

SOFTWARE TESTING METHODOLOGIES

COURSE FILE

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
(2015-2016)

Contents

S.No	Topic	Page. No.
1	Cover Page	1
2	Syllabus copy	2
3	Vision of the Department	3
4	Mission of the Department	4
5	PEOs and POs	5
6	Course objectives and outcomes	6
7	Course mapping with POs	7
8	Brief notes on the importance of the course and how it fits into the curriculum	8
9	Prerequisites if any	9
10	Instructional Learning Outcomes	11
11	Class Time Table	15
12	Individual Time Table	18
13	Lecture schedule with methodology being used/adopted	22
14	Detailed notes	23
15	Additional topics	81
16	University Question papers of previous years	85
17	Question Bank	88
18	Assignment Questions	89
19	Unit wise Quiz Questions and long answer questions	91
20	Tutorial problems	109
21	Known gaps ,if any and inclusion of the same in lecture schedule	109
22	Discussion topics , if any	109
23	References, Journals, websites and E-links if any	110
24	Quality Measurement Sheets	111
A	Course End Survey	111
B	Teaching Evaluation	111
25	Student List	112
26	Group-Wise students list for discussion topic	117

Course coordinator

Program Coordinator

HOD

JNTU World

Syllabus:**UNIT-I:**

Introduction:- Purpose of testing, Dichotomies, model for testing, consequences of bugs, taxonomy of bugs

UNIT-II:

Flow graphs and Path testing:- Basics concepts of path testing, predicates, path predicates and achievable paths, path sensitizing, path instrumentation, application of path testing.

UNIT-III:

Transaction Flow Testing:-transaction flows, transaction flow testing techniques. Dataflow testing:- Basics of dataflow testing, strategies in dataflow testing, application of dataflow testing.

UNIT-IV:

Domain Testing:-domains and paths, Nice & ugly domains, domain testing, domains and interfaces testing, domain and interface testing, domains and testability.

UNIT-V:

Paths, Path products and Regular expressions:- path products & path expression, reduction procedure, applications, regular expressions & flow anomaly detection.

UNIT-VI:

Logic Based Testing:- overview, decision tables, path expressions, kv charts, specifications.

UNIT-VII:

State, State Graphs and Transition testing:- state graphs, good & bad state graphs, state testing, Testability tips.

UNIT-VIII:

Graph Matrices and Application:-Motivational overview, matrix of graph, relations, power of a matrix, node reduction algorithm, building tools. (Student should be given an exposure to a tool like JMeter or Win-runner).

TEXT BOOKS :

1. Software Testing techniques – Baris Beizer, Dreamtech, second edition.
2. Software Testing Tools – Dr.K.V.K.K.Prasad, Dreamtech.

REFERENCES :

1. The craft of software testing – Brian Marick, Pearson Education.
2. Software Testing Techniques – SPD(Oreille)
3. Software Testing in the Real World – Edward Kit, Pearson.
4. Effective methods of Software Testing, Perry, John Wiley.
5. Art of Software Testing – Meyers, John Wiley.

Vision of the Department

To produce globally competent and socially responsible computer science engineers contributing to the advancement of engineering and technology which involves creativity and innovation by providing excellent learning environment with world class facilities.

Mission of the Department

1. To be a center of excellence in instruction, innovation in research and scholarship, and service to the stake holders, the profession, and the public.
2. To prepare graduates to enter a rapidly changing field as a competent computer science engineer.
3. To prepare graduate capable in all phases of software development, possess a firm understanding of hardware technologies, have the strong mathematical background necessary for scientific computing, and be sufficiently well versed in general theory to allow growth within the discipline as it advances.
4. To prepare graduates to assume leadership roles by possessing good communication skills, the ability to work effectively as team members, and an appreciation for their social and ethical responsibility in a global setting.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

1. To provide graduates with a good foundation in mathematics, sciences and engineering fundamentals required to solve engineering problems that will facilitate them to find employment in industry and / or to pursue postgraduate studies with an appreciation for lifelong learning.
2. To provide graduates with analytical and problem solving skills to design algorithms, other hardware / software systems, and inculcate professional ethics, inter-personal skills to work in a multi-cultural team.
3. To facilitate graduates to get familiarized with the art software / hardware tools, imbibing creativity and innovation that would enable them to develop cutting-edge technologies of multi-disciplinary nature for societal development.

PROGRAM OUTCOMES (PO)

1. An ability to apply knowledge of mathematics, science and engineering to develop and analyze computing systems.
2. an ability to analyze a problem and identify and define the computing requirements appropriate for its solution under given constraints.
3. An ability to perform experiments to analyze and interpret data for different applications.
4. An ability to design, implement and evaluate computer-based systems, processes, components or programs to meet desired needs within realistic constraints of time and space.
5. An ability to use current techniques, skills and modern engineering tools necessary to practice as a CSE professional.
6. An ability to recognize the importance of professional, ethical, legal, security and social issues and addressing these issues as a professional.
7. An ability to analyze the local and global impact of systems /processes /applications /technologies on individuals, organizations, society and environment.
8. An ability to function in multidisciplinary teams.
9. An ability to communicate effectively with a range of audiences.
10. Demonstrate knowledge and understanding of the engineering, management and economic principles and apply them to manage projects as a member and leader in a team.
11. A recognition of the need for and an ability to engage in life-long learning and continuing professional development
12. Knowledge of contemporary issues.
13. An ability to apply design and development principles in producing software systems of varying complexity using various project management tools.
14. An ability to identify, formulate and solve innovative engineering problems.

Course Objectives

The aim of this course is,

- To study the fundamental concepts of software testing which includes objectives, process, criteria, strategies, and methods.
- To discuss various software testing types and levels of testing like black and white box testing along with levels unit test, integration, regression, and system testing.

- It also helps to learn the types of bugs, testing levels with which the student can very well identify a bug and correct as when it happens.
- It provides knowledge on transaction flow testing and data flow testing techniques so that the flow of the program is tested as well.
- To learn the domain testing, path testing and logic based testing to explore the testing process easier.
- To know the concepts of state graphs, graph matrixes and transition testing along with testability tips to enhance the testing process in different way.
- To expose the advanced software testing topics, such as object-oriented software testing methods, and component-based software testing issues, challenges, and solutions.
- To gain software testing experience by applying software testing knowledge and methods to practice-oriented software testing projects.
- To gain the techniques and skills on how to use modern software testing tools to support software testing projects

Course Outcomes

- Know the basic concepts of software testing and its essentials.
- Able to identify the various bugs and correcting them after knowing the consequences of the bug.
- Use of program's control flow as a structural model is the corner stone of testing.
- Performing functional testing using control flow and transaction flow graphs.
- Know the basic techniques for deriving test cases
- Follow an effective, step-by-step process for identifying needed areas of testing, designing test conditions and building and executing test cases.
- Able to test a domain or an application and identifying the nice and ugly domains.
- Able to make a path expression and reduce them very well when needed.
- Can use the testing tools and perform the testing of any type with good perfection.
- Apply appropriate software testing tools, techniques and methods for even more effective systems during both the test planning and test execution phases of a software development project.
- Well developed knowledge in comparing the various testing strategies as well.

Mapping of Course to PEOs and POs

S.No.	Course Outcome	POs
1	Know the basic concepts of software testing and its essentials.	1,2,3
2	Able to identify the various bugs and correcting them after knowing the consequences of the bug	2,3,4,12
3	Use of program's control flow as a structural model is the corner stone of testing.	1,3,5
4	Performing functional testing using control flow and transaction flow graphs.	1,8,5
5	Know the basic techniques for deriving test cases	2,5,8,13
6	Follow an effective, step-by-step process for identifying needed areas of testing, designing test conditions and building and executing test cases.	
7	Able to test a domain or an application and identifying the nice and ugly domains	1,3,14
8	Able to make a path expression and reduce them very well when needed.	3,7
11	Apply appropriate software testing tools, techniques and methods for even more effective systems during both the test planning and test execution phases of a software development project.	3,5,10,13

12	Well developed knowledge in comparing the various testing strategies as well.	7,13,14
----	---	---------

Course	PEOS	POs
STM	PEO1,PEO2,PEO3	PO1PO2,PO3,PO4,PO5,PO6,PO7,PO11,PO12,PO13

Mapping of Course outcomes with Programme outcomes:

*When the course outcome weightage is < 40%, it will be given as moderately correlated (1).

*When the course outcome weightage is >40%, it will be given as strongly correlated (2).

Pos	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Professional core
STM															
CO1: Know the basic concepts of software testing and its essentials.	2	1	2			2		1			2				
CO 2: Able to identify the various bugs and correcting them after knowing the consequences of the bug	2		2	2		1			1				2		
CO 3: Use of program's control flow as a structural model is the corner stone of testing.	2	2			1		2		1			1		1	
CO 4: Performing	2		1	1		2			2		2		1		

functional testing using control flow and transaction flow graphs.															
CO 5: Know the basic techniques for deriving test cases	1			1			2		2		2	2			
CO 6: Follow an effective, step-by-step process for identifying needed areas of testing, designing test conditions and building and executing test cases.	2	1	1	2		2	1				1				
CO 7: Able to test a domain or an application and identifying the nice and ugly domains	1			1			1			2		1		2	
CO 8: Able to make a path expression and reduce them very well when needed.	2		2		1		1			2		1			

CO 9: Apply appropriate software testing tools, techniques and methods for even more effective systems during both the test planning and test execution phases of a software development project.	2	1			1	1	2		2			2	2	1	
CO 10: Well developed knowledge in comparing the various testing strategies as well.	2	2	2		1	1	2			1	1	2			

Learning Outcomes

Upon successful completion of this course, the student will be able to:

- Have an ability to apply software testing knowledge and engineering methods.
- To apply the fundamental knowledge of testing real time scenarios
- To test a simple application of their choice and to understand those learnt techniques in software development life cycle.
- Have an ability to design and conduct a software test process for a software testing project.
- Have an ability to identify the needs of software test automation, and define and develop a test tool to support test automation.
- Have an ability understand and identify various software testing problems, and solve these problems by designing and selecting software test models, criteria, strategies, and methods.
- Have an ability to use various communication methods and skills to communicate with their teammates to conduct their practice-oriented software testing projects.
- Have basic understanding and knowledge of **contemporary** issues in software testing, such as component-based software testing problems.
- Have an ability to use software testing methods and modern software testing tools for their testing projects.

Prerequisites

- Proper knowledge on software engineering and their concepts
- Enough knowledge on object oriented modeling and techniques
- Knowing the different types and levels of software testing process
- Good programming skills and debugging skills.

Instructional learning outcomes

S.No	Unit	Contents	Outcomes
1	I	Introduction:- Purpose of testing, Dichotomies, model for testing, consequences of bugs, taxonomy of bugs	<p>At the end of the chapter</p> <ul style="list-style-type: none"> • Convinces the student that this course is indeed important • Gives a broad overview on purpose and goals for testing • Gives a better idea on bugs. • Optimistic notions about bugs • It gives good scenario on tests and testing levels on different software models
2	II	Flow graphs and Path testing:- Basics concepts of path testing, predicates, path predicates and achievable paths, path sensitizing, path instrumentation, application of path testing.	<p>At the end of the chapter</p> <ul style="list-style-type: none"> • Student will be able to create control flow graphs from programs • Students will able to test the path with loops. • It finds a set of solutions to the path predicate expression.

3	III	Transaction Flow Testing:-transaction flows, transaction flow testing techniques. Dataflow testing:- Basics of dataflow testing, strategies in dataflow testing, application of dataflow testing.	At the end of the chapter <ul style="list-style-type: none"> • Student has a clear understanding of Specifying requirements of big, online and complicated transaction flow. • Students develop Confidence in the program data flow. data dominated design. source code for data declarations.
4	IV	Domain Testing:-domains and paths, Nice & ugly domains, domain testing, domains and interfaces testing, domain and interface testing, domains and testability.	At the end of the chapter <ul style="list-style-type: none"> • Ability to perform Domain testing by viewing programs as input data classifiers. • Performing functional and structural testing. • Classification of different domains for testing program
5	V	Paths, Path products and Regular expressions:- path products & path expression, reduction procedure, applications, regular expressions & flow anomaly detection.	At the end of the chapter <ul style="list-style-type: none"> • Understanding purpose and application of flow graphs, performing syntax and state testing • Representing path and regular expressions • Identifying structured and unstructured flow graphs

6	VI	Logic Based Testing:- overview, decision tables, path expressions, kv charts, specifications.	<p>At the end of the chapter</p> <ul style="list-style-type: none"> • Modeling logic based testing with decision tables • Constructing decision tables, and reducing Boolean expressions by karnaugh maps • Specifying path expressions and Kv charts for consistency
7	VII	State, State Graphs and Transition testing:- state graphs, good & bad state graphs, state testing, Testability tips.	<p>At the end of the chapter</p> <ul style="list-style-type: none"> • Graphical representation of state graphs • Knowing the properties of state graphs, advantages and its disadvantages • Software implementation of state graphs • Differentiating between good and bad state graphs

8	VIII	Graph Matrices and Application:- Motivational overview, matrix of graph, relations, power of a matrix, node reduction algorithm, building tools. (Student should be given an exposure to a tool like JMeter or Win-runner).	At the end of the chapter <ul style="list-style-type: none"> Introducing graph matrices, understanding testing theory Applications of graph matrices Implementation of node-reduction algorithms Brief idea on software testing tools like JMeter or Win Runner
----------	-------------	---	---

11.Class Time Tables IV A,B,C sec

Department of Computer Science & Engineering

ODD SEMESTER

Year/Sem/Sec: IV-B.Tech I-Semester A-Section

Room No: LH-30

A.Y : 2015 - 16

WEF:22-06-2015(V1)

Class Teacher: T.SOWMYA

Time	09.30-10.20	10.20-11.10	11.10 - 12.00	12.00-12.50	12.50 -1.30	1.30 - 2.20	2.20 - 3.10	3.10-4.00
Period	1	2	3	4	LUNCH	5	6	7
Monday	DWDM	DP	STM	LP		DWDM	CG	CC
Tuesday	LP	LP LAB				CG	DP	CC
Wednesday	CG	DWDM	CC	LP		DP	STM	LP
Thursday	STM	STM LAB				CC	CG	STM
Friday	DP	DWDM	CRT			CRT		LIBRARY
Saturday	CC	STM	CG	DP		DWDM	LP	SPORTS

S.No	Subject(T/P)	Faculty Name
1	LINUX PROGRAMMING	M.VAMSI KRISHNA
2	SOFTWARE TESTING METHODOLOGIES	A.LALITHA

3	DATA WAREHOUSING DATA MINING	T.SOUMYA	
4	COMPUTER GRAPHICS	P.SWATHI	
5	CLOUD COMPUTING	Y.PHANI KISHORE	
6	DESIGN PATTENS	CH.V.ANUPNMA	
7	SOFTWARE TESTING METHODOLOGIES LAB	A.LALITHA/CH.V.ANUPAMA	
8	DATA WAREHOUSING DATA MINING LAB	M.VAMSI KIRSHNA/T.SOUMYA/MADHURI	
9	*-Tutorial Hour/Discussion Hour		

TT. Coord: _____

HOD: _____

Dean Academics:-

Principal: _____

Geethanjali College of Engineering & Technology
Department of Computer Science & Engineering

ODD SEMESTERYear/Sem/Sec: IV-B.Tech I-Semester B-
Section

Room No: LH-31

A.Y : 2015 - 16 WEF:22-06-
2015(V1)Class Teacher: N.RADHIKA
AMARESHWARI

Time	09.30-10.20	10.20-11.10	11.10 - 12.00	12.00-12.50	12.50 -1.30	1.30 - 2.20	2.20 - 3.10	3.10-4.00	
Period	1	2	3	4	LUNCH	5	6	7	
Monday	LP	DWDM LAB				DP	STM	CG	
Tuesday	STM	DWD M	CC	LP		STM LAB			
Wednesday	DWDM	CG	CC	CG		STM	LP	DP	
Thursday	DP	DWD M	LP	DWDM		STM	CG	CC	
Friday	CC	DP	CRT			CRT		LIBRARY	
Saturday	CG	STM	CC	LP		DP	DWDM	SPORTS	
S.No	Subject(T/P)					FACULTY NAME			
1	LINUX PROGRAMMING			V.SHIVA NARAYANA REDDY					

2	SOFTWARE TESTING METHODOLOGIES		N.RADHIKA				
3	DATA WAREHOUSING DATA MINING		Y.SWATHI TEJA				
4	COMPUTER GRAPHICS		B.SUBBA RAO				
5	CLOUD COMPUTING		G.PRASOONA				
6	DESIGN PATTENS		K.VIJAY BHASKAR				
7	SOFTWARE TESTING METHODOLOGIES LAB		N.RADHIKA/B.SUBBA RAO/G.PRASOONA				
8	DATA WAREHOUSING DATA MINING LAB		V S N REDDY/Y.SWATHI TEJA/B.SUBBA RAO				
9	*-Tutorial Hour/Discussion Hour						

TT. Coord: _____

HOD: _____

Dean Academics:-

Principal: _____

Geethanjali College of Engineering & Technology

Department of Computer Science & Engineering

ODD SEMESTERYear/Sem/Sec: IV-B.Tech I-Semester C-
Section

Room No: LH-32

A.Y : 2015 - 16 WEF:22-06-
2015(v1)

Class Teacher: A.LALITHA

Time	09.30-10.20	10.20-11.10	11.10 - 12.00	12.00-12.50	12.50 -1.30	1.30 - 2.20	2.20 - 3.10	3.10-4.00
Period	1	2	3	4	LUNCH	5	6	7
Monday	CG	LP	DWD M	DP		CC	STM	LP
Tuesday	DP	STM	CG	DP		CC	LP	CG
Wednesday	STM	STM LAB				CG	DW DM	DP
Thursday	CC	DWD M	CG	DWDM		DMDW LAB		
Friday	LP	STM	CRT			CRT		LIB RAR Y

Satur day	DWDM	LP	CC	STM		CG	DP	SPOR TS
S.No	Subject(T/P)			FACULTY NAME				
1	LINUX PROGRAMMING			M.VAMSI KRISHNA				
2	SOFTWARE TESTING METHODOLOGIES			A.LALITHA				
3	DATA WAREHOUSING DATA MINING			T.SOUMYA				
4	COMPUTER GRAPHICS			P.SWATHI				
5	CLOUD COMPUTING			Y.PHANI KISHORE				
6	DESIGN PATTENS			CH.V. ANUPAMA				
7	SOFTWARE TESTING METHODOLOGIES LAB			A.LALITHA/CH.V.ANUPAMA/B. SUBBA RAO				
8	DATA WAREHOUSING DATA MINING LAB			M.VAMSI KRISHNA/T.SOUMYA/MADHURI/Y.SWATHI TEJA				
9	*-Tutorial Hour/Discussion Hour							

TT. Coord: _____

HOD: _____

Dean Academics:-

Principal: _____

12.Individual Time Table

Faculty Name:A.LALITHA			Sub/Lab: STM / CT& ST LAB (A & C) IV CSE I Sem			A.Y:2015-16 I semester		
Time	09.30-10.20	10.20-11.10	11.10-12.00	12.00-12.50	12.50-1.30	1.30-2.20	2.20-3.10	3.10-4.00
Period	1	2	3	4	LUNCH	5	6	7
Monday			A				C	
Tuesday		C						
Wednesday	C	CT& ST LAB C					A	
Thursday	A	CT& ST LAB A						A
Friday		C						
Saturday		A		C				

Faculty Name:N RADHIKA			Sub/Lab: STM / CT&ST LAB (A & B) IV CSE I Sem				A.Y:2015-16 I semester	
Time	09.30-10.20	10.20-11.10	11.10-12.00	12.00-12.50	12.50-1.30	1.30-2.20	2.20-3.10	3.10-4.00
Period	1	2	3	4	LUNCH	5	6	7
Monday							B	
Tuesday	B					CT &ST LAB B		
Wednesday		CT &ST LAB C				B		
Thursday		CT &ST LAB A				B		
Friday		B						
Saturday								

13. Lecture Schedule

S. No	Unit No	Total no. of Periods	Topics to be covered	Regular / Additional	Teaching aids used LCD/OHP/BB	Date
1	1	1	Importance of the subject	regular	BB	
2		1	Course objectives and outcomes	regular	BB	
3		1	Purpose of testing	regular	BB	
4		1	Goals of testing	regular	BB	
5		2	Dichotomies	regular	BB	
6		1	Model of testing	regular	BB	
7		1	Role of models	regular	BB	
8		1	Consequences of bugs	regular	BB	
9		1	Bugs effects	regular	BB	
10		2	Taxonomy of bugs	regular	BB	
11		1	A Project case study	regular	BB	
12	2	1	Summary & assignment questions	regular		
12		1	Basic concepts of path testing	regular	BB	
13		1	Predicates	regular	BB	
14		2	Path predicates and achievable paths	regular	PPT	
15		1	Path sensitizing	regular	BB	
16		1	Path instrumentation	regular	BB	
17		1	Application of path testing	regular	BB	
18	3	1	Summary & assignment questions	regular	BB	
19		1	Transaction flow , testing	regular	PPT	

20		2	Transaction flow testing techniques	regular	PPT	
21		1	Dataflow testing: Basics of dataflow testing	regular	BB	
22		1	Strategies in dataflow testing	regular	BB	
23		1	Applications of dataflow testing	additional	BB	
24	4	2	Domains and paths	regular	BB	
25		2	Nice & ugly domains	regular	BB	
26		2	Domain testing	regular	BB	
27		3	Domains and interfaces testing	regular	BB	
28		3	Domain and interface testing	regular	BB	
29		2	Domains and testability	regular	BB	
30		1	Revision of domain testing	regular	BB	
31	5	1	Path products & path expressions	regular	BB/OHP	
32		2	Reduction procedure	regular	BB	
33		1	applications	regular	BB	
34		1	Regular expressions	regular	BB/OHP	
35		2	Flow anomaly detection	regular	BB	
36		1	Flow anomaly detection	regular	BB	
37		1	Flow anomaly detection	regular	BB	
38		1	Revision of path expressions	regular	BB	
39	6	1	overview	regular	BB	
40		1	Decision tables	regular	BB/OHP	
41		1	Path expressions	regular	BB	
42		1	KV charts	regular	BB/OHP/PPT	
43		1	KV charts	regular	BB/OHP/PPT	
44		2	specifications	regular	BB	
45		1	Revision of decisions tables	regular	BB	
46	7	1	State graphs	regular	BB/OHP	
47		1	State graphs	regular	BB/OHP	
48		1	Good & Bad state graphs	regular	BB/OHP	
49		2	Good & Bad state graphs	regular	BB/OHP	
50		1	State testing	regular	BB	
51		1	Testability tips	regular	BB	
52		1	Revision of state graphs	regular	BB	
53	8	2	Motivational overview	regular	BB	
54		1	Matrix of graph ,relations	regular	BB	

			, power of a matrix			
55		1	Node reduction algorithm	regular	BB	
56		1	Building tools	regular	BB	
57		1	Usage of JMeter and Win runner tools for functional/Regression testing	regular	PPT	
58		1	Creation of test script for unattended testing	regular	PPT	
59		1	Synchronization of test case	regular	BB	
60		1	Common modeling technique	regular	BB	
61		1	Rapid testing, performance testing of a data base application	regular	PPT	

Total no of classes: 61

14. Detailed Notes

UNIT-I

The Purpose of Testing

Test design and testing takes longer than program design and coding. In testing each and every module is tested. Testing is a systematic approach that the given one is correct or not. Testing find out the bugs in a given software.

Productivity and Quality in Software

Once in production, each and every stage is subjected to quality control and testing from component source inspection to final testing before shipping. If flaws are discovered at any stage, that part will be discarded or cycled back for rework and correction. The assembly line's productivity is measured by the sum of the cost of the materials, the rework, discarded components and the cost of quality assurance and testing.

Note :

By testing we get the quality of a software.

If we give the guarantee for the quality of a product then it is called Quality Assurance.

Goals for Testing

Testing and test design as a parts of quality assurance, should also focus on bug prevention. To the extent that testing and test design do not prevent bugs, they should be able to discover symptoms caused by bugs. Bug prevention is testing first goal. Bug prevention is better than the detection and correction. There is no retesting and no time wastage. The act of testing, the act of designing tests is one of the bug preventions. Before coding test can be performed. "Test, then code". Testing discover and eliminates bugs before coding.

Bug prevention – Primary goal

Bug discovery – Secondary goal

Phases in a Tester's Mental Life

Why testing ?

Phase 0 : There's no difference between testing and debugging. Here there is no effective testing, no quality assurance and no quality.

Phase 1 : The purpose of testing is to show that the software works. Testing increases, software works decreases. There is a difference between testing and debugging. If testing fails the software doesn't work.

Phase 2 : The purpose of testing is to show that the software doesn't works. The test reveals a bug, the programmer corrects it, the test designer designs and executes another test intended to demonstrate another bug. It is never ending sequence.

Phase 3 : The purpose of testing is not to prove anything, but to reduce the perceived risk of not working to an acceptable value. Here testing implements the quality control. To the extent that testing catches bugs and to the extent that those bugs are fixed, testing does improve the product. If a test is passed, then the product's quality does not change, but our perception of that quality does.

Note : testing pass or fail reduces our perception of risk about a software product.

Phase 4 : Here what testing can do and not to do. The testability is that goal for two reasons : 1.Reduce the labor of testing.

2.Testable code has fewer bugs than code that's hard to test.

Test design : Design means documenting or modeling. In test design phase the given system is tested that bugs are present or not. If test design is not formally designed no one is sure whether there was a bug or not. So, test design is an important one to get the system without any bugs.

Testing isn't everything

We must first review, inspect, read, do walkthroughs and then test. The major methods in decreasing order of effectiveness as follows :

Inspection methods : It includes walkthroughs, desk checking, formal inspection and code reading. These methods appear to be as effective as testing, but the bugs caught do not completely overload.

Design style : It includes testability, openness and clarity to prevent bugs.

Static Analysis Methods : It includes of strong typing and type checking. It eliminates an entire category of bugs.

Languages : The source language can help reduce certain kinds of bugs. Programmers find new kinds of bugs in new languages, so the bug rate seems to be independent of the languages used.

Design methodology and Development Environment : Design methodology can prevent many kinds of bugs. Development process used and the environment in which what methodology is embedded.

The pesticide paradox and the complexity Barrier

1.Pesticide Paradox :. Every method you use to prevent or find bugs leaves a residue of subtler bugs again which those methods are ineffectual

2.Complexity Barrier : Software complexity grows to the limits of our ability to manage that complexity.

Some Dichotomies

Testing versus Debugging : The phrase “Test and Debug “ is treated as a single word.

The purpose of testing is to show that a program has bugs.

The purpose of debugging is find the error or misconception that led to the program's failure and to design and implement the program changes that correct the error.

Note : Debugging usually follows testing, but they differ as to goals, methods and psychology.

Function versus Structure : Tests can be designed from a functional or structural point of view. In functional testing the program or system is treated as a blackbox.

Black box Testing : Here we don't know the internal functionality we know only about the input and the outcome. In structural testing does look at the implementation details, as programming style, control method, source language, database design and coding details.

White box Testing : Here inter functionality is considered

Designer versus the Tester : Designing depends on a system's structural details. The more you know about the design, the likelier you are to eliminate useless tests.

Tester, test-team member or test designer contrast to the programmer and program designer. Testing includes unit testing to unit integration, component testing to component integration, system testing to system integration.

Modularity versus Efficiency : Both tests and systems can be modular. A module is a discrete, well defined small component of a system. The smaller the component, the easier is to understand but every component has interfaces with other components and all component interfaces are sources of confusion. Smaller the component less the bugs. Large components reduce external interfaces but have complicated internal logic that may be difficult or impossible to understand. Testing can and should likewise be originated in to modular components, small, independent test cases have the virtue of easy repeatability.

Small versus Large : Programming in the large means constructing programs that consist of many components written by many different persons. Programming in the small is what we do for ourselves in the privacy of our own offices or as homework exercises in an undergraduate programming course. Qualitative changes occur with size and so must testing methods and quality criteria.

The Builder Versus the Buyer :

Just as programmers and testers can merge and become one, so can builder and buyer.

1. The builder, who designs for and is accountable to
2. The buyer, who pays for the system in the hope of profits from providing services to

A MODEL FOR TESTING

The Project: a real-world context characterized by the following model project.

Application: It is a real-time system that must provide timely responses to user requests for services. It is an online system connected to remote terminals.

Staff: The programming staff consists of twenty to thirty programmers. There staff is used for system's design.

Schedule: The project will take 24 months from the start of design to formal acceptance by the customer. Acceptance will be followed by a 6-month cutover period. Computer resources for development and testing will be almost adequate.

Specification: means requirements.

Acceptance Test: The system will be accepted only after a formal acceptance test. The application is not new, so part of the formal test already exists. At first the customer will intend to design the acceptance test, but later it will become the software design team's responsibility.

Personnel: Management's attitude is positive and knowledgeable about the realities of such projects.

Standards: Programming and test standards exist and are usually followed. They understand the role of interfaces and the need for interface standards.

Objectives: The system is the first of many similar systems that will be implemented in the future. No two will be identical, but they will have 75% of the code in common. Once installed, the system is expected to operate profitably for more than 10 years.

Source: One-third of the code is new, one-third extracted from a previous, reliable, but poorly documented system, and one-third is being rehosted

History: One programmer will quit before his components are tested. Another programmer will be fired before testing begins. A facility and/or hardware delivery problem will delay testing for several weeks and force second- and third-shift work. Several important milestones will slip but the delivery date will be met.

Overview

The process starts with a program embedded in an environment, such as a computer, an operating system, or a calling program. This understanding leads us to create three models:

- a model of the environment,
- a model of the program,
- a model of the expected bugs.

From these models we create a set of tests, which are then executed. The result of each test is either expected or unexpected. If unexpected, it may lead us to revise the test, our model or concept of how the program behaves, our concept of what bugs are possible, or the program itself. Only rarely would we attempt to modify the environment.

The Environment

A program's environment is the hardware and software required to make it run. For online systems the environment may include communications lines, other systems, terminals, and operators. The environment also includes all programs that interact with—and are used to create—the program under test, such as operating system, loader, linkage editor, compiler, utility routines. If testing reveals an unexpected result, we may have to change our beliefs (our model of the environment) to find out what went wrong. But sometimes the environment could be wrong: the bug could be in the hardware or firmware after all.

Bugs

A bad specification may lead us to mistake good behavior for bugs, and vice versa. An unexpected test result may lead us to change our notion of what a bug is—that is to say, our model of bugs. If you hold any of the following beliefs, then disabuse yourself of them because as long as you believe in such things you will be unable to test effectively and unable to justify the dirty tests most programs need.

Benign Bug Hypothesis: The belief that bugs are nice, tame, and logical. Only weak bugs have a logic to them and are amenable to exposure by strictly logical means. Subtle bugs have no definable pattern—they are wild cards.

Bug Locality Hypothesis: The belief that a bug discovered within a component affects only that component's behavior; that because of structure, language syntax, and data organization, the symptoms of a bug are localized to the component's designed domain. Only weak bugs are so localized. Subtle bugs have consequences that are arbitrarily far removed from the cause in time and/or space from the component in which they exist.

Control Bug Dominance : The belief that errors in the control structure of programs dominate the bugs. While many easy bugs, especially in components, can be traced to control-flow errors, data-flow and data-structure errors are as common. Subtle bugs that violate data-structure boundaries and data/code separation can't be found by looking only at control structures.

Code/Data Separation: The belief, especially in HOL programming, that bugs respect the separation of code and data.* Furthermore, in real systems the distinction between code and data can be hard to make, and it is exactly that blurred distinction that permit such bugs to exist.

Lingua Salvator Est : The hopeful belief that language syntax and semantics (e.g., structured coding, strong typing, complexity hiding) eliminates most bugs. True, good language features do help prevent the simpler component bugs but there's no statistical evidence to support the notion that such features help with subtle bugs in big systems.

Corrections Abide: The mistaken belief that a corrected bug remains corrected. Here's a generic counterexample. A bug is believed to have symptoms caused by the interaction of components A and B but the real problem is a bug in C, which left a residue in a data structure used by both A and B. The bug is "corrected" by changing A and B. Later, C is modified or removed and the symptoms of A and B recur. Subtle bugs are like that.

Silver Bullets: The mistaken belief that X (language, design method, representation, environment— name your own) grants immunity from bugs. Easy-to-moderate bugs may be reduced, but remember the pesticide paradox.

Sadism Suffices: The common belief, especially by independent testers, that a sadistic streak, low cunning, and intuition are sufficient to extirpate most bugs. You only catch easy bugs that way. Tough bugs need methodology and techniques, so read on.

Angelic Testers: The ludicrous belief that testers are better at test design than programmers are at code design.

Tests

Tests are formal procedures. Inputs must be prepared, outcomes predicted, tests documented, commands executed, and results observed; all these steps are subject to error. There is nothing magical about testing and test design that immunizes testers against bugs. An unexpected test result is as often cause by a test bug as it is by a real bug.* Bugs can creep into the documentation, the inputs, and the commands and becloud our observation of results. An unexpected test result, therefore, may lead us to revise the tests. Because the tests are themselves in an environment, we also have a mental model of the tests, and instead of revising the tests, we may have to revise that mental model.

Testing and Levels

We do three distinct kinds of testing on a typical software system: unit/ component testing, integration testing, and system testing. The objectives of each class is different and therefore, we can expect the mix of test methods used to differ. They are:

Unit, Unit Testing: A unit is the smallest testable piece of software, by which I mean that it can be compiled or assembled, linked, loaded, and put under the control of a test harness or driver. A unit is usually the work of one programmer and it consists of several hundred or fewer, lines of source code. Unit testing is the testing we do to show that the unit does not satisfy its functional specification and/or that its implemented structure does not match the intended design structure. When our tests reveal such faults, we say that there is a unit bug.

Component, Component Testing : A **component** is an **integrated aggregate** of one or more units. A unit is a component; a component with subroutines it calls is a component, etc. By this (recursive)definition, a component can be anything from a unit to an entire system.

Component testing is the testing we do to show that the component does not satisfy its functional specification and/or that its implemented structure does not match the intended design structure. When our tests reveal such problems, we say that there is a **component bug**.

Integration, Integration Testing : **Integration** is a *process* by which components are aggregated to create larger components. **Integration testing** is testing done to show that even though the components were individually satisfactory, as demonstrated by successful passage of component tests, the combination of components are incorrect or inconsistent. For example, components A and B have both passed their component tests. Integration

testing is aimed at showing inconsistencies between A and B. Examples of such inconsistencies are improper call or return sequences, inconsistent data validation criteria, and inconsistent handling of data objects. Integration testing should not be confused with testing integrated objects, which is just higher level component testing. Integration testing is specifically aimed at exposing the problems that arise from the combination of components. The sequence, then, consists of component testing for components A and B, integration testing for the combination of A and B, and finally, component testing for the “new” component (A,B).*

System, System Testing : A **system** is a big component. **System testing** is aimed at revealing bugs that cannot be attributed to components as such, to the inconsistencies between components, or to the planned interactions of components and other objects. System testing concerns issues and behaviors that can only be exposed by testing the entire integrated system or a major part of it. System testing includes testing for performance, security, accountability, configuration sensitivity, start-up, and recovery.

The Role of Models

Testing is a process in which we create mental models of the environment, the program, human nature, and the tests themselves. Each model is used either until we accept the behavior as correct or until the model is no longer sufficient for the purpose.

PLAYING POOL AND CONSULTING ORACLES

Playing Pool : Testing is like playing pool. There’s real pool and there’s kiddie pool and real testing and kiddie testing. In kiddie testing the tester says, after the fact, that the observed outcome of the test was the expected outcome. In real testing *the outcome is predicted and documented before the test is run.*

Oracles : An oracle is any program, process, or body of data that specifies the expected outcome of a set of tests as applied to a tested object. There are as many different kinds of oracles as there are testing concerns. The most common oracle is an input/outcome oracle—an oracle that specifies the expected outcome for a specified input.

Sources of Oracles

If every test designer had to analyze and predict the expected behavior for every test case for every component, then test design would be very expensive. The hardest part of test design is predicting the expected outcome, but we often have oracles that reduce the work. Here are some sources of oracles:

Kiddie Testing : Run the test and see what comes out. The observed outcome of the test was the expected outcome.

Regression Test Suites : Most of the tests you need will have been run on a previous version. Most of those tests should have the same outcome for the new version. Outcome prediction is therefore needed only for changed parts of components.

Purchased Suites and Oracles : Highly standardized software that (should) differ only as to implementation often has commercially available test suites and oracles. The most common examples are compilers for standard languages, communications protocols, and mathematical routines. As more software becomes standardized, more oracles will emerge as products and services.

Existing Program : A working, trusted program is an excellent oracle. The typical use is when the program is being rehosted to a new language, operating system, environment, configuration, or to some combination of these, with the intention that the behavior should not change as a result of the rehosting.

COMPLETE TESTING POSSIBLE?

If the objective of testing were to *prove* that a program is free of bugs, then testing not only would be practically impossible, but also would be theoretically impossible. Three different approaches can be used to demonstrate that a program is correct: tests based on structure, tests based on function, and formal proofs of correctness.

Functional Testing : Every program operates on a finite number of inputs. These inputs are of binary input stream. A complete functional test would consist of subjecting the program to all possible input streams. For each input the routine either accepts the stream and produces a correct outcome, accepts the stream and produces an incorrect outcome, or rejects the stream and tells us that it did so

Structural Testing : One should design enough tests to ensure that every path through the routine is exercised at least once. Right off that's impossible, because some loops might never terminate. A small routine can have millions or billions of paths, so total **path testing** is usually impractical, although it can be done for some routines.

Correctness Proofs : Formal proofs of correctness rely on a combination of functional and structural concepts. Requirements are stated in a formal language (e.g., mathematics), and each program statement is examined and used in a step of an inductive proof that the routine will produce the correct outcome for all possible input sequences.

THE TAXONOMY OF BUGS **CONSEQUENCES OF BUGS**

Importance of Bugs :

Frequency : There are different bug frequency statistics.

Correction Cost : Cost of correcting the bug. That cost is the sum of two factors: (1) the cost of discovery and (2) the cost of correction. These costs go up dramatically the later in the development cycle the bug is discovered. Correction cost also depends on system size. The larger the system the more it costs to correct the same bug.

Installation Cost : Installation cost depends on the number of installations; small for a single-user program, but how about a PC operating system bug? Installation cost can dominate all other costs—fixing one simple bug and distributing the fix could exceed the entire system's development cost.

Consequences : This is measured by the mean size of the awards made by juries to the victims of your bug. A reasonable metric for bug importance is:

$$\text{importance}(\$) = \text{frequency} * (\text{correction cost} + \text{installation cost} + \text{consequential cost})$$

How Bugs Affect Us -- Consequences

Bug consequences range from mild to catastrophic. Consequences should be measured in human rather than machine terms because it is ultimately for humans that we write programs.

Consequences :

1. **Mild** : The symptoms of the bug offend us aesthetically; a misspelled output or a misaligned printout.
2. **Moderate** : Outputs are misleading or redundant. The bug impacts the system's performance.
3. **Annoying** : The system's behavior, because of the bug, is dehumanizing. Names are truncated or arbitrarily modified. Bills for \$0.00 are sent. Operators must use unnatural command sequences and must trick the system into a proper response for unusual bug-related cases.
4. **Disturbing** : It refuses to handle legitimate transactions. The automatic teller machine won't give you money. My credit card is declared invalid.
5. **Serious** : It loses track of transactions; not just the transaction itself (your paycheck), but the fact that the transaction occurred. Accountability is lost.
6. **Very Serious** : Instead of losing your paycheck, the system credits it to another account or converts deposits into withdrawals. The bug causes the system to do the wrong transaction.
7. **Extreme** : The problems aren't limited to a few users or to a few transaction types. They are frequent and arbitrary instead of sporadic or for unusual cases.
8. **Intolerable** : Long-term, unrecoverable corruption of the data base occurs and the corruption is not easily discovered. Serious consideration is given to shutting the system down.
9. **Catastrophic** : The decision to shut down is taken out of our hands because the system fails.
10. **Infectious** : What can be worse than a failed system? One that corrupts other systems even though it does not fail in itself; that erodes the social or physical environment; that melts nuclear reactors or starts wars; whose influence, because of malfunction, is far greater than expected; a system that kills.

Flexible Severity Rather Than Absolutes

Many programmers, testers, and quality assurance workers have an absolutist attitude toward bugs. “Everybody knows that a program must be *perfect* if it’s to work: if there’s a bug, it *must* be fixed.” Metrics :

- **Correction Cost** : The cost of correcting a bug has almost nothing to do with symptom severity.
- Catastrophic, life-threatening bugs could be trivial to fix, whereas minor annoyances could require major rewrites to correct.
- **Context and Application Dependency** : The severity of a bug, for the same bug with the same symptoms, depends on context. For example, a roundoff error in an orbit calculation doesn’t mean much in a spaceship video game but it matters to real astronauts.
- **Creating Culture Dependency** : What’s important depends on the creators of the software and their cultural aspirations. Test tool vendors are more sensitive about bugs in their products than, say, games software vendors.
- **User Culture Dependency** : What’s important depends on the user culture. An R&D shop might accept a bug for which there’s a workaround; a banker would go to jail for that same bug; and naive users of PC software go crazy over bugs that pros ignore.
- **The Software Development Phase** : Severity depends on development phase. Any bug gets more severe as it gets closer to field use and more severe the longer it’s been around—more severe because of the dramatic rise in correction cost with time. Also, what’s a trivial or subtle bug to the designer means little to the maintenance programmer for whom all bugs are equally mysterious.

The Nightmare List and When to Stop Testing

Quantifying the Nightmare :

1. List your worst software nightmares. State them in terms of the symptoms they produce and how your user will react to those symptoms.
2. Convert the consequences of each nightmare into a cost. Usually, this is a labor cost for correcting the nightmare.
3. Order the list from the costliest to the cheapest and then discard the low-concern nightmares with which you can live.
4. Measure the kinds of bugs that are likely to create the symptoms expressed by each nightmare. Most bugs are simple goofs once you find and understand them. This can be done by bug design process.
5. For each nightmare, then, you’ve developed a list of possible causative bugs. Order that list by decreasing probability. Judge the probability based on your own bug statistics, intuition, experience, etc. The same bug type will appear in different nightmares. The importance of a bug type is calculated by multiplying the expected cost of the nightmare by the probability of the bug and summing across all nightmares:

$$\text{importance of bug type } i = \sum_{\text{all nightmares } j} C_j P_{(i \text{ in nightmare } j)}$$

6. Rank the bug types in order of decreasing importance to you.
7. Design tests (based on your knowledge of test techniques) and design your quality assurance inspection process by using the methods that are most effective against the most important bugs.
8. If a test is passed, then some nightmares or parts of them go away. If a test is failed, then a nightmares possible, but upon correcting the bug, it too goes away. Testing, then, gives you information you can use to revise your estimated nightmare probabilities.
9. Stop testing when the probability of all nightmares has been shown to be inconsequential as a result of hard evidence produced by testing.

