptance range, e.g. area overhead may be between [30%, 20%]. In this case,  $r_1$  is called the lower requirement, and  $r_2$  is called the higher requirement. Any value within this range is accepted without penalty. The value may be accepted with a credit, and the credit may be a function of the value itself.
- Some values are accepted with credit. The values exceeding a

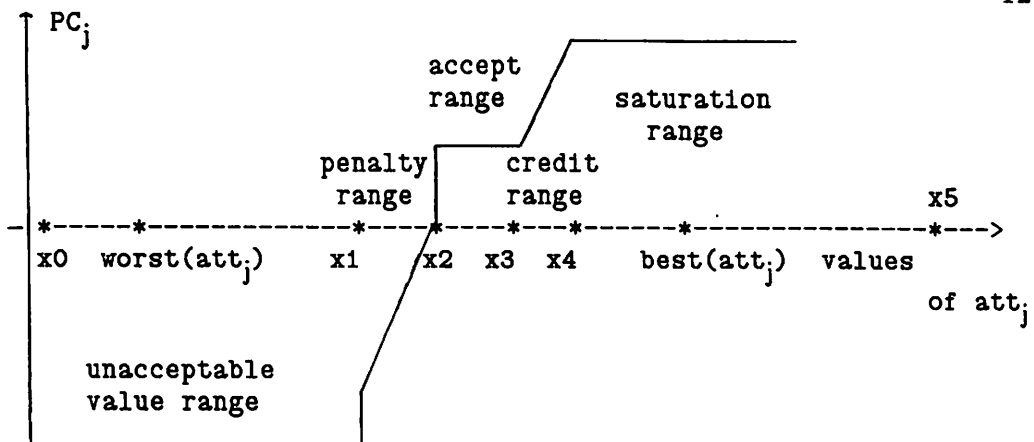
requirement constitute the credit range. A credit can be associated with each value in this range. The amount of credit may depend on the positive distance between the requirement and the value. The credit may not increase indefinitely, because it is possible that after some point, called the saturation point, the benefit caused by a larger positive distance would not increase any more. In this case, there is a saturation range in which all values are associated with a fixed credit.

A selection process is not simply matching the requirements and attribute values of candidates. The key consideration in making a selection is how much each attribute value affects the final solution. To describe the impact of an attribute value with respect to a requirement value, a penalty-credit function (PCF) should be established for every attribute.

**Definition 3.8:** Let  $PC_j$  denote the PCF for  $att_j$ .  $PC_j$  is a function from  $R_j \times V_j$  to  $\mathcal{R}$ .  $PC_j(r_j, x_1) > PC_j(r_j, x_2)$  if and only if  $x_1$  is a more preferable value than  $x_2$  for  $att_j$ .

### 3.2.3.1. General format of PCFs for numeric attributes

For a numeric attribute  $att_j$ , the general form for the PCF is shown in Figure 3-2. The horizontal axis represents possible values for  $att_j$  where values are sorted by the  $\leq$  relation. The vertical axis represents the amount of penalty or credit. Several critical points divide the chain of possible values of  $att_j$  into several important segments.



**Figure 3-2:** The typical form of the PCF for numeric attributes

- $[x_0, x_1]$  is the unacceptable value range where  $x_0$  is the lower bound of  $att_j$ 's value, and  $x_1$  is the break point.
- $(x_1, x_2)$  is the penalty range, where  $x_2$  is the lower requirement.
- $[x_2, x_3]$  is the acceptance range which is also the requirement range, and  $x_3$  is the higher requirement.
- $(x_3, x_4)$  is the credit range where credit increases with increasing distance from  $x_3$ .
- $[x_4, x_5]$  is the saturation range, where  $x_4$  is the saturation point and  $x_5$  is the upper bound of  $att_j$ 's value.

Note that  $worst(att_j)$  and  $best(att_j)$ , representing the worst value and the best value of  $att_j$  for a given set of candidates, respectively, may fall in any segment along the x-axis, not necessarily at the locations shown in Figure 3-2.

The segments on the x-axis divide the PCF into parts. Different functions, such as constant, linear, exponential and polynomial, can be used in different segments. A subfunction may be defined as a function of the distance of a given requirement and the values in the corresponding segment. For example, in Figure 3-2, the credit function used in the credit range is

$$h + p \times \text{compare}(x_3, v),$$

where  $h$  is the lowest value of the credit function,  $p$  is the slope, and  $v$  is any value in  $[x_3, x_4]$ .

Figure 3-2 illustrates a typical example of a PCF for numeric attributes. It is by no means the only one. A PCF may not be monotonic for a chain of values. In general, a PCF for a numeric attribute is of the following form.

$$PC_j(r, v) = \begin{cases} f_1(r, v), & \text{if } v \in [x_{11}, x_{12}], \\ f_2(r, v), & \text{if } v \in [x_{21}, x_{22}], \\ \dots \\ f_k(r, v), & \text{if } v \in [x_{k1}, x_{k2}], \end{cases}$$

where  $r \in R_j$ ,  $v \in V_j$ ,  $f_i$  is any well-defined function for  $r$  and  $v$ , and  $\{ [x_{i1}, x_{i2}] \mid i = 1, \dots, k \}$

is a (non-overlapping) partition of the chain containing all  $v \in V_j$ .

### 3.2.3.2. General format of PCFs for logical attributes

For a logical attribute, there are four possible combinations of requirements and values. The general form of a PCF for logical attributes is given in Table 3-3, where A, B, C and D are real numbers. A positive number represents a credit and a negative number represents a penalty.

requirement	value	penalty-credit
YES	YES	A
YES	NO	B
NO	YES	C
NO	NO	D

**Table 3-3:** General form of the PCF for logical attributes

**Example.** A typical PCF for a lower-bound logical attribute is shown in Table 3-4:

requirement	value	credit	penalty
YES	YES	A	
YES	NO	B	
NO	YES		C
NO	NO	D	

$$B \geq D \geq A \geq 0 \text{ and } C < 0.$$

**Table 3-4:** A typical PCF for a lower-bound logical attribute

### 3.2.3.3. General format of PCFs for complex attributes

Since the values and requirements of a complex attribute are sets, the PC function for a complex attribute consists of subfunctions for different subsets of the attribute's values. The general form of a PCF for complex attributes is as follows.

$$PC_j = \begin{cases} f_1, & \text{if } v \subseteq S_1, \\ f_2, & \text{if } v \subseteq S_2, \\ \dots & \\ f_k, & \text{if } v \subseteq S_k, \end{cases}$$

where  $v \subseteq V_j$ , the  $f_i$ 's are any well defined functions which return a numeric value,  $S_i$ 's are subsets of  $V_j$ , and  $\cup S_i \subseteq V_j$ .

Definition of PCFs are completely dependent on the selector and the application. There is no universal PCF that suits every case. However, the general forms of PCFs defined in this section provide a basic format for all possible PCFs and can be customized to represent various penalty or credit criteria. More examples of PCFs will be given in Chapter 4.

The PCFs play multiple roles in making a selection.

- PCFs convert incommensurable attribute values into a common measure, namely the resulting penalty-credit values.
- PCFs translate attribute values of different magnitudes into a comparable scale.
- PCFs specify the fuzziness of the requirements and provide more precise information about the criteria for selection which cannot be represented by the requirements alone.

It should also be noted that in defining the PCFs we assume that each  $PC_j$  can be assessed independent of other attribute values.

### 3.2.4. The score function

So far we have only considered evaluation of values in individual dimensions. In practice, a solution space is multi-dimensional. Each candidate corresponds to a point<sup>6</sup> in a multi-dimensional space. Comparisons among candidates and between candidates and requirements take place in multiple dimensions. The final result should be a combination of results from all subdimensions. A problem then is how to combine  $N$  things with different characteristics, meanings and units. Our solution is the following.

First, convert each of the  $N$  different attribute values of a candidate into a numeric value representing the contribution of that attribute to the global solution. Then combine the  $N$  unitless values into a single numerical *score* representing the overall quality of the candidate.

The first problem is solved by the PCFs defined in the last subsection. The second problem is solved by using a score function described next.

**Definition 3.9:** Let  $W = (w_1, w_2, \dots, w_N)$  be the weight vector for  $(att_1, att_2, \dots, att_N)$ , and  $PC_j$  be the penalty-credit function for  $att_j$ . The

score associated with a candidate  $c_i = (v_{i1}, v_{i2}, \dots, v_{iN})$  is

$$score(c_i) = F(PC_j(r_j, v_{ij}), w_j \mid j=1, \dots, N)$$

where  $F$  is a well-defined function which returns a numeric value.

---

<sup>6</sup>A candidate can be generalized to a subspace rather than a single point in the solution space. See section 4.2.3.



The score function should have the following desirable properties.

1. If every  $PC_j$  is monotone increasing with increasing values (ordered by the  $\Leftarrow$  relation) in the  $j$ th subdimension, and  $\exists c_i$  and  $c_k$  such that for all  $att_j$ ,  $v_{ij} \Leftarrow v_{kj}$ , then  $score(c_i) \leq score(c_k)$ . That is, if candidate  $c_k$  is better than candidate  $c_i$  in every attribute, the score of  $c_k$  is greater than the score of  $c_i$ .
2. For any two candidates  $c_i$  and  $c_k$ ,  $score(c_i) \leq score(c_k)$  if and only if  $(v_{k1}, v_{k2}, \dots, v_{kN})$  is more or equally preferable than  $(v_{i1}, v_{i2}, \dots, v_{iN})$ .
3. The score should reflect tradeoffs in satisfying different requirements. Each candidate's advantages and disadvantages tend to cancel each other in the score function. The candidates which offer more weighted credits than penalties should have higher scores.
4. The score should reflect precedences among attributes. The attributes with higher weights make more positive or negative contribution to the total score, such that a candidate satisfying important requirements is more likely to have a higher score, while a candidate lacking in important features is more likely to be rejected.

Many score function can be defined for different purposes. One example is

$$\text{score}(c_i) = \sum_j \text{PC}_j(r_j, v_{ij}) \times w_j.$$

In summary, we have defined partial ordering relations for every subdimension in a multi-dimensional solution space. Based on this relation, a *compare* function is defined which determines a distance between a value and a requirement in a subspace. This distance is then converted, by the penalty-credit function, to a number representing its effect on a global solution. Finally a score function is used to combine the numeric value of the penalty or credit of each attribute value of a candidate into a single score for the candidate. The higher the score, the more favorable is the candidate. Hence scores can be used to evaluate and compare candidates and suggest better solutions.

### 3.3. Analysis of solution space

#### 3.3.1. Structure of the solution space

For a non-trivial selection problem, the solution space is multi-dimensional and contains multiple candidates. Each dimension may be a 1-dimensional continuous space, a 1-dimensional discrete space or a multi-dimensional subspace. In general, a solution space is a complex multi-dimensional hierarchical space. Each dimension has its own value range and units.

#### **Examples.**

1. Area overhead in TDM selection defines a 1-dimensional

continuous space. Its possible values vary from 0 to any positive real number.

2. Extra I/O pins in TDM selection defines a 1-dimensional discrete space. Its possible values are non-negative integers.
3. Fault model in TDM selection defines a hierarchical discrete subspace. It may contain subspaces for single faults and multiple faults, respectively. Possible values in these subspaces are well-defined fault classes, e.g. stuck-at faults, crosspoint faults, and bridging faults.

The structure of a solution space can be represented by a *structure tree* constructed as follows.

1. The root of the tree is the name of the domain.
2. Partition attributes of the domain into levels. An attribute is on the 1st level if it defines a 1-dimensional subspace or a hierarchical subspace which is not contained in other subspaces. An attribute is on the  $j$ th ( $j > 1$ ) level if it defines a subspace within a subspace defined by an attribute on the  $(j-1)$ th level.

Division of dimensions and hierarchies is determined by their inherent relation between attributes of the domain and is not unique. As an example, the major part of the structure tree of the TDM domain is shown in Figure 3-3.

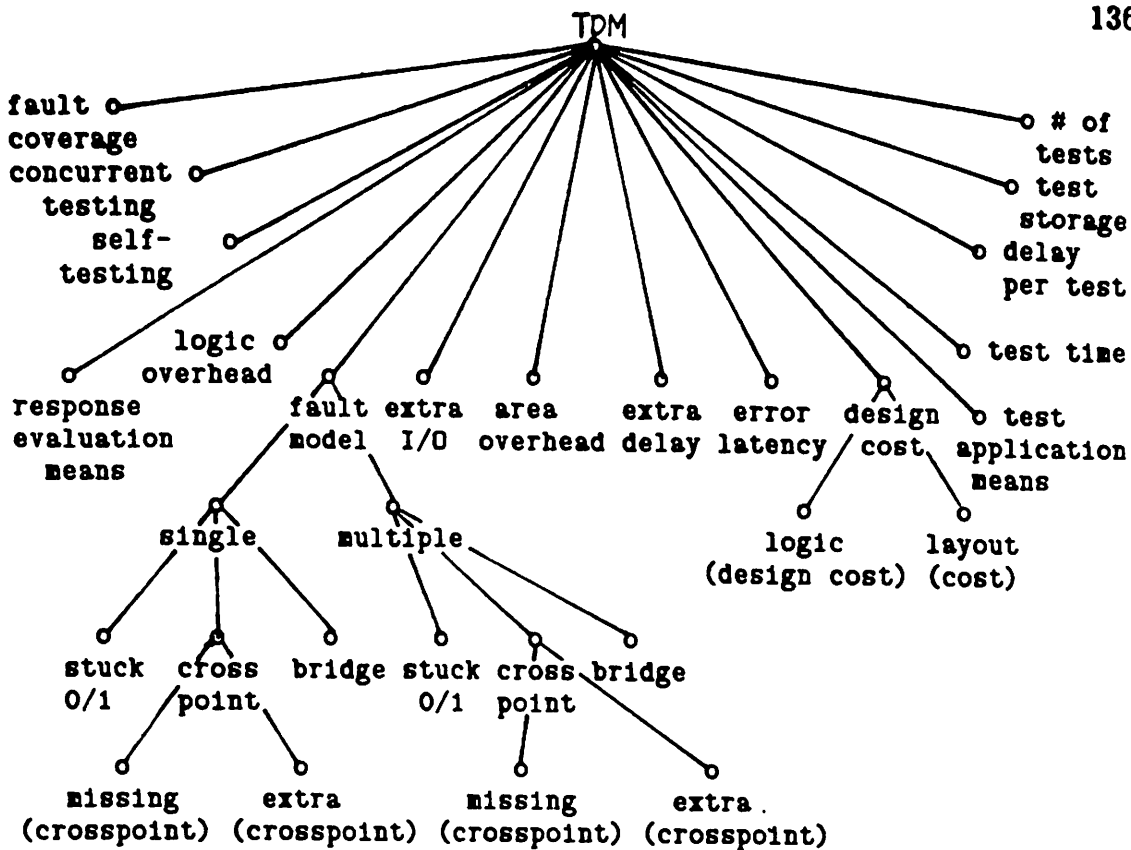


Figure 3-3: A structure tree of the TDM domain

**Theorem 3.10:** Any multiple hierarchical solution space can be flattened into a two-level non-hierarchical solution space.

**Proof:** Consider a subtree in a structure tree as shown in Figure 3-4 (a). One can remove the edge between  $att_j$  and  $att_{jk}$ , and make it connect to the father of  $att_j$ , for  $k=1, \dots, i$ . In the flattened tree shown in Figure 3-4 (b), the attribute of the leaf nodes inherit the attribute  $att_j$ , and the value of node  $(att_j.att_{jk})$  is the same as that of node  $att_{jk}$  of Figure 3-4(a). Since the node  $att_j$  is eliminated, whenever  $att_j$ 's value is required, it should be replaced by a linear combination of the values of  $(att_j.att_{jk})$ , for  $k = 1, \dots, i$ . This method can be recursively applied to a structure tree until it only contains a root and leave nodes.

The entire flattening operation can be done by calling the recursive procedure Flatten(root) given in Procedure 3-1.

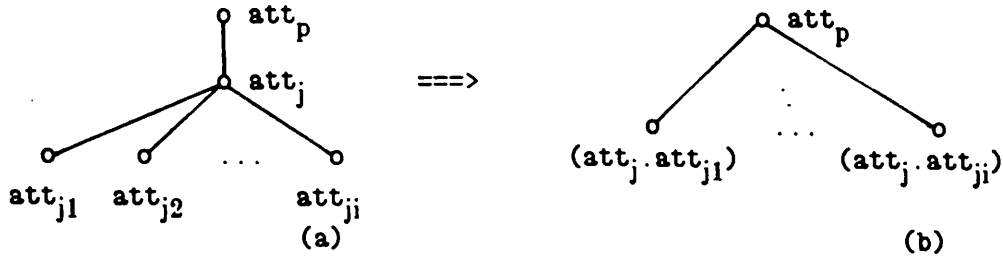


Figure 3-4: Flattening a subtree

\*\*\*\*\*

### Procedure 3-1: Flatten(NODE)

Do for each son node SN of NODE:

1. If SN is a leaf node, go to next SN. If all SNs have been checked, stop.
2. If SN's sons are not all leaf nodes, Flatten(SN).
3. If SN's sons are all leaf nodes, remove each son from SN, and make it the son of the NODE.

\*\*\*\*\*

In the following, without loss of generality, we will only discuss flattened two-level solution spaces.

### 3.3.2. Relationship among subdimensions

It is important to know the relationship between attribute values. In a multi-dimensional solution space, if subdimensions are all independent, a global problem can be partitioned into subproblems and solved in each subdimension. Unfortunately this is not the case for selection problems. The complex relationship between subdimensions makes selection complicated. There are three kinds of relations among subdimensions in a solution space.

1. Some subdimensions are independent of each other. A subdimension  $S_i$  is independent of  $S_j$ , if for every candidate  $c_k$ , the value  $v_{ki}$  is not related to  $v_{kj}$  in any way. For examples, a car's color is independent of the car's size.
2. Some subdimensions are directly dependent on others. A subdimension  $S_i$  is directly dependent on  $S_j$  if for every  $c_k$  in the domain,  $v_{ki} = f(v_{kj})$ . For example, test application time is directly dependent on the number of tests and delay per test, since test application time is equal to delay per test  $\times$  (# of tests + # of transition patterns).
3. Some subdimensions are implicitly related. For example, there is no explicit function relation between self-testing and area overhead. However on the average self-testable TDMs require more area than non-self-testable TDMs.

The dependencies between attributes are reflexive and transitive, but may not be symmetric.

### 3.3.3. Regions in a solution space

Given a selection problem  $S = \langle \text{domain, requirements, receiver} \rangle$ , a solution space for  $S$  is dynamically formed by the candidates in the domain which are applicable to the receiver and the requirements. Recall that knowing the receiver's parameters, all attribute values for a candidate can be computed, thus each candidate  $c_i$  becomes a point  $(v_{i1}, v_{i2}, \dots, v_{iN})$  in the space, as is the requirement vector  $(r_1, r_2, \dots, r_N)$ . Due to mixed relations between attribute values and the dynamic construction of the solution space, these points are scattered in the multi-dimensional solution space with no regular distribution. The minimal dimensions of the solution space is the number of  $r_j$ 's which are not equal to don't-care.

To simplify the discussion, consider only two attributes, e.g.,  $(att_1, att_2) = (\text{fault coverage, extra I/O pins})$ , and there are three candidates,  $C1 = (80, 3)$ ,  $C2 = (98, 5)$  and  $C3 = (95, 7)$  in the solution space, as shown in Figure 3-5. The locations of  $C1$ ,  $C2$  and  $C3$  divide the two-dimensional solution space into four regions, namely,

- region 1: the candidate solution region,
- region 2: the single critical failure region,
- region 3: the tradeoff region,
- region 4: the multiple critical failure region.

Any requirement vector  $(r_1, r_2)$  which contains constant entries also defines a point in this two dimensional space, and therefore must fall in one of the four regions. We will discuss each of these regions in detail.

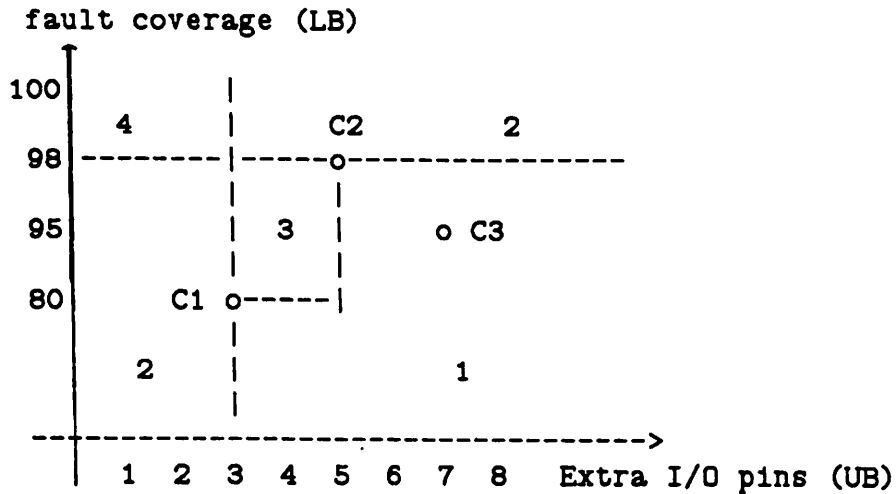


Figure 3-5: A sample two-dimensional solution space

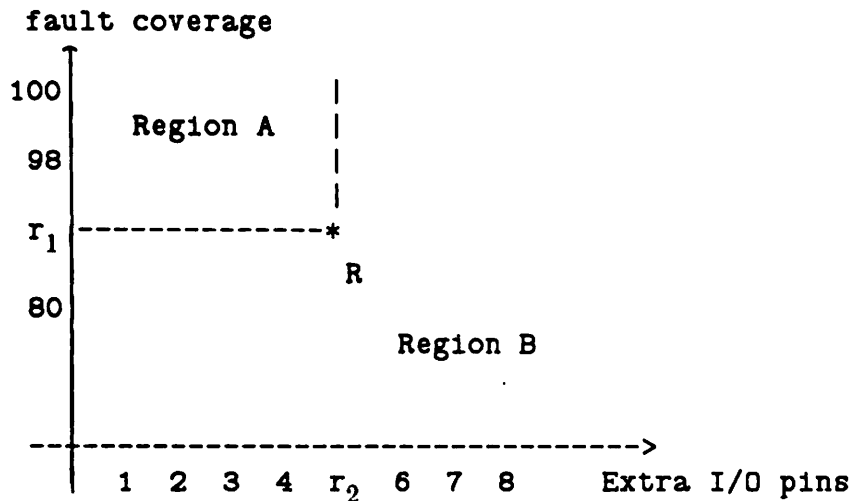


Figure 3-6: A two-dimensional solution space divided by the requirements

A requirement vector  $R = (r_1, r_2)$  divides the solution space into two regions, as shown in Figure 3-6. Region A is called the feasible region in which a solution may possibly exist. In region A, every point  $(x_1, x_2)$  is such that  $r_1 \leq x_1$  and  $r_2 \leq x_2$ . Therefore any candidate located in region A



satisfies the requirements. Region B is called the infeasible region, and no solution exists in this region. A solution exists if the intersection of the candidate solution region and the feasible region defined by the requirement vector is non-empty.

### 3.3.3.1. The candidate solution region

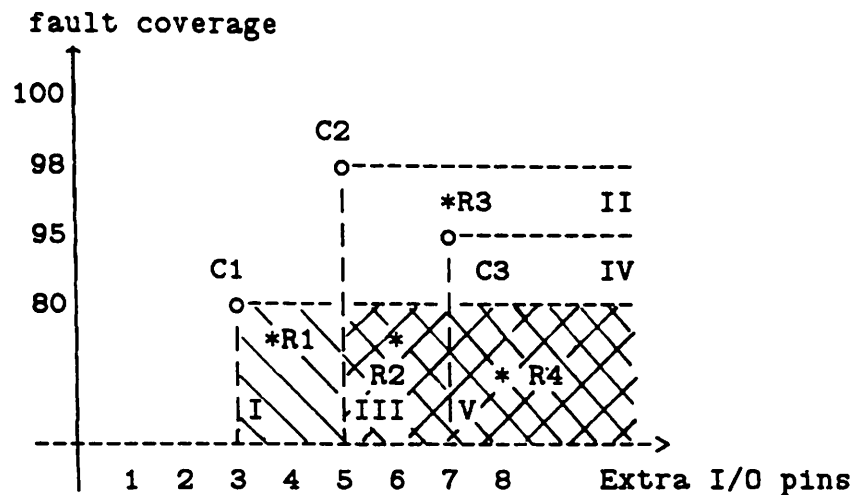


Figure 3-7: Region1: the solution region

Region 1 is the solution region. The primary attribute of region 1 is that for every requirement vector  $(r_1, r_2)$  falling in region 1,  $\exists c_i$  such that  $\forall j, r_j \leq v_{ij}$ . That is, there exists some candidate which satisfies all the requirements. A closer look at region 1 is shown in Figure 3-7. Any requirement in this region is satisfied by at least one candidate, e.g., R1 is only satisfied by C1, since C1 has higher fault coverage and uses less pins. Similarly, R3 is satisfied by C2 only, R2 is satisfied by both C1 and C2, and R4 is satisfied by all three candidates.

**Definition 3.11:** The satisfaction region of a candidate  $c_i = (v_{i1}, \dots, v_{iN})$  is the subspace in the N-dimensional solution space defined by

$$\{(x_1, x_2, \dots, x_N) \mid x_j \leq v_{ij}, \text{ for } j=1, \dots, N\}.$$

The satisfaction region of a set of candidates  $C_g = \{c_1, c_2, \dots, c_k\}$  is the intersection of the  $k$  satisfaction regions for  $c_1, \dots, c_k$ , that is, the subspace

$$\{(x_1, x_2, \dots, x_N) \mid x_j \leq \text{worst}\{v_{ij} \mid c_i \in C_g\}\}.$$

The satisfaction region of a set of candidates  $C' = \{c_1, c_2, \dots, c_k\}$  only is a subspace  $G$  in the satisfaction regions of  $C'$  such that for any  $c_i \notin C'$ , the intersection of  $G$  and the satisfaction region of  $c_i$  is empty.

In Figure 3-7, the satisfaction region of  $C_1$  is the shaded area at the lower right corner. The satisfaction region of  $C_1$  and  $C_2$  is the double shaded area. The candidate solution region in a solution space is the union of the satisfaction regions of all individual candidates.

In general, in a solution space containing  $m$  candidates, there are at most  $2^m - 1$  satisfaction regions corresponding to the  $2^m - 1$  non-empty subsets of candidates. Some of these regions overlap, and some are contained in others. The candidate solution region can always be partitioned into at most  $2^m - 1$  non-overlapping parts, called basic regions. In Figure 3-7, there are five basic regions.

- I. Satisfaction region of  $C_1$  only.
- II. Satisfaction region of  $C_2$  only.
- III. Satisfaction region of  $C_1$  and  $C_2$  only.

IV. Satisfaction region of C2 and C3 only.

V. Satisfaction region of C1, C2 and C3.

Each basic region defines a unique way in which a requirement vector can be satisfied. For example, if a requirement vector falls in region I, the solution to the selection problem is C1. If a requirement vector falls in region V, C1, C2 and C3 are all feasible solutions. When multiple solution exist, the final solution has to be chosen from among the qualified candidates.

### 3.3.3.2. The single failure region

**Definition 3.12:** Given a set of candidates  $C$  and a requirement vector  $R$ .

- A candidate  $c_i$  is said to have a single failure if there exists only one  $att_j$  such that  $v_{ij} \Leftarrow r_j$ , and for all  $att_k \neq att_j$ ,  $r_k \Leftarrow v_{ik}$ .
- A candidate  $c_i$  is said to have a multiple failure if there exist  $att_j$  and  $att_k$  ( $j \neq k$ ) such that  $v_{ij} \Leftarrow r_j$  and  $v_{ik} \Leftarrow r_k$ .
- A single critical failure refers to the case where there exists exactly one  $j$  such that for all  $c_i$ ,  $v_{ij} \Leftarrow r_j$ .
- A multiple critical failure refers to the case where there exist two or more  $j$  such that for all  $c_i$ ,  $v_{ij} \Leftarrow r_j$ .

Region 2 is the single critical failure region. If a requirement vector falls in this region, no solution exists since exactly one of the requirements cannot be satisfied by any candidate. For example, in Figure 3-8, the fault

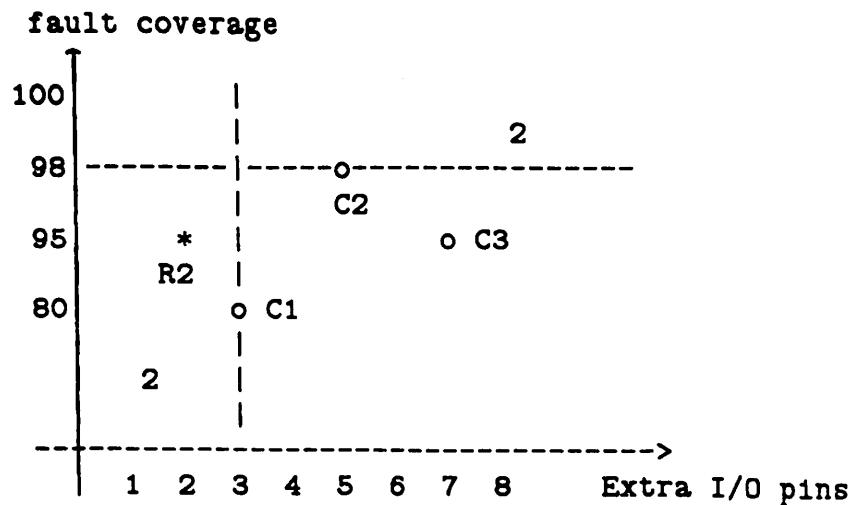


Figure 3-8: Region 2: the single critical failure region

coverage requirement of R2 is satisfied by C2, but the pin count requirement of R2 is not satisfied by any candidate. This kind of failure is critical because the requirement is unattainable. If it is not changed, no solution is possible. A requirement which causes this type of failure is called a critical requirement. In an N-dimensional solution space, a critical failure may be a single failure, or a part of a multiple failure. In case of a critical failure, in order to obtain a solution all critical requirements must be made less than the best possible value of the corresponding attribute among all candidates.

### 3.3.3.3. The tradeoff region

In region 3, each component of a requirement vector can be satisfied individually, but not by the same candidate. There is a conflict between the requirements. The requirements which cause the conflict are called conflicting requirements. To resolve such a conflict, tradeoffs must be made between conflicting requirements. In the example shown in Figure 3-9, a tradeoff must be made between higher fault coverage and more pins. If one

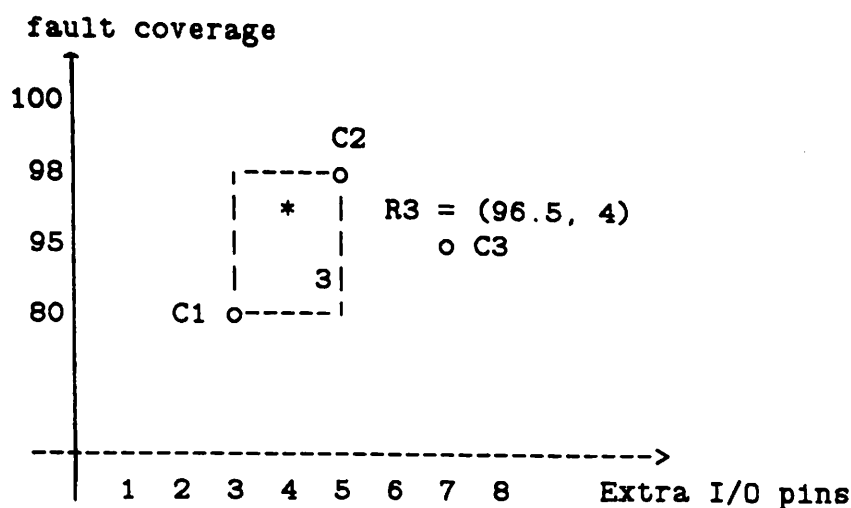


Figure 3-9: Region 3: the tradeoff region

more pin is allowed and fault coverage cannot be lowered, C2 is a solution, and the result obtained is (98, 5). If the pin count requirement is very tight and fault coverage can be lowered, C1 may be a solution, and the result is (80, 3). If both of the requirements can be relaxed, the decision depends on how to weigh the 16.5% lower fault coverage and one more pin. In a two-dimensional solution space, tradeoffs occur between a pair of requirements. In general, however, a conflict may be among sets of requirements, and multiple conflicts may exist. For a given requirement vector, there are several ways to make tradeoffs, each leading to a different solution. How to resolve such a situation is a hard problem.

Note that for C1 and C2, R3 presents a single failure. But for C3, R3 presents a multiple failure. It is common that single and multiple failures occur simultaneously. To remove a single failure, only one requirement needs to be changed. To remove a multiple failure, two or more requirements have to be changed.

### 3.3.3.4. The complete failure region

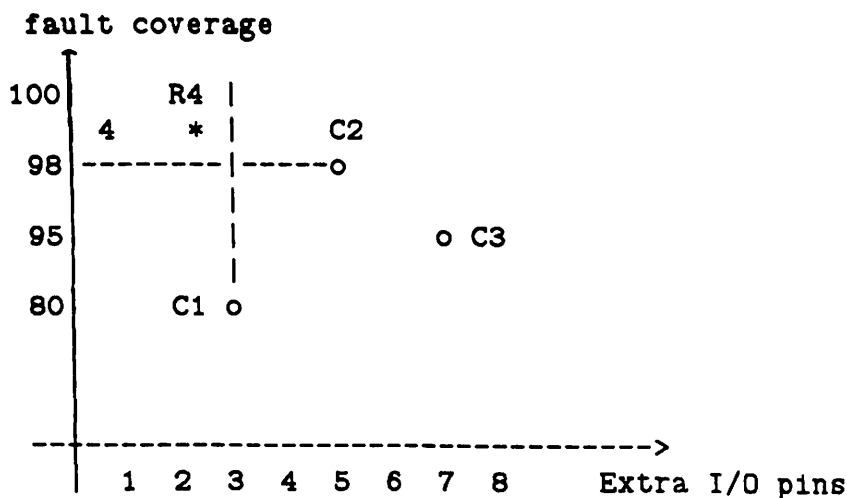


Figure 3-10: Region 4: the complete failure region

A requirement vector which lies in region 4 has the property that none of its components can be satisfied by any candidate. In other words, it presents a multiple critical failure. However in a general N-dimensional space, a multiple failure may not be critical or may be partially critical, i.e., some requirements may not be satisfiable by any candidates, while other requirements may be satisfied by one or more candidates. In this case, in order to get a solution, one should "move" R4 toward the candidate which is *closest* to it. The score function defined in the last subsection gives the weighted "distances" between the requirement vector and the candidates, and therefore may point to the correct direction for movement.

In practice, the design space is N-dimensional (e.g. in TDM selection for PLAs,  $N > 20$ ) and can be divided into numerous regions. Still there is only one solution region and one complete failure region. However there are N single failure regions and i-out-of-N i-failure regions, for  $1 < i < N$ . For

a random requirement vector, many failures may occur, which are either totally critical or partially critical or non-critical. The point here is that the selector's initial estimate of requirement values is usually overly optimistic or tight, and are made without knowledge of the solution space. Hence, via intelligent feedback, the requirements can be modified until a good selection is made. The next sections discuss how to efficiently deal with such situations.

#### 3.4. Ramification analysis in a selection processes

The domain of selection has multiple attributes with various relations existing among them. These interrelations cause conflicts between requirements, which then lead to failures in finding a solution. For example, one cannot find a car with eight cylinders which gets 40 MPG. Since a car with eight cylinders implies high gas consumption, these two requirements are contradictory. Most selectors are not experts in the domain of selection. They often do not know the interactions among attribute values, hence they may specify conflicting requirements. Searching under these requirements leads to failure, and it is often difficult to move into a feasible region without knowing the reason for the failure. To be intelligent and efficient in selection, it is important to understand the relation among various attribute values. This is the task of ramification analysis.

In this section, we first introduce ramification trees to represent relations among attribute values. We then give procedures for building

ramification trees from knowledge about the candidates. Finally we discuss some applications of ramification analysis in a selection process.

### 3.4.1. Ramification graphs

A ramification graph is a knowledge structure for representing relationships among attribute values upon which ramification analysis can be carried out.

**Definition 3.13:** Let  $C = \{c_1, c_2, \dots\}$  be a set of candidates where  $c_i = (v_{i1}, \dots, v_{iN})$ , and  $(att_x, v_x)$  represent an attribute value pair where  $v_x$  is a value of  $att_x$ . The direct consequence of  $(att_j, y)$  on  $att_p$  is  $(att_p, x)$ , if (1) all  $v_{ip}$ 's are known, (2)  $y \neq$  unknown, and (3)

- $att_p$  is logical, and  $\forall c_i, c_k \in \{c_l \mid c_l \in C \text{ and } y \leq v_{lj}\}$ ,  $v_{ip} = v_{kp} = x$ .
- $att_p$  is numeric, and  $x = \text{range}\{v_{ip} \mid c_i \in C \text{ and } y \leq v_{ij}\}$ , where  $\text{range}\{n_1, \dots, n_x\} = [\min\{n_1, \dots, n_x\}, \max\{n_1, \dots, n_x\}]$ .
- $att_p$  is complex and  $x = \text{worst}\{v_{ip} \mid c_i \in C \text{ and } y \leq v_{ij}\}$ .

