SOFTWARE TESTING METHODOLOGY(AIT008)

IV B. Tech I semester (Autonomous IARE R-16)

BY

Ms. M Geetha Yadav
Assistant Professor

DEPARTMENT OF INFORMATION TECHNOLOGY
INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
DUNDIGAL, HYDERABAD - 500 043

UNIT - I
INTRODUCTION

Course learning outcomes

CLO’s	Course Learning outcomes
CLO1	Explain the importance of testing and purpose of testing.
CLO2	Illustrate different and compare dichotomies of testing.
CLO3	Demonstrate the model for testing and different testing levels and role of models.
CLO4	Describe the consequences and taxonomy of bugs and different bugs in project environment.
CLO5	Illustrate the concepts of path testing and predicate loops and path sensitization.
CLO6	Explain Path instrumentation and their applications and link markers.

Introduction

Testing:

- It is a process used to estimate the productivity and quality of software. Testing techniques provide systematic guidance for designing tests that

  1) Exercise the internal logic of s/w components

  2)Exercise the input and output domains of program to uncover errors in program function, behavior and performance

Testing

What is Testing?

- Related terms : SQA, QC, Verification, Validation
- Verification of functionality for conformation against given specifications By execution of the software application.
- A Test
- Passes: Functionality OK.
- Fails: Application functionality NOK
- Bug/Defect/Fault: Deviation from expected functionality.

Purpose of Testing

- Testing is performed for following purposes
- For improving and assuring software quality
- For verification and validation
- For estimating reliability

Productivity and Quality in Software

- It is used to monitor the s/w engineering process and methods (audits) for ensuring its quality.
- There is a trade off between quality assurance costs and manufacturing costs. If insufficient effort is spent in quality assurance, the reject rate will be high and so will the net cost.
- If inspection is so good that all faults are caught as they occur, inspection costs will dominate, and again net cost will suffer.

Purpose of Testing

Phase 1: Testing is to show that the software works

- A failed test shows software does not work, even if many tests pass.
- Objective not achievable.

Phase 2: Software does not work

- One failed test proves that.
- Tests are to be redesigned to test corrected software.
- But we do not know when to stop testing.

Phase 3: Test for Risk Reduction

- We apply principles of statistical quality control.
- Our perception of the software quality changes – when a test passes/fails.
- Consequently, perception of product Risk reduces.
- Release the product when the Risk is under a predetermined limit.

Purpose of Testing

- 5 Phases in tester's thinking
- Phase 4: A state of mind regarding "What testing can do & cannot do. What makes software testable".
    - Applying this knowledge reduces amount of testing.
    - Testable software reduces effort
    - Testable software has less bugs than the code hard to test
- Cumulative goal of all these phases:
    - Cumulative and complementary. One leads to the other.
    - Phase2 tests alone will not show software works
    - Use of statistical methods to test design to achieve good testing at acceptable risks.
    - Most testable software must be debugged, must work, must be hard to break.

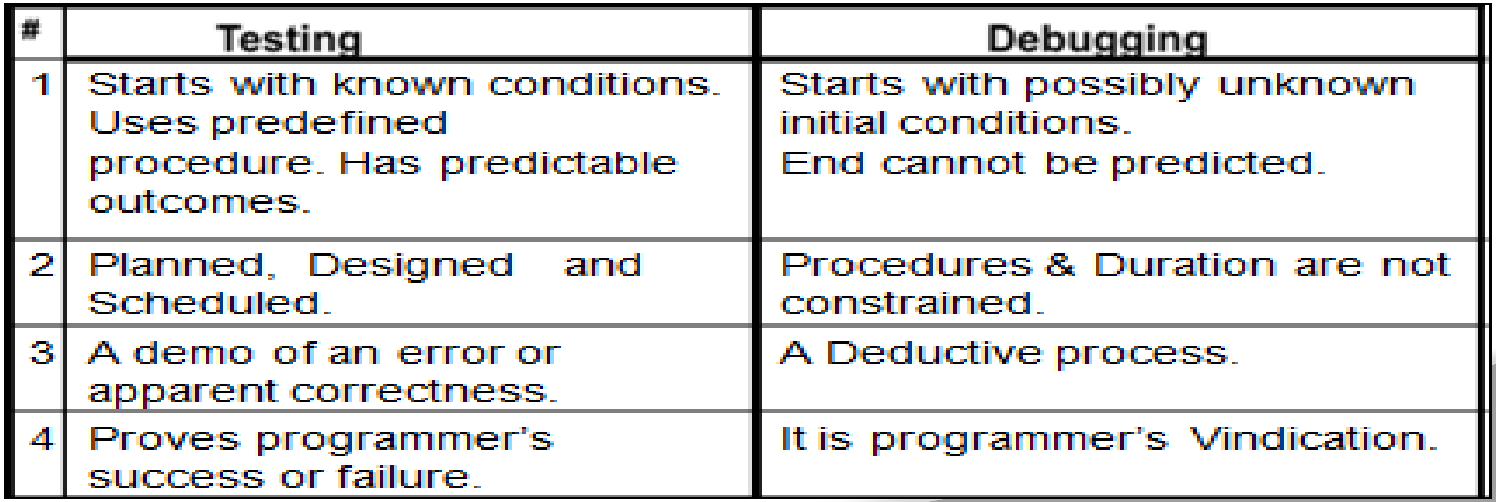
Purpose of Testing

5. TESTING AND INSPECTION

- Inspection is also called static testing.
- Methods and Purposes of testing and inspection are different, but the objective is to catch & prevent different kinds of bugs.
- To prevent and catch most of the bugs, we must
  - Review
  - Inspect &
  - Read the code

Dichotomies

- Testing Vs Debugging
  - Testing is to find bugs.
  - Debugging is to find the cause or misconception leading to the bug.
  - Their roles are confused to be the same. But, there are differences in goals, methods and psychology applied to these



#	Testing	Debugging
1	Starts with known conditions. Uses predefined procedure. Has predictable outcomes.	Starts with possibly unknown initial conditions. End cannot be predicted.
2	Planned, Designed and Scheduled.	Procedures & Duration are not constrained.
3	A demo of an error or apparent correctness.	A Deductive process.
4	Proves programmer's success or failure.	It is programmer's Vindication.

Dichotomies

S.No	Testing	Debugging
6	Much of testing can be without design knowledge.	Impossible without a detailed design knowledge.
7	Can be done by outsider to the development team.	Must be done by an insider (development team).
8	A theory establishes what testing can do or cannot do.	There are only Rudimentary Results (on how much can be done. Time, effort, how etc. depends on human ability).
9	Test execution and design can be automated.	Debugging - Automation is a dream.

Dichotomies

- For a given model of programs, Structural tests may be done first and later the Functional,
  Or vice-versa. Choice depends on which seems to be the natural choice.
- Both are useful, have limitations and target different kind of bugs. Functional tests can detect all bugs in principle, but would take infinite amount of time. Structural tests are inherently finite, but cannot detect all bugs.
- The Art of Testing is how much allocation % for structural vs how much % for functional.

Dichotomies

	Programmer/Designer	Tester
1	Tests designed by designers are more oriented towards structural testing and are limited to its limitations.	With knowledge about internal test design, the tester can eliminate useless tests, optimize & do an efficient test design.
2	Likely to be biased.	Tests designed by independent testers are bias- free.
3	Tries to do the job in simplest & cleanest way, trying to reduce the complexity.	Tester needs to suspicious, uncompromising, hostile and obsessed with destroying program.

Dichotomies

4.Modularity vs Efficiency

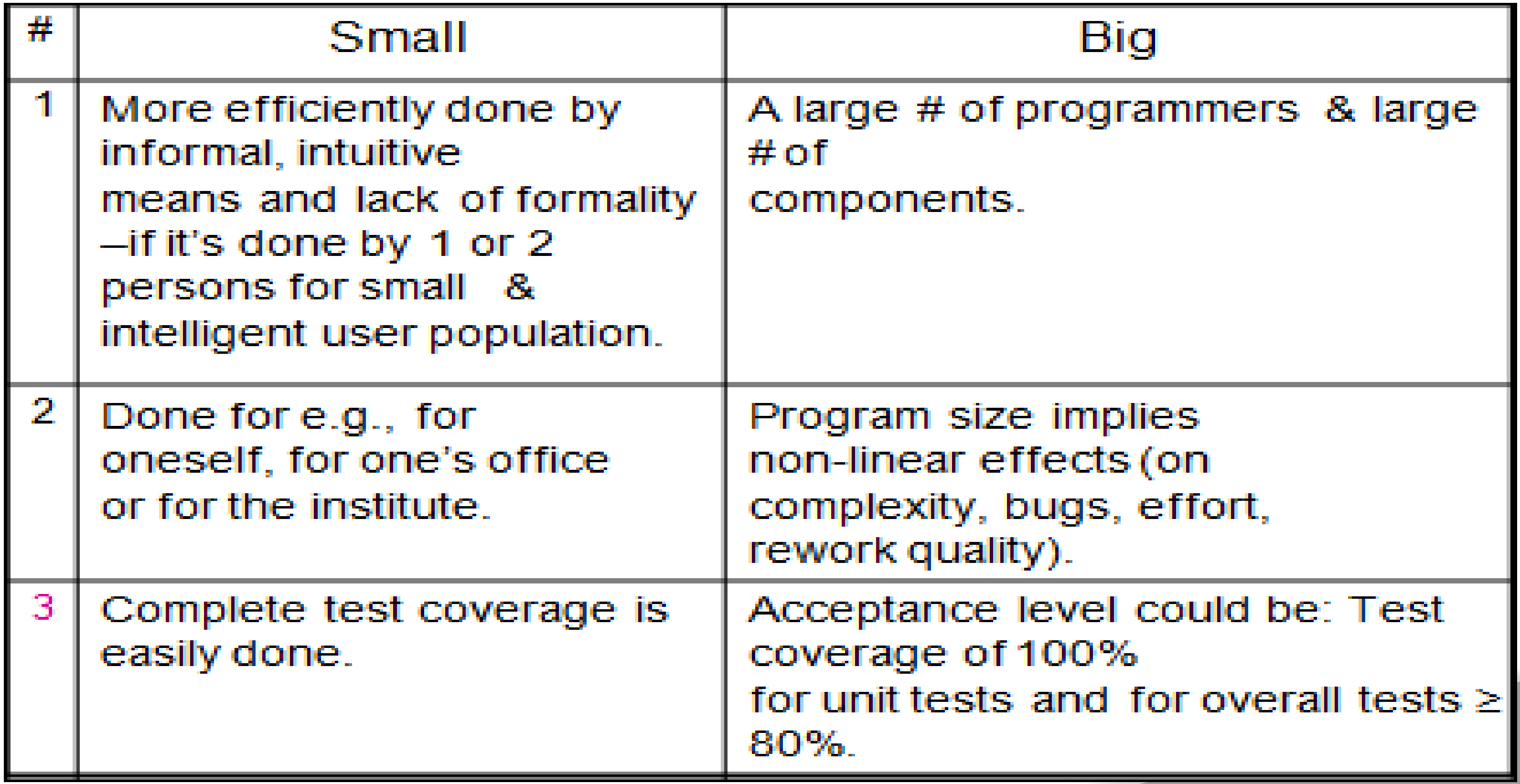
- System and test design can both be modular.
- A module implies a size, an internal structure and an interface, Or, in other words.
- A module (well defined discrete component of a system) consists of internal complexity & interface complexity and has a size.

Dichotomies

	Modularity	Efficiency
1	Smaller the component easier to understand.	Implies more number of components & hence more # of interfaces increase complexity & reduce efficiency (=> more bugs likely)
2	Small components/modules are repeatable independently with less rework (to check if a bug is fixed).	Higher efficiency at module level, when a bug occurs with small components.
3	Microscopic test cases need individual setups with data, systems & the software. Hence can have bugs.	More # of test cases implies higher possibility of bugs in test cases. Implies more rework and hence less efficiency with microscopic test cases
4	Easier to design large modules & smaller interfaces at a higher level.	Less complex & efficient. (Design may not be enough to understand and implement. It may have to be broken down to implementation level.)

Dichotomies

- Programming in Small Vs Programming in Big
  - Impact on the development environment due to the volume of customer requirements.



#	Small	Big
1	More efficiently done by informal, intuitive means and lack of formality –if it's done by 1 or 2 persons for small & intelligent user population.	A large # of programmers & large # of components.
2	Done for e.g., for oneself, for one's office or for the institute.	Program size implies non-linear effects (on complexity, bugs, effort, rework quality).
3	Complete test coverage is easily done.	Acceptance level could be: Test coverage of 100% for unit tests and for overall tests ≥ 80%.