General : There is no universally correct way to categorize bugs. This taxonomy is not rigid. Bugs are difficult to categorize. A given bug can be put into one or another category depending on its history and the programmer’s state of mind.

Requirements, Features, and Functionality Bugs

- **Requirements and Specifications :**

Requirements and the specifications developed from them can be incomplete, ambiguous, or self-contradictory. They can be misunderstood or impossible to understand. The specification may assume, but not mention, other specifications and prerequisites that are known to the specifier but not to the designer. And specifications that don't have these flaws may change while the design is in progress. Features are modified, added, and deleted. The designer has to hit a moving target and occasionally misses.

Requirements, especially as expressed in a specification (or often, as *not* expressed because there is no specification) are a major source of expensive bugs.

- **Feature Bugs :** Specification problems usually create corresponding feature problems. A feature can be wrong, missing, or superfluous. A missing feature or case is the easiest to detect and correct. A wrong feature could have deep design implications. Extra features were once considered desirable. "Free" features are rarely free. Removing the features might complicate the software, consume more resources, and foster more bugs.

- **Feature Interaction :** Features usually come in groups of related features. The features of each group and the interaction of features within each group are usually well tested. The problem is unpredictable interactions between feature groups or even between individual features.

Specification and Feature Bug Remedies

Short-Term Support : Specification languages facilitate formalization of requirements and (partial)* inconsistency and ambiguity analysis. With formal specifications, partially to fully automatic test case generation is possible. Generally, users and developers of such products have found them to be cost-effective.

Long-Term Support : Assume that we have a great specification language and that it can be used to create unambiguous, complete specifications with unambiguous, complete tests and consistent test criteria. A specification written in that language could theoretically be compiled into object code (ignoring efficiency and practicality issues). But this is just programming in HOL squared. The specification problem has been shifted to a higher level but not eliminated.

Testing Techniques

Most **functional test techniques**—that is, those techniques which are based on a behavioral description of software, such as **transaction flow testing**, **syntax testing**, **domain testing**, **logic testing**, and **state testing** are useful in testing functional bugs

Structural Bugs

Control and Sequence Bugs : Control and sequence bugs include paths left out, unreachable code, improper nesting of loops, loop-back or loop-termination criteria incorrect, missing process steps, duplicated processing, unnecessary processing, rampaging GOTO's, ill-conceived switches, spaghetti code, and worst of all, pachinko code. Control and sequence bugs at all levels are caught by testing, especially structural testing, more specifically, path testing, combined with a bottom-line functional test based on a specification. These bugs are partially prevented by language choice (e.g., languages that restrict control-flow options) and style, and most important, lots of memory.

Logic Bugs : Bugs in logic, especially those related to misunderstanding how case statements and logic operators behave singly and in combinations, include nonexistent cases, improper layout of cases, "impossible" cases that are not impossible, a "don't-care" case that matters, improper negation of a boolean expression (for example, using "greater than" as the negation of "less than"), improper simplification and combination of cases, overlap of exclusive cases, confusing "exclusive OR" with "inclusive OR."

Logic bugs are not really different in kind from arithmetic bugs. They are likelier than arithmetic bugs because programmers, like most people, have less formal training in logic at an early age than they do in arithmetic. The best defense against this kind of bug is a systematic analysis of cases. Logic-based testing is helpful.

Processing Bugs : Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection, and general processing. Many problems in this area are related to incorrect conversion from one data representation to another.

Initialization Bugs : Initialization bugs are common, and experienced programmers and testers know they must look for them. Both improper and superfluous initialization occur. The latter tends to be less harmful but can affect performance. Typical bugs are as follows: forgetting to initialize working space, registers, or data areas before first use or assuming that they are initialized elsewhere; a bug in the first value of a loop-control parameter; accepting an initial value without a validation check; and initializing to the wrong format, data representation, or type.

Data-Flow Bugs and Anomalies

A **data-flow anomaly** occurs when there is a path along which we expect to do something unreasonable with data, such as using an uninitialized variable, attempting to use a variable before it exists, modifying data and then not storing or using the result, or initializing twice without an intermediate use. Although part of data-flow anomaly detection can be done by the compiler based on information known at compile time, much can be detected only by execution and therefore is a subject for testing.

Data Bugs : Data bugs include all bugs that arise from the specification of data objects, their formats, the number of such objects, and their initial values. Data bugs are at least as common as bugs in code, but they are often treated as if they did not exist at all. Underestimating the frequency of data bugs is caused by poor bug accounting.

Dynamic Versus Static : Dynamic data are transitory. Whatever their purpose, they have a relatively short lifetime, typically the processing time of one transaction. A storage object may be used to hold dynamic data of different types, with different formats, attributes, and residues. The basic problem is leftover garbage in a shared resource. This can be handled in one of three ways:

- (1) cleanup after use by the user,
- (2) common cleanup by the resource manager, and
- (3) no cleanup. Static data are fixed in form and content. Whatever their purpose, they appear in the source code or data base, directly or indirectly, as, for example, a number, a string of characters, or a bit pattern. Static data need not be explicit in the source code.

Coding Bugs : Coding errors of all kinds can create any of the other kinds of bugs. Syntax errors are generally not important in the scheme of things if the source language translator has adequate syntax checking. The most common kind of coding bug, and often considered the least harmful, are documentation bugs (i.e., erroneous comments). Although many documentation bugs are simple spelling errors or the result of poor writing, many are actual errors—that is, misleading or erroneous comments.

Interface, Integration, and System Bugs

External Interfaces

The external interfaces are the means used to communicate with the world. These include devices, actuators, sensors, input terminals, printers, and communication lines. Other external interface bugs include: invalid timing or sequence assumptions related to external signals; misunderstanding external input and output formats; and insufficient tolerance to bad input data.

Internal Interfaces

Internal interfaces are in principle not different from external interfaces, but there are differences in practice because the internal environment is more controlled. The external environment is fixed and the system must adapt to it but the internal environment, which consists of interfaces with other components, can be negotiated. Internal interfaces have the same problems external interfaces have, as well as a few more that are more closely related to implementation details: protocol-design bugs, input and output format bugs, inadequate protection against corrupted data, wrong subroutine call sequence, call-parameter bugs, misunderstood entry or exit parameter values. The main objective of integration testing is to test all internal interfaces.

Hardware Architecture

Software bugs related to hardware architecture originate mostly from misunderstanding how the hardware works. Here are examples: paging mechanism ignored or misunderstood, address-generation error, I/O-device operation or instruction error, I/O-device address error, misunderstood device-status code, improper hardware simultaneity assumption, hardware race condition ignored, data format wrong for device, etc. The remedy

for hardware architecture and interface problems is two-fold: (1) good programming and testing and (2) centralization of hardware interface software in programs written by hardware interface specialists.

Operating System

Program bugs related to the operating system are a combination of hardware architecture and interface bugs, mostly caused by a misunderstanding of what it is the operating system does. and, of course, the operating system could have bugs of its own. Operating systems can lull the programmer into believing that all hardware interface issues are handled by it.

Software Architecture

Software architecture bugs are often the kind that are called “interactive.” Routines can pass unit and integration testing without revealing such bugs. Many of them depend on load, and their symptoms emerge only when the system is stressed. They tend to be the most difficult kind of bug to find and exhumed. Here is a sample of the causes of such bugs: assumption that there will be no interrupts, failure to block or unblock interrupts, assumption that code is reentrant or not reentrant, bypassing data interlocks, failure to close or open an interlock, etc.

Control and Sequence Bugs

System-level control and sequence bugs include: ignored timing; assuming that events occur in a specified sequence; starting a process before its prerequisites are met (e.g., working on data before all the data have arrived from disc); waiting for an impossible combination of prerequisites; not recognizing when prerequisites have been met; specifying wrong priority, program state, or processing level; missing, wrong, redundant, or superfluous process steps.

The remedy for these bugs is in the design. Highly structured sequence control is helpful. Specialized, internal, sequence-control mechanisms, such as an internal job control language, are useful.

Integration Bugs

Integration bugs are bugs having to do with the integration of, and with the interfaces between, presumably working and tested components. Most of these bugs result from inconsistencies or incompatibilities between components. All methods used to transfer data directly or indirectly between components and all methods by which components share data can host integration bugs and are therefore proper targets for integration testing. The communication methods include data structures, call sequences, registers, semaphores, communication links, protocols, and so on.

System Bugs

System bugs is a catch-all phrase covering all kinds of bugs that cannot be ascribed to components or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operating system.

Test and Test Design Bugs

Testing

Testers have no immunity to bugs. Tests, especially system tests, require complicated scenarios and databases. They require code or the equivalent to execute, and consequently they can have bugs. Test bugs are not software bugs, it's hard to tell them apart, and much labor can be spent making the distinction.

Test Criteria

The specification is correct, it is correctly interpreted and implemented, and a seemingly proper test has been designed; but the criterion by which the software's behavior is judged is incorrect or impossible.

Remedies

Test Debugging : The first remedy for test bugs is testing and debugging the tests. The differences between test debugging and program debugging are not fundamental. Test debugging is usually easier because tests, when properly designed, are simpler than programs and do not have to make concessions to efficiency. Also, tests tend to have a localized impact relative to other tests, and therefore the complicated interactions that usually plague software designers are less frequent. We have no magic prescriptions for test debugging—no more than we have for software debugging

Test Quality Assurance : Programmers have the right to ask how quality in independent testing and test design is monitored. Should we implement test testers and test—tester tests? This sequence does not converge. Methods for test quality assurance are discussed in *Software System Testing and Quality Assurance*

Test Execution Automation : The history of software bug removal and prevention is indistinguishable from the history of programming automation aids. Assemblers, loaders, compilers, and the like were all developed to reduce the incidence of programmer and/or operator errors. Test execution bugs are virtually eliminated by various test execution automation tools, many of which are discussed throughout this book. The point is that “manual testing” is self-contradictory. If you want to get rid of test execution bugs, get rid of manual execution.

Test Design Automation : Just as much of software development has been automated (what is a compiler, after all?) much test design can be and has been automated. For a given productivity rate, automation reduces bug count—be it for software or be it for tests.

UNIT II

FLOW GRAPHS AND PATH TESTING

Path Testing Basics

- **Path Testing:** Path Testing is the name given to a family of test techniques based on judiciously selecting a set of test paths through the program.
- If the set of paths are properly chosen then we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement has been executed at least once.
- Path testing techniques are the oldest of all structural test techniques.
- Path testing is most applicable to new software for unit testing. It is a structural technique.
- It requires complete knowledge of the program’s structure.
- It is most often used by programmers to unit test their own code.
- The effectiveness of path testing rapidly deteriorates as the size of the software aggregate under test increases.

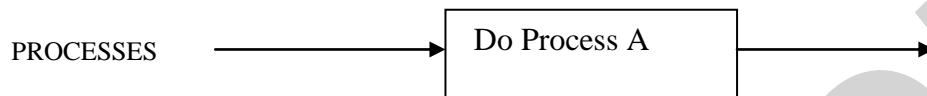
The Bug Assumption

- The bug assumption for the path testing strategies is that something has gone wrong with the software that makes it take a different path than intended.
- As an example “GOTO X” where “GOTO Y” had been intended.
- Structured programming languages prevent many of the bugs targeted by path testing: as a consequence the effectiveness for path testing for these languages is reduced and for old code in COBOL, ALP, FORTRAN and Basic, the path testing is indispensable.
 - Control Flow Graphs
- The control flow graph is a graphical representation of a program’s control structure. It uses the elements named process blocks, decisions, and junctions.
- The flow graph is similar to the earlier flowchart, with which it is not to be confused.
 - Flow Graph Elements
- A flow graph contains four different types of elements.
 - Process Block

- Decisions
- Junctions
- Case Statements

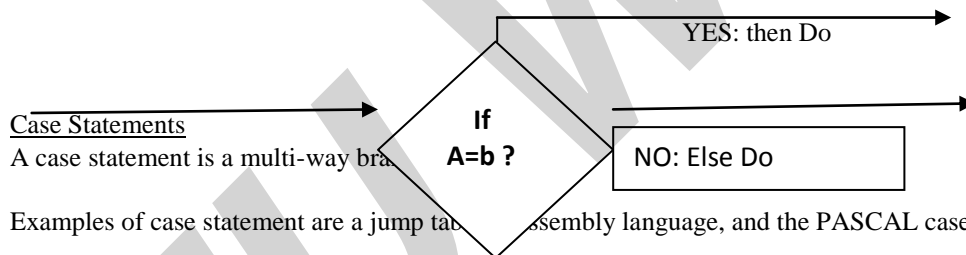
Process Block

- A process block is a sequence of program statements uninterrupted by either decisions or junctions. It is a sequence of statements such that if any one of statement of the block is executed, then all statement thereof are executed.
- Formally, a process block is a piece of straight line code of one statement or hundreds of statements.
- A process has one entry and one exit. It can consists of a single statement or instruction, a sequence of statements or instructions, a single entry/exit subroutine, a macro or function call, or a sequence of these.
-



Decisions

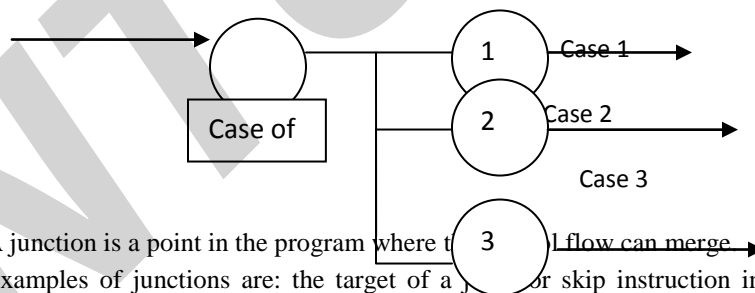
- A decision is a program point at which the control flow can diverge.
- Machine language conditional branch and conditional skip instructions are examples of decisions.
- Most of the decisions are two-way but some are three way branches in control flow.



Case Statements

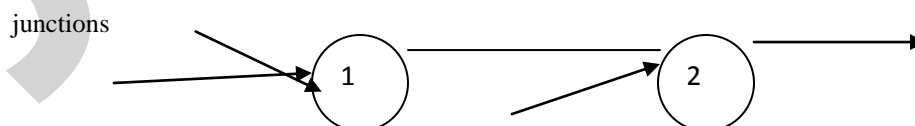
- A case statement is a multi-way branch.
- Examples of case statement are a jump table in assembly language, and the PASCAL case statement.

From the point of view of test design, there are no differences between Decisions and Case Statements



Junctions

- A junction is a point in the program where the control flow can merge.
- Examples of junctions are: the target of a jump or skip instruction in ALP, a label that is a target of GOTO.



Control Flow Graphs Versus Flow Charts

- A program's flow chart resembles a control flow graph.
- In flow graphs, we don't show the details of what is in a process block.

- In flow charts every part of the process block is drawn.
- The flowchart focuses on process steps, where as the flow graph focuses on control flow of the program
- The act of drawing a control flow graph is a useful tool that can help us clarify the control flow and data flow issues.

Notational Evolution

- The control flow graph is simplified representation of the program's structure.
- The notation changes made in creation of control flow graphs:
- The process boxes weren't really needed. There is an implied process on every line joining junctions and decisions.
- We don't need to know the specifics of the decisions, just the fact that there is a branch.
- The specific target label names aren't important-just the fact that they exist. So we can replace them by simple numbers.

Program – Example

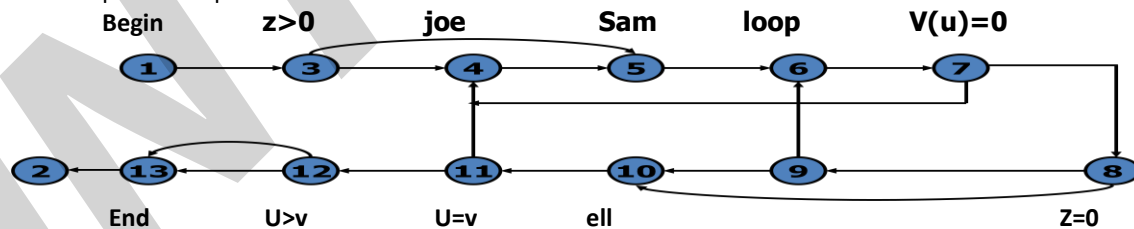
CODE (PDL)

```

      INPUT X,Y
      Z:=X+Y
      V:=X-Y
      IF Z>0 GOTO SAM
JOE:   Z:=Z-1
SAM:   Z:=Z+V
      FOR U=0 TO Z
      V(U),U(V) :=(Z+V)*U
      IF V(U)=0 GOTO JOE
Z:=Z-1
      IF Z=0 GOTO ELL
      U:=U+1
      NEXT U
      V(U-1) :=V(U+1)+U(V-1)
ELL:   V(U+U(V)) :=U+V
      IF U=V GOTO JOE
      IF U>V THEN U:=Z
      Z:=U
      END

```

Flow Graph – Example



- Although graphical representations of flowgraphs are revealing the details of the control flow inside a program they are often inconvenient.
- In linked list representation, each node has a name and there is an entry on the list for each link in the flow graph. only the information pertinent to the control flow is shown.

Example - Linked list representation

```

1 (begin)      : 3
2 (end) : exit, no outlink
3 (z>0)  : 4 (false)

```

```

: 5 (true)
4 (joe)      : 5
5 (sam) : 6
6 (loop): 7
7 (v(u)=0?)) : 4 (true)
: 8 (false)
8 (z=0?)    : 9 (false)
: 10 (true)
9 (u=z?)    : 6 (false)=loop
: 10 (true)=ell
10 (ell)    : 11
11 (u=v?)   : 4 (true)=joe
: 12 (false)
12 (u>v?)   : 13 (true)
: 13 (false)
13          : 2 (end)

```

Flowgraph – Program Correspondence

- A flow graph is a pictorial representation of a program and not the program itself, just as a topographic map.
- You cant always associate the parts of a program in a unique way with flowgraph pats because many program structures, such as if-then-else constructs, consists of a combination of decisions, junctions, and processes.
- The translation from a flowgraph element to a statement and vice versa is not always unique.
- An improper translation from flowgraph to code during coding can lead to bugs, and improper translation during the test design lead to missing test cases and causes undiscovered bugs.

Flow graph and flowchart generation

- Flow charts can be
- Hand written by the programmer
- Automatically produced by a flowcharting program based on a mechanical analysis of the source code
- Semi automatically produced by a flow charting program based in part on structural analysis of the source code and in part on directions given by the programmer
- There are relatively few control flow graph generators.

Path Testing- paths, nodes and links

- **Path:** a path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same junction, decision, or exit.
- A path may go through several junctions, processes, or decisions, one or more times.
- Paths consists of **segments**.
- The **segment** is a link – a single process that lies between two nodes.
- A **path segment** is succession of consecutive links that belongs to some path.
- The **length of path** measured by the number of links in it and not by the number of the instructions or statements executed along that path.
- The **name of a path** is the name of the nodes along the path.

Fundamental Path Selection Criteria

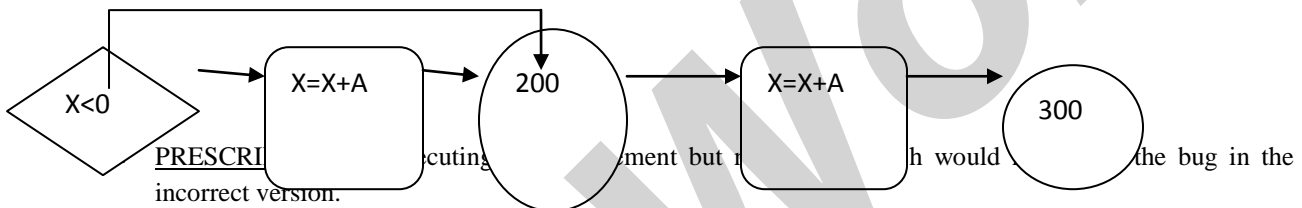
- There are many paths between the entry and exit of a typical routine.
- Every decision doubles the number of potential paths. And every loop multiplies the number of potential paths by the number of different iteration values possible for the loop.
- Defining complete testing:
 - Exercise every path from entry to exit
 - Exercise every statement or instruction at least once
 - Exercise every branch and case statement, in each direction at least once
- If prescription 1 is followed then 2 and 3 are automatically followed. But it is impractical for most routines. It can be done for the routines that have no loops, in which it is equivalent to 2 and 3 prescriptions.

Path selection criteria – examplePRESCRIPTION 1:

```

If ( X, LT,0) goto200
X=X+A
200 X=X+A
300 CONTINUE

```

PRESCRIPTION 1:

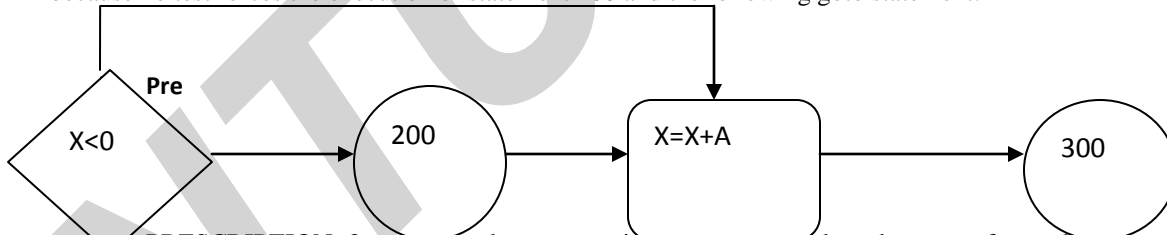
```

If ( X, LT,0) goto200
200 X=X+A
300 CONTINUE

```

Negative value produces the correct answer. Every statement can be executed, but if the test cases do not force each branch to be taken, the bug can remain hidden.

because no test forces the execution of statement 100 and the following goto statement.



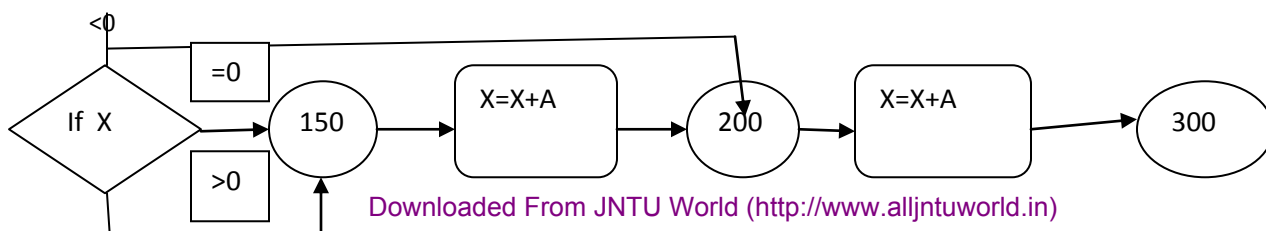
PRESCRIPTION 3: test based on executing each branch but does not force the execution of all statements;

```

If (X) 200,150,150
100 X=X+A
goto 100
150 X=X+A
200 X=X+A
300 CONTINUE

```

The hidden loop around label 100 is not revealed by tests based on prescription 3 alone because no test forces the execution of statement 100 and the following goto statement.



Path Testing Criteria

- Any testing strategy based on paths must at least both exercise every instruction and take branches in all directions.
- A set of tests that does this is not complete in an absolute sense, but it is complete in the sense that anything less must leave something untested.
- So we have explored three different testing criteria or strategies out of a potentially infinite family of strategies.
 1. Path testing
 2. Statement testing
 3. Branch testing
- Path Testing (Pinf)
- Path Testing: execute all possible control flow paths through the program: typically, this is restricted to all possible entry/exit paths through the program.
- If we achieve this prescription, we are said to have achieved **100% path coverage**. This is the strongest criterion in the path testing strategy family: it is generally impossible to achieve.

Statement Testing (P1)

- Execute all statements in the program at least once under some test. If we do enough tests to achieve this, we are said to have achieved **100% statement coverage**.
- An alternate equivalent characterization is to say that we have achieved **100% node coverage**. We denote this by C1.
- This is the weakest criterion in the family: testing less than this for new software is unconscionable and should be criminalized.

Branch Testing (P2)

- Execute enough tests to assure that every branch alternative has been exercised at least once under some test.
- If we do enough tests to achieve this prescription, then we have achieved **100% branch coverage**.
- An alternative characterization is to say that we have achieved **100% link coverage**.
- For structured software, branch testing and therefore branch coverage strictly includes statement coverage.
- We denote branch coverage by C2.
- Common sense and strategies
- Branch and statement coverage are accepted today as the minimum mandatory testing requirement.
- The question “ why not use a judicious sampling of paths?”, what’s wrong with leaving some code, untested?” is ineffectual in the view of common sense and experience since:
 1. not testing a piece of a code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs.
 2. the high probability paths are always thoroughly tested if only to demonstrate that the system works properly.

Which paths to be tested?

- You must pick enough paths to achieve C1+C2.
- The question of what is the fewest number of such paths is interesting to the designer of test tools that help automate the path testing, but it is not crucial to the pragmatic design of tests.
- Path Selection – Example
- Draw the control flow graph on a single sheet of paper.
- Make several copies – as many as you will need for coverage (C1+C2) and several more.
- Use a yellow highlighting marker to trace paths. Copy the paths onto a master sheets.
- Continue tracing paths until all lines on the master sheet are covered, indicating that you appear to have achieved C1+C2.
- As you trace the paths, create a table that shows the paths, the coverage status of each process, and each decision. The above paths lead to the following table.

Tracing table for Path Selection

Paths	Decisions				Process links
	4	6	7	9	
abcde	yes	Yes			1 1 1 1 1
abhkgde	No	Yes		No	1 1 1 1 1 1
abhlbcde	No,yes	Yes		Yes	1 1 1 1 1 1 1
abcdfjgde	Yes	No,yes	Yes		1 1 1 1 1 1 1
Abcdfmibcde	Yes	No,yes	no		1 1 1 1 1 1 1 1

- After you have traced a covering path set on the master sheet and filled in the table for every path. Check the following:
- Does every decision have a YES and NO in its column? (C2)

- Has every case of all case statements been marked? (C2)
- Is every three way branch covered? (C2)
- Is every link covered at least once? (C1)

Revised path selection rules

- Pick the simplest, functionally sensible entry/exit path.
- Pick additional paths as small variation from previous paths. Pick paths that do not have loops rather than paths that do. Favor short paths that make sense over paths that don't.
- Pick additional paths that have no obvious functional meaning only if it's necessary to provide coverage.
- Be comfortable with your chosen paths. Play your hunches and give your intuition free reign as long as you achieve C1+C2.
- Don't follow rules slavishly-except for coverage.

Loops

- There are only three kinds of loops with respect to path testing:
 - Nested loops
 - Concatenated loops
 - Horrible loops
- Cases for a single loop
- A single loop can be covered with two cases:
 - Looping
 - Not looping
- Experience shows that loop related bugs are not discovered by C1+c2.
- Bugs lurk in some corners and congregate at boundaries-in the case of loops at or around the minimum and maximum number of times the loop can be iterated.

Case 1: single loop, Zero minimum, N maximum, No Excluded Values

- Try bypassing the loop. If you can't you either have a bug, or zero is not the minimum and you have the wrong case.
- Could the loop control variable be negative
- One pass through the loop
- A typical number of iterations, unless covered by a previous test.
- One less than the maximum number of iterations
- The maximum number of iterations
- Attempt one more than the maximum number of iterations.

Case 2: single loop, Non Zero Minimum, No Excluded Values

- Try one less than the expected minimum
- The minimum number of iterations
- One more than the minimum number of iterations
- Once, unless covered by a previous test
- Twice, unless covered by a previous test
- A typical value

- One less than the maximum value
- The maximum number of iterations
- Attempt one more than the maximum number of iterations

Case 3: single loops with excluded values

- Treat single loops with excluded values as two sets of tests consisting of loops without excluded values, such as case 1 and 2 above
- Example, the total range of the loop control variable was 1 to 20, but that values 7,8,9,10 were excluded. The two sets of tests are 1-6 and 11-20.
- The test cases to attempt would be 0,1,2,4,6,7 for the first range and 10,11,15,19,20,21 for the second range.

Nested Loops

- The number of tests to be performed on nested loops will be the exponent of the tests performed on single loops.
- As we cannot always afford to test all combinations of nested loops' iterations values. Here's a tactic use to discard some of these values:
 - Start at the inner most loop. Set all the outer loops to their minimum values.
 - Test the minimum, minimum+1, typical, maximum-1, and maximum for the innermost loop, while holding the outer loops at their minimum iteration parameter values. Expand the tests as required for out of range and excluded values
 - If you've done the outmost loop, GOTO step5, else move out one loop and set it up as in step2 with all other loops set to typical values
 - Continue outward in this manner until all loops have been covered
 - Do the five cases for all loops in the nest simultaneously.

Concatenated loops

- Concatenated loops fall between single and nested loops with respect to test cases. Two loops are concatenated if it's possible to reach one after exiting the other while still on a path from entrance to exit.
- If the loops cannot be on the same path, then they are not concatenated and can be treated as individual loops.

Horrible loops

- A horrible loop is a combination of nested loops, the use of code that jumps into and out of loops, intersecting loops, hidden loops, and cross connected loops.
- makes iteration value selection for test cases an awesome and ugly task, which is another reason such structures should be avoided.

Loop Testing Time

- Any kind of loop can lead to long testing time, especially if all the extreme value cases are to attempted (Max-1, Max, Max+1).
- This situation is obviously worse for nested and dependent concatenated loops.
- Consider nested loops in which testing the combination of extreme values lead to long test times. Several options to deal with:
 - Prove that the combined extreme cases are hypothetically possible, they are not possible in the real world
 - Put in limits or checks that prevent the combined extreme cases. Then you have to test the software that implements such safety measures.

Predicates, paths predicates, and achievable paths

- Predicate: The logical function evaluated at a decision is called Predicate.
- Some examples are: $A > 0$, $x + y \geq 90$
- The direction taken at a decision depends on the value of decision variable.

Path Predicates

- A predicate associated with a path is called a Path Predicate.
- For example “x is greater than zero”
 - “ $x + y \geq 90$ ”
 - “w is either negative or equal to 10 is true”

Is a sequence of predicates whose truth values will cause the routine to take a specific path

Multiway Branches

- The path taken through a multiway branch such as a computed GOTO'S, case statement, or jump tables cannot be directly expressed in TRUE/FALSE terms.
- Although, it is possible to describe such alternatives by using multi valued logic, an expedient is to express multiway branches as an equivalent set of if..then..else statements.
- For example a three way case statement can be written as

If case=1 DO A1 ELSE

(IF

Case=2 DO A2 ELSE DO A3 ENDIF)ENDIF.

Inputs

- In testing, the word input is not restricted to direct inputs, such as variables in a subroutine call, but includes all data objects referenced by the routine whose values are fixed prior to entering it.
- For example, inputs in a calling sequence, objects in a data structure, values left in registers, or any combination of object types.
- The input for a particular test is mapped as a one dimensional array called as an

Input Vector.Predicate Expressions Predicate Interpretation

- The simplest predicate depends only on input variables.
- For example if x_1, x_2 are inputs, the predicate might be $x_1 + x_2 \geq 7$, given the values of x_1 and x_2 the direction taken through the decision is based on the predicate is determined at input time and does not depend on processing.
- Another example, assume a predicate $x_1 + y \geq 0$ that along a path prior to reaching this predicate we had the assignment statement $y = x_2 + 7$. although our predicate depends on processing we can substitute the symbolic expression for y to obtain an equivalent predicate $x_1 + x_2 + 7 \geq 0$.
- The act of symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called **predicate interpretation**.
- Some times the interpretation may depend on the path; for example,
- Input x

On x goto a,b,c

a: z:=7 @ goto hem

b: z:=-7 @ goto hem

c: z:=0 @ goto hem

.....

HEM: do some thing

.....

HEN: if y+z>0 goto ELL else goto EMM

- The predicate interpretation at HEN depends on the path we took through the first multiway branch. It yields for the three cases respectively, if $Y+7>0$, $y-7>0$, $y>0$.
- The path predicates are the specific form of the predicates of the decisions along the selected path after interpretation.
 - Independence of Variables and Predicates
- The path predicates take on truth values based on the values of input variables, either directly or indirectly.
- If a variable's value does not change as a result of processing, that variable is **independent** of the processing.
- If the variable's value can change as a result of the processing the variable is **process dependent**.
- A **predicate** whose truth value can change as a result of the processing is said to be **process dependent** and one whose truth value does not change as a result of the processing is **process independent**.
- Process dependence of a predicate does not always follow from dependence of the input variables on which that predicate is based.

Correlation of Variables and Predicates

- Two variables are correlated if every combination of their values cannot be independently specified.
- Variables whose values can be specified independently without restriction are called uncorrelated.
- A pair of predicates whose outcomes depend on one or more variables in common are said to be correlated predicates.

Path Predicate Expressions

- A path predicate expression is a set of boolean expressions, all of which must be satisfied to achieve the selected path.
- Example: $X1+3 X2=17 \geq 0$

$$\begin{array}{l} X3=17 \\ X4-X1 \geq 14 X2 \end{array}$$
- Any set of input values that satisfy all of the conditions of the path predicate expression will force the routine to the path.
- Some times a predicate can have an OR in it.
 - Example if $X5>0$. OR. $X6<0$
 - This will give two set of expressions, either of which if solved forces the path.

A: $X5>0$	E: $X6<0$
B: $X1+3 X2=17 \geq 0$	B: $X1+3 X2=17 \geq 0$
C: $X3=17$	C: $X3=17$
D: $X4-X1 \geq 14 X2$	D: $X4-X1 \geq 14 X2$

- Boolean algebra notation to denote the boolean expression:

$$ABCD+EB CD=(A+E)BCD$$

Predicate Coverage

- Compound Predicate: predicates of the form A OR B, A AND B and more complicated boolean expressions are called as compound predicates.
- Some times even a simple predicate becomes compound after interpretation. Example: the predicate if $(x=17)$ whose opposite branch is if $x \neq 17$ which is equivalent to $x > 17$. Or. $x < 17$.
- Predicate coverage is being the achieving of all possible combinations of truth values corresponding to the selected path have been explored under some test.
- As achieving the desired direction at a given decision could still hide bugs in the associated predicates.

Testing Blindness

- Testing Blindness is a pathological situation in which the desired path is achieved for the wrong reason.
- There are three types of Testing Blindness:
 - Assignment Blindness
 - Equality Blindness
 - Self Blindness

Assignment Blindness

- Assignment blindness occurs when the buggy predicate appears to work correctly because the specific value chosen for an assignment statement works with both the correct and incorrect predicate.
- For example,

Correct
X:=7

buggy
x:=7

.....
If $y > 0$ then

.....
if $x + y > 0$ then

If the test case sets $y=1$ the desired path is taken in either case, but there is still a bug.

Equality Blindness

- Equality blindness occurs when the path selected by a prior predicate results in a value that works both for the correct and buggy predicate.
- For example,

Correct
If $y := 2$ then

buggy

$y := 2$ then

if

.....
If $x + y > 3$ then

.....
if $x > 1$ then

- The first predicate if $y=2$ forces the rest of the path, so that for any positive value of x. the path taken at the second predicate will be the same for the correct and buggy version.

Self Blindness

- Self blindness occurs when the buggy predicate is a multiple of the correct predicate and as a result is indistinguishable along that path.
- For example,

Correct
X:=a

buggy
x:=a

.....
If $x - 1 > 0$ then

.....
if $x + a - 2 > 0$ then

The assignment ($x=a$) makes the predicates multiples of each other, so the direction taken is the same for the correct and buggy version.

Path Sensitizing

- We want to select and test enough paths to achieve a satisfactory notion of test completeness such as C1 and C2.
- Extract the programs control flowgraph and select a set of tentative covering paths.
- For any path in that set, interpret the predicates along the path as needed to express them in terms of the input vector. In general individual predicates are compound or may become compound as a result of interpretation.
- Trace the path through, multiplying the individual compound predicates to achieve a boolean expression such as $(A+BC) (D+E) (FGH) (IJ) (K) (L)$
- Multiply out the expression to achieve a sum of products form:
 - $ADFGHIJKL+AEFGHIJKL+BCDFGHIJKL+BCEFGHIJKL$
- Each product term denotes a set of inequalities that if solved will yield an input vector that will drive the routine along the designated path.
- Solve any one of the inequality sets for the chosen path and you have found a set of input values for the path.
- If you can find a solution, then the path is achievable.
- If you cant find a solution to any of the sets of inequalities, the path is un achievable.
- The act of finding a set of solutions to the path predicate expression is called PATH SENSITIZATION

Heuristic Procedures for Sensitizing Paths

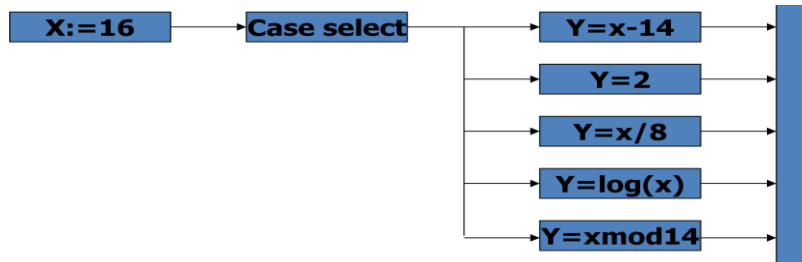
This is a workable approach, instead of selecting the paths without considering how to sensitize, attempt to choose a covering path set that is easy to sensitize and pick hard to sensitize paths only as you must to achieve coverage.

- Identify all variables that affect the decision.
- Classify the predicates as dependent or independent.
- Start the path selection with un correlated, independent predicates.
- If coverage has not been achieved using independent uncorrelated predicates, extend the path set using correlated predicates.
- If coverage has not been achieved extend the cases to those that involve dependent predicates.
- Last, use correlated, dependent predicates.
 - Path Instrumentation
- Path instrumentation is what we have to do to confirm that the outcome was achieved by the intended path.
- Types of instrumentation methods are:
 - An interpretive trace program

- Traversal marker or link marker
- The two link marker method

Co- incidental Correctness

The coincidental correctness stands for achieving the desired outcome for wrong reason.



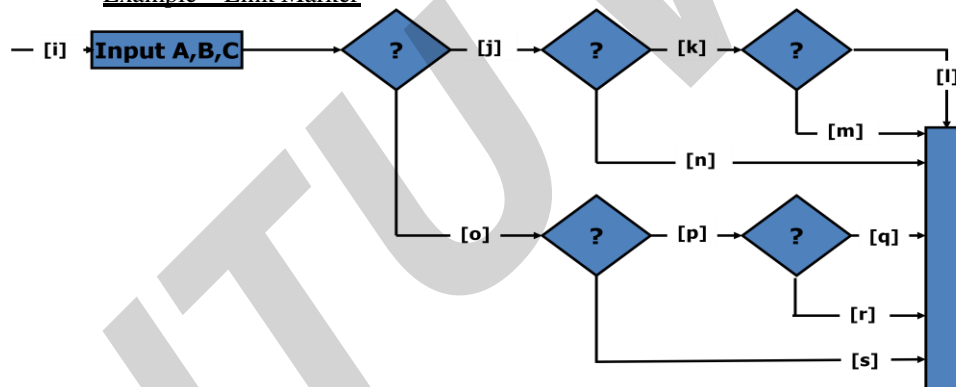
Interpretive trace program

- An interpretive trace program is one that executes every statement in order and records the intermediate values of all calculations, the statement labels traversed etc.,

Traversal marker or link marker

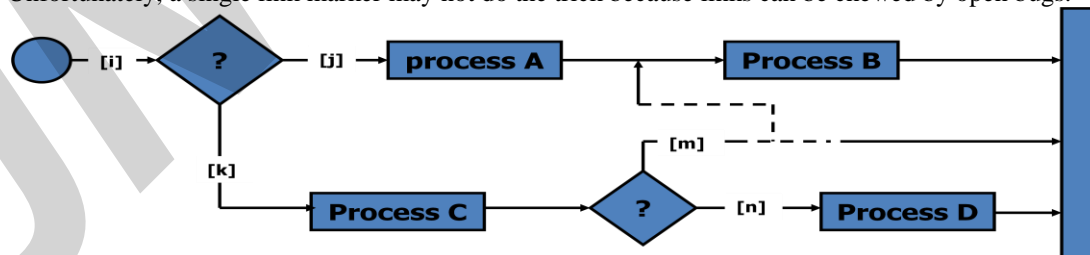
- A simple and effective form of instrumentation is called a traversal marker or link marker.
- Name every link by a lower case letter.
- Instrument the links so that the link's name is recorded when the link is executed.
- The succession of letters produced in going from the routine's entry to its exit should, if there are no bugs, exactly correspond to the path name.

Example – Link Marker



Why single link markers aren't enough

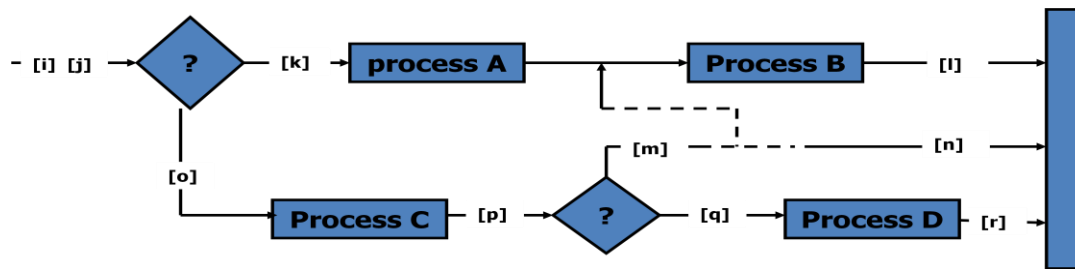
Unfortunately, a single link marker may not do the trick because links can be chewed by open bugs.



Two link marker method

- The solution to the problem of single link marker method is to implement two markers per link: one at the beginning of each link and one at the end.
- The two link markers now specify the path name and confirm both the beginning and end of the link.

Example - Two link marker



UNIT – III

TRANSACTION FLOW TESTING

Transaction flow Graphs

- Transaction flows are introduced as a representation of a system's processing.
- The methods that were applied to control flow graphs are then used for functional testing.
- Transaction flows and transaction flow testing are to the independent system tester what control flows are to the programmer.
- The transaction flow graph is to create a behavioral model of the program that leads to functional testing.
- The transaction flow graph is a model of the structure of the system's behavior (functionality).