Here for numeric and complex attributes " $\leq$ " stands for " $\Leftarrow$ ," while for logical attributes, " $\leq$ " stands for " $=$ ." Otherwise, the direct consequence of  $(att_j, y)$  on  $att_p$  is  $(att_p, \text{unknown})$ .

**Example.** Suppose there are five TDMs and six attributes, as shown in Table 3-5. Let  $att_j = \text{self-testing (st)}$  and  $y = \text{YES}$ .

- The direct consequence of  $(st, \text{yes})$  on test-generation (tg) is  $(tg, \text{no})$ ,



because for all self-testable TDMs, the values for test generation is NO.

- The direct consequence of (st, yes) on fault model (fm) is (fm, {a}), since  $\text{worst}\{\{a,c\}, \{a,b,c\}, \{a,d\}\} = \cap\{\{a,c\}, \{a,b,c\}, \{a,d\}\} = \{a\}$ .
- The direct consequence of (st, yes) on fault coverage (fc) is (fc, [95,100]), since  $\text{range}\{95, 100, 98\} = [95, 100]$ .
- The direct consequence of (st, yes) on area overhead(ao) is (ao, unknown), since some TDM's values for area overhead are variables and thus unknown at this time.

**Definition 3.14:** Given a set of attributes  $ATT = \{att_1, \dots, att_n\}$  and a set of candidates C. A ramification bush (R-bush) for an attribute  $att_j$ , denoted by  $RB(att_j)$ , is a two level directed tree in which

- there is a root node  $att_j$  which points to all internal nodes;
- there are  $t$  ( $1 < t \leq k$ ) internal nodes labeled  $(att_j, v_i)$ , for  $1 \leq i \leq k$ , where  $k$  is the number of different known values  $v_1, \dots, v_k$  which  $att_j$  can take, i.e.,  $\forall 1 \leq l \leq k, v_l = v_{ij}$  for some  $c_i \in C$ ;
- each internal node of the R-bush links to  $N-1$  leaf nodes corresponding to the  $N-1$  attributes in  $ATT - \{att_j\}$ ;
- each leaf node connecting to an internal node  $(att_j, v_i)$  is labeled  $(att_p, x_{ip})$ , where  $p \neq j$  and  $(att_p, x_{ip})$  is the direct consequence of  $(att_j, v_i)$  on  $att_p$ .

From the definition of R-bushes, there are three kinds of nodes in an R-bush, namely a root, some internal nodes and some leaf nodes. Every leaf in an R-bush must connect to an internal node which is directly connected to the root. Hence the length of any path from a leaf to the root is 2.

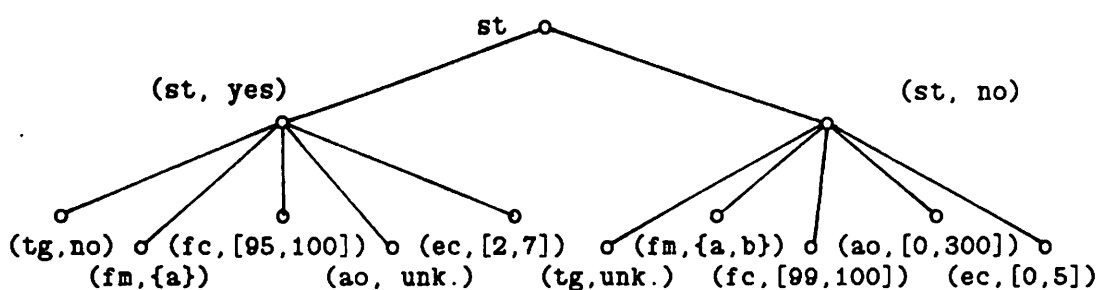
**Definition 3.15:** A complete ramification bush (R-bush) for an attribute  $att_j$  is an R-bush in which there are exactly  $k$  internal nodes, where  $k = | \{v_{ij} \mid c_i \in C\} |$ . Each internal node corresponds to a unique value of  $v_{ij}$ .

**Example.** For the TDMs in Table 3-5, the complete R-bush for self-testing is given in Figure 3-11. The leaf nodes represent the direct consequences of the possible values of self-testing on other attributes.

TDM	self-testing (st)	test generation (tg)	fault model (fm)	fault coverage (fc)	area overhead (ao)	extra connections (ec)
TDM1	no	yes	{a, b}	99	0	0
TDM2	yes	no	{a, c}	95	x1	2
TDM3	yes	no	{a,b,c}	100	x2	4
TDM4	yes	no	{a, d}	98	x3	7
TDM5	no	no	{a,b,c,d}	100	300	5

x1, x2, x3 > 0.

**Table 3-5:** A small evaluation matrix



**Figure 3-11:** The complete R-bush for self-testing

An incomplete R-bush is an R-bush which is a non-trivial subtree of a complete R-bush. An important subset of incomplete R-bushes is the conditional R-bushes defined below.

**Definition 3.16:** A conditional R-bush  $RB(att_j, v)$  is an incomplete R-bush for  $att_j$  which only contains one internal node  $(att_j, v)$ .

**Definition 3.17:** Let  $RB_1(att_j, v_1), \dots, RB_t(att_j, v_t)$  be distinct conditional R-bushes for  $att_j$ . The union of  $RB_1, \dots, RB_t$  is the R-bush for  $att_j$ , which has  $t$  internal nodes  $(att_j, v_1), \dots, (att_j, v_t)$ .

Conditional R-bushes are very useful when one is only interested in a particular value of an attribute. A complete R-bush is the union of all conditional R-bushes for the same attribute. Each internal node of an R-bush defines a branch of the R-bush. For the R-bush shown in Figure 3-11, the conditional R-bush  $RB(st, no)$  is the right branch of the R-bush defined by the internal node  $(st, no)$ .

### 3.4.2. Static ramification bushes

There are some general relations among attribute values which are independent of requirements or receivers. These relations are static and can be represented by static ramification bushes. Procedure 3-2 outlines how to build static R-bushes from an evaluation matrix.

\*\*\*\*\*

#### Procedure 3-2. Building a static R-bush

Inputs: A set of attribute  $A = \{att_1, att_2, \dots, att_N\}$ .

A set of candidates  $C = \{c_1, c_2, \dots \mid c_i = (v_{i1}, \dots, v_{iN})\}$ <sup>7</sup>

A chosen attribute  $\text{att}_j \in A$ .

Output: An R-bush representing static relations between  $\text{att}_j$  and other attributes' values.

1. Construct a root node labeled  $\text{att}_j$ .
2. Remove  $\text{att}_j$  from  $A$ .
3. Let  $V_j = \{v_{ij} \mid c_i \in C \text{ and } v_{ij} \text{ is known}\}$  and set  $k = |V_j|$ .
4. If  $k = 0$ , return  $\text{RB} = \emptyset$  and stop.
5. Construct  $k$  internal nodes and label them  $(\text{att}_j, x_{lj})$ , for  $x_{lj} \in V_j$  and  $l = 1, \dots, k$ , respectively.<sup>8</sup>
6. Do for every internal node  $(\text{att}_j, x_{lj})$ :
  - a. Find  $C_1 = \{c_i \mid x_{lj} \Leftarrow v_{ij}\}$ .
  - b. Find  $\text{DC}_1 = \{(\text{att}_p, v_p) \mid \text{att}_p \in A\}$ , where  $(\text{att}_p, v_p)$  is the direct consequence of  $(\text{att}_j, x_{lj})$  on  $\text{att}_p$  for candidates in  $C_1$ .
  - c. Construct a leaf node which connects to  $(\text{att}_j, x_{lj})$  for each attribute value pair in  $\text{DC}_1$ .

---

<sup>7</sup>Note: Some  $v_{ij}$  values may be variables because they are a function of the receiver.

<sup>8</sup>If  $\text{att}_j$  takes numeric values and  $k$  is very large, it is possible to define  $t < k$  boundary points and make  $t$  internal nodes, one for each of such points.

d. Associate the local candidate set  $C_1$  with  $(att_j, x_{ij})$ .

\*\*\*\*\*

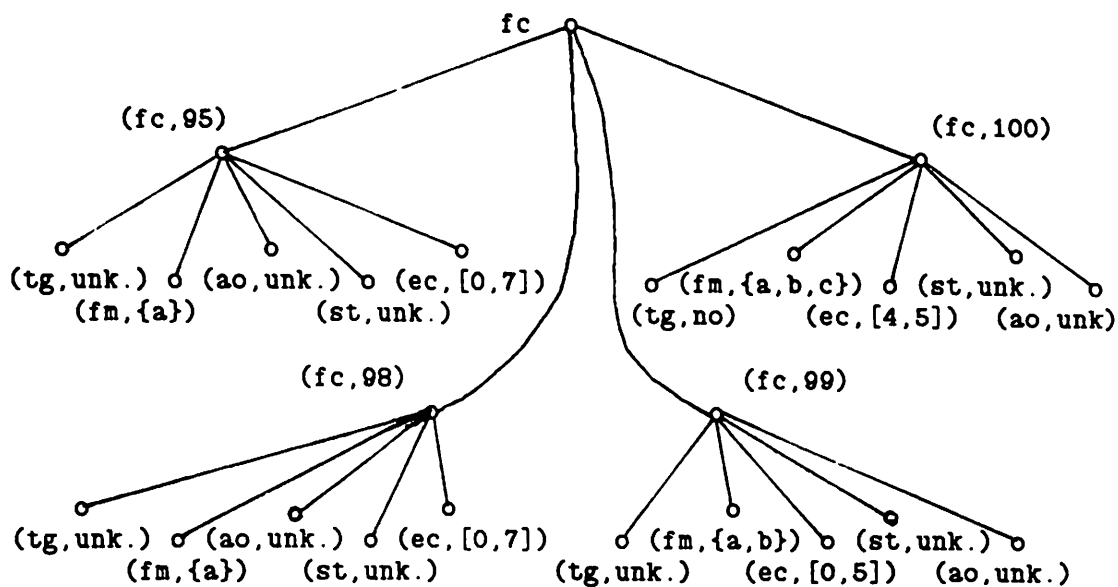


Figure 3-12: The static R-bush for fault coverage

**Example.** The static R-bush for fault coverage is given in Figure 3-12. Consider the upper right branch of the R-bush defined by  $(fc, 100)$ . It indicates that if the fault coverage must be 100%, test generation is not required; the number of extra connections is either 4 or 5; faults in classes a, b and c can definitely be detected, but the area overhead and the self-testing feature are not yet known, and finally the candidate set for the branch is  $\{TDM3, TDM5\}$ .

It is important to note that the ramifications of an attribute value may change with the candidate set  $C$ . The same attribute value can result in different ramifications for different subsets of  $C$ . For a fixed set of

candidates, a static R-bush  $RB(att_j)$  contains information about other attribute values that can be derived from the values of  $att_j$ . This information is true for any selector and receiver. A great deal of information can be obtained from static R-bushes which contain many constant values. But in the domain where many candidate attribute values vary as a function of the receiver, static R-bushes only provide limited information. In general, static R-bushes do not efficiently deal with numeric attributes, since their values may be unknown or too numerous to enumerate.

### **Construction of ramification trees**

A ramification bush indicates the impact of one attribute's value on others. Individual R-bushes for different attributes can be concatenated together into a ramification tree (R-tree) to show the ramification of multiple attribute values on remaining attribute values.

\*\*\*\*\*

### **Procedure 3-3. Building a complete ramification tree**

Inputs:           A set A of attributes.

                  A set C of candidates.

Output:           A complete R-tree representing static relations among all attribute values.

1. Set the candidate set  $T = C$ , and the attribute set  $AT = A$ .
2. Select an attribute  $att_j \in A$  such that values of  $att_j$  are known.

If no such attribute exists or  $T$  is empty, terminate with an empty R-tree.

3. Set  $\text{root} = \text{att}_j$ .
4. Remove  $\text{att}_j$  from  $AT$ .
5. Create the complete static R-bush  $RB(\text{att}_j)$  for  $T$  and  $AT$ .
6. Replace the node  $\text{att}_j$  with its R-bush.
7. Push all leaf nodes of  $RB(\text{att}_j)$  onto a stack. Associate the current  $AT$  and the local candidate set for the father of each leaf with every node.
8. If the stack is not empty, pop the stack; otherwise, output the R-tree and stop.
9. Let  $ND$  be the node  $(\text{att}_p, x_p)$  popped from the stack,  $T$  and  $AT$  be the candidate set and attribute set associated with  $ND$ .
10. Remove  $\text{att}_p$  from the current  $AT$ .
11.
  - a. If  $T$  is a singleton, or  $x_p$  is completely specified, go to 8.
  - b. If  $x_p = \text{unknown}$ , set  $RB = RB(\text{att}_p)$  for  $T$  and  $AT$ .

c. Otherwise, let RB equal the union of conditional R-bushes

$$RB(\text{att}_p, y), \text{ where } y = v_{ip} \text{ for some } c_i \in T \text{ and } x_p \Leftarrow y.$$

12. If RB is empty, go to 8.

13. Replace  $(\text{att}_p, x_p)$  with RB.

14. Push leaves of RB into the stack, together with AT and their candidate sets.

15. Go to 8.

\*\*\*\*\*

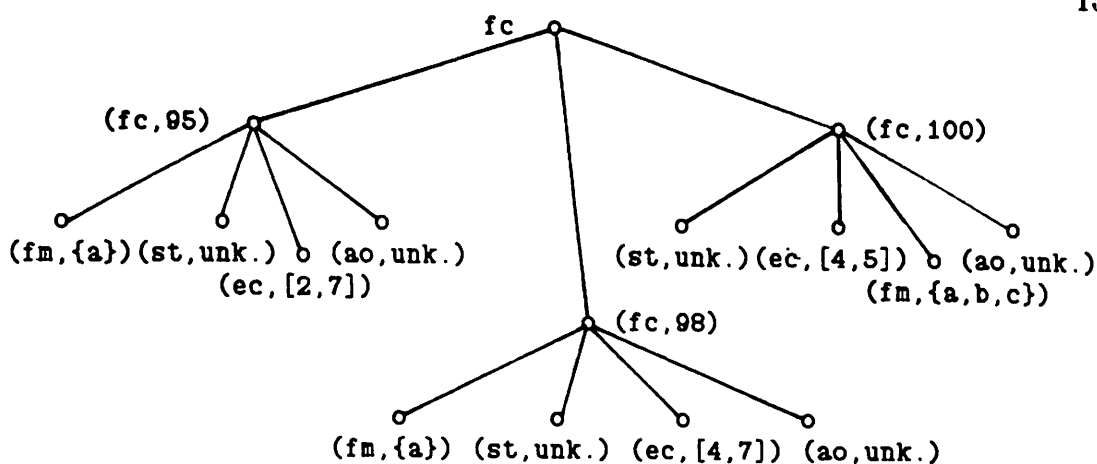
**Example.** Again consider Table 3-5. Suppose we start with the attribute test generation (tg). Initially the R-tree is the complete R-bush for tg, as shown in Figure 3-14 (a). Since the complete R-tree is large, we only demonstrate Procedure 3-3 by expanding one branch of the R-tree. First expand the leaf node (fc,[95,100]) under (tg,no) in the right branch. The candidate set associated with this node is  $T = \{\text{TDM2}, \text{TDM3}, \text{TDM4}, \text{TDM5}\}$ . Since under the condition that  $\text{tg} = \text{NO}$ , fc has only three possible values 95, 98 and 100, then

$$RB = RB(\text{fc},95) \cup RB(\text{fc},98) \cup RB(\text{fc},100),$$

as shown in Figure 3-13.

Comparing Figure 3-13 with the complete R-bush for fc in Figure 3-12,

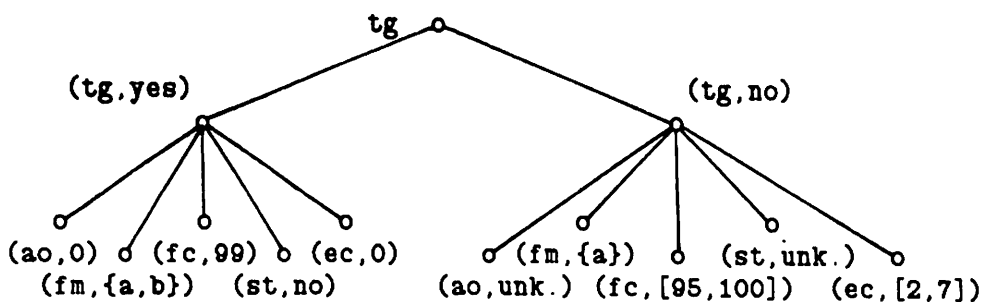




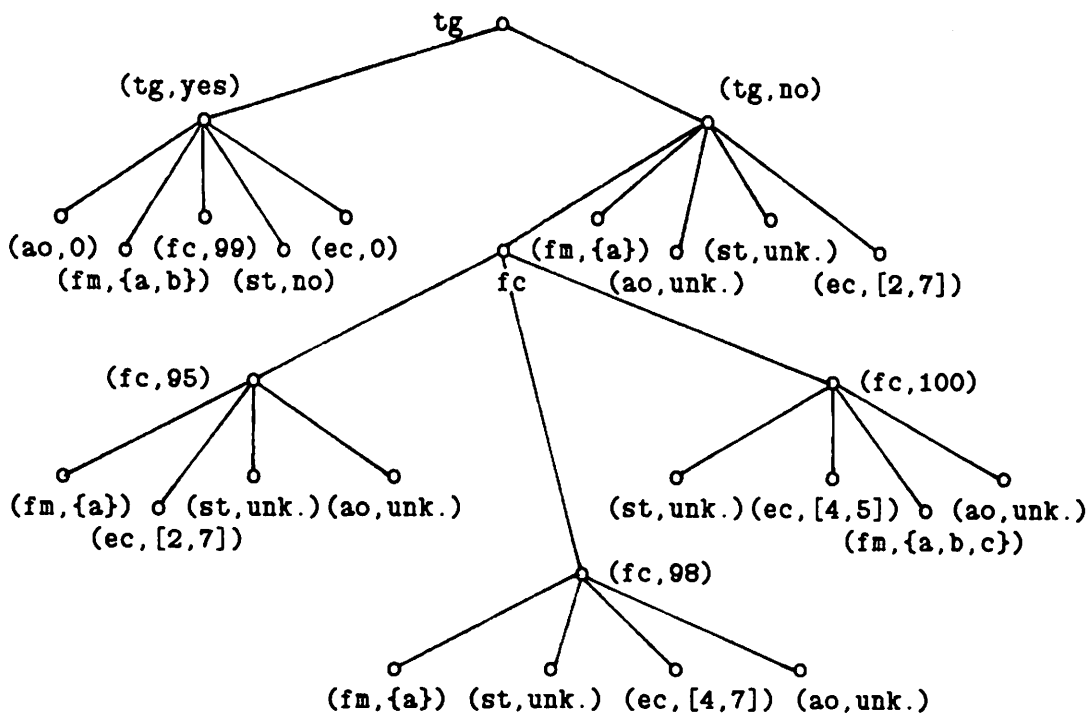
**Figure 3-13:** The R-bush for  $fc$  when  $tg = no$

note that the R-bush of Figure 3-13 does not have the branch defined by  $(fc, 99)$  and the leaf nodes labeled  $(tg, -)$ , because the value of  $tg$  has been specified at the top level of the tree and  $tg$  will no longer be considered.

At step 11 of procedure 3-3,  $(fc, [95, 100])$  is replaced by RB of Figure 3-13. The expanded R-tree is shown in Figure 3-14 (b), where many leaves, for example,  $(st, unk.)$  for  $RB(fc, 100)$  is still expandable. When  $(st, unk.)$  is popped from the stack,  $T = \{TDM2, TDM3, TDM4\}$  and  $AT = \{ao, ec, fm, st\}$ . An R-bush  $RB(st)$  for the current  $T$  and  $AT$  is built which has two branches  $(st, yes)$  and  $(st, no)$ . Finally the node  $(st, unk.)$  is replaced by  $RB(st)$ , and one branch of a completely expanded R-tree is shown in Figure 3-14 (c).



(a). The initial R-tree



(b). The partially expanded R-tree

Figure 3-14: A static R-tree

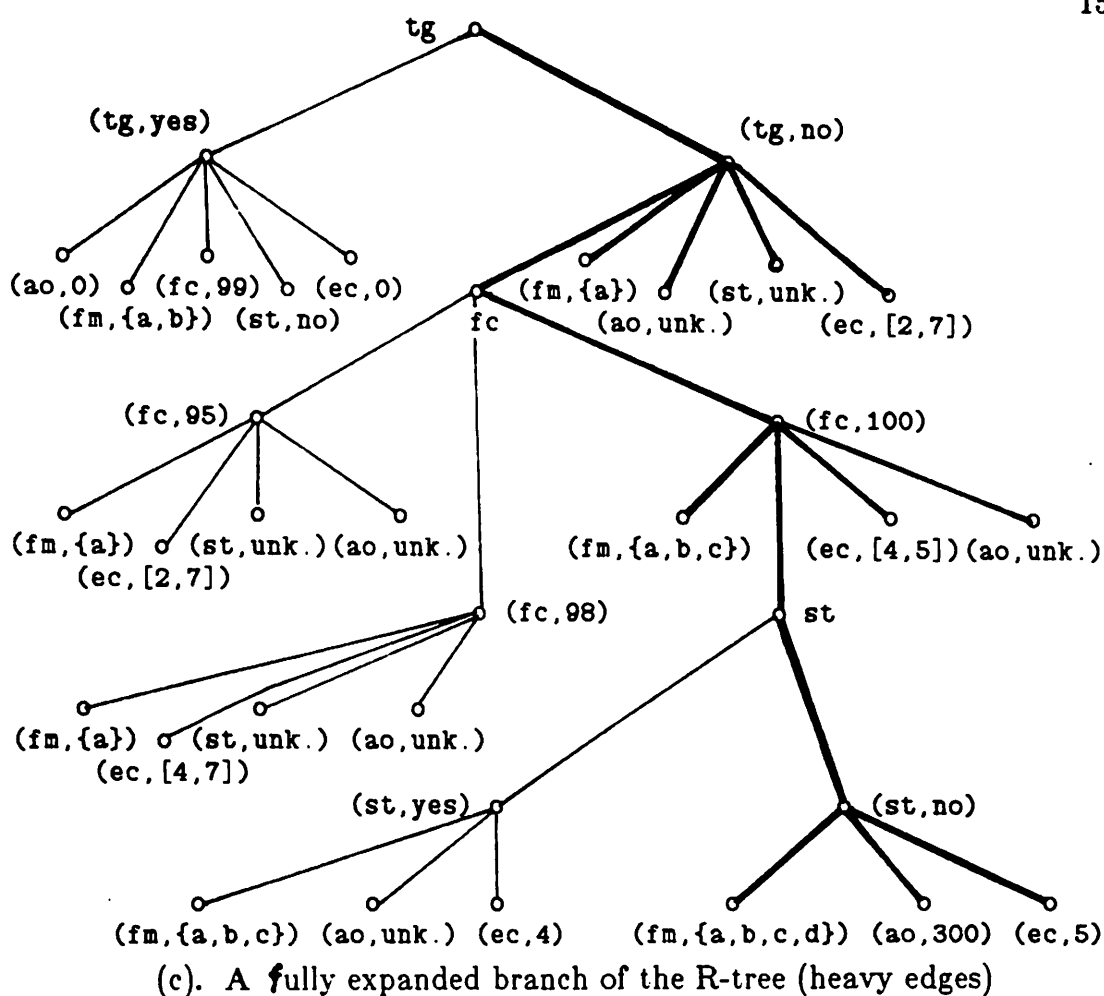


Figure 3-14: A static R-tree (cont'd)

**Definition 3.18:** A stage of an R-tree is a two-level subtree of the R-tree which forms an R-bush. A branch of an R-tree is a subtree of the R-tree that (1) contains the root, (2) consists of no more than one branch of the R-bushes from each stage, and (3) has leaves all of which are leaf nodes of the R-tree.

An R-tree, or a branch of an R-tree, consists of cascaded R-bushes. A branch of an R-tree can be uniquely defined by a set of internal nodes at different stages. For example, in the part of the R-tree shown in Figure

3-14 (c), there are three stages, namely the R-bush for *tg*, *fc* and *st*, respectively. One completely expanded branch is defined by  $\{(tg,no), (fc,100), (st,no)\}$ , and is illustrated by dark lines.

Every branch of a static R-tree defines a set of consistent requirements. A completely specified requirement vector  $R = (r_1, \dots, r_N)$  can be satisfied if there exists a branch in which there is no node  $(att_j, x_j)$  such that  $r_j \neq x_j$ , for  $1 \leq j \leq N$ .

A complete static R-tree may be very large. There are many different static R-trees for a set of candidates, since the R-bushes can be concatenated in different orders. These R-trees may provide different information. For example, the information given in Figure 3-11 (which can be considered as a one-stage incompletely expanded R-tree) is only partially contained in the R-tree in Figure 3-14 (c). Therefore a family of static R-trees is needed to represent all possible relations among attribute values.

### 3.4.3. Dynamic ramification trees

Static R-trees provide some information concerning possible combinations of feasible requirements. It is wasteful to generate all static R-trees, because there are too many of them and each contains some redundant information. In a real selection process, however, only one requirement vector is active at a time, and most parts of the complete R-tree may never be accessed, therefore there is no need to generate a

complete R-tree. To represent the useful relations among attribute values in a real selection process, dynamic ramification trees are defined.

**Definition 3.19:** Given a set of attributes  $ATT = \{ att_1, \dots, att_n \}$  and a set of candidates  $C$ . A dynamic R-bush (DRB) for a requirement  $r_j$ , denoted by  $DRB(ATT, C, r_j)$ , is a two level directed tree in which

- there is a root node  $att_j$  having a directed edge to an internal node  $(att_j, r_j)$ .
- the internal node of the DRB has directed edges to at most  $n$  leaf nodes corresponding to the  $n$  attributes in  $ATT$ .
- each leaf node is labeled  $(att_p, x_p)$ , where  $(att_p, x_p)$  is the direct consequence of  $(att_j, r_j)$  on  $att_p$ . In particular,  $(att_j, x_j)$  is a leaf node if and only if  $x_j \neq r_j$ .

\*\*\*\*\*

#### Procedure 3-4. Building a dynamic R-bush

Inputs:            A requirement  $r_j$  for  $att_j$ ,  $r_j \neq$  "don't care."  
                     A set of attributes  $A$  under consideration,  
                     A chosen attribute  $att_j \in A$ .  
                     A set of candidates  $C$ .  
                     A specific receiver.

Outputs:            An R-bush representing the direct consequences of  $r_j$ .  
                     A set of candidates which satisfy  $r_j$ .

1. If any  $c_i \in C$  has unknown values, use the receiver's data to make every  $v_{ij}$  known.

2. DRB =  $\emptyset$ .
3. Create a root  $att_j$  for DRB.
4. Create an internal node  $(att_j, r_j)$ .
5. Find  $T = \{ c_i \mid c_i \in C \text{ and } r_i \Leftarrow v_{ij} \}$ .
6. If T is empty, remove all nodes and return DRB =  $\emptyset$ .
7. Find  $DC = \{ (att_p, v_p) \mid att_p \in A \}$ , where  $(att_p, v_p)$  is the direct consequence of  $(att_j, r_j)$  on  $att_p$  for the candidates in T.
8. Let  $DC = DC - \{(att_j, r_j)\}$ .
9. Make each  $(att_p, r_p)$  a leaf node of DRB which connects to the internal node  $(att_j, r_j)$ .
10. Output DRB and T. Stop.

\*\*\*\*\*

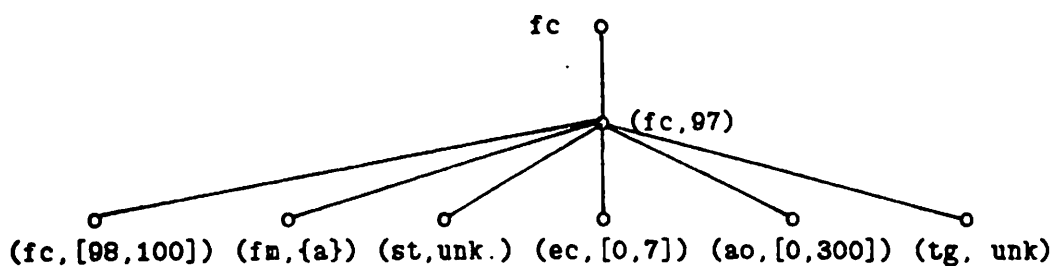
The differences between a dynamic R-bush and a conditional static R-bush are that a dynamic R-bush depends on a requirement and the receiver's characteristics; it may also indicate a refinement to the requirement. A static R-bush represents general ramifications of one attribute's values on other attribute values, while a dynamic R-bush only

represents the effect of a specific requirement on other attribute values in one selection process given a specific receiver. A dynamic R-bush may contain more specific leaf nodes, since some "unknown" nodes will become known.

TDM	self-testing (st)	test generation (tg)	fault model (fm)	fault coverage (fc)	area overhead (ao)	extra connections (ec)
TDM1	no	yes	{a, b}	99	0	0
TDM2	yes	no	{a, c}	95	35	2
TDM3	yes	no	{a, b, c}	100	42	4
TDM4	yes	no	{a, d}	98	17	7
TDM5	no	no	{a, b, c, d}	100	300	5

**Table 3-6:** A small fully specified evaluation matrix

**Example.** Suppose for a given receiver, the evaluation matrix in Table 3-5 takes on the values shown in Table 3-6 (only column ao has been changed). Let  $att_j = fc$  and  $r_j = 97$ . A dynamic R-bush  $DRB(fc, 97)$  is given in Figure 3-15, which is similar to the conditional R-bush  $RB(fc, 98)$  in Figure 3-13 and refines the expected value for fc.



**Figure 3-15:** A dynamic R-bush for fc

**Definition 3.20:** A dynamic ramification tree (DRT) corresponding to a requirement vector  $R=(r_1, \dots, r_n)$ , an attribute set A, a candidate set C, and a receiver is a single-branch R-tree consisting of a cascade of DRBs, in which internal nodes are in the set  $\{(att_1, r_1), \dots, (att_n, r_n)\}$ .

\*\*\*\*\*

### Procedure 3-5. Building a dynamic ramification tree

Inputs:            A requirement vector  $R = (r_1, r_2, \dots, r_N)$ .  
                     A set of candidates  $C = \{c_1, c_2, \dots \mid c_i = (v_{i1}, \dots, v_{iN})\}$   
                     A set of attributes  $A = \{att_1, att_2, \dots, att_N\}$ .  
                     A specific receiver.

Output:            An R-tree representing the ramifications of R.

1. Set  $DRT = \emptyset$ .  
     Set the candidate set  $T = C$ .  
     Set the attribute set  $AT = A$ .
2. Select an attribute  $att_j$  from  $AT$  such that  $r_j \neq$  don't-care.
3. Let  $att_j$  be the root of the DRT.
4. Use Procedure 3-4 to create the dynamic R-bush  $DRB(att_j, r_j)$  for the candidate set  $T$  and attribute set  $A$ .
5. If  $DRB$  is empty, return  $DRT = \emptyset$ . Stop.
6. Let  $T$  be the set of currently qualified candidates produced by Procedure 3-4.
7. Replace  $att_j$  with  $DRB$ .
8. Let  $SL$  be the set of leaves.
9. Check every  $(att_p, x_p)$  in  $SL$ :

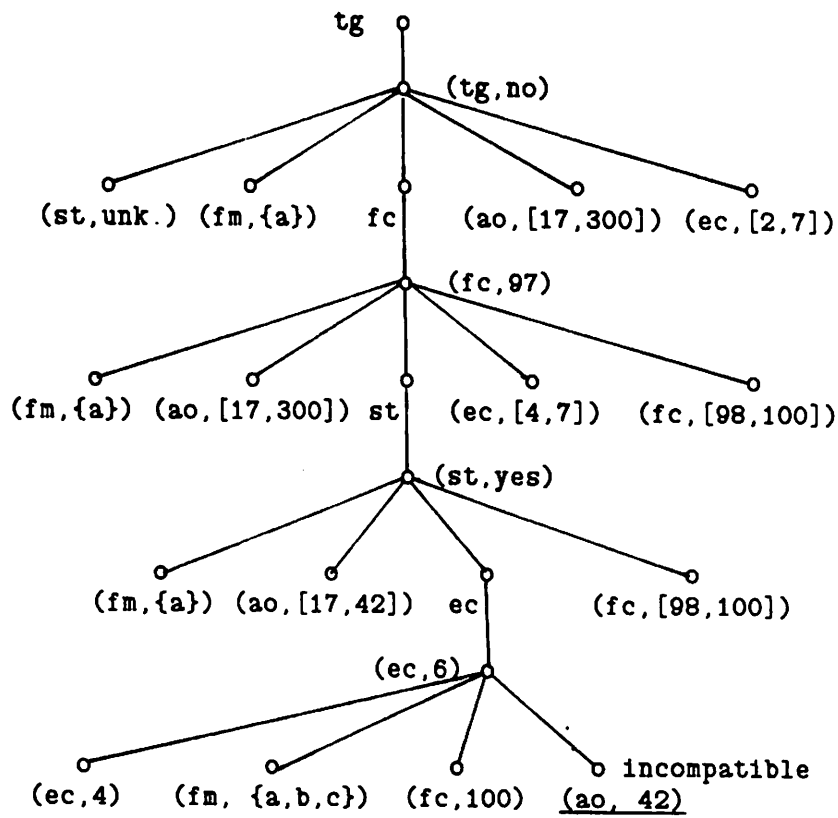


- If  $x_p$  is incompatible with  $r_p$ , mark  $(att_p, x_p)$  "incompatible."
  - If  $(att_p, x_p) = (att_j, x_j)$ , remove  $(att_p, x_p)$  from SL and DRT. Remove  $att_j$  from AT.
  - If  $x_p$  is completely specified, remove  $att_p$  from AT, and remove  $(att_p, x_p)$  from SL.
10. If there is any node marked "incompatible," return DRT and stop.
11. Choose from SL a node  $(att_j, x_j)$  such that  $att_j \in AT$ , and  $r_j \neq$  don't-care.
12. If there is no such node, return DRT and stop.
- Otherwise go to 4.

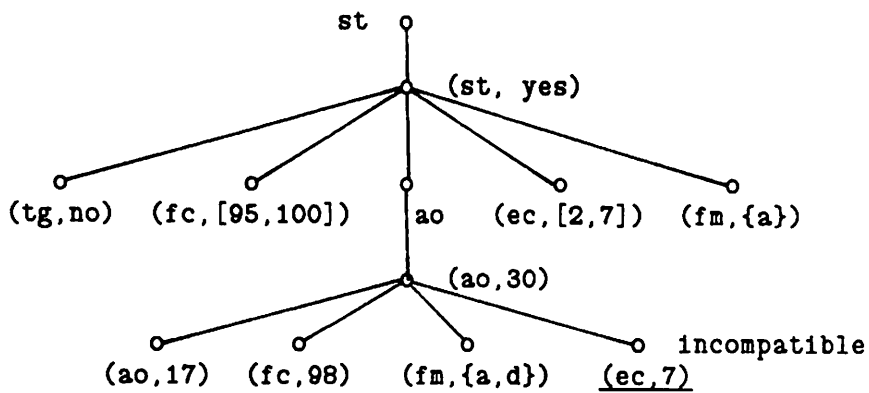
\*\*\*\*\*

**Example.** Consider the evaluation matrix in Table 3-6. Let the requirement vector be  $(st, tg, fm, fc, ao, ec) = (\text{yes}, \text{no}, \{a\}, 97, 30, 6)$ . Two different dynamic R-trees are shown in Figure 3-16. They both contain incompatible nodes, since this set of requirements is inconsistent.

The size of a dynamic R-tree is a function of the order in which



(a). A dynamic R-tree with 4 stages



(b). A dynamic R-tree with 2 stages

Figure 3-16: Two dynamic R-trees for a requirement vector

attributes are selected. Heuristics can be employed to generate dynamic R-trees having different attributes. For example, if an R-tree having a relatively small number of stages is desired, one can select the node  $(att_j, x_j)$  for expansion such that  $|\{v_{ij}\}|$  is maximum. If satisfying some set of weighted requirements is important, one can sort the attributes by their weights, and then expand the R-tree using the sorted attribute order. Thus the ramifications of the more important requirements will be processed first, and the less important requirements may be adjusted using the information from the dynamic R-tree. Next, we will discuss some applications of ramification trees.

#### 3.4.4. Ramification analysis

One problem in selection is that the consequences of the various requirements are not obvious, so that it is hard to avoid inconsistent requirements or to determine the best way for making tradeoffs. Ramification analysis is aimed at revealing relationships among different attribute values and helping a selector in specifying requirements and making tradeoffs.

##### 3.4.4.1. Consistency checking

**Definition 3.21:** A set of requirements is consistent if there exists a candidate which satisfies it, otherwise the requirements are inconsistent.