A Model for Testing

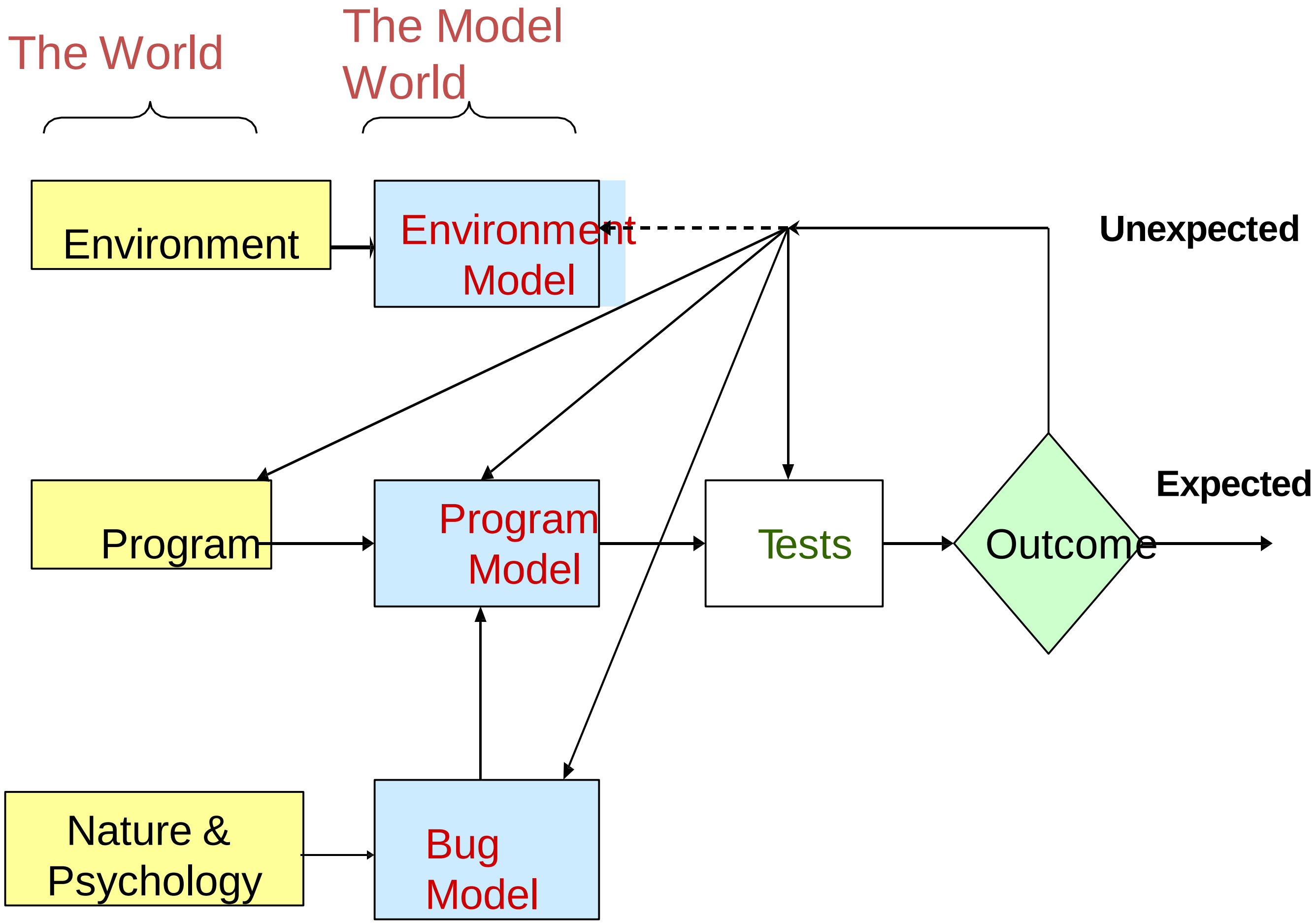
Acceptance test Application is accepted after a formal acceptance test. At first it’s the customer’s & then the software design team’s responsibility.

Personnel: The technical staff comprises of :A combination of experienced professionals & junior programmers (1– 3 yrs) with varying degrees of knowledge of the application.

Standards:

- Programming, test and interface standard (documented and followed).
- A centralized standards data base is developed & administrated

A Model for Testing



Consequences of Bugs

- Very serious

  System does another transaction instead of requested.
- Extreme

  Frequent & Arbitrary - not sporadic & unusual.
- Intolerable

  Long term unrecoverable corruption of the Data base.(not easily discovered and may lead to system down.)
- Catastrophic

  System fails and shuts down.
- Infectious

  Corrupts other systems, even when it may not fail.

Consequences of Bugs

Assignment of severity:

- Assign flexible & relative rather than absolute values to the bug (types).
- Number of bugs and their severity are factors in determining the quality quantitatively.
- Organizations design & use quantitative, quality metrics based on the above.
- Parts are weighted depending on environment, application, culture, correction cost, current SDLC phase & other factors.

Consequences of Bugs

When to stop Testing

- List all nightmares in terms of the symptoms & reactions of the user to their consequences.
- Convert the consequences of into a cost. There could be rework cost. (but if the scope extends to the public, there could be the cost of lawsuits, lost business, nuclear reactor meltdowns.)
- Order these from the costliest to the cheapest. Discard those with which you can live with.
- Based on experience, measured data, intuition, and published statistics postulate the kind of bugs causing each symptom. This is called 'bug design process'. A bug type can cause multiple symptoms

Consequences of Bugs

- Rank the bug types in order of decreasing importance.
- Design tests & QA inspection process with most effective against the most important bugs.
- If a test is passed or when correction is done for a failed test, some nightmares disappear.
- Stop testing when probability (importance & cost) proves to be inconsequential.
- This procedure could be implemented formally in SDLC.

Taxonomy of Bugs

Importance of Bugs- statistical quantification of impact

Consequences of Bugs - causes, nightmares, to stop testing

We will now see the:

Taxonomy of Bugs - along with some remedies

- In order to be able to create an organization's own Bug Importance Model for the sake of controlling associated cost.

Why Taxonomy ?

- To study the consequences, nightmares, probability, importance, impact and the methods of prevention and correction.

Taxonomy of Bugs

- Adopt known taxonomy to use it as a statistical framework on which your testing strategy is based.
- 6 main categories with sub-categories..

1)Requirements, Features, Functionality Bugs	24.3%
2)Structural Bugs	25.2%
3)Data Bugs	22.3%
4)Coding Bugs	9.0%
5)Interface, Integration and System Bugs	10.7%
6)Testing & Test Design Bugs	2.8 %

Taxonomy of Bugs

Requirements, Features, Functionality Bugs

- Incompleteness ,Requirements & Specs.
- ambiguous or self-contradictory
- Analyst's assumptions not known to the designer
- Some thing may miss when specs change
- These are expensive: introduced early in SDLC and removed at the last

I. Feature Bugs

- Specification problems create feature bugs
- Wrong feature bug has design implications
- Missing feature is easy to detect & correct

Taxonomy of Bugs and remedies

III. Feature Interaction Bugs

- Arise due to unpredictable interactions between feature groups or individual features. The earlier removed the better as these are costly if detected at the end.
- Examples: call forwarding & call waiting. Federal, state & local tax laws.
- No magic remedy. Explicitly state & test important combinations.

Remedies

- Use high level formal specification languages to eliminate human-to-human communication
- It's only a short term support & not a long term solution.

Taxonomy of Bugs

2. Structural Bugs

we look at the 5 types, their causes and remedies.

I. Control & Sequence bugs
II. Logic Bugs
III. Processing bugs
IV. Initialization bugs
V. Data flow bugs & anomalies

Control & Sequence Bugs:

- Paths left out, unreachable code, spaghetti code, and pachinko code.
- Improper nesting of loops, Incorrect loop-termination or look-back, ill-conceived switches.
- Missing process steps, duplicated or unnecessary processing, rampaging GOTOs.

- Unit, structural, path, & functional testing

Taxonomy of Bugs

Structural Bugs

- Logic Bugs
  - Misunderstanding of the semantics of the control structures & logic operators
  - Improper layout of cases, including impossible & ignoring necessary cases,
  - Using a look-alike operator, improper simplification, confusing Ex-OR with inclusive OR.
  - Deeply nested conditional statements & using many logical operations in 1 stmt.

Taxonomy of Bugs

Initialization Bugs

- Forgetting to initialize work space, registers, or data areas.
- Wrong initial value of a loop control parameter.
- Accepting a parameter without a validation check.
- Initialize to wrong data type or format.
- Very common.

Dataflow Bugs & Anomalies

- Run into an un-initialized variable.
- Not storing modified data.
- Re-initialization without an intermediate use.
- Detected mainly by execution (testing).

Taxonomy of Bugs

Dynamic data	Static data
Transitory. Difficult to catch.	Fixed in form & content.
Due to an error in a shared storage object initialization.	Appear in source code or data base, directly or indirectly
Due to unclean / leftover garbage in a shared resource.	Software to produce object code creates a static data table – bugs possible
Examples	**Examples**
Generic & shared variable	Telecom system software: generic parameters, a generic large program & site adapter program to set parameter values, build data declarations etc.
Shared data structure	Postprocessor : to install software packages. Data is initialized at run time – with configuration handled by tables.
Prevention Data validation, unit testing	**Prevention** Compile time processing Source language features

Taxonomy of Bugs

- Coding errors create other kinds of bugs.
- Syntax errors are removed when compiler checks syntax.
- Coding errors typographical, misunderstanding of operators or statements or could be just arbitrary.
- Documentation Bugs
- Erroneous comments could lead to incorrect maintenance.
- Testing techniques cannot eliminate documentation bugs.

Control Flow Graphs and Path Testing

Path Testing : A family of structural test techniques based on judiciously selecting a set of test paths through the programs.

Goal: Pick enough paths to assure that every source statement is executed at least once.

- It is a measure of thoroughness of code coverage.
- It is used most for unit testing on new software.
- Its effectiveness reduces as the software size increases.
- We use Path testing techniques indirectly.
- Path testing concepts are used in and along with other testing techniques
- Code Coverage: During unit testing: # stmts executed at least once / total # stmts

Control Flow Graphs and Path Testing

Assumptions:

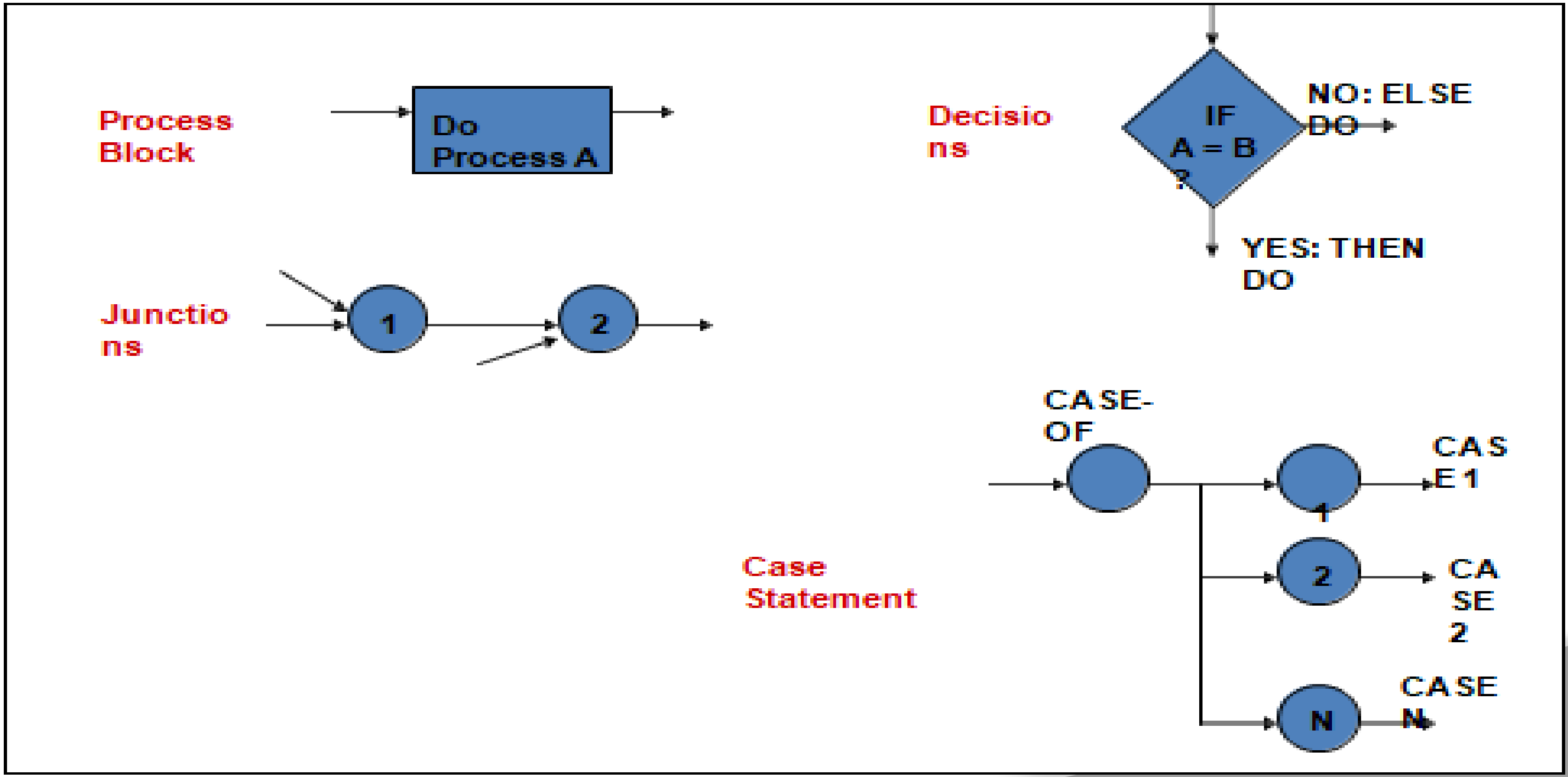
- Software takes a different path than intended due to some error.
- Specifications are correct and achievable.
- Processing bugs are only in control flow statements
- Data definition & access are correct

Observations

- Structured programming languages need less of path testing.
- Assembly language, Cobol, Fortran, Basic & similar languages make path testing necessary.

Control Flow Graphs and Path Testing

- A simplified, abstract, and graphical representation of a program's control structure using process blocks, decisions and junctions.



Control Flow Graphs and Path Testing

Process Block:

- A sequence of program statements uninterrupted by decisions or junctions with a single entry and single exit.

Junction:

- A point in the program where control flow can merge (into a node of the graph)
- Examples: target of GOTO, Jump, Continue

Decisions:

- A program point at which the control flow can diverge (based on evaluation of a condition).
- Examples: IF stmt. Conditional branch and Jump instruction.

Case Statements:

- A Multi-way branch or decision.
- For test design, Case statement and decision are similar.