Transaction Flows

- A transaction is a unit of work seen from a system user's point of view.
- A transaction consists of a sequence of operations, some of which are performed by a system, persons or devices that are outside of the system.
- Transaction begin with Birth-that is they are created as a result of some external act.
- At the conclusion of the transaction's processing, the transaction is no longer in the system.

Example of a Transaction

A transaction for an online information retrieval system might consist of the following steps or tasks:

- Accept input (tentative birth)
- Validate input (birth)
- Transmit acknowledgement to requester
- Do input processing
- Search file
- Request directions from user
- Accept input
- Validate input
- Process request
- Update file
- Transmit output
- Record transaction in log and clean up (death)

Complications in transaction flow graphs

- In simple cases, the transactions have a unique identity from the time they're created to the time they're completed.
- In many systems the transactions can give birth to others, and transactions can also merge.

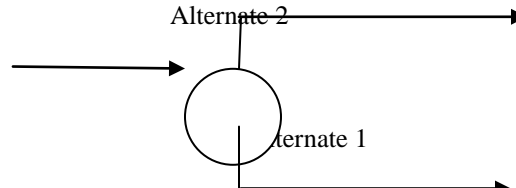
Births

There are three different possible interpretations of the decision symbol, or nodes with two or more out links.

- Decision
- Biosis
- Mitosis

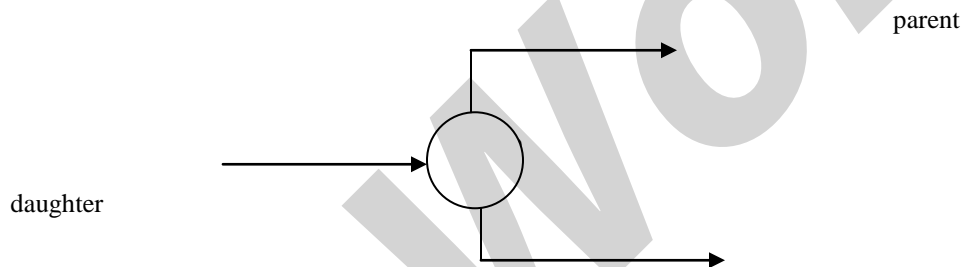
Decision

Here the transaction will take one alternative or the other alternative but not both



Biosis

Here the incoming transaction gives birth to a new transaction, and both transaction continue on their separate paths, and the parent retains its identity.



Mitosis

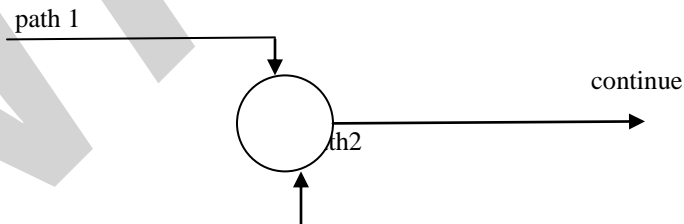
Here the parent transaction is destroyed and two new transactions are created.

Mergers

- Transaction flow junction points are potentially as troublesome as transaction flow splits.
- There are three types of junctions:
 - Ordinary Junction
 - Absorption
 - conjugation

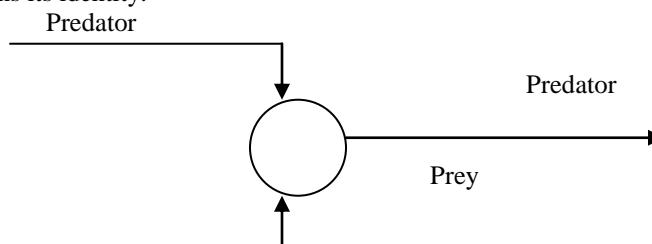
Ordinary junction

An ordinary junction which is similar to the junction in a control flow graph. A transaction can arrive either on one link or the other.



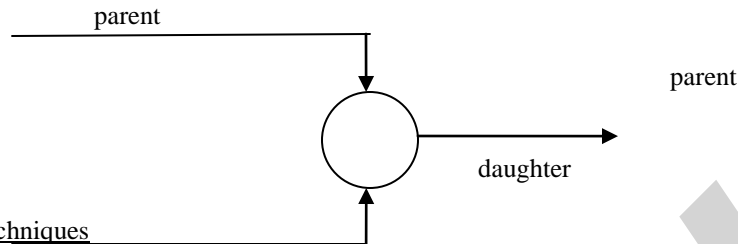
Absorption

In absorption case, the predator transaction absorbs prey transaction. The prey gone but the predator retains its identity.



Conjugation

In conjugation case, the two parent transactions merge to form a new daughter. In keeping with the biological flavor this case is called as conjugation.

Transaction flow testing techniques

- Get the transaction flows:
- Complicated systems that process a lot of different, complicated transactions should have explicit representations of the transactions flows, or the equivalent.
- Transaction flows are like control flow graphs, and consequently we should expect to have them in increasing levels of detail.
- The system's design documentation should contain an overview section that details the main transaction flows.
- Detailed transaction flows are a mandatory pre requisite to the rational design of a system's functional test.

Inspections, reviews, walkthroughs

1. Transaction flows are natural agenda for system reviews or inspections.
2. In conducting the walkthroughs, you should:
 - a. discuss enough transaction types to account for 98%-99% of the transaction the system is expected to process.
 - b. discuss paths through flows in functional rather than technical terms.
 - c. Ask the designers to relate every flow to the specification and to show how that transaction, directly or indirectly, follows from the requirements.
3. Make transaction flow testing the corner stone of system functional testing just as path testing is the corner stone of unit testing.
4. Select additional flow paths for loops, extreme values, and domain boundaries.
5. Design more test cases to validate all births and deaths.
6. Publish and distribute the selected test paths through the transaction flows as early as possible so that they will exert the maximum beneficial effect on the project.

Path Selection

- Select a set of covering paths (c1+c2) using the analogous criteria you used for structural path testing.
- Select a covering set of paths based on functionally sensible transactions as you would for control flow graphs.
- Try to find the most tortuous, longest, strangest path from the entry to the exit of the transaction flow.

Sensitization

- Most of the normal paths are very easy to sensitize-80% - 95% transaction flow coverage (c1+c2) is usually easy to achieve.
- The remaining small percentage is often very difficult.
- Sensitization is the act of defining the transaction. If there are sensitization problems on the easy paths, then bet on either a bug in transaction flows or a design bug.

Instrumentation

- Instrumentation plays a bigger role in transaction flow testing than in unit path testing.

- The information of the path taken for a given transaction must be kept with that transaction and can be recorded by a central transaction dispatcher or by the individual processing modules.
- In some systems such traces are provided by the operating systems or a running log.

DATA FLOW TESTING

Data Flow Testing Basics

- Data flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects.
- For example, pick enough paths to assure that every data object has been initialized prior to use or that all defined objects have been used for something.

Motivation

“ it is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test, one should not feel confident about a program without having seen the effect of using the value produced by each and every computation.”

Data flow machines

- There are two types of data flow machines with different architectures.
- Von Neumann machines
- Multi-instruction, multi-data machines (MIMD)

Von Neumann machine Architecture

- Most computers today are von-neumann machines.
- This architecture features interchangeable storage of instructions and data in the same memory units.
- The Von Neumann machine Architecture executes one instruction at a time in the following, micro instruction sequence:
 - Fetch instruction from memory
 - Interpret instruction
 - Fetch operands
 - Process or Execute
 - Store result
 - Increment program counter
 - GOTO 1

Multi-instruction, multi-data machines (MIMD) Architecture

- These machines can fetch several instructions and objects in parallel.
- They can also do arithmetic and logical operations simultaneously on different data objects.
- The decision of how to sequence them depends on the compiler.

Data Flow Graphs

- The data flow graph is a graph consisting of nodes and directed links.
- We will use an control graph to show what happens to data objects of interest at that moment.
- Our objective is to expose deviations between the data flows we have and the data flows we want.

Data Object State and Usage

- Data objects can be created, killed and used.
- They can be used in two distinct ways:
 - In a calculation
 - As a part of a control flow predicate
- The following symbols denote these possibilities.
 - d- defined, created, initialized, etc.,

- k- killed, undefined, released.
- u- used for some thing.
 - c- used in calculations
 - p- used in a predicate

Defined

- an object is defined explicitly when it appears in a data declaration.
- Or implicitly when it appears on the left hand side of the assignment.
- It is also to be used to mean that a file has been opened.
- A dynamically allocated object has been allocated.
- Something is pushed on to the stack.
- A record written.

Killed or Undefined

- An object is killed on undefined when it is released or otherwise made unavailable.
- When its contents are no longer known with certitude.
- Release of dynamically allocated objects back to the availability pool.
- Return of records
- The old top of the stack after it is popped.
- An assignment statement can kill and redefine immediately.

Usage

- A variable is used for computation (c) when it appears on the right hand side of an assignment statement.
- A file record is read or written.
- It is used in a Predicate (p) when it appears directly in a predicate.

Data Flow Anomalies

- An anomaly is denoted by a two-character sequence of actions.
- For example, ku means that the object is killed and then used, where as dd means that the object is defined twice without an intervening usage.
- What is an anomaly is depend on the application.
- There are nine possible two-letter combinations for d, k and u. some are bugs, some are suspicious, and some are okay.
- dd- probably harmless but suspicious. Why define the object twice without an intervening usage.
- dk- probably a bug. Why define the object without using it.
- du- the normal case. The object is defined and then used.
- kd- normal situation. An object is killed and then redefined.
- kk- harmless but probably buggy. Did you want to be sure it was really killed?
- ku- a bug. the object doesnot exist.
- ud- usually not a bug because the language permits reassignment at almost any time.
- uk- normal situation.
- uu- normal situation.
- In addition to the two letter situations there are six single letter situations.
- We will use a leading dash to mean that nothing of interest (d,k,u) occurs prior to the action noted along the entry-exit path of interest.
- A trailing dash to mean that nothing happens after the point of interest to the exit.
- -k: possibly anomalous because from the entrance to this point on the path, the variable had not been defined. We are killing a variable that does not exist.
- -d: okay. This is just the first definition along this path.

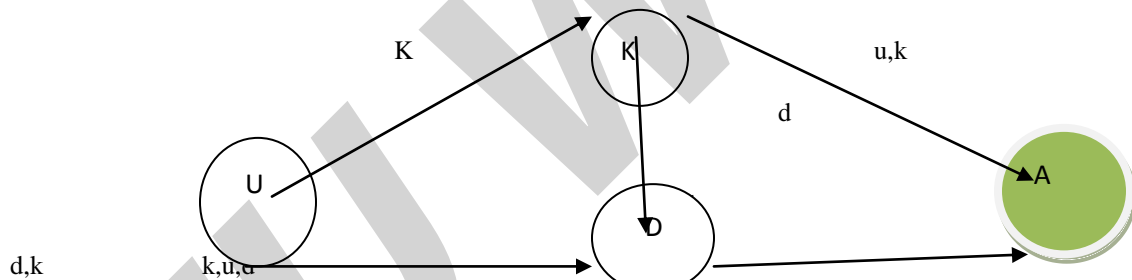
- -u: possibly anomalous. Not anomalous if the variable is global and has been previously defined.
- k-: not anomalous. The last thing done on this path was to kill the variable.
- d-: possibly anomalous. The variable was defined and not used on this path. But this could be a global definition.
- u-: not anomalous. The variable was used but not killed on this path. Although this sequence is not anomalous, it signals a frequent kind of bug. If d and k mean dynamic storage allocation and return respectively, this could be an instance in which a dynamically allocated object was not returned to the pool after use.

Data-Flow Anomaly State Graph

- Data flow anomaly model prescribes that an object can be in one of four distinct states:
 - K- undefined, previously killed, does not exist
 - D- defined but not yet used for anything
 - U- has been used for computation or in predicate
- These capital letters (K,D,U,A) denote the state of the variable and should not be confused with the program action, denoted by lower case letters.

Unforgiving Data-Flow Anomaly State Graph

- Unforgiving model, in which once a variable becomes anomalous it can never return to a state of grace.



Static Vs Dynamic Anomaly Detection

- Static analysis is analysis done on source code without actually executing it.
- For example: source code syntax error detection is the static analysis result.
- Dynamic analysis is done on the fly as the program is being executed and is based on intermediate values that result from the program's execution.
- For example: a division by zero warning is the dynamic result.
- If a problem, such as a data flow anomaly, can be detected by static analysis methods, then it does not belong in testing- it belongs in the language processor.
- There is actually a lot more static analysis for data flow analysis for data flow anomalies going on in current language processors.
- For example, language processors which force variable declarations can detect (-u) and (ku) anomalies.

But still there are many things for which current notions of static analysis are **inadequate**

Why isn't static analysis enough?

- there are many things for which current notions of static analysis are **inadequate they are:**
 - Dead Variables.

- Arrays.
- Records and pointers.
- Dynamic subroutine or function name in a call.
- False anomalies.
- Recoverable anomalies and alternate state graphs.
- Concurrency, interrupts, system issues.
- Although static analysis methods have limits, they are worth using and a continuing trend in language processor design has been better static analysis methods for data flow anomaly detection.

The Data Flow Model

- The data flow model is based on the program's control flow graph.
- Here we annotate each link with symbols or sequences of symbols that denote the sequence of data operations on that link with respect to the variable of interest. Such annotations are called **link weights**.
- the control flow graph structure is same for every variable: it is the weights that change.

Components of the Model

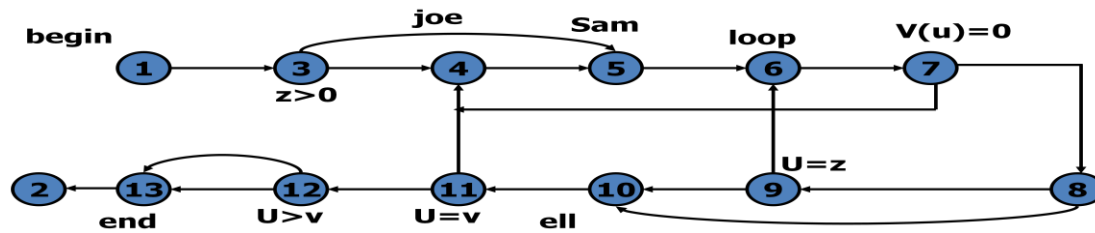
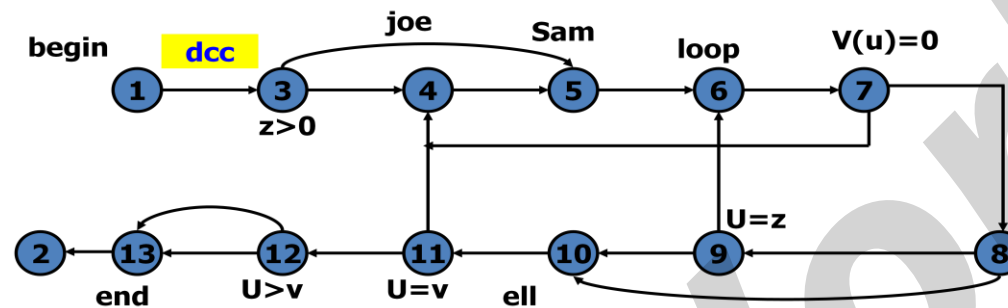
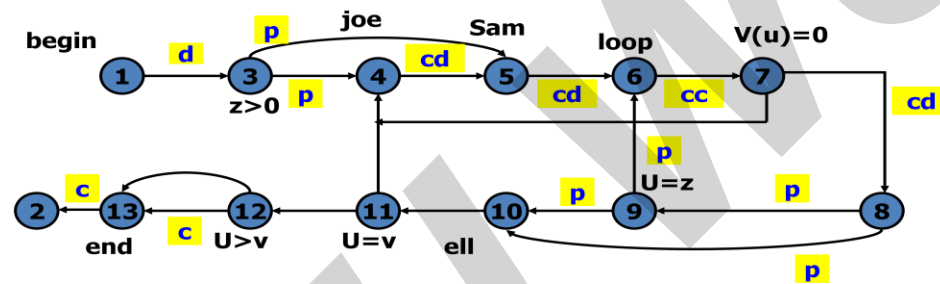
- Here are the modeling rules:
- To every statement there is a node, whose name is unique. Every node has at least one outlink and at least one inlink except for exit nodes and entry nodes.
- Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statements to complete the graph. The entry nodes are dummy nodes placed at entry statements for the same reason.
- The outlink of simple statements are weighted by the proper sequence of data flow actions for that statement.
- Predicate nodes are weighted with the p- use on every outlink, appropriate to that outlink.
- Every sequence of simple statements can be replaced by a pair of nodes that has, as weights on the link between them, the concatenation of link weights.
- If there are several data flow actions on a given link for a given variable, then the weight of the link is denoted by the sequence of actions on that link for that variable.
- Conversely, a link with several data flow actions on it can be replaced by a succession of equivalent links, each of which has at most one data flow action for any variable.

Example-Data Flow Graph

```

CODE (PDL)
  INPUT X,Y
  Z:=X+Y
  V:=X-Y
  IF Z>0 GOTO SAM
JOE:  Z:=Z-1
SAM:  Z:=Z+V
      FOR U=0 TO Z
      V(U),U(V) :=(Z+V)*U
      IF V(U)=0 GOTO JOE
Z:=Z-1
      IF Z=0 GOTO ELL
      U:=U+1
      NEXT U
      V(U-1) :=V(U+1)+U(V-1)
ELL:  V(U+U(V)) :=U+V
      IF U=V GOTO JOE
      IF U>V THEN U:=Z
      Z:=U
      END

```

Un annotated Control Flow GraphControl Flow Graph Annotated for X and Y Data FlowsControl Flow Graph Annotated for Z Data FlowData Flow Testing Strategies

- Data Flow Testing Strategies are structural strategies.
- Data Flow Testing Strategies require data flow link weights.
- Data Flow Testing Strategies are based on selecting test path segments (sub paths) that satisfy some characteristic of data flows for all data objects.
- For example, all sub paths that contain a d.

Definition Clear Path Segment

- Definition Clear Path Segment with respect to a variable X is a connected sequence of links such that X is defined on the first link and not redefined or killed on any sub sequent link of that path segment.
- The fact that there is a definition clear sub path between two nodes does not imply that all sub paths between those nodes are definition clear.

Loop Free Path Segment

A loop free path segment is a path segment for which every node is visited at most once. A simple path segment

- A simple path segment is a path segment in which at most one node is visited twice. du path
- A du path from node i to k is a path segment such that if the last link has a computational use of X, then the path is simple and definition clear; if the penultimate node is j- that is, the path is (i, p, q, ..., r, s, t, j, k) and link (j, k) has a predicate use- then the path from i to j is both loop free and definition clear.

The data flow testing strategies

- Various types of data flow testing strategies in decreasing order of their effectiveness are:
 - All du paths Strategy
 - All uses Strategy
 - All p-uses/some c-uses Strategy
 - All c-uses/some p-uses Strategy
 - All definitions Strategy
 - All predicates uses, all computational uses Strategy

All du paths Strategy

- The all du path (ADUP) strategy is the strongest data flow testing strategy.
- It requires that every du path from every definition of every variable to every use of that definition be exercised under some test.

All Uses Strategy

- The all uses strategy is that at least one definition clear path from every definition of every variable to every use of that definition be exercised under some test.
- The all uses strategy is that at least one definition clear path from every definition of every variable to every use of that definition be exercised under some test.

All P-uses/some c-uses Strategy

All p-uses/some c-uses (APU+c) strategy is defined as follows: for every variable and every definition of that variable, include at least one definition free path from the definition to every predicate use; if there are definitions of the variables that are not covered by the above prescription, then add computational use test cases as required to cover every definition.

All C-uses/some p-uses Strategy

The all c-uses/some p-uses strategy (ACU+P) is to first ensure coverage by computational use cases and if any definition is not covered by the previously selected paths, add such predicate use cases as are needed to assure that every definition is included in some test.

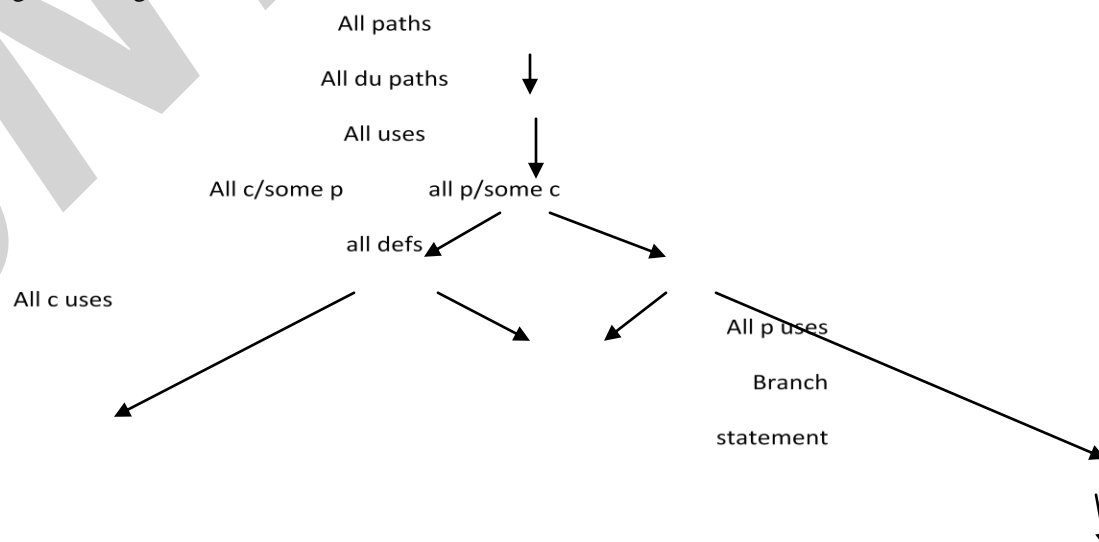
All Definitions Strategy

The all definitions strategy asks only every definition of every variable be covered by atleast one use of that variable, be that use a computational use or a predicate use.

All predicate-uses, All Computational uses Strategies

- The all predicate uses strategy is derived from APU+C strategy by dropping the requirement that we include a c-use for the variable if there are no p-uses for the variable.
- The all computational uses strategy is derived from ACU+P strategy by dropping the requirement that we include a p-use for the variable if there are no c-uses for the variable.

Ordering the Strategies



Slicing and Dicing

- A program slice is a part of a program defined with respect to a given variable X and a statement i: it is the set of all statements that could affect the value of X at statement i- where the influence of a faulty statement could result from an improper computational use or predicate use of some other variable at prior statements. If X is incorrect at statement i, it follows that the bug must be in the program slice for X with respect to i;
- A program dice is a part of a slice in which all statements which are known to be correct have been removed.

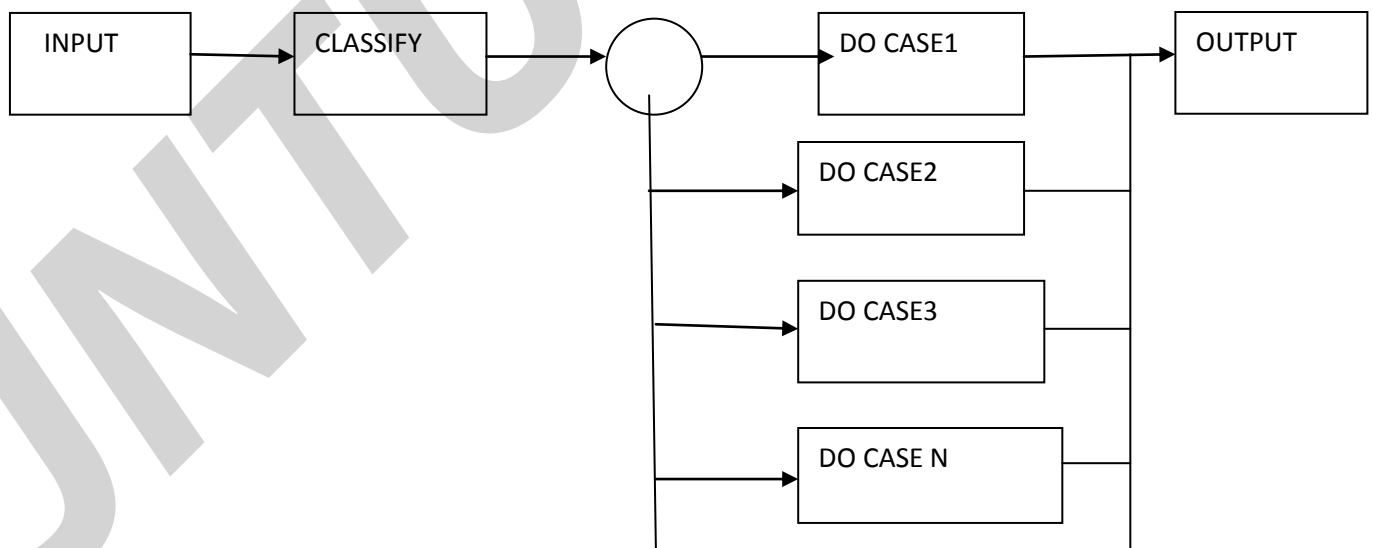
UNIT-4**Domain Testing**

- Programs as input data classifiers: domain testing attempts to determine whether the classification is or is not correct

Domains and Paths

- Domain testing can be based on specifications or equivalent implementation information.
- If domain testing is based on specifications, it is a functional test technique.
- If domain testing is based implementation details, it is a structural test technique.
- For example, you're doing domain testing when you check extreme values of an input variable.
- Before doing whatever it does, a routine must classify the input and set it moving on the right path.

In domain testing we focus on the classification aspect of the routine rather than on the calculations



- Structural knowledge is not needed for this model-only a consistent, complete specification of input values for each case.
- We can infer that for each case there must be atleast one path to process that case

A domain is a set

- An input domain is a set.
- If the source language supports set definitions less testing is needed because the compiler does much of it for us.
- Domain testing does not work well with arbitrary discrete sets of data objects.

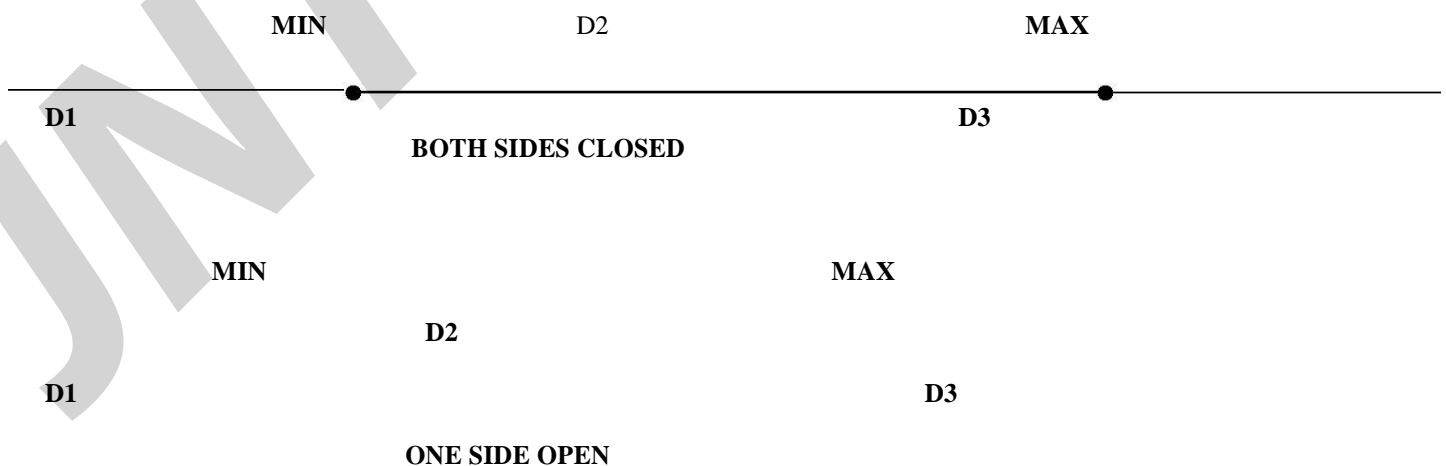
Domains, Paths, and Predicates

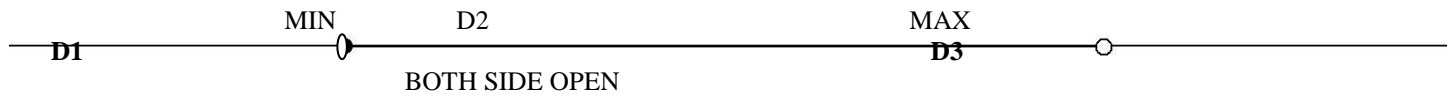
- In domain testing, predicates are assumed to be interpreted in terms of input vector variable.
- For every domain there is at least one path through the routine.
- There may be more than one path if the domain consists of disconnected parts or if the domain is defined by the union of two or more domains.
- Domains are defined their boundaries.
- For every boundary there is at least one predicate that specifies what numbers belong to the domain and what numbers don't.
- For example, if the predicate is $x^2+y^2 < 16$, the domain is the inside of a circle of radius 4 about the origin.

Domain Closure

- A domain boundary is closed with respect to a domain if the points on the boundary belong to the domain.
- If the boundary points belong to some other domain, the boundary is said to be open.

Open and Closed Domains





Domain Dimensionality

- Every input variable adds one dimension to the domain.
- One variable defines domains on a number line.
- Two variables define planar domains.
- Three variables define solid domains.
- Every new predicate slices through previously defined domains and cuts them in half.
- Every boundary slices through the input vector space with a dimensionality which is less than the dimensionality of the space.
- Thus, planes are cut by lines and points, volumes by planes, lines and points and n-spaces by hyper planes.

The bug assumption

- The bug assumption for the domain testing is that processing is okay but the domain definition is wrong.
- An incorrectly implemented domain means that boundaries are wrong, which may in turn mean that control flow predicates are wrong.
- Many different bugs can result in domain errors.

Domain Errors

- Double zero representation: In computer or Languages that have a distinct positive and negative zero, boundary errors for negative zero are common.
- Floating point zero check: A floating point number can equal zero only if the previous definition of that number set it to zero or if it is subtracted from it self or multiplied by zero. So the floating point zero check to be done against a epsilon value.
- Contradictory domains: a contradictory domain specification means that at least two supposedly distinct domains overlap.
- Ambiguous domains: Ambiguous domains means that union of the domains is incomplete. That is there are missing domains or holes in the specified domains.
- Over specified domains: the domain can be overloaded with so many contradictions that the result is null domain.
- Boundary errors: Errors caused in and around the boundary of a domain. Example, boundary closure bug, shifted, tilted, missing, extra boundary.
- Closure reversal: Some times the logical complementation can be done improperly. For example, complementation of \leq can be wrongly implemented as \geq instead of $>$.
- Faulty logic: compound predicates are subject to faulty logic transformations and improper simplification. If the predicate define domain boundaries, all kinds of domain bugs can result from faulty logic manipulations.

Restrictions to domain testing

- Coincidental correctness
- Representative Outcome
- Simple domain boundaries and compound predicates
- Functional Homogeneity of Bugs
- Linear vector space
- Loop free software

Nice Domains

- Some important properties of nice domains are:
- Linear, complete, systematic, orthogonal, consistently closed, simply connected and convex.
- To the extent that domains have these properties domain testing is easy as testing gets.
- The bug frequency is lesser for nice domain than for ugly domains.

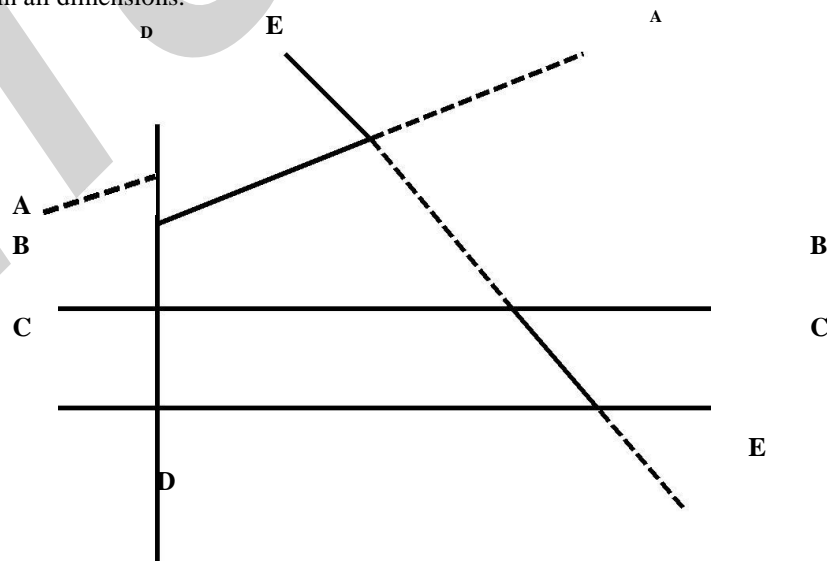
Linear and NonLinear Boundaries

- Nice domain boundaries are defined by linear inequalities or equations.
- The impact on testing stems from the fact that it takes only two points to determine a straight line and three points to determine a plane and in general $n+1$ points to determine a n -dimensional hyper plane.

In practice more than 99.99% of all boundary predicates are either linear or can be linearized by simple variable transformations

Complete Boundaries

- Nice domain boundaries are complete in that they span the number space from plus to minus infinity in all dimensions.



Systematic Boundaries

- By systematic boundary means that boundary inequalities related by a simple function such as a constant.
- Example:

$$\begin{array}{l} f_1(x) \geq k_1 \quad \text{or} \\ f_1(x) \geq g(1,c) \quad f_2(x) \geq k_2 \\ \text{or } f_2(x) \geq g(2,c) \end{array}$$

$$\dots\dots\dots f_i(x) \geq k_i \text{ or } f_i(x) \geq g(i,c)$$

- Where f_i is an arbitrary function, x is the input vector, k_i and c are constants, and $g(i,c)$ is a decent function over i and c that yields a constant such as $k+ic$.

Orthogonal Boundaries

If two boundary sets U and V are said to be orthogonal if every inequality in V is perpendicular to every inequality in U .

Ugly Domains

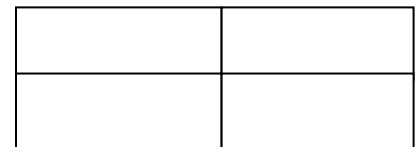
- Some domains are born ugly and uglified by bad specifications.
- Every simplification of ugly domains by programmers can be either good or bad.

Ambiguities and Contradictions

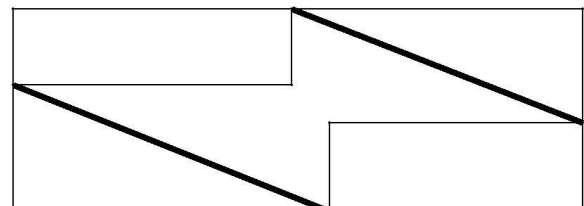
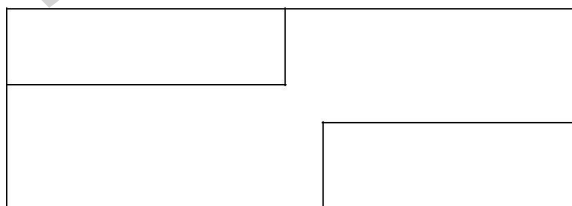
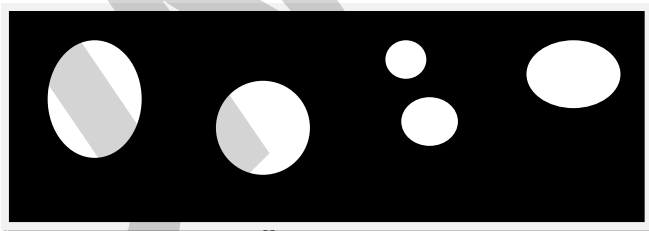
- Domain ambiguities are holes in the input space.
- The holes may lie within the domains or in cracks between domains.
- Two kinds of contradictions are possible: overlapped domain specifications and overlapped closure specifications.

Simplifying the Topology

- The programmer's and tester's reaction to complex domains is the same- simplify.
- There are three generic cases: concavities, holes and disconnected pieces.



Making it Convex



Simplifying the topology

Domain testing overview

- Domains are defined by their boundaries; therefore, domain testing concentrates test points on or near boundaries.
- Classify what can go wrong with boundaries, then define a test strategy for each case. Pick enough points to test for all categorized kinds of boundary errors.
- Run the tests by post test analysis determine if any boundaries are faulty and if so, how.
- Run enough tests to verify every boundary of every domain.

Domain bugs and how to test for them

- An interior point is a point in the domain such that all points within an arbitrarily small distance are also in the domain.
- A boundary point is one such that within an epsilon neighborhood there are points both in the domain and not in the domain.
- An extreme point is a point that does not lie between any two other arbitrary but distinct points of a domain.
- An on point is a point on the boundary.
- If the domain boundary is closed, an off point is a point near the boundary but in the adjacent domain.
- If the domain is open, an off point is a point near the boundary but in the domain being tested.

Generic Domain Bugs

- The generic domain Bugs are: shifted boundaries, tilted boundaries, open/closed errors, extra boundary and missing boundary.



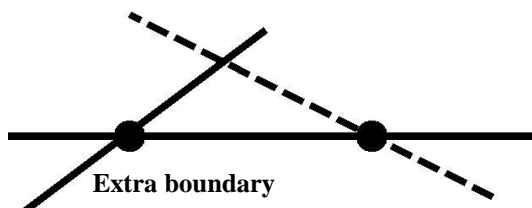
Shifted Boundaries



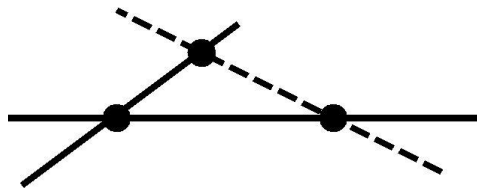
Tilted Boundaries



Open/closed error



Extra boundary



Missing Boundary

Testing one dimensional domains-open boundaries

Procedure for Testing

- The procedure is conceptually is straight forward.
- It can be done by hand for two dimensions and for a few domains and practically impossible for more than two variables.
- 1. identify input variables.
- 2. identify variable which appear in domain defining predicates, such as control flow predicates.
- 3. interpret all domain predicates in terms of input variables.
- 4. for p binary predicates, there are at most 2^p combinations of TRUE-FALSE values and therefore, at most 2^p domains. Find the set of all non null domains. the result is a boolean expression in the predicates consisting a set of AND terms joined by OR's. for example $ABC+DEF+GHI....$ Where the capital letters denote predicates. Each product term is a set of linear inequality that defines a domain or a part of a multiply connected domains.
- 5. Solve these inequalities to find all the extreme points of each domain using any of the linear programming methods.

UNIT V :

Paths, Path products and Regular expressions : Path products & path expression, reduction procedure, applications, regular expressions & flow anomaly detection.

PATHS, PATH PRODUCTS AND REGULAR EXPRESSIONS

This unit gives an in depth overview of Paths of various flow graphs, their interpretations and application.

PATH PRODUCTS AND PATH EXPRESSION:

- **MOTIVATION:**
 - Flow graphs are being an abstract representation of programs.
 - Any question about a program can be cast into an equivalent question about an appropriate flowgraph.
 - Most software development, testing and debugging tools use flow graphs analysis techniques.
- **PATH PRODUCTS:**
 - Normally flow graphs used to denote only control flow connectivity.
 - The simplest weight we can give to a link is a name.

- Using link names as weights, we then convert the graphical flow graph into an equivalent algebraic like expressions which denotes the set of all possible paths from entry to exit for the flow graph.
- Every link of a graph can be given a name.
- The link name will be denoted by lower case italic letters.
- In tracing a path or path segment through a flow graph, you traverse a succession of link names.
- The name of the path or path segment that corresponds to those links is expressed naturally by concatenating those link names.
- For example, if you traverse links a,b,c and d along some path, the name for that path segment is abcd. This path name is also called a **path product**. Figure 5.1 shows some examples:

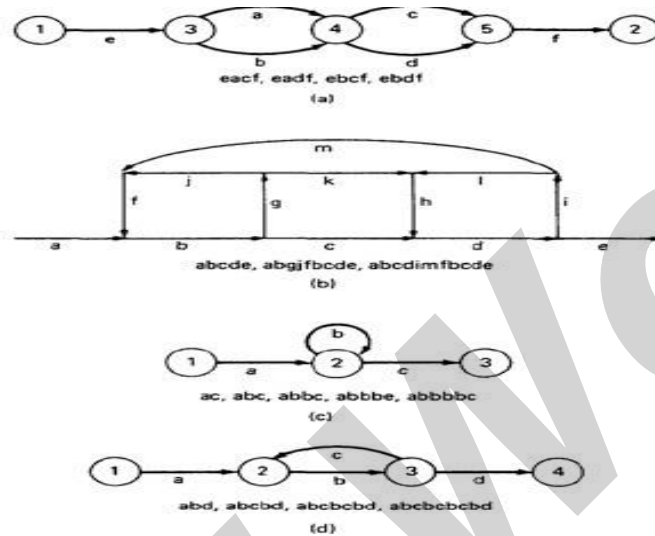


Figure 5.1: Examples of paths.

• **PATH EXPRESSION:**

- Consider a pair of nodes in a graph and the set of paths between those node.
- Denote that set of paths by Upper case letter such as X,Y. From Figure 5.1c, the members of the path set can be listed as follows:

ac, abc, abbc, abbbc, abbbbc.....

- Alternatively, the same set of paths can be denoted by :

ac+abc+abbc+abbbc+abbbbc+.....

- The + sign is understood to mean "or" between the two nodes of interest, paths ac, or abc, or abbc, and so on can be taken.
- Any expression that consists of path names and "OR"s and which denotes a set of paths between two nodes is called a "**Path Expression**".

• **PATH PRODUCTS:**

- The name of a path that consists of two successive path segments is conveniently expressed by the concatenation or **Path Product** of the segment names.
- For example, if X and Y are defined as X=abcde,Y=fghij,then the path corresponding to X followed by Y is denoted by

XY=abcdefghij

- Similarly,
- $YX = fghijabcde$
- $aX = aabcde$
- $Xa = abcdea$
- $XaX = abcdeaabcde$
- If X and Y represent sets of paths or path expressions, their product represents the set of paths that can be obtained by following every element of X by any element of Y in all possible ways. For example,
- $X = abc + def + ghi$
- $Y = uvw + z$
- Then,
- $XY = abcuvw + defuvw + ghiuvw + abcz + defz + ghiz$
- If a link or segment name is repeated, that fact is denoted by an exponent. The exponent's value denotes the number of repetitions:
- $a^1 = a$; $a^2 = aa$; $a^3 = aaa$; $a^n = aaaa \dots n$ times.