From the above definition, if  $r_k$  is inconsistent with  $r_j$ , then  $r_j$  is inconsistent with  $r_k$ . Given two requirements  $r_k$  and  $r_j$ , their consistency

can be easily checked by constructing the R-bush for either  $\text{att}_k$  or  $\text{att}_j$ . If the direct consequence of  $(\text{att}_k, r_k)$  on  $\text{att}_j$  is  $(\text{att}_j, x_j)$ , where  $x_j$  is incompatible with  $r_j$ , then  $r_k$  and  $r_j$  are inconsistent, and vice versa. The order in which attributes are considered is immaterial.

To test if the values in a requirement vector  $R = (r_1, r_2, \dots, r_N)$  are consistent, a dynamic R-tree based on  $R$  can be built. For a domain of  $N$  attributes, an R-tree has at most  $N$  stages. There may be up to  $N!$  different dynamic R-trees for the same requirement vector, each corresponding to a permutation of the  $N$  attributes. An important question is: in order to detect consistencies among a set of requirements, does it matter which R-tree is generated?

**Theorem 3.22:** Any dynamic R-tree  $\text{DRT}(R, C)$  contains incompatible node(s) or is empty if and only if the requirement vector  $R$  is inconsistent for the set of candidates  $C$ .

**Proof: If part:** We arbitrarily select a sequence of attributes,  $r_1', \dots, r_N'$ , to expand the R-tree, where  $r_1', \dots, r_N'$  is a permutation of  $r_1, \dots, r_N$ . Suppose  $R = (r_1, r_2, \dots, r_N)$  is inconsistent. There must exist an  $r_j'$  such that either

- (a)  $\{ c_i \mid c_i \text{ satisfies } r_j' \} = \emptyset$  or
- (b)  $\{ c_i \mid c_i \text{ satisfies } r_1' \ \& \ r_2' \ \& \ \dots \ \& \ r_{j-1}' \}, 1 < j \leq N \neq \emptyset$  and  
 $\{ c_i \mid c_i \text{ satisfies } r_1' \ \& \ \dots \ \& \ r_{j-1}' \ \& \ r_j' \}, 1 \leq j \leq N = \emptyset$ .

For case (a), if  $\text{att}_j'$  is the first attribute being considered, the R-tree is empty. Otherwise, following Procedure 3-5, in the  $k$ th ( $1 \leq k \leq j-1$ ) stage

of the DRT there should be a leaf node  $(att_j', x_j')$ , where  $x_j'$  is the direct consequence of  $r_1' \& \dots \& r_k'$  on  $att_j'$ , and  $x_j'$  is incompatible with  $r_j'$ . Then at Step 9, the node  $(att_j, x_j')$  will be marked "incompatible."

**Only if part:** Suppose a dynamic R-tree has an incompatible node  $(att_p, x_p)$  at the  $j$ th stage. By the definition of Procedure 3-5, the initial candidate set is  $T = C$ , the set of all candidates in the solution space. At the end of the  $k$ th stage, the candidate set becomes

$$T_k = \{ c_i \mid c_i \text{ satisfies } r_1' \& \dots \& r_k' \}.$$

Any possible solution must be in  $T_k$ . Let  $k = j-1$ .  $(att_p, x_p)$  is marked "incompatible" at the  $j$ th stage means that for all  $c_i$ , if  $c_i$  satisfies  $r_1'$  through  $r_j'$ , then the value  $v_{ip}$  for  $att_p$  is bounded by  $x_p$ , which is incompatible with  $r_p'$ . That is, no  $c_i$  can satisfy all requirements in  $R$ , so  $R$  is inconsistent.

**Definition 3.23:** Let  $Level[node]$  equal the length of the path from the *node* to the root in an R-tree DRT. Let  $SN$  be the set of nodes in the DRT. A node  $(att_j, x_j)$  in a DRT is an essential node if

$$Level[(att_j, x_j)] = \max\{ Level[(att_j, x)] \mid (att_j, x) \in SN \}.$$

In short,  $(att_p, x_p)$  is an essential node if it is the last appearance of  $att_p$  in DRT. All leaf nodes of the last stage are essential. Some internal nodes or leaf nodes at earlier stages may be essential.

**Lemma 3.24:** Let  $EN$  be the set of essential nodes of a dynamic R-tree  $DRT(R, C, A)$ , where  $A$  is the attribute set  $A = \{ att_1, \dots, att_N \}$ .

Then  $EN$  is unique for a DRT.  $|EN| = N$ , and there is exactly one essential node for each attribute.

**Proof:** From the definition and construction of dynamic R-trees, it is obvious that for every  $att_j \in A$ ,  $att_j$  must appear at least once, and thus there must be a lowest level at which it appears. Thus  $|EN| = N$  and  $EN$  is unique.

**Lemma 3.25:** Suppose  $R$  is a consistent requirement vector,  $DRT(R, C, A)$  a dynamic R-tree for  $R = (r_1, \dots, r_N)$ ,  $C = \{c_1, \dots, c_M\}$ ,  $A = \{att_1, \dots, att_N\}$ , and  $EN = \{ (att_j, x_j) \mid 1 \leq j \leq N \}$  is the set of essential nodes of DRT. For any  $att_j$ , if  $x_j$  is completely specified, then for every solution  $c_i$ ,  $v_{ij} = x_j$ . If  $x_j$  is not completely specified, then the following conditions hold.

For numeric attributes,  $x_j = [x_{j1}, x_{j2}] = \text{range}\{v_{ij} \mid c_i \text{ satisfies } R\}$ .

For logical attributes,  $x_j = \text{unknown}$ , and there exist solutions  $c_i$  and  $c_k$  such that  $v_{ij} = \text{yes}$  and  $v_{kj} = \text{no}$ .

For lower-bound complex attributes,  $x_j = \cap \{v_{ij} \mid c_i \text{ satisfies } R\}$ .

For upper-bound complex attributes,  $x_j = \cup \{v_{ij} \mid c_i \text{ satisfies } R\}$ .

**Proof:** From the construction of a dynamic R-tree, if an essential node  $(att_j, x_j)$  is an internal node or a leaf node of an intermediate stage,  $x_j$  must be uniquely specified, otherwise  $att_j$  must remain in  $A$ , and a node  $(att_j, x)$  will be generated at the next stage. Since  $R$  is consistent, a DRT is generated until no further expansion is possible. The rest of the essential nodes are all leaves of the last stage. Suppose the last stage is  $k$ , and the

last internal node is  $(att_q, r_q)$ . Before constructing the last stage of the DRT, the candidate set is  $T' = \{ c_i \mid c_i \text{ satisfies } r_1' \ \& \ \dots \ \& \ r_{k-1}' \}$ . Following Procedure 3-4, when constructing the last stage, the candidate set considered is  $T = \{ c_i \mid c_i \in T' \text{ and } c_i \text{ satisfies } r_q \}$ . If a node  $(att_p, x_p)$  is a leaf node of the last stage, then  $x_p$  is the direct consequence of  $r_q$  on  $att_p$ . By the definition of direct consequences, the lemma follows.

**Definition 3.26:** Let EN1 and EN2 be the set of essential nodes for DRT1 and DRT2, respectively. DRT1 and DRT2 are said to be equivalent if  $EN1 = EN2$ .

**Theorem 3.27:** If  $R = (r_1, r_2, \dots, r_N)$  is a consistent requirement vector, all dynamic R-trees  $DRT(R,C,A)$  are equivalent.

**Proof:** Given  $R$ ,  $C$ , and  $A$ , the set of candidates  $\{c_i \mid \forall j, r_j \Leftarrow v_{ij}\}$  is unique, and is called the solution set. We have shown that for any consistent requirement vector  $R$ , a solution set can be generated during the process of building a dynamic R-tree  $DRT(R,C,A)$ . Obviously, the solution sets associated with any  $DRT(R,C,A)$  should be the same.

Suppose two dynamic R-trees  $DRT1(R,C,A)$  and  $DRT2(R,C,A)$  are not equivalent, i.e., there exist two essential nodes  $n1 = (att_j, x1_j) \in EN1$  and  $n2 = (att_j, x2_j) \in EN2$ , such that  $x1_j \neq x2_j$ . Assume  $att_j$  is a numeric attribute, where  $x1_j = [x1_{j1}, x1_{j2}]$ , and  $x2_j = [x2_{j1}, x2_{j2}]$ . Without loss of generality, assume  $x2_{j2}$  is not in the range of  $x1_j$ . This means that there exists a solution  $c_i$  with  $v_{ij} = x2_{j2}$ , and  $c_i$  is not a solution generated from

DRT1(R, C, A). This is a contradiction. Similar arguments can be made for logical and complex attributes.

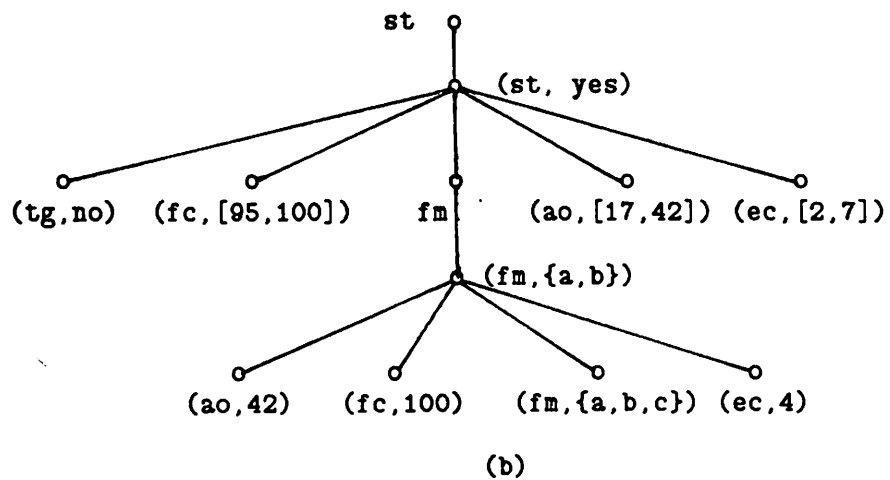
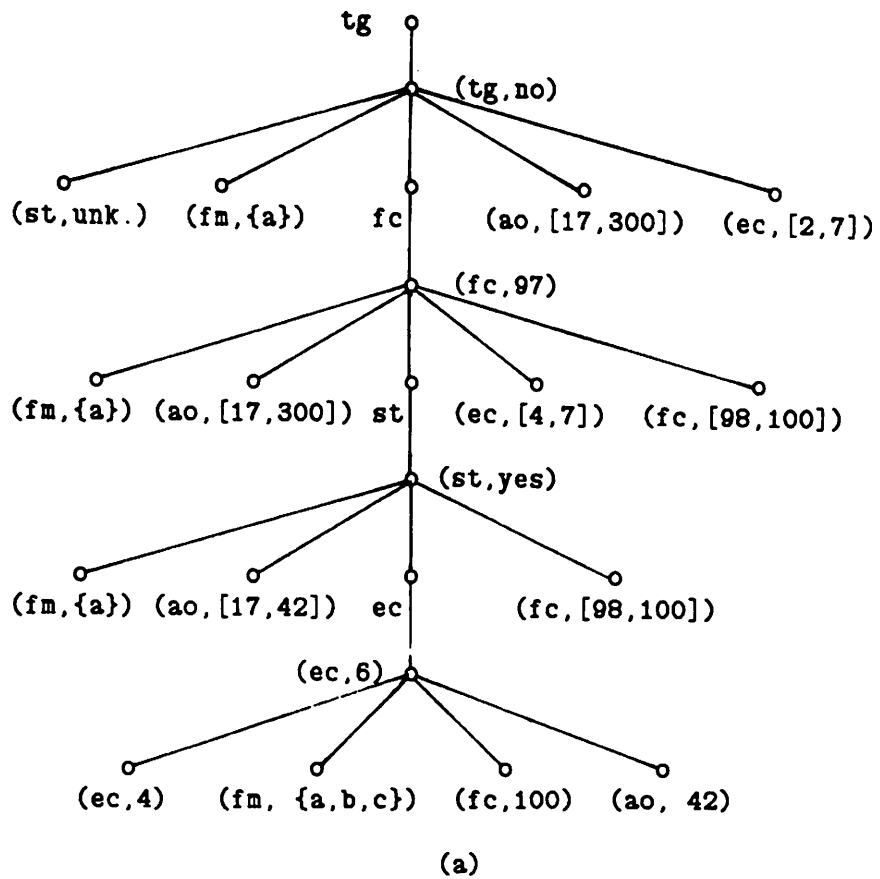
**Example.** Again consider the evaluation matrix in Table 3-6. Let the requirement vector be  $R = (\text{yes}, \text{no}, \{a, b\}, 97, 45, 6)$ . Two different dynamic R-trees are shown in Figure 3-17. Although they are different, they are equivalent because they have the same set of essential nodes, i.e.,  $EN = \{(st, \text{yes}), (tg, \text{no}), (fm, \{a, b\}), (fc, 100), (ao, 42), (ec, 4)\}$ .

From the above discussion we conclude that although there are many dynamic R-trees for the same requirement vector, candidate set and attribute set, it is sufficient to generate only one of the R-trees in order to detect inconsistency among requirements. If the R-tree has "incompatible" nodes, then these requirements are inconsistent, otherwise they are consistent. The ramifications of a requirement vector  $R = (r_1, r_2, \dots, r_N)$  can be specified by the set of essential nodes of a dynamic R-tree built using Procedure 3-5. When requirements are inconsistent, tradeoffs have to be made in order to obtain a solution.

#### **3.4.4.2. Consequences of changing requirements**

Ramification trees can be used to predict the consequences of changing requirements.





**Figure 3-17:** Two non-conflict dynamic R-trees for a requirement vector

### Single change

During a selection process, it is useful to determine what will happen if a requirement  $r_j$  is changed from  $r_j(1)$  to  $r_j(2)$ . A requirement  $r_j(2)$  for  $\text{att}_j$  is stronger than  $r_j(1)$  if  $r_j(1) \Leftarrow r_j(2)$ ;  $r_j(2)$  is weaker than  $r_j(1)$  if  $r_j(2) \Leftarrow r_j(1)$ ; where  $r_j(1) \neq r_j(2)$ . It is obvious that if  $r_j(2)$  is weaker than  $r_j(1)$ , the change does not cause any new inconsistencies. As a result, it may make more candidates qualified and hence some requirements which were hard to satisfy now may become easier to satisfy. If  $r_j(2)$  is stronger than  $r_j(1)$ , it does not necessarily result in any inconsistency, but may shift the search region to another part of the solution space.

Let  $R1 = (r_1, \dots, r_j(1), \dots)$  and  $R2 = (r_1, \dots, r_j(2), \dots)$ . Suppose we have not generated a DRT for  $R1$ . A DRT for  $R2$  can be generated, in which the root is  $\text{att}_j$  and the first internal node is  $(\text{att}_j, r_j(2))$ . In this way, the ramifications of the new requirement can be quickly identified. Suppose there is already a dynamic R-tree DRT based on  $R1$ . A DRT for  $R2$  can be generated by modifying the DRT for  $R1$ . Consider three cases.

Case 1.  $(\text{att}_j, r_j(1))$  is an essential leaf node.

- If  $r_j(2)$  is stronger than  $r_j(1)$ , the change causes an inconsistency.

The leaf  $(\text{att}_j, r_j(1))$  should be marked "incompatible." In this case, the achievable value for  $\text{att}_j$  is restricted to be  $r_j(1)$  due to the other requirements (which are internal nodes of the DRT). Since these internal node requirements are not changed,

tightening  $r_j(1)$  to  $r_j(2)$  leads to a conflict between  $r_j$  and other requirements.

- If  $r_j(2)$  is weaker than  $r_j(1)$ , the DRT is unchanged. In this case changing  $r_j(1)$  to  $r_j(2)$  does not cause any change to the solution set.

Case 2.  $(att_j, r_j(1))$  is an internal node of the DRT. All ancestor nodes of  $(att_j, r_j(1))$  can be kept, because they are not affected by the change of  $r_j$ .

- If at this point the possible values of  $att_j$  are  $V_j = \{v_{ij} \mid c_i \in T\}$ , and  $\{v_{ij} \mid r_j(1) \Leftarrow v_{ij}, v_{ij} \in V_j\} = \{v_{ij} \mid r_j(2) \Leftarrow v_{ij}, v_{ij} \in V_j\}$ , the DRT is unchanged. For example, if the requirement for area overhead is changed from  $(ao, 30)$  to  $(ao, 40)$ , the dynamic R-tree shown in Figure 3-16 (b) is unchanged.
- Otherwise, the subtree rooted by  $(att_j, r_j(1))$  should be replaced by a dynamic R-tree for the candidate set  $T$  and the attribute set  $A$  for that stage and the requirements in  $R2$  not yet processed, starting from  $r_j(2)$ .

Case 3.  $(att_j, r_j(1))$  is not a node of DRT. Since there exists an essential node for every attribute, there must be an essential leaf node  $(att_j, x_j)$ . If  $r_j(2) \Leftarrow x_j$ , then the node is a compatible node, otherwise it should be labeled incompatible.

## Multiple changes

Multiple changes in the requirements can be viewed as a sequence of single changes. The ramifications of multiple changes can be determined by repeatedly applying the analysis for single changes. In some cases, however, there are more straightforward ways to find the consequences of multiple changes. For simplicity, we only discuss double changes.

Let  $R1 = (r1_1, \dots, r1_N)$  and  $R2 = (r2_1, \dots, r2_N)$ . A double change occurs if there exist exactly two attributes  $att_j$  and  $att_k$  such that  $r1_j \neq r2_j$  and  $r1_k \neq r2_k$ . Obviously, if  $r2_j$  and  $r2_k$  are weaker than  $r1_j$  and  $r1_k$ , respectively, the changes do not cause any new problem, and may resolve some existing inconsistencies. If  $r2_j$  is inconsistent with  $r2_k$ , the changes cause an inconsistency. When  $r2_j$  is consistent with  $r2_k$ , several possibilities exist.

**Lemma 3.28:** Assume there is a dynamic R-tree  $DRT(R1, C, A)$  in which  $(att_k, x_k)$  is the essential leaf for  $att_k$  and  $(att_j, r1_j)$  is not an ancestor node of  $(att_k, x_k)$ .

- A double change causes an inconsistency if  $r2_k$  is incompatible with  $x_k$ .
- If  $(att_k, x_k)$  is an incompatible node, the double change makes it compatible if  $r2_k \Leftarrow x_k$ .
- Otherwise,  $(att_k, x_k)$  is unchanged.

**Lemma 3.29:** If  $(att_k, r1_k)$  is a leaf node of a DRT and  $(att_j, r1_j)$  is

not an ancestor node of  $(att_k, r1_k)$ , a double change causes an inconsistency if  $r2_k$  is stronger than  $r1_k$ .

Double changes can be handled differently in the following three cases.

1. If  $(att_j, r1_j)$  is not an ancestor node of  $(att_k, r1_k)$ , and vice versa, check for inconsistency using the above lemmas. If an inconsistency is found, mark the incompatible nodes and stop.

2. If the essential node for  $att_k$  ( $att_k, x_k$ ) is a leaf node, and  $(att_j, r1_j)$  is an ancestor node of  $(att_k, x_k)$ , check if the subtree rooted at  $att_j$  needs to be changed due to the change from  $r1_j$  to  $r2_j$  (see the conditions for case 2 under single changes).

- If so, replace the subtree rooted by  $att_j$  with  $DRT(att_j, r2_j)$ , which is based on R2 and the candidate and attribute set at this point.
- If not, the double change can be treated as a single change for  $att_k$ . The rules used for case 1 for single changes can be applied to determine the ramifications of  $r2_k$ .

3. If  $(att_k, r1_k)$  and  $(att_j, r1_j)$  are both internal nodes, assume  $(att_j, r1_j)$  is an ancestor node of  $(att_k, r1_k)$ .

- If the change from  $r1_j$  to  $r2_j$  does not result in any change to the stage defined by  $att_j$ , the change from  $r1_k$  to  $r2_k$  can be treated as a single change.

- Otherwise, the subtree rooted by  $att_j$  should be replaced by the dynamic R-tree  $DRT(att_j, r2_j)$  based on  $R2$ , and the candidate and attribute sets at that point.

During a selection process, the requirements may be changed dynamically. Such changes are necessary when no qualified candidates exist. Due to the size of the solution space and the dependences among attribute values, sometimes it is hard for a selector to know the result of changing requirements without searching through the solution space. Ramification analysis provides a fast way for predicting the consequences of requirement changes.

#### **3.4.4.3. Incremental requirement specification**

The first step in a selection process is to specify a set of requirements. There are two ways to do this. First, consider  $r_1, r_2, \dots, r_N$  as independent requirements and specify each of them individually. A requirement vector  $R = (r_1, r_2, \dots, r_N)$  formed in this way is likely to be inconsistent because of the dependencies among attribute values. The second method is to take the dependencies into account in specifying requirements incrementally, such that inconsistency among requirements can be avoided or detected as early as possible. Ramification analysis can be used for detecting such inconsistencies.

\*\*\*\*\*

**Procedure 3-6. Interactive incremental requirement specification**

Inputs:           A set of attributes  $A = \{att_1, att_2, \dots, att_N\}$ .  
                   An array of weighting factors  $(w_1, w_2, \dots, w_N)$ .  
                   A set of candidates  $C = \{c_1, c_2, \dots\}$ .  
                   A specific receiver.

Outputs:          A requirement vector  $R = (r_1, r_2, \dots, r_N)$ .  
                   A set of candidates which satisfy all requirements, if any.

1. Initialize R such that for all j,  $r_j = \text{don't-care}$ .
2. Let the dynamic R-tree DRT be an empty tree.
3. If A is empty, output R and T. Stop.
4. Choose an  $att_j$  such that  $w_j = \max\{w_k \mid att_k \in A\}$ .
5. Remove  $att_j$  from A.
6. Ask the selector for the requirement value  $r_j$  for  $att_j$ . Provide explanations and help if needed. .
7. If  $r_j = \text{don't-care}$ , go to 3.
8. Expand the DRT using the dynamic R-bush  $DRB(att_j, r_j)$  for the current T and A.
9. Let LN be the set of leaves of the R-bush.
10. Report the ramifications of  $att_j$  to the selector.

11. If  $r_j$  causes inconsistencies or some of the ramifications are not acceptable, ask if the selector would like to change  $r_j$ .

If so, remove the  $DRB(att_j, r_j)$  from the DRT. Go to 6.

If not, ask about the rest of the requirements. Set  $A = T = \emptyset$ . Go to 3.

12. Update T to the candidate set for the  $DRB(att_j, r_j)$ .

13. For each node  $(att_p, x_p)$  in LN do

if  $x_p$  is uniquely specified, remove  $att_p$  from A.

14. Go to 3.

\*\*\*\*\*

When a selector does not know how to specify a requirement for an attribute  $att_j$ , helpful information can be given, which, for example, explains what is the definition of the attribute, what are its possible values, and based on the current requirements what feasible consistent requirement values exist for  $att_j$ . If no implication on  $r_j$  can be drawn from the requirements given so far, and the selector is not sure what the requirement for  $att_j$  should be, one can skip to the next attribute and deal with the requirement for  $att_j$  later. It is important to point out that specifying a set of meaningful requirements requires extensive knowledge about the application domain. For example, specifying requirements for selecting a



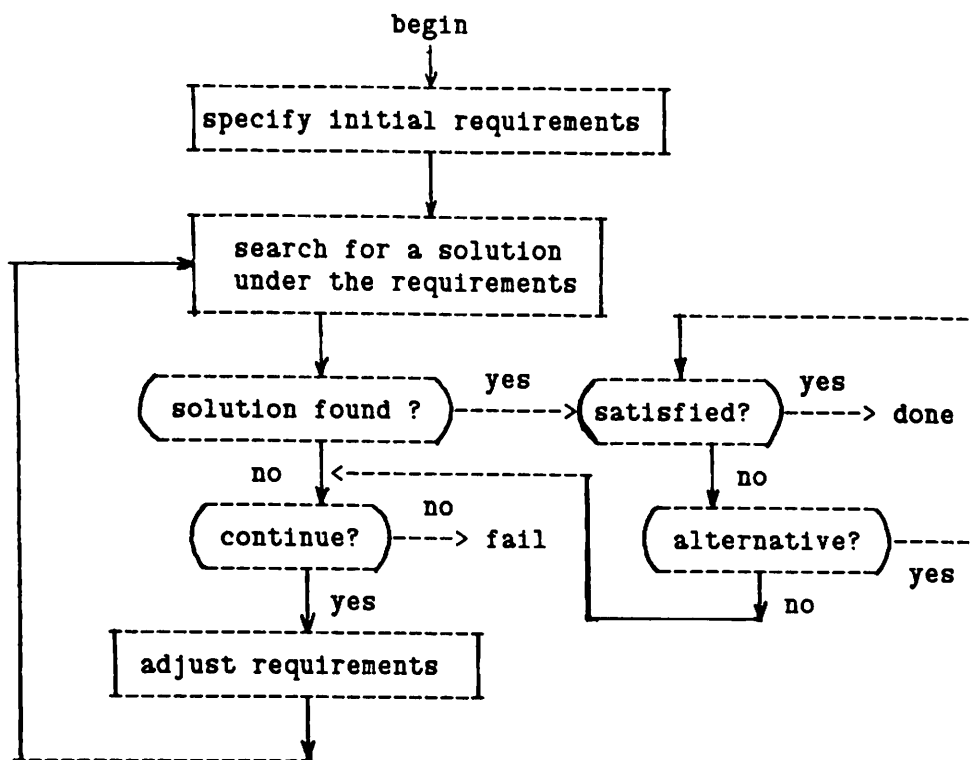
TDM involves knowledge such as VLSI chip design and manufacturing constraints. An expert system may be needed to help in making decisions concerning appropriate requirements from high level specification. This is beyond the scope of this research.

In short, incremental requirement specification starts from the most important attribute, and tries to infer as many other requirements as possible. In this way, some requirements can be automatically specified. As a side product, inconsistencies are detected while the requirements are specified, and a set of totally qualified candidates are found at the end of requirement specification, if any exist.

An alternative to ramification analysis using R-trees and the procedures presented is the use of rules to represent the relationships among attribute values. The rules are formed from expert knowledge or common sense. This method was not adopted for two reasons. First, we want to develop a general mechanism which is independent of an individual expert but emulates an expert's way of reasoning, such that the result will not be affected by the incompleteness of rules or the expert's mistakes. Secondly, ramifications vary with the candidate set. Even "expert knowledge" may not be true for a specific selection problem due to a limited solution space. Since the candidate set may often change, the rules based on specific domain knowledge may need to change too. This again hinders the use of rules.

### 3.5. The dynamic selection process

A selection process is mainly controlled by the selector and constrained by the requirements. There is no deterministic algorithm which guarantees finding the most satisfactory solution when conflicts exist. Selection is usually not a "problem-in-result-out" process, but rather a dynamic process which involves exploring the solution space, changing goals and constraints (requirements), and backtracking.



**Figure 3-18:** A typical selection process

A typical selection process is shown in Figure 3-18. Usually the selector first specifies an initial requirement vector  $R_0 = (r_{01}, \dots, r_{0N})$ . Search in the solution space is carried out using these requirements. If a satisfactory solution is found, the selection terminates with success.

Otherwise, alternative solutions (if any exists) are examined by the selector. If there is no acceptable solution and the selector wants to continue, one or more of the requirement values must be changed, and the search restarts using the new requirement vector  $R_1 = (r_{11}, \dots, r_{1N})$ . This process is repeated until a satisfactory solution is found or the selector decides to quit with no solution.

### 3.5.1. Searching under fixed requirements

A basic operation in a selection process is searching for a qualified candidate under a given requirement vector. In general, there are two classes of search procedures. One tends to examine all possibilities until a solution is found. It does not require backtracking, but requires exploring the entire solution space. Another class of search procedures tries to explore a small portion of the solution space first. If a solution is found, then the search stops. If no solution is found, it backtracks to try other unexplored portions of the solution space. However, if all solutions are to be found, exhaustive search is necessary. Since the searching space for determining if a given requirement vector is satisfied by some candidates is not very large, and initially finding all possible solutions is often desirable, the exhaustive search strategy is useful for searching under fixed requirements.

There are two ways to search through the solution space, namely, horizontal search and vertical search. The solution space for a given receiver is represented by an evaluation matrix  $EM[M, N]$ , where the  $M$  rows

correspond to  $M$  candidates and the  $N$  columns correspond to the  $N$  attributes of the domain. In a horizontal search, each candidate is checked to see if it satisfies all requirements. If a candidate fails to satisfy any of the requirements, it is disqualified as a solution. In a vertical search, each individual requirement is checked to see if any candidates satisfy it. If they do, those candidates are partial solutions for that requirement. A solution must be in the intersection of the partial solutions for all requirements.

\*\*\*\*\*

### Procedure 3-7. Horizontal search under fixed requirements

Inputs:            A set of candidates  $C = \{c_1, \dots, c_M \mid c_i = (v_{i1}, \dots, v_{iN})\}$ .  
                     A requirement vector  $R = (r_1, r_2, \dots, r_N)$ .

Output:            A solution set containing all qualified candidates.

Let  $T = C$ .

For  $i = 1$  to  $M$  do

  For  $j = 1$  to  $N$  do

    if  $r_j \neq v_{ij}$ , go to next  $j$ ;

    otherwise remove  $c_i$  from  $T$  and go to next  $i$ .

Output  $T$ .

\*\*\*\*\*

### Procedure 3-8. Vertical search under fixed requirements

Inputs:            A set of candidates  $C = \{c_1, \dots, c_M \mid c_i = (v_{i1}, \dots, v_{iN})\}$ .  
                     A requirement vector  $R = (r_1, r_2, \dots, r_N)$ .

Output:            A solution set containing all qualified candidates.

Let  $T = C$ .

For  $j = 1$  to  $N$  do

  For each  $c_i \in T$  do

    if  $r_j \neq v_{ij}$ , go to next  $c_i$ ;

    otherwise remove  $c_i$  from  $T$  and go to next  $c_i$ .

Output  $T$ .

\*\*\*\*\*

It is interesting to note that vertical search is automatically conducted in building a dynamic ramification tree. The difference is that the requirements are not examined in a fixed order.

The number of comparisons in both search procedures is bounded by  $N \times M$ . Usually the number of attributes in a domain is constant during a search process. Thus the complexity of both search procedures is linear with the number of candidates under consideration.

Search under fixed requirements may often end up with an empty solution set. An important feature of the selection processes is that requirements may be changed. The next sections will discuss how to search for a solution under these flexible requirement conditions.

### 3.5.2. Reason analysis directed backtracking

Selection is a process which often requires backtracking. In a selection process, backtracking means changing one or more requirement and restarting the search. In general, there are two classes of backtracking procedures, namely, chronological backtracking and dependency-directed backtracking [61, 15]. In chronological backtracking, the entire solution space is systematically searched, either explicitly or implicitly. Whenever a failure is encountered, the process simply returns to the most recent decision point, and examines the next alternative. An example is depth first search. This is not a suitable approach for solving most selection problems, since a requirement vector has many components each of which may take on numerous values. Thus the search space is quite large. Chronological backtracking leads to a blind search. In dependency-directed backtracking, dependency records are maintained which link conclusions with the assumptions behind them. Each time the search fails, the process traces back through the dependency relations from the contradiction to the assumptions supporting it, and then takes actions to negate the contradiction. This technique has been used in inference programs, in debugging databases and in planning robot actions, where dependencies among assumptions and conclusions are tight and complicated. In general, finding the best way to efficiently record and use dependencies and resolve ambiguities when several revisions exist is still an open problem.

In a selection process, failures are usually caused by one or more components of the requirement vector. Backtracking always involves modifying requirements. The dependencies between search failures and requirements are relatively clear but not unique, i.e. there may be numerous ways to modify the requirement vector in order to find a solution. Knowing the dependencies is not enough. Several kinds of failures (as discussed in Section 3.3.3) may occur at the same time. The important thing is to decide what changes should be made in the requirement vector. To solve this problem, a strategy called reason directed backtracking is proposed here. It differs from dependency-directed backtracking in that it determines how to backtrack based upon the nature of the failure, rather than only knowing which requirements cause the failure.

The basic idea of reason analysis directed backtracking is to classify failures in such a way that for each failure type, a remedy can be suggested which specifies the reason for the failure and possible ways to resolve the failure.

The classification of failures is a kind of meta knowledge, i.e., the knowledge about what we know. This meta knowledge deals with the following concepts concerning possible failures in selection.

- What are the possible failures encountered in searching for a solution with a problem solving procedure. Let  $F = \{f_1, f_2, \dots, f_k\}$  be such a set of failures, where  $k$  may be quite large but

finite. For example, a frequently occurring failure in selection is that the value of a candidate's attribute does not satisfy a requirement.

- How to abstract general features from this set of failures and classify failures into a few categories  $FC_1, FC_2, \dots, FC_h$ , such that

- (a)  $h \ll k$ ,
- (b)  $\forall i \exists j$  such that  $f_i \in FC_j$ , and
- (c) no  $FC_i$  is a subset of  $FC_j$ , for all  $i \neq j$ .

The failure categories may not be disjoint, i.e., a  $f_i$  may belong to more than one  $FC_j$ .

- How to identify each category of failures. There are some unique syndromes for each category of failures. A failure belongs to a category if it has the corresponding syndrome, i.e., there is a predicate  $P_i$  for each  $FC_i$ , such that  $FC_i = \{f_j \mid P_i(f_j)\}$ .
- How to recover from each category of failures. Different actions should be taken depending upon the category of failures.
- What are the priorities among failure categories, i.e., which  $FC_i$ 's are more critical and should be handled before others.

In a selection process, failures indicate unsatisfiable requirements, and remedies are suggestions for good ways to change these requirements. When a search fails, the following steps can be taken.



1. Identify the category of failures.
2. Determine the most critical category among the current failures.
3. Specify the reason for the most critical failure.
4. Suggest a good way to adjust the requirements such that a suitable solution with minimal penalty may be obtained.
5. Change the requirements as suggested and restart the search process.

A detailed example of reason directed backtracking will be presented in the next chapter.

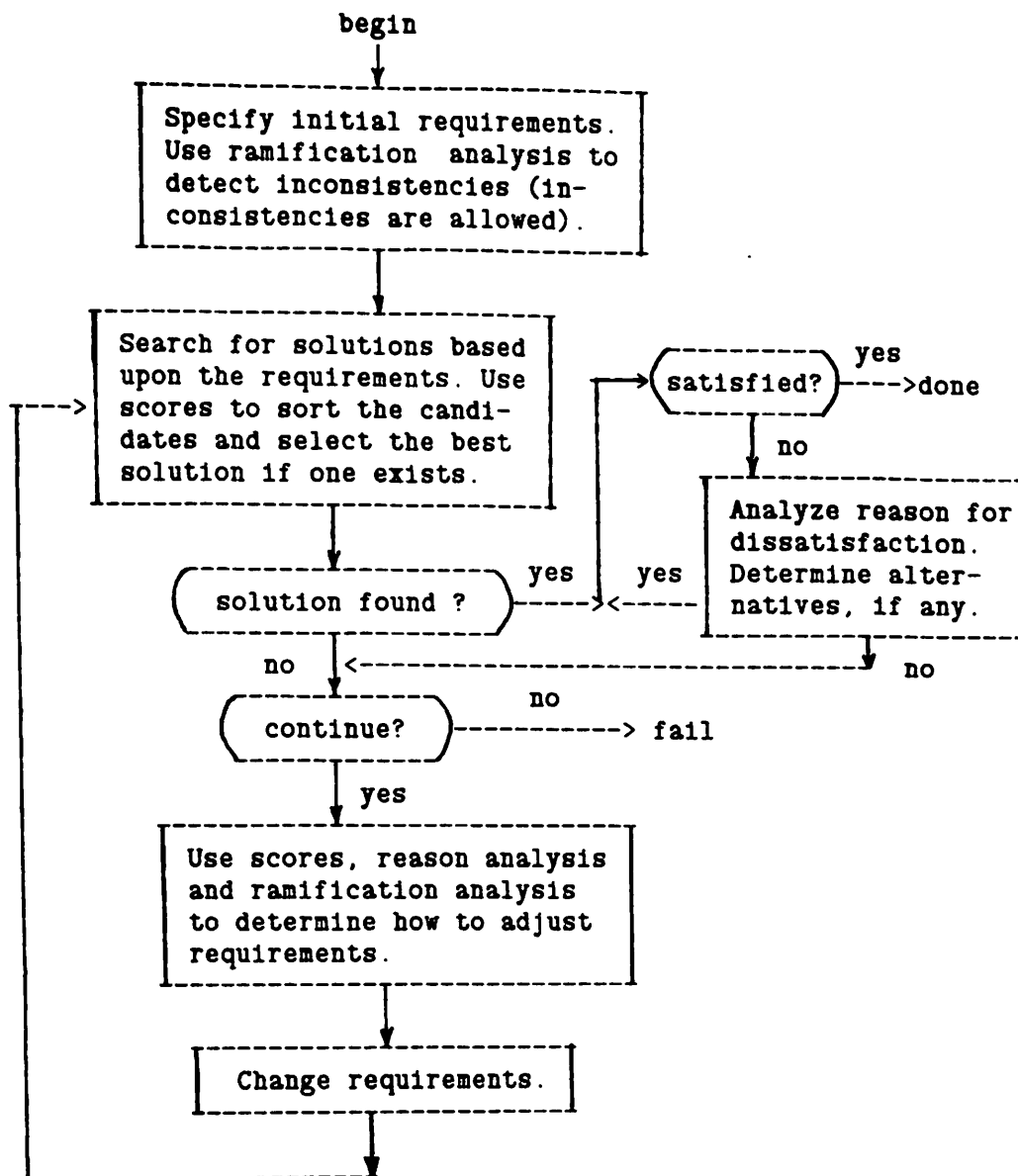
### **3.5.3. Knowledge guided search and selection**

Searching within a solution space deals with moving the feasible region defined by the requirement vector until both enter the satisfaction region of some candidate(s). Since there are numerous directions for movement within a solution space, a blind search may take a long time before a solution is found. A smart search will take a few steps to reach a solution. On the other hand, a selector may want to look at and compare various possible alternative solutions before making a final selection. If one set of requirements does not lead to a satisfactory solution, the requirements may be changed. The key to a successful selection is the ability to explore the solution space and make judicious changes in requirements such that a solution can be found in an efficient manner.