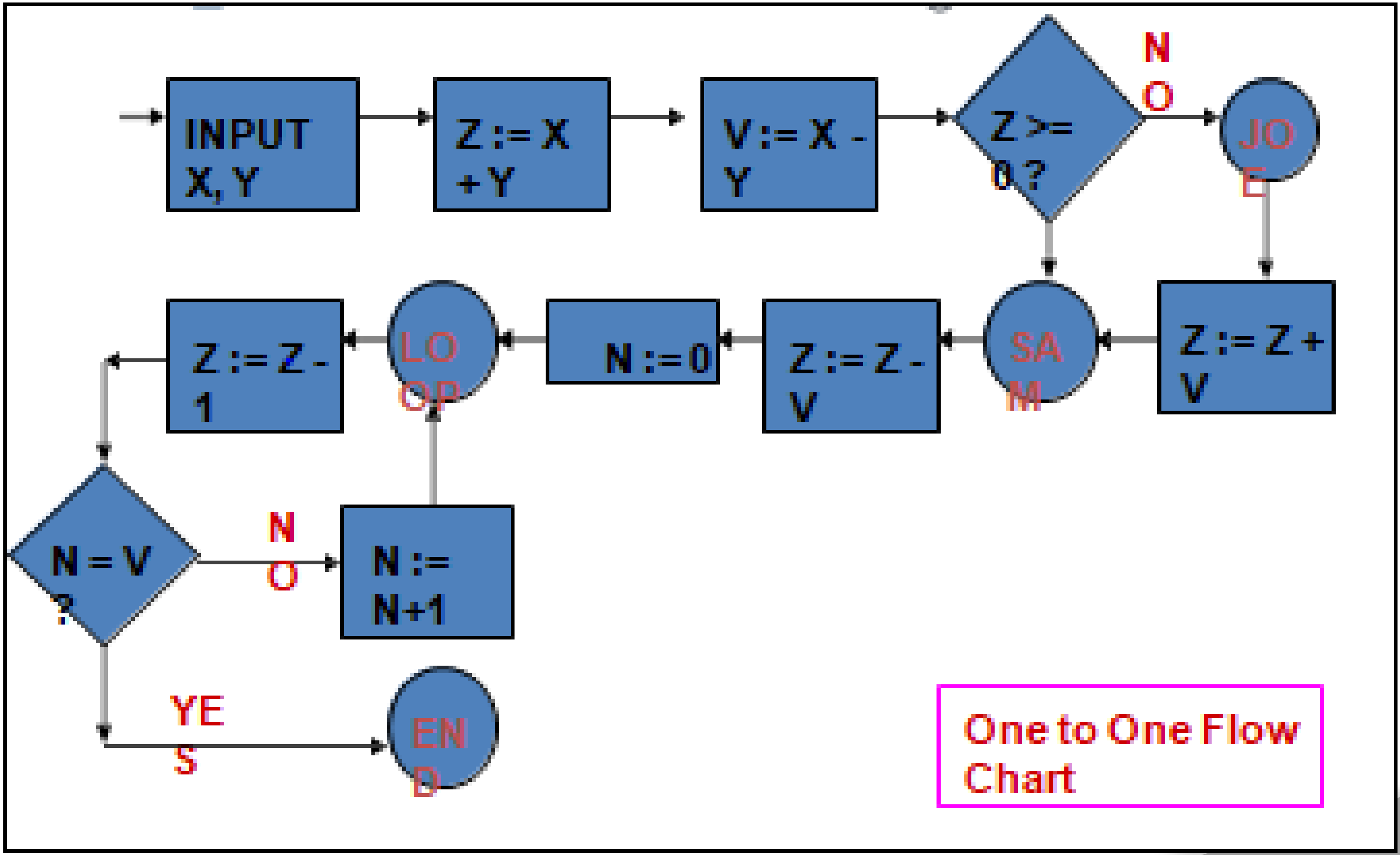
Control Flow Graph Vs Flow Charts

Control Flow Graph	Flow Chart
Compact representation of the program	Usually a multi-page description
Focuses on Inputs, Outputs, and the control flow into and out of the block.	Focuses on the process steps inside
Inside details of a process block are not shown	Every part of the process block are drawn

Control Flow Graphs and Path Testing

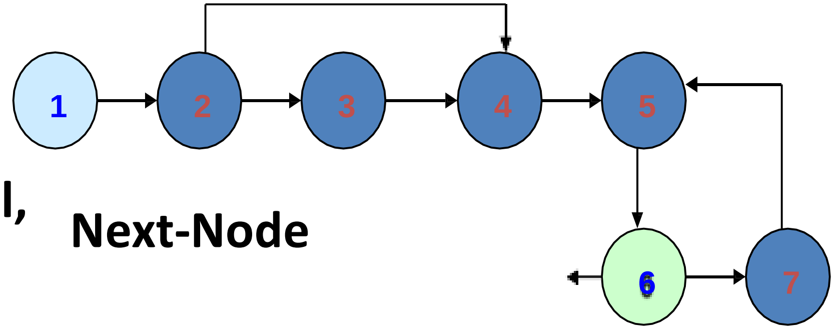
- One statement to one element translation to get a Classical Flow chart
- Add additional labels as needed
- Merge process steps
- A process box is implied on every junction and decision
- Remove External Labels
- Represent the contents of elements by numbers.
- We have now Nodes and Links

One to one flow chart



Control flow graph

Linked List Notation of a Control Flow Graph



Node	Processing, label, Decision	Next-Node
1	(BEGIN; INPUT X, Y; Z := X+Y ; V := X-Y)	: 2
2	(Z >= 0 ?)	: 4 (TRUE) : 3 (FALSE)
3	(JOE: Z := Z + V)	: 4
4	(SAM: Z := Z – V; N := 0)	: 5
5	(LOOP; Z := Z -1)	: 6
6	(N = V ?)	: 7 (FALSE) : END (TRUE)
7	(N := N + 1)	: 5

Control Flow Graphs and Path Testing

Path is a sequence of statements starting at an entry, junction or decision and ending at another, or possibly the same junction or decision or an exit point.

- Link is a single process (*block*) in between two nodes.
- Node is a junction or decision.
- Segment is a sequence of links. A path consists of many segments.
- Length of a path is measured by # of links in the path or # of nodes traversed

Control Flow Graphs and Path Testing

- Name of a path is the set of the names of the nodes along the path.
- Path-Testing Path is an “entry to exit” path through a processing block.
- Path segment is a succession of consecutive links that belongs to the same path.

Entry / Exit for a routines, process blocks and nodes.

Single entry and single exit routines are preferable.

Called well-formed routines.

Formal basis for testing exists.

Tools could generate test cases.

Test Strategy for Multi-entry / exit routines

Test Strategy for Multi-entry / exit routines

1. Get rid of them.
2. Control those you cannot get rid of.
3. Convert to single entry / exit routines.
4. Do unit testing by treating each entry/exit combination as if it were a completely different routine.
5. Recognize that integration testing is heavier
6. Understand the strategies & assumptions in the automatic test generators and confirm that they do (or do not) work for multi-entry/multi exit routines.

Control Flow Graphs and Path Testing

Fundamental Path Selection Criteria

A minimal set of paths to be able to do complete testing.

- Each pass through a routine from entry to exit, as one traces through it, is a potential path.
- The above includes the tracing of 1….n times tracing of an interactive block each separately.
- Note: A bug could make a mandatory path not executable or could create new paths not related to processing.

Complete Path Testing prescriptions:

1. Exercise every path from entry to exit.
2. Exercise every statement or instruction at least once.
3. Exercise every branch and case statement in each direction, at least once.

Control Flow Graphs and Path Testing

Path Testing Criteria :

- Path Testing (P⍰):Execute all possible control flow paths thru the program; but typically restricted to entry-exit paths. Implies 100% path coverage. Impossible to achieve.
- Statement Testing (P_1) :Execute all statements in the program at least once under the some test. 100% statement coverage => 100% node coverage. Denoted by C1.
- Branch Testing (P_2) :Execute enough tests to assure that every branch alternative has been exercised at least once under some test. Denoted by C2
- Objective: 100% branch coverage and 100% Link coverage. For well structured software, branch testing & coverage include statement coverage

Control Flow Graphs and Path Testing

Revised path selection Rules

1. Pick the simplest and functionally sensible entry/exit path.
2. Pick additional paths as small variations from previous paths. (pick those with no loops, shorter paths, simple and meaningful).
3. Pick additional paths but without an obvious functional meaning (only to achieve C1+C2 coverage).
4. Be comfortable with the chosen paths. play hunches, use intuition to achieve C1+C2.
5. Don’t follow rules slavishly – except for coverage.

Control Flow Graphs and Path Testing

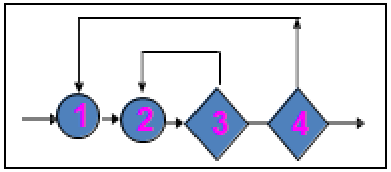
4. Testing of Paths involving loops:

   Bugs in iterative statements apparently are not discovered by C1+C2.But by testing at the boundaries of loop variable.

   - Types of Iterative statements:
     1. Single loop statement.
     2. Nested loops.
     3. Concatenated Loops.
     4. Horrible Loops

Control Flow Graphs and Path Testing

2.Testing a Nested Loop Statement



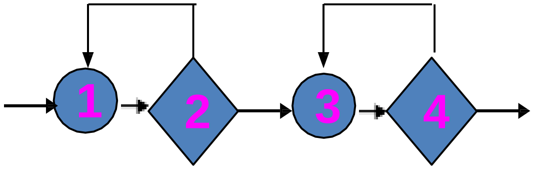
Multiplying # of tests for each nested loop => very large # of tests

A test selection technique:

1. Start at the inner-most loop. Set all outer-loops to Min iteration parameter values: **V_{min}.**
2. Test the **V_{min}, V_{min} + 1, typical V, V_{max} - 1, V_{max}** for the inner-most loop. Hold the outer- loops to **$V_{min.}$** Expand tests are required for out-of-range & excluded values.

Control Flow Graphs and Path Testing

3. Testing Concatenated Loop Statements



- Two loops are concatenated if it’s possible to reach one after exiting the other while still on the path from entrance to exit.
- If these are independent of each other, treat them as independent loops.
- If their iteration values are inter-dependent & these are same path, treat these like a nested loop.
- Processing times are additive.

Control Flow Graphs and Path Testing

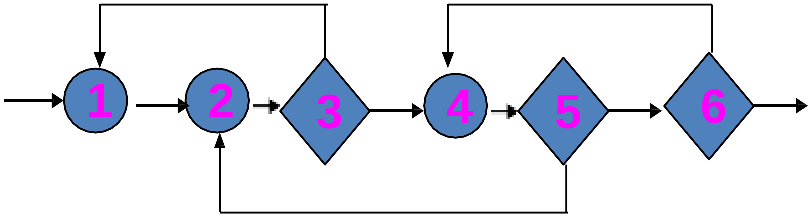
- If you have done with outer most loop, Go To step 5. Else, move out one loop and do step 2 with all other loops set to typical values.
- Do the five cases for all loops in the nest simultaneously.

– Assignment: check # test cases = 12 for 16 for 3, 19 for 4.

– nesting = 2

Control Flow Graphs and Path Testing

4. Testing Horrible Loops



- Avoid these.
- Even after applying some techniques of paths, resulting test cases not definitive.
- Too many test cases.
- Thinking required to check end points etc. is unique for each program.
- Jumps in & out of loops and intersecting loops etc, makes test case selection an ugly task.
- etc. etc.

Control Flow Graphs and Path Testing

Loop Testing Times

- Longer testing time for all loops if all the extreme cases are to be tested.
- Unreasonably long test execution times indicate bugs in the s/w or specs.

  Case: Testing nested loops with combination of extreme values leads to long test times.

  - Show that it's due to incorrect specs and fix the specs.
  - Prove that combined extreme cases cannot occur in the real world. Cut-off those tests.
  - Put in limits and checks to prevent the combined extreme cases.

Control Flow Graphs and Path Testing

Effectiveness of Path Testing

- Path testing (with mainly P1 & P2) catches ~65% of Unit Test Bugs ie., ~35% of all bugs.
- More effective for unstructured than structured software.
- Limitations
  - Path testing may not do expected coverage if bugs occur.
  - Path testing may not reveal totally wrong or missing functions.
  - Unit-level path testing may not catch interface errors among routines.
  - Data base and data flow errors may not be caught.
  - Unit-level path testing cannot reveal bugs in a routine due to another.

Control Flow Graphs and Path Testing

- Effectiveness of Path Testing
  - A lot of work
    - Creating flow graph, selecting paths for coverage, finding input data values to force these paths, setting up loop cases & combinations.
    - Careful, systematic, test design will catch as many bugs as the act of testing.
    - Test design process at all levels at least as effective at catching bugs as is running the test designed by that process.

Predicates, Predicate Expressions

- Path: A sequence of process links (& nodes)
- Predicate
- Compound Predicate: Two or more predicates combined with AND, OR etc.
- Path Predicate:
  - Every path corresponds to a succession of True/False values for the predicates traversed on that path.
  - A predicate associated with a path.
    - " X > 0 is True " AND "W is either negative or equal to 122" is True Multi-valued Logic / Multi-way branching.

Control Flow Graphs and Path Testing

- The symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called predicate interpretation.
- An input vector is a set of inputs to a routine arranged as a one dimensional array.

Example:

- INPUT X, Y
- ON X GOTO A, B, C
- A: Z := 7 @ GOTO H
- B: Z := -7 @ GOTO H
- C: Z := 0 @ GOTO H
- H: DO SOMETHING
- K: IF X + Z > 0 GOTO GOOD ELSE GOTO BETTER

```
INPUT X
IF X < 0 THEN
Y:= 2
ELSE Y := 1
IF X + Y*Y > 0 THEN ...IF -7 > 3.
```

Path Testing

Process Dependency

- An input variable is independent of the processing if its value does not change as a result of processing.
- An input variable is process dependent if its value changes as a result of processing.
- A predicate is process dependent if its truth value can change as a result of processing.
- A predicate is process independent if its truth value does not change as a result of processing.
- Process dependence of a predicate doesn’t follow from process dependence of variables

Correlation

Correlation

- Two input variables are correlated if every combination of their values cannot be specified independently.
- Variables whose values can be specified independently without restriction are uncorrelated.
- A pair of predicates whose outcomes depend on one or more variables in common are
  correlated predicates.
- Every path through a routine is achievable only if all predicates in that routine are
  uncorrelated.
- If a routine has a loop, then at least one decision's predicate must be process dependent.

Path Predicate Expression

Path Predicate Expression

- Every selected path leads to an associated boolean expression, called the path predicate expression, which characterizes the input values (if any) that will cause that path to be traversed.
- Select an entry/exit path. Write down un-interpreted predicates for the decisions along the path. If there are iterations, note also the value of loop-control variable for that pass. Converting these into predicates that contain only input variables, we get a set of boolean expressions called path predicate expression.

Control Flow Graphs and Path Testing

Predicate Coverage:

- Look at examples & possibility of bugs: A B C D A + B + C + D
  - Due to semantics of the evaluation of logic expressions in the languages, the entire expression may not be always evaluated.
  - A bug may not be detected.
  - A wrong path may be taken if there is a bug.
  - Realize that on our achieving C2, the program could still hide some control flow bugs.