Similarly, if

$$X = abcde$$

then

$$X^1 = abcde$$

$$X^2 = abcdeabcde = (abcde)^2$$

$$X^3 = abcdeabcdeabcde = (abcde)^2 abcde$$

$$= abcde(abcde)^2 = (abcde)^3$$

- The path product is not commutative (that is $XY \neq YX$).
- The path product is Associative.

RULE 1: $A(BC) = (AB)C = ABC$

where A,B,C are path names, set of path names or path expressions.

- The zeroth power of a link name, path product, or path expression is also needed for completeness. It is denoted by the numeral "1" and denotes the "path" whose length is zero - that is, the path that doesn't have any links.
- $a^0 = 1$
- $X^0 = 1$
- **PATH SUMS:**
 - The "+" sign was used to denote the fact that path names were part of the same set of paths.
 - The "PATH SUM" denotes paths in parallel between nodes.
 - Links a and b in Figure 5.1a are parallel paths and are denoted by $a + b$. Similarly, links c and d are parallel paths between the next two nodes and are denoted by $c + d$.
 - The set of all paths between nodes 1 and 2 can be thought of as a set of parallel paths and denoted by $eacf + eadf + ebcf + ebdf$.
 - If X and Y are sets of paths that lie between the same pair of nodes, then $X+Y$ denotes the UNION of those set of paths. For example, in Figure 5.2:

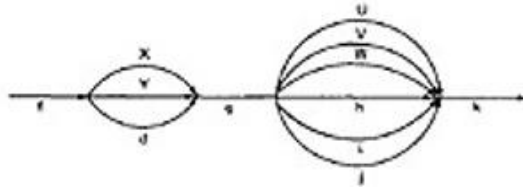


Figure 5.2: Examples of path sums.

The first set of parallel paths is denoted by $X + Y + d$ and the second set by $U + V + W + h + i + j$. The set of all paths in this flowgraph is $f(X + Y + d)g(U + V + W + h + i + j)k$

- The path is a set union operation, it is clearly Commutative and Associative.
- RULE 2: $X+Y=Y+X$
- RULE 3: $(X+Y)+Z=X+(Y+Z)=X+Y+Z$

• **DISTRIBUTIVE LAWS:**

- The product and sum operations are distributive, and the ordinary rules of multiplication apply; that is

$$\text{RULE 4: } A(B+C)=AB+AC \text{ and } (B+C)D=BD+CD$$

- Applying these rules to the below Figure 5.1a yields
- $e(a+b)(c+d)f=e(ac+ad+bc+bd)f = eacf+eadf+ebcf+ebdf$

• **ABSORPTION RULE:**

- If X and Y denote the same set of paths, then the union of these sets is unchanged; consequently,

$$\text{RULE 5: } X+X=X \text{ (Absorption Rule)}$$

- If a set consists of paths names and a member of that set is added to it, the "new" name, which is already in that set of names, contributes nothing and can be ignored.
- For example,
- if $X=a+aa+abc+abcd+def$ then

$$X+a = X+aa = X+abc = X+abcd = X+def = X$$

It follows that any arbitrary sum of identical path expressions reduces to the same path expression.

• **LOOPS:**

- Loops can be understood as an infinite set of parallel paths. Say that the loop consists of a single link b. then the set of all paths through that loop point is $b^0+b^1+b^2+b^3+b^4+b^5+\dots$

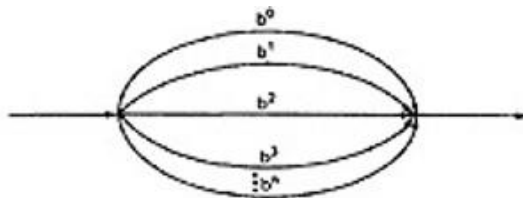
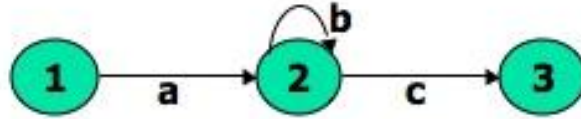


Figure 5.3: Examples of path loops.

- This potentially infinite sum is denoted by b^* for an individual link and by X^* when X is a path expression.

**Figure 5.4: Another example of path loops.**

- The path expression for the above figure is denoted by the notation:

$$ab^*c = ac + abc + abbc + abbbc + \dots$$

- Evidently,

$$aa^* = a^*a = a^+ \text{ and } XX^* = X^*X = X^+$$

- It is more convenient to denote the fact that a loop cannot be taken more than a certain, say n , number of times.
- A bar is used under the exponent to denote the fact as follows:

$$X^{\bar{n}} = X^0 + X^1 + X^2 + X^3 + X^4 + X^5 + \dots + X^n$$

• **RULES 6 - 16:**

- The following rules can be derived from the previous rules:

- RULE 6: $X^n + X^m = X^n$ if $n > m$

$$\text{RULE 6: } X^n + X^m = X^m \text{ if } m > n$$

$$\text{RULE 7: } X^n X^m = X^{n+m}$$

$$\text{RULE 8: } X^n X^* = X^* X^n = X^*$$

$$\text{RULE 9: } X^n X^+ = X^+ X^n = X^+$$

$$\text{RULE 10: } X^* X^+ = X^+ X^* = X^+$$

$$\text{RULE 11: } 1 + 1 = 1$$

$$\text{RULE 12: } 1X = X1 = X$$

Following or preceding a set of paths by a path of zero length does not change the set.

$$\text{RULE 13: } 1^n = 1^{\bar{n}} = 1^* = 1^+ = 1$$

No matter how often you traverse a path of zero length, it is a path of zero length.

$$\text{RULE 14: } 1^+ + 1 = 1^* = 1$$

The null set of paths is denoted by the numeral 0. it obeys the following rules:

$$\text{RULE 15: } X + 0 = 0 + X = X$$

$$\text{RULE 16: } 0X = X0 = 0$$

If you block the paths of a graph for or after by a graph that has no paths, there won't be any paths.

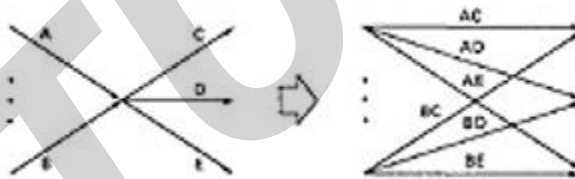
REDUCTION PROCEDURE:

- **REDUCTION PROCEDURE ALGORITHM:**

- This section presents a reduction procedure for converting a flowgraph whose links are labeled with names into a path expression that denotes the set of all entry/exit paths in that flowgraph. The procedure is a node-by-node removal algorithm.
- The steps in Reduction Algorithm are as follows:
 1. Combine all serial links by multiplying their path expressions.
 2. Combine all parallel links by adding their path expressions.
 3. Remove all self-loops (from any node to itself) by replacing them with a link of the form X^* , where X is the path expression of the link in that loop.

STEPS 4 - 8 ARE IN THE ALGORITHM'S LOOP:

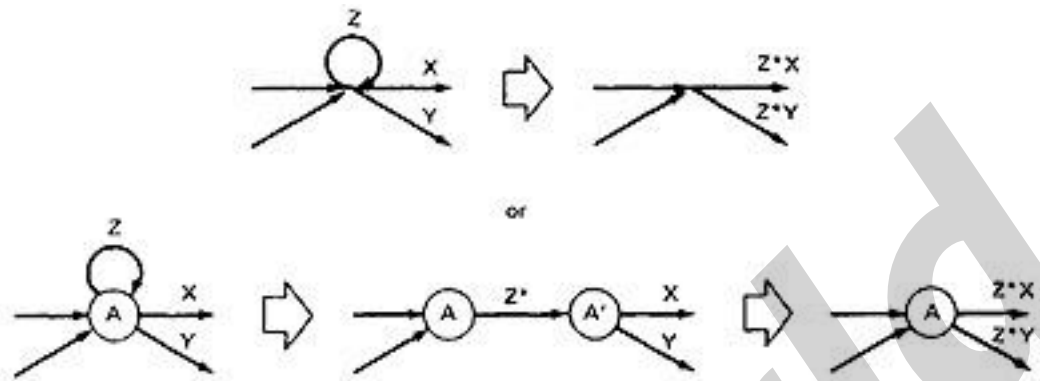
4. Select any node for removal other than the initial or final node. Replace it with a set of equivalent links whose path expressions correspond to all the ways you can form a product of the set of inlinks with the set of outlinks of that node.
 5. Combine any remaining serial links by multiplying their path expressions.
 6. Combine all parallel links by adding their path expressions.
 7. Remove all self-loops as in step 3.
 8. Does the graph consist of a single link between the entry node and the exit node? If yes, then the path expression for that link is a path expression for the original flowgraph; otherwise, return to step 4.
- A flowgraph can have many equivalent path expressions between a given pair of nodes; that is, there are many different ways to generate the set of all paths between two nodes without affecting the content of that set.
 - The appearance of the path expression depends, in general, on the order in which nodes are removed.
- **CROSS-TERM STEP (STEP 4):**
 - The cross - term step is the fundamental step of the reduction algorithm.
 - It removes a node, thereby reducing the number of nodes by one.
 - Successive applications of this step eventually get you down to one entry and one exit node. The following diagram shows the situation at an arbitrary node that has been selected for removal:



- From the above diagram, one can infer:
- $(a + b)(c + d + e) = ac + ad + ae + bc + bd + be$

- **LOOP REMOVAL OPERATIONS:**

- There are two ways of looking at the loop-removal operation:



- In the first way, we remove the self-loop and then multiply all outgoing links by Z^* .
- In the second way, we split the node into two equivalent nodes, call them A and A' and put in a link between them whose path expression is Z^* . Then we remove node A' using steps 4 and 5 to yield outgoing links whose path expressions are Z^*X and Z^*Y .
- **A REDUCTION PROCEDURE - EXAMPLE:**
 - Let us see by applying this algorithm to the following graph where we remove several nodes in order; that is

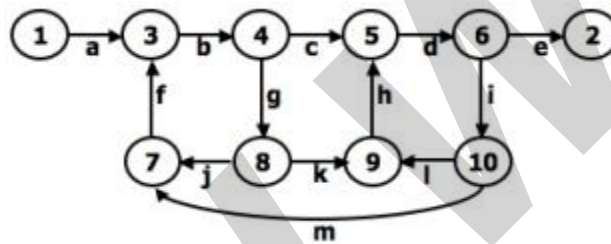
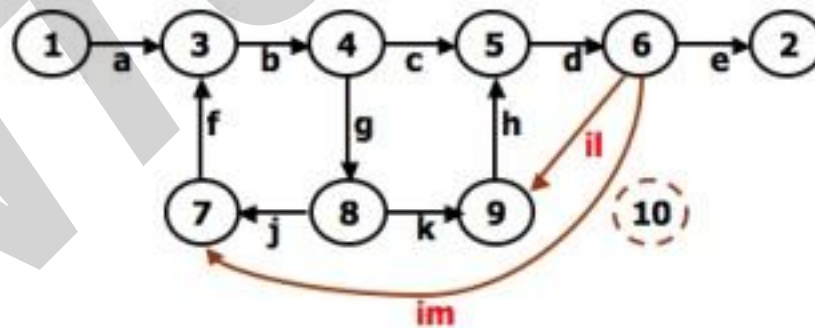
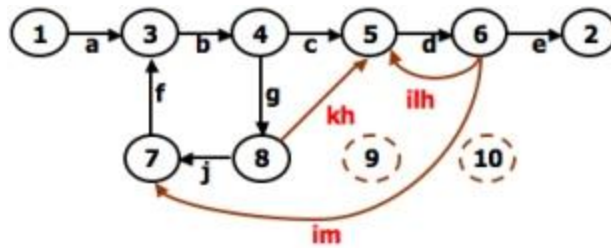


Figure 5.5: Example Flowgraph for demonstrating reduction procedure.

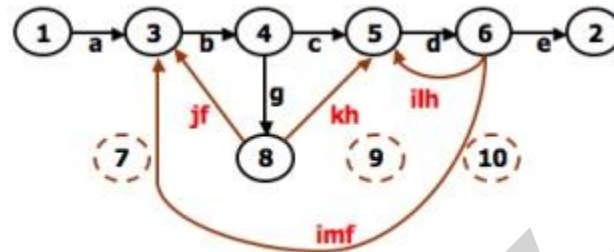
- Remove node 10 by applying step 4 and combine by step 5 to yield



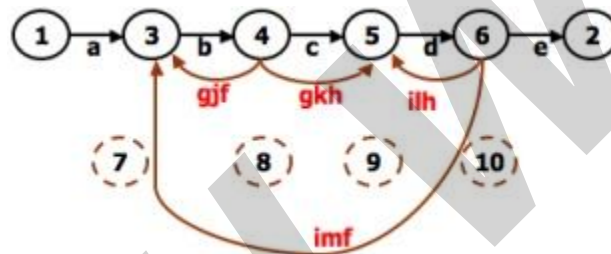
- Remove node 9 by applying step 4 and 5 to yield



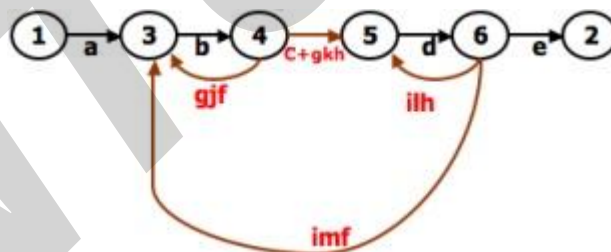
- Remove node 7 by steps 4 and 5, as follows:



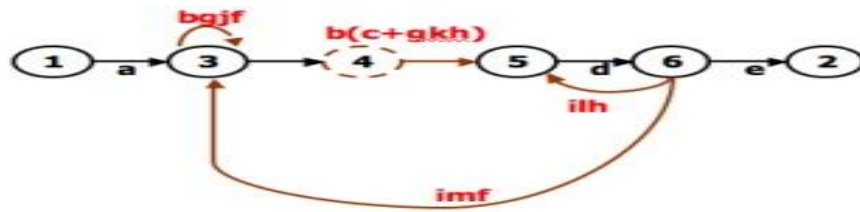
- Remove node 8 by steps 4 and 5, to obtain:



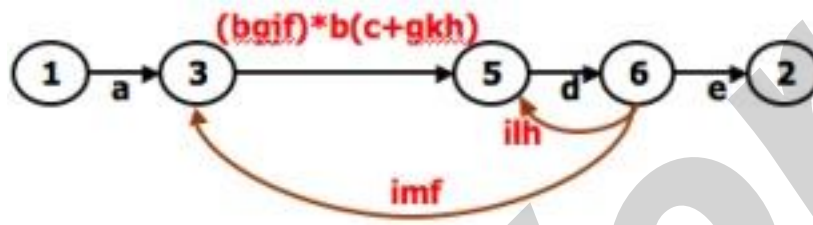
- PARALLEL TERM (STEP 6):**
Removal of node 8 above led to a pair of parallel links between nodes 4 and 5. combine them to create a path expression for an equivalent link whose path expression is $c+gkh$; that is



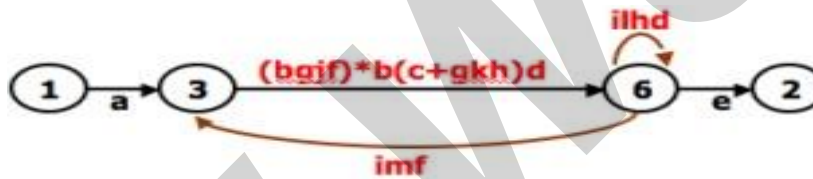
- LOOP TERM (STEP 7):**
Removing node 4 leads to a loop term. The graph has now been replaced with the following equivalent simpler graph:



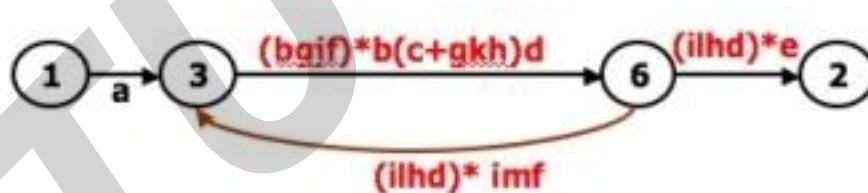
- Continue the process by applying the loop-removal step as follows:



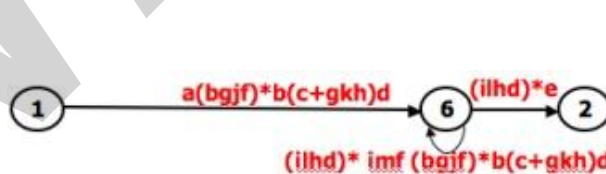
- Removing node 5 produces:



- Remove the loop at node 6 to yield:



- Remove node 3 to yield:



- Removing the loop and then node 6 result in the following expression:
- $a(bgjf)*b(c+gkh)d((ilh d)*imf(bgjf)*b(c+gkh)d)*(ilh d)*e$
- You can practice by applying the algorithm on the following flowgraphs and generate their respective path expressions:

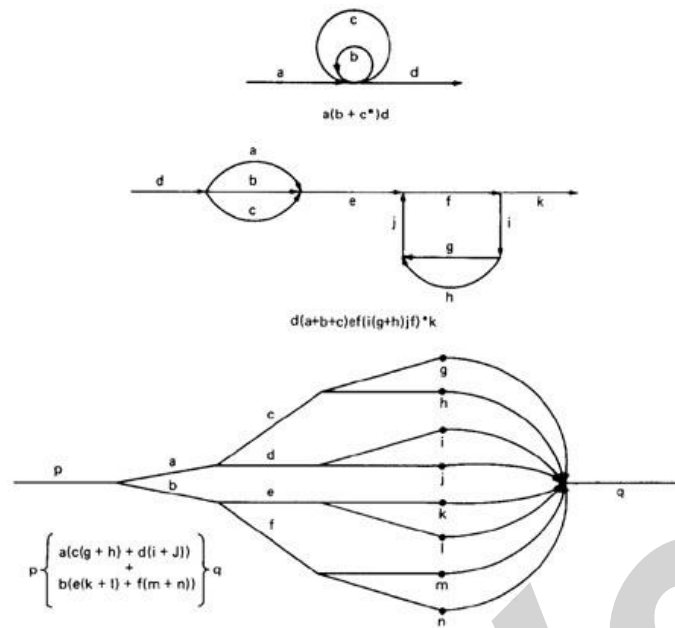


Figure 5.6: Some graphs and their path expressions.

APPLICATIONS:

• APPLICATIONS:

- The purpose of the node removal algorithm is to present one very generalized concept- the path expression and way of getting it.
- Every application follows this common pattern:
 1. Convert the program or graph into a path expression.
 2. Identify a property of interest and derive an appropriate set of "arithmetic" rules that characterizes the property.
 3. Replace the link names by the link weights for the property of interest. The path expression has now been converted to an expression in some algebra, such as ordinary algebra, regular expressions, or boolean algebra. This algebraic expression summarizes the property of interest over the set of all paths.
 4. Simplify or evaluate the resulting "algebraic" expression to answer the question you asked.

• HOW MANY PATHS IN A FLOWGRAPH ?

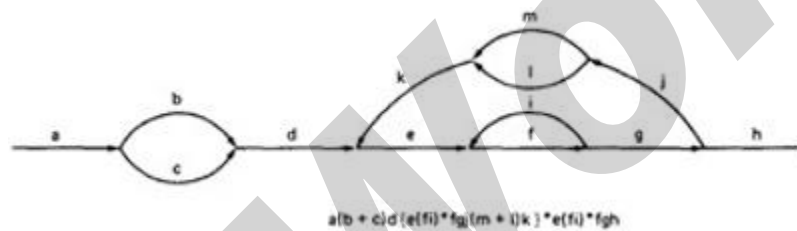
- The question is not simple. Here are some ways you could ask it:
 1. What is the maximum number of different paths possible?
 2. What is the fewest number of paths possible?
 3. How many different paths are there really?
 4. What is the average number of paths?
- Determining the actual number of different paths is an inherently difficult problem because there could be unachievable paths resulting from correlated and dependent predicates.
- If we know both of these numbers (maximum and minimum number of possible paths) we have a good idea of how complete our testing is.
- Asking for "the average number of paths" is meaningless.

• MAXIMUM PATH COUNT ARITHMETIC:

- Label each link with a link weight that corresponds to the number of paths that link represents.
- Also mark each loop with the maximum number of times that loop can be taken. If the answer is infinite, you might as well stop the analysis because it is clear that the maximum number of paths will be infinite.
- There are three cases of interest: parallel links, serial links, and loops.

Case	Path expression	Weight expression
Parallels	$A+B$	W_A+W_B
Series	AB	$W_A W_B$
Loop	A^n	$\sum_{j=0}^n W_A^j$

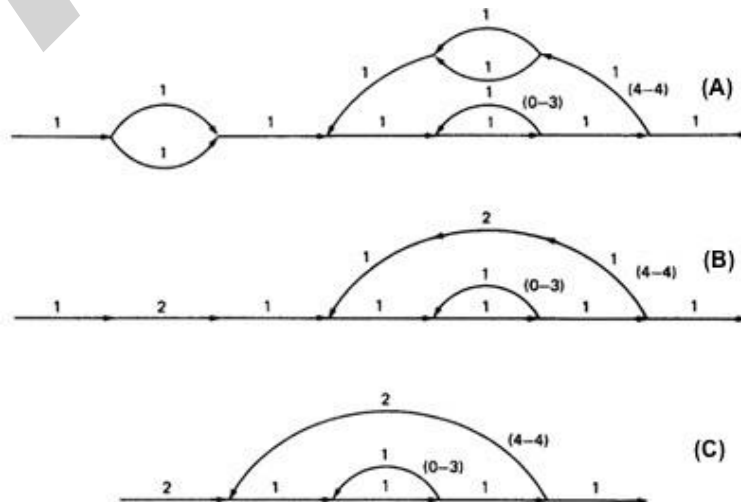
- This arithmetic is an ordinary algebra. The weight is the number of paths in each set.
- **EXAMPLE:**
 - The following is a reasonably well-structured program.



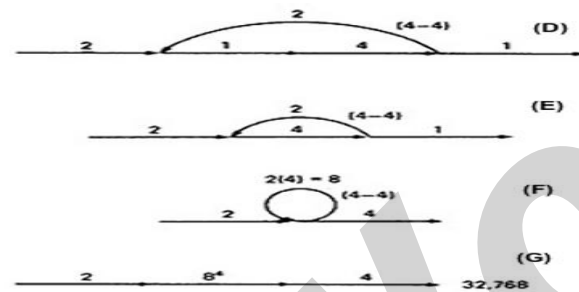
Each link represents a single link and consequently is given a weight of "1" to start. Lets say the outer loop will be taken exactly four times and inner Loop Can be taken zero or three times Its path expression, with a little work, is:

Path expression: $a(b+c)d\{e(fi)*fgj(m+l)k\}*e(fi)*fgh$

- **A:** The flow graph should be annotated by replacing the link name with the maximum of paths through that link (1) and also note the number of times for looping.
- **B:** Combine the first pair of parallel loops outside the loop and also the pair in the outer loop.
- **C:** Multiply the things out and remove nodes to clear the clutter.



- **For the Inner Loop:**
D: Calculate the total weight of inner loop, which can execute a min. of 0 times and max. of 3 times. So, it inner loop can be evaluated as follows:
 $1^3 = 1^0 + 1^1 + 1^2 + 1^3 = 1 + 1 + 1 + 1 = 4$
- **E:** Multiply the link weights inside the loop: $1 \times 4 = 4$
- **F:** Evaluate the loop by multiplying the link weights: $2 \times 4 = 8$.
- **G:** Simplifying the loop further results in the total maximum number of paths in the flowgraph:
 $2 \times 8^4 \times 2 = 32,768$.



Alternatively, you could have substituted a "1" for each link in the path expression and then simplified, as follows:

$$\begin{aligned}
 & \mathbf{a(b+c)d\{e(fi)*fgj(m+l)k\}*e(fi)*fgh} \\
 &= 1(1 + 1)1(1(1 \times 1)^3 1 \times 1 \times 1(1 + 1)1^4 1(1 \times 1)^3 1 \times 1 \times 1 \\
 &= 2(1^3 1 \times 2^4 \times (2)^4 1^3 \\
 &= 2(4 \times 2^4 \times 4 \\
 &= 2 \times 8^4 \times 4 = 32,768
 \end{aligned}$$

This is the same result we got graphically.

Actually, the outer loop should be taken exactly four times. That doesn't mean it will be taken zero or four times. Consequently, there is a superfluous "4" on the outlink in the last step. Therefore the maximum number of different paths is 8192 rather than 32,768.

STRUCTURED FLOWGRAPH:

Structured code can be defined in several different ways that do not involve ad-hoc rules such as not using GOTOs.

A structured flowgraph is one that can be reduced to a single link by successive application of the transformations of Figure 5.7.

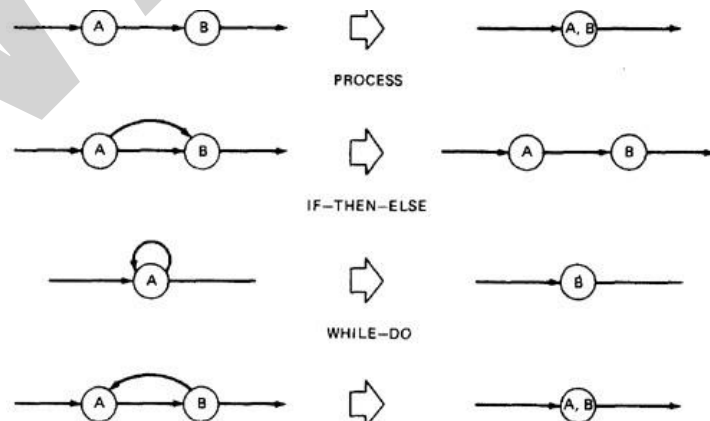


Figure 5.7: Structured Flowgraph Transformations.

The node-by-node reduction procedure can also be used as a test for structured code.

Flow graphs that DO NOT contain one or more of the graphs shown below (Figure 5.8) as subgraphs are structured.

0. Jumping into loops
1. Jumping out of loops
2. Branching into decisions
3. Branching out of decisions

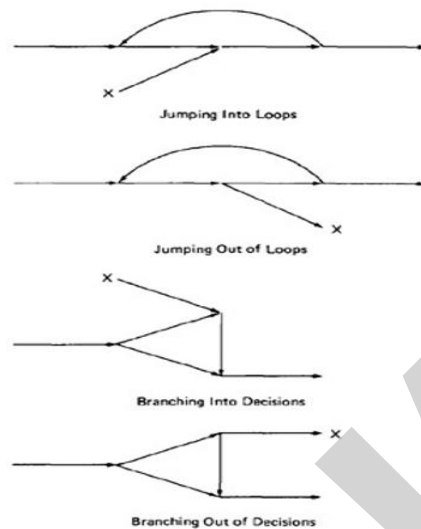


Figure 5.8: Un-structured sub-graphs.

LOWER PATH COUNT ARITHMETIC:

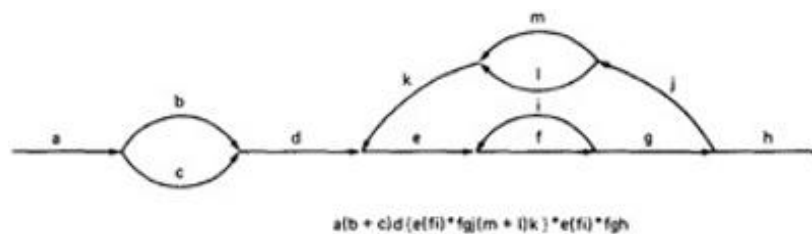
A lower bound on the number of paths in a routine can be approximated for structured flow graphs. The arithmetic is as follows:

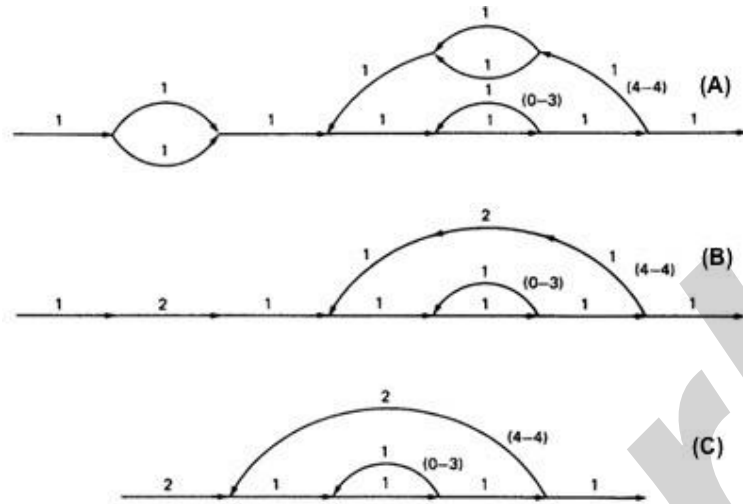
Case	Path expression	Weight expression
Parallels	$A+B$	W_A+W_B
Series	AB	$\max(W_A, W_B)$
Loop	A^n	$1, W_1$

The values of the weights are the number of members in a set of paths.

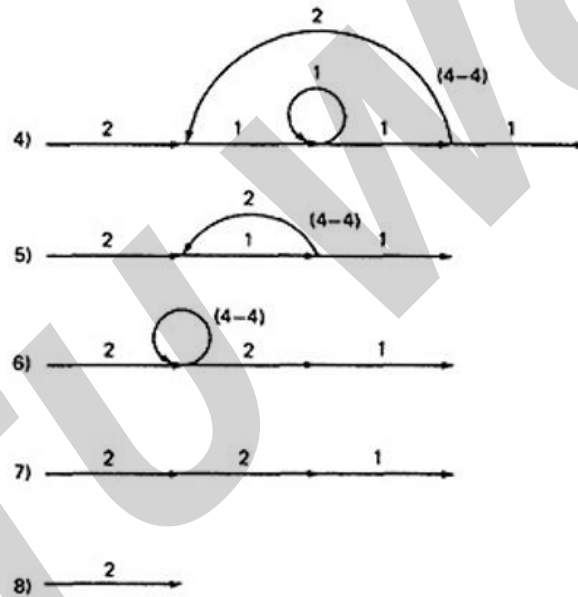
EXAMPLE:

- Applying the arithmetic to the earlier example gives us the identical steps until step 3 (C) as below:





- From Step 4, the it would be different from the previous example:



- If you observe the original graph, it takes at least two paths to cover and that it can be done in two paths.
- If you have fewer paths in your test plan than this minimum you probably haven't covered. It's another check.

CALCULATING THE PROBABILITY:

Path selection should be biased toward the low - rather than the high-probability paths.

This raises an interesting question:

What is the probability of being at a certain point in a routine?

This question can be answered under suitable assumptions, primarily that all probabilities involved are independent, which is to say that all decisions are independent and uncorrelated.

We use the same algorithm as before : node-by-node removal of uninteresting nodes.

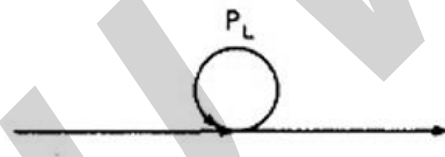
Weights, Notations and Arithmetic:

- Probabilities can come into the act only at decisions (including decisions associated with loops).
- Annotate each outlink with a weight equal to the probability of going in that direction.

- Evidently, the sum of the outlink probabilities must equal 1
- For a simple loop, if the loop will be taken a mean of N times, the looping probability is $N/(N + 1)$ and the probability of not looping is $1/(N + 1)$.
- A link that is not part of a decision node has a probability of 1.
- The arithmetic rules are those of ordinary arithmetic.

Case	Path expression	Weight expression
Parallel	$A+B$	P_A+P_B
Series	AB	P_AP_B
Loop	A^*	$P_A / (1-P_L)$

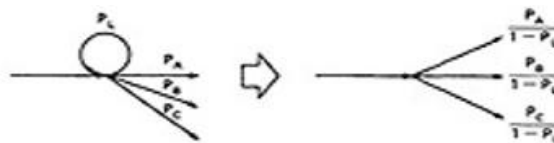
- In this table, in case of a loop, P_A is the probability of the link leaving the loop and P_L is the probability of looping.
- The rules are those of ordinary probability theory.
 1. If you can do something either from column A with a probability of P_A or from column B with a probability P_B , then the probability that you do either is $P_A + P_B$.
 2. For the series case, if you must do both things, and their probabilities are independent (as assumed), then the probability that you do both is the product of their probabilities.
- For example, a loop node has a looping probability of P_L and a probability of not looping of P_A , which is obviously equal to $1 - P_L$.



$$P_A = 1 - P_L$$

$$P_{\text{NEW}} = \frac{P_A}{1 - P_L} = \frac{1 - P_L}{1 - P_L} = 1$$

- Following the above rule, all we've done is replace the outgoing probability with 1 - so why the complicated rule? After a few steps in which you've removed nodes, combined parallel terms, removed loops and the like, you might find something like this:



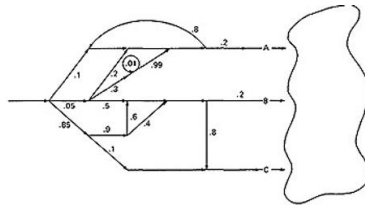
because $P_L + P_A + P_B + P_C = 1$, $1 - P_L = P_A + P_B + P_C$, and

$$\frac{P_A}{1 - P_L} + \frac{P_B}{1 - P_L} + \frac{P_C}{1 - P_L} = \frac{P_A + P_B + P_C}{1 - P_L} = 1$$

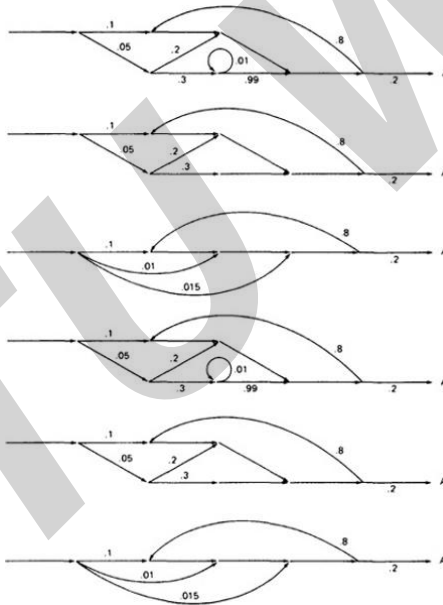
which is what we've postulated for any decision. In other words, division by $1 - P_L$ renormalizes the outlink probabilities so that their sum equals unity after the loop is removed.

EXAMPLE:

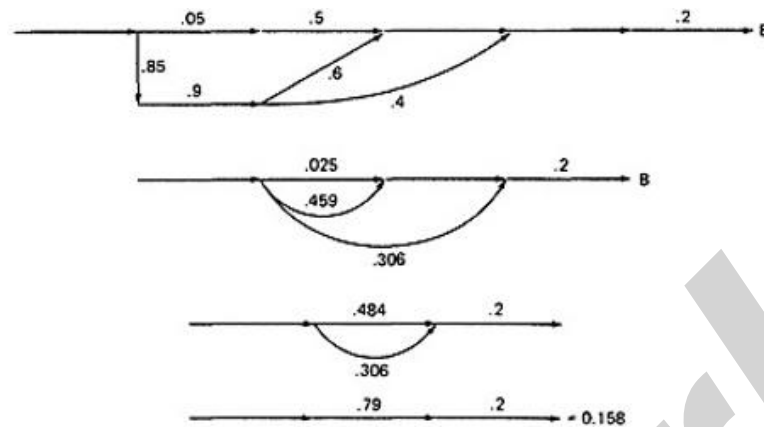
- Here is a complicated bit of logic. We want to know the probability associated with cases A, B, and C.



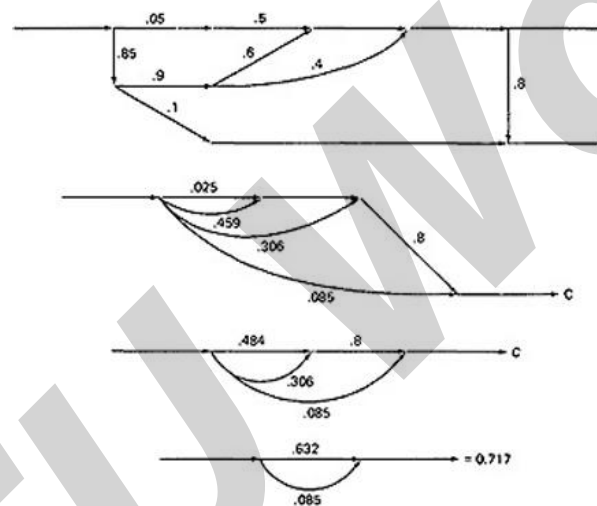
- Let us do this in three parts, starting with case A. Note that the sum of the probabilities at each decision node is equal to 1. Start by throwing away anything that isn't on the way to case A, and then apply the reduction procedure. To avoid clutter, we usually leave out probabilities equal to 1. CASE A:



- Case B is simpler:



- Case C is similar and should yield a probability of $1 - 0.125 - 0.158 = 0.717$:



- This checks. It's a good idea when doing this sort of thing to calculate all the probabilities and to verify that the sum of the routine's exit probabilities does equal 1.
- If it doesn't, then you've made calculation error or, more likely, you've left out some branching probability.
- How about path probabilities? That's easy. Just trace the path of interest and multiply the probabilities as you go.
- Alternatively, write down the path name and do the indicated arithmetic operation.
- Say that a path consisted of links a, b, c, d, e, and the associated probabilities were .2, .5, 1., .01, and 1 respectively. Path *abc bcb cde ab ddea* would have a probability of 5×10^{-10} .
- Long paths are usually improbable.

MEAN PROCESSING TIME OF A ROUTINE:

Given the execution time of all statements or instructions for every link in a flowgraph and the probability for each direction for all decisions are to find the mean processing time for the routine as a whole.

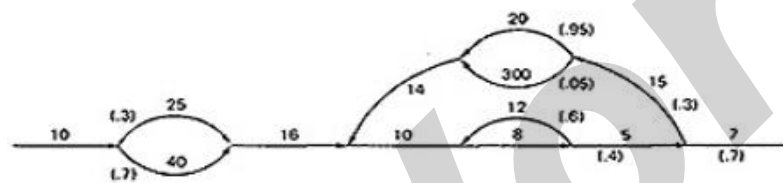
The model has two weights associated with every link: the processing time for that link, denoted by **T**, and the probability of that link **P**.

The arithmetic rules for calculating the mean time:

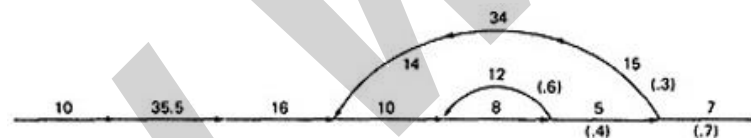
Case	Path expression	Weight expression
Parallel	$A+B$	$T_{A+B} = (P_A T_A + P_B T_B) / (P_A + P_B)$ $P_{A+B} = P_A + P_B$
Series	AB	$T_{AB} = T_A + T_B$ $P_{AB} = P_A P_B$
Loop	A^*	$T_A = T_A + T_L P_L / (1 - P_L)$ $P_A = P_A / (1 - P_L)$

EXAMPLE:

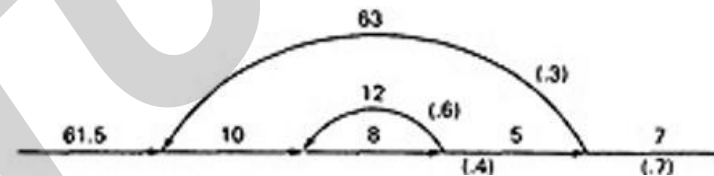
0. Start with the original flow graph annotated with probabilities and processing time.



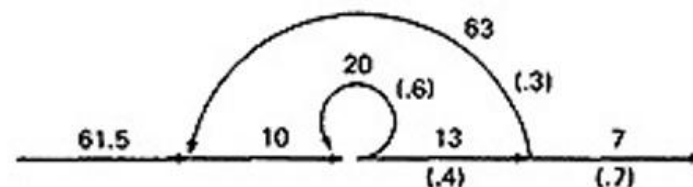
1. Combine the parallel links of the outer loop. The result is just the mean of the processing times for the links because there aren't any other links leaving the first node. Also combine the pair of links at the beginning of the flowgraph..



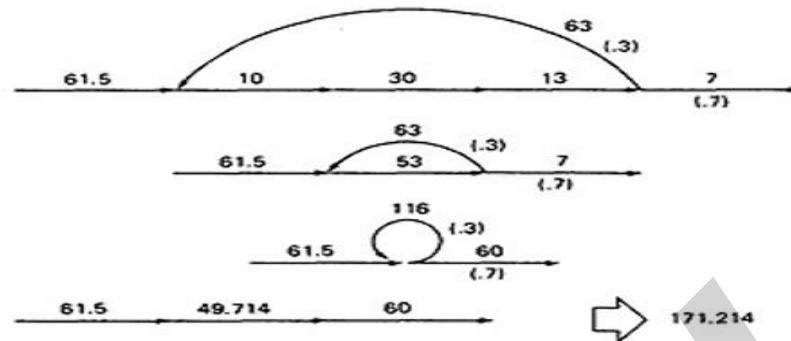
2. Combine as many serial links as you can.



3. Use the cross-term step to eliminate a node and to create the inner self-loop.



4. Finally, you can get the mean processing time, by using the arithmetic rules as follows:



PUSH/POP, GET/RETURN:

This model can be used to answer several different questions that can turn up in debugging. It can also help decide which test cases to design.

The question is:

Given a pair of complementary operations such as PUSH (the stack) and POP (the stack), considering the set of all possible paths through the routine, what is the net effect of the routine? PUSH or POP? How many times? Under what conditions?

Here are some other examples of complementary operations to which this model applies:

GET/RETURN a resource block.

OPEN/CLOSE a file.

START/STOP a device or process.

EXAMPLE 1 (PUSH / POP):

- Here is the Push/Pop Arithmetic:

Case	Path expression	Weight expression
Parallels	$A+B$	$W_A + W_B$
Series	AB	$W_A W_B$
Loop	A^*	W_A^*

- The numeral 1 is used to indicate that nothing of interest (neither PUSH nor POP) occurs on a given link.
- "H" denotes PUSH and "P" denotes POP. The operations are commutative, associative, and distributive.