The basis for a smart search is using knowledge to guide the search. Knowledge about the candidates in a selection domain indicates where the possible solutions lie. Knowledge about the evaluation of candidates can be used to determine how far the current requirement vector is from the candidates, which candidate is closest to the current requirement vector, and hence give a clue on how to best change the requirements. Knowledge about ramifications of requirements indicates which changes result in inconsistencies. The combination of ramification analysis and scoring can help in making tradeoffs. Knowledge about reason analysis indicates how best to process through the search space. In short, various levels of knowledge can be used to direct an efficient search. A knowledge directed selection process is shown in Figure 3-19.

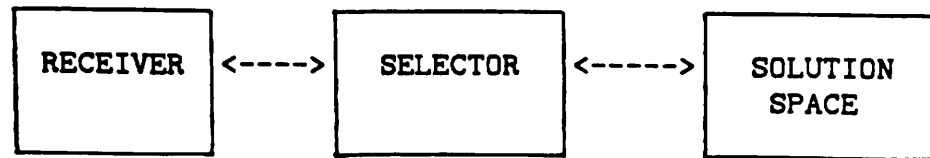
### 3.6. Knowledge based expert consultant systems

From the above analysis of the selection problem, it can be seen that a selection process is one which requires extensive domain knowledge and reasoning. Making a selection is a three-party game, as illustrated in Figure 3-20. The selector gives requirements based on the receiver and the application, and then looks for a solution in the solution space. Selection is difficult because the selector does not have a thorough knowledge of the entire domain and cannot efficiently handle the large amount of information in the solution space. An intelligent knowledge based system is ideal in helping to make the selection. We suppose to change the selection process

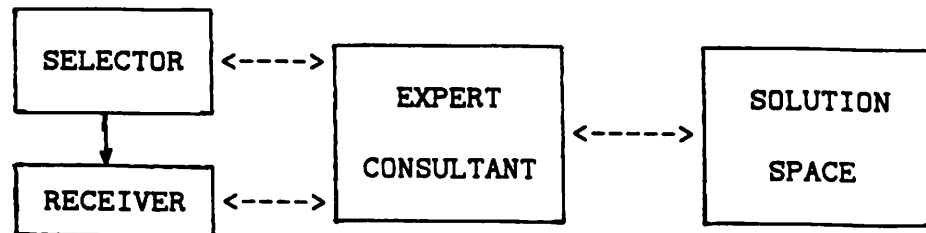


**Figure 3-19:** A knowledge guided selection process

into a four-party game as shown in Figure 3-21. The expert consultant is a computer program which understands the domain knowledge and knows how to reason about this knowledge. By introducing the consultant, the function of the selector is greatly reduced, and a better result can be expected if the knowledge based system is suitably constructed.



**Figure 3-20:** Selection by the selector only



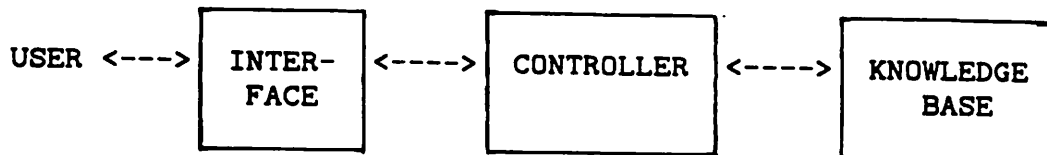
**Figure 3-21:** Selection with help of an expert consultant

### 3.6.1. General characteristics of the expert consultant system

The expert consultant system should have the following features:

- \* maintain a large amount of domain knowledge.
- \* understand the domain knowledge.
- \* be easy to expand and change the knowledge base.
- \* interact with a selector in a friendly manner.
- \* be able to reason about the domain knowledge based on the selector's requirements.
- \* be able to make a good selection and suggest alternative solutions or new requirements.
- \* be able to give advice.
- \* provide explanation for reasoning and information about the selection domain.

The basic architecture of an expert consultant system is like other expert systems; it consists of a knowledge base, a controller and a user interface, as shown in Figure 3-22. The knowledge base is separated from the controller for the sake of changeability.



**Figure 3-22:** Basic configuration of an expert consultant system

### 3.6.2. The selector-consultant relationship

To simulate a human expert consultant, the role and right of the user and the expert system should be as follows.

#### Selector

- Specify requirements
- Provide information when asked
- Decide whether or not to change any requirements
- Decide whether or not to accept a solution or a piece of advice
- Ask the expert consultant to make certain decisions
- Ask for explanations.

#### Consultant

- Ask questions when information is needed
- Search for solutions based upon requirements
- Report a solution when found and ask for confirmation
- Report failures and the reason for the failures

Give advice and suggestions

Provide explanations

Make a decision when asked.

In short, the selector is a master who has full right of making decisions, but may not be an expert in the domain of selection. The consultant is a knowledgeable assistant which knows more about the domain knowledge than does the selector, and is responsible for leading the selector to some well-specified goals, but cannot make decisions without the selector's permission. The interface between the selector and the consultant should reflect this relationship.

### **3.6.3. The knowledge base**

The knowledge base is an important part of an expert consultant system. To insure changeability and expandability, the knowledge base is separated from the controller, and is relatively static. The control program gets information from the knowledge base and reasons with this knowledge. The knowledge base should be designed in such a way that

- it contains all the information needed by both the expert and the designer;
- it is easily accessible, that is, knowledge should be well organized so as to be efficiently retrieved;
- it is understandable, that is, knowledge should be represented in

a form so that it can be easily used by the program without complicated translation;

- it is changeable, that is, any knowledge can be easily added, deleted or updated without causing major changes to other parts of the system.

### **3.6.3.1. Information content of the knowledge base**

Basically, the consultant system has two kinds of knowledge.

#### **1. Domain specific knowledge which includes**

- a. knowledge about the attributes of the selection domain and the properties of the attributes, such as upper and lower bounds.
- b. attribute values of all candidates in the domain of selection.

#### **2. Domain independent knowledge such as knowledge about the art of controlling a selection process.**

The knowledge base contains all domain specific knowledge. Domain independent knowledge is incorporated in the controller.

### 3.6.3.2. Knowledge representation

Knowledge representation is the process of creating formal data objects that are understandable by the program and adequately represent interesting aspects of the problem. It plays a crucial role in making a knowledge based system meaningful and efficient. There are a number of knowledge representation techniques [6] used in various AI systems, such as

predicate logic representation

procedure representation

semantic networks

production systems

analogical representation

graphical representation

frames and scripts.

For a selection problem, the knowledge to be represented is of a diverse nature. Therefore different representations may be needed.

For most selection problems, it appears that candidates are most suitably represented by frames. A frame provides a number of *slot* which is the place for knowledge to be filled in. The slots correspond to the generic concepts which define a domain. Every candidate in the domain can be characterized by the same set of concepts. The multi-dimensional domain of selection is naturally represented by a frame, in which each attribute corresponds to a slot in the frame, and its value is the entry in the slot. The general structure of a candidate frame is as follows.



**CANDIDATE Frame**

Specification-of:	candidate
Measures:	
att <sub>1</sub> :	integer
att <sub>2</sub> :	sets
...	...
att <sub>N</sub> :	real
Other-attribute <sub>1</sub> :	procedure
...	...
Other-attribute <sub>n</sub> :	text

The candidate frame defines a general representation of the candidates. It may not be unique because slots can be organized into different hierarchies. It may not be fixed because new slots can be added to the frame. However, once a frame is defined, all candidates should be represented in a uniform way. Each particular candidate  $c_i$  defines a specific *instance* of the general candidate frame. The number of instances is not limited. When a new candidate is added to the domain, a new instance is created. When a candidate is removed from the domain, its corresponding instance is eliminated too. A basic TDM frame is given in Figure 3-23.

Knowledge about searching methods (such as vertical search) and ramification analysis is best represented by procedures, because they are relatively fixed and procedures are good at handling a large amount of random data efficiently.

Knowledge about reasoning analysis and selection making can be

**The TDM Frame**

Specification-of:	a-TDM
Name:	text
Kernel-type:	set
Measures:	
Fault model	set
Fault coverage	real
Fault masking	binary
Concurrent testing	binary
Error latency	real
Self testing	binary
Function dependence	binary
Logic overhead	integer
Extra I/O connections	integer
Area overhead	real
Extra delay	integer
Delay per test	integer
# of test patterns	integer
Test storage	integer
Test application time	real
Means of test generation	set
Means of test application	set
Means of response evaluation	set
Logic design cost	integer
Layout cost	integer
Hardware-modification:	procedure
Test-plan	procedure
Test-set	procedure
Connections	graph
Explanation:	text

**Figure 3-23:** A basic TDM frame

represented by either rules or procedures. The advantage of using rules is that it is easier to change selection strategies via rules than to change a program.

### 3.6.4. Control mechanism

The controller of an expert consultant system is responsible for controlling the selection process. It interacts with a selector through a friendly interface to get information and requirements, and to give results and advice. It extracts information from the knowledge base by calling search routines. When there are no candidates to match the requirements, it initiates reason analysis to determine the reason for failure and suggests ways for overcoming the failure. The basic function of the controller will be discussed in Chapter 4 along with an example implementation.

### 3.7. Related research

In this chapter we have formulated the selection problem and presented a general framework of a knowledge based system for solving a selection problem. A selection problem belongs to a broader class of problems, referred to as multiple criteria decision making (MCDM) [16, 36, 37, 38, 45, 74, 84, 85, 86]. More specifically, it is a problem of multiple attribute decision making (MADM) [37], which deals with selecting / evaluating an alternative (candidate) from a finite, explicit list of alternatives. Another category of MCDM problems, called multiple objective decision making (MODM) [36], deals with designing the best alternative among an infinite set of alternatives implicitly defined by multiple constraints.

MADM methods have been studied by many researchers in various

disciplines such as operational research, management science, economics, marketing research, applied mathematics, decision theory, and psychology. In MADM, a finite set of alternatives is explicitly described by a set of attribute values in a decision matrix

$$D = \begin{array}{cccc} & \text{att}_1 & \text{att}_2 & \dots & \text{att}_n & \text{alternative} \\ \left. \begin{array}{l} x_{11} \\ x_{21} \\ \dots \\ x_{m1} \end{array} \right\} & \begin{array}{l} x_{12} \\ x_{22} \\ \dots \\ x_{m2} \end{array} & \dots & \dots & \begin{array}{l} x_{1n} \\ x_{2n} \\ \dots \\ x_{mn} \end{array} & \begin{array}{l} A_1 \\ A_2 \\ \dots \\ A_m \end{array} \end{array}$$

where the  $A_i$ 's are the alternatives and  $x_{ij}$  is the value of  $A_i$  with respect to  $\text{att}_j$ . However the objective for MADM is often implicit or ill defined. As far as the purpose is concerned, MADM problems can be classified into two categories:

1. **optimization**: trying to find the best alternative, and
2. **satisfaction**: identifying any alternatives which satisfy some given requirements.

There are two general models for MADM, namely, a noncompensatory model which does not permit tradeoffs between attributes, and a compensatory model which permits tradeoffs between attributes. About 20 different methods for MADM have been proposed. They apply to different situations depending upon the model used and the information obtained from the decision maker (selector), for example, whether or not any preference on attributes and/or alternatives is given. A taxonomy of these methods can be found in [37].

The selection problem discussed in this chapter uses the compensatory model, and the goal is to find an alternative which satisfies the requirements and is optimal. It is assumed that the information about the selector's preferences on attributes is given in a weight vector, and no selector's preference about alternatives is given.<sup>9</sup> Five methods can be applied to this class of MADM problems.

### 3.7.1. Linear assignment method

Suppose that for every attribute  $att_j$  the values  $x_{ij}$ ,  $i = 1, \dots, m$ , can be ranked. Let  $\pi$  be an  $m \times m$  nonnegative matrix where each element  $\pi_{ik}$  represents the number of times that  $A_i$  ranks in the  $k$ th place in a ranking based on attributes. If different weights for attributes are considered,  $\pi_{ik}$  will be the summation of  $w_j$  for those  $att_j \in \{att_j \mid x_{ij} \text{ is ranked the } k\text{th place for } att_j\}$ . Observe that a larger  $\pi_{ik}$  value indicates more concordance in assigning  $A_i$  to the  $k$ th overall ranking, thus the problem of evaluating the candidates becomes that for each  $k$ , find a  $A_i$  which maximizes  $\sum_{k=1}^m \pi_{ik}$ . In order to give a unique preference ranking of the  $A_i$ 's, a 0-1 integer programming problem can be formed as follows.

---

<sup>9</sup>This is a reasonable assumption, because at the beginning of a selection process the selector is not familiar with the alternatives, and those methods which are based on information about alternatives usually require pairwise comparison among alternatives, which is not feasible for a domain with many attributes and alternatives.

$$\begin{aligned}
 & \max \sum_{i=1}^m \sum_{k=1}^m \pi_{ik} P_{ik} \\
 & \text{subject to } \sum_{k=1}^m P_{ik} = 1, \quad i = 1, 2, \dots, m \\
 & \quad \quad \quad \sum_{i=1}^m P_{ik} = 1, \quad k = 1, 2, \dots, m
 \end{aligned}$$

where  $P_{ik} = 1$  if  $A_i$  is assigned to overall rank  $k$ , otherwise  $P_{ik} = 0$ . The solution to this LP problem indicates an appropriate ordering of the  $m$  alternatives.

### 3.7.2. Simple additive weighting method (SAW)

Simple additive weighting method may be the most commonly used method of MADM. Suppose that the decision maker assigns a set of weights  $W = (w_1, w_2, \dots, w_n)$  to the attributes, which are normalized to  $\sum_{j=1}^n w_j = 1$ , and that the intra-attribute values have been numericalized and translated into a comparable scale. Then a total score for each  $A_i$  can be defined as

$$\text{score}(A_i) = \frac{\sum_{j=1}^n w_j \times x_{ij}}{\sum_{j=1}^n w_j}$$

If the attributes are *mutually preferentially independent*, an additive utility function [38] can be substituted for the score function, which has the form

$$U(x_1, x_2, \dots, x_n) = \sum_{j=1}^n w_j U_j(x_j)$$

where  $U_j$  is a utility function for att <sub>$j$</sub> .

In either case, the most preferred candidate is the one which has the highest score (or utility). This is the method we used in our prototype system because of its simplicity and intuitive nature. It is reported that "theory, simulation computations, and experience all suggest that simple additive weighting method yields extremely close approximations to very much more complicated nonlinear forms, while remaining far easier to use and understand" [37].

### **3.7.3. Hierarchical additive weighting method**

When an MADM problem has a large number of attributes, the hierarchical structure of the selection domain may be utilized to simplify the evaluation process. The objectives of MADM can be partitioned into levels: the top level consists of a single element and each element on a given level covers some or all of the elements in the level immediately below. Then some MADM approach can be applied to compare elements in a single level with respect to a purpose from the adjacent higher level. This process is repeated higher up in the hierarchy and the results are successively weighted and composed into an overall priority ranking of the alternatives.

### 3.7.4. ELECTRE method

One way to solve an MADM problem is to find the partial ordering relation among alternatives. The ELECTRE (Elimination et Choice Translating Reality) method uses an *outranking relationship* to define the partial order for alternatives, where  $A_i$  *outranks*  $A_k$  if the decision maker considers  $A_i$  to be better than  $A_k$ , although  $A_i$  does not dominate  $A_k$  mathematically.<sup>10</sup> ELECTRE uses the following steps to assess the outranking relationship.

1. Calculate the weighted normalized decision matrix  $V$ , where each entry in  $V$  is defined by the formula

$$v_{ij} = w_j \times \frac{x_{ij}}{\sqrt{\sum_{i=1}^m x_{ij}^2}}$$

2. Determine the concordance and discordance set. For every pair of  $A_i$  and  $A_k$  ( $i \neq k$ ), the concordance set  $C_{ik}$  contains the indices of all attributes for which  $A_i$  is preferred to  $A_k$ , i.e.,

$$C_{ik} = \{ j \mid v_{ij} \geq v_{kj} \}.$$

The discordance set is defined by  $D_{ik} = \{ j \mid v_{ij} < v_{kj} \}$ .

3. Determine the concordance dominance matrix  $F$ , which is an  $m \times m$  Boolean matrix, where

---

<sup>10</sup>Mathematically,  $A_i$  *dominates*  $A_k$  if  $A_i$  is better than  $A_k$  in one or more attributes and is equal to  $A_k$  in the remainder. The dominance relation can be used to eliminate the dominated alternatives.



$$f_{ik} = \begin{cases} 1 & \text{if } \sum_{j \in C_{ik}} w_j \geq \bar{c} \\ 0 & \text{otherwise} \end{cases}$$

where  $\bar{c}$  is a predetermined threshold value.  $f_{ik} = 1$  implies that  $A_i$  dominates  $A_k$  for the concordance attributes.

4. Determine the discordance dominance matrix  $G$ , which is an  $m \times m$  Boolean matrix, where

$$g_{ik} = \begin{cases} 1 & \text{if } \frac{\max \{v_{ij} - v_{kj} \mid j \in D_{ik}\}}{\max \{v_{ij} - v_{kj} \mid \forall j\}} \leq \bar{d} \\ 0 & \text{otherwise} \end{cases}$$

where  $\bar{d}$  is a predetermined threshold value.  $g_{ik} = 1$  implies that  $A_i$  dominates  $A_k$  for the discordance attributes.

5. Determine the aggregate dominance matrix  $E$  which is the intersection of  $F$  and  $G$ , where each entry  $e_{ik} = f_{ik} \times g_{ik}$ , and  $e_{ik} = 1$  implies that  $A_i$  is preferred to  $A_k$ .

6. Eliminate the less favorable alternatives. The aggregate dominance matrix gives the partial-preference ordering of the candidates. Any  $A_i$  which is outranked by some other candidate can be eliminated.

### 3.7.5. Technique for ordering preference by similarity to ideal solution (TOPSIS)

Assume that each attribute takes a monotonical utility. Let the *ideal solution* be composed of all best attribute values attainable, and the *negative-ideal solution* be composed of all worst attribute values attainable. The TOPSIS method is based on the concept that a preferred alternative should have the shortest distance from the ideal solution and the farthest from the negative-ideal solution. The algorithm is as follows.

1. Construct the weighted normalized decision matrix V, where

$$v_{ij} = w_j \times \frac{x_{ij}}{\sqrt{\sum_{i=1}^m x_{ij}^2}}, \quad \text{for } i = 1, \dots, m. \text{ \& } j = 1, \dots, n.$$

2. Determine the ideal and negative-ideal solutions. Suppose the ideal solution is  $A^* = (v_1^*, v_2^*, \dots, v_n^*)$ , and the negative-ideal solution is  $A^- = (v_1^-, v_2^-, \dots, v_n^-)$ .

3. Calculate the separation measure which is defined by the n-dimensional Euclidean distance. The separation of each alternative from the ideal solution is given by

$$S_i^* = \sqrt{\sum_{j=1}^n (v_{ij} - v_j^*)^2}, \quad i = 1, 2, \dots, m.$$

Similarly, the separation of each alternative from the negative ideal solution is given by

$$S_i^- = \sqrt{\sum_{j=1}^n (v_{ij} - v_j^-)^2}, \quad i = 1, 2, \dots, m.$$

4. Calculate the relative closeness to the ideal solution, which is defined as

$$C_{i*} = \frac{S_i^-}{S_i^+ + S_i^-}, \quad i = 1, 2, \dots, m \quad (0 < C_{i*} < 1).$$

5. Rank the preference for alternatives according to the descending order of  $C_{i*}$ 's. As  $C_{i*}$  approaches 1,  $A_i$  approaches the ideal solution.

Comparing the selection problem model we assumed and the MADM problems, the main differences are the following.

- The importance of requirements. Note that in the MADM methods which aim at finding the optimal solution, there are usually no explicitly specified requirements. In the case of satisfaction, there is a set of requirements which define the minimal acceptable levels of attributes. In our model, the importance of the requirements is emphasized. There is a set of clearly defined requirements which are fuzzy bounds on the acceptable values for attributes. This is a more realistic model for many engineering problems where requirements can be quantitatively defined and there is some freedom for adjustment.
- Optimization and satisfaction. For those optimization methods, the goal is to optimize every attribute as much as possible, while for those satisfaction approaches, it is not important to

distinguish between the alternatives as long as they satisfy all the requirements. In our model of selection, the goal is a combination of satisfaction and optimization, i.e., to satisfy all requirements and optimize the total penalty-credit. Our ideal solution is defined by the requirement vector rather than the best possible attribute values, because our goal is to find the alternative which is closest to or most exceeds the requirements.

- Explicit utility function associated with each attribute. Many MADM methods assume that there is an implicit utility function associated with each attribute, for example, it is monotonically increasing or decreasing. In some cases only ordinal information among attribute values are used. We defined the PCFs explicitly and quantitatively. The penalty-credit functions defined in this chapter resemble the value functions (which are utility functions used in the case of certainty) employed in some MODM methods [38]. But we linked the PCFs with the requirements such that they can be used to more precisely define the criteria for a selection.
- Employ artificial intelligence methods to deal with the entire selection process. The MADM methods end up with either a single alternative or a partial ordering of the alternatives. The main focus of those methods is to help a decision maker assess

weights, scale attribute values, and compare alternatives. No backtracking and ramification analysis are involved. In our proposed approach, selection is viewed not only as an evaluation problem but as a complete process. The problems of efficient backtracking and intelligent help, such as ramification analysis, are addressed. A knowledge based system for assisting selection is proposed, and the framework of such a system is described.

In summary, the model of selection problems defined in this chapter is different from the models used in most MADM methods, and is more suitable for the class of selection problems we are considering. For evaluating candidates, our method is a variation of some MADM methods, especially the utility functions and simple additive weighting method. The approach proposed here deals with an entire selection process which has not been addressed in the MADM methods.

## Chapter 4

# A knowledge based system for selecting PLA TDMs

In Chapter 3, we presented a general model of the selection problem, and suggested using a knowledge based consultant system to assist in the selection. To help designers create testable VLSI systems, a design for testability expert system TDES [1, 88] is being developed. An important part of TDES is the selection of a suitable TDM for a kernel under a set of design constraints. This is a typical selection problem, in which the selector is a circuit designer, the receiver is a custom circuit design, and the domain of selection consists of testable design methodologies. In this chapter we will describe a prototype of the knowledge based consultant system for TDM selection, referred to as PLA-TSS, which stands for the PLA TDM Selection System.

### 4.1. Overview of PLA-TSS

As stated in Chapter 1, design for testability for VLSI systems involves partitioning a large circuit into tractable kernels, selecting suitable TDMs for every kernel, and embedding the TDMs into the chip. There are several basic types of kernels for which TDMs exist. PLAs are a typical example.

As discussed in Chapter 2, there are many TDMs for PLAs. The performances of these TDMs vary with the PLAs' parameters and/or personalities. It is difficult for a circuit designer to select a TDM that best meets certain requirements, because very often the designer is not familiar with all the TDMs and there are too many factors to consider. The knowledge based consultant system, PLA-TSS, has been developed to assist a designer in selecting the most suitable TDM for a given PLA.

#### 4.1.1. The task of PLA-TSS

From Chapter 3 we have shown that  $T_i = (v_{i1}, v_{i2}, \dots, v_{iN})$  is a solution to a selection problem  $\langle \text{TDM}, R, \text{PLA} \rangle$ , if for the given PLA,  $T_i$  satisfies the requirement vector  $R$ , i.e.,  $\forall j, r_j \leftarrow v_{ij}$ .  $T_i$  is a better solution than  $T_j$  if  $\text{score}(T_i) > \text{score}(T_j)$ .  $T_i$  is the best solution if  $T_i$  is a solution and for all  $T_j$  satisfying  $R$ ,  $\text{score}(T_i) \geq \text{score}(T_j)$ .

Given a knowledge base containing a number of TDMs for PLAs, and a specific PLA to be made testable, the task of PLA-TSS is as follows.

1. Assist a designer in specifying a set of requirements and evaluation criteria for selecting TDMs.
2. Determine which TDM satisfies the requirements. Suggest the best solution, if any.
3. If no solution exists, provide advice and help the designer search through the solution space in order to find an acceptable solution.

4. Manage design data and history.

5. Maintain the knowledge base.

It is often the case that either the initial requirements are not realistic (over-constrained problem), or realistic requirements cannot be completely satisfied due to the solution space. The important feature of such an expert consultant system is that it must not only have the ability of finding a solution when one exists, but also be responsible for helping a selector find a satisfactory solution when no ideal solution exist. The system should be intelligent in that it can reason in a "natural way" like a human expert. This requires expert domain knowledge, an intelligent control mechanism and friendly selector-consultant interaction.

#### 4.1.2. The structure of PLA-TSS

The basic structure of PLA-TSS is given in Figure 4-1.

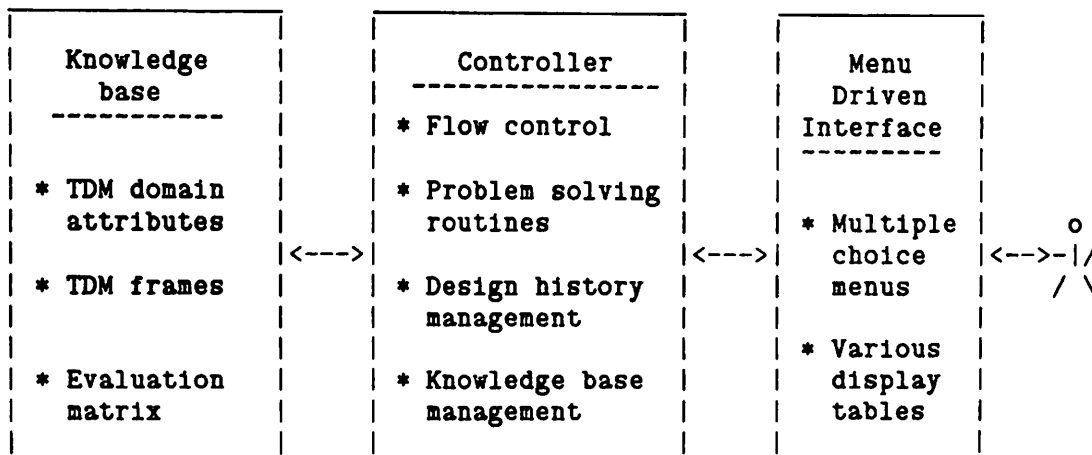


Figure 4-1: Overview of PLA-TSS



## The knowledge base

An important part of PLA-TSS is a knowledge base which contains the following information.

- A set of attributes which characterize TDMs. These attributes have been discussed in Chapter 2 and are listed in Table 3-1. Associated with each attribute are its properties, unit of measurement, and definition.
- TDMs for PLAs. TDMs are represented by TDM frames. Each specific TDM is an instant of the TDM frame. An example of a TDM representation has been given in Chapter 3. Currently, there are fifteen TDMs in the knowledge base. Some measures associated with each TDM are expressed as functions of a PLA's parameters, and their values determined when a particular PLA is specified.
- An evaluation matrix. For a given PLA, a completely specified evaluation matrix can be built in which rows correspond to TDMs and columns correspond to attributes. Each entry  $v_{ij}$  in the evaluation matrix is the value of  $TDM_i$  for the attribute  $att_j$ .

## The controller

The controller of PLA-TSS is separated from the knowledge base and has four basic functions.

1. Control a dynamic selection process.

The controller is responsible for determining when to do what. It first receives a custom designed PLA, and helps a designer to define goals and constraints, i.e., the requirement vector. Then it initiates the search process through the knowledge base to see whether or not there is any TDM satisfying these requirements. If so, the best one is suggested to the designer. If there is no solution, a process called **reason analysis** is invoked to find out why the search has failed and how to resolve the failure. At this point, the process will continue following the designer's responses. Once a solution is found by any means, the system will inform the designer about the details of the selected TDM, and give a chance for confirmation. If the designer is not satisfied with the solution, PLA-TSS will ask for the reason for rejection and again do reason analysis to see what can be done to rectify the problem. This process continues until either a satisfactory solution is found or the designer concludes that no solution exists and quits.

2. Make a selection.

To simulate a human expert, the consultant system knows about the art of selection. This knowledge is represented in a set of problem solving procedures, such as ramification analysis, search routines, and the comparison, PC and score functions for evaluating TDMs. These procedures will be called at the appropriate time to allow the system to make a good selection decision.

### 3. Keep track of design data and history.

In order to allow backtracking in a selection process, the controller keeps a record of the entire design history. A history consists of a sequence of states in a selection process. Each state is represented by the values of global variables. At every decision point or when the system state is changed, the old state is recorded so that a designer can trace back (review) or back up to any earlier state and undo some actions.

### 4. Manage the knowledge base.

Tools are provided for maintaining the knowledge base, such as changing information, reviewing the contents of the knowledge base, deleting information or acquiring new TDMs.

### **The interface**

Interaction between a designer and the system is through a menu driven interface. Many different tables are used to display system states at various situations. Menus with multiple choices are shown which allow a designer to select his most preferable actions.

In this chapter four major aspects of PLA-TSS will be discussed in detail, namely, the evaluation functions, the search procedures, reason analysis, and the control flow.

## 4.2. The evaluation functions

Since the consultant system must be able to determine the goodness of TDMs with respect to the designer's requirements, a method for evaluating TDMs is necessary. In Chapter 3 it has been shown that in order to evaluate a TDM, one needs a requirement vector, a weight vector, a compare function, a penalty-credit function for each attribute, and a score function which combines different attribute values of a TDM into a single numeric "score" representing the overall performance of the TDM. In the following we will discuss the evaluation functions used in PLA-TSS.

### 4.2.1. The weight vector

In general, attributes of TDMs do not have the same importance to all designers and applications. Priorities among attributes will greatly affect the result of selection. To indicate these priorities, a weight vector

$$W = (w_1, w_2, \dots, w_N)$$

should be specified by the designer, where  $w_j$  is the relative weight of the  $j$ th attribute. The  $w_j$ 's are non-negative numbers, and  $w_i > w_j$  implies that  $att_i$  is more important than  $att_j$ . The relative value of the weights  $w_j$ 's can be converted into normalized weights  $nw_j$ s using the formula

$$nw_j = \frac{w_j}{\sum_{j=1}^N w_j}. \quad \text{Thus } 0 \leq nw_j \leq 1.$$

### 4.2.2. Penalty-credit functions

Penalty-credit functions (PCFs) defined in Chapter 3 provide a very useful way for designers to express their criteria of selection. PCFs are also application-dependent. However knowledge about the general forms of PCFs can be used to help designers in defining appropriate PCFs. In PLATSS, there are three ways a designer can specify PCFs.

1. Define own PCFs.
2. Modify the standard PCFs.
3. Use the default PCFs.

These methods differ in two aspects: (1) how to divide the possible values into groups, and (2) what functions are used in each group to produce the desired penalty or credit.

#### 4.2.2.1. Custom-defined PCFs

User defined PCFs can be any function following the general form of PCFs given in Chapter 3. The main features of a custom-designed PCF is that the value range of an attribute can be divided into any number of regions, and any meaningful function can be used in each region.

**Example.** Consider a PCF for fault models. Assume there are four fault models: a, b, c and d. If the designer wants faults in class a to be definitely detected, and prefers faults in class b and c to be detected as well, but does not care about faults in class d, the PCF for fault models can be defined as follows.

$$PC_{fm}(r, v) = \begin{cases} 40, & \text{if } \{ a, b, c \} \subseteq v; \\ 30, & \text{if } \{ a, b \} \subseteq v \text{ or } \{ a, c \} \subseteq v; \\ 20, & \text{if } a \in v; \\ -300, & \text{if } \{ b, c \} \subseteq v \text{ and } a \notin v; \\ -500, & \text{if } b \in v \text{ or } c \in v, \text{ but } a \notin v; \\ -2000, & \text{if } v \text{ does not contain } a \text{ or } b \text{ or } c. \end{cases} \quad (4.1)$$

Note that a custom defined PCF may be independent of the requirement, although it should imply the requirement. It may include the weight factor in it, since the value of  $PC(r, v)$  is arbitrary. This provides a way of achieving non-linear weights. The designer can emphasize his likes or dislikes by letting the PCF produce various values in different ranges. He can also define non-monotonic PCFs which do not follow the general form of PCFs.

#### 4.2.2.2. Standard PCFs

Since there is usually some commonality in most PCFs, a parameterized standard form of a PCF is defined for each type of attribute. A designer can use his own parameters to modify these standard forms. The main difference between standard PCFs and custom PCFs is in the way the entire value range is divided, especially for numeric attributes.

For numeric attributes, the standard PCF has the form shown in Figure 3-2. Let  $r_j = (r_{j1}, r_{j2})$ <sup>11</sup> represent the current requirement for att<sub>j</sub>. The designer should specify a *saturation point* (SP) and an *unacceptable point* (UP). The entire value range is divided into five segments, namely,

<sup>11</sup>It is possible that  $r_{j1} = r_{j2}$ .

- segment I:           the saturation segment    [ best(att<sub>j</sub>), SP ],
- segment II:           the credit segment           ( SP, r<sub>j1</sub> ),
- segment III:          the acceptance segment    [ r<sub>j1</sub>, r<sub>j2</sub> ],
- segment IV:          the penalty segment       ( r<sub>j2</sub>, UP ),
- segment V:           the unacceptable segment [ UP, worst(att<sub>j</sub>) ].

The designer is then asked to choose the function to be used in each segment, which can be a constant function or a function of  $r_j$  and a value for att<sub>j</sub>.

**Example.** Assume  $SP = 10$  and  $UP = 50$ . Using the standard PCF for numerical attributes, a PCF for area overhead can be defined as follows.

segment	value range	Function used within the segment	Remark
I	[0 , 10 ]	C	credit
II	[10 , r ]	$A \times e^{\ln(C/A) \times \text{compare}(r,v) / \text{compare}(r,10)}$	credit
III	[r , r ]	A	accept
IV	[r , 50 ]	$- [\text{compare}(r,v)]^2$	penalty
V	> 50	D	unacceptable

$$C > A > 0, \quad D \ll 0.$$

This function is plotted in Figure 4-2.

For logical attributes, the standard form for a PCF is the same as in Figure 3-3. The designer is asked to specify the values of A, B, C and D. In general, for a value of  $v$  such that  $\text{compare}(r, v) \geq 0$ , then  $PC(r, v) \geq 0$ . Otherwise  $PC(r, v) < 0$ . For example, if a designer does not want test generation, he may define the PCF for test generation as shown in table 4-1.

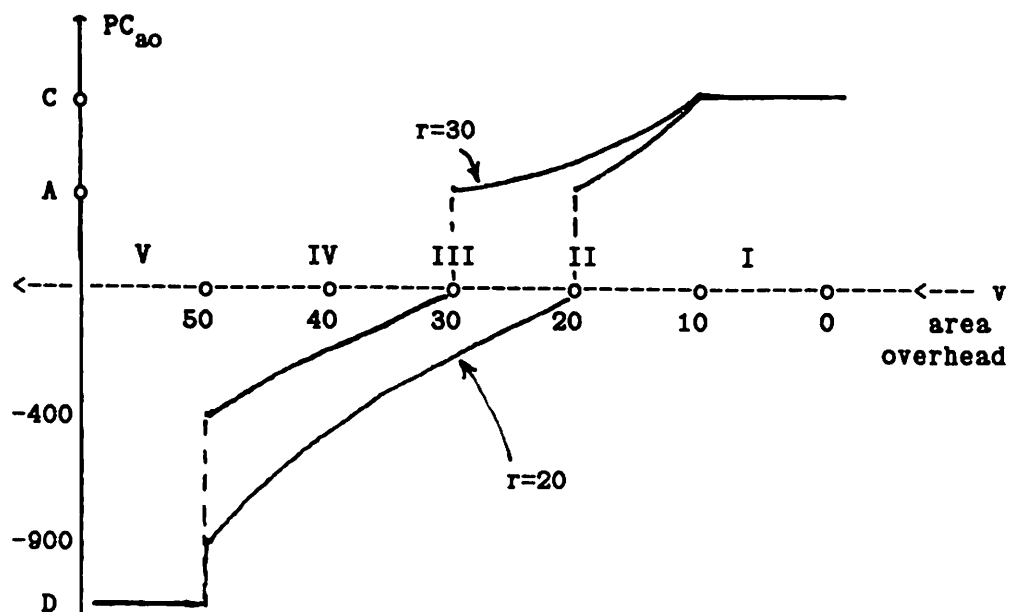


Figure 4-2: The PCF for area overhead for  $r = 20$  and  $r = 30$

requirement	value	credit	penalty
YES	YES		- 100
YES	NO	30	
NO	YES		-2000
NO	NO	20	

Table 4-1: A PCF for test generation

#### 4.2.2.3. Default PCFs

Being an expert consultant, default knowledge is necessary. PLA-TSS has a procedure for automatically defining default PCFs according to the user's requirements. The system defined PCFs are designed to accommodate general situations and be fair to all attributes.