Predicate coverage

- Predicate coverage:
  - If all possible combinations of truth values corresponding to selected path have been explored under some test, we say predicate coverage has been achieved.
  - Stronger than branch coverage.
  - If all possible combinations of all predicates under all interpretations are covered, we
  - have the equivalent of total path testing

Achievable Paths

1. Objective is to select & test just enough paths to achieve a satisfactory notion of test completeness such as C1 + C2.
2. Extract the program's control flow graph & select a set of tentative covering paths.
3. For a path in that set, interpret the predicates.
4. Trace the path through, multiplying the individual compound predicates to achieve a boolean expression.
5. Multiply & obtain sum-of-products form of the path predicate expression: AD + AE + BCD + BCE
6. Each product term denotes a set of inequalities that, if solved, will yield an input vector that will drive the routine along the selected path.

Path Sensitization

It’s the act of finding a set of solutions to the path predicate expression.

In practice, for a selected path finding the required input vector is not difficult. If there is difficulty, it may be due to some bugs. Heuristic procedures:

Choose an easily sensitizable path set, & pick hard-to-sensitize paths to achieve more coverage.

1. Identify all the variables that affect the decisions. For process dependent variables, express the nature of the process dependency as an equation, function, or whatever is convenient and clear. For correlated variables, express the logical, arithmetic, or functional relation defining the correlation.

Path Sensitization

1. Identify correlated predicates and document the nature of the correlation as for variables. If the same predicate appears at more than one decision, the decisions are obviously correlated.
2. Start path selection with uncorrelated & independent predicates. If coverage is achieved, but the path had dependent predicates, something is wrong.

Path Sensitization

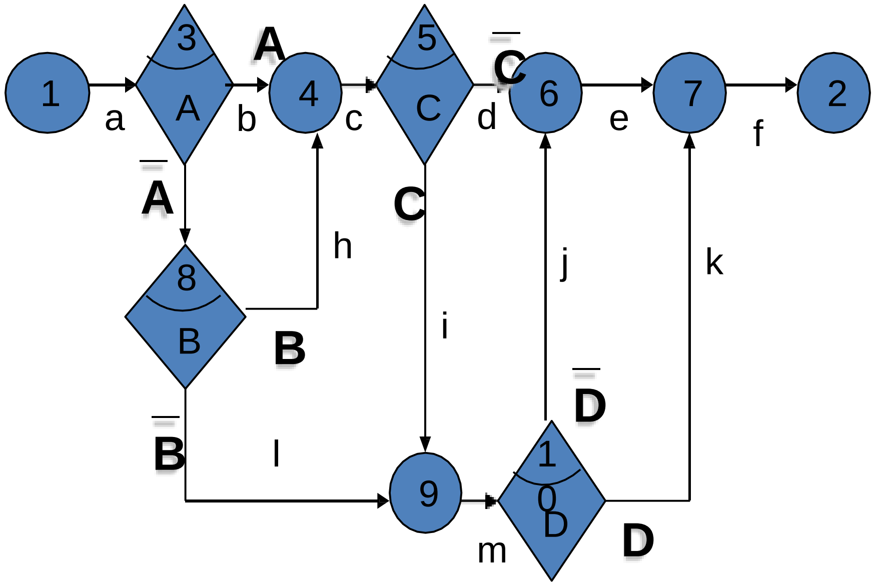
4. If the coverage is not achieved yet with independent uncorrelated predicates, extend the path set by using correlated predicates; preferably process independent (not needing interpretation)
5. If the coverage is not achieved, extend the path set by using dependent predicates (typically required to cover loops), preferably uncorrelated.
6. Last, use correlated and dependent predicates.

Path Sensitization

4. For each of the path selected above, list the corresponding input variables. If the variable is independent, list its value. For dependent variables, interpret the predicate ie., list the relation. For correlated variables, state the nature of the correlation to other variables. Determine the mechanism (relation) to express the forbidden combinations of variable values, if any.
5. Each selected path yields a set of inequalities, which must be simultaneously satisfied to force the path.

Examples for Path Sensitization

1. Simple Independent Uncorrelated Predicates



4 predicates => 16 combinations
Set of possible paths = 8

PathPredicateValues			
abcdef	**A**		**$\bar{C}$**
aghci	**$\bar{A}$**	**B**	**C**
mkf		**D**	
aglmje	**$\bar{A}$**		
f	**$\bar{B}$**		
	$\bar{D}$		

PathPredicate Values				
abcdef	**A**		**$\bar{C}$**	
abcimjef	**A**		**C**	**$\bar{D}$**
abcimkf	**A**		**C**	**D**
aghcdef	**$\bar{A}$**	**B**	**$\bar{C}$**	
aglmkf	**$\bar{A}$**	**$\bar{B}$**		**$\bar{D}$**

(obtained by the procedure for finding a covering set of paths)

A Simple case of solving inequalities.

Dependent Predicates

- Dependent Predicates

  Usually most of the processing does not affect the control flow. Use computer simulation for sensitization in a simplified way. Dependent predicates contain iterative loop statements usually.

- For Loop statements:

  Determine the value of loop control variable for a certain # of iterations, work backward to determine the value of input variables (input vector).

Dependent Predicates

The General Case

No simple procedure to solve for values of input vector for a selected path.

1. Select cases to provide coverage on the basis of functionally sensible paths. Well structured routines allow easy sensitization. Intractable paths may have a bug.
2. Tackle the path with the fewest decisions first. Select paths with least # of loops.
3. Start at the end of the path and list the predicates while tracing the path in reverse. Each predicate imposes restrictions on the subsequent (in reverse order) predicate.
4. Continue tracing along the path. Pick the broadest range of values for variables affected and consistent with values that were so far determined.

Control Flow Graphs and Path Testing

Alternately:

1. In the forward direction, list the decisions to be traversed. For each decision list the broadest range of input values.
2. Pick a path & adjust all input values. These restricted values are used for next decision.
3. Continue. Some decisions may be dependent on and/or correlated with earlier ones.
4. The path is unachievable if the input values become contradictory, or, impossible. If the path is achieved, try a new path for additional coverage.

Path Instrumentation

Output of a test: Results observed. But, there may not be any expected output for a test.

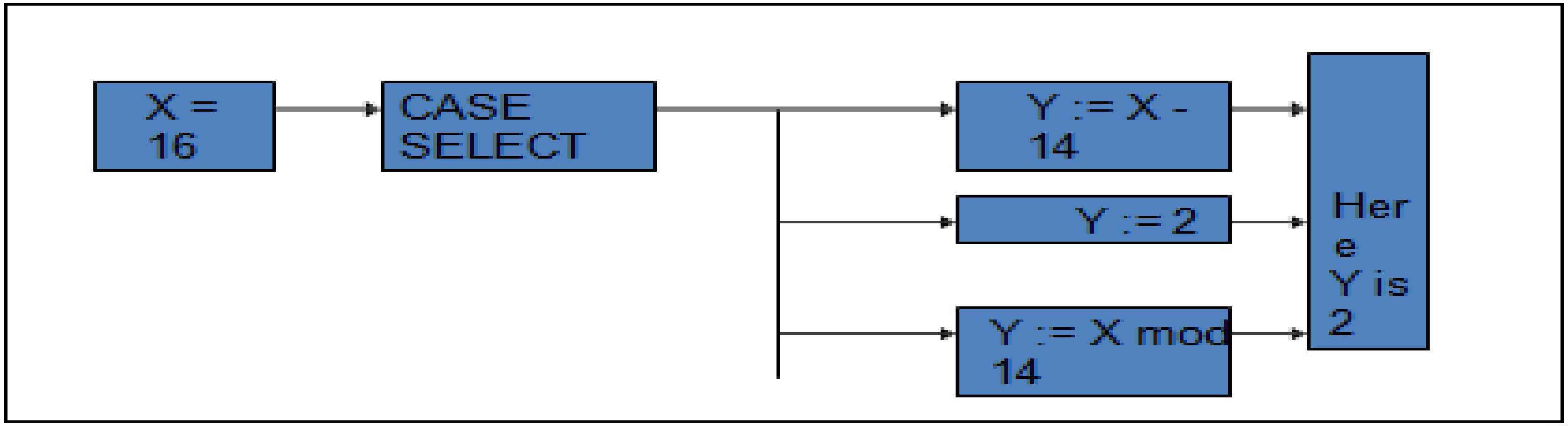
Outcome: Any change or the lack of change at the output.

Expected Outcome: Any expected change or the lack of change at the output (predicted as part of design).

Actual Outcome: Observed outcome

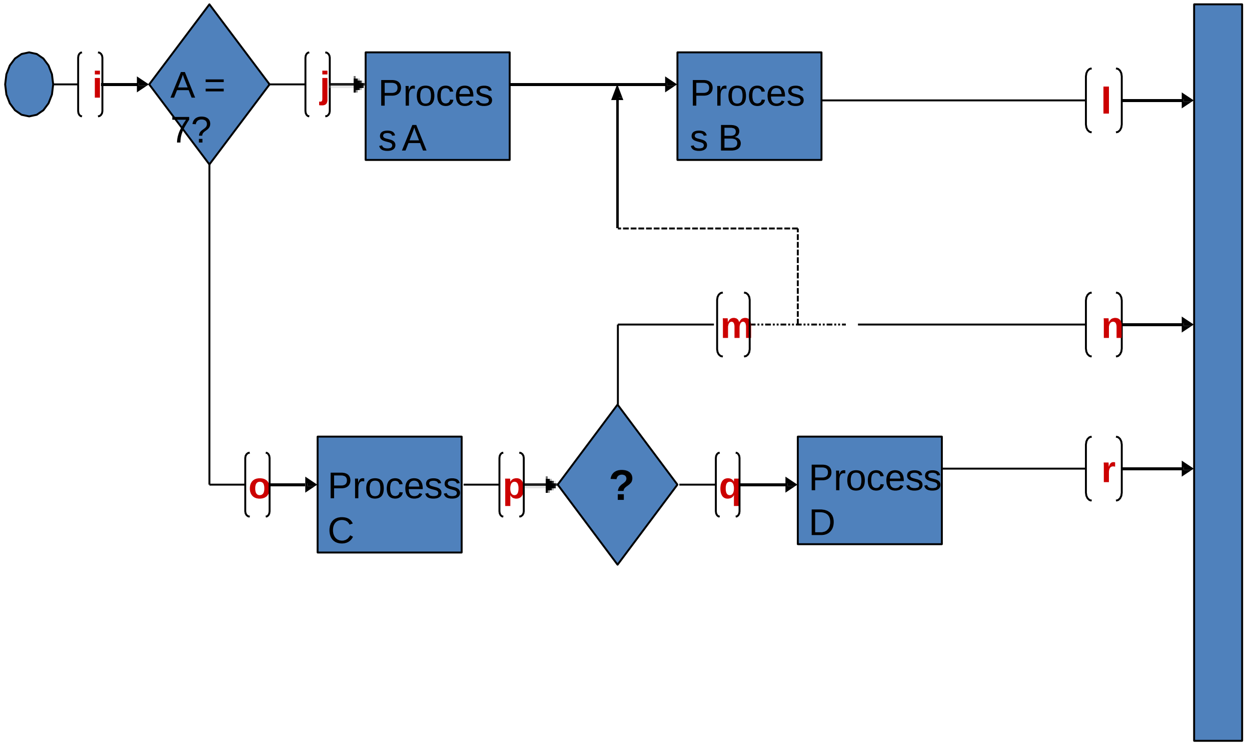
Coincidental Correctness

- Coincidental Correctness:
  - When expected & actual outcomes match,
    - Necessary conditions for test to pass are met.
    - Conditions met are probably not sufficient. (the expected outcome may be achieved due to a wrong reason)



- Path Instrumentation is what we have to do confirm that the outcome was achieved by the intended path.

Double Link Marker Instrumentation



The problem is solved. Two link markers specify the path name and both the beginning & end of the link.

Double Link Marker Instrumentation

- Less disruptive and less informative.
- Increment a link counter each time a link is traversed.Path length could confirm the intended path.
- For avoiding the same problem as with markers, use double link counters.

  Expect an even count = double the length.
- Now, put a link counter on every link.(earlier it was only between decisions)

  If there are no loops, the link counts are = 1.
- Sum the link counts over a series of tests, say, a covering set. Confirm the total link counts with the expected.
- Using double link counters avoids the same & earlier mentioned problem.

Link Counters Technique

Link Counters Technique:

Check list for the procedure:

- Do begin-link counter values equal the end-link countervalues?
- Does the input-link count of every decision equal to the sum of the link counts of the output links from that decision?
- Do the sum of the input-link counts for a junction equal the output-link count for that junction?
- Do the total counts match the values you predicted when you designed the covering test set?

This procedure and the checklist could solve the problem of Instrumentation.

Limitations

Limitations

- Instrumentation probe (marker, counter) may disturb the timing relations & hide racing condition bugs. Instrumentation probe (marker, counter) may not detect location dependent Bugs.
- If the presence or absence of probes modifies things (for example in the data base) in a faulty way, then the probes hide the bug in the program.

Implementation & Applications Path Testing

Application of path testing to New Code

- Do Path Tests for C1 + C2 coverage
- Use the procedure similar to the idealistic bottom-up integration testing, using a mechanized test suite.
- A path blocked or not achievable could mean a bug.
- When a bug occurs the path may be blocked.

Application of path testing

Application of path testing to Maintenance

- Path testing is applied first to the modified component.
- Use the procedure similar to the idealistic bottom-up integration testing, but without using stubs.
- Select paths to achieve C2 over the changed code.
- Newer and more effective strategies could emerge to provide coverage in maintenance phase.

Application of path testing

Application of path testing to Rehosting

- Path testing with C1 + C2 coverage is a powerful tool for rehosting old software.
- Software is rehosted as it’s no more cost effective to support the application environment.
- Use path testing in conjunction with automatic or semiautomatic structural test generators.

Process of path testing during rehosting

Process of path testing during rehosting

- A translator from the old to the new environment is created & tested. Rehosting process is to catch bugs in the translator software.
- A complete C1 + C2 coverage path test suite is created for the old software. Tests are run in the old environment. The outcomes become the specifications for the rehosted software.
- Another translator may be needed to adapt the tests & outcomes to the new environment.
- The cost of the process is high, but it avoids risks associated with rewriting the code.
- Once it runs on new environment, it can be optimized or enhanced for new functionalities (which were not possible in the old environment.)