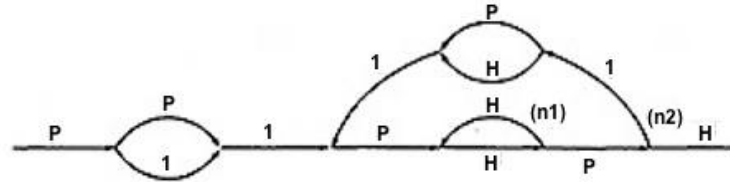
PUSH/POP MULTIPLICATION TABLE

X	H PUSH	P POP	1 NONE
H	H^2	1	H
P	1	P^2	P
1	H	P	1

PUSH/POP ADDITION TABLE

+	H PUSH	P POP	1 NONE
H	H	$P+H$	$H+1$
P	$P+H$	P	$P+1$
1	$H+1$	$P+1$	1

- Consider the following flowgraph:



$$P(P + 1)1\{P(HH)^{n1}HP1(P + H)1\}^{n2}P(HH)^{n1}HPH$$

- Simplifying by using the arithmetic tables,
- $= (P^2 + P)\{P(HH)^{n1}(P + H)\}^{n1}(HH)^{n1}$
- $= (P^2 + P)\{H^{2n1}(P^2 + 1)\}^{n2}H^{2n1}$
- Below Table 5.9 shows several combinations of values for the two looping terms - M1 is the number of times the inner loop will be taken and M2 the number of times the outer loop will be taken.

M ₁	M ₂	PUSH/POP
0	0	$P + P^2$
0	1	$P + P^2 + P^3 + P^4$
0	2	$\sum_{i=1}^6 P^i$
0	3	$\sum_{i=1}^8 P^i$
1	0	$1 + H$
1	1	$\sum_{i=0}^3 H^i$
1	2	$\sum_{i=0}^5 H^i$
1	3	$\sum_{i=0}^7 H^i$
2	0	$H^2 + H^3$
2	1	$\sum_{i=4}^7 H^i$
2	2	$\sum_{i=6}^{11} H^i$
2	3	$\sum_{i=8}^{16} H^i$

Figure 5.9: Result of the PUSH / POP Graph Analysis.

- These expressions state that the stack will be popped only if the inner loop is not taken.
- The stack will be left alone only if the inner loop is iterated once, but it may also be pushed.
- For all other values of the inner loop, the stack will only be pushed.

EXAMPLE 2 (GET / RETURN):

- Exactly the same arithmetic tables used for previous example are used for GET / RETURN a buffer block or resource, or, in fact, for any pair of complementary operations in which the total number of operations in either direction is cumulative.
- The arithmetic tables for GET/RETURN are:

Multiplication Table

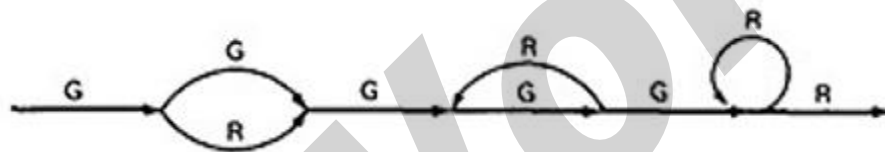
X	G	R	1
G	G^2	1	G
R	1	R^2	R
1	G	R	1

Addition Table

+	G	R	1
G	G	$G+R$	$G+1$
R	$G+R$	R	$R+1$
1	$G+1$	$R+1$	1

"G" denotes GET and "R" denotes RETURN.

- Consider the following flowgraph:



- $$\begin{aligned}
 &G(G \\
 &= \quad \quad \quad + \quad \quad \quad R)G(GR)*GGR*R \\
 &= \quad \quad \quad G(G \quad \quad \quad + \quad \quad \quad R)G^3R*R \\
 &= \quad \quad \quad (G \quad \quad \quad + \quad \quad \quad R)G^3R* \\
 &= (G^4 + G^2)R*
 \end{aligned}$$
- This expression specifies the conditions under which the resources will be balanced on leaving the routine.
- If the upper branch is taken at the first decision, the second loop must be taken four times.
- If the lower branch is taken at the first decision, the second loop must be taken twice.
- For any other values, the routine will not balance. Therefore, the first loop does not have to be instrumented to verify this behavior because its impact should be nil.

LIMITATIONS AND SOLUTIONS:

The main limitation to these applications is the problem of unachievable paths.

The node-by-node reduction procedure, and most graph-theory-based algorithms work well when all paths are possible, but may provide misleading results when some paths are unachievable.

The approach to handling unachievable paths (for any application) is to partition the graph into subgraphs so that all paths in each of the subgraphs are achievable.

The resulting subgraphs may overlap, because one path may be common to several different subgraphs.

Each predicate's truth-functional value potentially splits the graph into two subgraphs. For n predicates, there could be as many as 2^n subgraphs.

REGULAR EXPRESSIONS AND FLOW ANOMALY DETECTION:

• THE PROBLEM:

- The generic flow-anomaly detection problem (note: not just data-flow anomalies, but any flow anomaly) is that of looking for a specific sequence of options considering all possible paths through a routine.
- Let the operations be SET and RESET, denoted by s and r respectively, and we want to know if there is a SET followed immediately a SET or a RESET followed immediately by a RESET (an ss or an rr sequence).
- Some more application examples:

1. A file can be opened (o), closed (c), read (r), or written (w). If the file is read or written to after it's been closed, the sequence is nonsensical. Therefore, *cr* and *cw* are anomalous. Similarly, if the file is read before it's been written, just after opening, we may have a bug. Therefore, *or* is also anomalous. Furthermore, *oo* and *cc*, though not actual bugs, are a waste of time and therefore should also be examined.
2. A tape transport can do a rewind (d), fast-forward (f), read (r), write (w), stop (p), and skip (k). There are rules concerning the use of the transport; for example, you cannot go from rewind to fast-forward without an intervening stop or from rewind or fast-forward to read or write without an intervening stop. The following sequences are anomalous: *df*, *dr*, *dw*, *fd*, and *fr*. Does the flowgraph lead to anomalous sequences on any path? If so, what sequences and under what circumstances?
3. The data-flow anomalies discussed in Unit 4 requires us to detect the *dd*, *dk*, *kk*, and *ku* sequences. Are there paths with anomalous data flows?

• **THE METHOD:**

- Annotate each link in the graph with the appropriate operator or the null operator 1.
- Simplify things to the extent possible, using the fact that $a + a = a$ and $12 = 1$.
- You now have a regular expression that denotes all the possible sequences of operators in that graph. You can now examine that regular expression for the sequences of interest.
- **EXAMPLE:** Let A, B, C, be nonempty sets of character sequences whose smallest string is at least one character long. Let T be a two-character string of characters. Then if T is a substring of (i.e., if T appears within) AB^nC , then T will appear in AB^2C . (**HUANG's Theorem**)
- As an example, let

A	=	pp
B	=	srr
C	=	rp
T	=	ss

The theorem states that *ss* will appear in $pp(srr)^n rp$ if it appears in $pp(srr)^2 rp$.

- However, let

A	=	p	+	pp	+	ps
B	=	psr	+	ps(r	+	ps)
C	=					rp
T	=					p ⁴

Is it obvious that there is a p^4 sequence in AB^nC ? The theorem states that we have only to look at $(p + pp + ps)[psr + ps(r + ps)]^2 rp$. Multiplying out the expression and simplifying shows that there is no p^4 sequence.

- Incidentally, the above observation is an informal proof of the wisdom of looping twice discussed in Unit 2. Because data-flow anomalies are represented by two-character sequences, it follows the above theorem that looping twice is what you need to do to find such anomalies.

• **LIMITATIONS:**

- Huang's theorem can be easily generalized to cover sequences of greater length than two characters. Beyond three characters, though, things get complex and this method has probably reached its utilitarian limit for manual application.
- There are some nice theorems for finding sequences that occur at the beginnings and ends of strings but no nice algorithms for finding strings buried in an expression.
- Static flow analysis methods can't determine whether a path is or is not achievable. Unless the flow analysis includes symbolic execution or similar techniques, the impact of unachievable paths will not be included in the analysis.
- The flow-anomaly application, for example, doesn't tell us that there will be a flow anomaly - it tells us that if the path is achievable, then there will be a flow anomaly. Such analytical problems go away, of course, if you take the trouble to design routines for which all paths are achievable.

SUMMARY:

- A flowgraph annotated with link names for every link can be converted into a path expression that represents the set of all paths in that flowgraph. A node-by-node reduction procedure is used.
- By substituting link weights for all links, and using the appropriate arithmetic rules, the path expression is converted into an algebraic expression that can be used to determine the minimum and maximum number of possible paths in a flowgraph, the probability that a given node will be reached, the mean processing time of a routine, and other models.
- With different, suitable arithmetic rules, and by using complementary operators as weights for the links, the path expression can be converted into an expression that denotes, over the set of all possible paths, what the net effect of the routine is.
- With links annotated with the appropriate weights, the path expression is converted into a regular expression that denotes the set of all operator sequences over the set of all paths in a routine. Rules for determining whether a given sequence of operations are possible are given. In other words, we have a generalized flow-anomaly detection method that'll work for data-flow anomalies or any other flow anomaly.
- All flow analysis methods lose accuracy and utility if there are unachievable paths. Expand the accuracy and utility of your analytical tools by designs for which all paths are achievable. Such designs are always possible.

UNIT VI :

Logic Based Testing : Overview, decision tables, path expressions, kv charts, specifications.

LOGIC BASED TESTING:**OVERVIEW OF LOGIC BASED TESTING :**

- **INTRODUCTION:**

- The functional requirements of many programs can be specified by **decision tables**, which provide a useful basis for program and test design.
- Consistency and completeness can be analyzed by using boolean algebra, which can also be used as a basis for test design. Boolean algebra is trivialized by using **Karnaugh-Veitch charts**.
- "Logic" is one of the most often used words in programmers' vocabularies but one of their least used techniques.
- Boolean algebra is to logic as arithmetic is to mathematics. Without it, the tester or programmer is cut off from many test and design techniques and tools that incorporate those techniques.
- Logic has been, for several decades, the primary tool of hardware logic designers.
- Many test methods developed for hardware logic can be adapted to software logic testing. Because hardware testing automation is 10 to 15 years ahead of software testing automation, hardware testing methods and its associated theory is a fertile ground for software testing methods.
- As programming and test techniques have improved, the bugs have shifted closer to the process front end, to requirements and their specifications. These bugs range from 8% to 30% of the total and because they're first-in and last-out, they're the costliest of all.
- The trouble with specifications is that they're hard to express.
- Boolean algebra (also known as the sentential calculus) is the most basic of all logic systems.
- Higher-order logic systems are needed and used for formal specifications.
- Much of logical analysis can be and is embedded in tools. But these tools incorporate methods to simplify, transform, and check specifications, and the methods are to a large extent based on boolean algebra.
- **KNOWLEDGE BASED SYSTEM:**
 - The **knowledge-based system** (also expert system, or "artificial intelligence" system) has become the programming construct of choice for many applications that were once considered very difficult.

- Knowledge-based systems incorporate knowledge from a knowledge domain such as medicine, law, or civil engineering into a database. The data can then be queried and interacted with to provide solutions to problems in that domain.
- One implementation of knowledge-based systems is to incorporate the expert's knowledge into a set of rules. The user can then provide data and ask questions based on that data.
- The user's data is processed through the rule base to yield conclusions (tentative or definite) and requests for more data. The processing is done by a program called the **inference engine**.
- Understanding knowledge-based systems and their validation problems requires an understanding of formal logic.
- Decision tables are extensively used in business data processing; Decision-table preprocessors as extensions to COBOL are in common use; boolean algebra is embedded in the implementation of these processors.
- Although programmed tools are nice to have, most of the benefits of boolean algebra can be reaped by wholly manual means if you have the right conceptual tool: the Karnaugh-Veitch diagram is that conceptual tool.

DECISION TABLES:

- Figure 6.1 is a limited - entry decision table. It consists of four areas called the condition stub, the condition entry, the action stub, and the action entry.
- Each column of the table is a rule that specifies the conditions under which the actions named in the action stub will take place.
- The condition stub is a list of names of conditions.

		CONDITION ENTRY			
		RULE 1	RULE 2	RULE 3	RULE 4
CONDITION STUB	CONDITION 1	YES	YES	NO	NO
	CONDITION 2	YES	I	NO	I
	CONDITION 3	NO	YES	NO	I
	CONDITION 4	NO	YES	NO	YES
ACTION STUB	ACTION 1	YES	YES	NO	NO
	ACTION 2	NO	NO	YES	NO
	ACTION 3	NO	NO	NO	YES
		ACTION ENTRY			

Figure 6.1 : Examples of Decision Table.

- A more general decision table can be as below:

Printer troubleshooter									
		Rules							
Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognised	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

Figure 6.2 : Another Examples of Decision Table.

- A rule specifies whether a condition should or should not be met for the rule to be satisfied. "YES" means that the condition must be met, "NO" means that the condition must not be met, and "I" means that the condition plays no part in the rule, or it is immaterial to that rule.
- The action stub names the actions the routine will take or initiate if the rule is satisfied. If the action entry is "YES", the action will take place; if "NO", the action will not take place.
- The table in Figure 6.1 can be translated as follows: Action 1 will take place if conditions 1 and 2 are met and if conditions 3 and 4 are not met (rule 1) or if conditions 1, 3, and 4 are met (rule 2).
- "Condition" is another word for predicate.
- Decision-table uses "condition" and "satisfied" or "met". Let us use "predicate" and TRUE / FALSE.
- Now the above translations become:
 1. Action 1 will be taken if predicates 1 and 2 are true and if predicates 3 and 4 are false (rule 1), or if predicates 1, 3, and 4 are true (rule 2).
 2. Action 2 will be taken if the predicates are all false, (rule 3).
 3. Action 3 will take place if predicate 1 is false and predicate 4 is true (rule 4).
- In addition to the stated rules, we also need a **Default Rule** that specifies the default action to be taken when all other rules fail. The default rules for Table in Figure 6.1 is shown in Figure 6.3

	Rule 5	Rule 6	Rule 7	Rule 8
CONDITION 1	I	NO	YES	YES
CONDITION 2	I	YES	I	NO
CONDITION 3	YES	I	NO	NO
CONDITION 4	NO	NO	YES	I
DEFAULT ACTION	YES	YES	YES	YES

Figure 6.3 : The default rules of Table in Figure 6.1

- **DECISION-TABLE PROCESSORS:**
 - Decision tables can be automatically translated into code and, as such, are a higher-order language
 - If the rule is satisfied, the corresponding action takes place
 - Otherwise, rule 2 is tried. This process continues until either a satisfied rule results in an action or no rule is satisfied and the default action is taken
 - Decision tables have become a useful tool in the programmers kit, in business data processing.

DECISION-TABLES AS BASIS FOR TEST CASE DESIGN:

0. The specification is given as a decision table or can be easily converted into one.
1. The order in which the predicates are evaluated does not affect interpretation of the rules or the resulting action - i.e., an arbitrary permutation of the predicate order will not, or should not, affect which action takes place.
2. The order in which the rules are evaluated does not affect the resulting action - i.e., an arbitrary permutation of rules will not, or should not, affect which action takes place.
3. Once a rule is satisfied and an action selected, no other rule need be examined.
4. If several actions can result from satisfying a rule, the order in which the actions are executed doesn't matter

DECISION-TABLES AND STRUCTURE:

- Decision tables can also be used to examine a program's structure.
- Figure 6.4 shows a program segment that consists of a decision tree.
- These decisions, in various combinations, can lead to actions 1, 2, or 3.

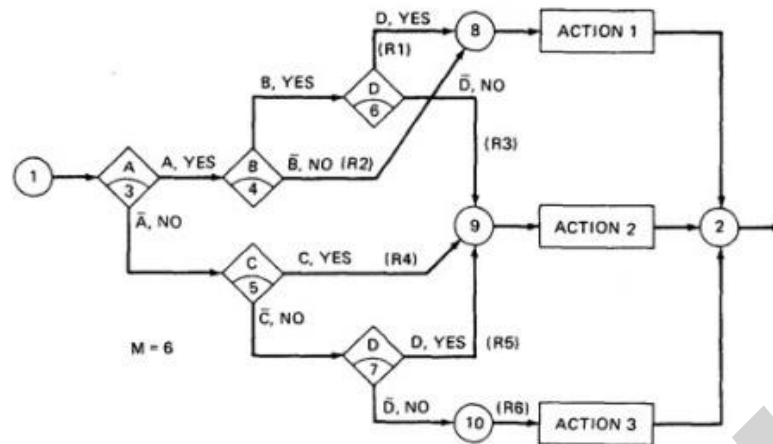


Figure 6.4 : A Sample Program

- If the decision appears on a path, put in a YES or NO as appropriate. If the decision does not appear on the path, put in an I, Rule 1 does not contain decision C, therefore its entries are: YES, YES, I, YES.
- The corresponding decision table is shown in Table 6.1

		RULE 1	RULE 2	RULE 3	RULE 4	RULE 5	RULE 6
CONDITION A		YES	YES	YES	NO	NO	NO
CONDITION B		YES	NO	YES	I	I	I
CONDITION C		I	I	I	YES	NO	NO
CONDITION D		YES	I	NO	I	YES	NO
ACTION 1		YES	YES	NO	NO	NO	NO
ACTION 2		NO	NO	YES	YES	YES	NO
ACTION 3		NO	NO	NO	NO	NO	YES

Table 6.1 : Decision Table corresponding to Figure 6.4

- As an example, expanding the immaterial cases results as below:

	RULE 1	RULE 2		RULE 1.1	RULE 1.2	RULE 2.1	RULE 2.2
CONDITION 1	YES	YES		YES	YES	YES	YES
CONDITION 2	I	NO		YES	NO	NO	NO
CONDITION 3	YES	I		YES	YES	YES	NO
CONDITION 4	NO	NO		NO	NO	NO	NO
ACTION 1	YES	NO		YES	YES	NO	NO
ACTION 2	NO	YES		NO	NO	YES	YES

- Similarly, If we expand the immaterial cases for the above Table 6.1, it results in Table 6.2 as below:

		R 1	RULE 2	R 3	RULE 4	R 5	R 6
CONDITION A		YY	YYYY	YY	NNNN	NN	NN

CONDITION B	YY	NNNN	YY	YYNN	NY	YN
CONDITION C	YN	NNYY	YN	YYYY	NN	NN
CONDITION D	YY	YNNY	NN	NYYN	YY	NN

○ **Table 6.2 : Expansion of Table 6.1**

- Sixteen cases are represented in Table 6.1, and no case appears twice.
- Consequently, the flowgraph appears to be complete and consistent.
- As a first check, before you look for all sixteen combinations, count the number of Y's and N's in each row. They should be equal. We can find the bug that way.

ANOTHER EXAMPLE - A TROUBLE SOME PROGRAM:

- Consider the following specification whose putative flowgraph is shown in Figure 6.5:
 1. If condition A is met, do process A1 no matter what other actions are taken or what other conditions are met.
 2. If condition B is met, do process A2 no matter what other actions are taken or what other conditions are met.
 3. If condition C is met, do process A3 no matter what other actions are taken or what other conditions are met.
 4. If none of the conditions is met, then do processes A1, A2, and A3.
 5. When more than one process is done, process A1 must be done first, then A2, and then A3. The only permissible cases are: (A1), (A2), (A3), (A1,A3), (A2,A3) and (A1,A2,A3).
- Figure 6.5 shows a sample program with a bug.

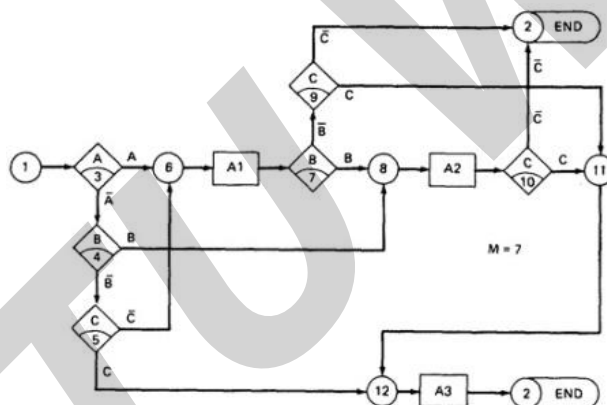


Figure 6.5 : A Troublesome Program

- The programmer tried to force all three processes to be executed for the $\bar{A}\bar{B}\bar{C}$ cases but forgot that the B and C predicates would be done again, thereby bypassing processes A2 and A3.
- Table 6.3 shows the conversion of this flowgraph into a decision table after expansion.

RULES								
	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$	$\bar{A}B\bar{C}$	$\bar{A}BC$	$A\bar{B}\bar{C}$	$A\bar{B}C$	$AB\bar{C}$	ABC
CONDITION A	NO	NO	NO	NO	YES	YES	YES	YES
CONDITION B	NO	NO	YES	YES	YES	YES	NO	YES
CONDITION C	NO	YES	YES	NO	NO	YES	YES	NO
ACTION 1	YES	NO	NO	NO	YES	YES	YES	YES
ACTION 2	YES	NO	YES	YES	YES	YES	NO	NO
ACTION 3	YES	YES	YES	NO	NO	YES	YES	NO

Table 6.3 : Decision Table for Figure 6.5

PATH EXPRESSIONS:

- GENERAL:**

- Logic-based testing is structural testing when it's applied to structure (e.g., control flowgraph of an implementation); it's functional testing when it's applied to a specification.
- In logic-based testing we focus on the truth values of control flow predicates.
- A **predicate** is implemented as a process whose outcome is a truth-functional value.
- For our purpose, logic-based testing is restricted to binary predicates.
- We start by generating path expressions by path tracing as in Unit V, but this time, our purpose is to convert the path expressions into boolean algebra, using the predicates' truth values (e.g., A and \bar{A}) as weights.

- BOOLEAN ALGEBRA:**

- STEPS:**

- Label each decision with an uppercase letter that represents the truth value of the predicate. The YES or TRUE branch is labeled with a letter (say A) and the NO or FALSE branch with the same letter overscored (say \bar{A}).
- The truth value of a path is the product of the individual labels. Concatenation or products mean "AND". For example, the straight-through path of Figure 6.5, which goes via nodes 3, 6, 7, 8, 10, 11, 12, and 2, has a truth value of ABC. The path via nodes 3, 6, 7, 9 and 2 has a value of $AB\bar{C}$.
- If two or more paths merge at a node, the fact is expressed by use of a plus sign (+) which means "OR".

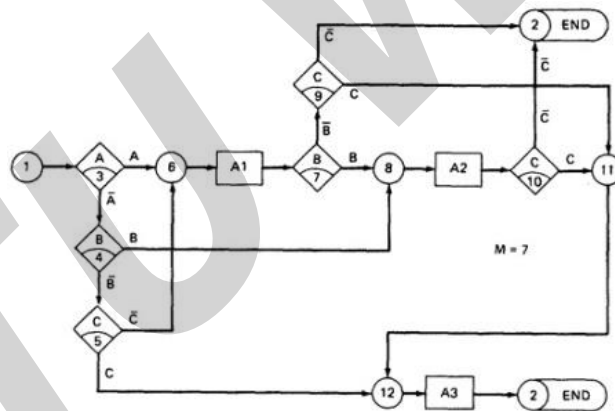


Figure 6.5 : A Troublesome Program

- Using this convention, the truth-functional values for several of the nodes can be expressed in terms of segments from previous nodes. Use the node name to identify the point.

$$\begin{aligned}
 N6 &= A + \bar{A}\bar{B}\bar{C} \\
 N8 &= (N6)B + \bar{A}B = AB + \bar{A}\bar{B}\bar{C}B + \bar{A}B \\
 N11 &= (N8)C + (N6)\bar{B}C \\
 N12 &= N11 + \bar{A}\bar{B}C \\
 N2 &= N12 + (N8)\bar{C} + (N6)\bar{B}\bar{C}
 \end{aligned}$$

- There are only two numbers in boolean algebra: zero (0) and one (1). One means "always true" and zero means "always false".
- RULES OF BOOLEAN ALGEBRA:**

- Boolean algebra has three operators: \times (AND), $+$ (OR) and $\bar{}$ (NOT)
- \times : meaning AND. Also called multiplication. A statement such as AB ($A \times B$) means "A and B are both true". This symbol is usually left out as in ordinary algebra.
- $+$: meaning OR. " $A + B$ " means "either A is true or B is true or both".
- $\bar{}$ meaning NOT. Also negation or complementation. This is read as either "not A" or "A bar". The entire expression under the bar is negated.
- The following are the laws of boolean algebra:

1. $\frac{A + A}{\bar{A} + \bar{A}}$	$= \frac{A}{\bar{A}}$	If something is true, saying it twice doesn't make it truer, ditto for falsehoods. If something is always true, then "either A or true or both" must also be universally true.
2. $A + 1$	$= 1$	
3. $A + 0$	$= A$	Commutative law. If either A is true or not-A is true, then the statement is always true.
4. $A + B$	$= B + A$	
5. $A + \bar{A}$	$= 1$	A statement can't be simultaneously true and false. "You ain't not going" means you are. How about, "I ain't not never going to get this nohow."?
6. $\frac{AA}{\bar{A}\bar{A}}$	$= \frac{A}{\bar{A}}$	
7. $A \times 1$	$= A$	Called "De Morgan's theorem or law."
8. $A \times 0$	$= 0$	
9. AB	$= BA$	Distributive law. Multiplication is associative. So is addition.
10. $A\bar{A}$	$= 0$	
11. $\bar{\bar{A}}$	$= A$	Absorptive law.
12. $\bar{0}$	$= 1$	
13. $\bar{1}$	$= 0$	Distributive law. Multiplication is associative. So is addition.
14. $\overline{A + B}$	$= \bar{A}\bar{B}$	
15. \overline{AB}	$= \bar{A} + \bar{B}$	Absorptive law.
16. $A(B + C)$	$= AB + AC$	
17. $(AB)C$	$= A(BC)$	Absorptive law.
18. $(A + B) + C$	$= A + (B + C)$	
19. $A + \bar{A}B$	$= A + B$	Absorptive law.
20. $A + AB$	$= A$	

In all of the above, a letter can represent a single sentence or an entire boolean algebra expression.

Individual letters in a boolean algebra expression are called **Literals** (e.g. A,B)

The product of several literals is called a **product term** (e.g., ABC, DE).

An arbitrary boolean expression that has been multiplied out so that it consists of the sum of products (e.g., $ABC + DEF + GH$) is said to be in **sum-of-products form**.

The result of simplifications (using the rules above) is again in the sum of product form and each product term in such a simplified version is called a **prime implicant**. For example, $ABC + AB + DEF$ reduces by rule 20 to $AB + DEF$; that is, AB and DEF are prime implicants.

The path expressions of Figure 6.5 can now be simplified by applying the rules.

The following are the laws of boolean algebra:

$$\begin{aligned}
 N6 &= A + \overline{A} \overline{B} \overline{C} \\
 &= A + \overline{B} \overline{C} && : \text{Use rule 19, with "B" = } \overline{B} \overline{C}. \\
 N8 &= (N6)B + \overline{A} B \\
 &= (A + \overline{B} \overline{C})B + \overline{A} B && : \text{Substitution.} \\
 &= AB + \overline{B} \overline{C} B + \overline{A} B && : \text{Rule 16 (distributive law).} \\
 &= AB + \overline{B} \overline{C} B + \overline{A} B && : \text{Rule 9 (commutative multiplication).} \\
 &= AB + 0C + \overline{A} B && : \text{Rule 10.} \\
 &= AB + 0 + \overline{A} B && : \text{Rule 8.} \\
 &= AB + \overline{A} B && : \text{Rule 3.} \\
 &= (A + \overline{A})B && : \text{Rule 16 (distributive law).} \\
 &= 1 \times B && : \text{Rule 5.} \\
 &= B && : \text{Rules 7, 9.}
 \end{aligned}$$

Similarly,

$$\begin{aligned}
 N11 &= (N8)C + (N6)\overline{B} \overline{C} \\
 &= BC + (A + \overline{B} \overline{C})\overline{B} \overline{C} && : \text{Substitution.} \\
 &= BC + A\overline{B} \overline{C} && : \text{Rules 16, 9, 10, 8, 3.} \\
 &= C(B + \overline{B} \overline{A}) && : \text{Rules 9, 16.} \\
 &= C(B + A) && : \text{Rule 19.} \\
 &= AC + BC && : \text{Rules 16, 9, 9, 4.} \\
 N12 &= N11 + \overline{A} \overline{B} \overline{C} \\
 &= AC + BC + \overline{A} \overline{B} \overline{C} \\
 &= C(B + \overline{A} \overline{B}) + AC \\
 &= C(\overline{A} + B) + AC \\
 &= C\overline{A} + AC + BC \\
 &= C + BC \\
 &= C \\
 N2 &= N12 + (N8)\overline{C} + (N6)\overline{B} \overline{C} \\
 &= C + \overline{B} \overline{C} + (A + \overline{B} \overline{C})\overline{B} \overline{C} \\
 &= C + \overline{B} \overline{C} + \overline{B} \overline{C} \\
 &= C + \overline{C}(B + \overline{B}) \\
 &= C + \overline{C} \\
 &= 1
 \end{aligned}$$

The deviation from the specification is now clear. The functions should have been:

$$\begin{aligned}
 N6 &= A + \overline{A} \overline{B} \overline{C} = A + \overline{B} \overline{C} && : \text{correct.} \\
 N8 &= B + \overline{A} \overline{B} \overline{C} = B + \overline{A} \overline{C} && : \text{wrong, was just B.} \\
 N12 &= C + \overline{A} \overline{B} \overline{C} = C + \overline{A} \overline{B} && : \text{wrong, was just C.}
 \end{aligned}$$

Loops complicate things because we may have to solve a boolean equation to determine what predicate-value combinations lead to where.

KV CHARTS:

- **INTRODUCTION:**

- If you had to deal with expressions in four, five, or six variables, you could get bogged down in the algebra and make as many errors in designing test cases as there are bugs in the routine you're testing.
- **Karnaugh-Veitch chart** reduces boolean algebraic manipulations to graphical trivia.
- Beyond six variables these diagrams get cumbersome and may not be effective.

- **SINGLE VARIABLE:**

- Figure 6.6 shows all the boolean functions of a single variable and their equivalent representation as a KV chart.

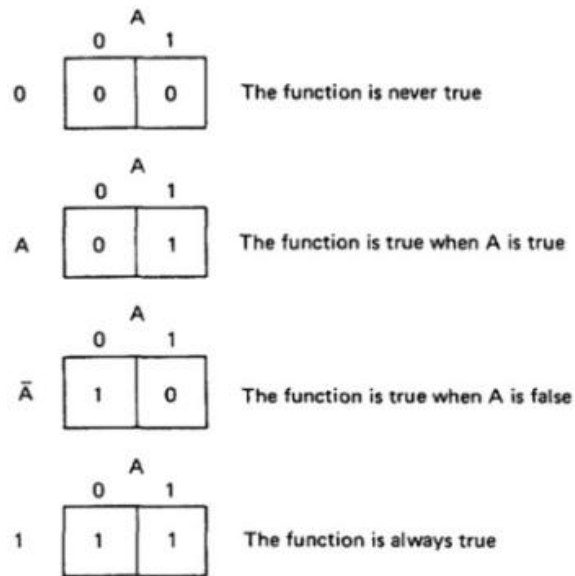


Figure 6.6 : KV Charts for Functions of a Single Variable.

- The charts show all possible truth values that the variable A can have.
 - A "1" means the variable's value is "1" or TRUE. A "0" means that the variable's value is 0 or FALSE.
 - The entry in the box (0 or 1) specifies whether the function that the chart represents is true or false for that value of the variable.
 - We usually do not explicitly put in 0 entries but specify only the conditions under which the function is true.
- **TWO VARIABLES:**
 - Figure 6.7 shows eight of the sixteen possible functions of two variables.

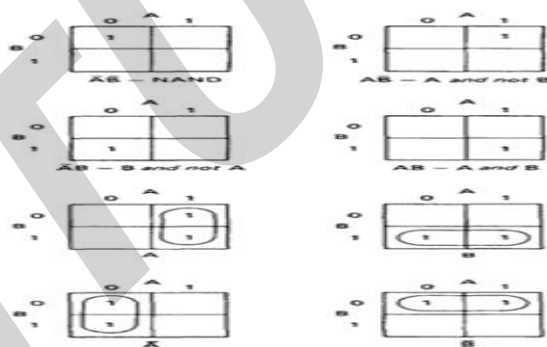


Figure 6.7 : KV Charts for Functions of Two Variables.

- Each box corresponds to the combination of values of the variables for the row and column of that box.
 - A pair may be adjacent either horizontally or vertically but not diagonally.
 - Any variable that changes in either the horizontal or vertical direction does not appear in the expression.
 - In the fifth chart, the B variable changes from 0 to 1 going down the column, and because the A variable's value for the column is 1, the chart is equivalent to a simple A.
 - Figure 6.8 shows the remaining eight functions of two variables.

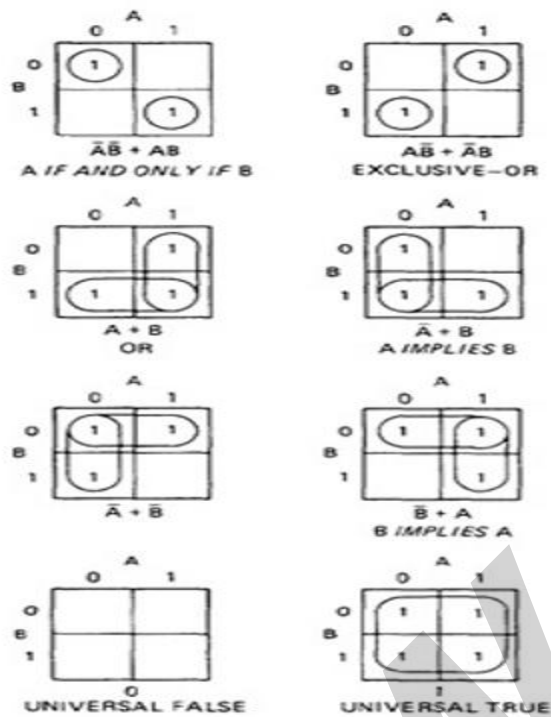
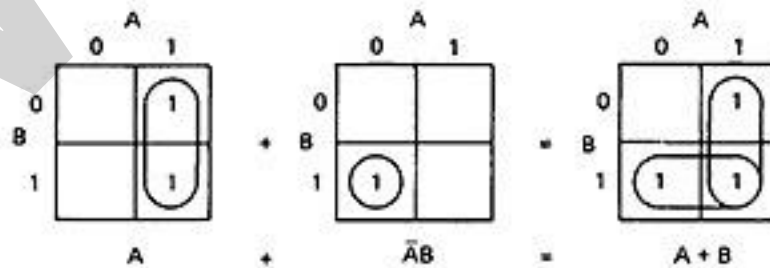
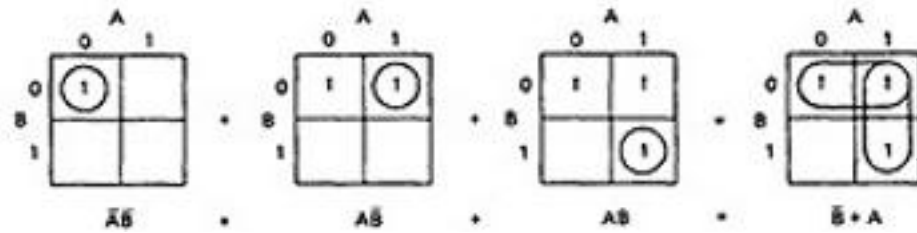


Figure 6.8 : More Functions of Two Variables.

- The first chart has two 1's in it, but because they are not adjacent, each must be taken separately.
- They are written using a plus sign.
- It is clear now why there are sixteen functions of two variables.
- Each box in the KV chart corresponds to a combination of the variables' values.
- That combination might or might not be in the function (i.e., the box corresponding to that combination might have a 1 or 0 entry).
- Since n variables lead to 2^n combinations of 0 and 1 for the variables, and each such combination (box) can be filled or not filled, leading to 2^{2^n} ways of doing this.
- Consequently for one variable there are $2^{2^1} = 4$ functions, 16 functions of 2 variables, 256 functions of 3 variables, 16,384 functions of 4 variables, and so on.
- Given two charts over the same variables, arranged the same way, their product is the term by term product, their sum is the term by term sum, and the negation of a chart is gotten by reversing all the 0 and 1 entries in the chart.

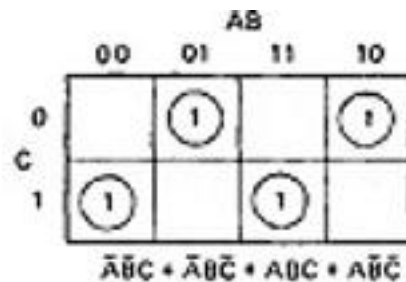
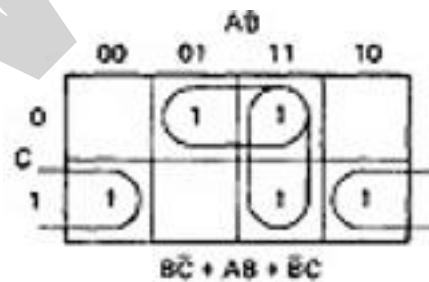
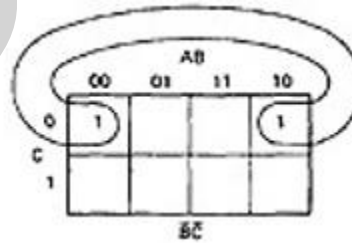
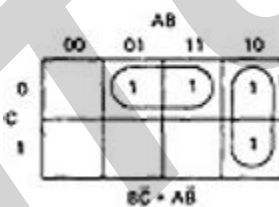
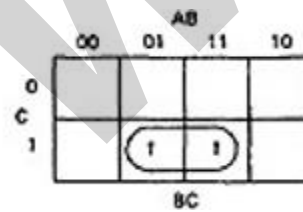
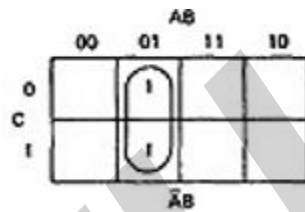
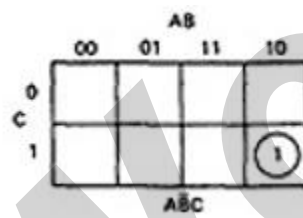
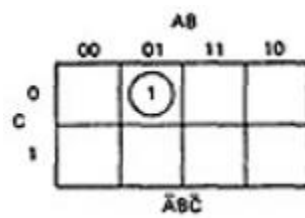


OR



• **THREE VARIABLES:**

- KV charts for three variables are shown below.
- As before, each box represents an elementary term of three variables with a bar appearing or not appearing according to whether the row-column heading for that box is 0 or 1.
- A three-variable chart can have groupings of 1, 2, 4, and 8 boxes.
- A few examples will illustrate the principles:



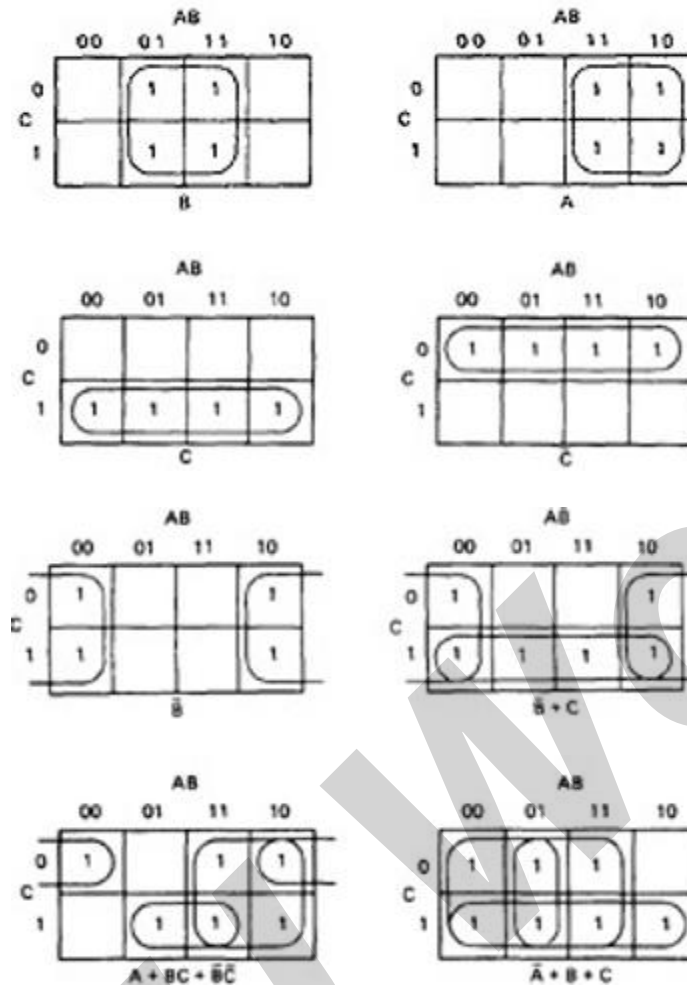


Figure 6.8 : KV Charts for Functions of Three Variables.

- You'll notice that there are several ways to circle the boxes into maximum-sized covering groups.

TESTABILITY TIPS

Logic-intensive software designed by the seat of the pants is almost never right. We learned this lesson decades ago in the simpler hardware logic design arena. It is in our interest as software engineers to use the simplest possible predicate expressions in our design. The objective is not to simplify the code in order to save a few bytes of memory but to reduce the opportunities for bugs. Hardware logic designers learned that there were many advantages to designing their logic in a **canonical form**—that is, a form that followed certain rules. The testability considerations of this chapter apply to loop-free software, or to the portion of the software that is loop-free; for example, a logic-intensive program segment within a loop can be examined by these means. You can start either from specifications or, if you're doing a redesign, from code. I'll speak to the latter case because it's more general. Think in terms of redesign if you have sensitization difficulties.

1. Identify your predicates (simple or compound).
2. If starting from code, get a branch covering set of path predicates.
3. Interpret the predicates so that they are expressed in terms of the input vector for the chosen path.
4. Simplify the path predicate expression for each selected path. If any expression is logically zero, the path is unachievable. Pick another path or paths to achieve branch coverage.

5. If any path predicate expression equals logical 1 then all other paths must be unachievable—find and fix the design bug.
6. The logical sum of the path predicate expressions must equal 1 or else there is an unsuspected loop, dangling code, or branch coverage is an inadequate test criterion.