For numeric attributes, the default PCF is defined below, where A, B, C and D are constants and can be changed by the user or an expert. The graphic illustration of the default PCF is shown in Figure 4-3.



$$PC_j(r_j, v) = C, \text{ if } compare(r_j, v) > 0.5r_j$$

$$PC_j(r_j, v) = A + \frac{C-A}{0.5r_j} \times compare(r_j, v), \text{ if } 0 < compare(r_j, v) \leq 0.5r_j$$

$$PC_j(r_j, v) = A, \text{ if } compare(r_j, v) = 0.$$

$$PC_j(r_j, v) = B - \frac{B-D}{0.5r_j} \times compare(r_j, v), \text{ if } 0 > compare(r_j, v) \geq -0.5r_j$$

$$PC_j(r_j, v) = E, \text{ if } compare(r_j, v) < -0.5r_j$$

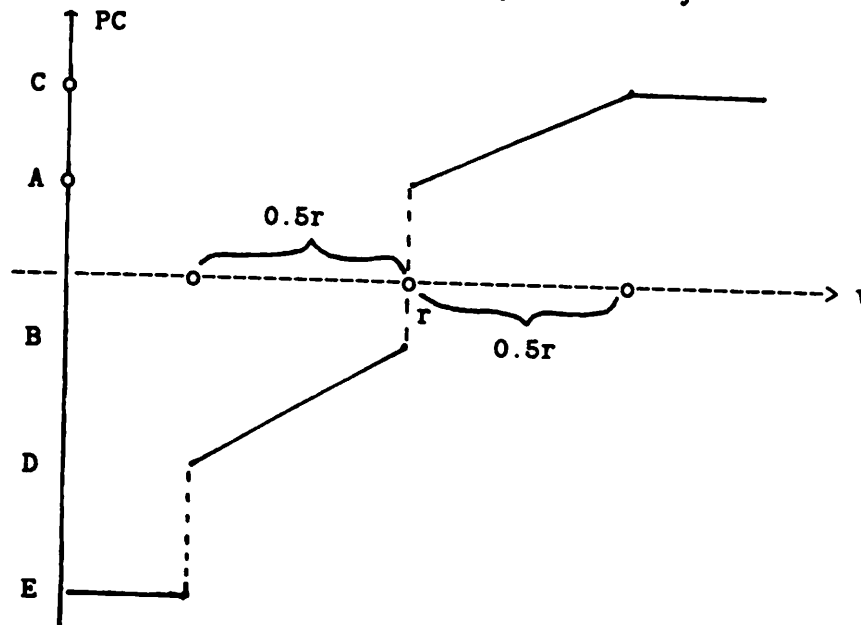
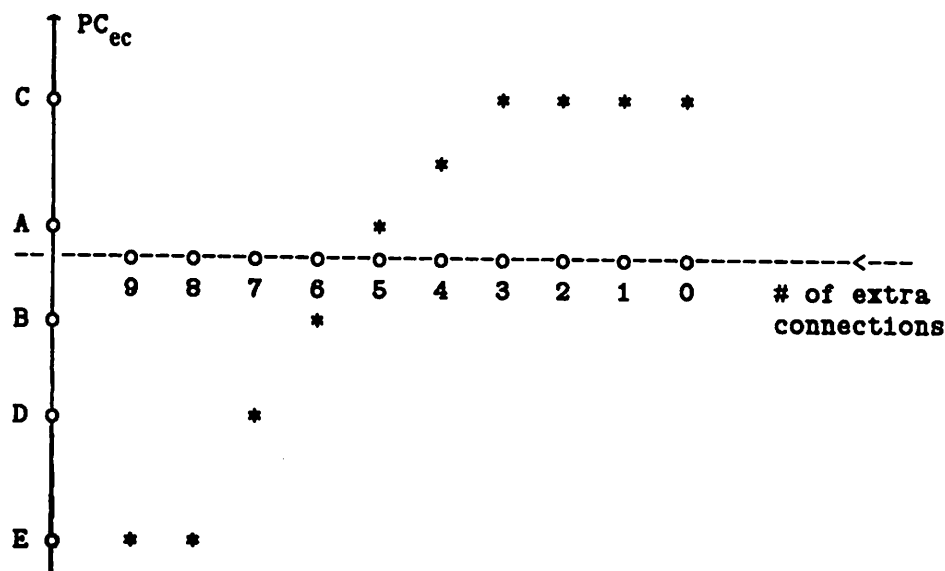


Figure 4-3: The default PCF for numeric attributes

Similar to the standard PCF, for a default PCF the value range is automatically divided into the five segments based upon a requirement. The requirement defines the acceptance segment. The credit segment includes those values which do not exceed the requirement by more than 50%, and the penalty segment includes those values which are worse than the requirement by no more than 50%. The functions used in the saturation segment, acceptance segment and unacceptable segment are constants, and

the functions used in the credit and penalty segments are linear functions of the distance between the requirement and the value. To be fair for all attributes, if a value is in the acceptance segment it gets a fixed credit A. If a value reaches or exceeds the saturation point, it receives a fixed credit C. If a value is in the unacceptable segment, it receives a fixed penalty of E. The slope of the PCF in the penalty and credit segments varies with the magnitude of the attribute values, such that the penalty or credit for attribute values with different units and magnitudes can be automatically mapped into the same scale. As an example the system's default PCF for the requirement "extra I/O pins is 5" is shown in Figure 4-4.



**Figure 4-4:** The PCF for extra connections

For logical attributes, there are only four combinations of requirements and values. The differences among PCFs lie in the choices for penalty or credit for each of the four cases. For complex attributes, numerous

combinations exist. However they can be classified into three cases, namely a value either exceeds, satisfies or fails a requirement. The default PCFs for logical and complex attributes are defined uniformly as follows.

$$PC_j(r_j, v) = D, \text{ if } compare(r_j, v) < 0.$$

$$PC_j(r_j, v) = A, \text{ if } compare(r_j, v) = 0.$$

$$PC_j(r_j, v) = C, \text{ if } compare(r_j, v) > 0.$$

Here A, C and D ( $D < 0$ ) are the same as for the numeric attributes.

**Example.** The default PCF for self-testing is shown in Table 4-2.

requirement	value	credit	penalty
YES	YES	A	
YES	NO		D
NO	YES	C	
NO	NO	A	

**Table 4-2:** The default PCF for self-testing

For this case a large credit is given to a value YES when  $r = \text{NO}$  to increase the attractiveness of self-testing even when not specified as a requirement.

Default PCFs do not imply any priority among attributes. Priorities are specified via the weight vector. The three methods for defining PCFs provide enough capability and flexibility for a designer to conveniently specify most meaningful PCFs.

It is the combination of requirements and PCFs that completely defines the criteria for selection. PCFs defines the tightness or looseness of the requirements, and are very important in making a decision.

### 4.2.3. The requirements

Normally the requirements for selection can be represented by a requirement vector  $R = (r_1, r_2, \dots, r_N)$ . There are several cases in which a requirement vector may take a different form.

**1. Highlighted requirements.** Some attributes may be subordinate to others, and their requirements can be implied by the requirements for other attributes. On the other hand, a designer may not care about some attributes. Therefore not all attributes need appear explicitly in the requirement vector. In general, when specifying requirements we consider a subset of attributes  $(att_1', att_2', \dots, att_n')$ , where  $n \leq N$ ,  $att_j' \in \{att_1, att_2, \dots, att_N\}$ , and the requirement vector is  $R = (r_1', r_2', \dots, r_n')$ .

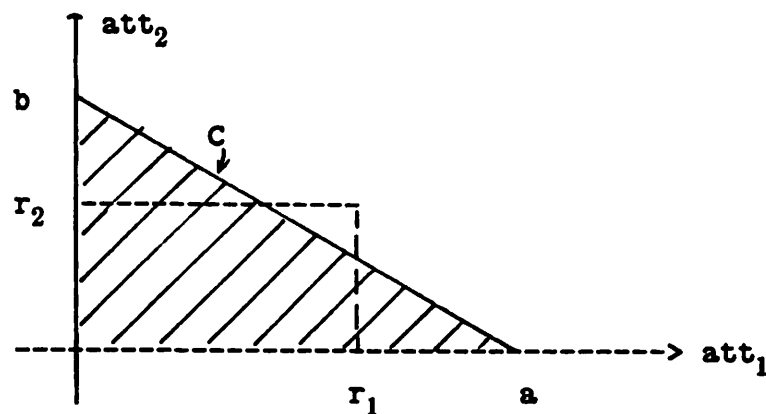
**2. Extended requirements.** A requirement, especially for numeric attributes, need not be a single value but can be a value range  $[r1, r2]$ , where  $r1$  is called the higher requirement and  $r2$  the lower requirement. For example, the requirement for area-overhead can be  $[5, 15]$ . Such a requirement defines a non-single-point acceptance segment. In this case a value  $v$  is said to satisfy the requirement if  $r1 \leq v \leq r2$ .  $v$  exceeds the requirement if  $compare(r1, v) > 0$ .  $v$  fails the requirement if  $compare(r2, v) < 0$ . The procedure for defining PCFs can be easily modified to handle these extended requirements.

**3. Joint requirements.** So far we have just considered separate

requirements for individual attributes. It is possible to have joint requirements which specify the relationship between two or more requirements.

**Example.** Suppose there are two upper-bound numeric attributes  $att_1$  and  $att_2$ . When considering individual attributes separately, the requirements  $r_1$  and  $r_2$  define a square solution area as shown in Figure 4-5. However, if the desired solution area is a triangle bounded by the line C and the X and Y axes, as shown in Figure 4-5, individual requirements are not enough. This requirement can be expressed by the joint requirement

$$\frac{v_1}{a} + \frac{v_2}{b} \leq 1.$$



**Figure 4-5:** Solution area indicated by requirements

In general, a joint requirement for  $k$  attributes  $att_1, \dots, att_k$  has the form

$$Q[f(v_1, \dots, v_k), r_1, \dots, k]$$

where  $Q$  is a predicate,  $f$  is a function on the values of  $att_1, \dots, att_k$ , and  $r_1,$

$\dots, k$  is the required value of  $f(v_1, \dots, v_k)$ . The joint requirement is satisfied if  $Q[f(v_1, \dots, v_k), r_1, \dots, k]$  is true. There should be a joint PCF for each joint requirement, which has the form  $PC[f(v_1, \dots, v_k), r_1, \dots, k]$ .

There is no problem in handling joint requirements using a slightly modified version of the scheme developed for single requirements. In the following we will only consider single requirements.

#### 4.2.4. The score function

Using the normalized weights, the PCFs and the compare function defined in Chapter 3, many evaluation functions can be defined to combine multiple features of a TDM into a single score. Two score functions are used in PLA-TSS.

1. The total score is defined by the equation

$$score(TDM_i) = \sum_{j=1}^N PC_j(v_{ij}, r_j) \times nw_j.$$

The total score reflects tradeoffs between different features of a TDM so as to represent the overall performance of the TDM. A TDM whose attribute values receive more credits and less penalties will have a higher score. In general, if  $score(TDM_i) > score(TDM_j)$ ,  $TDM_i$  has more desirable attribute values than  $TDM_j$ . However, since the credits and penalties of a TDM offset each other in the total score,  $score(TDM_i) > score(TDM_j)$  does not necessarily imply that  $TDM_i$  satisfies more requirements than  $TDM_j$ . This can be detected, if needed, by modifying the score function.

2. The loss score (LS) is defined by the equation

$$LS(TDM_i) = \sum_{j=1}^N \delta(\text{compare}(r_j, v_{ij})) \times [PC_j(v_{ij}, r_j) - PC_j(r_j, r_j)] \times nw_j,$$

where  $\delta(x) = 1$  if  $x < 0$ , otherwise  $\delta(x) = 0$ .

The loss score of a TDM indicates the degree to which a TDM does not satisfy the requirements. It is the summation of the weighted negative distance between a TDM and the requirement vector. If TDM<sub>i</sub> satisfies all requirements,  $LS(TDM_i) = 0$ , otherwise,  $LS(TDM_i) < 0$ . The larger the value of  $LS(TDM_i)$ , the closer the TDM is to the requirements.

These two score functions have different properties. Under the assumption that the goal of selection is to choose a TDM with the best overall performance, the total score is used to compare and rate TDMs. The total score indicates a relative ranking of TDMs. It not only helps identify the "best" TDM, but provides a broad range of choices. In case a designer wants to make a minimal change to the requirements in order to obtain a solution, the loss score will suggest the best candidate.

Note that both the weights and the PCFs affect the score. There are three ways to combine weights and PCFs.

1. Merge weights into PCFs and set all weights to 1 when calculating the scores. For instance, one can use custom defined PCF in which weights are implied by the magnitude of the penalties and credits. The advantage is that the weight can be non-uniform.

2. Define weightless PCFs and use weights to enhance the influence of important attributes. For example, the default PCFs are totally weightless. For different attributes, they return a normalized penalty or credit value. By multiplying the PC values by corresponding weights, the contributions of various attributes to the total score will be different. This is an easier way to ensure the consistency among PCFs, and makes the weights explicitly represent the relative importance of attributes.
3. Define partially weighted PCFs and use weights to enhance or reduce their influences to the total score. This is a strategy mixing 1 and 2 above. It has the advantages and disadvantages of both.

It is possible to put these three strategies into a score function. The three options provided by PLA-TSS allow a user to define any reasonable PCF for any attribute, either weighted or unweighted. The weight factors used to calculate the scores may not be the normalized weights, but rather weights adjusted according to definitions of the PCFs. The designer is responsible for keeping the definitions consistent with the actual needs.

**Example.** Suppose there are five TDMs and six attributes as shown in Table 3-6. We will use the PCFs for self-testing, test generation, fault-model, area overhead and extra I/O connections defined in Table 4-2 and



4-1, equation (4.1) and Figure 4-2 and 4-4, respectively. Assume the PCF for fault coverage is as follows.

$$PC_{fc} = A + \text{compare}(r,v) \times 10, \text{ if } v \geq 97;$$

$$A + \text{compare}(r,v) \times 20, \text{ if } v < 97.$$

Let  $A = 20$ ,  $C = 30$ ,  $B = -5$  and  $D = -2000$ . For the given requirements shown in row "Req." of Table 4-3, and the relative weights given in row "R.W.," the normalized weights (N.W.) and scores are as shown. Each entry corresponding to  $TDM_i$  and  $att_j$  is the value of  $PC_j(r_j, v_{ij})$ . The total scores for the TDMs are listed in the last column.

	self- testing	test generation	fault model	fault coverage	area overhead	extra i/o connections	
Req.	yes	no	{a}	97	30	5	total
R.W.	3	2	1	6	8	4	score
N.W.	0.125	0.083	0.042	0.25	0.333	0.167	
TDM1	-2000	-2000	30	40	50	20	-384.75
TDM2	20	20	30	-20	-25	20	-4.565
TDM3	20	20	40	50	-144	15	-32.117
TDM4	20	20	20	30	26.03	-2000	-312.83
TDM5	30	20	40	50	-2000	10	-687.26

**Table 4-3:** Calculation of scores [I]

These scores show that no TDM completely satisfies the requirements, though TDM2 comes closest. Now if we change the weight of fault coverage and area overhead to 9 and 5, respectively, and re-calculate the scores, the score of TDM3 becomes the largest, as shown in the Table 4-4. This means that if fault coverage is more important than area overhead, TDM3 is the best alternative.

	self- testing	test generation	fault model	fault coverage	area overhead	extra i/o connections	
Req.	yes	no	{a}	97	30	5	total score
R.W.	3	2	1	9	5	4	
N.W.	0.125	0.083	0.042	0.375	0.208	0.167	
TDM1	-2000	-2000	30	40	50	20	-388.86
TDM2	20	20	30	-20	-25	20	-3.94
TDM3	20	20	40	50	-144	15	-2.875
TDM4	20	20	20	30	26.03	-2000	-312.34
TDM5	30	20	40	50	-2000	10	-388.49

Table 4-4: Calculation of scores [II]

If the constraint for extra connections is not very tight, and we change a part of the PCF for extra I/O connections to

$$PC_{ec} = - (B * compare(r,v)), \text{ if } v > 5 ,$$

then the scores are as shown in Table 4-5. In this case, TDM4 is the best.

	self- testing	test generation	fault model	fault coverage	area overhead	extra i/o connections	
Req.	yes	no	{a}	97	30	5	total score
R.W.	3	2	1	9	5	4	
N.W.	0.125	0.083	0.042	0.375	0.208	0.167	
TDM1	-2000	-2000	30	40	50	20	-388.86
TDM2	20	20	30	-20	-25	20	-3.94
TDM3	20	20	40	50	-144	15	-2.875
TDM4	20	20	20	30	26.03	-10	19.99
TDM5	30	20	40	50	-2000	10	-388.49

Table 4-5: Calculation of scores [III]

Scores are the main criteria used by the expert consultant to select a TDM. The preceding example shows how the scores change with weights,

requirements and PCFs, and help in identifying the most suitable candidates. The main property of the score function is that if the value of an attribute is better than the corresponding requirement value, it will increase the score, otherwise the score decreases. The larger the value of a weight  $w_j$  associated with an attribute  $att_j$ , the more the value of  $att_j$  will influence the total score, unless  $PC_j(r_j, v_{ij}) = 0$ . In general, the better a TDM satisfies a requirement vector, the higher will be its score. Using the score function one can sort the TDMs to identify potential candidates, or to select a TDM with optimal overall performance.

#### 4.3. Searching in the three dimensional design space

##### 4.3.1. Variations of TDMs

In the previous discussion we considered each TDM to be a single point in the multi-dimensional solution space. However, many TDMs are not of a unique form. A TDM may be implemented in several ways. There are three kinds of variations associated with most TDMs.

1. **Characteristic variations.** These are internal variations of a TDM which lead to different attribute values. All characteristic variations of a TDM use the same test scheme, have the same testability features and logical attribute values. They only differ in minor ways used to implement the test scheme, for example, using a parity tree or a parity chain in order to compute the parity.

**2. Environmental variations.** These are external variations of a TDM which use different auxiliary hardware to support the TDM. As an example, for PLAs with universal tests, the test set can be either generated on-line by some special hardware, stored in an on-chip ROM/RAM, or applied from off-chip ATE.

**3. Embedding variations.** These are choices to be made among the various elements within the circuit under test (CUT) when deciding how to realize the required support hardware defined by the environmental variation of a TDM. For example, if a TDM needs a linear feedback shift register (LFSR) for on-line test generation, and the CUT has one LFSR and two shift registers which can be modified into LFSRs, there is a choice of which of these three registers one should use as the test generator.

**Definition 4.1:** Let  $V1 = (v_{11}, v_{12}, \dots)$  and  $V2 = (v_{21}, v_{22}, \dots)$  be two variations of a TDM.  $V1$  is dominated by  $V2$  if and only if for all  $j$ ,  $v_{1j} \leftarrow v_{2j}$ .

There may be numerous variations of a TDM; they can differ from one another by a small or large amount. We will only consider non-dominated variations, and simply call them "variations." It is clear that each variation is superior to the other variations in at least one attribute.

A TDM may have a number of characteristic variations. For each characteristic variation, there may be several environmental variations. For

each environmental variation, there may be many embedding variations. Some characteristic variations are independent of environmental variation in the sense that every environmental variation can be applied to it. However some characteristic variations are dependent on the environment so that specific environments are required. Embedding variations are dependent on environmental variations. On the other hand, selection of an environmental variation is affected by its embedding variations, since the embedding variations indicate the feasibility and economy of the environmental variation.

A complete version of a TDM is a valid combination of one of the TDM's characteristic variations and one of its environmental variations. A TDM is a solution if any of its versions satisfies all requirements. A TDM fails to satisfy a requirement vector  $R$  if all of its versions fail to satisfy  $R$ .

Every variation will affect some aspect of a final solution. The relation between each type of variations and the TDM's attribute values are summarized in Table 4-6. Therefore, selection of a TDM for a given kernel involves searching in three directions, namely

1. among different TDMs,
2. among different implementation variations of one TDM, and
3. among different embeddings in a given environment.

Attribute	characteristic	environmental	embedding
Fault model	no	no	no
Fault coverage	probably	no	no
Fault masking	no	no	no
Concurrent testing	no	no	no
Error latency	probably	no	probably
Self testing	no	no	no
Function dependence	no	no	no
Logic overhead	probably	no	probably
Extra I/O connections	probably	probably	probably
Area overhead	probably	probably	probably
Extra delay	probably	no	no
Delay per test	probably	probably	probably
# of test patterns	probably	no	no
Test storage	probably	probably	no
Test application time	probably	probably	probably
Test generation means	probably	no	no
Test application means	no	probably	no
Response evaluation means	probably	probably	no
Logic design cost	probably	probably	probably
Layout cost	probably	probably	probably

Table 4-6: Effect of the three variations on TDM attribute values

### 4.3.2. Characteristic Variations of TDMs

Characteristic variations usually involve tradeoffs between several numeric attributes.

**Example.** The autonomous testable design for PLAs, shown in Figure 2-11, has three variations.

- **Variation 1.** No flag circuit. The final signature in the product term selector will be shifted out for inspection. The advantage of this variation is that area overhead is reduced due to the elimination of the flag decoder. The disadvantages are that test application time is increased by  $m+4$  clock cycles in order to shift out the signature, and the method for response evaluation is changed from checking a one-bit go/no-go signal at the last cycle of testing, to shifting and evaluating an  $m+4$  bit long signature.
- **Variation 2.** The feedback value generator and the flag circuit are not duplicated. In this variation, area is saved but fault coverage will drop since faults in FVG and FC may not be detected and some faults in the original PLA may be masked.
- **Variation 3.** Use a parity tree instead of a parity chain for an OR array parity checker. The delay caused by a cascade of XOR gates will be reduced from  $\max\{k+1, \log_2(m+1)\}$  to  $\max\{\log_2(K+1), \log_2(m+1)\}$ , so that the clock cycle can be

shorten and the total test application time reduced. The side effect is that some faults in the OR parity tree may not be detected and some faults in other parts of the PLA may be masked [83].

Characteristic variations of a TDM can be considered as new TDMs. But that would increase the number of TDMs considerably, resulting in much redundant information in the knowledge base and a slow down in the selection process. Note that the value of the logical attributes is the same in all characteristic variations of a TDM; different variations only affect some numeric attribute values. Thus it appears preferable not to handle variations as new TDMs. In PLA-TSS, each TDM has a basic form as shown in Table 2-3, and there is a slot in the TDM frame called *variation*. If a TDM has characteristic variations, there will be one sub-slot for each variation, and only those attribute values of a variation which are different from the basic version are recorded. During searching, each TDM is checked using the following procedure.

**Procedure 4.1 Checking characteristic variations of a TDM**

1. Evaluate the basic form of  $TDM_i$ .
2. If it satisfies the requirements,  $TDM_i$  is a solution. Determine the best variation of  $TDM_i$ . Replace the basic form by that variation if it is different from the basic form. Stop.



3. If  $TDM_i$  has variations, for each variation, evaluate the attribute values which are different from the basic form.
4. If some variations satisfy the requirements,  $TDM_i$  is a solution. Determine the best variation, and replace the basic form by that variation. Stop.
5. If no variations of  $TDM_i$  satisfy the requirements, remove  $TDM_i$  from the solution set.

#### 4.3.3. Environment consideration in selecting a TDM

Once a TDM's characteristic variation is chosen, an environmental variation needs to be selected, if choices exist. This step is important because it may not only affect the final solution in many aspects such as area overhead, test application time, test storage, and extra IO connections, but may also greatly affect the embedding of the TDM into a real circuit and test schedules for the kernel [1].

An environment for a TDM consists of two parts: the means for real-time test application (the driver) and the means for response evaluation (the receiver). Generally speaking, these two parts are independent. Let

$TA_i = \{ ta_1, \dots, ta_p \}$  be a set of test application hardware for  $TDM_i$ , and  $RE_i = \{ re_1, \dots, re_q \}$  a set of response evaluation hardware for  $TDM_i$ ,

where the  $ta_k$ 's and  $re_j$ 's are sets of hardware which constitute sufficient means for providing a given test set and evaluating the test results,

respectively. A valid environment  $ENV_i$  for  $TDM_i$  is a combination of a  $ta_k$  and a  $re_j$ , i.e.,

$$ENV_i = \{ (ta_k, re_j) \mid ta_k \in TA_i \text{ and } re_j \in RE_i \}.$$

Some rules for combining  $ta_k$ 's and  $re_j$ 's exist, e.g., if  $ta_k = \{ATE\}$  and  $re_j = \{ATE\}$ , then the combination of  $ta_k$  and  $re_j$  is still  $\{ATE\}$ . But if  $ta_k = \{LFSR\}$  (for random pattern generation) and  $re_j = \{LFSR, \text{flag-circuit}\}$  (for signature analysis), the two LFSRs can not be combined into one, because one LFSR cannot be shared to perform these two functions at the same time.

Let  $ENV_i = \{e_1, e_2, \dots, e_l\}$  be a set of valid environments for  $TDM_i$ . An  $e_k$  is available if the hardware contained in it exists on or off the CUT and there is an identity transfer path (I-path)<sup>12</sup> between components of  $e_k$  and the kernel. An  $e_k$  is partially available if there exists some hardware  $h$  which can be modified to function as  $e_k$ , and  $h$  is connected to the kernel by an I-path.

The problem of finding the availability of the on-chip  $e_k$ 's has been solved by Abadir [1, 2]. An embedding algorithm is developed which takes a TDM template and finds all elements which can be used or modified as a driver and/or a receiver. In an available environment, there may be a number of feasible embeddings. Embedding variations affect area overhead,

---

<sup>12</sup>An I-path between port X to port Y is a transfer path over which data can be transferred from X to Y without being modified. For more details, see [2].

testing time and test scheduling. A set of measures are associated with each embedding, which includes the costs of each driver / receiver.

The cost of an environment variation is associated with a subset of the TDM's attributes, as listed in Table 4-6. The environment cost can be represented by a vector

$$C(e_k) = (c_{k1}, c_{k2}, \dots, c_{kN}),$$

where  $c_{kj} = 0$  if  $att_j$  is not affected by any environmental variation. Otherwise  $c_{kj}$  is the value of  $att_j$  associated with  $e_k$ , and can be found in the embedding process. Due to possible multiple embeddings for each environment variation, an  $e_k$  can be implemented at different costs.

Note that the environment cost is independent of the costs caused by the characteristic variation of a TDM. The real cost of applying  $TDM_i$  is

$$TDM_i + C(e_k) = ((v_{i1} + c_{k1}), \dots, (v_{iN} + c_{kN})).$$

The procedure for environment selection is given below.

#### **Procedure 4.2. Environment selection for a TDM**

- Inputs:**           A partial solution  $TDM_i$ .  
                       A requirement vector R.  
                       A circuit containing the kernel to be made testable.
- Outputs:**         The measures of  $TDM_i$ .  
                       The most suitable environment variation of  $TDM_i$ , if any.
1. Obtain  $TDM_i$ 's environment variations from the knowledge base.
  2. Obtain the designer's environmental constraints which specify what external test aids can be provided.

3. Use the embedding procedure to carry out the following steps.
  - Identify all available test drivers and receivers in the CUT.
  - Find all possible implementations of each environment variation of  $TDM_i$ . If an environmental variation is currently not implementable, find possible ways to modify the CUT to construct the necessary environment,
  - Calculate the costs of each environment variation of  $TDM_i$ .
4. If no low cost environment exists, negotiate with the designer to see whether it is possible to provide a required environment. If possible, add the necessary hardware features and go to 3. Otherwise,  $TDM_i$  cannot be a solution. Stop.
5. If there are several feasible environment variations,
  - choose the one with minimal cost,
  - choose an embedding for the environment which, when combined with the chosen version of the TDM, results in the highest score.Otherwise, go to 8.
6. Update the measures of  $TDM_i$  by adding the environment cost.

7. If  $TDM_i$  satisfies the requirements, stop. Otherwise, go to 5 to see if other environment variations are better.
8. Stop with an environment variation which results in the highest score of  $TDM_i$ .

#### 4.3.4. The general search procedure

We have seen that searching for a good TDM under given requirements is not a simple matter of matching the requirements with attribute values, but involves searching among different TDMs, among various variations of TDMs, and among different supporting environments as well as embeddings. Observe that

- as far as the major attributes are concerned, the differences between variations of one TDM are minor compared with the differences between different TDMs,
- environment variations of TDMs are dependent on the characteristic variations,
- embedding variations of an environment depend on the environmental variation, and determine the availability and costs of the environmental variation.

The complete search process is carried out in two interactive steps: first searching for a good (characteristic version of) TDM, and then determining the test environment. A complete search procedure is outlined below.

**Procedure 4.3. The complete search process**

**Inputs:** A requirement vector  $R = (r_1, r_2, \dots, r_N)$ .  
 A CUT containing the PLA to be made testable.  
 A knowledge base containing TDMs  $\{ TDM_1, TDM_2, \dots, TDM_M \}$ , where each  $TDM_i$  has a basic form  $TDM_i = (v_{i1}, v_{i2}, \dots, v_{iN})$ , and may have several characteristic and environmental variations.

**Outputs:** A solution set  $S$  containing all qualified TDMs.

1. For every  $TDM_i$  in the knowledge base:
  - Check if  $TDM_i$  (including its characteristic variations) satisfies  $R$  using Procedure 4.1.
  - If yes, put  $TDM_i$  into a partial solution set  $PS$ .
2. If  $PS$  is empty, the search fails. Go to 4.
3. For each  $TDM_i$  in  $PS$ :
  - Select an environmental variation for  $TDM_i$  using Procedure 4.2.
  - If the updated measures of  $TDM_i$  fail to satisfy  $R$ , remove  $TDM_i$  from  $PS$ .
    - If  $PS$  is not empty, output  $PS$ . Every TDM in  $PS$  is a solution. Suggest the TDM with the highest score as the best solution. Stop.
    - If  $PS$  is empty, no solution exists.

4. Go through reason directed backtracking until a suitable TDM is found.

#### 4.4. Reason Analysis

A selection process is usually not straightforward, but consists of a sequence of failures and backtracking. There are two primary kinds of failures:

1. **search failure:** no solution is found in a search,
2. **confirmation failure:** a solution is rejected by the designer.

In order to make a selection efficiently, it is important to identify the reason for failure in order to determine where and how to backtrack. This is the task of reason analysis. In PLA-TSS, whenever a failure is encountered reason analysis is used to determine the cause of the failure and then suggest the best way to resolve the problem. The basic principles of reason analysis have been discussed in Chapter 3. In this section we will describe some detailed implementation issues related to reason analysis in PLA-TSS.

##### 4.4.1. Reason analysis for search failures

A search failure occurs when no TDM completely satisfies a requirement vector  $R$ . Since TDM attribute values are fixed for a given kernel, the only way to resolve a search failure is to change  $R$  such that a solution can be obtained. How to change  $R$  depends on why the failure occurred.

**Example.** Consider the evaluation matrix in Table 3-6. If the requirements are

self-testing: yes                      test generation: no  
 fault model: {a}                      fault coverage: 97  
 area overhead: 30                      extra connections: 6,

no TDM is completely satisfied. Let

$$\text{FAIL}(\text{TDM}_i) = \{ (\text{att}_j, v_{ij}) \mid \text{TDM}_i \text{ fails to satisfy } r_j \}.$$

The FAIL lists associated with the five TDMs are shown in Table 4-7.

TDM	FAIL(TDM <sub>i</sub> )
TDM1	(self-testing, no), (test generation, yes)
TDM2	(area overhead, 35), (fault coverage, 95)
TDM3	(area overhead, 42)
TDM4	(extra connections, 7)
TDM5	(self-testing, no), (area overhead, 300)

**Table 4-7: The FAIL lists**

Each row in Table 4-7 represents a reason for failure. For example, the third row indicates that the requirement for area overhead is too high. If it is changed to 42, TDM3 will be a solution. The fourth row implies that the requirement for extra connections causes the failure. If one more extra connection is possible, TDM4 will be a solution. However the first row suggests that if the requirement for self-testing is changed to "no" and test generation becomes allowable, TDM1 will be a solution without increasing the requirements on area overhead or extra I/O connections.

This example shows that there usually many reasons for search failure,



and there are many ways to revise requirements. It is necessary to identify the most critical reason for failure, and find out how to backtrack such that a good solution can be found. In PLA-TSS, each time a search fails, reason analysis is used to classify failures and identify the most critical failure category which contains the current failure mode. Once the reason for failure is understood, the system informs the designer of the reason, suggests possible ways to recover from the failure, then restarts a search process if the designer accepts the advice and/or adjusts the requirement values.

#### **4.4.1.1. Classification of failures**

In section 3.3.3 we have pointed out that a solution space can be divided into many different regions. When a requirement vector falls into a failure region, a search failure occurs. Five classes of search failures exist, which are caused by five classes of conflicts between the requirements and TDM's attribute values, namely three types of critical conflicts, a single conflict and multiple conflicts. There is a specific way for solving each class of failure.

#### **Critical Conflicts**

**1. First class critical conflict:** There is a requirement  $r_j$  that no TDM can satisfy.

**Syndrome:** There exists at least one  $att_j$  such that  $compare(req_j, best(att_j)) < 0$ .

**Remedy:** When such a critical conflict occurs, the only way to continue the selection process is to relax the corresponding requirement to below  $\text{best}(\text{att}_j)$ . In this case, a conflict table is displayed which indicates which requirements cause critical failures, and to what extent these requirement values should be relaxed in order to possibly obtain a solution. If the designer agrees to change the requirement values, the system will request that he modify the requirements shown in the table, and then restart the search process. Otherwise, the selection process ends with no solution.

**2. Second class critical conflict:** There is a requirement  $r_j$  such that no TDMs which are close to the requirements can satisfy it.

**Syndrome:** There exists  $\text{att}_j$  such that  $\text{compare}(\text{req}_j, \text{best}\{v_{ij} \mid \text{score}(\text{TDM}_i) > L\}) < 0$ , where  $L$  is a pre-set lower bound on TDM scores.

Given  $R$ , the TDMs can be divided into the two sets

$T1 = \{ \text{TDM}_i \mid \text{score}(\text{TDM}_i) > L \}$ , and

$T2 = \{ \text{TDM}_k \mid \text{score}(\text{TDM}_k) \leq L \}$ .

$T1$  contains TDMs which are relatively close to  $R$ , and therefore is referred to as the set of potential solutions. If  $T1$  is not empty, a selection will be made from within  $T1$ , and TDMs in  $T2$  are very unlikely to be selected unless the requirements are significantly changed.

The difference between first and second class critical conflicts is that the critical requirement  $r_j$  can not be satisfied by any TDM in  $T1$ , although it can be met by some TDMs in  $T2$ .

**Remedy:** In this case, there are two ways to solve the problem. First, relax the requirement on  $att_j$  to below  $\text{best}\{v_{ij} \mid TDM_i \in T1\}$ , such that it can be satisfied by some TDMs in T1 and a solution may be found. Secondly, change other requirements such that some TDMs in T2 which satisfy  $r_j$  will become close to the new requirement. However, since all TDMs in T2 have poor scores, any such TDM which satisfies  $r_j$  must have some very bad or unacceptable value in at least one attributes other than  $att_j$ . Comparing these two choices, the second one requires radical changes in order to lead to a solution, while the first one may only involve small changes. The system will inform the user of these two ways of changing requirements, and suggest the user change  $r_j$ .

**3. Third class critical conflict:** No TDMs are close to R.

**Syndrome:** T1 is empty.

**Remedy:** This is the case where every TDM has a very bad score. Major changes to several requirements may be necessary, and these may involve changing an unacceptable value to an acceptable one. The system first determines several necessary changes to make, and then suggests the best ones to the selector. Since all TDMs have low scores, for each  $TDM_i$ , a *critical fail set*  $CF_i$  can be found. Let

$$CF_i = \{(att_j, v_{ij}) \mid PC_{att_j}(v_{ij}, r_j) < K \},$$

where  $K$  is a pre-set negative number and  $K \leq \text{PENALTY}$  (the system's default fixed penalty value).

**Definition 4.2:** Let  $CF_i$  and  $CF_j$  be two critical fail sets. We say that  $CF_i$  is weaker than  $CF_j$ , denoted by  $CF_i \preceq CF_j$ , if for every  $(att_k, v_{ik}) \in CF_i$ , there exists a pair  $(att_k, v_{jk}) \in CF_j$  such that  $v_{jk} \leq v_{ik}$ . If  $CF_i \preceq / CF_j$  and  $CF_j \not\preceq CF_i$ , we say  $CF_i$  and  $CF_j$  are incomparable.