UNIT-II
Transaction Flow Testing

Course Learning Outcome

CLOs	Course Learning Outcome
CLO7	List Transaction flows techniques and transaction flow structures and their test databases.
CLO8	State Basics of data flow testing and Strategies in data flow testing, applications of dataflow testing.

Definitions

- Transaction-flow

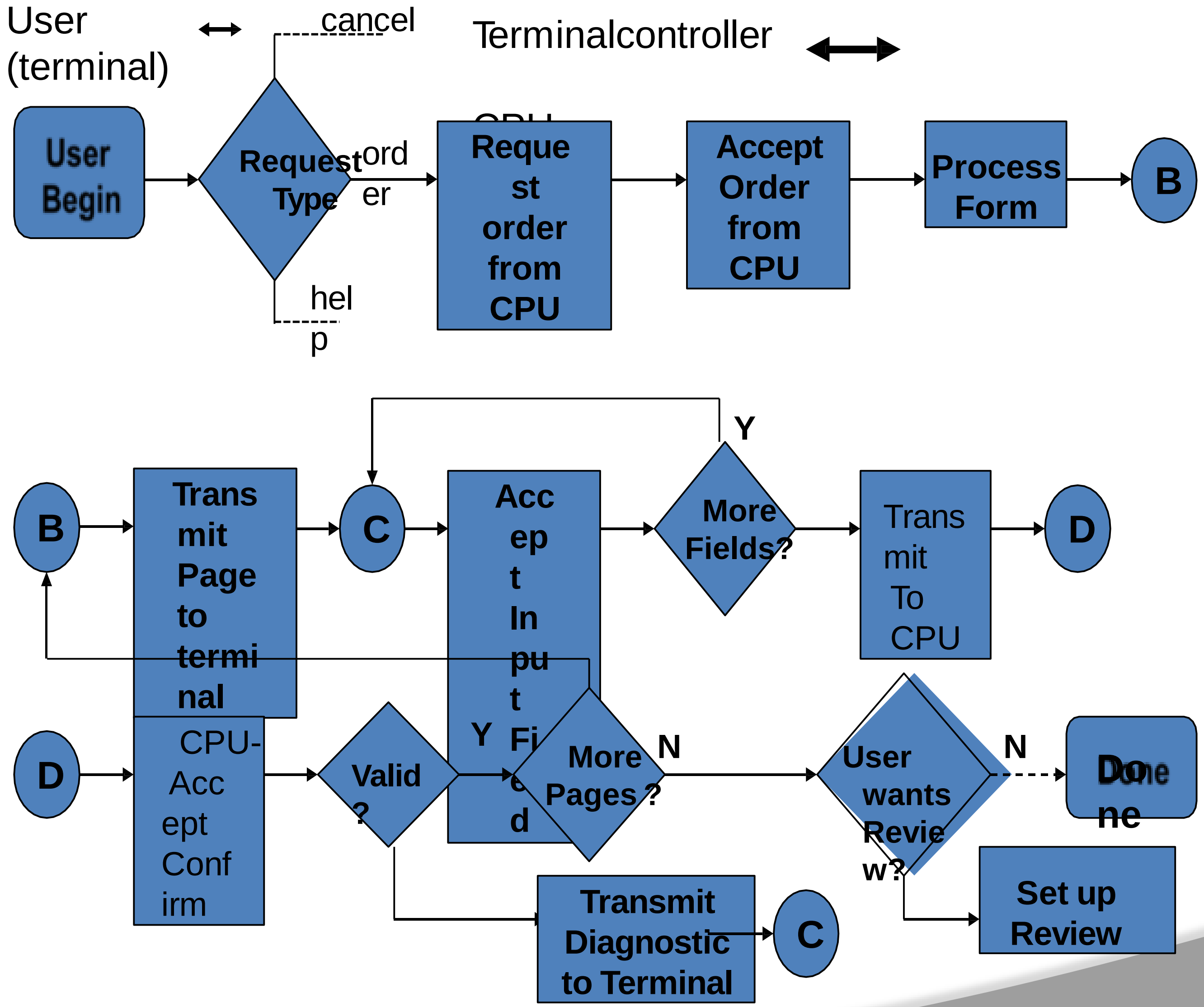
  Transaction-flow represents a system's processing. Functional testing methods are applied for testing T-F.

- Transaction-flow Graph

  TFG represents a behavioral (functional) model of the program (system) used for functional testing by an independent system tester.

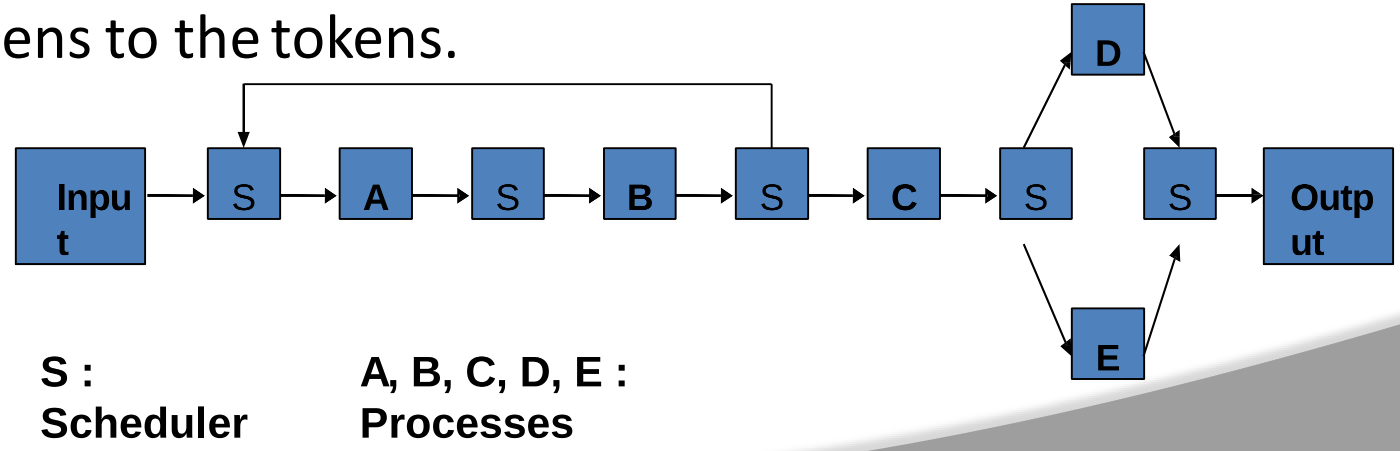
- Transaction
  - It is a unit of work seen from system's user point of view.
  - consists of a sequence of operations performed by a system, persons or external devices.
  - It is created (birth) due to an external act & up on its completion (closure), it remains in the form of historical records.

Example of a Transaction flow



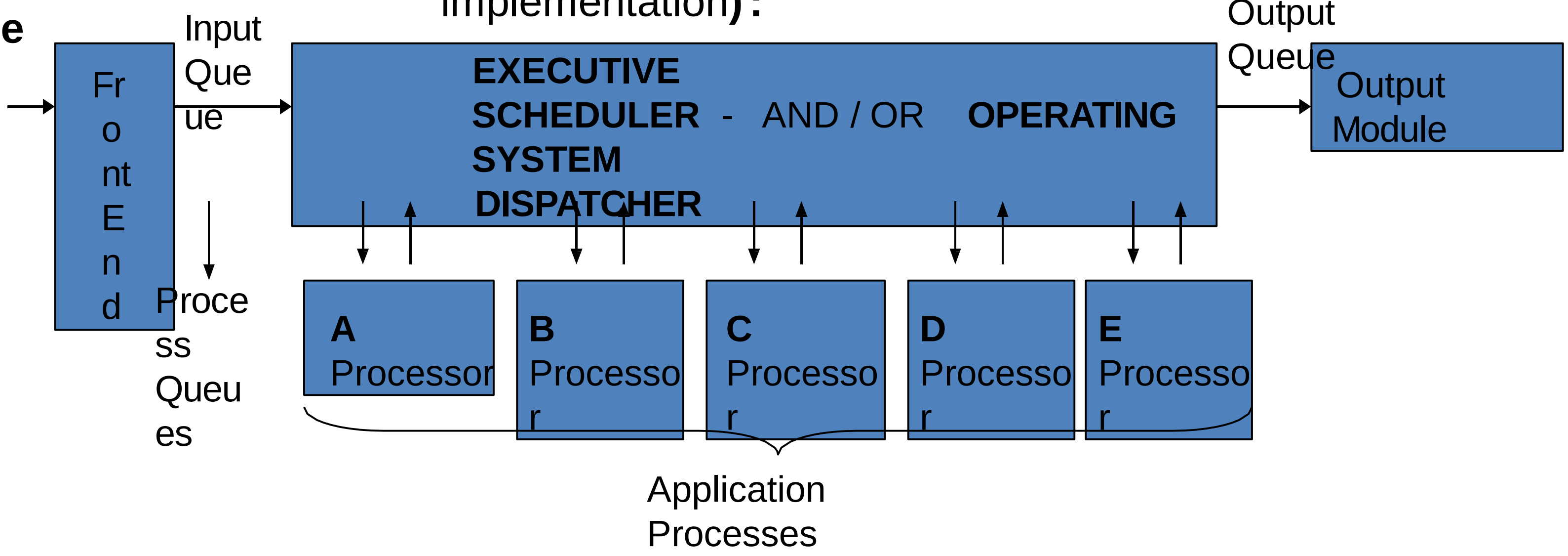
Implementation of Transaction-Flow (in a system)

- Implicit in the design of system's control structure & associated database.
- No direct one-to-one correspondence between the processes" and "decisions" of transaction-flow, and the corresponding program component.
- A transaction-flow is a path taken by the transaction through a succession of processing modules.
- A transaction is represented by a token.
- A transaction-flow graph is a pictorial representation of what happens to the tokens.



Implementation of Transaction-Flow

System Control Structure (architecture of the implementation) :



Front End

Input Queue

EXECUTIVE SCHEDULER - AND / OR **OPERATING SYSTEM DISPATCHER**

Output Queue

Output Module

Process Queues

A Processor

B Processor

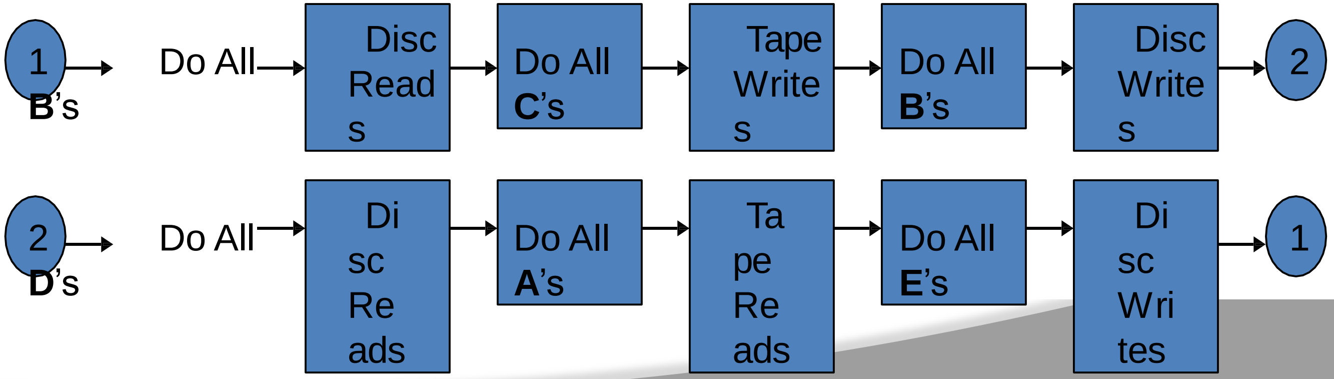
C Processor

D Processor

E Processor

Application Processes

Executive / Dispatcher Flowchart *(a sample sequence)*



1 Do All **B**'s → Disc Reads → Do All **C**'s → Tape Writes → Do All **B**'s → Disc Writes → 2

2 Do All **D**'s → Disc Reads → Do All **A**'s → Tape Reads → Do All **E**'s → Disc Writes → 1

Implementation of Transaction-Flow

- System control structure:

  System is controlled by a scheduler

  A Transaction is created by filling in a Transaction Control Block (TCB) by user inputs and by placing that token on input Q of Scheduler.

  Scheduler examines and places it on appropriate process Q such as A. When A finishes with the Token, it places the TCB back on the scheduler Q.

- Scheduler routes it to the next process after examining the token

  1. It contains tables or code to route a token to the next process.
  2. It may contain routing information only in tables.
  3. Scheduler contains no code / data. Processing modules contain code for routing.

Transaction Processing System:

- There are many Tr. & Tr-flows in the system.
- Scheduler invokes processes A to E as well as disk & tape read & writes.
- The order of execution depends on priority & other reasons.
- Cyclic structure like in this example is common in process control & communication systems.
- Cyclic structure like in this example is common in process control & communication systems

A perspective of Transaction-Flow

Transaction-flow testing is a block box technique.

1. TFG is a kind of DFG.

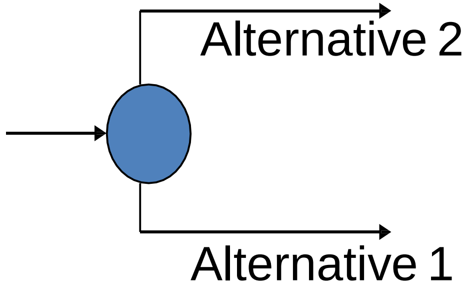
   TFG has tokens, & DFG has data objects with history of operations applied on them. Many techniques of CFG apply to TFG & DFG

2. Decision nodes of TFG have exception exits to the central recovery process.So we ignore the effect of interrupts in a Transaction-flow.