The canonical processor has three successive stages:

1. Predicate calculator.
2. Logic analyzer.
3. Domain processor.

The **predicate calculator** transforms (e.g., processes) the input vector to get the values of the variables that are actually used in the predicates. Every predicate is evaluated exactly once, so that its truth value is known. The **logic analyzer** forms the predicate expression appropriate to the case and directs the control flow to the appropriate domain processor. Because each predicate defines a domain boundary and each predicate expression defines a domain, there is a one-to-one correspondence between the various outcomes of the logic analyzer and the domains. The **domain processor** does the processing appropriate to each domain, for example, with a separate hunk of code for each domain. Only one control-flow statement (a case statement) is needed—one case, one predicate expression, one domain. The canonical form, if it is achieved, has the following obvious advantages:

1. Branch coverage and all-paths coverage are identical.
2. All paths are achievable and easy to sensitize.
3. Separation simplifies maintenance.

The above canonical form is an ideal that you cannot expect to achieve. Achieving it could mean redundant software, excessive nesting depth (if you encapsulate the redundancies in subroutines), or slow execution on some paths; conversely, however, the canonical form can be faster and tighter. You may be able to achieve it locally, or globally but not both; but you don't know unless you try. And why try? Because it works. The proof comes from hardware design, where we learned, three decades ago, that seat-of-the-pants logic was buggy, slow, dangerous, and hard to build, test, and maintain.

SUMMARY

1. Use decision tables as a convenient way to organize statements in a specification—possibly as an intermediate step toward a more compact and more revealing equivalent boolean algebra expression.
2. Label the links following binary decisions with a weight that corresponds to the predicate's logical value, and evaluate the boolean expressions to the nodes of interest.
3. Simplify the resulting expressions or solve equations and then simplify if you cannot directly express the boolean function for the node in terms of the path predicate values.
4. The boolean expression for the exit node should equal 1. If it does not, or if attempting to solve for it leads to a loop of equations, then there are conditions under which the routine will loop indefinitely. The negation of the exit expression specifies all the combinations of predicate values that will lead to the loop or loops.
5. Any node of interest can be reached by a test case derived from the expansion of any prime implicant in the boolean expression for that node.
6. The set of all paths from the entry to a node can be obtained by expanding all the prime implicants of the boolean expression that corresponds to that node. A branch-covering set of paths, however, may not require all the terms of the expansion.
7. You don't do boolean algebra by algebra. You use KV charts for up to six variables. Keep quadrille-ruled paper pads handy.
8. For logic-intensive routines, examine specification completeness and consistency by using boolean algebra via KV charts. Use the canonical form as a model of clean logic.

9. Be careful in translating English into boolean algebra. Retranslate and discuss the retranslation of the algebra with the specifier. Be tricky and use alternate, logically equivalent forms to see whether they (specifiers) are consistent and whether they really want what they say they want.

10. Question all missing entries, question overlapped entries if there was no explicit statement of multiple actions, question all almost-complete groups.

11. Don't take advantage of don't-care cases or impossible cases unless you're willing to pay the maintenance penalties; but if you must, get the maximum payoff by making the resulting logic as simple as you can and document all instances in which you take advantage of don't-care cases.

UNIT VII :

State, State Graphs and Transition testing : State graphs, good & bad state graphs, state testing, Testability tips.

STATES, STATE GRAPHS, AND TRANSITION TESTING

1. SYNOPSIS

The **state graph** and its associated **state table** are useful models for describing software behavior. The **finite-state machine** is a functional testing tool and testable design programming tool. Methods analogous to path testing are described and discussed.

2. MOTIVATIONAL OVERVIEW

The **finite-state machine** is as fundamental to software engineering as boolean algebra. State testing strategies are based on the use of finite-state machine models for software structure, software behavior, or specifications of software behavior. Finite-state machines can also be implemented as table-driven software, in which case they are a powerful design option. Independent testers are likeliest to use a finite-state machine model as a guide to the design of functional tests—especially system tests. Software designers are likelier to want to exploit and test finite-state machine software implementations. Finally, finite-state machine models of software abound in the testing literature, much of which will be meaningless to readers who don't know this subject. Among the affected testing topics are protocols, concurrent systems, system failure and recovery, system configuration, and distributed data bases (BARN72, DAVI88A, HOLZ87, PETE76).

3. STATE GRAPHS

3.1. States

The word “**state**” is used in much the same way it's used in ordinary English, as in “state of the union,” or “state of health.” The Oxford English Dictionary defines “state” as: “A combination of circumstances or attributes belonging for the time being to a person or thing.”

A program that detects the character sequence “ZCZC” can be in the following states:

1. Neither ZCZC nor any part of it has been detected.
2. Z has been detected.
3. ZC has been detected.
4. ZCZ has been detected.
5. ZCZC has been detected.

A moving automobile whose engine is running can have the following states with respect to its transmission:

1. Reverse gear
2. Neutral gear
3. First gear
4. Second gear
5. Third gear
6. Fourth gear

A person's checkbook can have the following states with respect to the bank balance:

1. Equal
2. Less than
3. Greater than

A word processing program menu can be in the following states with respect to file manipulation:

1. Copy document
2. Delete document
3. Rename document
4. Create document
5. Compress document
6. Copy disc
7. Format disc
8. Backup disc
9. Recover from backup

States are represented by **nodes**. States are numbered or may be identified by words or whatever else is convenient. Figure 11.1 shows a typical **state graph**. The automobile example is really more complicated because: (1) the engine might or might not be running, (2) the car itself might be moving forward or backward or be stopped, and (3) the clutch might or might not be depressed. These factors multiply the above six states by $2 \times 3 \times 2 = 12$, for a total of 72 rather than 6 states. Each additional factor that has alternatives multiplies the number of states in a model by the number of alternatives. The number of states of a computer is 2 raised to the power of the number of bits in the computer; that is, all the bits in main memory, registers, discs, tapes, and so on. Because most interesting factors are binary, and because each factor doubles the number of states, state graphs are most useful for relatively simple functional models involving at most a few dozen states and only a few factors.

3.2. Inputs and Transitions

Whatever is being modeled is subjected to inputs. As a result of those inputs, the state changes, or is said to have made a **transition**. Transitions are denoted by links that join the states. The input that causes the transition are marked on the link; that is, the inputs are link weights. There is one outlink from every state for every input. If several inputs in a state cause a transition to the same subsequent state, instead of drawing a bunch of parallel links we can abbreviate the notation by listing the several inputs as in: "input1, input2, input3. . .". A **finite-state machine** is an abstract device that can be represented by a state graph having a finite number of states and a finite number of transitions between states.

The ZCZC detection example can have the following kinds of inputs:

1. Z
2. C
3. Any character other than Z or C, which we'll denote by A

The state graph of Figure 11.1 is interpreted as follows:

1. If the system is in the “NONE” state, any input other than a Z will keep it in that state.
2. If a Z is received, the system transitions to the “Z” state.
3. If the system is in the “Z” state and a Z is received, it will remain in the “Z” state. If a C is received, it will go to the “ZC” state; if any other character is received, it will go back to the “NONE” state because the sequence has been broken.
4. A Z received in the “ZC” state progresses to the “ZCZ” state, but any other character breaks the sequence and causes a return to the “NONE” state.
5. A C received in the “ZCZ” state completes the sequence and the system enters the “ZCZC” state. A Z breaks the sequence and causes a transition back to the “Z” state; any other character causes a return to the “NONE” state.
6. The system stays in the “ZCZC” state no matter what is received.

As you can see, the state graph is a compact representation of all this verbiage.

3.3. Outputs

An output^{*} can be associated with any link. Outputs are denoted by letters or words and are separated from inputs by a slash as follows: “input/output.” As always, “output” denotes anything of interest that’s observable and is not restricted to explicit outputs by devices. Outputs are also link weights. If every input associated with a transition causes the same output, then denote it as: “input 1, input 2, . . . input 3/output.” If there are many different combinations of inputs and outputs, it’s best to draw a separate parallel link for each output.

^{*}“Output” rather than “outcome” because the outcome consists of the output *and* a transition to the new state. “Output” used in this context can mean almost anything observable and is not restricted to tangible outputs by devices, say.

Consider now, as an example, a simplified specification for a tape transport write-error recovery procedure, such as might be found in a tape driver routine:^{**}

^{**}Our objective here is not to design a tape driver but to illustrate how a specification, good or bad, sensible or not, can be modeled by a state graph.

“If no write errors are detected, (input = OK), no special action is taken (output = NONE). If a write error is detected (input = ERROR), backspace the tape one block and rewrite the block (output = REWRITE). If the rewrite is successful (input = OK), ignore the fact that there has been a rewrite. If the rewrite is not successful, try another backspace and rewrite. Return to the original state if and only if there have been two successive successful writes. If there have been two successive rewrites and a third error occurs, backspace ten centimeters and erase forward from that point (output = ERASE). If the erasure works (input = OK), return to the initial state. If it does not work, backspace another ten centimeters, erase and treat the next write attempt as for the first erasure. If the second erasure does not clear the problem, put the tape transport out of service.”

The state graph is shown in Figure 11.2. As in the previous example, the inputs and actions have been simplified. There are only two kinds of inputs (OK, ERROR) and four kinds of outputs (REWRITE, ERASE, NONE, OUT-OF-SERVICE). Don’t confuse outputs with transitions or states. This can be confusing because sometimes the name of the output is the same as the name of a state.^{*} Similarly, don’t confuse the input with the state, as in the first transition and the second state of the ZCZC detection example.

^{*}An alternate, but equivalent, representation of behavior, called a “Moore model” (MOOR56), associates outputs with states rather than with transitions. The model used in this book is called a “Mealy model” (MEAL55), in which outputs are associated with transitions. Mealy models are more useful because of the way software of this kind is usually implemented. Also, the Mealy model makes both inputs and outputs link weights, which makes it easier to use the methods of Chapter 12 for analysis.

3.4. State Tables

Big state graphs are cluttered and hard to follow. It's more convenient to represent the state graph as a table (the **state table** or **state-transition table**) that specifies the states, the inputs, the transitions, and the outputs. The following conventions are used:

INPUT		
STATE	OKAY	ERROR
1	1/NONE	2/REWRITE
2	1/NONE	4/REWRITE
3	1/NONE	2/REWRITE
4	3/NONE	5/ERASE
5	1/NONE	6/ERASE
6	1/NONE	7/OUT
7

Table 11.1. State Table for Figure 11.2.

1. Each row of the table corresponds to a state.
2. Each column corresponds to an input condition.
3. The box at the intersection of a row and column specifies the next state (the transition) and the output, if any.

The state table for the tape control is shown in Table 11.1.

I didn't specify what happens in state 7 because it's not germane to the discussion. You would have to complete the state graph for that state and for all the other states (not shown) that would be needed to get the tape back into operation. Compare the tabular representation with the graphical representation so that you can follow the action in either notation.

3.5. Time Versus Sequence

State graphs don't represent time—they represent sequence. A transition might take microseconds or centuries; a system could be in one state for milliseconds and another for eons, or the other way around; the state graph would be the same because it has no notion of time. Although the finite-state machine model can be elaborated to include notions of time in addition to sequence, such as timed Petri nets (DAVI88A, MURA89, PETE81), the subject is beyond the scope of this book.

3.6. Software Implementation

3.6.1. Implementation and Operation

There is rarely a direct correspondence between programs and the behavior of a process described as a state graph. In the tape driver example, for instance, the inputs would occur over a period of time. The routine is probably activated by an executive, and the inputs might be status-return interrupts from the tape control hardware. Alternatively, the inputs might appear as status bits in a word in memory reserved for that transport. The tape control routine itself is probably reentrant, so it can be used simultaneously by all transports.

The state graph represents the total behavior consisting of the transport, the software, the executive, the status returns, interrupts, and so on. There is no simple correspondence between lines of code and states. The state table,

however, forms the basis for a widely used implementation shown in the PDL program below. There are four tables involved:

1. A table or process that encodes the input values into a compact list (INPUT_CODE_TABLE).
2. A table that specifies the next state for every combination of state and input code (TRANSITION_TABLE).
3. A table or case statement that specifies the output or output code, if any, associated with every state-input combination (OUTPUT_TABLE).
4. A table that stores the present state of every device or process that uses the same state table—e.g., one entry per tape transport (DEVICE_TABLE).

The routine operates as follows, where # means concatenation:

```
BEGIN
PRESENT_STATE := DEVICE_TABLE(DEVICE_NAME)
ACCEPT INPUT_VALUE
INPUT_CODE := INPUT_CODE_TABLE(INPUT_VALUE)
POINTER := INPUT_CODE#PRESENT_STATE
NEW_STATE := TRANSITION_TABLE(POINTER)
OUTPUT_CODE := OUTPUT_TABLE(POINTER)
CALL OUTPUT_HANDLER(OUTPUT_CODE)
DEVICE_TABLE(DEVICE_NAME) := NEW_STATE
END
```

1. The present state is fetched from memory.
2. The present input value is fetched. If it is already numerical, it can be used directly; otherwise, it may have to be encoded into a numerical value, say by use of a case statement, a table, or some other process.
3. The present state and the input code are combined (e.g., concatenated) to yield a pointer (row and column) of the transition table and its logical image (the output table).
4. The output table, either directly or via a case statement, contains a pointer to the routine to be executed (the output) for that state-input combination. The routine is invoked (possibly a trivial routine if no output is required).
5. The same pointer is used to fetch the new state value, which is then stored.

There could be a lot of code between the end of this flow and the start of a new pass. Typically, there would be a return to the executive, and the state-control routine would only be invoked upon an interrupt. Many variations are possible. Sometimes, no input encoding is required. In other situations, the invoked routine is itself a state-table-driven routine that uses a different table.

3.6.2. Input Encoding and Input Alphabet

Only the simplest finite-state machines, such as a character sequence detector in a compiler's lexical analyzer, can use the inputs directly. Typically, we're not interested in the actual input characters but in some attribute represented by the characters. For example, in the ZCZC detector, although there are 256 possible ASCII characters (including the inverse parity characters), we're only interested in three different types: "Z," "C," and "OTHER." The input encoding could be implemented as a table lookup in a table that contained the following codes: "OTHER" = 0, "Z" = 1 and "C" = 2. Alternatively, we could implement it as a process: IF INPUT = "Z" THEN CODE := 1 ELSE IF INPUT = "C" THEN CODE := 2 ELSE CODE := 0 ENDIF.

The alternative to input encoding is a huge state graph and table because there must be one outlink in every state for every possible different input. Input encoding compresses the cases and therefore the state graph. Another advantage of input encoding is that we can run the machine from a mixture of otherwise incompatible input events, such as characters, device response codes, thermostat settings, or gearshift lever positions. The set of different encoded input values is called the **input alphabet**. The word "input" as used in the context of finite-state machines always means a "character" from the input alphabet.

3.6.3. Output Encoding and Output Alphabet

There can be many different, incompatible, kinds of outputs for transitions of a finite-state machine: a single character output for a link is rare in actual applications. We might want to output a string of characters, call a subroutine, transfer control to a lower-level finite-state machine, or do nothing. Whatever we might want to do, there are only a finite number of such distinct actions, which we can encode into a convenient **output alphabet**. We then have a hypothetical (or real) output processor that invokes the action appropriate to the output code. Doing nothing is also considered an action and therefore requires its own code in the output alphabet. The word “output” as used in the context of finite-state machines means a “character” from the output alphabet.

3.6.4. State Codes and State-Symbol Products

We speak about finite-state machines as if the states are numbered by an integer. If there are n states and k different inputs, both numbered from zero, and the state code and input code are S and I respectively, then the pointer value is $Sk + I$ or $In + S$ depending on how you want to organize the tables. If the state machine processor is coded in an HOL then you can use a two-dimensional array and use two pointers (state code and input code); the multiplication will be done by object code. Finite-state machines are often used in time-critical applications because they have such fast response times. If a multiplication has to be done, the speed is seriously affected. A faster implementation is to use a binary number of states and a binary number of input codes, and to form the pointer by concatenating the state and input code. The speed advantage is obvious, but there are also some disadvantages. The table is no longer compact; that is, because the number of states and the number of input codes are unlikely to be both binary numbers, the resulting table must have holes in it. Like it or not, those holes correspond to state-input combinations and you have to fill them, if only with a call to an error recovery routine. The second disadvantage is size. Even in these days of cheap memory, excessive table size can be a problem, especially, for example, if the finite-state machine is part of embedded software in a ROM. For the above reasons, there may be another encoding of the combination of the state number and the input code into the pointer. The term **state-symbol product** is used to mean the value obtained by any scheme used to convert the combined state and input code into a pointer to a compact table without holes. This conversion could be done by multiplication and addition, by concatenation, or even by a hash-coding scheme for very big tables. When we talk about “states” and “state codes” in the context of finite-state machines, we mean the (possibly) hypothetical integer used to denote the state and not the actual form of the state code that could result from an encoding process. Similarly, “state-symbol product” means the hypothetical (or actual) concatenation used to combine the state and input codes.

3.6.5. Application Comments for Designers

An explicit state-table implementation is advantageous when either the control function is likely to change in the future or when the system has many similar, but slightly different, control functions. Their use in telecommunications, especially telephony, is common. This technique can provide fast response time—one pass through the above program can be done in ten to fifteen machine instruction execution times. It is not an effective technique for very small (four states or less) or big (256 states or more) state graphs. In the small case, the overhead required to implement the state-table software would exceed any time or space savings that one might hope to gain. In big state tables, the product of input values and states is big—in the thousands—and the memory required to store the tables becomes significant. The usual approach for big state graphs is to partition the problem into a hierarchy of finite-state machines. The output of the top level machine is a call to a subsidiary machine that processes the details. In telephony, for example, two-level tables are common and three- and four-level tables are not unusual.

3.6.6. Application Comments for Testers

Independent testers are not usually concerned with either implementation details or the economics of this approach but with how a state-table or state-graph representation of the behavior of a program or system can help us to design effective tests. If the programmers have implemented an explicit finite-state machine then much of our work has been done for us and we have to be concerned with the kinds of bugs that are inherent in the implementation—which is good reason for understanding such implementations. There is an interesting correlation, though: when a

finite-state machine *model* is appropriate, so is a finite-state machine *implementation*. Sometimes, showing the programmers the kinds of tests developed from a state-graph description can lead them to consider it as an implementation technique.

4. GOOD STATE GRAPHS AND BAD

4.1. General

This is a book on testing so we deal not just with good state graphs, but also with bad ones. What constitutes a good or a bad state graph is to some extent biased by the kinds of state graphs that are likely to be used in a software test design context. Here are some principles for judging:

1. The total number of states is equal to the product of the possibilities of factors that make up the state.
2. For every state and input there is exactly one transition specified to exactly one, possibly the same, state.
3. For every transition there is one output action specified. That output could be trivial, but at least one output does something sensible.*
- *State graphs without outputs can't do anything in the pragmatic world and can consequently be ignored. For output, include anything that could cause a subsequent action—perhaps setting only one bit.
4. For every state there is a sequence of inputs that will drive the system back to the same state.**
- **In other words, we've restricted the state graphs to be strongly connected. This may seem overly narrow, because many state graphs are not strongly connected; but in a software context, the only nonstrongly connected state graphs are those used to set off bombs and other infernal machines or those that deal with bootstraps, initialization, loading, failure, recovery, and illogical, unrecoverable conditions. A state graph that is not strongly connected usually has bugs.

Figure 11.3 shows examples of improper state graphs.

A state graph must have at least two different input codes. With only one input code, there are only a few kinds of state graphs you can build: a bunch of disconnected individual states; disconnected strings of states that end in loops and variations thereof; or a strongly connected state graph in which all states are arranged in one grand loop. The latter can be implemented by a simple counter that resets at some fixed maximum value, so this elaborate modeling apparatus is not needed.

If I seem to have violated my own rules regarding outputs—I have. The ZCZC detector example didn't have output codes. There are two aspects of state graphs: (1) the states with their transitions and the inputs that cause them, and (2) the outputs associated with transitions. Just as in the flowgraph model we concentrated on control structure and tended to ignore processing that did not directly affect control flow, in state testing we may ignore outputs because it is the states and transitions that are of primary interest. Two state graphs with identical states, inputs, and transitions could have vastly different outputs, yet from a state-testing point of view, they could be identical. Consequently, we reduce the clutter caused by explicit output specifications if outputs are not interesting at the moment.

4.2. State Bugs

4.2.1. Number of States

The number of states in a state graph is the number of states we choose to recognize or model. In practice, the state is directly or indirectly recorded as a combination of values of variables that appear in the data base. As an example, the state could be composed of the value of a counter whose possible values ranged from 0 to 9, combined with the setting of two bit flags, leading to a total of $2 \times 2 \times 10 = 40$ states. When the state graph represents an explicit state-table implementation, this value is encoded so bugs in the number of states are less likely; but the encoding can be wrong. Failing to account for all the states is one of the more common bugs in

software that can be modeled by state graphs. Because an explicit state-table mechanization is not typical, the opportunities for missing states abound. Find the number of states as follows:

1. Identify all the component factors of the state.
2. Identify all the allowable values for each factor.
3. The number of states is the product of the number of allowable values of all the factors.

Before you do anything else, before you consider one test case, discuss the number of states you think there are with the number of states the programmer (or you, if you're wearing a programmer's hat) thinks there are. Differences of opinion are common. There's no point in designing tests intended to check the system's behavior in various states if there's no agreement on how many states there are. And if there's no agreement on how many states there are, there must be disagreement on what the system does in which states and on the transitions and the outputs. If it seems that I'm giving undue emphasis to the seemingly trivial act of counting states, it's because that act often exhumes fundamental design deficiencies. You don't need to wait until the design is done. A functional specification is usually enough, inasmuch as state testing is primarily a functional test tool. I read the functional specification and identify the factors and then the number of possible values for each factor. Then I question the designer. I want to get an identification or recognition for each state—with one state corresponding to each combination of condition values. It's gratifying work. It's gratifying to hear, "Oh yeah, I forgot about that one." Make up a table, with a column for every factor, such that all combinations of factors are represented. Before you get concurrence on outputs or transitions or the inputs that cause the transitions, get concurrence from the designer (or confirm for yourself) that every combination listed makes sense.

4.2.2. Impossible States

Some combinations of factors may appear to be impossible. Say that the factors are:

GEAR	R, N, 1, 2, 3, 4	= 6 factors
DIRECTION	Forward, reverse, stopped	= 3 factors
ENGINE	Running, stopped	= 2 factors
TRANSMISSION	Okay, broken	= 2 factors
ENGINE	Okay, broken	= 2 factors
TOTAL		= 144 states

But broken engines can't run, so the combination of factors for engine condition and engine operation yields only 3 rather than 4 states. Therefore, the total number of states is at most 108. A car with a broken transmission won't move for long, thereby further decreasing the number of feasible states. The discrepancy between the programmer's state count and the tester's state count is often due to a difference of opinion concerning "impossible states."

4.2.3. Equivalent States

Two states are **equivalent** if every sequence of inputs starting from one state produces exactly the same sequence of outputs when started from the other state. This notion can also be extended to sets of states. [Figure 11.4](#) shows the situation.

Say that the system is in state S and that an input of *a* causes a transition to state A while an input of *b* causes a transition to state B. The blobs indicate portions of the state graph whose details are unimportant. If, starting from state A, *every* possible sequence of inputs produces *exactly* the same sequence of outputs that would occur when starting from state B, then there is no way that an outside observer can determine which of the two sets of states the system is in without looking at the record of the state. The state graph can be reduced to that of [Figure 11.5](#) without harm.

The fact that there is a notion of state equivalency means that there is an opportunity for bugs to arise from a difference of opinion concerning which states are equivalent. If you insist that there is another factor, not recognized by the programmer, such that the resulting output sequence for a given input sequence is different depending on the value of that factor, then you are asserting that two inequivalent sets of states have been inadvertently merged. Conversely, if you cannot find a sequence of inputs that results in at least one different output when starting from either of two supposedly inequivalent states, then the states *are* equivalent and should be merged if only to simplify the software and thereby reduce the probability of bugs. Be careful, though, because equivalent states could come about as a result of good planning for future enhancements. The two states are presently indistinguishable but could in the future become distinguished as a result of an enhancement that brings with it the distinguishing factor.

Equivalent states can be recognized by the following procedures:

1. The rows corresponding to the two states are identical with respect to input/output/next state but the name of the next state could differ. The two states are differentiated only by the input that distinguishes between them. This situation is shown in [Figure 11.6](#). Except for the *a, b* inputs, which distinguish between states A and B, the system's behavior in the two states is identical for every input sequence; they can be merged.
2. There are two sets of rows which, except for the state names, have identical state graphs with respect to transitions and outputs. The two sets can be merged (see [Figure 11.7](#)).

The rows are not identical, but except for the state names ($A1 = B2$, $A2 = B2$, $A3 = B3$), the system's action, when judged by the relation between the output sequence produced by a given input sequence, is identical for either the A or the B set of states. Consequently, this state graph can be replaced by the simpler version shown in [Figure 11.7c](#).

4.3. Transition Bugs

4.3.1. Unspecified and Contradictory Transitions

Every input-state combination must have a specified transition. If the transition is impossible, then there must be a mechanism that prevents that input from occurring in that state—look for it. If there is no such mechanism, what will the program do if, through a malfunction or an alpha particle, the impossible input occurs in that state? The transition for a given state-input combination may not be specified because of an oversight. *Exactly one transition must be specified for every combination of input and state.* However you model it or test it, the system will do *something* for every combination of input and state. It's better that it does what you want it to do, which you assure by specifying a transition rather than what some bugs want it to do.

A program can't have contradictions or ambiguities. Ambiguities are impossible because the program will do *something* (right or wrong) for every input. Even if the state does not change, by definition this is a transition to the same state. Similarly, software can't have contradictory transitions because computers can only do one thing at a time. A seeming contradiction could come about in a model if you don't account for *all* the factors that constitute the state and all the inputs. A single bit may have escaped your notice; if that bit is part of the definition of the state it can double the number of states, but if you're not monitoring that factor of the state, it would appear that the program had performed contradictory transitions or had different outputs for what appeared to be the same input from the same state. If you, as a designer, say while debugging "sometimes it works and sometimes it doesn't," you've admitted to a state factor of which you're not aware—a factor probably caused by a bug. Exploring the real state graph and recording the transitions and outputs for each combination of input and state may lead you to discover the bug.

4.3.2. An Example

Specifications are one of the most common source of ambiguities and contradictions. Specifications, unlike programs, can be full of ambiguities and contradictions. The following example illustrates how to convert a

specification into a state graph and how contradictions can come about. The tape control routine will be used. Start with the first statement in the specification and add to the state graph one statement at a time. Here is the first statement of the specification:

Rule 1: The program will maintain an error counter, which will be incremented whenever there's an error.

There are only two input values, "okay" and "error." A state table will be easier to work with, and it's much easier to spot ambiguities and contradictions. Here's the first state table:

INPUT

STATE	OKAY	ERROR
0	0/none	1/
1		2/
2		3/
3		4/
4		5/
5		6/
6		7/
7		8/

There are no contradictions yet, but lots of ambiguities. It's easy to see how ambiguities come about—just stop the specification before it's finished. Let's add the rules one at a time and fill in the state graph as we go. Here are the rest of the rules; study them to see if you can find the problems, if any:

Rule 2: If there is an error, rewrite the block.

Rule 3: If there have been three successive errors, erase 10 centimeters of tape and then rewrite the block.

Rule 4: If there have been three successive erasures and another error occurs, put the unit out of service.

Rule 5: If the erasure was successful, return to the normal state and clear the error counter.

Rule 6: If the rewrite was unsuccessful, increment the error counter, advance the state, and try another rewrite.

Rule 7: If the rewrite was successful, decrement the error counter and return to the previous state.

Adding rule 2, we get

INPUT

STATE	OKAY	ERROR
0	0/NONE	1/REWRITE
1		2/REWRITE
2		3/REWRITE
3		4/REWRITE
4		5/REWRITE
5		6/REWRITE
6		7/REWRITE
7		8/REWRITE

Rule 3: If there have been three successive errors, erase 10 centimeters of tape and then rewrite the block.

INPUT

STATE	OKAY	ERROR
0	0/NONE	1/REWRITE
1		2/REWRITE
2		3/REWRITE, ERASE, REWRITE
3		4/REWRITE, ERASE, REWRITE
4		5/REWRITE, ERASE, REWRITE
5		6/REWRITE, ERASE, REWRITE
6		7/REWRITE, ERASE, REWRITE
7		8/REWRITE, ERASE, REWRITE

Rule 3, if followed blindly, causes an unnecessary rewrite. It's a minor bug, so let it go for now, but it pays to check such things. There might be an arcane security reason for rewriting, erasing, and then rewriting again.

Rule 4: If there have been three successive erasures and another error occurs, put the unit out of service.

INPUT

STATE	OKAY	ERROR
0	0/NONE	1/RW
1		2/RW
2		3/ER, RW
3		4/ER, RW
4		5/ER, RW
5		6/OUT
6		
7		

Rule 4 terminates our interest in this state graph so we can dispose of states beyond 6. The details of state 6 will not be covered by this specification; presumably there is a way to get back to state 0. Also, we can credit the specifier with enough intelligence not to have expected a useless rewrite and erase prior to going out of service.

Rule 5: If the erasure was successful, return to the normal state and clear the counter.

INPUT

STATE	OKAY	ERROR
0	0/NONE	1/RW
1		2/RW
2		3/ER, RW
3	0/NONE	4/ER, RW
4	0/NONE	5/ER, RW
5	0/NONE	6/OUT
6		

Rule 6: If the rewrite was unsuccessful, increment the error counter, advance the state, and try another rewrite.

Because the value of the error counter is the state, and because rules 1 and 2 specified the same action, there seems to be no point to rule 6 unless yet another rewrite was wanted. Furthermore, the order of the actions is wrong. If the state is advanced before the rewrite, we could end up in the wrong state. The proper order should have been: output = attempt-rewrite and then increment the error counter.

Rule 7: If the rewrite was successful, decrement the error counter and return to the previous state.

INPUT

STATE	OKAY	ERROR
0	0/NONE	1/RW
1	0/NONE	2/RW
2	1/NONE	3/ER, RW
3	0/NONE 2/NONE	4/ER, RW
4	0/NONE 3/NONE	5/ER, RW
5	0/NONE 4/NONE	6/OUT
6		

Rule 7 got rid of the ambiguities but created contradictions. The specifier's intention was probably:

Rule 7A: If there have been no erasures and the rewrite is successful, return to the previous state.

4.3.3. Unreachable States

An **unreachable state** is like unreachable code—a state that no input sequence can reach. An unreachable state is not impossible, just as unreachable code is not impossible. Furthermore, there may be transitions from the unreachable state to other states; there usually are because the state became unreachable as a result of incorrect transitions.

Unreachable states can come about from previously “impossible” states. You listed all the factors and laid out a state table. Some of these states corresponded to previously “impossible” states. The designer, perhaps after some rough persuasion, agrees that something should be done about the unreachable states. “Easy,” he thinks, “provide no transitions into them.” Yet there should still be a transition *out* of all such states. At least there should be a transition to an error-recovery procedure or an exception handler.

An isolated, unreachable state here and there, which clearly relates to impossible combinations of real-world state-determining conditions, is acceptable, but if you find groups of connected states that are isolated from others, there's cause for concern. There are two possibilities: (1) There is a bug; that is, some transitions are missing. (2) The transitions are there, but you don't know about it; in other words, there are other inputs and associated transitions to reckon with. Typically, such hidden transitions are caused by software operating at a higher priority level or by interrupt processing.

4.3.4. Dead States

A **dead state**, (or set of dead states) is a state that once entered cannot be left. This is not necessarily a bug, but it is suspicious. If the software was designed to be the fuse for a bomb, we would expect at least one such state. A set of

states may appear to be dead because the program has two modes of operation. In the first mode it goes through an initialization process that consists of several states. Once initialized, it goes to a strongly connected set of working states, which, within the context of the routine, cannot be exited. The initialization states are unreachable to the working states, and the working states are dead to the initialization states. The only way to get back might be after a system crash and restart. Legitimate dead states are rare. They occur mainly with system-level issues and device handlers. In normal software, if it's not possible to get from any state to any other, there's reason for concern.

4.4. Output Errors

The states, the transitions, and the inputs could be correct, there could be no dead or unreachable states, but the output for the transition could be incorrect. Output actions must be verified independently of states and transitions. That is, you should distinguish between a program whose state graph is correct but has the wrong output for a transition and one whose state graph is incorrect. The likeliest reason for an incorrect output is an incorrect call to the routine that executes the output. This is usually a localized and minor bug. Bugs in the state graph are more serious because they tend to be related to fundamental control-structure problems. If the routine is implemented as a state table, both types of bugs are comparably severe.

4.5. Encoding Bugs

It would seem that encoding bugs for input coding, output coding, state codes, and state-symbol product formation could exist as such only in an explicit finite-state machine implementation. The possibility of such bugs is obvious for a finite-state machine implementation, but the bugs can also occur when the finite-state machine is implicit. If the programmer has a notion of state and has built an implicit finite-state machine, say by using a bunch of program flags, switches, and "condition" or "status" words, there may be an encoding process in place.

Make it a point *not* to use the programmer's state numbers and/or input codes. As a tester, you're dealing with an abstract machine that you're going to use to develop tests. *The behavior of a finite-state machine is invariant under all encodings.* That is, say that the states are numbered 1 to n . If you renumber the states by an arbitrary permutation, the finite-state machine is unchanged—similarly for input and output codes. Therefore, if you present your version of the finite-state machine with a different encoding, and if the programmer objects to the renaming or claims that behavior is changed as a result, then use that as a signal to look for encoding bugs. You may have to look at the implementation for these, especially the data dictionary. Look for "status" codes and read the list carefully. The key words are "unassigned," "reserved," "impossible," "error," or just gaps.

The implementation of the fields as a bunch of bits or bytes tells you the potential size of the code. If the number of code values is less than this potential, there is an encoding process going on, even if it's only to catch values that are out of range. In strongly typed languages with user-defined semantic types, the encoding process is probably a type conversion from a set membership, say, to a pointer type or integer. Again, you may have to look at the program to spot potential bugs of this kind.

5. STATE TESTING

5.1. Impact of Bugs

Let's say that a routine is specified as a state graph that has been verified as correct in all details. Program code or tables or a combination of both must still be implemented. A bug can manifest itself as one or more of the following symptoms:

1. Wrong number of states.
2. Wrong transition for a given state-input combination.
3. Wrong output for a given transition.
4. Pairs of states or sets of states that are inadvertently made equivalent (factor lost).
5. States or sets of states that are split to create inequivalent duplicates.

6. States or sets of states that have become dead.
7. States or sets of states that have become unreachable.

5.2. Principles

The strategy for state testing is analogous to that used for path-testing flowgraphs. Just as it's impractical to go through every possible path in a flowgraph, it's impractical to go through every path in a state graph. A path in a state graph, of course, is a succession of transitions caused by a sequence of inputs. The notion of coverage is identical to that used for flowgraphs—pass through each link (i.e., each transition must be exercised). Assume that some state is especially interesting—call it the initial state. Because most realistic state graphs are strongly connected, it should be possible to go through all states and back to the initial state, when starting from there. But don't do it. Even though most state testing can be done as a single case in a grand tour, it's impractical to do it that way for several reasons:

1. In the early phases of testing, you'll never complete the grand tour because of bugs.
2. Later, in maintenance, testing objectives are understood, and only a few of the states and transitions have to be retested. A grand tour is a waste of time.
3. There's so much history in a long test sequence and so much has happened that verification is difficult.

The starting point of state testing is:

1. Define a set of covering input sequences that get back to the initial state when starting from the initial state.
2. For each step in each input sequence, define the expected next state, the expected transition, and the expected output code.

A set of tests, then, consists of three sets of sequences:

1. Input sequences.
2. Corresponding transitions or next-state names.
3. Output sequences.

5.3. Limitations and Extensions

Just as link coverage in a flowgraph model of program behavior did not guarantee "complete testing," state-transition coverage in a state-graph model does not guarantee complete testing. Things are slightly better because it's not necessary to consider any sequence longer than the total number of states. *Note:* Everything discussed in this section applies equally well to control flowgraphs with suitable translation.

Chow (CHOW78) defines a hierarchy of paths and methods for combining paths to produce covers of a state graph. The simplest is called a "0 switch," which corresponds to testing each transition individually. The next level consists of testing transition sequences consisting of two transitions, called "1 switches." The maximum-length switch is an $n - 1$ switch, where n is the number of states. Chow's primary result shows that in general, a 0 switch cover (which we recognize as branch cover for control flowgraphs) can catch output errors but may not catch some transition errors. In general, one must use longer and longer covering sequences to catch transition errors, missing states, extra states, and the like. The theory of what constitutes a sufficient number of tests (i.e., input sequences) to catch specified kinds of state-graph errors is still in its infancy and is beyond the scope of this book. Furthermore, practical experience with the application of such theory to software as exists is limited, and the efficacy of such methods as bug catchers has yet to be demonstrated sufficiently well to earn these methods a solid place in the software tester's tool repertoire. Work continues and progress in the form of semiautomatic test tools and effective methods are sure to come. Meanwhile, we have the following experience:

1. Simply identifying the factors that contribute to the state, calculating the total number of states, and comparing this number to the designer's notion catches some bugs.

2. Insisting on a justification for all supposedly dead, unreachable, and impossible states and transitions catches a few more bugs.
3. Insisting on an explicit specification of the transition and output for every combination of input and state catches many more bugs.
4. A set of input sequences that provide coverage of all nodes and links is a mandatory minimum requirement.
5. In executing state tests, it is essential that means be provided (e.g., instrumentation software) to record the sequence of states (e.g., transitions) resulting from the input sequence and not just the outputs that result from the input sequence.

5.4. What to Model

Because every combination of hardware and software can in principle be modeled by a sufficiently complicated state graph, this representation of software behavior is applicable to every program. The utility of such tests, however, is more limited. The state graph is a behavioral model—it is functional rather than structural and is thereby far removed from the code. As a testing method, it is a bottom-line method that ignores structural detail to focus on behavior. It is advantageous to look into the database to see how the factors that create the states are represented in order to get a state count. More than most test methods, state testing yield the biggest payoffs during the design of the tests rather than during the running thereof. Because the tests can be constructed from a design specification long before coding, they help catch deep bugs early in the game when correction is inexpensive. Here are some situations in which state testing may prove useful:

1. Any processing where the output is based on the occurrence of one or more sequences of events, such as detection of specified input sequences, sequential format validation, parsing, and other situations in which the order of inputs is important.
2. Most protocols between systems, between humans and machines, between components of a system (CHOI84, CHUN78, SARI88).
3. Device drivers such as for tapes and discs that have complicated retry and recovery procedures if the action depends on the state.
4. Transaction flows where the transactions are such that they can stay in the system indefinitely—for example, online users, tasks in a multitasking system.
5. High-level control functions within an operating system. Transitions between user states, supervisor's states, and so on. Security handling of records, permission for read/write/modify privileges, priority interrupts and transitions between interrupt states and levels, recovery issues and the safety state of records and/or processes with respect to recording recovery data.
6. The behavior of the system with respect to resource management and what it will do when various levels of resource utilization are reached. Any control function that involves responses to thresholds where the system's action depends not just on the threshold value, but also on the direction in which the threshold is crossed. This is a normal approach to control functions. A threshold passage in one direction stimulates a recovery function, but that recovery function is not suspended until a second, lower threshold is passed going the other way.
7. A set of menus and ways that one can go from one to the other. The currently active menus are the states, the input alphabet is the choices one can make, and the transitions are invocations of the next menu in a menu tree. Many menu-driven software packages suffer from dead states—menus from which the only way out is to reboot.
8. Whenever a feature is directly and explicitly implemented as one or more state-transition tables.

5.5. Getting the Data

As is so often the case in the independent tester's life, getting the data on which the model is to be based is half the job or more. There's no magic for doing that: reading documents, interviews, and all the rest. State testing, more than most functional test strategies, tends to have a labor-intensive data-gathering phase and tends to need many more meetings to resolve issues. This is the case because most of the participants don't realize that there's an essential state-machine behavior. For nonprogrammers, especially, the very concept of finite-state machine

behavior may be missing. Be prepared to spend more time on getting data than you think is reasonable and be prepared to do a lot of educating along the way.

5.6. Tools

Good news and bad news: The telecommunications industry, especially in telephony, has been using finite-state machine implementations of control functions for decades (BAUE79). They also use several languages/systems to code state tables directly. Similarly, there are tools to do the same for hardware logic designs. That's the good news. The bad news is that these systems and languages are proprietary, of the home-brew variety, internal, and/or not applicable to the general use of software implementations of finite-state machines. The most successful tools are not published and are unlikely to be published because of the competitive advantage they give to the users of those tools.

6. TESTABILITY TIPS

6.1. A Balm for Programmers

Most of this chapter has taken the independent tester's viewpoint and has been a prescription for making programmers squirm. What is testability but means by which programmers can protect themselves from the ravages of sinister independent testers? What is testability but a guide to cheating—how to design software so that the pesticide paradox works and the tester's strongest technique is made ineffectual? The key to testability design is easy: build explicit finite-state machines.

6.2. How Big, How Small?

I understand every two-state finite-state machine because, including the good and bad ones, there are only eight of them. There are about eighty possible good and bad three-state machines, 2700 four-state machines, 275,000 five-state machines, and close to 100 million six-state machines, most of which are bad. We learned long ago, as hardware logic designers, that it paid to build explicit finite-state machines for even very small machines. I think you can safely get away with two states, it's getting difficult for three states, a heroic act for four, and beyond human comprehension for five states. That doesn't mean that you have to build your finite-state machine as in the explicit PDL example given above, but that you must do a finite-state machine model and identify how you're implementing every part of that model for anything with four or more states.

6.3. Switches, Flags, and Unachievable Paths

Something may look like a finite-state machine but not be one. [Figure 11.9a](#) shows a program with a switch or flag. Someplace early in the routine we set a flag, A, then later we test the flag and go one way or the other depending on its value. In [Figure 11.9b](#) we've rewritten the routine to eliminate the flag. As soon as the flag value is calculated, we branch. The cost is the cost of converting segment V into a subroutine and calling it twice. But note that we went from four paths, two of which are unachievable to two paths, both of which are achievable and both of which are needed to achieve branch coverage.

In [Figure 11.10](#), the situation is more complicated. There are three switches this time. Again, where we go depends on the switch settings calculated earlier in the program. We can put the decision up front and branch directly, and again use subroutines to make each path explicit and do without the switches. The advantages of this implementation is that if any of the combinations are not needed, we merely clip out that part of the decision tree, as in [Figure 11.10c](#). Again, all paths are achievable and all paths are needed for branch cover.