Each  $CF_i$  specifies necessary changes to the requirements in order to make  $TDM_i$  satisfiable. However if  $CF_i \preceq CF_j$ , the changes specified by  $CF_j$  are not necessary for obtaining a solution. A set of necessary changes can be identified using Procedure 4.4.

**Procedure 4.4. Identify necessary changes**

**Inputs:** A set of  $TDM_i$ s involved in a critical failure, each having a critical fail set  $CF_i$ .

**Outputs:** A set  $CF$  containing pairwise incomparable  $CF_i$ s.

1. Set  $CF = \emptyset$ .
2. For every  $TDM_i$ :
  - If there exists  $CF_j \in CF$  such that  $CF_i \preceq CF_j$ , replace  $CF_j$  by  $CF_i$ .
  - Otherwise, if there is no  $CF_j \in CF$  such that  $CF_j \preceq CF_i$ , add  $CF_i$  to  $CF$ .

At the end of this procedure, each element of  $CF$  represents a unique way to change requirements such that at least one  $TDM$  will become free of unacceptable values. The system will suggest a way in which a few requirements can be modified so as to obtain a solution.

**Example.** Suppose that when a third class critical conflict occurs, the 15 critical fail sets associate with 15 TDMs are as shown in Table 4-8.

TDM <sub>i</sub>	CF <sub>i</sub>
BIST2	{(AREA-OVERHEAD 172.5) (TEST-APPLICATION-TIME 0.1E-3)}
BIT	{(AREA-OVERHEAD 387.2)}
BILBO	{(AREA-OVERHEAD 786.4) (TEST-APPLICATION-TIME 0.1E-3)}
CONC	{(AREA-OVERHEAD 605.2) (TEST-APPLICATION-TIME 0.2E-3) (TEST-STORAGE 231)}
EXH	{(TEST-STORAGE 96.0)}
FIST	{(AREA-OVERHEAD 249.0)}
FIT	{(AREA-OVERHEAD 191.9) (EXTRA-IO-PINS 9) (TEST-APPLICATION-TIME 0.1E-3) (TEST-STORAGE 270.0)}
MFT	{(AREA-OVERHEAD 224.0) (EXTRA-IO-PINS 7) (TEST-APPLICATION-TIME 0.3E-3) (TEST-STORAGE 329)}
SIG	{(AREA-OVERHEAD 846.0) (TEST-APPLICATION-TIME 0.2E-3)}
AUTO	{(AREA-OVERHEAD 319.6) (TEST-APPLICATION-TIME 0.3E-3)}
SMITH	{(TEST-STORAGE 208)}
TLO	{(AREA-OVERHEAD 35.4) (TEST-STORAGE 234)}
TSBF	{(AREA-OVERHEAD 19.3) (TEST-STORAGE 21)}
UT2	{(AREA-OVERHEAD 79.3) (TEST-STORAGE 140)}
UTS	{(AREA-OVERHEAD 133.3) (TEST-STORAGE 128)}

**Table 4-8:** The critical fail lists for 15 TDMs

By applying Procedure 4.4, CF is identified as

$$CF = \{[(AREA-OVERHEAD 19.3) (TEST-STORAGE 21)] \\ [(TEST-STORAGE 96.0)] \\ [(AREA-OVERHEAD 249.0)] \\ [(AREA-OVERHEAD 172.5) (TEST-APPLICATION-TIME 0.1E-3)]\}.$$

**Single conflict:** There exists a TDM which meets all but one requirement.

**Syndrome:** There exists a TDM whose FAIL list has only one element.

**Remedy:** If a single conflict occurs, a solution can be obtained by changing a single  $r_j$ . In this case, a conflict table is displayed which shows

which TDM fails which requirement. The designer is advised that if he is willing to change the value of a requirement to the extent shown in the table, a solution is obtained. If several single conflicts occur simultaneously, the consultant can suggest which change leads to the best solution. Since the designer can see how close he is to a solution, he may decide which requirement to change or which TDM to choose. In case a requirement is changed, its new value is compared with the corresponding TDM's value. If the TDM satisfies the new requirement, a solution is found, otherwise, a search process should be carried out again.

**Multiple conflict:**

A TDM fails to satisfy at least two requirements.

**Syndrome:** There exists a  $TDM_i$  such that  $|FAIL(TDM_i)| > 1$ .

**Remedy:** If a multiple conflict occurs, many alternatives exist, especially when many TDMs have multiple conflicts. The designer is informed that tradeoffs must be made among the various attributes. A conflict table is presented which informs the designer as to what conflicts exist. Scores of TDMs are calculated which reflect the overall performance of each TDM. Several TDMs with high scores are suggested as possible candidates. Advice about the most beneficial tradeoffs is given, which is based on the TDM scores. The designer can select a TDM, or ask the system to select the best one, or change some requirements and let the system search again.

#### 4.4.1.2. Ordering of failure classes

It is not enough to only classify failures, because several classes of failures may be present at the same time. Precedences among failure classes must be defined such that an intelligent decision about which failure to solve first can be automatically made. In PLA-TSS, the failure classes are ordered and processed according to the following rules.

1. Critical failures are dealt with first. Among them, the precedence is first class critical conflicts, followed by the second class, and finally the third class.
2. If there are no critical failures, single failures are handled before multiple failures. But if the failure class associated with the best (highest score) TDM is a multiple failure, it is treated together with single failures. Such a multiple failure may lead to a minimal degree of changes. If two TDMs are equally good and they fall into single and multiple conflicts, respectively, then the single conflict will be processed first.

Reason analysis for search failures is implemented as a small rule based system, combined with failure handling procedures. The rules (translated into English) are listed below.

```
If      search fails,  
        reason is unknown,  
then    find the first class critical conflicts if any exist.
```

```
If      search fails,  
        reason is unknown,
```

then     there is no first class critical conflict,  
          find the second class critical conflicts if any exist.

If       search fails,  
          reason is unknown,  
          there is no first or second class critical conflict,  
then     find the third class critical conflict if any exists.

If       search fails,  
          reason is unknown,  
          there is no critical conflicts,  
then     find single conflicts if any exist,  
          otherwise there are multiple conflicts.

If       search fails,  
          reason is "critical conflict,"  
then     solve the critical conflict,  
          stop.

If       search fails,  
          reason is "single conflict,"  
then     solve the single conflict,  
          solve the multiple conflict associated with  
          the TDM of the highest score, if any,  
          stop.

If       search fails,  
          reason is "multiple conflicts,"  
then     solve multiple conflicts,  
          stop.

Here, "find" and "solve" are procedures. The rules are checked and fired sequentially until a STOP is encountered. Priorities of failure classes can be changed by changing these rules. More failure classes and procedures for resolving these classes can be added, in which cases the rules for controlling reason analysis should be modified too.



#### 4.4.2. Reason analysis for confirmation failures

In an interactive selection process involving a designer and a consultant, there are four possible ways in which a solution can be obtained, namely

1. chosen by the designer,
2. chosen by the consultant under the designer's request,
3. found during a search,
4. reached by mistake, e.g., typed a wrong key.

In any case, the designer may decide that a "solution" is unacceptable, and has the right to reject it. To implement this process, the consultant requires confirmation before a solution is finalized. A confirmation failure occurs if a solution is rejected at this point.

Confirmation failures are usually due to the fact that the designer does not like some of the attribute values in a solution. The reason for failure can only be identified by the designer. In PLA-TSS, during last-chance confirmation, if the designer does not accept a solution, a table will be displayed which indicates the attribute values of the selected TDM together with the requirements. The designer is asked why the TDM is not acceptable, and all attributes and their values so identified are put into a set DL. For any consistent response,  $DL \neq \emptyset$ .

Once the reasons for a confirmation failure are known, the next action

is usually dependent on whether there is any TDM which may satisfy the designer. Since the designer does not like the attribute values in DL, an alternative solution should be better than the current solution in these attributes. Based on this principle, four failure classes and their corresponding resolution procedures are defined as follows.

**FC1.** There exist some TDMs which are better than the current solution in all attributes in DL, although they may be worse in other aspects.

**Syndrome:**  $T1 = \{ TDM_i \mid \forall j \text{ s.t. } (att_j, x_j) \in DL, x_j \leftarrow v_{ij}, \text{score}(TDM_i) > L \} \neq \emptyset.$

**Remedy:** In this case, TDMs in T1 may be potential solutions, since they have the features the designer likes. TDMs with very bad scores are not included in T1 since some of their features are unacceptable. If T1 is not empty, a table showing a comparison of these alternatives with the current solution is displayed, and the designer is informed that these TDMs are better than the current solution in those attributes in which he is concerned. The designer may look at these alternatives, select a solution from T1, or make some changes to the requirements, weights or PCFs and let the system search again.

**FC2.** There are some TDMs which are better than the current solution in some attributes in DL, but are worse in other attributes in DL.

**Syndrome:**  $T2 = \{ TDM_i \mid score(TDM_i) > L \ \& \ \exists j, k \text{ s.t. } (att_j, x_j) \in DL, (att_k, x_k) \in DL, x_j = v_{ij} \text{ and } x_k \neq v_{ik} \} \neq \emptyset$ .

**Remedy:** In this case, it is impossible to obtain a totally satisfied solution. The designer is told that tradeoff must be made in order to find a solution. A comparison table is also displayed showing possible alternatives which are TDMs in T2. Suggestions about good tradeoff can be given. At this point, it is up to the designer to decide what to do next.

**FC3.** There is no TDM which is better than the current solution in any attribute in DL.

**Syndrome:** Both T1 and T2 are empty.

**Remedy:** This means that the current solution is the best as far as the attributes in DL are concerned. There does not appear to be a better alternative. The system will inform the designer of this situation, and suggest reconsidering the current solution.

**FC4.** The designer doesn't like the current solution at all.

**Syndrome:** DL contains all attributes.

**Remedy:** In this case, there is no need to check the attributes one by one. The ranking of the current solution is checked. If some TDMs score higher than the current solution, they will be displayed and suggested as

possible solutions. Otherwise, the designer is informed that this is the best solution, and a table comparing the current solution and other top candidates is displayed. If the designer really wants a solution, he can reconsider the current one, or change some requirements and restart again.

The rules for ordering and processing these failure classes are

1. if FC4 exists, process it first;
2. if T1 is not empty, process FC1;
3. if T2 is not empty, process FC2;
4. process FC3.

The prototype PLA-TSS has demonstrated that reason analysis directed backtracking is a suitable control strategy for the selection process. This is because a selection process is often failure-driven. Actions to take depend on what kind of failure has been encountered. Reason analysis, combined with scores and ramification analysis, provides very useful information on where and how to backtrack after a failure has occurred.

#### 4.5. The control mechanism

In this section the general structure of PLA-TSS and the major control flow will be described in detail.

An important feature of PLA-TSS is that the consultant system and the designer act at the same functional level, i.e., the control of the selection

process may switch between the two parties. At any point in a selection process, the designer may ask for explanations or a comparison of possible choices, make a decision by himself, let the system select a solution for him, change some requirements, or quit with no solution. On the other hand, the system should follow the designer's requests as well as try to give useful advice to lead the designer through the selection process. This feature implies that the control flow in a selection process cannot be unidirectional. Multiple branching may happen at many points in a selection process. To implement such a system, PLA-TSS uses a combination of three strategies.

- Sequential control. At some stage of a selection process, certain actions must take place one by one. In these cases, the system controls the process by issuing a sequence of instructions or questions.
- Reason directed flow control. At points where different actions can be taken according to the previous system state, reason analysis is used to direct the selection process.
- Multiple branching. At points where different actions can be taken according to the user's needs, multiple-choice menus are presented which allow a designer to select an action.

The basic flowchart of PLA-TSS is given in Figure 4-6.

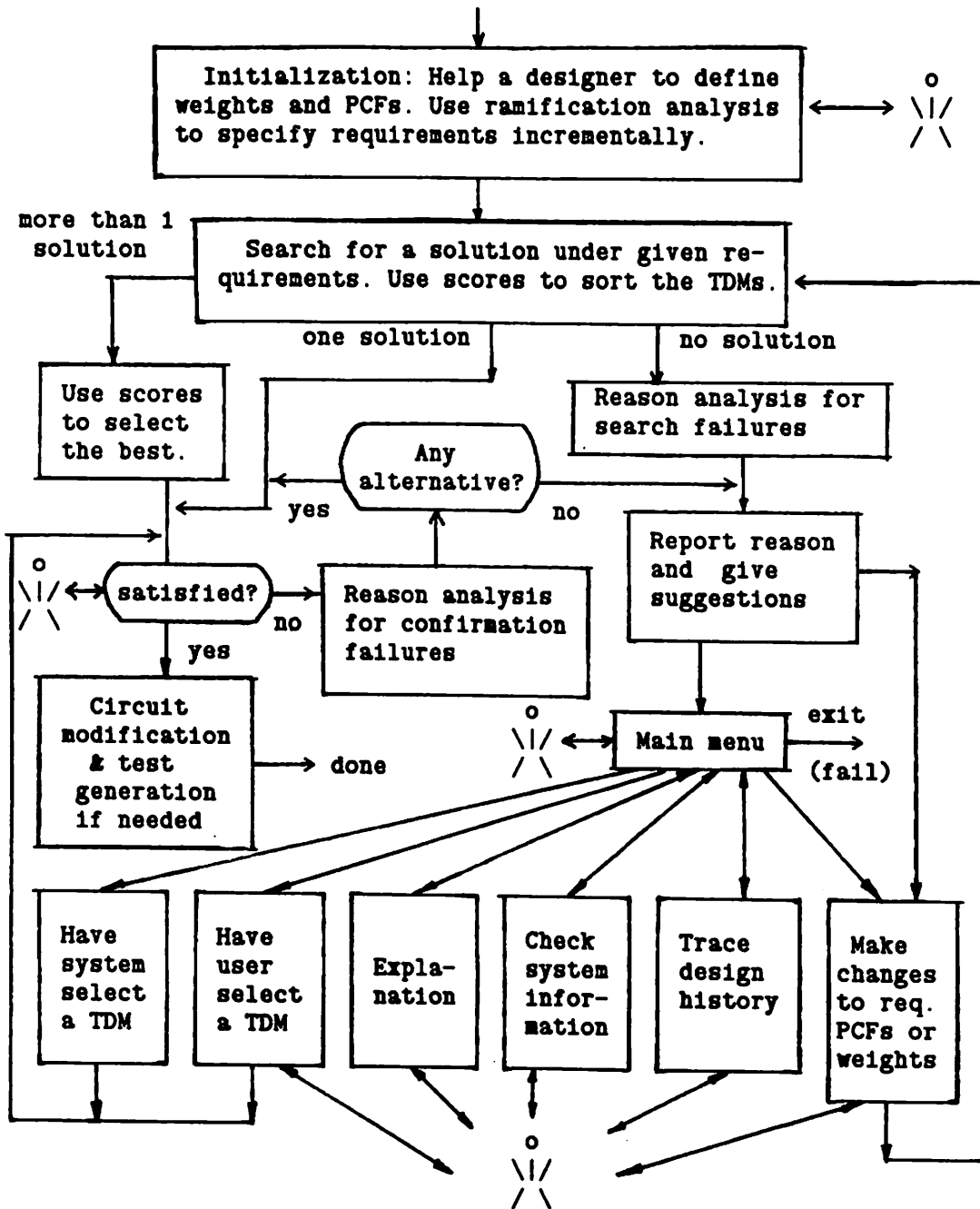


Figure 4-6: Control flow in PLA-TSS

#### 4.5.1. Initialization of a selection process

Initially after a brief system explanation, the system asks where the PLA to be considered is stored and reads in the PLA data. The next steps deal with helping a designer in defining weights, requirements and PCFs.

Weights are usually application-dependent, and hence are specified by the designer. The designer is asked to give the relative weights in the form of a sequence of positive numbers. These relative weights are then converted into normalized weights.

A set of initial requirements must be defined before searching for a solution. PLA-TSS offers a designer three ways to define the initial requirements.

1. Use the default requirements. The default requirements give examples of possible requirements, but are not necessarily appropriate for a given PLA and application. These requirements can be changed to suit one's particular needs.
2. Use the requirements defined last time. This is for the convenience of designers who enter the system repeatedly. The designer can change any of the requirements until a set of initial requirements is specified.
3. Define new requirements. In this case, the incremental

requirement specification method described in Chapter 3 is applied to help a designer in specifying requirements. Starting from the most important attribute, the system interacts with the designer and tries to infer as many ramifications on other requirements as possible from the known requirements using a dynamic ramification tree. It also gives warnings about any inconsistency in the requirement vector. At the end of requirement specification, a set of TDMs which satisfy all requirements can be found, if any.

Semantic checking is made while requirements are specified. For example, if a requirement value should be numeric and the designer specifies a non-numeric value, the system will issue an error message and asks the designer to try again until a numeric value is received.

The last step in initialization is to define a set of penalty-credit functions. The options provided by PLA-TSS, namely, define custom PCFs, modify standard PCFs and use the default PCFs, have been described in section 4.2.2 of this chapter.

After the first pass of specification, the designer has a choice to either modify the initial specification or search for a solution.



#### 4.5.2. Search for a solution

Searching for a solution begins after initialization. The details of the search procedure have been given in section 4.3. Initially, all TDMs are in a solution set SS. If a TDM satisfies the requirements, it remains in SS. Otherwise it is removed from the solution set and a FAIL list is attached to it which contains those attributes which the TDM fails to satisfy.

The outcome of a search is indicated by the solution set SS. There are three possibilities.

- 1). If  $|SS| = 1$ , there is a unique solution. In this case the TDM in SS is suggested to the designer. The system informs the designer that this TDM is the only solution for his requirements. The designer may either accept or reject the solution. If the solution is accepted, the system gives a last chance for confirmation. Once it is confirmed, a test set for the chosen TDM can be generated, and actual circuit modification as well as layout can be carried out. If the solution is rejected, the system enters reason analysis for confirmation failures.

- 2). If  $|SS| > 1$ , there are multiple solutions. In this case the system uses the scores to suggest the best TDM, and a comparison table containing the top three TDMs in SS is displayed. At this point many actions may be taken, for example, the designer can choose a TDM by himself, or look at other TDMs before making a decision. The system displays a main menu

and lets the designer decide what to do. This menu is used repeatedly and will be discussed later.

3). If  $|SS| = 0$ , no TDM satisfies the requirements, i.e., a search failure occurs. In this case, the reason for the failure is determined and actions are taken corresponding to the results of reason analysis. If there is a critical failure, the process allows the designer to modify the requirements. Otherwise, a number of actions are possible. The main menu is displayed, and the designer can determine what to do.

#### **4.5.3. The main user's actions in a selection process**

The main menu of the system contains 10 choices. It gives a designer enough freedom for selecting his preferred action. Six principle actions are discussed below.

- a. Designer selects a TDM.
- b. Ask system to select a TDM.
- c. Make changes in the specifications.
- d. Ask for an explanation.
- e. Check system information.
- f. Trace design history.

#### 4.5.3.1. Designer selects a TDM

The designer always has the right to select a solution. Usually he may do so after he becomes familiar with the TDMs. When the designer chooses this option from the main menu, the system displays a table showing several top choices which have high scores. For each candidate TDM, a group of failed attribute values and the corresponding requirements as well as weights and TDM scores are also displayed. This information helps the designer compare TDMs when making his selection, and to see the distance between each TDM and the requirements. The scores represent the system's point of view which can be used by the designer as a reference. Having displayed this table, the system asks the designer to identify his preferred TDM. If the designer selects a valid TDM, the system goes to the confirmation phase.

By definition, a TDM is a solution if it satisfies all requirements. A TDM selected either by the designer or by the system may not be a solution since it may fail to satisfy some requirements. Confirmation is a process intended to make sure that a selected TDM is really a solution. This can be achieved by changing the requirements if the designer so desires.

In confirmation, the system checks whether the selected TDM satisfies all the requirements. If so, the solution is valid. Otherwise, the system checks to make sure that by choosing this TDM the designer intends to change the requirements such that the selected TDM is completely satisfied. If this is the case, the requirements are changed and a solution is found. If

the designer indicates he does not intend to do this, a contradiction occurs and the solution is not confirmed. In this case, the system suggests that the designer reconsider the selected TDM again, if it has the highest score. Otherwise, a better TDM is suggested. If no TDM can be accepted, reason analysis for confirmation failures will be carried out.

#### **4.5.3.2. Selecting a TDM by the system**

The expert consultant has knowledge of the TDM domain and the responsibility of helping a designer select a good TDM. It keeps track of the best TDM, but it selects a TDM only when explicitly asked to do so.

The system selects a TDM based on the scores. The score for each TDM is calculated during search and changed when the requirements are adjusted. TDMs are sorted by their scores. The TDM with the highest score is considered to be the best and hence selected by the system. A comparison table is displayed which shows the attribute values of the top three TDMs, which, on the one hand, may convince the designer that the TDM selected by the system is the best, and, on the other hand, lets the designer see other potentially good choices.

The TDM selected by the system need not satisfy all requirements nor be acceptable to the designer, due in part to the fact that the PCFs and weights may not correctly represent the designer's selection criteria, and because there may be no satisfactory solutions for the current requirements.

After displaying the table, the system asks whether or not the designer will accept this TDM. If the designer does not like the TDM, the system goes to reason analysis to find out why. Otherwise a confirmation phase follows to make sure that the selected TDM will be accepted as a solution.

#### **4.5.3.3. Making changes**

Selection of TDMs is a dynamic process in which the direction of search may change because the initial requirements are not satisfiable. PLATSS gives designers the opportunity to change the requirement specification at most decision points. Information specified by the designer, such as requirements, PCFs and weights, can all be changed dynamically.

Since the requirements determine the direction and outcome of a selection, the system provides vital aids for changing the requirements.

- **Simple consistency checking.** Semantics of each requirement value is checked to ensure that there is no mistake in requirements. For example, the requirement for extra I/O pins must be an integer and cannot be symbolic or decimal.

- **Ramification analysis.** The ramifications of changing a requirement can be determined using the dynamic R-trees and methods defined in Chapter 3.

- **Suggestion of the best change.** Since the system has full

knowledge about TDMs and the current situation, it can provide the designer information about how to make a minimal number of requirement changes and obtain a solution, or how to change requirements such that the best solution can be obtained.

The "minimal change" is calculated by the loss scores of TDMs. Since a loss score represents the negative distance between a TDM and the requirement vector, the TDM with the largest loss score is closest to the requirements. The changes needed to obtain such a TDM will be suggested to the designer as the minimal changes. However the best TDM may not be the one that requires minimal changes, since it may have some special advantages which outweigh its disadvantages. For this case, the system will also inform the designer that there are other ways of making changes in order to obtain the best solution, and those changes will be suggested as well.

#### **4.5.3.4. Checking system information**

The consultant system does bookkeeping during a selection process so that a designer can check system information while seeking a solution. The system provides information about what has been done, what are the past and current system states, and useful references to aid the designer in making a decision. Information which can be checked includes the following.

**SCORES** shows the scores of all TDMs calculated by two score

functions. SCORE1 is the total score of each TDM. SCORE2 is the loss score. Two lists of TDMs, sorted by the two score functions, respectively, are also given which helps the designer rate the TDMs.

- HISTORY** shows the complete design history from the last decision made to the initial state. The designer can thus see what he has done in previous steps.
- STATE [stage]** gives values of the global variables at the specified stage. These values, such as the requirements, weights, PCFs and the solution set, constitute a complete system state, which can be used in backtracking to restore an earlier state.
- FAIL** shows which TDMs fail to satisfy which requirements.
- OPTIMAL** shows the best possible value for each attribute. This informs the designer as to what can or cannot be achieved, so that he can know the limitations imposed by the solution space and avoid over-specified requirements.
- REQS** shows the current requirements.
- WEIGHT** shows the current relative weights and the normalized weights.

UNIT gives the unit or value range for each attribute.

#### 4.5.3.5. Tracing design history

In a selection process, a designer may have to go back and forth to try different possibilities before reaching a solution. It is very useful to remember what has been done and be able to undo previous steps. To meet this need, PLA-TSS manages design history and allows a designer to review design history, or to back up a certain number of steps to a previous point. Three procedures are provided for this purpose.

##### 1. RECORD changes or decisions made at each step

Recording is transparent to the designer. Values of global variables which constitute the system state and the actions taken are recoded at every decision point or whenever the system state is changed.

##### 2. TRACE the design history

In tracing design history, the system gives two choices.

- Review the whole design history. This process displays, in reverse order, all actions taken from the current stage to the initial state.
- Look backward or forward a number of steps. This allows for the ability to trace the history step by step, or just look at certain steps so that one can concentrate on the most interesting part of the selection process. The designer gives an integer  $i$  representing the number of steps to see. If  $i$  is negative, the



system displays  $|i|$  steps back from the current state pointer (CSP); if  $i$  is positive, the system displays  $i$  steps forward from CSP. If the number of steps specified exceeds the number of those actually stored, tracing will automatically stop at the initial or the last state.

### **3. BACKTRACK in the design process**

This option allows one to backtrack to any previous stage and restart from that point. All actions taken since the specified stage are undone and erased from the recorded history. Currently the system restarts by searching for a solution under the restored state.

#### **4.5.3.6. Explanation**

Explanation is an important issue in expert systems. The expert should be able to explain its line of reasoning in order to convince the user of its conclusions. This is a difficult problem in a complex inference system, but is easier in a selection system. PLA-TSS provides explanation concerning both its knowledge and its reasoning. Explanation for reasoning is embedded in reason analysis and system control flow. The system makes choices based upon scores, reason analysis and ramification analysis. Each time a failure occurs, the system displays relevant information and tells a designer the reason for the failure, and then gives suggestions. Explanation dealing with knowledge includes the following.

- Explaining possible choices. At any point after initialization and

the first pass of search, a designer can ask the system to explain possible solutions. The top three TDMs with the highest scores will be presented as the promising candidates, and tables containing detailed information on these TDMs displayed.

- Explaining principles of TDMs. Details of each TDM in the knowledge base can be viewed by specifying the TDM's name; included in the explanation is how the TDM works and a description of its measures.
- Explaining concepts of the TDM domain. Definitions of the attributes of the TDM domain can be provided during requirement specification and any time in a selection process.

PLA-TSS is programmed in LISP and runs on a DEC-20 machine. Currently it works in an interactive mode. When combining it with an expert synthesis system or design for testability system, it is possible to make the selection process more automatic by using default options. In the next chapter, we will give an example of PLA-TSS operation.

## Chapter 5

### A case study

We have shown in Chapter 2 that there are many TDMs available for PLAs. A methodology for selecting a good TDM is presented in Chapter 3, and a prototype of a knowledge based consultant system for TDM selection, PLA-TSS, is described in Chapter 4. In this chapter, we will go through a selection process using PLA-TSS to demonstrate the principles discussed in the previous chapters and to show how the program works.

In the following we will present a complete working example of PLA-TSS's operation. The script is taken from the screen. Some comments are included in brackets <>. The user's responses are printed in boldface. Everything else are the system's messages. Since this example is provided to demonstrate the various functions of the program, the selection process we are going through may not be typical in the sense that a user can usually use the system in a much more expeditious way to reach a solution. We have tried here to demonstrate as many features as possible.

#### **A selection process through PLA-TSS**

## &lt; Initial computer response &gt;

\*(initial)

Hello! Welcome to our expert consultant system on testable PLA designs. This system is for assisting a circuit designer in the selection and implementation of a suitable testable design methodology (we call it a TDM) for a PLA under a set of design requirements. I'd like to do whatever I can to help you.

Do you want to

1. make a PLA testable,
2. maintain or update the knowledge base,
3. quit ? \*1

## &lt; Obtain PLA data &gt;

Where is your PLA data stored ?

(If you don't have a PLA, use one of the followings as an example:

data --- a PLA with 5 inputs, 2 product lines and 3 outputs.  
 tt1 --- a PLA 5 inputs, 15 product lines and 10 outputs.  
 tt2 --- a PLA with 5 inputs, 9 product lines and 7 outputs.  
 tt4 --- a PLA with 5 inputs, 10 product lines and 6 outputs.  
 pla40 --- a file containing 40 PLAs. )

Please give the input file name : \*pla40

Please choose one from the following PLAs, where each 5-tuple represents

(PLA-name  
 #-of-inputs  
 #-of-outputs  
 #-of-product-lines  
 #-of-used-crosspoints).

(CO14 14 1 14 210)	(DK27 9 9 20 200)
(ALU1 12 8 19 60)	(DC2 8 7 58 455)
(SQR6 6 12 63 637)	(BCD 26 38 243 4483)
(DK17 10 11 57 631)	(DK48 15 17 42 672)
(ALU2 10 8 87 593)	(RISC 8 31 74 407)
(APLA 10 12 112 1277)	(IN5 24 14 62 752)
(IN8 33 23 54 552)	(IN7 27 10 84 563)
(IN1 16 17 110 2100)	(MLP4 8 8 225 2478)
(INO 15 11 135 1825)	(VG2 25 8 110 914)
(IN2 19 10 137 1527)	(RCKL 32 7 96 1238)
(SYM 9 1 420 4200)	(GARY 15 11 214 2240)
(CHKN 29 7 153 1865)	(MISG 58 23 75 255)
(EXEP 30 63 149 1944)	(IN4 32 20 234 3291)
(MISH 94 43 91 255)	(BCF 30 38 243 4483)
(JBP 36 57 166 1252)	(TIAL 14 8 640 5627)
(BCC 26 45 245 4903)	(X2DN 82 56 112 578)
(BCO 26 11 419 6673)	(BCB 26 39 299 5684)
(BCA 26 46 301 5905)	(ADD6 12 7 1092 9840)
(OPA 17 69 342 2461)	(TI 47 72 241 3171)
(X7DN 66 15 622 5642)	(CPS 24 109 654 7810)

Which one do you want to make testable ? (Give the PLA's name) : \*bcd

Ok. I got it.

Here is your PLA BCD :

The number of inputs = 28

The number of product lines = 243

The number of outputs = 38

The PLA's personality matrix is not available.

### < Specify weights >

So far we have 15 TDMs in our knowledge base, which are distinguished by a set of attributes. Among them the following may be of most interest to you.

	attribute	default value	unit/range	bound
1	TEST-GENERATION	NO	(YES NO)	
2	TEST-STORAGE	2000	BITS	UPPER
3	TEST-APPLICATION-TIME	1	MS	UPPER
4	SELF-TESTING	YES	(YES NO)	
5	CONCURRENT-TESTING	NO	(YES NO)	
6	EXTRA-IO-PINS	4	#-OF-PINS	UPPER
7	AREA-OVERHEAD	15	%	UPPER
8	&FAULT-COVERAGE	99	%	LOWER

For different applications, some of attributes may be more important than others. It is very useful if you can specify the priorities of the attributes. This information can be given by a sequence of positive numbers indicating the relative weights among the attributes. The larger a number, the more important the corresponding attribute.

Here are the current weights:

	attribute	relative weight	normalized weight
1	TEST-GENERATION	5	0.5E-1
2	TEST-STORAGE	10	0.1
3	TEST-APPLICATION-TIME	15	0.1
4	SELF-TESTING	10	0.1
5	CONCURRENT-TESTING	1	0.1E-1
6	EXTRA-IO-PINS	9	0.1
7	AREA-OVERHEAD	20	0.2
8	&FAULT-COVERAGE	19	0.2

If you want to change the weights, please give a sequence of numbers to indicate the relative weight of each attribute in the order of the attribute list in the above table. If you want to use the same weights, please type 'SAME'.

\*same

## &lt; Specify requirements &gt;

To select a TDM that most suits your needs, you should first specify a set of requirements in terms of the above attributes. A requirement for a numeric attribute should be a number. A requirement for a logic attribute should be 'yes' or 'no'.

Do you want to

1. use default requirements,
2. use the requirements defined last time,
3. define new requirements ? \*3

*< Incremental requirement specification is used here.  
The dynamic ramification tree is transparent to the user.  
Attributes are ordered by their weights. >*

What is your requirement for AREA-OVERHEAD ?  
( 'd' for don't care, 'h' for help ) \*20

This requirement has the following implications.

TEST-STORAGE is restricted to (2.5E+9 8832).  
TEST-APPLICATION-TIME is restricted to (201.3 0.7E-3).  
SELF-TESTING is restricted to NO.  
CONCURRENT-TESTING is restricted to NO.  
EXTRA-IO-PINS is restricted to (2 0).  
AREA-OVERHEAD is restricted to (1.4 0.0).  
&FAULT-COVERAGE is restricted to (50.7 100.0).  
Is this OK for you? \*y

What is your requirement for &FAULT-COVERAGE ?  
( 'd' for don't care, 'h' for help ) \*h

Fault coverage is closely related to fault models. If only one fault model FM<sub>1</sub> is used, fault coverage is the probability of a fault in FM<sub>1</sub> being detected. If several fault models FM<sub>1</sub>, FM<sub>2</sub>, ..., FM<sub>j</sub> are considered and their fault coverage is fc<sub>1</sub>, fc<sub>2</sub>, ..., fc<sub>j</sub>, respectively, the total fault coverage is given by the formula

$$FC = \frac{\text{Sum-over-1}(\text{\#-of-faults-in-FM}_i * fc_i)}{\text{Sum-over-1}(\text{\# of faults in FM}_i)}$$

Note that we only take into account the proven fault coverage for each TDM although a TDM may also detect other classes of faults. For some TDMs, there is no theoretical or experimental results dealing with fault coverage. For such cases an estimate of fault coverage must be used. For example, for signature analysis we will assume a coverage of 99% of single stuck-at faults. For simplicity, we have not weighted the fault classes based upon their probability of occurrence. Possible values for &FAULT-COVERAGE are from 50.7 to 100.0. Your requirement may be within or beyond this range. However based on other requirements you have given, the value of &FAULT-COVERAGE has been restricted to (50.7 100.0).

What is your requirement for &FAULT-COVERAGE ?  
 ('d' for don't care, 'h' for help) \*98

This requirement has the following implications.

TEST-STORAGE bounds have changed from (2.5E+9 6832) to  
 (2.5E+9 1399680).

TEST-APPLICATION-TIME bounds have changed from (201.3 0.7E-3) to  
 (201.3 0.6E-1).

EXTRA-IO-PINS bounds have changed from (2 0) to 0.

AREA-OVERHEAD bounds have changed from (1.4 0.0) to 0.0.

&FAULT-COVERAGE bounds have changed from (50.7 100.0) to (99.2 100.0).

Is this OK for you? \*y

What is your requirement for TEST-APPLICATION-TIME ?  
 ('d' for don't care, 'h' for help) \*h

Test application time specifies a lower bound for the time required to complete the entire test process. Suppose (1) there is no overlapping of tests between successive test cycles, (2) each test cycle finishes in one system clock cycle, and (3) all clock cycles are of the same length. Then

test application time = delay per test \* length of the shortest test sequence which contains the entire test set in the right order.

Note that a test sequence may be different from a sequence of patterns in a test set, because some transition sequences may be necessary in order to apply a given set of tests.

Possible values for TEST-APPLICATION-TIME are from 402.6 to 0.7E-3.

Your requirement may be within or beyond this range.

However based on other requirements you have given, the value of TEST-APPLICATION-TIME has been restricted to (0.6E-1 201.3).

What is your requirement for TEST-APPLICATION-TIME ?  
 ('d' for don't care, 'h' for help) \*0.0001

TEST-APPLICATION-TIME=1.0E-4 is incompatible with other requirements.  
 Do you want to reconsider this requirement? \*n

Requirement for TEST-APPLICATION-TIME is confirmed even though it is inconsistent with others.

*< Since the requirement for test-application-time is incompatible with other requirements specified so far, the set of requirements will not be consistent any more. Therefore in the following no consistency checking will be carried out, and some requirements which are implied by the former requirements will be re-asked. >*

What is your requirement for TEST-STORAGE ?  
 ('d' for don't care, 'h' for help) \*5000

What is your requirement for EXTRA-IO-PINS ?  
 ('d' for don't care, 'h' for help) \*1

What is your requirement for TEST-GENERATION ?  
 ('d' for don't care, 'h' for help) \*no

Requirement specification finished.

### < Specification of penalty-credit functions >

The other important factor in making a selection is how tight or loose your requirements are. Most requirements are not strict upper or lower bounds. If a TDM's value exceeds a requirement, it may be accepted with extra credit. If a TDM's value fails to meet a requirement, it may still be acceptable with some penalty. You can specify these concept of extra credit and penalty by means of a PENALTY-CREDIT function, we call it a PC function (PCF).

Do you want to

1. use the PC functions defined last time for all attributes,
2. use the system's default PC functions for all attributes,
3. define new PC functions ? \*3

Now let's define PC functions for every attribute one by one. You will have chance to choose among several methods for defining each PC function.

The current PC function for TEST-GENERATION is :

Penalty-Credit Function for Attribute TEST-GENERATION		
req.,value	PC function	Remark
[NO, NO]	50	ACCEPT
[NO, YES]	-15000	PENALTY
[YES, NO]	80	CREDIT
[YES, YES]	40	ACCEPT

Do you want to use it ? \*yes

No PC function for TEST-STORAGE is stored.