Transaction Flows – splitting & merging decisions

Splits of transactions(Births)

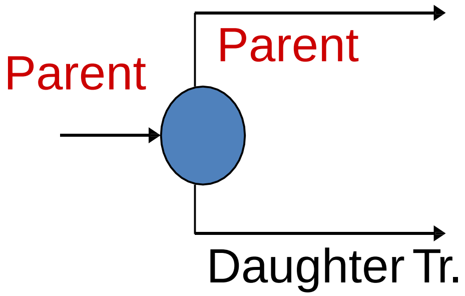
1. A decision point in TFG



Alternative 2

Alternative 1

2.Biosis

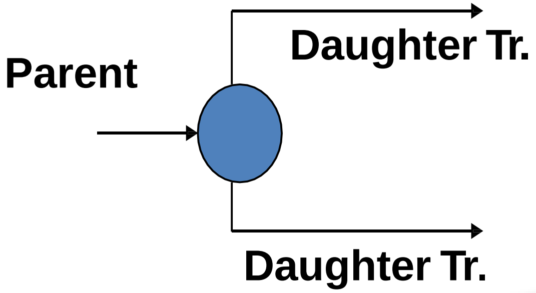


Parent

Parent

Daughter Tr.

3.Mitosis



Parent

Daughter Tr.

Daughter Tr.

Data - Flow Testing

Anomaly:

Unreasonable processing on data

- Use of data object before it is defined
- Defined data object is not used

Data Flow Testing (DFT) uses Control Flow Graph (CFG) to explore dataflow anomalies.

- DFT Leads to testing strategies between P and P1 / P2

Data - Flow Testing

Definition:

DFT is a family of test strategies based on selecting paths through the program’s control flow in order to explore the sequence of events related to the status of data objects.

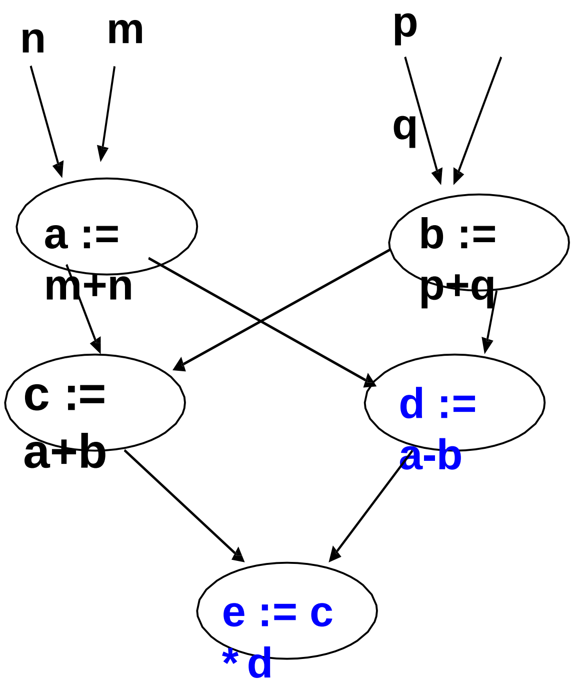
Example:

Pick enough paths to assure that every data item has been initialized prior to its use, or that all objects have been used for something.

Data - Flow Testing

Program Flow using Data Flow Machines paradigm

```
BEGIN
 PAR DO
          READ m, n, n, p, q
END PAR
PAR DO
          a := m+n
          b := p+q  END
PAR  PAR DO
          c := a+b
          d := a-b  END
PAR  PAR DO
          e := c * d  END
PAR  END
```

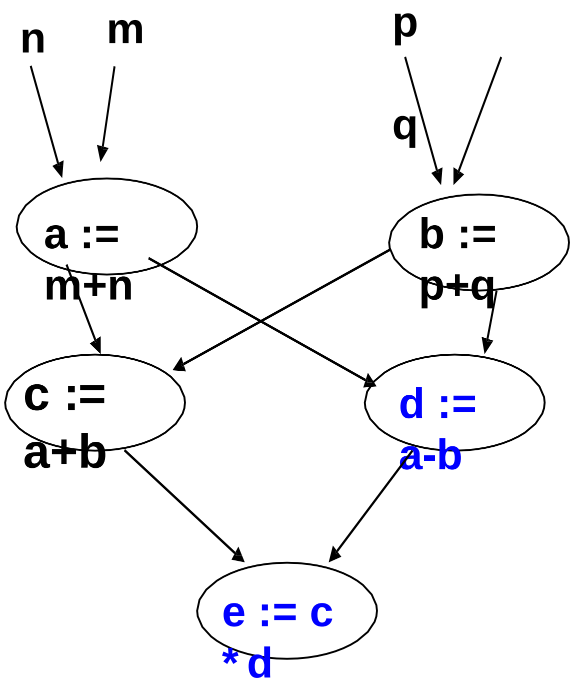


The interrelations among the data items remain same.

Data - Flow Testing

Program Flow using Data Flow Machines paradigm

```
BEGIN  PAR DO
        READ m, n, n, p, q
END PAR
PAR DO
        a := m+n  b :=
        p+q
END PAR
PAR DO
        c := a+b
        d := a-b  END
PAR  PAR DO
        e := c * d  END
PAR  END
```

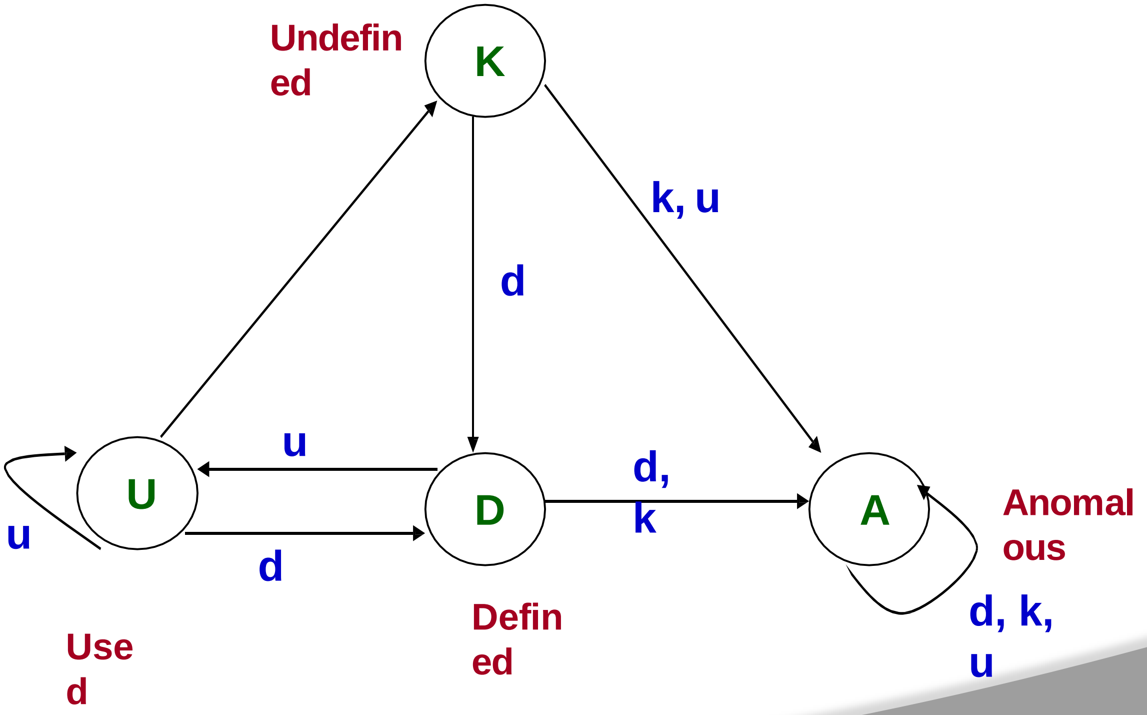


The interrelations among the data items remain same.

Data - Flow Testing

Data Flow Anomaly State graph

- Object state
- Unforgiving Data flow state graph



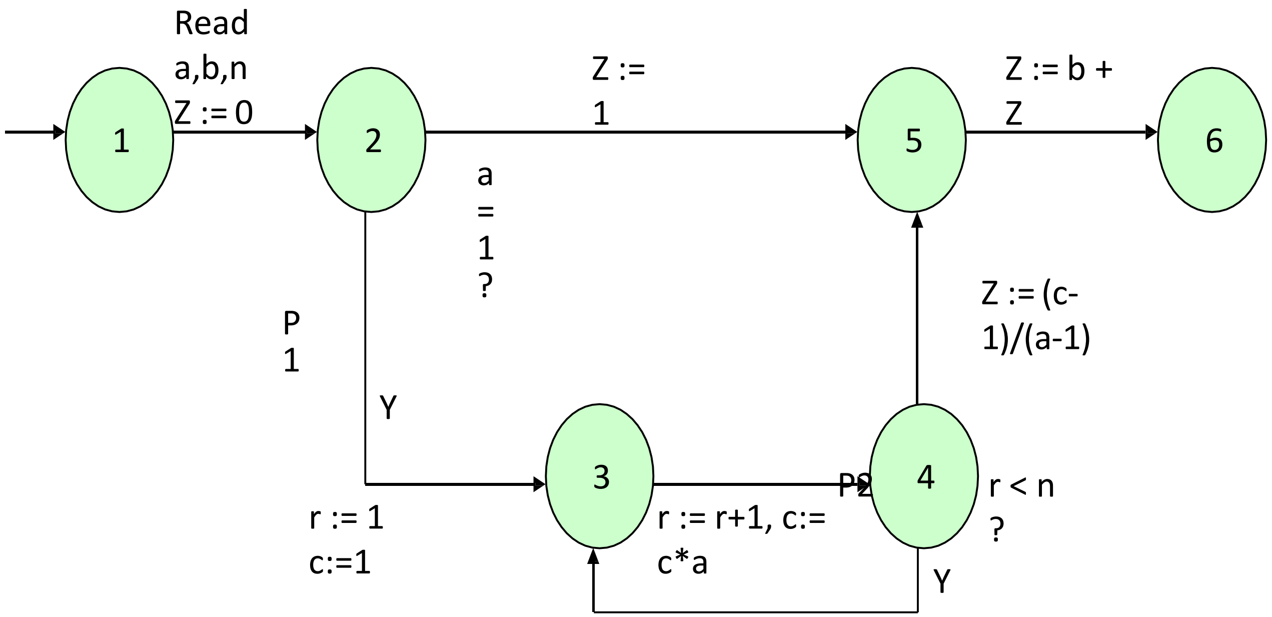
Data - Flow Testing

Procedure To Build:

1. Entry & Exit nodes
2. Unique node identification
3. Weights on out link
4. Predicated nodes
5. Sequence of links
    - Join
    - Concatenate weights
    - The converse

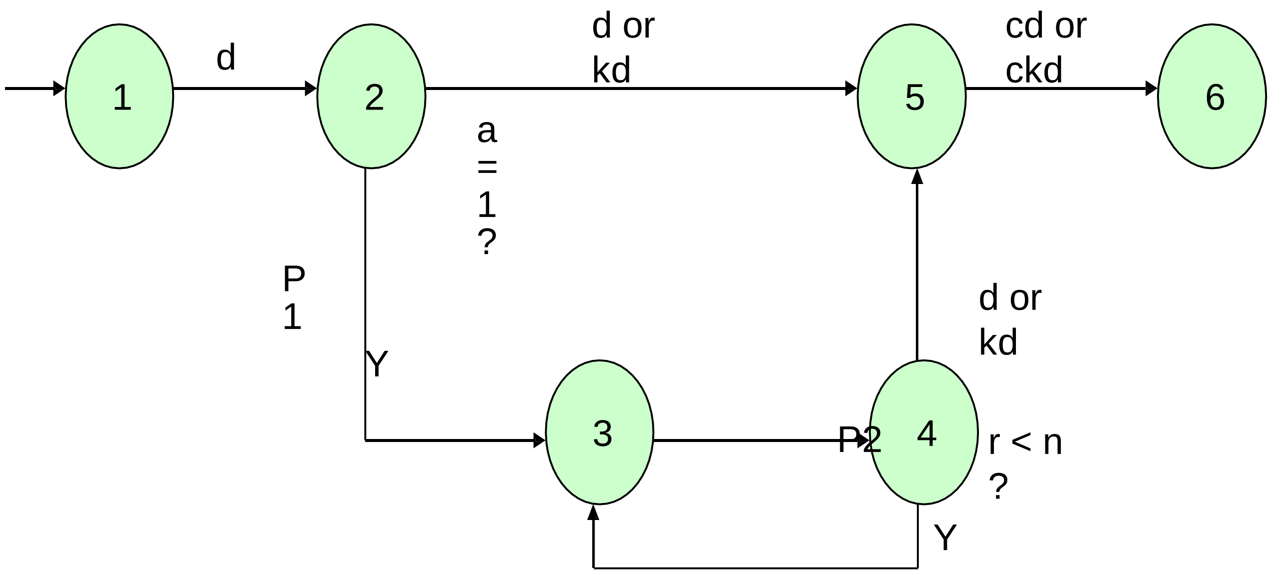
Data - Flow Testing

CFG for the Example



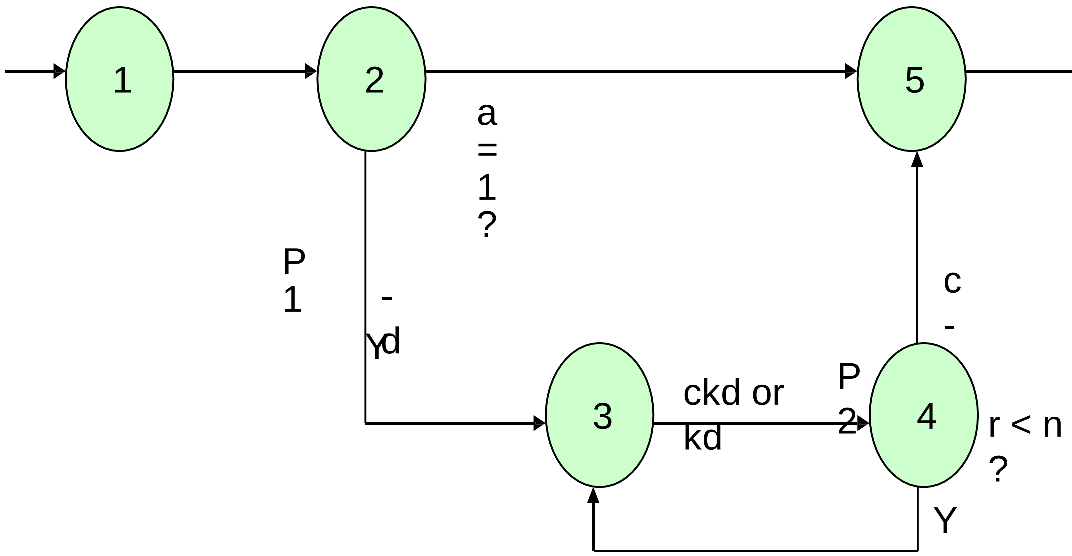
Data - Flow Testing-Basics – Data Flow model