[Figure 11.11](#) is similar to the previous two except that we've put the switched parts in a loop. It's even worse if the loop includes the switch value calculations (dotted link). We now have a very difficult situation. We don't know which of these paths are achievable and which are or are not required. What is or is not achievable depends on the switch settings. Branch coverage won't do it: we must do or attempt branch coverage in every possible state.

6.4. Essential and Inessential Finite-State Behavior

Program flags and switches are predicates deferred. There is a significant, qualitative difference between finite-state machines and combinational machines. A combinational machine selects paths based on the values of predicates, the predicates depend only on prior processing and the predicates' truth values will not change once they have been determined. Any path corresponds to a boolean algebra expression over the predicates. Furthermore, it does not matter in which order the decisions are made. The fact that there is an ordering is a consequence of a sequential, Von Neumann computer architecture. In a parallel-data-flow machine, for example, the decisions and path selections could be made simultaneously. Sequence and finite-state behavior are in this case implementation consequences and not essential. The combinational machine has exactly one state and one transition back to itself for all possible inputs. The control logic of a combinational program can be described by a decision table or a decision tree.

The simplest essential finite-state machine is a flip-flop. There is no logic that can implement it without some kind of feedback. You cannot describe this behavior by a decision table or decision tree unless you provide feedback into the table or call it recursively. It must have a loop or the equivalent.

The problem with nontrivial finite-state machine behavior is that to do the equivalent of branch testing, say, you must do it over for every state. Why take on that extra burden if the finite-state machine behavior isn't essential?

Most programmers' implementation of finite-state behavior is not essential—it appears to be convenient. Most programmers, having implemented finite-state behavior, will not test it properly. I've yet to see a programmer who implemented a routine with 10 flags rerun the tests for all 1024 possible flag settings.

Learn to distinguish between essential and inessential finite-state behavior. It's not essential if you can do it by a parallel program in a hypothetical data-flow machine. It's not essential if a decision-table model will do it for you or if you can program it as a big decision tree. It's not essential if the program's exit expression ([Chapter 10](#)), even with explicit loops, equals unity. It's not essential if there's a nonunity exit expression but it turns out that you don't really want to loop under the looping conditions. I'm not telling you to throw away your "harmless" little flags and switches and not to implement inessential finite-state machine behavior. All I ask is that you be prepared to repeat your tests in every state.

6.5. Design Guidelines

I'll assume that you've checked the design and the specification and that you're not about to implement inessential finite-state machine behavior. What should you do if you must build finite-state machines into your code?

1. Learn how it's done in hardware. I know of no books on finite-state machine design for programmers. There are only books on hardware logic design and switching theory, with a distinct hardware flavor and you'll have to adapt their methods to software.
2. Start by designing the abstract machine. Verify that it is what you want to do. Do an explicit analysis, in the form of a state graph or table, for anything with three states or more.
3. Start with an explicit design—that is, input encoding, output encoding, state code assignment, transition table, output table, state storage, and how you intend to form the state-symbol product. Do this at the PDL level. But be sure to document that explicit design.
4. Before you start taking shortcuts, see if it really matters. Neither the time nor the memory for the explicit implementation usually matters. Do a prototype based on the explicit design and analyze that or measure it to see what the processing time actually is and if that's significant. Remember that explicit finite-state machines are usually very fast and that the penalty is likelier to be a memory cost than a time cost. Test the prototype thoroughly, as discussed above. The prototype test suite should be kept for later use.
5. Take shortcuts by making things implicit only as you must to make significant reductions in time or space and only if you can show that such savings matter in the context of the whole system. After all, doubling the speed of your implementation may mean nothing if all you've done is shaved 100

microseconds from a 500-millisecond process. The order in which you should make things implicit are: output encoding, input encoding, state code, state-symbol product, output table, transition table, state storage. That's the order from least to most dangerous.

6. Consider a hierarchical design if you have more than a few dozen states.
7. Build, buy, or implement tools and languages that implement finite-state machines as software if you're doing more than a dozen states routinely.
8. Build in the means to initialize to any arbitrary state. Build in the transition verification instrumentation (the coverage analyzer). These are much easier to do with an explicit machine.

7. SUMMARY

1. State testing is primarily a functional testing tool whose payoff is best in the early phases of design.
2. A program can't have contradictory or ambiguous transitions or outputs, but a specification can and does. Use a state table to verify the specification's validity.
3. Count the states.
4. Insist on a specification of transition and output for every combination of input and states.
5. Apply a minimum set of covering tests.
6. Instrument the transitions to capture the sequence of states and not just the sequence of outputs.
7. Count the states.

UNIT VIII :

Graph Matrices and Application : Motivational overview, matrix of graph, relations, power of a matrix, node reduction algorithm, building tools.

GRAPH MATRICES AND APPLICATIONS

1. SYNOPSIS

Graph matrices are introduced as another representation for graphs; some useful tools resulting therefrom are examined. Matrix operations, relations, node-reduction algorithm revisited, equivalence class partitions.

2. MOTIVATIONAL OVERVIEW

2.1. The Problem with Pictorial Graphs

Graphs were introduced as an abstraction of software structure early in this book and used throughout. Yet another graph that modeled software behavior was introduced in Chapter 11. There are many other kinds of graphs, not discussed in this book, that are useful in software testing. Whenever a graph is used as a model, sooner or later we trace paths through it—to find a set of covering paths, a set of values that will sensitize paths, the logic function that controls the flow, the processing time of the routine, the equations that define a domain, whether the routine pushes or pops, or whether a state is reachable or not. Even algebraic representations such as BNF and regular expressions can be converted to equivalent graphs. Much of test design consists of tracing paths through a graph and most testing strategies define some kind of cover over some kind of graph.

Path tracing is not easy, and it's subject to error. You can miss a link here and there or cover some links twice—even if you do use a marking pen to note which paths have been taken. You're tracing a long complicated path through a routine when the telephone rings—you've lost your place before you've had a chance to mark it. I get confused tracing paths, so naturally I assume that other people also get confused.

One solution to this problem is to represent the graph as a matrix and to use matrix operations equivalent to path tracing. These methods aren't necessarily easier than path tracing, but because they're more methodical and mechanical and don't depend on your ability to "see" a path, they're more reliable.

Even if you use powerful tools that do everything that can be done with graphs, and furthermore, enable you to do it graphically, it's still a good idea to know how to do it by hand; just as having a calculator should not mean that you don't need to know how to do arithmetic. Besides, with a little practice, you might find these methods easier and faster than doing it on the screen; moreover, you can use them on the plane or anywhere.

2.2. Tool Building

If you build test tools or want to know how they work, sooner or later you'll be implementing or investigating analysis routines based on these methods—or you should be. Think about how a naive tool builder would go about finding a property of all paths (a possibly infinite number) versus how one might do it based on the methods of [Chapter 8](#). But [Chapter 8](#) was graphical and it's hard to build algorithms over visual graphs. The properties of graph matrices are fundamental to test tool building.

2.3. Doing and Understanding Testing Theory

We talk about graphs in testing theory, but we prove theorems about graphs by proving theorems about their matrix representations. Without the conceptual apparatus of graph matrices, you'll be blind to much of testing theory, especially those parts that lead to useful algorithms.

2.4. The Basic Algorithms

This is not intended to be a survey of graph-theoretic algorithms based on the matrix representation of graphs. It's intended only to be a basic toolkit. For more on this subject, see EVEN79, MAYE72, PHIL81. The basic toolkit consists of:

1. Matrix multiplication, which is used to get the path expression from every node to every other node.
2. A partitioning algorithm for converting graphs with loops into loop-free graphs of equivalence classes.
3. A collapsing process (analogous to the determinant of a matrix), which gets the path expression from any node to any other node.

3. THE MATRIX OF A GRAPH

3.1. Basic Principles

A **graph matrix** is a square array with one row and one column for every node in the graph. Each row-column combination corresponds to a relation between the node corresponding to the row and the node corresponding to the column. The relation, for example, could be as simple as the link name, if there is a link between the nodes. Some examples of graphs and their associated matrices are shown in [Figure 12.1](#) a through g. Observe the following:

1. The size of the matrix (i.e., the number of rows and columns) equals the number of nodes.
2. There is a place to put every possible direct connection or link between any node and any other node.
3. The entry at a row and column intersection is the link weight of the link (if any) that connects the two nodes in that direction.
4. A connection from node i to node j does not imply a connection from node j to node i . Note that in [Figure 12.1h](#) the (5,6) entry is m , but the (6,5) entry is c .
5. If there are several links between two nodes, then the entry is a sum; the “+” sign denotes parallel links as usual.

In general, an entry is not just a simple link name but a path expression corresponding to the paths between the pair of nodes. Furthermore, as with the graphs, an entry can be a link weight or an expression in link weights (see [Chapter 8](#) for a refresher). Finally, “arithmetic operations” are the operations appropriate to the weights the links represent.

3.2. A Simple Weight

The simplest weight we can use is to note that there is or isn't a connection. Let "1" mean that there is a connection and "0" that there isn't. The arithmetic rules are:

$$\begin{array}{lll} 1 + 1 = 1, & 1 + 0 = 1, & 0 + 0 = 0, \\ 1 \times 1 = 1, & 1 \times 0 = 0, & 0 \times 0 = 0. \end{array}$$

A matrix with weights defined like this is called a **connection matrix**. The connection matrix for Figure 12.1h is obtained by replacing each entry with 1 if there is a link and 0 if there isn't. As usual, to reduce clutter we don't write down 0 entries. Each row of a matrix (whatever the weights) denotes the outlinks of the node corresponding to that row, and each column denotes the inlinks corresponding to that node. A branch node is a node with more than one nonzero entry in its row. A junction node is a node with more than one nonzero entry in its column. A self-loop is an entry along the diagonal. Because rows 1, 3, 6, and 8 of Figure 12.1h all have more than one entry, those nodes are branch nodes. Using the principle that a case statement is equivalent to $n - 1$ binary decisions, by subtracting 1 from the total number of entries in each row and ignoring rows with no entries (such as node 2), we obtain the equivalent number of decisions for each row. Adding these values and then adding 1 to the sum yields the graph's cyclomatic complexity.

3.3. Further Notation

Talking about the "entry at row 6, column 7" is wordy. To compact things, the entry corresponding to node i and column j , which is to say the link weights between nodes i and j , is denoted by a_{ij} . A self-loop about node i is denoted by a_{ii} , while the link weight for the link between nodes j and i is denoted by a_{ji} . The path segments expressed in terms of link names and, in this notation, for several paths in the graph of Figure 12.1h are:

$$\begin{aligned} abmd &= a_{13}a_{35}a_{56}a_{67}; \\ degef &= a_{67}a_{78}a_{87}a_{78}a_{82}; \\ ahekmllld &= a_{13}a_{37}a_{78}a_{85}a_{56}a_{66}a_{66}a_{67}; \end{aligned}$$

because

$$a_{13} = a, a_{35} = b, a_{56} = m, a_{66} = l, a_{67} = d, \text{ etc.}$$

The expression " $a_{ij}a_{jj}a_{jm}$ " denotes a path from node i to j , with a self-loop at j and then a link from node j to node m . The expression " $a_{ij}a_{jk}a_{km}a_{mi}$ " denotes a path from node i back to node i via nodes j , k , and m . An expression such as " $a_{ik}a_{km}a_{mj} + a_{in}a_{np}a_{pj}$ " denotes a pair of paths between nodes i and j , one going via nodes k and m and the other via nodes n and p .

This notation may seem cumbersome, but it's not intended for working with the matrix of a graph but for expressing operations on the matrix. It's a very compact notation. For example,

denotes the set of all possible paths between nodes i and j via one intermediate node. But because " i " and " j " denote any node, this expression is the set of all possible paths between any two nodes via one intermediate node.

The **transpose** of a matrix is the matrix with rows and columns interchanged. It is denoted by a superscript letter "T," as in A^T . If $C = A^T$ then $c_{ij} = a_{ji}$. The **intersection** of two matrices of the same size, denoted by $A \# B$ is a matrix obtained by an element-by-element multiplication operation on the entries. For example, $C = A \# B$ means $c_{ij} = a_{ij} \# b_{ij}$. The multiplication operation is usually boolean AND or set intersection. Similarly, the **union** of two matrices is defined as the element-by-element addition operation such as a boolean OR or set union.

4. RELATIONS

4.1. General

This isn't a section on aunts and uncles but on abstract relations that can exist between abstract objects, although family and personal relations can also be modeled by abstract relations, if you want to. A **relation** is a property that exists between two (usually) objects of interest. We've had many examples of relations in this book. Here's a sample, where a and b denote objects and R is used to denote that a has the relation R to b :

1. "Node a is connected to node b " or aRb where " R " means "is connected to."
2. " $a \geq b$ " or aRb where " R " means "greater than or equal."
3. " a is a subset of b " where the relation is "is a subset of."
4. "It takes 20 microseconds of processing time to get from node a to node b ." The relation is expressed by the number 20.
5. "Data object X is defined at program node a and used at program node b ." The relation between nodes a and b is that there is a *du* chain between them.

Let's now redefine what we mean by a graph.

graph consists of a set of abstract objects called **nodes** and a relation R between the nodes. If aRb , which is to say that a has the relation R to b , it is denoted by a **link** from a to b . In addition to the fact that the relation exists, for some relations we can associate one or more properties. These are called **link weights**. A link weight can be numerical, logical, illogical, objective, subjective, or whatever. Furthermore, there is no limit to the number and type of link weights that one may associate with a relation.

"Is connected to" is just about the simplest relation there is: it is denoted by an unweighted link. Graphs defined over "is connected to" are called, as we said before, **connection matrices**.^{*} For more general relations, the matrix is called a **relation matrix**.

^{*} Also "adjacency matrix"; see EVEN79.

4.2. Properties of Relations

4.2.1. General

The least that we can ask of relations is that there be an algorithm by which we can determine whether or not the relation exists between two nodes. If that's all we ask, then our relation arithmetic is too weak to be useful. The following sections concern some properties of relations that have been found to be useful. Any given relation may or may not have these properties, in almost any combination.

4.2.2. Transitive Relations

A relation R is **transitive** if aRb and bRc implies aRc . Most relations used in testing are transitive. Examples of transitive relations include: is connected to, is greater than or equal to, is less than or equal to, is a relative of, is faster than, is slower than, takes more time than, is a subset of, includes, shadows, is the boss of. Examples of **intransitive** relations include: is acquainted with, is a friend of, is a neighbor of, is lied to, has a *du* chain between.

4.2.3. Reflexive Relations

A relation R is **reflexive** if, for every a , aRa . A reflexive relation is equivalent to a self-loop at every node. Examples of reflexive relations include: equals, is acquainted with (except, perhaps, for amnesiacs), is a relative of. Examples of **irreflexive relations** include: not equals, is a friend of (unfortunately), is on top of, is under.

4.2.4. Symmetric Relations

A relation R is **symmetric** if for every a and b , aRb implies bRa . A symmetric relation means that if there is a link from a to b then there is also a link from b to a ; which furthermore means that we can do away with arrows and replace the pair of links with a single **undirected** link. A graph whose relations are not symmetric is called a **directed graph** because we must use arrows to denote the relation's direction. A graph over a symmetric relation is called an **undirected graph**.^{*} The matrix of an undirected graph is symmetric ($a_{ij} = a_{ji}$ for all i, j).

^{*} Strictly speaking, we should distinguish between undirected graphs (no arrows) and bidirected graphs (arrow in both directions); but in the context of testing applications, it doesn't matter.

Examples of symmetric relations: is a relative of, equals, is alongside of, shares a room with, is married (usually), is brother of, is similar (in most uses of the word), OR, AND, EXOR. Examples of **asymmetric** relations: is the boss of, is the husband of, is greater than, controls, dominates, can be reached from.

4.2.5. Antisymmetric Relations

A relation R is **antisymmetric** if for every a and b , if aRb and bRa , then $a = b$, or they are the same elements.

Examples of antisymmetric relations: is greater than or equal to, is a subset of, time. Examples of **nonantisymmetric** relations: is connected to, can be reached from, is greater than, is a relative of, is a friend of.

4.3. Equivalence Relations

An **equivalence relation** is a relation that satisfies the reflexive, transitive, and symmetric properties. Numerical equality is the most familiar example of an equivalence relation. If a set of objects satisfy an equivalence relation, we say that they form an **equivalence class** over that relation. The importance of equivalence classes and relations is that any member of the equivalence class is, with respect to the relation, equivalent to any other member of that class. The idea behind **partition-testing strategies** such as domain testing and path testing, is that we can partition the input space into equivalence classes. If we can do that, then testing any member of the equivalence class is as effective as testing them all. When we say in path testing that it is sufficient to test one set of input values for each member of a branch-covering set of paths, we are asserting that the set of all input values for each path (e.g., the path's domain) is an equivalence class with respect to the relation that defines branch-testing paths. If we furthermore (incorrectly) assert that a strategy such as branch testing is sufficient, we are asserting that satisfying the branch-testing relation implies that all other possible equivalence relations will also be satisfied—that, of course, is nonsense.

4.4. Partial Ordering Relations

A **partial ordering relation** satisfies the reflexive, transitive, and antisymmetric properties. Partial ordered graphs have several important properties: they are loop-free, there is at least one maximum element, there is at least one minimum element, and if you reverse all the arrows, the resulting graph is also partly ordered. A **maximum element** a is one for which the relation xRa does not hold for any other element x . Similarly, a **minimum element** a , is one for which the relation aRx does not hold for any other element x . Trees are good examples of partial ordering. The importance of partial ordering is that while strict ordering (as for numbers) is rare with graphs, partial ordering is common. Loop-free graphs are partly ordered. We have many examples of useful partly ordered graphs: call trees, most data structures, an integration plan. Also, whereas the general control-flow or data-flow graph is not always partly ordered, we've seen that by restricting our attention to partly ordered graphs we can sometimes get new, useful strategies. Also, it is often possible to remove the loops from a graph that isn't partly ordered to obtain another graph that is.

5. THE POWERS OF A MATRIX

5.1. Principles

Each entry in the graph's matrix (that is, each link) expresses a relation between the pair of nodes that corresponds to that entry. It is a direct relation, but we are usually interested in indirect relations that exist by virtue of intervening nodes between the two nodes of interest. Squaring the matrix (using suitable arithmetic for the weights) yields a new matrix that expresses the relation between each pair of nodes via one intermediate node under the assumption that the relation is transitive. The square of the matrix represents all path segments two links long. Similarly, the third power represents all path segments three links long. And the k th power of the matrix represents all path segments k links long. Because a matrix has at most n nodes, and no path can be more than $n - 1$ links long without incorporating some path segment already accounted for, it is generally not necessary to go beyond the $n - 1$ power of the matrix. As usual, concatenation of links or the weights of links is represented by multiplication, and parallel links or path expressions by addition.

Let A be a matrix whose entries are a_{ij} . The set of all paths between any node i and any other node j (possibly itself), via all possible intermediate nodes, is given by

As formidable as this expression might appear, it states nothing more than the following:

1. Consider the relation between every node and its neighbor.
2. Extend that relation by considering each neighbor as an intermediate node.
3. Extend further by considering each neighbor's neighbor as an intermediate node.
4. Continue until the longest possible nonrepeating path has been established.
5. Do this for every pair of nodes in the graph.

5.2. Matrix Powers and Products

Given a matrix whose entries are a_{ij} , the square of that matrix is obtained by replacing every entry with. More generally, given two matrices A and B , with entries a_{ik} and b_{kj} , respectively, their product is a new matrix C , whose entries are c_{ij} , where:

The indexes of the product [e.g., (3,2) in C_{32}] identify, respectively, the row of the first matrix and the column of the second matrix that will be combined to yield the entry for that product in the product matrix. The C_{32} entry is obtained by combining, element by element, the entries in the third row of the A matrix with the corresponding elements in the second column of the B matrix. I use two hands. My left hand points and traces across the row while the right points down the column of B . It's like patting your head with one hand and rubbing your stomach with the other at the same time: it takes practice to get the hang of it. Applying this to the matrix of [Figure 12.1g](#) yields

$A^2A = AA^2$; that is, matrix multiplication is associative (for most interesting relations) if the underlying relation arithmetic is associative. Therefore, you can get A^4 in any of the following ways: A^2A^2 , $(A^2)^2$, A^3A , AA^3 . However, because multiplication is not necessarily commutative, you must remember to put the contribution of the left-hand matrix in front of the contribution of the right-hand matrix and not inadvertently reverse the order. The loop terms are important. These are the terms that appear along the **principal diagonal** (the one that slants down to the right). The initial matrix had a self-loop about node 5, link h . No new loop is revealed with paths of length 2, but the cube of the matrix shows additional loops about nodes 3 (bfe), 4 (feb), and 5 (ebf). It's clear that these are the same loop around the three nodes.

If instead of link names you use some other relation and associated weight, as in [Chapter 8](#), and use the appropriate arithmetic rules, the matrix displays the property corresponding to that relation. Successive powers of the matrix

display the property when considering paths of length exactly 2, exactly 3, and so on. The methods of [Chapter 8](#) and the applications discussed there carry over almost unchanged into equivalent matrix methods.

5.3. The Set of All Paths

Our main objective is to use matrix operations to obtain the set of all paths between all nodes or, equivalently, a property (described by link weights) over the set of all paths from every node to every other node, using the appropriate arithmetic rules for such weights. The set of all paths between all nodes is easily expressed in terms of matrix operations. It's given by the following infinite series of matrix powers:

This is an eloquent, but practically useless, expression. Let I be an n by n matrix, where n is the number of nodes. Let I 's entries consist of multiplicative identity elements along the principal diagonal. For link names, this can be the number "1." For other kinds of weights, it is the multiplicative identity for those weights. The above product can be re-phrased as:

$$A(I + A + A^2 + A^3 + A^4 \dots A^\infty)$$

But often for relations, $A + A = A$, $(A + I)^2 = A^2 + A + A + I$, $A^2 + A + I$. Furthermore, for any finite n ,

$$(A + I)^n = I + A + A^2 + A^3 \dots A^n$$

Therefore, the original infinite sum can be replaced by

This is an improvement, because in the original expression we had both infinite products and infinite sums, and now we have only one infinite product to contend with. The above is valid whether or not there are loops. If we restrict our interest for the moment to paths of length $n - 1$, where n is the number of nodes, the set of all such paths is given by

This is an interesting set of paths because, with n nodes, no path can exceed $n - 1$ nodes without incorporating some path segment that is already incorporated in some other path or path segment. Finding the set of all such paths is somewhat easier because it is not necessary to do all the intermediate products explicitly. The following algorithm is effective:

1. Express $n - 2$ as a binary number.
2. Take successive squares of $(A + I)$, leading to $(A + I)^2$, $(A + I)^4$, $(A + I)^8$, and so on.
3. Keep only those binary powers of $(A + I)$ that correspond to a 1 value in the binary representation of $n - 2$.
4. The set of all paths of length $n - 1$ or less is obtained as the product of the matrices you got in step 3 with the original matrix.

As an example, let the graph have 16 nodes. We want the set of all paths of length less than or equal to 15. The binary representation of $n - 2$ (14) is $2^3 + 2^2 + 2$. Consequently, the set of paths is given by

This required one multiplication to get the square, squaring that to get the fourth power, and squaring again to get the eighth power, then three more multiplications to get the sum, for a total of six matrix multiplications without additions, compared to fourteen multiplications and additions if gotten directly.

A matrix for which $A^2 = A$ is said to be **idempotent**. A matrix whose successive powers eventually yields an idempotent matrix is called an **idempotent generator**—that is, a matrix for which there is a k such that $A^{k+1} = A^k$. The point about idempotent generator matrices is that we can get properties over all paths by successive squaring. A graph matrix of the form $(A + I)$ over a transitive relation is an idempotent generator; therefore, anything of interest can be obtained by even simpler means than the binary method discussed above. For example, the relation

“connected” does not change once we reach A^{n-1} because no connection can take more than $n - 1$ links and, once connected, nodes cannot be disconnected. Thus, if we wanted to know which nodes of an n -node graph were connected to which, by whatever paths, we have only to calculate: $A^2, A^2A^2 = A^4, \dots, A^p$, where p is the next power of 2 greater than or equal to n . We can do this because the relation “is connected to” is reflexive and transitive. The fact that it is reflexive means that every node has the equivalent of a self-loop, and the matrix is therefore an idempotent generator. If a relation is transitive but not reflexive, we can augment it as we did above, by adding the unit matrix to it, thereby making it reflexive. That is, although the relation defined over A is not reflexive, $A + I$ is. $A + I$ is an idempotent generator, and therefore there’s nothing new to learn for powers greater than $n - 1$, the length of the longest nonrepeating path through the graph. The n th power of a matrix $A + I$ over a transitive relation is called the **transitive closure** of the matrix.

5.4. Loops

Every loop forces us into a potentially infinite sum of matrix powers. The way to handle loops is similar to what we did for regular expressions. Every loop shows up as a term in the diagonal of some power of the matrix—the power at which the loop finally closes—or, equivalently, the length of the loop. The impact of the loop can be obtained by preceding every element in the row of the node at which the loop occurs by the path expression of the loop term starred and then deleting the loop term. For example, using the matrix for the graph of [Figure 12.1e](#), we obtain the following succession of powers for $A + I$:

The first matrix ($A + I$) had a self-loop about node 5 link h . Moving link h out to the other entries in the row, leaving the “1” entry at the (5,5) position, yielded the $h*g$ and the $h*e$ entries at (5,2) and (5,3) respectively. No new loops were closed for the second power. The third-power matrix has a loop about node 3, whose expression is $bfh*e$. Consequently, all other entries in that row are premultiplied by $(bfh*e)^*$, to yield $(bfh*e)^*(d + bc + bfh*g)$ for (3,2), $(bfh*e)*b$ for (3,4), and $(bfh*e)*bf$ for (3,5). Similarly, the $fh*eb$ term in the (4,4) entry is removed by multiplying every other nonzero term in the fourth row by $(fh*eb)^*$, and the elements in the fifth row is multiplied by $(h*ebf)^*$ to get rid of the loop.

Applying this method of characterizing all possible paths is straightforward. The above operations are interpreted in terms of the arithmetic appropriate to the weights used. Note, however, that if you are working with predicates and you want the logical function (predicate function, truth-value function) between every node and every other node, this may lead to loops in the logical functions. The specific “arithmetic” for handling predicate loops has not been discussed in this book. The information can be found in any good text on switching and automata theory, such as MILL66. Code that leads to predicate loops is not very nice, not well structured, hard to understand, and harder to test—and anyone who codes that way deserves the analytical difficulties arising therefrom. Predicate loops come about from declared or undeclared program switches and/or unstructured loop constructs. This means that the routine’s code remembers. If you didn’t realize that you put such a loop in, you probably didn’t intend to. If you did intend it, you should have expected the loop.

5.5. Partitioning Algorithm (BEIZ71, SOH084)

Consider any graph over a transitive relation. The graph may have loops. We would like to partition the graph by grouping nodes in such a way that every loop is contained within one group or another. Such a graph is partly ordered. There are many used for an algorithm that does that:

1. We might want to embed the loops within a subroutine so as to have a resulting graph which is loop-free at the top level.
2. Many graphs with loops are easy to analyze if you know where to break the loops.
3. While you and I can recognize loops, it’s much harder to program a tool to do it unless you have a solid algorithm on which to base the tool.

The way to do this is straightforward. Calculate the following matrix: $(A + I)^n \# (A + I)^{nT}$. This groups the nodes into strongly connected sets of nodes such that the sets are partly ordered. Furthermore, every such set is an equivalence class so that any one node in it represents the set. Now consider all the places in this book where we said “except for graphs with loops” or “assume a loop-free graph” or words to that effect. If you can bury the loop in a real subroutine, you can as easily bury it in a conceptual subroutine. Do the analysis over the partly ordered graph obtained by the partitioning algorithm and treat each loop-connected node set as if it is a subroutine to be examined in detail later. For each such component, break the loop and repeat the process.

You can recognize equivalent nodes by simply picking a row (or column) and searching the matrix for identical rows. Mark the nodes that match the pattern as you go and eliminate that row. Then start again from the top with another row and another pattern. Eventually, all rows have been grouped. The algorithm leads to the following equivalent node sets:

A = [1]
 B = [2,7]
 C = [3,4,5]
 D = [6]
 E = [8]

5.6. Breaking Loops And Applications

And how do you find the point at which to break the loops, you ask? Easy. Consider the matrix of a strongly connected subgraph. If there are entries on the principal diagonal, then start by breaking the loop for those links. Now consider successive powers of the matrix. At some power or another, a loop is manifested as an entry on the principal diagonal. Furthermore, the regular expression over the link names that appears in the diagonal entry tells you all the places you can or must break the loop. Another way is to apply the node-reduction algorithm (see below), which will also display the loops and therefore the desired break points.

The divide-and-conquer, or rather partition-and-conquer, properties of the equivalence partitioning algorithm is a basis for implementing tools. The problem with most algorithms is that they are computationally intensive and require of the order of n^2 or n^3 arithmetic operations, where n is the number of nodes. Even with fast, cheap computers it's hard to keep up with such growth laws. The key to solving big problems (hundreds of nodes) is to partition them into a hierarchy of smaller problems. If you can go far enough, you can achieve processing of the order of n , which is fine. The partition algorithm makes graphs into trees, which are relatively easy to handle.

6. NODE-REDUCTION ALGORITHM

6.1. General

The matrix powers usually tell us more than we want to know about most graphs. In the context of testing, we're usually interested in establishing a relation between two nodes — typically the entry and exit nodes—rather than between every node and every other node. In a debugging context it is unlikely that we would want to know the path expression between every node and every other node; there also, it is the path expression or some other related expression between a specific pair of nodes that is sought: for example, “How did I get *here* from *there*?” The method of this section is a matrix equivalence to the node-by-node reduction procedure of [Chapter 8](#). The advantage of the matrix-reduction method is that it is more methodical than the graphical method of [Chapter 8](#) and does not entail continually redrawing the graph. It's done as follows:

1. Select a node for removal; replace the node by equivalent links that bypass that node and add those links to the links they parallel.
2. Combine the parallel terms and simplify as you can.
3. Observe loop terms and adjust the outlinks of every node that had a self-loop to account for the effect of the loop.

4. The result is a matrix whose size has been reduced by 1. Continue until only the two nodes of interest exist.

6.2. Some Matrix Properties

If you numbered the nodes of a graph from 1 to n , you would not expect that the behavior of the graph or the program that it represents would change if you happened to number the nodes differently. Node numbering is arbitrary and cannot affect anything. The equivalent to renumbering the nodes of a graph is to interchange the rows and columns of the corresponding matrix. Say that you wanted to change the names of nodes i and j to j and i , respectively. You would do this on the graph by erasing the names and rewriting them. To interchange node names in the matrix, you must interchange both the corresponding rows and the corresponding columns. Interchanging the names of nodes 3 and 4 in the graph of [Figure 12.1g](#) results in the following:

If you redraw the graph based on c, you will see that it is identical to the original except that node 3's name has been changed to 4, and node 4's name to 3.

6.3. The Algorithm

The first step is the most complicated one: eliminating a node and replacing it with a set of equivalent links. Using the example of [Figure 12.1g](#), we must first remove the self-loop at node 5. This produces the following matrix:

The reduction is done one node at a time by combining the elements in the last column with the elements in the last row and putting the result into the entry at the corresponding intersection. In the above case, the f in column 5 is first combined with $h*g$ in column 2, and the result ($fh*g$) is added to the c term just above it. Similarly, the f is combined with $h*e$ in column 3 and put into the 4,3 entry just above it. The justification for this operation is that the column entry specifies the links entering the node, whereas the row specifies the links leaving the node. Combining every column entry with the corresponding row entries for that node produces exactly the same result as the node-elimination step in the graphical-reduction procedure. What we did was: $a_{45}a_{52} = a_{42}$ or $f \times h*g = a_{52}$, but because there was already a c term there, we have effectively created a parallel link in the (5,2) position leading to the complete term of $c + fh*g$. The matrix resulting from this step is

If any loop terms had occurred at this point, they would have been taken care of by eliminating the loop term and premultiplying every term in that row by the loop term starred. There are no loop terms at this point. The next node to be removed is node 4. The b term in the (3,4) position will combine with the (4,2) and (4,3) terms to yield a (3,2) and a (3,3) term, respectively. Carrying this out and discarding the unnecessary rows and columns yields

Removing the loop term yields

There is only one node to remove now, node 3. This will result in a term in the (1,2) entry whose value is

$$a(bfh*e)*(d + bc + bfh*g)$$

This is the path expression from node 1 to node 2. Stare at this one for awhile before you object to the $(bfh*e)*$ term that multiplies the d ; any fool can see the direct path via d from node 1 to the exit, but you could miss the fact that the routine could circulate around nodes 3, 4, and 5 before it finally took the d link to node 2.

6.4. Applications

6.4.1. General

The path expression is usually the most difficult and complicated to get. The arithmetic rules for most applications are simpler. In this section we'll redo applications from Chapter 8, using the appropriate arithmetic rules, but this time using matrices rather than graphs. Refer back to the corresponding examples in Chapter 8 to follow the successive stages of the analysis.

6.4.2. Maximum Number of Paths

The matrix corresponding to the graph on page 261 is on the opposite page. The successive steps are shown. Recall that the inner loop about nodes 8 and 9 was to be taken from zero to three times, while the outer loop about nodes 5 and 10 was to be taken exactly four times. This will affect the way the diagonal loop terms are handled.

6.4.3. The Probability of Getting There

6.5. Some Hints

Redrawing the matrix over and over again is as bad as redrawing the graph to which it corresponds. You actually do the work in place. Things get more complicated, and expressions get bigger as you progress, so you make the low-numbered boxes larger than the boxes corresponding to higher-numbered nodes, because those are the ones that are going to be removed first. Mark the diagonal lightly so that you can easily see the loops. With these points in mind, the work sheet for the timing analysis graph on page 272 looks like this:

7. BUILDING TOOLS

7.1. Matrix Representation Software

7.1.1. Overview

We draw graphs or display them on screens as visual objects; we prove theorems and develop graph algorithms by using matrices; and when we want to process graphs in a computer, because we're building tools, we represent them as linked lists. We use linked lists because graph matrices are usually very sparse; that is, the rows and columns are mostly empty.

7.1.2. Node Degree and Graph Density

The **out-degree** of a node is the number of outlinks it has. The **in-degree** of a node is the number of inlinks it has. The **degree** of a node is the sum of the out-degree and in-degree. The average degree of a node (the mean over all nodes) for a typical graph defined over software is between 3 and 4. The degree of a simple branch is 3, as is the degree of a simple junction. The degree of a loop, if wholly contained in one statement, is only 4. A mean node degree of 5 or 6 say, would be a very busy flowgraph indeed.

7.1.3. What's Wrong with Arrays?

We can represent the matrix as a two-dimensional array for small graphs with simple weights, but this is not convenient for larger graphs because:

1. *Space*—Space grows as n^2 for the matrix representation, but for a linked list only as kn , where k is a small number such as 3 or 4.
2. *Weights*—Most weights are complicated and can have several components. That would require an additional weight matrix for each such weight.

3. Variable-Length Weights—If the weights are regular expressions, say, or algebraic expressions (which is what we need for a timing analyzer), then we need a two-dimensional string array, most of whose entries would be null.

4. Processing Time—Even though operations over null entries are fast, it still takes time to access such entries and discard them. The matrix representation forces us to spend a lot of time processing combinations of entries that we know will yield null results.

The matrix representation is useful in building prototype tools based on untried algorithms. It's a lot easier to implement algorithms by using direct matrix representations, especially if you have matrix manipulation subroutines or library functions. Matrices are reasonable to about 20–30 nodes, which is good enough for testing most prototype tools.

7.1.4. Linked-List Representation

7.2. Matrix Operations

7.2.1. Parallel Reduction

This is the easiest operation. Parallel links after sorting are adjacent entries with the same pair of node names. For example:

```
node 17,21;x
,44;y
,44;z
,44;w
```

We have three parallel links from node 17 to node 44. We fetch the weight expressions using the y , z , and w pointers and we obtain a new link that is their sum:

```
node 17,21;x
,44;y (where  $y = y + z + w$ ).
```

7.2.2. Loop Reduction

Loop reduction is almost as easy. A loop term is spotted as a self-link. The effect of the loop must be applied to all the outlinks of the node. Scan the link list for the node to find the loop(s). Apply the loop calculation to every outlink, except another loop. Remove that loop. Repeat for all loops about that node. Repeat for all nodes. For example removing node 5's loop:

List Entry	Content	Content After
5	node 5,2;g →	node 5,2;h*g
	,3;e →	,3;h*e
	,5;h →	
	4,	4,
	5,	

7.2.3. Cross-Term Reduction

Select a node for reduction (see Section 7.3 below for strategies). The cross-term step requires that you combine every inlink to the node with every outlink from that node. The outlinks are associated with the node you've selected. The inlinks are obtained by using the back pointers. The new links created by removing the node will be associated with the nodes of the inlinks. Say that the node to be removed was node 4.

<i>List Entry</i>	<i>Content Before</i>	
2	node2,exit	node2,exit
	3,	3,
	4,	4,
	5,	5,
3	node3,2;d	node3,2;d
	,4;b	,2;bc
		,5;bf
	1,	1,
	5,	5,
4	node4,2;c	
	,5;f	
	3,	
5	node5,2;h*g	node5,2;h*g
	,3;h*e	,3;h*e
	4,	

As implemented, you can remove several nodes in one pass if you do careful bookkeeping and keep your pointers straight. The links created by node removal are stored in a separate list which is then sorted and thereafter merged into the master list.

7.2.4 Addition, Multiplication, and Other Operations

Addition of two matrices is straightforward. If you keep the lists sorted, then simply merge the lists and combine parallel entries.

Multiplication is more complicated but also straightforward. You have to beat the node's outlinks against the list's inlinks. It can be done in place, but it's easier to create a new list. Again, the list will be in sorted order and you use parallel combination to do the addition and to compact the list.

Transposition is done by reversing the pointer directions, resulting in a list that is not correctly sorted. Sorting that list provides the transpose. All other matrix operations can be easily implemented by sorting, merging, and combining parallels.

7.3. Node-Reduction Optimization

The optimum order for node reduction is to do lowest-degree nodes first. The idea is to get the lists as short as possible as quickly as possible. Nodes of degree 3 (one in and two out or two in and one out) reduce the total link count by one link when removed. A degree-4 node keeps the link count the same, and all higher-degree nodes increase the link count. Although this is not guaranteed, by picking the lowest-degree node available for reduction you can almost prevent unlimited list growth. Because processing is dominated by list length rather than by the number of nodes on the list, this strategy is effective. For large graphs with 500 or more nodes and an average degree of 6 or 7, the difference between not optimizing the node-reduction order and optimizing it was about 50: 1 in processing time.

8. Summary

1. Working with pictorial graphs is tedious and a waste of time. Graph matrices are used to organize the work.

2. The graph matrix is the tool of choice for proving things about graphs and for developing algorithms.
3. As implemented in tools, graph matrices are usually represented as linked lists.
4. Most testing problems can be recast into an equivalent problem about some graph whose links have one or more weights and for which there is a problem—specific arithmetic over the link weights. The link-weighted graph is represented by a relation matrix.
5. Relations as abstract operators are well understood and have interesting properties which can be exploited to create efficient algorithms. Properties of interest include transitivity, reflexivity, symmetry, asymmetry, and antisymmetry. These properties in various combinations define ordering, partial ordering, and equivalence relations.
6. The powers of a relation matrix define relations spanning one, two, three, and up to the maximum number of links that can be included in a path. The powers of the matrix are the primary tool for finding properties that relate any node to any other node.
7. The transitive closure of the matrix is used to define equivalence classes and to convert an arbitrary graph into a partly ordered graph.
8. The node reduction algorithm first presented in [Chapter 8](#) is redefined in terms of matrix operations.
9. There is an old and copious literature on graph matrices and associated algorithms. Serious tool builders should learn that literature lest they waste time reinventing ancient algorithms or

15.JNTU Syllabus with additional topics :

Topic to be covered	Additional Topics
UNIT I	
Purpose of testing	
Dichotomies	
Model of testing	
Consequences of bugs	
Taxonomy of bugs	Is complete testing possible
A Project case study	Some bug statistics
UNIT II	
Basic concepts of path testing	
Predicates	
Path predicates and achievable paths	
Path sensitizing	

Path instrumentation	Path instrumentation
Application of path testing	Testability tips
Unit-III	
Transaction flow , testing	
Transaction flow testing techniques	
Dataflow testing: Basics of dataflow testing	
Strategies in dataflow testing	Application tools
Applications of dataflow testing	Implementation comments
Unit-IV	
Domains and paths	
Nice & ugly domains	
Domain testing	
Domains and interfaces testing	
Domain and interface testing	
Domains and paths	
Nice & ugly domains	
Domain testing	
Domains and interfaces testing	Applications
Unit-V	
Path products & path expressions	
Reduction procedure	
applications	
Regular expressions	

Flow anomaly detection	
Unit-VI	
overview	
Decision tables	
Path expressions	
KV charts	
specifications	
Unit-7	
State graphs	
Good & Bad state graphs	
State testing	
Testability tips	
Unit-VIII	
Motivational overview	
Matrix of graph ,relations ,power of a matrix	
Node reduction algorithm	
Building tools	
Usage of JMeter and Win runner tools for functional/Regression testing	Application programs for tools

16.University Question papers of previous years

B.Tech IV Year I Semester Examinations May/June - 2013

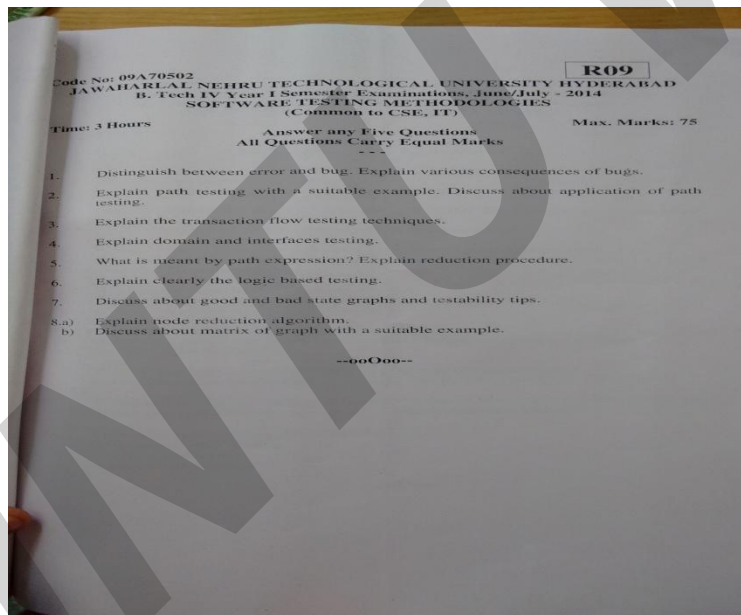
SOFTWARE TESTING METHODOLOGIES (Common to CSE, IT)

Time: 3 Hours Max. Marks: 75

Answer any FIVE Questions

All Questions carry equal marks

1. What is a bug? Give the taxonomy of bugs. Discuss the consequences of bugs. [15]
2. (a) What is meant by achievable path? Explain with illustrations.
(b) Discuss the applications of path testing. [7+8]
3. Explain the basics of data flow testing and the strategies in data flow testing. [15]
4. (a) Explain nice and ugly domains with examples.
(b) Discuss testability of domains and interfaces. [7+8]
5. What is a regular expression? Explain its role in flow anomaly detection with suitable examples. [15]
6. (a) Write about kv-chart.
(b) Discuss the significance of decision tables in logic based testing. [7+8]
7. What is meant by transition testing? With suitable state graphs explain transition testing. [15]
8. (a) How do you represent graphs in matrix formats? Discuss with examples.
(b) Explain node reduction algorithm.
(c) What is meant by power of a matrix. [15]



Code No: 09A70502

R09

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY, HYDERABAD

B. Tech IV Year I Semester Examinations, November/December-2012

SOFTWARE TESTING METHODOLOGIES

(Common to CSE, IT)

Time: 3 hours

Answer any five questions
All questions carry equal marks

Max. Marks: 75



- 1.a) Define the term "testing". Discuss its significance. [15]
- b) Explain about structural bugs and coding bugs. [15]
- 2.a) What are the assumptions of path testing? Discuss about effectiveness of path testing. [15]
- b) Discuss about assignment blindness, equality blindness and self blindness in path testing. [15]
- 3.a) The transaction flows are often ill structured. Discuss its reasons. [15]
- b) Discuss briefly about transaction flow testing techniques. [15]
- 4.a) What are domain bugs? Discuss how to test them? [15]
- b) State and explain with suitable examples various two-dimensional domain bugs. [15]
5. State various steps in path reduction procedure and explain their usage with a suitable example. [15]
- 6.a) Discuss how decision tables can be used as a basis for test case design. [15]
- b) Explain with an example the four variables KV charts. [15]
- 7.a) Discuss briefly about good state graphs and bad state graphs. [15]
- b) Explain the node reduction algorithm. [15]
8. Write short notes on the following:
 - a) Matrix powers and products.
 - b) Limitations of state testing.
 - c) Flow anomaly detection.