For attribute TEST-STORAGE, do you want

1. to define your own PCF,
2. to use the system's default PCF,
3. to modify the system's standard PCF,
4. no PCF,
5. help ? \*1

To define your own PC function for TEST-STORAGE, you should first divide the possible value range into as many segments as you like, and then specify functions to be used in each segments.



First, define the segments. Remember that the segments must cover the possible value range (2.5E+9 0) and should not overlap each other. When you finish, please type 'n'.

Section 1 = \*[0, 2000]  
 Section 2 = \*[2000, 6000]  
 Section 3 = \*[6000, 30000]  
 Section 4 = \*n

The divided segments are : ((0 2000) (2000 6000) (6000 30000))  
 The possible value range of TEST-STORAGE is from 2.5E+9 to 0.  
 The segments you defined did not cover the entire range.  
 Do you want to

1. try again, or
2. have me to do it for you ? \*2

The modified segments are :

((0 2000) (2000 6000) (6000 30000) (30000 2.5E+9))

Now select a function to be used in each section.

Please choose a function to be used in Section (0 2000)

Currently, the following functions can be used:

1. Constant
2. Linear:  $A + B * \text{distance}(\text{req}, \text{value})$
3. Polynomial:  $A * \text{distance}(\text{req}, \text{value}) ** n$

You can choose one of them. Which one do you prefer ? \*1

CONSTANT = \*1000

Please choose a function to be used in Section (2000 6000)

Currently, the following functions can be used:

1. Constant
2. Linear:  $A + B * \text{distance}(\text{req}, \text{value})$
3. Polynomial:  $A * \text{distance}(\text{req}, \text{value}) ** n$

You can choose one of them. Which one do you prefer ? \*1

CONSTANT = \*50

Please choose a function to be used in Section (6000 30000)

Currently, the following functions can be used:

1. Constant
2. Linear:  $A + B * \text{distance}(\text{req}, \text{value})$
3. Polynomial:  $A * \text{distance}(\text{req}, \text{value}) ** n$

You can choose one of them. Which one do you prefer ? \*2

A = \*-5

SLOPE B = \*0.2

Please choose a function to be used in Section (30000 2.5E+9)  
Currently, the following functions can be used:

1. Constant
2. Linear:  $A + B * \text{distance}(\text{req}, \text{value})$
3. Polynomial:  $A * \text{distance}(\text{req}, \text{value}) ** n$

You can choose one of them. Which one do you prefer ? \*1

CONSTANT = \*-20000

Is this OK? \*y

The current PC function for TEST-APPLICATION-TIME is :

Penalty-Credit Function for Attribute TEST-APPLICATION-TIME		
value range	PC function	Remark
[1.5, 402.6]	-20000	UNACCEPTABLE
[1, 1.5]	$4995 // (0.5 * \text{REQ}) * (\text{DISTANCE ATT REQ VALUE})$	PENALTY
[1, 1]	50	ACCEPT
[0.5, 1]	$50 + (50 // (0.5 * \text{REQ})) * (\text{DISTANCE ATT REQ VALUE})$	CREDIT
[0.7E-3, 0.5]	100	CREDIT

Do you want to use it ? \*n

For attribute TEST-APPLICATION-TIME, do you want

1. to define your own PCF,
2. to use the system's default PCF,
3. to modify the system's standard PCF,
4. no PCF,
5. help ? \*2

Penalty-Credit Function for Attribute TEST-APPLICATION-TIME		
value range	PC function	Remark
[0.1E-3, 402.6]	-20000	UNACCEPTABLE
[1.0E-4, 0.1E-3]	$4995 // (0.5 * \text{REQ}) * (\text{DISTANCE ATT REQ VALUE})$	PENALTY
[1.0E-4, 1.0E-4]	50	ACCEPT
[0.5E-4, 1.0E-4]	$50 + (50 // (0.5 * \text{REQ})) * (\text{DISTANCE ATT REQ VALUE})$	CREDIT

Is this OK ? \*y

The current PC function for SELF-TESTING is :

Penalty-Credit Function for Attribute SELF-TESTING		
req.,value	PC function	Remark
[YES, NO]	-5000	PENALTY
[NO, YES]	100	CREDIT
[NO, NO]	50	ACCEPT
[YES, YES]	50	ACCEPT

Do you want to use it ? \*y

The current PC function for CONCURRENT-TESTING is :

Penalty-Credit Function for Attribute CONCURRENT-TESTING

req.,value	PC function	Remark
[YES, NO]	-5000	PENALTY
[NO, YES]	100	CREDIT
[NO, NO]	50	ACCEPT
[YES, YES]	50	ACCEPT

Do you want to use it ? \*y

The current PC function for EXTRA-IO-PINS is :

Penalty-Credit Function for Attribute EXTRA-IO-PINS

value range	PC function	Remark
[6.0, 11]	-20000	UNACCEPTABLE
[4, 6.0]	$4995 // (0.5 * REQ) * (DISTANCE \text{ ATT REQ VALUE})$	PENALTY
[4, 4]	50	ACCEPT
[2.0, 4]	$50 + (50 // (0.5 * REQ)) * (DISTANCE \text{ ATT REQ VALUE})$	CREDIT
[0, 2.0]	100	CREDIT

Do you want to use it ? \*n

For attribute EXTRA-IO-PINS, do you want

1. to define your own PCF,
2. to use the system's default PCF,
3. to modify the system's standard PCF,
4. no PCF,
5. help ? \*2

Penalty-Credit Function for Attribute EXTRA-IO-PINS

value range	PC function	Remark
[1.5, 11]	-20000	UNACCEPTABLE
[1, 1.5]	$4995 // (0.5 * REQ) * (DISTANCE \text{ ATT REQ VALUE})$	PENALTY
[1, 1]	50	ACCEPT
[0.5, 1]	$50 + (50 // (0.5 * REQ)) * (DISTANCE \text{ ATT REQ VALUE})$	CREDIT
[0, 0.5]	100	CREDIT

Is this OK ? \*y

The current PC function for AREA-OVERHEAD is :

Penalty-Credit Function for Attribute AREA-OVERHEAD

value range	PC function	Remark
[37.5, 184.1]	-20000	UNACCEPTABLE
[25, 37.5]	$4995 // (0.5 * REQ) * (DISTANCE \text{ ATT REQ VALUE})$	PENALTY
[25, 25]	50	ACCEPT
[12.5, 25]	$50 + (50 // (0.5 * REQ)) * (DISTANCE \text{ ATT REQ VALUE})$	CREDIT
[0.0, 12.5]	100	CREDIT

Do you want to use it ? \*n

For attribute AREA-OVERHEAD, do you want

1. to define your own PCF,
2. to use the system's default PCF,
3. to modify the system's standard PCF,
4. no PCF,
5. help ? \*2

Penalty-Credit Function for Attribute AREA-OVERHEAD

value range	PC function	Remark
[30.0, 184.1]	-20000	UNACCEPTABLE
[20, 30.0]	$4995 // (0.5 * REQ) * (DISTANCE ATT REQ VALUE)$	PENALTY
[20, 20]	50	ACCEPT
[10.0, 20]	$50 + (50 // (0.5 * REQ)) * (DISTANCE ATT REQ VALUE)$	CREDIT
[0.0, 10.0]	100	CREDIT

Is this OK ? \*y

The current PC function for &FAULT-COVERAGE is :

Penalty-Credit Function for Attribute &FAULT-COVERAGE

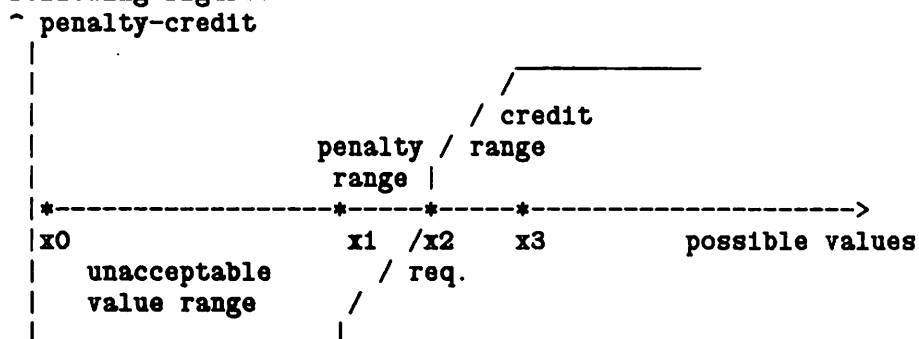
value range	PC function	Remark
[49.0, 98]	$4995 // (0.5 * REQ) * (DISTANCE ATT REQ VALUE)$	PENALTY
[98, 98]	50	ACCEPT
[98, 147.0]	$50 + (50 // (0.5 * REQ)) * (DISTANCE ATT REQ VALUE)$	CREDIT

Is this OK ? \*n

For attribute &FAULT-COVERAGE, do you want

1. to define your own PCF,
2. to use the system's default PCF,
3. to modify the system's standard PCF,
4. no PCF,
5. help ? \*3

For a numeric attribute ATT, the general form of the PCF is as shown in the following figure:



where, the horizontal axis represents the range of possible values for attribute ATT, and the vertical axis represents the amount of penalty or credit.

- [x0, x1) is the unacceptable value range;
- [x1, x2) is the penalty range, and x1 is called the unacceptable point;
- [x2, x3) is the credit range;
- [x3, - ) is the saturation range where credit fixes at the same level, and x3 is called the saturation point.

To modify a standard PCF for &FAULT-COVERAGE that suits your needs, you must specify the value of x1, x2, and x3, and choose functions to be used in each segment.

For &FAULT-COVERAGE and your PLA, the possible value range is from 50.7 to 100.0, and your current requirement is 98.

What is the unacceptable point ? (n for none) \*80

What is the saturation point ? (n for none) \*100

Currently, the following functions can be used in CREDIT segment:

1. Constant
2. Linear:  $A + B * \text{distance}(\text{req}, \text{value})$
3. Polynomial:  $A * \text{distance}(\text{req}, \text{value}) ** n$

You can choose one of them. Which one do you prefer ? \*2

A = \*50

SLOPE B = \*25

Currently, the following functions can be used in ACCEPT segment:

1. Constant
2. Linear:  $A + B * \text{distance}(\text{req}, \text{value})$
3. Polynomial:  $A * \text{distance}(\text{req}, \text{value}) ** n$

You can choose one of them. Which one do you prefer ? \*1

CONSTANT = \*50

Currently, the following functions can be used in PENALTY segment:

1. Constant
2. Linear:  $A + B * \text{distance}(\text{req}, \text{value})$
3. Polynomial:  $A * \text{distance}(\text{req}, \text{value}) ** n$

You can choose one of them. Which one do you prefer ? \*3

A = \*-12.5

n (must be an integer) = \*2

Currently, the following functions can be used in UNACCEPTABLE segment:

1. Constant
2. Linear:  $A + B * \text{distance}(\text{req}, \text{value})$
3. Polynomial:  $A * \text{distance}(\text{req}, \text{value}) ** n$

You can choose one of them. Which one do you prefer ? \*1

CONSTANT = \*-20000

Penalty-Credit Function for Attribute &FAULT-COVERAGE

value range	PC function	Remark
[50.7, 80]	-20000	UNACCEPTABLE
[80, 98]	$(-12.5 * ((\text{DISTANCE ATT REQ VALUE}) \text{ EXP } 2))$	PENALTY
[98, 98]	50	ACCEPT
[98, 100]	$(50 + (25 * (\text{DISTANCE ATT REQ VALUE})))$	CREDIT

Is this OK? \*y

Great! Now we've finished the definition of all PC functions.

Do you want to see them again ?\*n

Now you have defined your requirements, weights and penalty-credit functions. Do you want to

1. change some initial specifications,
2. start searching for a solution,
3. quit ? \*2

## &lt; Search for a solution &gt;

Please wait while I try to find a solution for you.

No solution found.

## &lt; First class critical conflict &gt;

There is a first class critical conflict, that is, one or more of your requirements are higher than the best possible value among all the TDMS can offer. The conflicts are shown below :

	YOUR-REQUIREMENT	BEST-POSSIBLE
TEST-APPLICATION-TIME	1.0E-4	0.7E-3

If you can change the above requirements to under the best attainable values indicated, there might be a solution, although I cannot guarantee one at this point. If some of these requirements are unchangeable, then there will be no solution.

Is it possible to change the requirements in conflict ?\*y

Good.

Here are your requirements:

	attribute	current value	unit/range
1	TEST-GENERATION	NO	(YES NO)
2	TEST-STORAGE	5000	BITS
3	TEST-APPLICATION-TIME	1.0E-4	MS
4	SELF-TESTING	NO	(YES NO)
5	CONCURRENT-TESTING	NO	(YES NO)
6	EXTRA-IO-PINS	2	#-OF-PINS
7	AREA-OVERHEAD	20	%
8	&FAULT-COVERAGE	98	%

Which one are you going to change ? (Give a number.

If none, type 'n'. If you need any advice, type 'y'). \*3

TEST-APPLICATION-TIME can be changed to \*1

Here are your requirements:

	attribute	current value	unit/range
1	TEST-GENERATION	NO	(YES NO)
2	TEST-STORAGE	5000	BITS
3	TEST-APPLICATION-TIME	1	MS
4	SELF-TESTING	NO	(YES NO)
5	CONCURRENT-TESTING	NO	(YES NO)
6	EXTRA-IO-PINS	2	#-OF-PINS
7	AREA-OVERHEAD	20	%
8	&FAULT-COVERAGE	98	%

Which one are you going to change ? (Give a number.

If none, type 'n'. If you need any advice, type 'y'.) \*n

You have changed the requirements on TEST-APPLICATION-TIME.  
Do you want to change the PCFs associated with these attributes  
as well? \*y

Do you want to use the default PCFs for all of them ? \*y

value range	PC function	Remark
[1.5, 402.6]	-20000	UNACCEPTABLE
[1, 1.5]	$4995 // (0.5 * REQ) * (DISTANCE ATT REQ VALUE)$	PENALTY
[1, 1]	50	ACCEPT
[0.5, 1]	$50 + (50 // (0.5 * REQ)) * (DISTANCE ATT REQ VALUE)$	CREDIT
[0.7E-3, 0.5]	100	CREDIT

Please wait while I try to find a solution for you.

No solution found.

< Third class critical conflict >

There is a third class critical conflict, that is, every TDM has at least one attribute value which is unacceptable to you. Some major changes have to be made in order to obtain a solution. The following table shows possible ways in which the requirements MUST be changed.

	Current requirement	Have to change to
FAULT-COVERAGE	98	50.7
TEST-GENERATION	NO	YES
TEST-STORAGE	5000	1399680
TEST-GENERATION	NO	YES
AREA-OVERHEAD	20	31.7
TEST-APPLICATION-TIME	1	201.3
TEST-STORAGE	5000	2.5E+9
EXTRA-IO-PINS	1	4

The problem may be caused by improper definitions of the PC functions or requirements.

I suggest you change the requirements on  
attribute from to at least  
AREA-OVERHEAD 20 27.2  
EXTRA-IO-PINS 1 4  
because in this way you can make minimal changes and obtain a  
solution. Do you want to modify the requirements and the PC  
functions? \*y

What do you want to change ?

1. Requirements.
2. Weights.
3. Penalty/Credit functions. \*1

Here are your requirements:

	attribute	current value	unit/range
1	TEST-GENERATION	NO	(YES NO)
2	TEST-STORAGE	5000	BITS
3	TEST-APPLICATION-TIME	1	MS
4	SELF-TESTING	NO	(YES NO)
5	CONCURRENT-TESTING	NO	(YES NO)
6	EXTRA-IO-PINS	1	#-OF-PINS
7	AREA-OVERHEAD	20	%
8	&FAULT-COVERAGE	98	%

Which one are you going to change ? (Give a number.  
If none, type 'n'. If you need any advice, type 'y'.) \*8

EXTRA-IO-PINS can be changed to \*2

Here are your requirements:

	attribute	current value	unit/range
1	TEST-GENERATION	NO	(YES NO)
2	TEST-STORAGE	5000	BITS
3	TEST-APPLICATION-TIME	1	MS
4	SELF-TESTING	NO	(YES NO)
5	CONCURRENT-TESTING	NO	(YES NO)
6	EXTRA-IO-PINS	2	#-OF-PINS
7	AREA-OVERHEAD	20	%
8	&FAULT-COVERAGE	98	%

Which one are you going to change ? (Give a number.  
If none, type 'n'. If you need any advice, type 'y'.) \*n

You have changed the requirements on EXTRA-IO-PINS. Do you want  
to change the PCFs associated with these attributes as well? \*y

Do you want to use the default PCFs for all of them ? \*y

value range	PC function	Remark
[3.0, 11]	-20000	UNACCEPTABLE
[2, 3.0]	4995//((0.5*REQ))*(DISTANCE ATT REQ VALUE)	PENALTY
[2, 2]	50	ACCEPT
[1.0, 2]	50+(50//((0.5*REQ))*(DISTANCE ATT REQ VALUE)	CREDIT
[0, 1.0]	100	CREDIT

Do you want to change other things as well ? \*n

Please wait while I try to find a solution for you.

No solution found.



## &lt; Multiple conflict &gt;

There are multiple conflicts, i.e., every TDM fails to satisfy at least two requirements. In this situation, the only way for you to get a solution is to make tradeoffs between different attributes. The following is a table showing all of the conflicts, where '\*' indicates that a value is worse than the corresponding requirement.

	REQUEST	BEST-FIT	SECOND-BEST	THIRD-BEST
TEST-GENERATION	NO	NO	YES*	YES*
TEST-STORAGE	5000	0	1399680*	2173691*
TEST-APPLICATION-TIME	1	0.6	0.6E-1	0.2E-1
SELF-TESTING	NO	YES	NO	NO
CONCURRENT-TESTING	NO	NO	NO	NO
EXTRA-IO-PINS	2	4*	0	3*
AREA-OVERHEAD	20	27.2*	0.0	22.7*
&FAULT-COVERAGE	98	100.0	99.2	99.3

What you could do now is either relaxing some requirements in order to get a solution, or finishing with no solution.

Please select one of the following:

1. I'd like to make some changes.
2. I'd like to see other choices.
3. I'd like to pick up one solution.
4. Please choose the best one for me.
5. Explain the possible choices.
6. I'd like to check the system state.
7. I don't like any of them.
8. Go back to previous choices.
9. Trace design history.
10. Exit.

\*1

## &lt; Make changes &gt;

What do you want to change ?

1. Requirements.
2. Weights.
3. Penalty/Credit functions. \*1

Here are your requirements:

	attribute	current value	unit/range
1	TEST-GENERATION	NO	(YES NO)
2	TEST-STORAGE	5000	BITS
3	TEST-APPLICATION-TIME	1	MS
4	SELF-TESTING	NO	(YES NO)
5	CONCURRENT-TESTING	NO	(YES NO)
6	EXTRA-IO-PINS	2	#-OF-PINS
7	AREA-OVERHEAD	20	%
8	&FAULT-COVERAGE	98	%

Which one are you going to change ? (Give a number.

If none, type 'n'. If you need any advice, type 'y'.) \*6

EXTRA-IO-PINS can be changed to \*4

Here are your requirements:

	attribute	current value	unit/range
1	TEST-GENERATION	NO	(YES NO)
2	TEST-STORAGE	5000	BITS
3	TEST-APPLICATION-TIME	1	MS
4	SELF-TESTING	NO	(YES NO)
5	CONCURRENT-TESTING	NO	(YES NO)
6	EXTRA-IO-PINS	4	#-OF-PINS
7	AREA-OVERHEAD	20	%
8	&FAULT-COVERAGE	98	%

Which one are you going to change ? (Give a number.

If none, type 'n'. If you need any advice, type 'y'.) \*n

You have changed the requirements on EXTRA-IO-PINS. Do you want to change the PCFs associated with these attributes as well? \*y

Do you want to use the default PCFs for all of them ? \*y

value range	PC function	Remark
[8.0, 11]	-20000	UNACCEPTABLE
[4, 8.0]	$4995 // (0.5 * REQ) * (DISTANCE ATT REQ VALUE)$	PENALTY
[4, 4]	50	ACCEPT
[2.0, 4]	$50 + (50 // (0.5 * REQ)) * (DISTANCE ATT REQ VALUE)$	CREDIT
[0, 2.0]	100	CREDIT

Do you want to change other things as well ? \*n

Please wait while I try to find a solution for you.

No solution found.

### < Second class critical conflict >

There is a second class critical conflict, that is, there are some requirements which cannot be satisfied by any of the TDMs which are close to satisfying your current requirements. These critical requirements are listed below:

CRITICAL ATTRIBUTE	YOUR-REQUIREMENT	POSSIBLE-VALUE
AREA-OVERHEAD	20	23.7

However, there exist some TDMs which do satisfy these requirements, but fail to satisfy some other requirements, as you can see from the following table. Note that for each TDM, at least one attribute value is far from the required value.

	request	weight	EXH	SMITH	TSBF
TEST-APPLICATION-TIME	1	0.1	201.3	0.6E-1	0.7E-3
TEST-STORAGE	5000	0.1	2.5E+9	1399680	6832
TEST-GENERATION	NO	0.5E-1	NO	YES	YES
&FAULT-COVERAGE	98	0.2	100.0	99.2	50.7
AREA-OVERHEAD	20	0.2	0	0.0	1.4
TOTAL SCORE			-5555	-3016.8	-5098.4

You have two choices now. First, I suggest you relax the requirements on AREA-OVERHEAD from 20 to 23.7, such that a satisfactory solution may be found. If you don't do that, you have to relax some of the requirements in the following table, which means large changes.

	Current requirement	Have to change to
&FAULT-COVERAGE	98	50.7
TEST-GENERATION	NO	YES
TEST-STORAGE	5000	1399680
TEST-GENERATION	NO	YES
TEST-APPLICATION-TIME	1	201.3
TEST-STORAGE	5000	2.5E+9

Otherwise there would be no solution.

Do you want to change some requirements ? \*y

Here are your requirements:

	attribute	current value	unit/range
1	TEST-GENERATION	NO	(YES NO)
2	TEST-STORAGE	5000	BITS
3	TEST-APPLICATION-TIME	1	MS
4	SELF-TESTING	NO	(YES NO)
5	CONCURRENT-TESTING	NO	(YES NO)
6	EXTRA-IO-PINS	4	#-OF-PINS
7	AREA-OVERHEAD	20	%
8	&FAULT-COVERAGE	98	%

Which one are you going to change ? (Give a number.

If none, type 'n'. If you need any advice, type 'y'.) \*7

AREA-OVERHEAD can be changed to \*25

Here are your requirements:

	attribute	current value	unit/range
1	TEST-GENERATION	NO	(YES NO)
2	TEST-STORAGE	5000	BITS
3	TEST-APPLICATION-TIME	1	MS
4	SELF-TESTING	NO	(YES NO)
5	CONCURRENT-TESTING	NO	(YES NO)
6	EXTRA-IO-PINS	4	#-OF-PINS
7	AREA-OVERHEAD	25	%
8	&FAULT-COVERAGE	98	%

Which one are you going to change ? (Give a number.

If none, type 'n'. If you need any advice, type 'y'.) \*n

You have changed the requirements on AREA-OVERHEAD. Do you want to change the PCFs associated with these attributes as well? \*y

Do you want to use the default PCFs for all of them ? \*y

value range	PC function	Remark
[37.5, 184.1]	-20000	UNACCEPTABLE
[25, 37.5]	$4995 // (0.5 * REQ) * (DISTANCE \text{ ATT REQ VALUE})$	PENALTY
[25, 25]	50	ACCEPT
[12.5, 25]	$50 + (50 // (0.5 * REQ)) * (DISTANCE \text{ ATT REQ VALUE})$	CREDIT
[0.0, 12.5]	100	CREDIT

Please wait while I try to find a solution for you.

No solution found.

### < Single conflict >

There is a single conflict, i.e., there exist TDMs which satisfy all but one of your requirements as shown in the following table.

TDM	failed goal	request	actual	weight	Score
BIST2	AREA-OVERHEAD	25	27.2	0.2	-38.9
FIST	AREA-OVERHEAD	25	31.7	0.2	-436.1
UT2	TEST-STORAGE	5000	504426	0.1	-2182.5

Now you are very close to a good solution. You can easily get one by changing one of the requirements to a sufficient extent as indicated in the above table.

Please select one of the following:

1. I'd like to make some changes.
2. I'd like to see other choices.
3. I'd like to pick up one solution.
4. Please choose the best one for me.
5. Explain the possible choices.
6. I'd like to check the system state.
7. I don't like any of them.
8. Go back to previous choices.
9. Trace design history.
10. Exit.

\*5

### < Explanation of possible choices >

Currently, the top 3 choices are as follows:

	NAME
BIST2 :	BUILT-IN-SELF-TESTING-PLA
FIST :	FUNCTION-INDEPENDENT-SELF-TESTING-PLA
UT2 :	PLA-DESIGN-FOR-UNIVERSAL-TESTABILITY

The detail performance of these TDMs are given below :

TDM = BIST2 Total score = -38.9

attribute	req.	value	weight	pc	w * pc
TEST-GENERATION	NO	NO	0.5E-1	50	2.8
TEST-STORAGE	5000	0	0.1	1000	112.3
TEST-APPLICATION-TIME	1	0.6	0.1	83.7	14.1
SELF-TESTING	NO	YES	0.1	100	11.2
CONCURRENT-TESTING	NO	NO	0.1E-1	50	0.5
EXTRA-IO-PINS	4	4	0.1	50.0	5.0
AREA-OVERHEAD	25	27.2	0.2	-918.4	-206.3
&FAULT-COVERAGE	98	100.0	0.2	100.0	21.3

TDM = FIST Total score = -436.1

attribute	req.	value	weight	pc	w * pc
TEST-GENERATION	NO	NO	0.5E-1	50	2.8
TEST-STORAGE	5000	38	0.1	1000	112.3
TEST-APPLICATION-TIME	1	0.8E-1	0.1	100	16.8
SELF-TESTING	NO	YES	0.1	100	11.2
CONCURRENT-TESTING	NO	NO	0.1E-1	50	0.5
EXTRA-IO-PINS	4	2	0.1	100	10.1
AREA-OVERHEAD	25	31.7	0.2	-2697.3	-606.1
&FAULT-COVERAGE	98	99.0	0.2	75.0	16.0

TDM = UT2 Total score = -2182.5

attribute	req.	value	weight	pc	w * pc
TEST-GENERATION	NO	NO	0.5E-1	50	2.8
TEST-STORAGE	5000	504426	0.1	-20000	-2247.1
TEST-APPLICATION-TIME	1	0.3E-1	0.1	100	16.8
SELF-TESTING	NO	NO	0.1	50	5.6
CONCURRENT-TESTING	NO	NO	0.1E-1	50	0.5
EXTRA-IO-PINS	4	4	0.1	50.0	5.0
AREA-OVERHEAD	25	23.7	0.2	55.1	12.4
&FAULT-COVERAGE	98	100.0	0.2	100.0	21.3

Now here is a summary :

	request	weight	BIST2	FIST	UT2
TEST-GENERATION	NO	0.5E-1	NO	NO	NO
TEST-STORAGE	5000	0.1	0	38	504426
TEST-APPLICATION-TIME	1	0.1	0.6	0.8E-1	0.3E-1
SELF-TESTING	NO	0.1	YES	YES	NO
CONCURRENT-TESTING	NO	0.1E-1	NO	NO	NO
EXTRA-IO-PINS	4	0.1	4	2	4
AREA-OVERHEAD	25	0.2	27.2	31.7	23.7
&FAULT-COVERAGE	98	0.2	100.0	99.0	100.0
TOTAL SCORE			-38.9	-436.1	-2182.5

Which one are you interested in ?

(Give a TDM name. Type 'n' for none, 'h' for help) : \*fist

#### FIST FUNCTION-INDEPENDENT-SELF-TESTING-PLA

This is a simple signature test design for PLAs. It attempts to reduce the number of test patterns and additional hardware, and also takes into account the impact on the PLA layout.

The input decoder is modified into an input register and test pattern generator. During the test mode, it shifts a single '1' across the bit lines. The selection of product lines is done using an  $m/2$  bit shift register SHR and an additional selector SEL. The SHR selects a pair of adjacent product lines at a time, so its length is only half of the number of product terms. The SEL will in turn connect product line 1, 3, 5, ... or 2, 4, 6, ... to ground, thus resulting in a unique selection of each product line. Splitting the product line selector into SHR and SEL allows the added circuit to physically fit into the narrow pitch of the PLA arrays, substantially reducing the area overhead.

During testing, every crosspoint in the AND array is addressed using the two shift registers. The results are analyzed in a multiple input signature register MSR which acts as a conventional latch register during normal operation. The complete set of attribute-value pairs is shown below.

	FIST
FAULT-MASKING	NO
CONCURRENT-TESTING	NO
SELF-TESTING	YES
TEST-GENERATION	NO
&FAULT-COVERAGE	99.0
EXTRA-IO-PINS	2
EXTRA-DELAY	3
DELAY-PER-TEST	7
#-OF-TESTS	12636
EXTRA-BIT-LINE	0
TEST-APPLICATION-TIME	0.8E-1
TEST-STORAGE	38
AREA-OVERHEAD	31.7
EXTRA-TRANSISTORS	2074
EXTRA-BIT-LINE	0
EXTRA-PRODUCT-LINE	0
EXTRA-OUTPUT-LINE	0
DESIGN-COST	2
LAYOUT-DIFFICULTY	2

FIST has no variation stored.

Do you want to see more ? \*n

Please select one of the following:

1. I'd like to make some changes.
2. I'd like to see other choices.
3. I'd like to pick up one solution.
4. Please choose the best one for me.
5. Explain the possible choices.
6. I'd like to check the system state.
7. I don't like any of them.
8. Go back to previous choices.
9. Trace design history.
10. Exit.

\*6

## &lt; Checking system state &gt;

The following system information can be checked.

SCORES gives scores and rankings of the TDMs.  
 HISTORY shows the complete design history.  
 STATE gives values of the global variables  
 REQS shows the current requirements.  
 FAIL shows which TDM failed to satisfy which requirements.  
 OPTIMAL shows the best possible value for each attribute.  
 WEIGHT shows the current weights.  
 UNIT gives the unit or range for each attribute.  
 VALUE displays selected attribute values for all TDMs.  
 PCF shows the current penalty-credit functions.

What do you want to see? (Type 'n' to back to the main menu.)

\*scores

The following TDM list is sorted by TDM's total scores:

TDM	total score	lose score
BIST2	-38.9	-968.4
FIST	-436.1	-2747.3
UT2	-2182.5	-20050
SMITH	-3016.8	-35100
TLO	-3028.3	-35100
SIG	-3845.5	-22850.6
MFT	-4214.2	-40100.0
AUTO	-4323.1	-20050.0
BILBO	-4329.4	-20050.0
BIT	-4581.7	-22597.5
TSBF	-5098.4	-35521.3
UTS	-5100.3	-26182.3
EXH	-5555.0	-40100.0
FIT	-6536.5	-40714.6
CONC	-7797.2	-57697.5

Ranking by the lose scores: (BIST2 FIST BILBO AUTO UT2 BIT SIG UTS  
 SMITH SMITH TSBF EXH MFT FIT CONC)

Do you want to see more ? \*y

The following system information can be checked.

SCORES gives scores and rankings of the TDMs.  
 HISTORY shows the complete design history.  
 STATE gives values of the global variables  
 REQS shows the current requirements.  
 FAIL shows which TDM failed to satisfy which requirements.  
 OPTIMAL shows the best possible value for each attribute.  
 WEIGHT shows the current weights.  
 UNIT gives the unit or range for each attribute.  
 VALUE displays selected attribute values for all TDMs.  
 PCF shows the current penalty-credit functions.

What do you want to see ? (Type 'n' to back to the main menu.) \*reqs

	attribute	current value	unit/range
1	TEST-GENERATION	NO	(YES NO)
2	TEST-STORAGE	5000	BITS
3	TEST-APPLICATION-TIME	1	MS
4	SELF-TESTING	NO	(YES NO)
5	CONCURRENT-TESTING	NO	(YES NO)
6	EXTRA-IO-PINS	4	#-OF-PINS
7	AREA-OVERHEAD	25	%
8	&FAULT-COVERAGE	98	%

Do you want to see more ? \*y

The following system information can be checked.

SCORES gives scores and rankings of the TDMs.  
 HISTORY shows the complete design history.  
 STATE gives values of the global variables  
 REQS shows the current requirements.  
 FAIL shows which TDM failed to satisfy which requirements.  
 OPTIMAL shows the best possible value for each attribute.  
 WEIGHT shows the current weights.  
 UNIT gives the unit or range for each attribute.  
 VALUE displays selected attribute values for all TDMs.  
 PCF shows the current penalty-credit functions.

What do you want to see ? (Type 'n' to back to the main menu.) \*fail

BIST2 (AREA-OVERHEAD)  
 BIT (AREA-OVERHEAD EXTRA-IO-PINS)  
 BILBO (AREA-OVERHEAD)  
 CONC (AREA-OVERHEAD EXTRA-IO-PINS TEST-STORAGE TEST-GENERATION)  
 EXH (TEST-APPLICATION-TIME TEST-STORAGE)  
 FIST (AREA-OVERHEAD)  
 FIT (AREA-OVERHEAD EXTRA-IO-PINS TEST-STORAGE)  
 MFT (EXTRA-IO-PINS TEST-STORAGE)  
 SIG (&FAULT-COVERAGE AREA-OVERHEAD TEST-APPLICATION-TIME)  
 AUTO (AREA-OVERHEAD)  
 SMITH (TEST-STORAGE TEST-GENERATION)  
 TLO (TEST-STORAGE TEST-GENERATION)  
 TSBF (&FAULT-COVERAGE TEST-STORAGE TEST-GENERATION)  
 UT2 (TEST-STORAGE)  
 UTS (AREA-OVERHEAD EXTRA-IO-PINS TEST-STORAGE)

Do you want to see more ? \*y

The following system information can be checked.



SCORES gives scores and rankings of the TDMs.  
 HISTORY shows the complete design history.  
 STATE gives values of the global variables  
 REQS shows the current requirements.  
 FAIL shows which TDM failed to satisfy which requirements.  
 OPTIMAL shows the best possible value for each attribute.  
 WEIGHT shows the current weights.  
 UNIT gives the unit or range for each attribute.  
 VALUE displays selected attribute values for all TDMs.  
 PCF shows the current penalty-credit functions.

What do you want to see? (Type 'n' to back to the main menu.) \*optimal

	attribute	current value
1	TEST-GENERATION	NO
2	TEST-STORAGE	0
3	TEST-APPLICATION-TIME	0.7E-3
4	SELF-TESTING	YES
5	CONCURRENT-TESTING	YES
6	EXTRA-IO-PINS	0
7	AREA-OVERHEAD	0.0
8	&FAULT-COVERAGE	100.0

Do you want to see more ? \*y

The following system information can be checked.

SCORES gives scores and rankings of the TDMs.  
 HISTORY shows the complete design history.  
 STATE gives values of the global variables  
 REQS shows the current requirements.  
 FAIL shows which TDM failed to satisfy which requirements.  
 OPTIMAL shows the best possible value for each attribute.  
 WEIGHT shows the current weights.  
 UNIT gives the unit or range for each attribute.  
 VALUE displays selected attribute values for all TDMs.  
 PCF shows the current penalty-credit functions.

What do you want to see ? (Type 'n' to back to the main menu.) \*value

Choose one attribute from the followings. I will display all TDMs' values for that attribute for you.

1	TEST-GENERATION
2	TEST-STORAGE
3	TEST-APPLICATION-TIME
4	SELF-TESTING
5	CONCURRENT-TESTING
6	EXTRA-IO-PINS
7	AREA-OVERHEAD
8	&FAULT-COVERAGE

Which attribute do you want to see ? \*7

## AREA-OVERHEAD

BIST2 :	27.2
BIT :	57.8
BILBO :	105.0
CONC :	184.1
EXH :	0
FIST :	31.7
FIT :	38.3
MFT :	24.3
SIG :	31.7
AUTO :	42.6
SMITH :	0.0
TLO :	22.7
TSBF :	1.4
UT2 :	23.7
UTS :	37.5

Do you want to see more attribute values ? \*y

Choose one attribute from the followings. I will display all TDMs' values for that attribute for you.

- |   |                       |
|---|-----------------------|
| 1 | TEST-GENERATION       |
| 2 | TEST-STORAGE          |
| 3 | TEST-APPLICATION-TIME |
| 4 | SELF-TESTING          |
| 5 | CONCURRENT-TESTING    |
| 6 | EXTRA-IO-PINS         |
| 7 | AREA-OVERHEAD         |
| 8 | &FAULT-COVERAGE       |

Which attribute do you want to see ? \*3

## TEST-APPLICATION-TIME

BIST2 :	0.6
BIT :	0.7E-1
BILBO :	0.5E-2
CONC :	0.5E-1
EXH :	201.3
FIST :	0.8E-1
FIT :	0.1E-1
MFT :	0.8E-1
SIG :	402.6
AUTO :	0.4E-1
SMITH :	0.6E-1
TLO :	0.2E-1
TSBF :	0.7E-3
UT2 :	0.3E-1
UTS :	0.3

Do you want to see more attribute values ? \*n

Do you want to see more ? \*n

Please select one of the following:

1. I'd like to make some changes.
2. I'd like to see other choices.
3. I'd like to pick up one solution.
4. Please choose the best one for me.
5. Explain the possible choices.
6. I'd like to check the system state.
7. I don't like any of them.
8. Go back to previous choices.
9. Trace design history.
10. Exit.