CFG annotated – Data Flow Model for Z



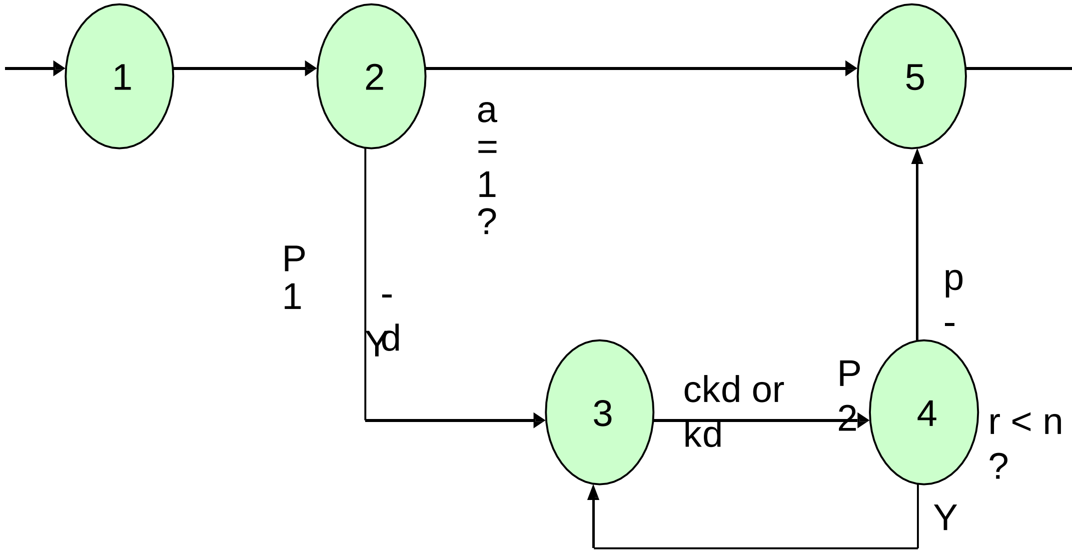
CFG annotated – Data Flow Model for c

CFG annotated – Data Flow Model for c



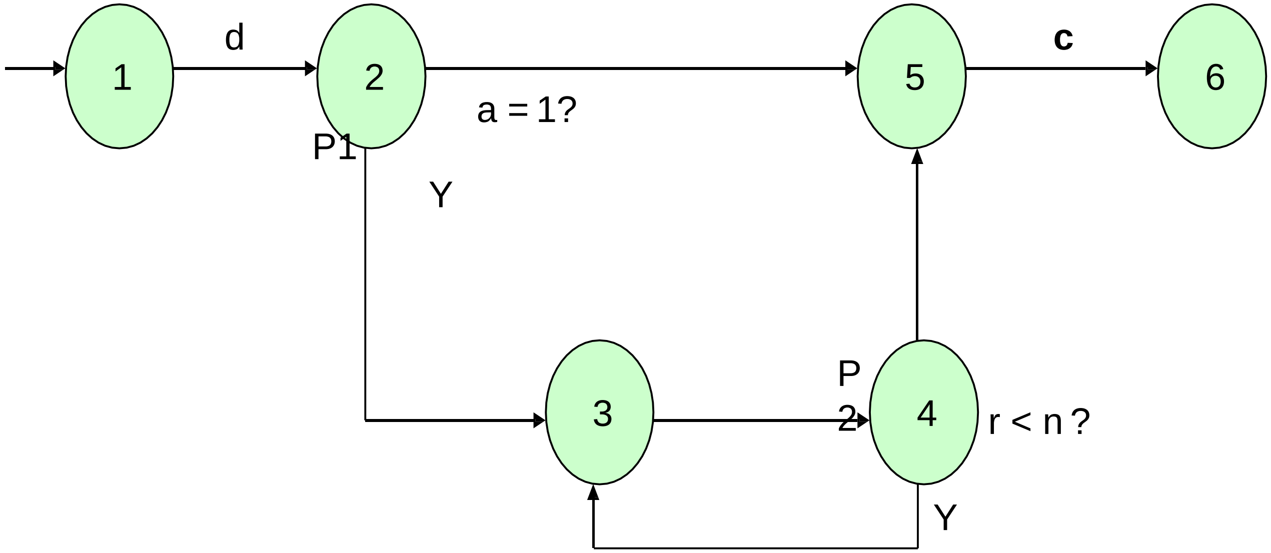
Data -Flow Testing-Basics – Data Flow model

CFG annotated – Data Flow Model for r



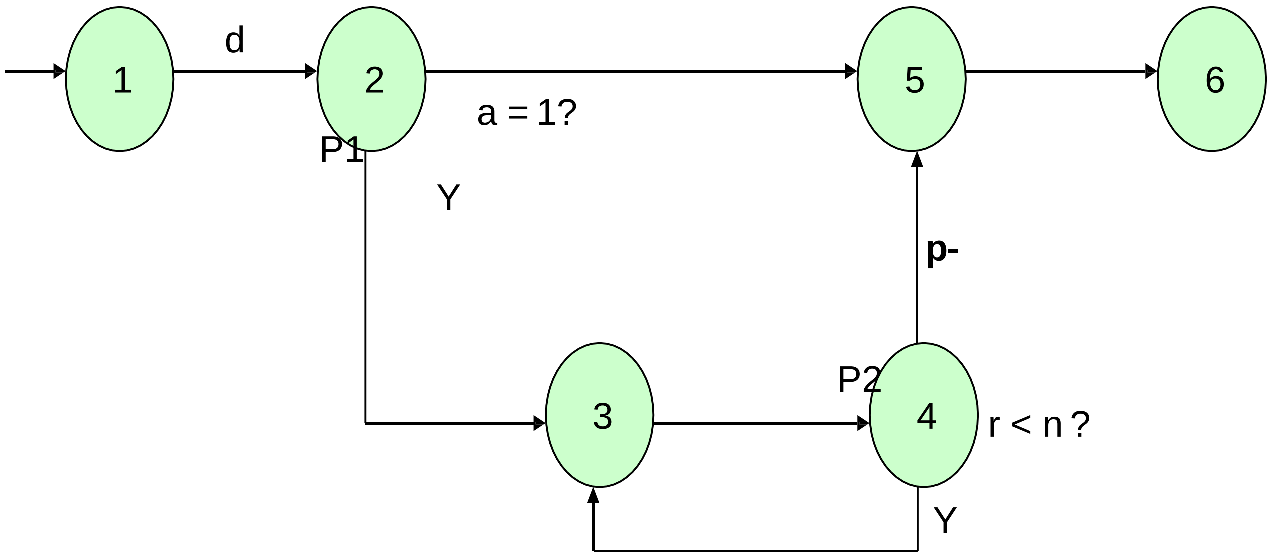
CFG annotated – Data Flow Model for b

CFG annotated – Data Flow Model for b



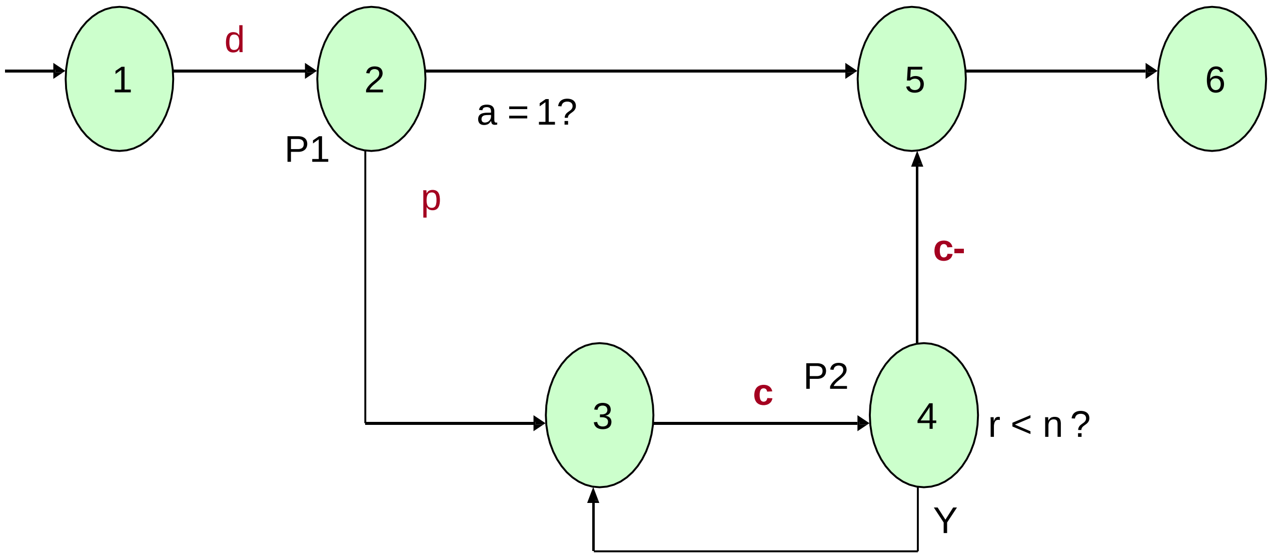
CFG annotated – Data Flow Model for n

CFG annotated – Data Flow Model for n



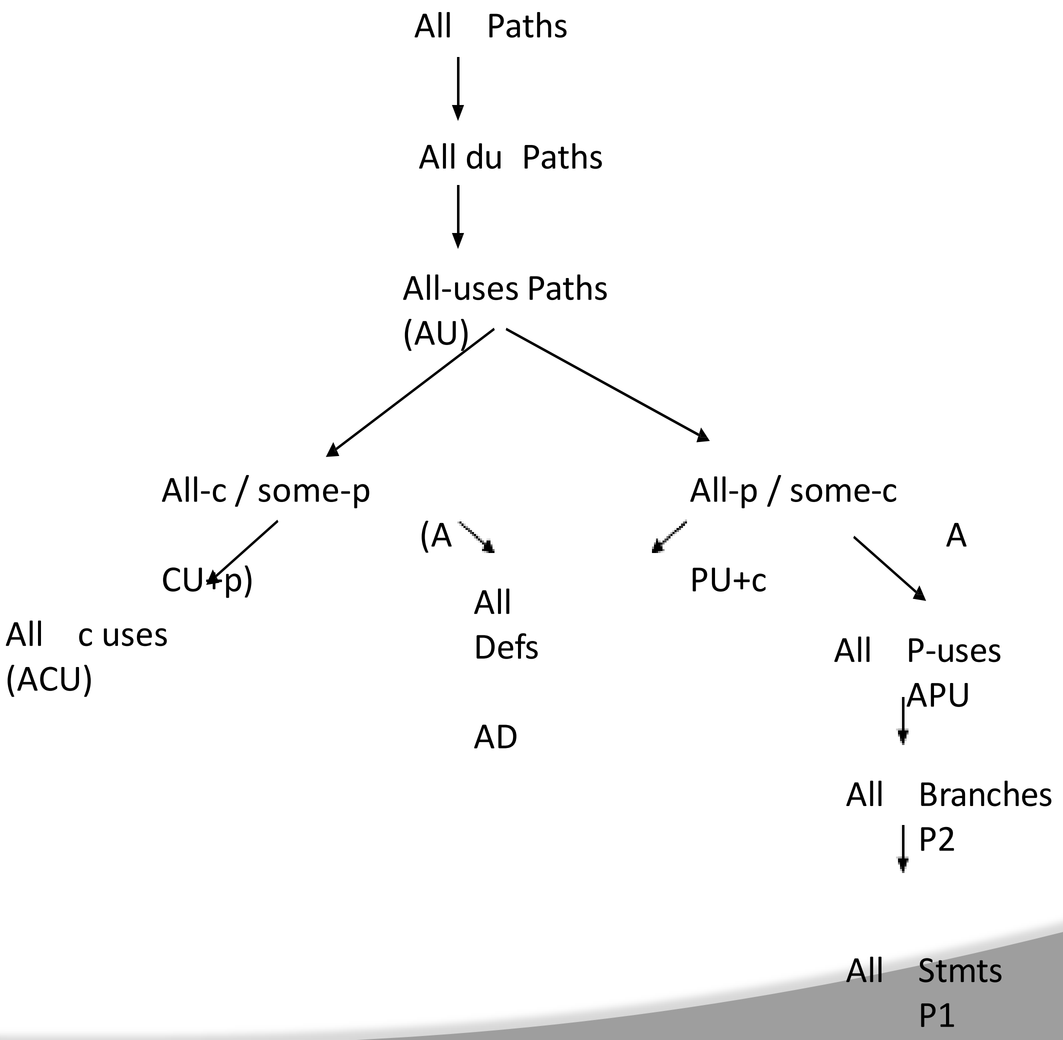
CFG annotated – Data Flow Model for a

CFG annotated – Data Flow Model for a



Data - Flow Testing– Data Flow Testing Strategies

Ordering the strategies



All Paths

All du Paths

All-uses Paths (AU)

All-c / some-p

(A

CU+p)

All c uses (ACU)

All Defs

AD

All-p / some-c

A

PU+c

All P-uses APU

All Branches P2

All Stmts P1

Debugging

Testing, Maintenance & Debugging in the Data Flow context Debugging:

- Select a slice.
- Narrow it to a dice.
- Refine the dice till it’s one faulty stmt.

UNIT-III
DOMAIN TESTING

Course Learning Outcomes

CLOs	Course Learning Outcomes
CLO9	Describe Domains and paths and explain about domains and bugs and their tools effectiveness.
CLO10	Demonstrate Domains and Interfaces testing.
CLO11	Explain linearising transformation and coordinate transformation
CLO12	Describe Logic based testing and Decision tables and compare hardware and software testing.