17. Question Bank

Descriptive Questions:

UNIT-1

1. Why is it impossible for a tester to find all the bugs in a system? Why might it not be necessary for a program to be completely free of defects before it is delivered to its customers? (10 M) ***
2. To what extent can testing be used to validate that the program is fit for its purpose. Discuss? (6 M) ***
3. What is meant by integration testing? Goals of Integration Testing? (8 M)
4. Explain white-box testing and behavioral testing? (8 M)
5. State and explain various dichotomies in software testing? (16 M)
6. Discuss about requirements, features and functionality bugs. (10 M)
7. What are control and sequence bugs? How they can be caught? (6 M)

UNIT-2

- 1) Consider the following flow - graph? Select optimal number of paths to achieve C1+C2 (statement coverage + branch coverage). (12 M) ***
- 2) Explain various loops with an example? (4 M)
- 3) Explain concatenated loops with an example? (4 M)
- 4) State and explain various kinds of predicate blindness with examples? (8 M)
- 5) What are link counters? Discuss their use in path testing? (8 M)*
- 6) Discuss Traversal marker with an example. (Link marker). (8 M)*
- 7) What is meant by Co - incidental Correctness with example (8 M)*
- 8) What is meant by statement testing and branch testing with an example. *(8 M)
- 9) State and explain various path selection rules.(8 M)*
- 10) What is meant by program's control flow? How is it useful for path testing? (8)
- 11) Discuss various flow graph elements with their notations. (8)

UNIT-3

1. Distinguish Control Flow and Transaction flow. (6 M) *
2. What is meant by transaction flow testing. Discuss its significance. (10 M)*
3. Discuss in detail data - flow testing strategies. (16 M)***
4. What are data - flow anomalies? How data flow testing can explore them? (16 M)
5. What are data-flow anomalies? How data flow testing can explore them? (16 M)
6. What is meant by a program slice? Discuss about static and dynamic program slicing. (8 M)
7. Explain the terms Dicing, Data-flow and Debugging. (8 M)
8. What is meant by data flow model? Discuss various components of it? (8 M)
9. Compare data flow and path flow testing strategies? (8 M)
10. Explain data-flow testing with an example. Explain its generalizations and limitations.(8 M)

UNIT-4

1. Discuss with example the equal - span range/Doman compatibility bugs.(8 M)*
2. Discuss in detail about testability of Domains.(8 M)*
3. What is meant by Domain Dimensionality. (8 M)
4. What is meant by nice - domain? Give an example for nice two - dimensional domain.(8 M).*
5. Discuss (8 M)
 - i. Linear domain boundaries
 - ii. Non linear domain boundaries
 - iii. Complete domain boundaries

- iv. Incomplete domain boundaries
- 6. Explain various properties related to Ugly-domains. (8 M) *
- 7. State and Explain various restrictions at domain testing processes. (10 M)
- 8. What is meant by domain testing? Discuss the various applications of domain testing? (10 M) *
- 9. With a neat diagram, explain the schematic representation of domain testing. (6 M)
- 10. Explain how one-dimensional domains are tested? (10 M)
- 11. Discuss in detail the domains and interface testing. (16 M)

UNIT-5

- 1. Explain Regular Expressions and Flow Anomaly detection. (16 M)**
- 2. Example Huang's theorem with examples (12 M)*
- 3. Reduction procedure algorithm for the following flow graph: (16 M)**
- 4. Write Short Notes on: (16 M) *
 - i. Distributive Laws
 - ii. Absorption Rule
 - iii. Loops
 - iv. Identity elements
- 5. Discuss Path Sums and Path Product. (8 M)
- 6. Discuss in brief applications of paths (8 M)

UNIT-6

- 1. Reduce the following functions using K-Maps (16 M) **
 $F(A,B,C,D) = P(4,5,6,7,8,12,13) + d(1,15)$
- 2. Whether the predicates are restricted to binary truth-values or not. Explain. (10 M)
- 3. What are decision tables? Illustrate the applications of decision tables. How is a decision table useful in testing. Explain with an example. (16 M) **
- 4. How can we determine paths in domains in Logic based testing? (8 M)
- 5. How the Boolean expression can be used in test case design (8 M)
- 6. Flow graphs are abstract representations of programs. Justify? (8 M)
- 7. Explain prime implicant, sum of product form and product of sum form. (8 M)
- 8. How can we form specifications into sentences? Write down different phrases that can be used for words? (8 M)
- 9. Explain about the ambiguities and contradictions in specifications.? (8 M)
- 10. Demonstrate by means of truth tables the validity of the following theorems of Boolean algebra: (8 M) **
 - i. Associative Laws
 - ii. Demorgan's theorems for three variables
 - iii. Distributive Law
 - iv. Absorption Rule

UNIT-7

- 1. The behavior of a finite state machine is invariant under all encodings. Justify? (16 M)**
- 2. Write testers comments about state graphs (8 M)**
- 3. What are the types of bugs that can cause state graphs? (8 M)*
- 4. What are the principles of state testing. Discuss advantages and disadvantages. (8 M)
- 5. Write the design guidelines for building finite state machine into code. (8 M)
- 6. What are the software implementation issues in state testing? (8 M)
- 7. Explain about good state and bad state graphs. (8 M)
- 8. Explain with an example how to convert specification into state-graph. Also discuss how contradictions can come out. (16 M)
- 9. Write short notes on: (16 M)
 - i. Transition Bugs
 - ii. Dead States
 - iii. State Bugs

iv. Encoding Bugs

UNIT-8

1. How can the graph be represented in Matrix form? (3 M)
2. Write a partition algorithm. (8 M)
3. Discuss node reduction algorithm. (8 M)**
4. How can a node reduction optimization be done. (6 M)
5. What are the matrix operations in tool building. (8 M)**
6. Discuss the algorithm for finding set of all paths (8 M)
7. How can a relation matrix be represented and what are the properties of relations? (8 M)
8. Explain cross-term reduction and node term reduction optimization. (8 M)
9. Write about matrix powers and products. (8 M)
10. Write about equivalence relation and partial ordering relation (8 M)
11. What are the advantages and disadvantages of array representations? (8 M)
12. Write about loops in matrix representation (8 M)
13. What are graph matrices and their applications? (16 M)
14. Discuss the linked list representation. (5 M)

Important concepts in exam point of view

1. Is complete testing possible? To what extent software should be tested?
2. Explain taxonomy of bugs? (May also ask to explain any one or two types of bugs in detail)
3. Explain path testing criteria and path selection criteria. Consider the following flow-graph? Select
optimal number of paths to achieve C1+C2 (statement coverage + branch coverage). (Diagram will change according to marks)
4. Explain the concept of Path Instrumentation. (May ask to explain path instrumentation or Co-incidental correctness or Link / Traversal marker or Link counters).
5. Explain Data-flow testing strategies. (May also ask to explain one of the strategies.)
6. Explain Data-flow anomalies in detail.
7. Explain the concept of Nice and Ugly domains.
8. Explain domains and interface testing. (includes closure compatibility, span compatibility etc)
9. Explain testing in one dimensional and two dimensional domains.
10. Explain the reduction procedure to evaluate the control flow graph to path expression.
(May ask to implement on a diagram).
11. Explain Regular expression and flow anomaly detection. (should also include Huang's theorem)
12. Explain the applications of paths (Max/Min. count of paths, probability, mean time).
13. Reduce the following functions using K-Maps
 $F(A,B,C,D) = P(4,5,6,7,8,12,13) + d(1,15)$
14. How can we form specifications into sentences? Write down different phrases that can be used for words? Explain about the ambiguities and contradictions in specifications?
15. Explain the theorems of Boolean algebra. How to get prime implicant with an example

18. ASSIGNMENT QUESTIONSUNIT-1

1. Why is it impossible for a tester to find all the bugs in a system? Why might it not be necessary for a program to be completely free of defects before it is delivered to its customers? (10 M) ***
2. To what extent can testing be used to validate that the program is fit for its purpose. Discuss? (6 M) ***
3. Explain white-box testing and behavioral testing? (8 M)
4. Discuss about requirements, features and functionality bugs. (10 M)

UNIT-2

1. 1) Consider the following flow - graph? Select optimal number of paths to achieve C1+C2 (statement coverage + branch coverage). (12 M) ***
- 5) What are link counters? Discuss their use in path testing? (8 M)*
- 6) Discuss Traversal marker with an example. (Link marker). (8 M)*
- 7) What is meant by Co - incidental Correctness with example (8 M)*
- 8) What is meant by statement testing and branch testing with an example. *(8 M)
- 9) State and explain various path selection rules.(8 M)*

UNIT-3

1. Distinguish Control Flow and Transaction flow. (6 M) *
2. What is meant by transaction flow testing. Discuss its significance. (10 M)*
3. Discuss in detail data - flow testing strategies. (16 M)***
9. Compare data flow and path flow testing strategies? (8 M)
- 10.Explain data-flow testing with an example. Explain its generalizations and limitations.(8 M)

UNIT-4

1. Discuss with example the equal - span range/Doman compatibility bugs.(8 M)*
2. Discuss in detail about testability of Domains.(8 M)*
3. What is meant by Domain Dimensionality. (8 M)
6. Explain various properties related to Ugly-domains. (8 M) *
8. What is meant by domain testing? Discuss the various applications of domain testing? (10 M) *

Unit – 5

1. Explain Regular Expressions and Flow Anomaly detection. (16 M)**
2. Example Huang's theorem with examples (12 M)*
3. Reduction procedure algorithm for the following flow graph: (16 M)**
4. Write Short Notes on: (16 M) *
 - i. Distributive Laws
 - ii. Absorption Rule
 - iii. Loops

Unit – 6

1. Reduce the following functions using K-Maps (16 M) **

$$F(A,B,C,D) = P(4,5,6,7,8,12,13)+d(1,15)$$
2. Whether the predicates are restricted to binary truth-values or not. Explain. (10 M)
3. What are decision tables? Illustrate the applications of decision tables. How is a decision table useful in testing. Explain with an example. (16 M) **
9. Explain about the ambiguities and contradictions in specifications.? (8 M)
- 10.Demonstrate by means of truth tables the validity of the following theorems of Boolean algebra: (8 M) **
 - i. Associative Laws
 - ii. Demorgan's theorems for three variables
 - iii. Distributive Law
 - iv. Absorption Rule

Unit – 7

1. The behavior of a finite state machine is invariant under all encodings. Justify? (16 M)**
2. Write testers comments about state graphs (8 M)**
3. What are the types of bugs that can cause state graphs? (8 M)*
8. Explain with an example how to convert specification into state-graph. Also

discuss how contradictions can come out. (16 M)

9. Write short notes on: (16 M)

- i. Transition Bugs
- ii. Dead States
- iii. State Bugs
- iv. Encoding Bugs

Unit – 8

2. Write a partition algorithm. (8 M)

3. Discuss node reduction algorithm. (8 M)**

5. What are the matrix operations in tool building. (8 M)**

11. What are the advantages and disadvantages of array representations? (8 M)

12. Write about loops in matrix representation (8 M)

13. What are graph matrices and their applications? (16 M)

19. Objective Questions :

UNIT-1

Choose the correct answer

1. According to phase 1 of testers mental life

a) testing shows software works

b) testing shows software doesn't work

c) testing = debugging

d) testing is not an act

2. Syntax errors are

a) data bugs

b) logic bugs

c) coding bugs

d) structure bugs

3. the first goal of testing is

a) bug detection

b) bug prevention

c) bug correction

d) to calculate the cost of the bug

4. the abbreviation of SQA is

a) system quality assistance

b) software quality assistance

c) system quality assurance

d) software quality assurance

5. what is the purpose of testing

a) to find the error

b) to show program has bugs

c) to correct the error

d) none of these

6. functional testing is also known as

a) black box

b) white box

c) glass box

d) open box

7. "the language syntax and semantics eliminates most of the bugs" this belief is known as

- a)control bug dominance
- b)data separation
- c)lingua salvator estimate**
- d)angelic testers

8. _____ is not the consequence of bugs.

- a)serious
- b)extreme
- c)infectious
- d)none**

9.the pesticide paradox testing is

- a)testing shows all bugs
- b)complexity of software grows to the limit of managerial ability
- c)a method to find bugs leaves subtler bugs**
- d)testing doesn't show all bugs

10.the remedies for testbugs bugs doesn't include

- a)debugging
- b)design automation
- c)test execution automation
- d)test consequences**

11.incorrect design of test cases is an example of

- a)logic bugs**
- b)data bugs
- c)processing bugs
- d)requirement bugs

12.the methods to prevent bugs other than testing are

- a)reviews
- b)syntax checking
- c)inspections
- d)all the above**

13.the program staff for the project should be

- a)10-20
- b)20-30**
- c)25-45
- d)15-20

14.the process of testing entire system is

- a)system testing**
- b)complete testing
- c)integration testing
- d)software testing

15.acceptance test is done

- a)after accepting a system
- b)to know users need
- c)to know whether to accept a system or not**
- d)to know the ability of the programmer

Fill in the blanks

- 1)can a bug free product delivery be guaranteed with testing? **No**
- 2)structural testing is also known as **white box testing**
- 3)the final recipient of the system is known as **user**

- 4)no two systems would be identical but they can have 75 %of code in common
- 5)the levels of testing are component,integration,system,acceptance
- 6)the cost that are included in calculating the importance of bugs are correction installation,consequential
- 7)main()
 {
 int a,b;
 c=a+b;
 }
 the above is example of initialization bugs
- 8)the most important criteria for designing interface is robustness
- 9) hardware and software are considered as programs environment
- 10)consequences of bugs range from mild to catastrophic
- 11) **Integration** is the process in which components are combined to form a larger component
- 12)two goals fo testing are bug prevention and bug detection
- 13) angelic testers is a belief that testers are better at testing than programmers are at code design
- 14) testing starts from known conditions
- 15)two types of test cases are formal and informal

UNIT-2

Choose the correct answer

1. _____ is the graphical representation of programs control structure

- a)flowchart
- b)control flowgraph**
- c)graphic matrix
- d)none

2.junction is generally denoted

- a)rectangle
- b)rhombus
- c)circle**
- d)line

3. _____ are more preferred in process design

- a)flow charts
- b)algorithms
- c)flow graphs**
- d)none

4.100% statement coverage is denoted by

- a)C1**
- b)C2
- c)C1+C2
- d)C1-C2

5.the path seleted doesn't follow the rule of

- a)from entry to exit
- b)simple
- c)short
- d)containing loops**

6.which is true in the case of flow charts

- a)there are fixed rules in designing
- b)there are no rules for designing
- c)detai;s of process blocks are not shown**
- d)it ignores all process steps

7.path segments are also known

a)sub paths

b)off paths

c)subways

d)path parameters

8.path testing with C1+C2 is tool for rehosting

a)new software

b)old software

c)off shelf software

d)component based software

9.the _____ is what we expect to happen as a result of test

a)outcome

b)output

c)result

d)none

10.the Boolean expression associated with the path is

a)predicate

b)Boolean path

c)path expression

d)predict expression

11.the methods of instrumentation in path testing does not include

a)link markers

b)link counters

c)bitmap

d)none of these

12.if every combination of values of two variables cannot be independently specified then they are said to be

a)correlated

b)independent

c)dependent

d)uncorrelated

13.the last step in heuristic procedure for path sensitizing is

a)examining the predicates

b)usage of predicates

c)identifying the variables

d)satisfying the inequalities

14.correct: $x=7 \dots \text{if } y>0 \text{ then}$

error: $x=7 \dots \text{if } x+y>0 \text{ then}$

is an example of

a)equality blindness

b)assignment blindness

c)self blindness

d)none of these

15the name given to the link is known as

a)link weight

b)link counter

c)link marker

d)none

fill in the blanks

- 1) **process block** is the sequence of the program sequence of program statements which do not have any decisions or junctions
- 2) a point where control flow can merge is **junction**
- 3) the arrows in the control flow graph are **links**
- 4) if all the paths in the program are covered then it is known as **100% path coverage**
- 5) in path testing, the path should be taken in such a way to achieve **C1+C2**
- 6) three kinds of loops **nested, concatenated, horrible**
- 7) path testing is first used on **modified** component in case of maintenance
- 8) path testing is used in **unit** testing for new software
- 9) link marker is the effective form of **instrumentation**
- 10) a predicate associated with a path is **path predicate**
- 11) the situation where a desired path is achieved for wrong reasons is **testing blindness**
- 12) the order of heuristic procedure for sensitizing path checks for predicate coverage **independent uncorrelated, correlated, dependent, correlated dependent**
- 13) 100% branch coverage can also be called as **100% link coverage**
- 14) **branch and statement** coverage are minimum mandatory requirements now a days
- 15) the case statements or jump tables are examples of **multiway** branches

UNIT 3

Choose the correct answer

1. absorption can be dealt in a best way by
 - a) **following the predicator as the primary flow**
 - b) following the parent flow from the starting o the end
 - c) following the birth of each parent
 - d) following the additional flow to the daughter
2. C, P in case of data object state and usage are
 - a) creation, participation
 - b) creation, postulation
 - c) **calculation, predicate**
 - d) calculation, postulation
3. the statement that is true in case of transactional flow instrumentation is
 - a) dispatchers are needed
 - b) pay off is counters
 - c) counters are useful
 - d) **counters are not useful**
4. _____ are assumed problematic for software designers.
 - a) births
 - b) absorptions
 - c) conjugations
 - d) **all the above**
5. the methods of sensitization in transaction flows include
 - a) using of patches
 - b) use break points
 - c) counters
 - d) **processing queue**
6. FSM implementation is an example of
 - a) recovery
 - b) **transaction dispatcher**
 - c) control flow
 - d) processing queues
7. the state of dataobjects cannot be
 - a) created

- b)killed
- c)used
- d)modified**

8.main()

```
{  
    int a=10;  
    int b=10;  
}
```

the above is the example of

- a)dk anomaly**
- b)du anomaly
- c)dd anomaly
- d)none

9.direct relation does not exist between processes and decisions in

- a)data flow diagram
- b)transaction flow diagram**
- c)both a) and b)
- d)none

10.the path segment for which every node is visited atleast once is said to be

- a)du path
- b)simple path segment
- c)loop free path**
- d)definition clear path segment

11.every definition of every variable is covered in

- a)ad strategy**
- b)apu strategy
- c)au strategy
- d)adup strategy

12.sequential machine consists of _____ processes

- a)one**
- b)two
- c)many
- d)infinite

13.the normal sequence with respect to life time of a variable among following is

- a)kk
- b)dd
- c)dk
- d)uu**

```
14.a=0;  
    b=a;
```

the statement represents the sequence of

- a)du**
- b)uu
- c)ud
- d)dd

15.the correct sequence of transaction flow is

- a)input,validate,acknowledge,record transaction**
- b)input,acknowledge,record transaction

- c) acknowledge, input, validate record, transaction
- d) record transaction, input, validate, acknowledge

fill in the blanks

- 1) the transaction flow graph is used for **functional** testing
- 2) a **transaction** is a unit of work done
- 3) the complications of transaction flows are **births** and **merges**
- 4) transaction flows are natural agenda for **reviews and inspections**
- 5) TCB stands for **transaction control block**
- 6) static analysis is done at **compiling** time
- 7) the three anomaly states are **ku, dk, dd**
- 8) to bridge the gap between debugging and testing is the concept of **data flow testing**
- 9) in arrays **dynamic** analysis is used
- 10) if then else, do while are examples of **predicate** nodes
- 11) maintenance testing is similar to **debugging**
- 12) **aw** is derived from ACP+P by dropping p-use
- 13) **break points** are used in case of hard to sensitize path
- 14) the strategy in which for variable and every definition of variable includes at least one definition free path from the definition to every predicate is known as all **p uses/some c uses**
- 15) **von neuman** machines have the feature of interchangeable storage of instructions

UNIT-4**Choose the correct answer**

- 1. using $x+y \geq 10$ when the correct equation is $x+y \geq 5$ is an example of
 - a) **closure bug**
 - b) titled bug
 - c) missing boundary
 - d) shifted boundary
- 2. if two distinct domains are overlapped, then they are said to be
 - a) ambiguous
 - b) overspecified
 - c) **contradictory**
 - d) unambiguous
- 3. the complex domains have the generic cases of
 - a) disconnected, concavities and connected edges
 - b) **disconnected, holes, concavities**
 - c) connected edges, holes, concavities
 - d) connected edges, disconnected and holes
- 4. the technique of expecting the process is true for all values in domain by checking for one random value in the same domain
 - a) coincidental correctness
 - b) linear vector space
 - c) loop free
 - d) **representative outcome**
- 5. if domain boundaries differ by a constant they are known as
 - a) **system boundaries**
 - b) irregular boundaries
 - c) orthogonal boundaries
 - d) convex boundaries
- 6. nice domains should not be
 - a) linear
 - b) simply connected
 - c) **concave**
 - d) complete
- 7. domain testing is a form of
 - a) black box testing
 - b) **white box testing**

- c) closed box testing
 - d) none
8. the point that lies between two specific points of a domain is
- a) off point
 - b) on point
 - c) boundary point
 - d) extreme point**
9. domains are more useful in
- a) interface testing**
 - b) system testing
 - c) unit testing
 - d) acceptance testing
10. span is defined as
- a) largest value
 - b) range of values**
 - c) smallest value
 - d) middle value
11. the linearization transformation which yields infinite polynomials is
- a) taylor series
 - b) laplace transforms
 - c) logarithmic transforms
 - d) polynomial transforms**
12. a domain can be considered as
- a) set**
 - b) routine
 - c) boundary
 - d) congregation
13. the span of the number space if nice domains are complete
- a) +infinity to -infinity in some dimensions
 - b) +infinity to -infinity in all dimensions
 - c) -infinity to +infinity in all dimensions**
 - d) -infinity to +infinity in some dimensions
14. the compatibility of callers range and called routines domain is confirmed by
- a) testing all variables at a time
 - b) testing only one variable
 - c) not testing any variable
 - d) testing every input individually**
15. the example of linearization transformation in domain testing
- a) domain variables
 - b) laplace transforms
 - c) polynomials**
 - d) algorithmic complexity

fill in the blanks

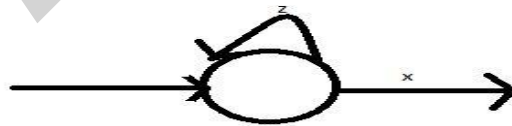
- 1) in closed boundary, the values of the boundary belongs to the same domain
- 2) the number of planes in which the domain exists is known as domain dimensionality
- 3) specified domains are incomplete or inconsistent
- 4) the special case of overlapping is dual closure assignment
- 5) COO OOI stand for "closed off outside, open off inside"
- 6) If $10x+7y \geq 70$ is taken instead of $7x+10y \geq 70$ tilted boundary error occurs
- 7) Predicates that are defined by equality are known as equality predicates
- 8) Random testing is done when direct test cases take long time
- 9) A strategy of testing which is applied to one or two variables based on specification is called domain testing
- 10) The interface test explores the correctness of caller unit test, integration test and called unit test
- 11) $F(x) \geq k$ represents systematic boundary in case of nice and ugly domains

- 12) The strategy of dividing a program input space into domain and the values in domain are equivalent is called as **partition testing**
- 13) If the span of boundary is from +infinity to -infinity then they are known as **complete boundaries**
- 14) **Boundary point** is a point with in epsilon neighbourhood where points are both in domain and not in domain
- 15) In union of specified domains is incomplete the domain is said to be **ambiguos**

Unit-5

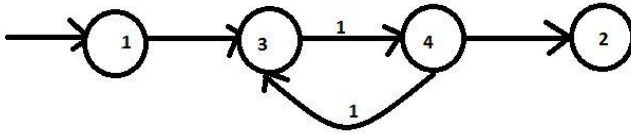
Choose the correct answer:

1. $aa^* =$
 - a) a^*a
 - b) a^+
 - c) a^*
 - d) **Both a and b**
2. Two successive path segments is expressed by concatenation is known as
 - a) **Path product**
 - b) Path sum
 - c) Path
 - d) Path Expression
3. The first steps in node reduction is
 - a) Combine all parallel links
 - b) **Combine all Serial links**
 - c) Remove self loops
 - d) Remove the node
4. The weight expression of series path of A and B in lower path count arithmetic is
 - a) $W_A + W_B$
 - b) W_A
 - c) **$\text{Max}(W_A, W_B)$**
 - d) $\text{Min}(W_A, W_B)$
5. If the probability of looping node is P_L , the probability of non looping node is
 - a) **$1 - P_L$**
 - b) $1 + P_L$
 - c) P_L
 - d) P_L^*
6. Which among the following is not a complementary operation
 - a) PUSH/POP
 - b) **START/STOP**
 - c) OPEN/CLOSE
 - d) **None**
7. The main goal of reduction process is
 - a) To reduce nodes
 - b) **To retain only entry and exit nodes**
 - c) Remain with single node
 - d) Remove loops
8. The main step in achieving the node reduction is
 - a) parallel term
 - b) Serial term
 - c) Loop term
 - d) **Cross Term**
9. To remove elements from stack, the operation used is
 - a) **POP**
 - b) Remove
 - c) Delete
 - d) None
10. If G is get operation then $G+G$ would be
 - a) $2G$
 - b) **G**
 - c) G^2
 - d) 0
11. The path expression for the following loop would be



- a) **z^*x**
- b) zx
- c) zx^*
- d) z^+x
12. $1^+ + 1 =$
 - a) 1^*
 - b) 1
 - c) **Both a and b**
 - d) None
13. The weight of loops in counting the no of paths in a flow graph is
 - a) $\prod_{i=0}^n W_A^i$
 - b) **$\sum_{i=0}^n W_A^i$**
 - c) $2 \sum_{i=0}^n W_A^i$
 - d) $2 \prod_{i=0}^n W_A^i$


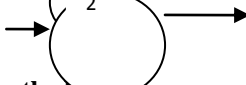
14. The equivalent link weight for the graph would be



- a) 1
b) 2
c) 3
d) 4

15. The transformation for the given graph would be



- a) 
b) 
c) 
d) 

Fill in the blanks:

- _____ denote the parallel set of paths.
- Self loop can be removed by replacing it with _____ if a is the path expression of that loop.
- The theorem that is helpful to reduce regular expression is _____
- Huang's theorem is generalized to cover the sequences of greater length than _____ characters.
- _____ Methods cannot detect whether a path is achievable or not.
- If A and B are two parallel paths then their equivalent path would be _____
- _____ are algebraic representation of sets of paths.
- Is path product commutative? _____
- The two kinds of loops are _____ and _____.
- Path expression would be easy to analyze compared to _____.
- If the loop is iterated k times then the contribution of loop is calculated by using _____.
- _____ operation is used to insert elements into stack.
- If $P1$ is PUSH and $P2$ is POP then $P1 * P2$ would be _____.
- The only limitation in node by node reduction is it may not work correctly in case of _____

Answers:

- Path sum
- a^*
- Huang's theorem
- Two
- Static Flow Analysis
- $A+B$
- Path Expression
- NO
- Self loop, loop on path expression
- Flow Graph
- $k/(K+1)$
- PUSH
- 1
- Unachievable paths
- Data Flow Anomaly

Unit-6

Choose the correct answer:

1. The decision table has a disadvantage of
 - a) Insufficient program logic
 - b) No clarity
 - c) Implicit relation to specification
 - d) Low level maintainability**
2. If there are k predicates then the maximum number of predicate cases possible are
 - a) 2^k**
 - b) 2k
 - c) $2^k - 1$
 - d) $2k - 1$
3. The propositional calculus in which there are more than two truth values is known as
 - a) Multiway Predicate
 - b) Multivalued logic**
 - c) Multiway truth
 - d) None
4. "FORTRAN 3 WAY IF" is an example of
 - a) Multiway Predicate**
 - b) Multivalued logic
 - c) both a and b
 - d) None
5. According to absorptive law, $A + AB =$
 - a) B
 - b) $A + B$**
 - c) $A + B$
 - d) $\bar{A} + B$
6. $ABC + BCD + CDE + EFG$ is in the form of
 - a) Sum of products**
 - b) Product of additions
 - c) Reduction of sums
 - d) Product of sums
7. $ABCD + BCD + CD + AB$ can be reduced to
 - a) $CD + AB$**
 - b) 1
 - c) CD
 - d) AB
8. The first step in reducing Boolean expression is
 - a) Replace identical terms
 - b) Remove parenthesis**
 - c) The term containing a variable and its complement is removed
 - d) None
9. The order of grouping in KV charts should be
 - a) Octets, islands, pairs, quads**
 - b) Octets, quads, pairs, islands
 - c) quads, pairs, islands, Octets
 - d) islands, pairs, quads, Octets
10. For n predicates, the no of interchanges for each combination of predicate truth values would be
 - a) $n(n-1)/2$**
 - b) 2n
 - c) n^2
 - d) $n(n-1)$
11. The list of names of conditions is known as
 - a) Condition Stub**
 - b) Condition Entry
 - c) Action Entry
 - d) Action stub
12. The rules that are used when all the rules failed to meet the conditions are
 - a) Contitional rules
 - b) Action rules
 - c) Action entries
 - d) Default rules**
13. If there are 2 immaterial cases in a table then the expansion would result in _____ columns or cases
 - a) 4**
 - b) 8
 - c) 16
 - d) 2
14. Which among the following is true for predicate values
 - a) They are not restricted to binary truth values
 - b) They are restricted to binary truth values in logic based testing
 - c) Both a and b
 - d) None
15. According to laws of Boolean Algebra expression $AB =$
 - a) $\bar{A} \cdot \bar{B}$
 - b) $A + B$**
 - c) $\bar{A} + \bar{B}$
 - d) $A + B$

Fill in Blanks:

1. The logic in the simplest form is _____
2. _____ gathers knowledge from knowledge repository or domain such as Engineering law into Database.
3. _____ is the table which consists of set of conditions and corresponding action.
4. The table's translator will check for _____ of source table and default rules to be filled.
5. The decision tables are used for test case design when decision table act as _____
6. The entries which do not play any role in decision table are known as _____
7. The process whose come out is a truth value is known as _____
8. $ABCD + ABC + AB + A$ can be reduces to _____
9. _____ is the one which covers min terms or max terms.
10. KV charts are used to analyze _____

11. Don't care conditions generally occur in _____
12. The elements of decision table are _____
13. The translator will convert decision table into _____
14. The consistency and completeness of the decision table can be done by expanding tables of _____.
15. $AB+AC=$ _____.

Answers:

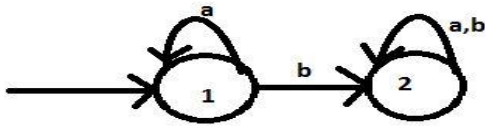
1. Boolean Algebra
2. Knowledge based systems
3. Decision Table
4. Consistency and completeness
5. Specification
6. Immaterial entries
7. Predicate
8. A
9. Implicant
10. Specifications
11. Digital electronics
12. Condition stub, condition entry, action stub, action entry
13. Source code
14. Immaterial cases
15. $A(B+c)$

Unit-7

1. The elements of state graph does not include
 - a) states
 - b) input/output
 - c) transitions
 - d) State machines**
2. The starting step of implementation of the state graph is
 - a) Implementation & operation**
 - b) Input encoding
 - c) output encoding
 - d) State codes
3. The bugs in state graphs are due to
 - a) No of states
 - b) Impossible states
 - c) Equivalent states
 - d) All of the above**
4. The state where no input sequence can reach is known as
 - a) Dead state
 - b) Reachable state
 - c) Unreachable state**
 - d) Ambiguous State
5. Outputs in a state graph are not dependent on
 - a) Transitions
 - b) Inputs
 - c) Both a and b**
 - d) None
6. The bugs that are not caused in state graphs are
 - a) State bugs
 - b) Output errors
 - c) Encoding bugs
 - d) None**
7. The advantage of state testing does not include,
 - a) less expensive
 - b) Cannot catch deep bugs**
 - c) provides rewards during test design
 - d) is used to detect specified input
8. State graph represents
 - a) time
 - b) Sequence**
 - c) States
 - d) None
9. The state where "engine is not working and vehicle is running" is example of
 - a) Reachable state
 - b) Impossible state**
 - c) Equivalent state
 - d) Contradictory state
10. The steps in finding no of states does not include
 - a) Identifying all component factors
 - b) Identifying all allowable factors
 - c) no of states=product of no of allowable values
 - d) of all factors

d) No of states = sum of all allowable values of all factors

11.



Is the above state graph good?

- a) Yes
 b) No
 c) Cannot be determined
 d) Both a and b

12. Contradictory and ambiguous state occur in case of

- a) Transaction bugs
 b) Encoding bugs
 c) Output errors
 d) State bugs

13. Which among the following is not the condition of inessential finite state behaviour

- a) It can be obtained by parallel program in dataflow machine
 b) It is obtained from decision table or tree.

c) It is obtained from flow graph.

d) expression of exit = 1

14. In a good state graph

- a) bugs are easy to find
 b) Sequence of inputs does not lead to initial state
 c) Bugs are more
 d) Both a and b

15. Outputs and inputs in state diagram are separated by

- a) *(asterisk)
 b) /(slash)
 c) -(eiphen)
 d) none

Fill in the blanks:

- The representation of state graph in the form of table is known as _____
- The computing in the finite state machine in the start state begins with _____
- The process of converting characters into numerical is _____
- The set of different input values is known as _____
- Impossible states increases the no of states and there by increases _____
- If every sequence of inputs from one state generates exactly the same sequence of outputs for other states, then two states are known as _____
- The bugs that occur due to dead states are known as _____
- The behavior of FSM is invariant under all _____
- The difference between FSM's & combinational machine are in terms of _____
- The implementation of combinational program is based on _____
- _____ are used to test finite state machine in every possible state.
- _____ changes the state of the machine.
- A _____ is the condition which describes the behavior of the system.
- Every input condition of state graph is specified in _____ in state tables.
- "For every unique combination of state and input, there should be only one transition specified" is the property of _____

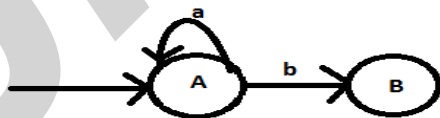
Answers:

- State table
- Input String
- Encoding
- Input Alphabet
- Test cases

6. Equivalent tests
7. Transition bugs
8. Encoding
9. Quality
10. Decision table
11. Switches and flags
12. Transition function
13. State
14. Columns
15. Good State graphs

Unit-8:

1. The problems with pictorial representation does not include
 - a) Tracking is not easy
 - b) Paths may be missed
 - c) Paths may be marked twice
 - d) Generating test cases is not difficult**
2. The false arithmetic rule for binary weight is
 - a) $1 + 1 = 1$
 - c) $1 + 0 = 0$**
 - b) $1 * 1 = 1$
 - d) $1 + 0 = 1$
3. If column of a matrix contains two or more 1's then it is
 - a) Branch node
 - b) Junction node**
 - c) loop
 - d) Both a and b
4. "If $a \subset b$ and $b \subset a$ then $a = b$ " the relation is known as
 - a) Symmetric
 - b) Anti symmetric**
 - c) Reflexive
 - d) Transitive
5. Equivalence relation need not satisfy
 - a) Reflexive**
 - c) Anti Symmetric
 - b) Symmetric
 - d) Transitive
6. The building tools of graph matrices does not include
 - a) Matrix operations
 - b) Matrix representation**
 - c) Node reduction optimization
 - d) None
7. Partial ordering relations need not satisfy
 - a) Reflexive
 - d) Transitive**
 - c) Anti Symmetric
 - b) Symmetric
8. In node reduction optimization, when nodes of degree 3 are removed then total links are
 - a) Reduced 1**
 - b) Not changed
 - c) Increased
 - d) Reduced by 2
9. The representation of matrix in an array is convenient because
 - a) It occupies large space
 - b) It takes long processing times
 - c) It takes more null values for string array
 - d) All the above**
10. The square of the matrix whose entry are a_s is obtained by replacing every entry with
 - a) $\sum_{k=1}^n a_{ik} a_{kj}$**
 - b) $\sum_{k=1}^n a_{ik} a_{ji}$
 - c) $\sum_{k=1}^n a_{ii} a_{jk}$
 - d) none
- 11.



Is the above graph in matrix form is

a)

a	b

b)

a	
	b

c)

d)

12. No
a) size
No. of

	a
b	

of nodes in state graph is equal to
of matrix
rows X No of columns

a	
b	

b)

c) Both a and b

d) None

13. In graph matrices, if the link does not exist between nodes, it is represented by

a) 0

b) 1

c) ∞

d) null

14. The cyclomatic for the given matrix is

		1		
	1		1	
	1			1
	1	1		1

a) 0

c) 3

b) 2

d) 5

15. The advantages of node reduction algorithm in matrices compared to that in graph is

a) It is more methodical in graphs

b) Errors would be less

c) Redrawing of graphs is not necessary

d) All of the above

Fill in the blanks:

- Every _____ in the graph is represented as single column in matrix.
- If the element of principal diagonal is '1' then the corresponding node is _____
- A relation which satisfies reflexive, transitive and anti symmetric properties is _____
- The matrix needed to be identified for partition algorithm is _____
- In software, the matrix is represented using _____
- The node of _____ degree should be removed first in the process of node reduction optimization.
- The automated tools used for regression testing are _____
- The set of all paths between nodes is expressed in matrix operation as _____
- The matrix for which $A^2 = A$ is known as _____
- The number of outlinks a node has can be considered as _____
- To find the path from any node to any other node _____ algorithm is used.
- Representation of a matrix with binary weights is known as _____
- Two kinds of matrix are _____.
- The number of decisions at the particular node can be calculated by formula _____ in state table.
- _____ of a matrix is obtained by adding all decision values and then adding 1 to it.

Answers:

- Node
- Loop Node
- Partial ordering relation
- $(P+1)^n \# (P+1)^{nl}$
- Linked lists

6. Lowest or least
7. Winrunner and Jmeter
8. $\sum_{i=1}^n (A^i) = A + A^2 + A^3 + \dots + A^n$
9. Idempotent
10. Out Degree
11. Matrix Determinant
12. Connection Matrix
13. Relational matrix and connection matrix
14. n-1
15. Cyclomatic complexity

20. Tutorial problems

- Discussion about win runner
- Discuss about selenium tool
- Discuss about test link and test director
- Discuss about Bugzilla

21. Known Gaps

No Practical's for Test Management Tools (Test link and Test Director), we discussed them as a case study.

Content Beyond Syllabus

For Lab we have discussed about QTP testing tool and made students to write Test Cases on Library management and Banking Applications and Test design for ATM system

22. Discussion topics

1. Purpose of testing
2. Goals of testing
3. Productivity and quality in software
4. Testing vs debugging
5. Modularity vs Efficiency
6. Role of models
7. Decision tables
8. Testing and design Style
9. Transaction flows
10. Data flow testing decision tables
11. KV charts
12. State graphs differentiation
13. Inputs and transitions
14. Software implementation
15. Integration testing

23. References, Journals, websites and E-links

Websites

1. Software Testing Training

Adwww.talentedge.in/Software-Testing

2. Software Test Techniques

www.webcrawler.com/

- 3 Winrunner Testing Tool

- 4 Performance Testing With JMeter 2.9

e books :

- 1 *Software Testing Tools: Covering Win Runner, Silk Test, ...*

Dr. K.V.K.K. Prasad - 2004 -

2. *Software Testing: Testing*

Software Gerald D. Everett, Raymond McLeod, Jr. – 2007

3. *Introducing Software Testing* Tamres

Subject Experts :

1. Prof. Shashi Kelkar, Department Of Computer Science & Engineering ,IIT Bombay .
2. Prof. N.L. Sarda, Prof. Umesh ,IIT Bombay .

List of relevant journals :

1. Journal of Emerging Trends in Computing and Information Sciences
2. IEEE Transactions on Software Engineering (TSE, Impact 3.569)
3. ACM Transactions on Software Engineering and Methodology (TOSEM, Impact 3.958)
4. International Journal on Software Engineering and Knowledge Engineering
5. Software Testing, Verification and Reliability