\*2

< See other TDM choices >

	request	weight	SMITH	TLO	SIG
TEST-GENERATION	NO	0.5E-1	YES	YES	NO
TEST-STORAGE	5000	0.1	1399680	2173691	104
TEST-APPLICATION-TIME	1	0.1	0.6E-1	0.2E-1	402.6
SELF-TESTING	NO	0.1	NO	NO	YES
CONCURRENT-TESTING	NO	0.1E-1	NO	NO	NO
EXTRA-IO-PINS	4	0.1	0	3	4
AREA-OVERHEAD	25	0.2	0.0	22.7	31.7
&FAULT-COVERAGE	98	0.2	99.2	99.3	97.3
TOTAL SCORE			-3016.8	-3028.3	-3845.5

Please select one of the following:

1. I'd like to make some changes.
2. I'd like to see other choices.
3. I'd like to pick up one solution.
4. Please choose the best one for me.
5. Explain the possible choices.
6. I'd like to check the system state.
7. I don't like any of them.
8. Go back to previous choices.
9. Trace design history.
10. Exit.

\*8

< Go back to previous choices >

	request	weight	BIST2	FIST	UT2
TEST-GENERATION	NO	0.5E-1	NO	NO	NO
TEST-STORAGE	5000	0.1	0	38	504426
TEST-APPLICATION-TIME	1	0.1	0.6	0.8E-1	0.3E-1
SELF-TESTING	NO	0.1	YES	YES	NO
CONCURRENT-TESTING	NO	0.1E-1	NO	NO	NO
EXTRA-IO-PINS	4	0.1	4	2	4
AREA-OVERHEAD	25	0.2	27.2	31.7	23.7
&FAULT-COVERAGE	98	0.2	100.0	99.0	100.0
TOTAL SCORE			-38.9	-436.1	-2182.5

Please select one of the following:

1. I'd like to make some changes.
2. I'd like to see other choices.
3. I'd like to pick up one solution.
4. Please choose the best one for me.
5. Explain the possible choices.
6. I'd like to check the system state.
7. I don't like any of them.
8. Go back to previous choices.
9. Trace design history.
10. Exit.

\*4

< System selects the best TDM based on score functions >

I would like to suggest BIST2 because it is optimal in general as you can see from the following table:

	request	weight	BIST2	FIST	UT2
TEST-GENERATION	NO	0.5E-1	NO	NO	NO
TEST-STORAGE	5000	0.1	0	38	504426
TEST-APPLICATION-TIME	1	0.1	0.6	0.8E-1	0.3E-1
SELF-TESTING	NO	0.1	YES	YES	NO
CONCURRENT-TESTING	NO	0.1E-1	NO	NO	NO
EXTRA-IO-PINS	4	0.1	4	2	4
AREA-OVERHEAD	25	0.2	27.2	31.7	23.7
&FAULT-COVERAGE	98	0.2	100.0	99.0	100.0
TOTAL SCORE			-38.9	-436.1	-2182.5

Do you like it ? \*n

< Reason analysis for confirmation failures >

Why don't you like it ? (choose numbers from the following)

	BIST2	REQUEST
1 TEST-GENERATION	NO	NO
2 TEST-STORAGE	0	5000
3 TEST-APPLICATION-TIME	0.6	1
4 SELF-TESTING	YES	NO
5 CONCURRENT-TESTING	NO	NO
6 EXTRA-IO-PINS	4	4
7 AREA-OVERHEAD	27.2	25
8 &FAULT-COVERAGE	100.0	98
9 All of the above		
10 Forget about this TDM. Go back to the main menu.		

\*7

Do you want to keep BIST2 in further consideration ? \*y

There exist TDMs which are better than BIST2 in all of those aspects you do not like. But they are worse than BIST2 in some other attributes.

	request	weight	BIST2	UT2
TEST-GENERATION	NO	0.5E-1	NO	NO
TEST-STORAGE	5000	0.1	0	504426
TEST-APPLICATION-TIME	1	0.1	0.6	0.3E-1
SELF-TESTING	NO	0.1	YES	NO
CONCURRENT-TESTING	NO	0.1E-1	NO	NO
EXTRA-IO-PINS	4	0.1	4	4
AREA-OVERHEAD	25	0.2	27.2	23.7
&FAULT-COVERAGE	98	0.2	100.0	100.0
TOTAL SCORE			-38.9	-2182.5

According to your requirements, PCFs and weights, BIST2 is better than UT2. By paying 3.5 (%) MORE AREA-OVERHEAD, 0.6 (MS) MORE TEST-APPLICATION-TIME, you can obtain the advantages of 504426 (BITS) LESS TEST-STORAGE, and SELF-TESTING.

Please select one of the following:

1. I'd like to make some changes.
2. I'd like to see other choices.
3. I'd like to pick up one solution.
4. Please choose the best one for me.
5. Explain the possible choices.
6. I'd like to check the system state.
7. I don't like any of them.
8. Go back to previous choices.
9. Trace design history.
10. Exit.

\*9

### < Trace design history >

Please select one of the following:

1. Look at the whole design history.
2. Look back a number of steps.
3. Go back to an earlier stage.
4. Go back to the main menu. \*1

ACTION7 :

The solution BIST2 is rejected.

ACTION6 : The solution BIST2

is selected by the system under the user's request.

ACTION5 : CHANGE REQUIREMENTS

attribute	old value	new value
TEST-GENERATION	NO	NO
TEST-STORAGE	5000	5000
TEST-APPLICATION-TIME	1	1
SELF-TESTING	NO	NO
CONCURRENT-TESTING	NO	NO
EXTRA-IO-PINS	4	4
AREA-OVERHEAD	20	25*
&FAULT-COVERAGE	98	98

\* indicates the value which has been changed.

## ACTION4 : CHANGE REQUIREMENTS

attribute	old value	new value
TEST-GENERATION	NO	NO
TEST-STORAGE	5000	5000
TEST-APPLICATION-TIME	1	1
SELF-TESTING	NO	NO
CONCURRENT-TESTING	NO	NO
EXTRA-IO-PINS	1	4*
AREA-OVERHEAD	20	20
&FAULT-COVERAGE	98	98

\* indicates the value which has been changed.

## ACTION3 : CHANGE REQUIREMENTS

attribute	old value	new value
TEST-GENERATION	NO	NO
TEST-STORAGE	5000	5000
TEST-APPLICATION-TIME	1	1
SELF-TESTING	NO	NO
CONCURRENT-TESTING	NO	NO
EXTRA-IO-PINS	2	1*
AREA-OVERHEAD	20	20
&FAULT-COVERAGE	98	98

\* indicates the value which has been changed.

## ACTION2 : CHANGE REQUIREMENTS

attribute	old value	new value
TEST-GENERATION	NO	NO
TEST-STORAGE	5000	5000
TEST-APPLICATION-TIME	1.0E-4	1*
SELF-TESTING	NO	NO
CONCURRENT-TESTING	NO	NO
EXTRA-IO-PINS	2	2
AREA-OVERHEAD	20	20
&FAULT-COVERAGE	98	98

\* indicates the value which has been changed.

## ACTION1 : INITIAL STATE

attribute	requirement	relative weight
TEST-GENERATION	NO	5
TEST-STORAGE	5000	10
TEST-APPLICATION-TIME	1.0E-4	15
SELF-TESTING	NO	10
CONCURRENT-TESTING	NO	1
EXTRA-IO-PINS	2	9
AREA-OVERHEAD	20	20
&FAULT-COVERAGE	98	19

Please select one of the following:

1. Look at the whole design history.
2. Look back a number of steps.
3. Go back to an earlier stage.
4. Go back to the main menu. \*2

How many steps do you want to look backward or forward ?

(For looking backward, give a negative integer.

For looking forward, give a positive integer.

If you want to exit from tracing, just type 'n'.) \*-3

ACTION7 :

The solution BIST2 is rejected.

ACTION6 : The solution BIST2

is selected by the system under the user's request.

ACTIONS5 : CHANGE REQUIREMENTS

attribute	old value	new value
TEST-GENERATION	NO	NO
TEST-STORAGE	5000	5000
TEST-APPLICATION-TIME	1	1
SELF-TESTING	NO	NO
CONCURRENT-TESTING	NO	NO
EXTRA-IO-PINS	4	4
AREA-OVERHEAD	20	25*
&FAULT-COVERAGE	98	98

\* indicates the value which has been changed.

How many steps do you want to look backward or forward ?

(For looking backward, give a negative integer.

For looking forward, give a positive integer.

If you want to exit from tracing, just type 'n'.) \*2

ACTION6 : The solution BIST2

is selected by the system under the user's request.

ACTION7 :

The solution BIST2 is rejected.

How many steps do you want to look backward or forward ?

(For looking backward, give a negative integer.

For looking forward, give a positive integer.

If you want to exit from tracing, just type 'n'.) \*n

Please select one of the following:

1. Look at the whole design history.
2. Look back a number of steps.
3. Go back to an earlier stage.
4. Go back to the main menu. \*3

I'd like to remind you that backing up to an earlier stage 1 means UNDO all the actions after step 1. Do you want to do that ? \*y

How many steps do you want to back up ? \*3

Please wait while I try to find a solution for you.

No solution found.

There is a single conflict, i.e., there exist TDMs which satisfy all but one of your requirements as shown in the following table.

TDM	failed goal	request	actual	weight	Score
BIST2	AREA-OVERHEAD	25	27.2	0.2	-38.9
FIST	AREA-OVERHEAD	25	31.7	0.2	-436.1
UT2	TEST-STORAGE	5000	504426	0.1	-2182.5

Now you are very close to a good solution. You can easily get one by changing one of the requirements to a sufficient extent as indicated in the above table.

Please select one of the following:

1. I'd like to make some changes.
2. I'd like to see other choices.
3. I'd like to pick up one solution.
4. Please choose the best one for me.
5. Explain the possible choices.
6. I'd like to check the system state.
7. I don't like any of them.
8. Go back to previous choices.
9. Trace design history.
10. Exit.

\*3

### < User selects a TDM >

The following TDMs are relatively good candidates:

TDM	failed goal	request	actual	weight	Score
BIST2	AREA-OVERHEAD	25	27.2	0.2	-38.9
FIST	AREA-OVERHEAD	25	31.7	0.2	-436.1
UT2	TEST-STORAGE	5000	504426	0.1	-2182.5

Which one do you prefer ? (Give a TDM name or type 'none')\*bist2

That means that you want to change  
AREA-OVERHEAD from 25 to 27.2, don't you ? \*yes

You get it !  
SOLUTION FOUND : BIST2

Do you want detailed description of BIST2 ? \*y



## &lt; Data on the solution &gt;

## BIST2 BUILT-IN-SELF-TESTING-PLA

This is a built-in self-testable design proposed by Treuer et al., which employs the idea of parity compression. That is, instead of using two parity checkers to monitor the parities of the two PLA arrays, the parity signals are only generated at the PLA's outputs and accumulated in a flip-flop, then compared with the expected values at a number of specific times. Area is saved by eliminating the parity checking circuit for the product lines. In the test mode, a universal test set of length  $2m(1 + n) + 1$  is applied to the PLA's inputs. An interesting property of this scheme is that the sequence of cumulated parity bits at  $2n+2m+1$  selected check points is simply a sequence of alternative 0's and 1's. Hence it is very easy to generate the expected values on-line. The proven fault coverage of this testable design is very high: all single and  $(1-2^{-(m+2n)})\%$  of all multiple crosspoint, stuck and bridging faults are covered. The complete set of attribute-value pairs is shown below.

	BIST2
FAULT-MASKING	NO
CONCURRENT-TESTING	NO
ERROR-LATENCY	NIL
SELF-TESTING	YES
TEST-GENERATION	NO
&FAULT-COVERAGE	100.0
EXTRA-IO-PINS	4
EXTRA-DELAY	1
DELAY-PER-TEST	50.5
#-OF-TESTS	13123
EXTRA-BIT-LINE	0
TEST-APPLICATION-TIME	0.6
TEST-STORAGE	0
AREA-OVERHEAD	27.2
EXTRA-TRANSISTORS	1907.0
EXTRA-BIT-LINE	0
EXTRA-PRODUCT-LINE	3
EXTRA-OUTPUT-LINE	1
DESIGN-COST	2
LAYOUT-DIFFICULTY	1

BIST2 has no variation stored.

LAST CHANCE CONFIRMATION : Are you satisfied with the solution ? \*y

To implement BIST2, the following circuit modification is necessary:

1. The input decoder is modified to a shift register in the test mode. Two control lines C1 and C2 are also added to the AND array for selecting individual bit lines.
2. A shift register is added to select product lines.
3. One or two product lines are appended to the AND array such that every bit line has both an odd number of used crosspoints and an odd number of unused crosspoints.
4. One product line is added to the OR array such that the number of used crosspoints on every product line in the OR array is odd.
5. One parity chain is employed at the PLA's outputs.
6. A cumulative parity comparator is added to detect faulty information.

Fortunately, BIST2 does not require software test generation.

Do you want to

1. make a PLA testable,
2. maintain or update the knowledge base,
3. quit ? \*3

Bye-bye.

This example demonstrates the basic features of an expert consultant system for selection. By inserting different candidates into the knowledge base, the program can be easily changed to select TDMs for different type of kernels or to select other objects rather than TDMs. As more TDMs are invented, they can be put into the knowledge base without changing the program. In the case where many alternatives exist, such a knowledge based system will be a very useful CAD tool for designers.

## Chapter 6

# Conclusions and future work

In this chapter we summarize the results presented in this thesis and discuss future research topics related to TDM selection.

### 6.1. Conclusions

It is shown in Chapter 1 that design for testability is a rich and growing field. The available DFT techniques can be modeled using the framework of testable design methodologies (TDMs). Based on various TDMs, a knowledge based approach for DFT is introduced in which a VLSI circuit is first partitioned into manageable kernels, then a suitable TDM is selected for each kernel, and finally the selected TDMs are synthesized into the logic to form a testable circuit. The main focus of the research presented is to develop a methodology and a CAD tool which can help designers in selecting the most suitable TDM for a given kernel. TDMs for PLAs are used to illustrate our work.

In order to select the best TDM, a scheme for evaluating TDMs is first established. In Chapter 2 a set of measures for TDMs are defined both qualitatively and quantitatively. About 30 TDMs for PLAs are surveyed and characterized using 20 attributes. Comparative analysis of the TDMs

are presented. The richness of the TDMs and the multiple features of each TDM make the problem of selecting a TDM non-trivial.

In Chapter 3 the problem of TDM selection is formulated as a multiple criteria decision making problem. A general model for a class of selection problems is defined, in which the solution space is multi-dimensional, and the goals and constraints in a selection is defined by a requirement vector  $R = (r_1, \dots, r_n)$  where each  $r_j$  is a fuzzy bound for acceptable values of  $att_j$ . Functions for comparing and evaluating TDMs are defined. Differing from most previous work in MCDM which solves the problem by ranking the candidates, a selection is considered as a complete process which involves searching and backtracking. Using a knowledge based consultant system for helping selection is proposed. A framework for such a knowledge based system is established which contains knowledge about the selection domain and the art of decision making. The concepts and procedures for ramification analysis are defined and developed, as well as a reason directed backtracking scheme for controlling selection processes with variable constraints.

A prototype system, PLA-TSS, which implements the knowledge based selection system defined in Chapter 3 and can be used for TDM selection for PLAs has been built. The details of PLA-TSS is described in Chapter 4. A working example of PLA-TSS is given in Chapter 5. This and other examples are used to validate our concepts and illustrate the usefulness of the system.

The main contributions of this thesis lie in two fields.

### **DFT and CAD**

Our study emphasized the fact that design for testability, usually as a secondary design goal, is a problem which is filled with design choices and subject to multiple constraints. TDMs for PLAs are used to demonstrate this observation. Therefore designing a testable VLSI circuit not only requires TDMs which can improve testability, but also requires careful selection of the TDMs which satisfies various constraints and results in an optimal design. Separation of functional design and testable design, and the complexity involved in TDM selection, impede the application of design for testability. The knowledge based TDM selection system discussed in this thesis forms a bridge between test experts who create the TDMs and designers who need to use the TDMs. It provides a useful tool for computer aided design for testability.

### **Artificial intelligence**

In the 60's, AI researchers explored "weak" methods applicable to a very broad class of problems, for example, depth-first search and breadth-first search. In the 70's, knowledge-intensive "strong" methods for solving quite specific types of problems were created, such as expert systems. A major trend in the 80's is to identify coherent problem classes of intermediate generality and to develop corresponding general problem-solving methods. Selection problems constitute a coherent problem class. In this work, we defined this class, formalized a model of the selection process,

established a knowledge based selection system architecture, created primitive tools for building such a system, and implemented a prototype selection system.

## 6.2. Future work

This thesis addressed some of the problems involved in selecting TDMS for VLSI circuits. The TDM selection problem can be divided into three subproblems.

1. Determine a global test strategy based on the architecture, specification, and general requirements of the circuit to be made testable.
2. Select a set of TDMS for a number of kernels such that the combined results satisfy the global requirements and are optimal.
3. Determine a TDM for each kernel under a set of local requirements.

Only the third subproblem is dealt with in this thesis. The first two problems deserve further investigation.

### **Global test strategy selection**

In selecting a TDM for a kernel, we have assumed that there is a set of local requirements for making the kernel testable. In practice, the local requirements for different kernels are usually derived from a *global test strategy* for the entire circuit. For example, if a chip is to be made self-

testable, we may want every kernel in the chip to be self-testable. Hence determining a global test strategy is the first step in TDM selection.

Currently, testability is not explicitly specified in the specification of circuits, thus a suitable global test strategy and global requirements associated with testability need to be worked out from information about the circuit architecture, test and application environment, manufacturing technology and constraints, as well as functional specification. Since few chips have been made testable, there is currently not enough knowledge about how to link testability with other information. Knowledge acquisition is necessary, which may include (1) research in the topics such as the relationship between test strategy & architecture and testable design & yield improvements; (2) conducting massive experiments; and (3) gathering information from industry. Then a knowledge based system can be built to perform this task.

### **The multiple selection problem**

In this thesis the problem of selecting one candidate for one receiver is discussed. We call this a *single selection problem*. For a VLSI circuit consisting of many kernels, if the proposed divide and conquer strategy is used, it is necessary to select one TDM for each kernel and combine the testable structures into a testable chip. This can be generalized into a *multiple selection problem* in which there are a set of receivers and a set of alternatives, and the problem is to select one alternative for each receiver

(the same alternative may be repeatedly applied to more than one receiver) such that the combined result is optimal with respect to a set of global requirements. The multiple selection problem is complicated due to the fact that local optimal solutions may not result in a global optimal solution, because of possible complementarity or sharing of resources between the alternatives. Therefore a multiple selection problem cannot be simply solved by applying the method for single selection multiple times. There are two strategies for solving a multiple selection problem.

**I. Sequential strategy** which processes the receivers in sequence through the following steps.

1. Order the receivers into a list according to some heuristic rules.

Let the current requirements be the initial global requirements.

2. Select an alternative for the first receiver in the list under the current requirements.
3. Calculate the cumulated measures and adjust the current requirements accordingly.
4. Remove the receiver just considered from the list.
5. If the list is not empty, go to 2, otherwise, stop.

If the cumulated measures violate the requirements, backtracking is necessary. The efficiency of this method is greatly affected by the order in which receivers are processed and the alternatives are selected. A main problem is that there are few guidelines in making a decision (especially in



the earlier stages of the process when many choices exist) since the outcome of an ordering or a selection is hard to predict. It is also difficult to guarantee an optimal solution without trying all possible orderings due to the circular effect discussed in [87].

## **II. Parallel strategy** which has three basic steps.

1. Distribute the global requirements into local requirements for each receiver.
2. Select an alternative for each receiver using the local requirements.
3. Calculate the combined measure from the local selections.

If the combined measures violate the global requirements, either the chosen alternative(s) for one or more receivers should be changed, or the requirements need to be re-distributed and the process repeated again. Using the parallel selection strategy, ordering receivers is no longer necessary, and circular effects are eliminated. The main problems with this method are the following.

- How to distribute the global requirements into local requirements?
- How to combine local measures into global measures?
- How to efficiently prune out bad combinations of local selections and quickly reach an optimal solution (e.g. using branch-and-bound)?

Generalization of ramification analysis and reason analysis to multiple selection case should also be studied.

## References

- [1] Abadir, M.S. and M.A. Breuer.  
A knowledge based system for designing testable VLSI chips.  
*IEEE Design & Test of Computers* 2(4):56-68, August, 1985.
- [2] Abadir, M.S.  
*A knowledge based system for designing testable VLSI circuits.*  
PhD thesis, Dept. of EE-Systems, University of Southern California,  
December, 1985.
- [3] Abadir, M.S. and M.A. Breuer.  
Test schedules for VLSI circuits having built-in test hardware.  
*IEEE Transactions on Computers* C-35(3), 1986.
- [4] Agrawal, V.K.  
Multiple fault detection in programmable logic arrays.  
*IEEE Trans. on Computers* C-29(6):518-522, June, 1980.  
also in 1979 test conference.
- [5] Ando, H.  
Testing VLSI with random access scan.  
In *Digest of Papers COMPCON 80*, pages 50-52. February, 1980.
- [6] Barr, A. and E.A. Feigenbaum.  
*The Handbook Of Artificial Intelligence.*  
William Kaufmann, Inc., 1981.
- [7] Bellon, C., Ch. Robach, G. Saucier.  
An intelligent assistant for test program generation: The Supercat  
system.  
In *IEEE Int'l Conf. on Computer Aided Design*, pages 32-33.  
September, 1983.
- [8] Berger, J.M.  
A note on error detection codes for asymmetric channels.  
*Inform. Control* 4(1):68-73, March, 1961.

- [9] Bose, P. and J. A. Abraham.  
Test generation for programmable logic arrays.  
In *Proceedings of the 19th Design Automation Conference*, pages  
574-580. Las Vegas, Nevada, June 14-16, 1982.
- [10] Bozorgui-Nesbat, S. and E. J. McCluskey.  
Lower Overhead design for testability for PLAs.  
In *Proc. 1984 Int. Test Conf.*, pages 856 - 865. October, 1984.
- [11] Brayton, R.K., G.D. Hachtel, C.T. McMullen and A.L. Sangiovanni-  
Vincentelli.  
*Logic minimization algorithms for VLSI synthesis*.  
Kluwer Academic Publishers, 1984.
- [12] Breuer, M.A. and A.D. Friedman.  
*Diagnosis and Reliable Design of Digital Systems*.  
Computer Science Press, Rochville, MD, 1976.
- [13] Breuer, M. A.  
Knowledge based system for the design of testable VLSI circuit chips.  
January, 1984.  
Research proposal.
- [14] Cha, C. W.  
A testing strategy for PLAs.  
In *Proceeding of the 15th Design Automation Conference*, pages  
326-331. Las Vegas, Nevada, June 19-21, 1978.
- [15] Charniak, E., C. K. Riesbeck and D. V. McDermott.  
*Artificial Intelligence Programming*.  
Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1980.
- [16] Cochrane, J.L. and M. Zeleny (eds).  
*Multiple Criteria Decision Making*.  
University of South Carolina Press, Columbia, South Carolina, 1973.
- [17] Daehn, W. and J. Mucha.  
A hardware approach to self-testing of large programmable logic  
arrays.  
*IEEE Trans. on Computers* C-30:829-833, November, 1981.
- [18] Danner, F. and W. Consolla.  
An objective testability rating system.  
In *Proceedings. 1979 IEEE Test Conference*, pages 23-28. Cherry  
Hill, New Jersey, 1979.

- [19] Dong, H. and E.J. McCluskey.  
*Matrix representation of PLA's and an application to characterizing errors.*  
CRC Technical Report 81-11, Stanford University, September, 1981.
- [20] Dong H. and E.J. McCluskey.  
*Design of fully testable programmable logic arrays.*  
CRC Technical Report 81-20, Stanford University, December, 1981.
- [21] Dong, H.  
Modified Berger codes for detection of unidirectional errors.  
In *Proc. of the 12th Ann. Symp. on Fault-Tolerant Computing*, pages 317-320. June, 1982.
- [22] Dong, H. and E. J. McCluskey.  
*Concurrent testing of programmable logic arrays.*  
CRC Technical Report 82-11, Stanford University, June, 1982.
- [23] Eichelberger, E.B. and T.W. Williams.  
A logic design structure for LSI testing.  
In *Proceedings 14th Design Automation Conference*, pages 462-468.  
June, 1977.
- [24] Eichelberger, E. B. and E. Lindbloom.  
A heuristic test-pattern generator for programmable logic array.  
*IBM J. Research & Development* 24(1):15 - 22, January, 1980.
- [25] Fujiwara, H. and K. Kinoshita.  
A design of programmable logic arrays with universal tests.  
*IEEE Trans. on Computers* C-30:823-828, November, 1981.
- [26] Fujiwara, H.  
A new PLA design for universal testability.  
*IEEE Trans. on Computers* C-33(8):745 - 750, August, 1984.
- [27] Funatsu S., N. Wakatsuki and T. Arima.  
Test generation systems in Japan.  
In *Proceedings 12th Design Automation Conference*, pages 114-122.  
June, 1975.
- [28] Fung, H.S. and S. Hirschhorn.  
An automatic DFT system for the Slic silicon compiler.  
*IEEE Design & Test* 3(1):45-57, February, 1986.

- [29] Franacki, J.D., D. Knapp and A. Parker.  
The ADAM advanced Design AutoMation system: overview, planner,  
and natural language interface.  
In *Proc. of the 22nd design Automation Conference*, pages 727-730.  
June, 1985.
- [30] Grassl, G. and H.-J. Pfeleiderer.  
A function-independent self-test for large programmable logic arrays.  
*Integration* 1:71-80, 1983.  
North-Holland Publishing Company.
- [31] Hassan, S. Z.  
*Testing PLAs using multiple parallel signature analyzers.*  
CRC Technical Report 82-9, Stanford University, June, 1982.
- [32] Hess, R. D.  
Testability analysis: an alternative to structured design for testability.  
*VLSI Design* :22-29, March, 1982.  
United Technologies Microelectronics Center.
- [33] Hong, S. J. and D. L. Ostapko.  
FITPLA: A programmable logic array for functional independent  
testing.  
In *Digest of Papers 10th International Symposium Fault-Tolerant  
Computing*, pages 131-136. October, 1980.
- [34] Hortsmann, P.W.  
Design for testability using logic programming.  
In *Digest of Papers International Test Conference*. 1983.
- [35] Hua, K.A., J.Y. Jou and J.A. Abraham.  
Built-in tests for VLSI finite-state machines.  
In *Proc. 14th Int'l Symp. on Fault-Tolerant Computing*, pages 292 -  
297. June, 1984.
- [36] Hwang, C.L. and A.S.M. Masud.  
*Multiple Objective Decision Making -- Methods and Applications: A  
State-of-the-art Survey.*  
Springer-Verlag, Berlin/Heidelberg/New York, 1979.
- [37] Hwang, C.L. and K. Yoon.  
*Multiple Attribute Decision Making -- Methods and Applications: A  
State-of-the-art Survey.*  
Springer-Verlag, Berlin/Heidelberg/New York, 1981.

- [38] Keeney, R.L. and H. Raiffa.  
*Decision with Multiple Objectives: Preferences and Value Tradeoffs.*  
Wiley, New York, 1976.
- [39] Khakbaz, J. and E. J. McCluskey.  
*Concurrent error detection and testing for large PLAs.*  
CRC Technical Report 81-14, Stanford University, September, 1981.
- [40] Khakbaz, J.  
*A testable PLA design with low overhead and high fault coverage.*  
CRC Report 82-17, Stanford University, October, 1982.
- [41] Khakbaz, J.  
A testable PLA design with low overhead and high fault coverage.  
*IEEE Trans. on Computers C-33(8):743 - 745, August, 1984.*  
Also in Digest FTCS-13, pp. 426-429, June 1983.
- [42] Konemann, B. et al.  
Built-in logic block observation techniques.  
In *Digest of Papers 1979 Test Conference*, pages 37-41. October,  
1979.
- [43] Kovijanic, P. G.  
Testability analysis.  
In *Proceedings 1979 IEEE Test Conference*, pages 310-316. Cherry  
Hill, New Jersey, 1979.
- [44] Law, H-F S. and M. Shoji.  
PLA design for the BELLMAC-32A microprocessor.  
In *Proc. Int'l Conf. on Circuits and Computers.*, pages 161 - 164.  
1982.
- [45] Leitmann, G. and A. Marzollo (eds).  
*Multicriteria Decision Making.*  
Springer-Verlag, Wien/New York, 1975.
- [46] Liu, C.L.  
*Elements of discrete mathematics.*  
McGraw-Hill Book Company, 1977.
- [47] Logue, J.C., N.F. Brickman, F. Howley, J.W. Jones and W.W. Wu.  
Hardware implementation of a small system in Programmable logic  
arrays.  
*IBM Journal of Res. & Develop.* :110-119, March, 1975.

- [48] Longendorfer, B. A.  
A testability measure for hybrid circuits.  
In *Proceedings 1979 IEEE Test Conference*, pages 298-305. Cherry Hill, New Jersey, 1979.
- [49] Mangir, T.E.,  
Design for testability an integrated approach to VLSI testing.  
In *IEEE International Conference on Computer Aided Design*, pages 68-70. September, 1983.
- [50] Mead, C. and L. Conway.  
*Introduction to VLSI Systems*.  
Addison Wesley, 1980.
- [51] Min, Y.  
*A unified fault model for programmable logic arrays*.  
CRC Technical Report 83-5, Stanford University, May, 1983.
- [52] Min, Y.  
*Generating a complete test set for programmable logic arrays*.  
CRC Technical Report 83-4, Stanford University, May, 1983.
- [53] Min, Y.  
A PLA design for ease of test generation.  
In *Proc. 14th Int. Symp. on Fault-Tolerant Computing*, pages 436-442. June, 1984.
- [54] Muehldorf, E. I. and T. W. Williams.  
Optimized stuck fault test pattern generation for PLA macros.  
In *Digest of Papers Semiconductor Test Symposium*, pages 89-101. October, 1977.
- [55] Muehldorf, E.I., G.P. Papp and T. W. Williams.  
Efficient test pattern generation for embedded PLAs.  
In *Digest of 1980 IEEE Test Conference*, pages 349-358. October, 1980.
- [56] Ostapko, D. L. and S. J. Hong.  
Fault analysis and test generation for programmable logic arrays.  
In *Digest of Papers 8th International Symposium Fault-Tolerant Computing*, pages 83-89. June 21-23, 1978.
- [57] Ostapko, D. L. and S. J. Hong.  
Fault analysis and test generation for programmable logic arrays.  
*IEEE Trans. on Computers* C-28(9):617-627, September, 1979.



- [58] Pradhan, D. K. and K. Son.  
The effect of undetectable faults in PLAs and a design for testability.  
In *Proceedings IEEE International Test Conference*, pages 359-367.  
November, 1980.
- [59] Ramanatha, K.S. and N. N. Biswas.  
A design for complete testability of programmable logic arrays.  
In *Proc. 1982 Int. Test Conference*, pages 67 - 73. November, 1982.
- [60] Ramanatha, K.S. and N.N. Biswas.  
A design for testability of undetectable crosspoint faults in  
programmable logic arrays.  
*IEEE Trans. on Computers* C-32(6):551-557, June, 1983.
- [61] Rich, E.  
*Artificial Intelligence*.  
McGraw-Hill, 1983.
- [62] Roth, J. P.  
Diagnosis of automata failures.  
*IBM J. Research and Development* 10:278-291, July, 1966.
- [63] Roth, J. P.  
*Computer Logic, Testing and Verification*.  
Computer Science Press, Potomac, MD, 1980.
- [64] Saluja, K. K., K. Kinoshita and H. Fujiwara.  
A multiple fault testable design of programmable logic arrays.  
In *Digest of Papers 11th International Symposium Fault-Tolerant  
Computing*, pages 44-46. June, 1981.
- [65] Saluja, K.K., K. Kinoshita and H. Fujiwara.  
An easily testable design of programmable logic arrays for multiple  
faults.  
*IEEE Trans. on Computers* C-32:1038-1046, November, 1983.
- [66] Saluja, K.K. and J.S. Upadhyaya.  
*A built-in self testing programmable logic array design with  
extremely high fault coverage*.  
Technical Report EE8517, Dept. of Electrical and Computer  
Engineering, University of Newcastle, N.S.W., 2308, Australia,  
March, 1985.

- [67] Saluja, K.K. and J.S. Upadhyaya.  
Divide and conquer strategy for testable design of programmable logic arrays.  
In *Proceeding of the 4th Australian Microelectronics Conference*.  
May, 1985.
- [68] Saluja, K.K., H. Fujiwara and K. Kinoshita.  
*Testable design of programmable logic arrays with universal control and minimal overhead*.  
Technical Report EE8508, Dept. of Electrical and Computer Eng.,  
University of Newcastle, N.S.W. Australia, 1985.
- [69] Saluja, K.K., K. Kinoshita and C. Boswell.  
A design of parallel testable programmable logic arrays.  
Technical report No. 8501, Dept. of Electrical and Computer  
Engineering, University of Newcastle, N.S.W., 2308, Australia.
- [70] Smith, J. E.  
Detection of faults in programmable logic arrays.  
*IEEE Trans. on Computers* C-28(11):848-853, November, 1979.
- [71] Son, K. and D.K. Pradhan.  
Design of programmable logic arrays for testability.  
In *Proceedings IEEE International Test Conference*, pages 163-166.  
November, 1980.
- [72] Son, K. and D.K. Pradhan.  
Complete self-checking checkers in PLAs.  
In *Proceedings of 1981 IEEE Test Conference*, pages 231-237. 1981.
- [73] Sridhar, T. and J.P. Hayes.  
A functional approach to testing bit-sliced microprocessors.  
*IEEE Trans. on Computers* C-30(8):563-571, August, 1981.
- [74] Starr, M.K., A.K. Romney, and S.B. Nerlove (eds).  
*Multiple Criteria Decision Making*.  
North Holland, New York, 1977.
- [75] Stewart, J.H.  
Future testing of large LSI circuit cards.  
In *Digest of Papers IEEE semiconductor Test Symposium*, pages  
6-17. Oct., 1977.
- [76] Takasaki, S., M. Kawai, S. Funatsu and A. Yamada.  
A calculus of testability measure at the functional level.  
In *Proceedings 1981 IEEE Test Conference*, pages 95-101. 1981.

- [77] Treuer, R., H. Fujiwara and V.K. Agrawal.  
Implementing a built-in self-test PLA design.  
*IEEE Design & Test of Computers* 2(2):37 - 48, April, 1985.  
Also see Technical Report No.84.13, VLSI Design Lab. Dept. of  
Electrical engineering, McGill University, 1984.
- [78] Wakerly, J.  
*Error Detecting Codes, Self Checking Circuits and Applications.*  
American-Elsevier, N.Y., 1978.
- [79] Wang, S.L. and A. Avizienis.  
The design of totally self-checking circuits using programmable logic  
arrays.  
In *Digest of Papers 9th International Symposium Fault-Tolerant  
Computing*, pages 173-180. June, 1979.
- [80] Wei, R.-S. and A. Sangiovanni-Vincentelli.  
PLATYPUS : A PLA test pattern generation tool.  
In *Proceedings of the 22nd Design Automation Conference*, pages  
197-203. June, 1985.
- [81] Williams, T.W. and E.B. Eichelberger.  
Random patterns within a structured sequential logic design.  
In *Digest of 1977 Semiconductor Test Symposium*, pages 19-28.  
October 25-27, 1977.
- [82] Williams, T.W. and K.P. Parker.  
Design for testability - A survey.  
*IEEE Trans. on Computers* C-31(1):2-15, Jan, 1982.
- [83] Yajima, S. and T. Aramaki.  
Autonomously testable programmable logic arrays.  
In *Digest of Papers 11th International Symposium Fault-Tolerant  
Computing*, pages 41-43. June, 1981.
- [84] Thiriez, H. and S. Zions (eds).  
*Multiple Criteria Decision Making: Jouy-en-Josas, France.*  
Springer-Verlag, Berlin/Heidelberg/New York, 1976.
- [85] Zeleny, M. (eds).  
*Multiple Criteria Decision Making: Kyoto 1975.*  
Springer-Verlag, Berlin/Heidelberg/New York, 1976.
- [86] Zions, S. (eds).  
*Multiple Criteria Problem Solving: Proceedings, Buffalo, NY, 1977.*  
Springer-Verlag, Berlin/Heidelberg/New York, 1978.

- [87] Zhu, X.  
Building an expert system for design for testability.  
1984.  
Dissertation proposal, Dept. of EE-Systems, USC, May 1984.
- [88] Zhu, X. and M.A. Breuer.  
A knowledge based system for selecting a test methodology for a  
PLA.  
In *Proc. 22nd Design Automation Conference*, pages 259-265. June,  
1985.