Domain Testing Model

Domain Testing Model

Two Views

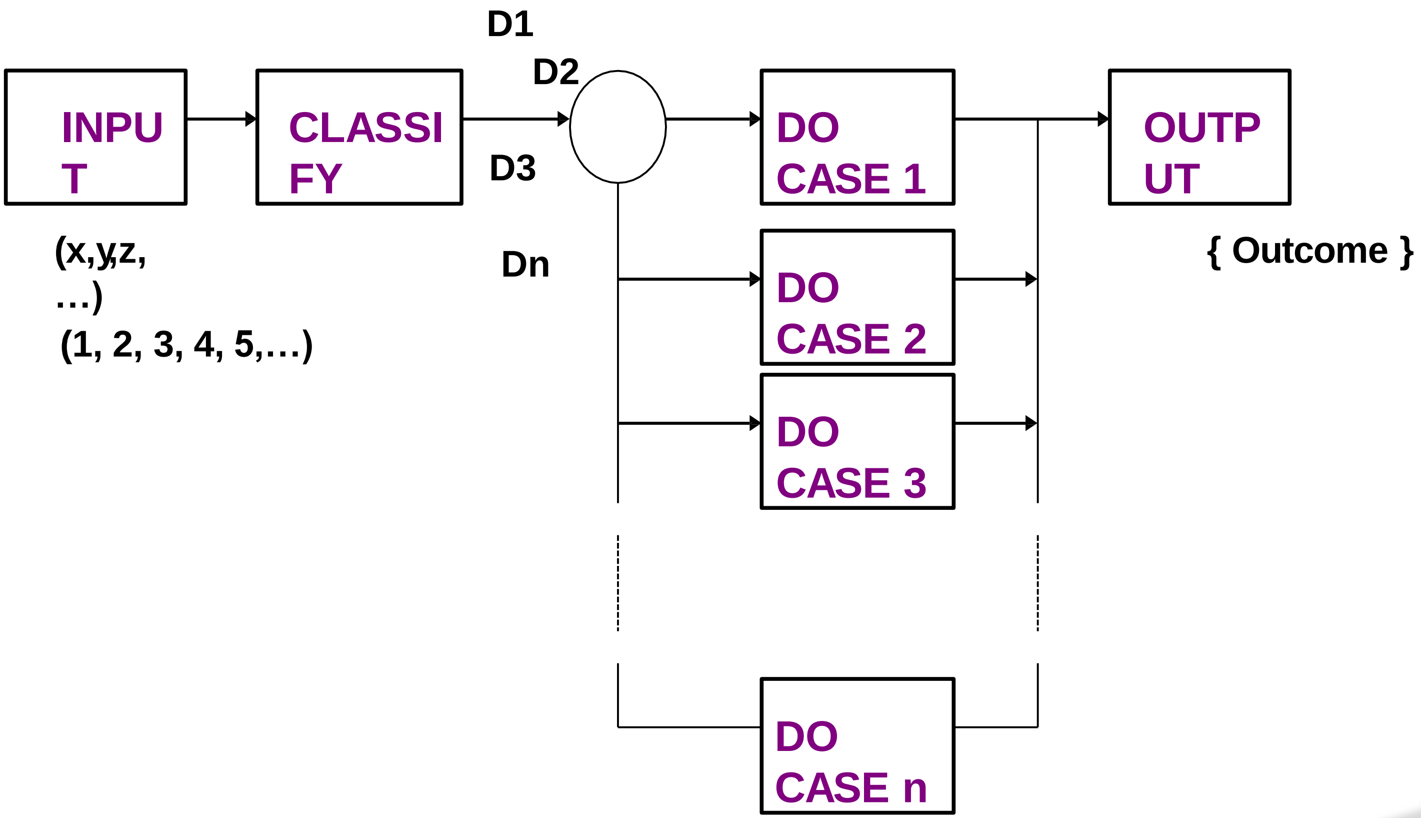
- Based on Specs

  Functional Testing
- Based on Implementation information

  Structural Technique

Domain Testing

Domain Testing Model



Domain Testing Model

Domain Testing Model Variables

- Simple combinations of two variables
- Numbers from-infinity to +infinity
- Input variables as numbers
- Loop-free Programs

Domain Testing Model

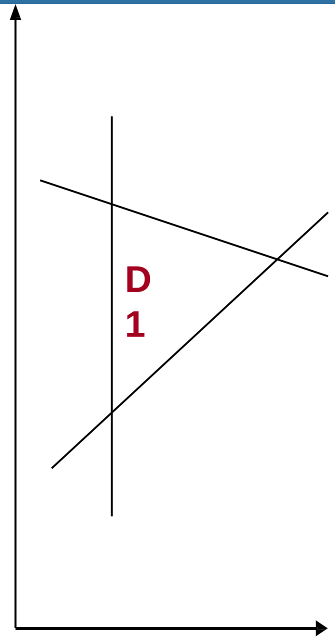
Domain Testing Model

- Structural Knowledge is not needed Only Specs.
- For each Input Case,

  1. A Hypothetical path for Functional testing
  2. An Actual path for Structural testing

Domain Testing Domains & Paths

- A Single Connected Set of numbers
- No arbitrary discrete sets
- Defined by the boundaries
    1. One or more boundaries
    2. Specified by predicates
    3. Bugs likely at the boundaries
- One or more variables



Predicates

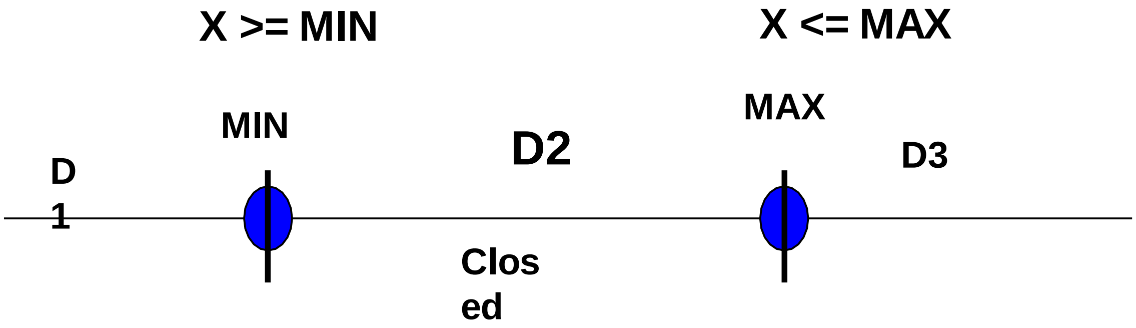
Predicates

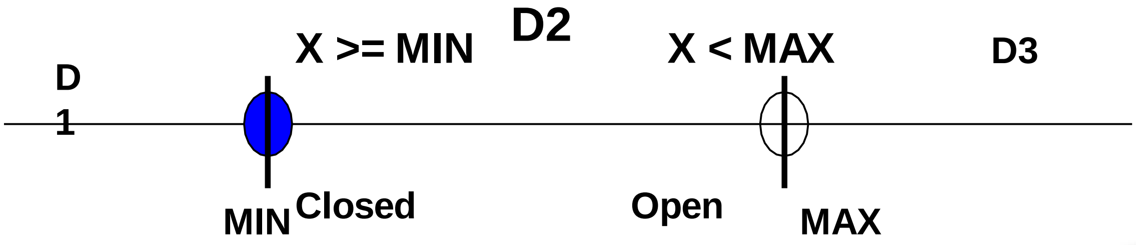
- Interpretation
  1. Structural Testing - CFG
  2. Functional Testing - DFG
- Specifies the Domain Boundary
- Predicates in Sequence => domains Or, just two domains .

  A .AND. B .AND. C

Boundary: Closed / Open

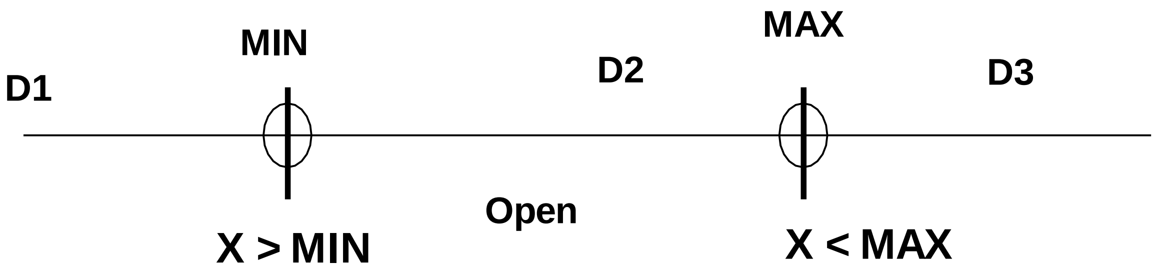
Boundary: Closed / Open





Domain Closure

Boundary: Closed / Open



Domain Dimensionality

- One dimension per variable
- At least one predicate
  - Slices thru previously defined Domain
- Slicing Boundary
- N-spaces are cut by Hyperplanes

Bug Assumptions

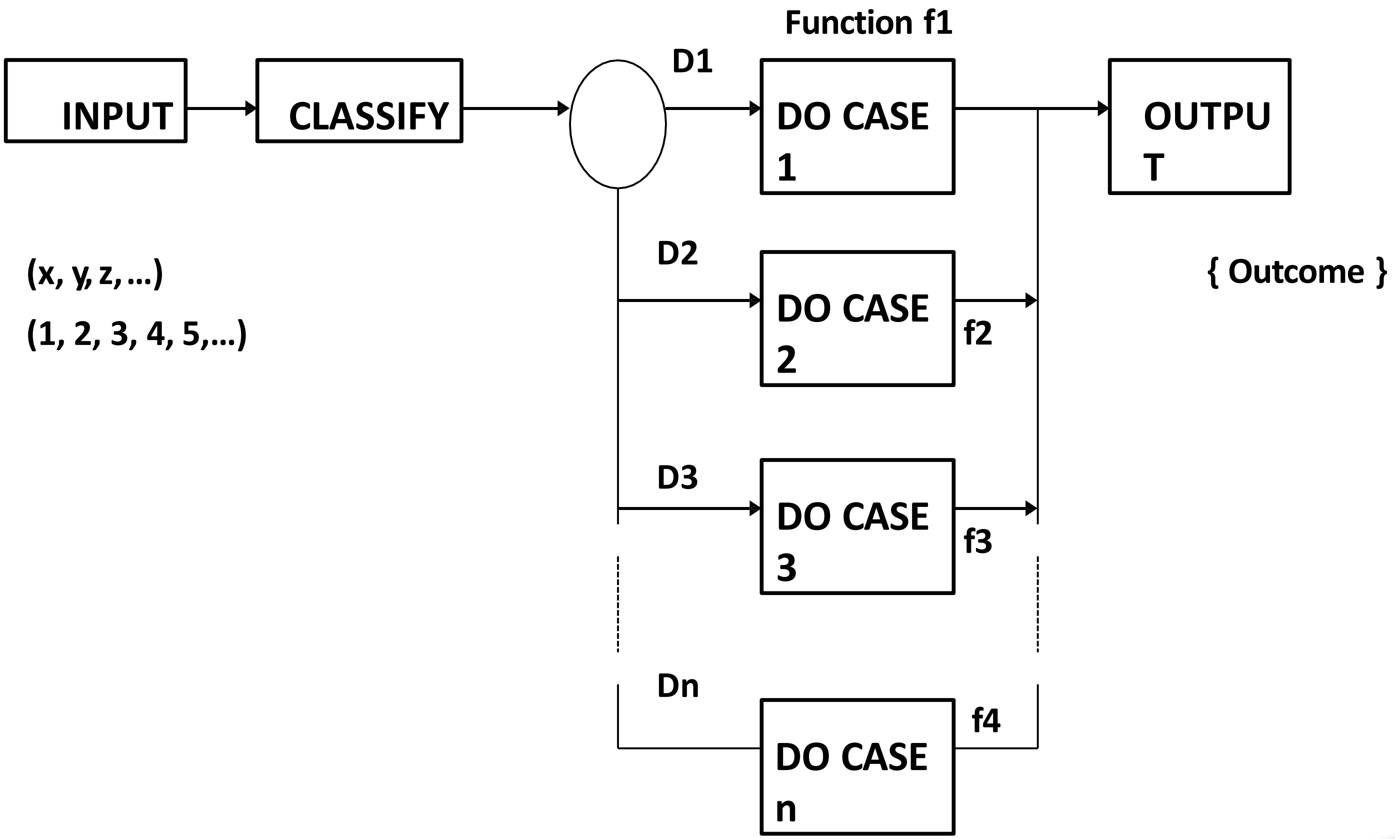
- Processing is OK.
- Domain definition may be wrong.

  Boundaries are wrong.

  Predicates are wrong.

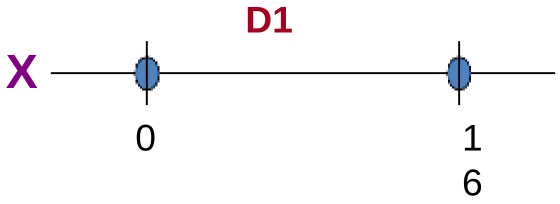
  Once input vector is set on the right path, it’s correctly processed. More bugs causing domain errors.

Domain Testing Model

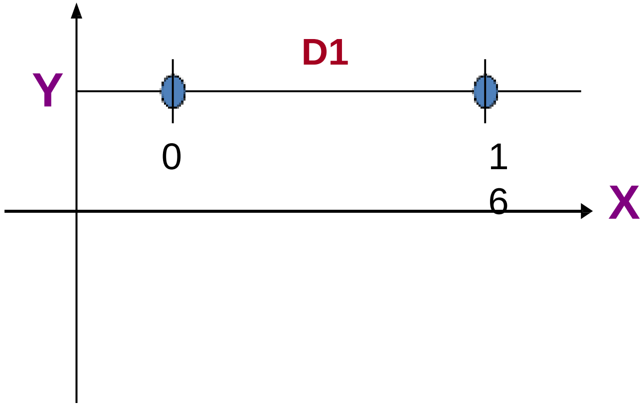


Simple boundaries

3. Simple boundaries



X >=0 .AND. X <= 16



Y = 1 .AND. X >=0 .AND. X <= 16

Compound predicates

Compound predicates….

- Impact of OR.
  - Concave, Disconnected
  - Adjacent domains with same function
  - Example

    A B C + D E F
- Eliminate compound predicates

Linear Vector Space

5. Linear Vector Space
   - Linear boundary predicate, Interpreted
   - Simple relational operators
   - Conversion to linear vector space
   - 2-d Polar co-ordinates
   - Polynomials Problems with Non-linear Boundaries

Nice & Ugly domains

Nice Domains

Requirements

- Bugs => ill-defined domains

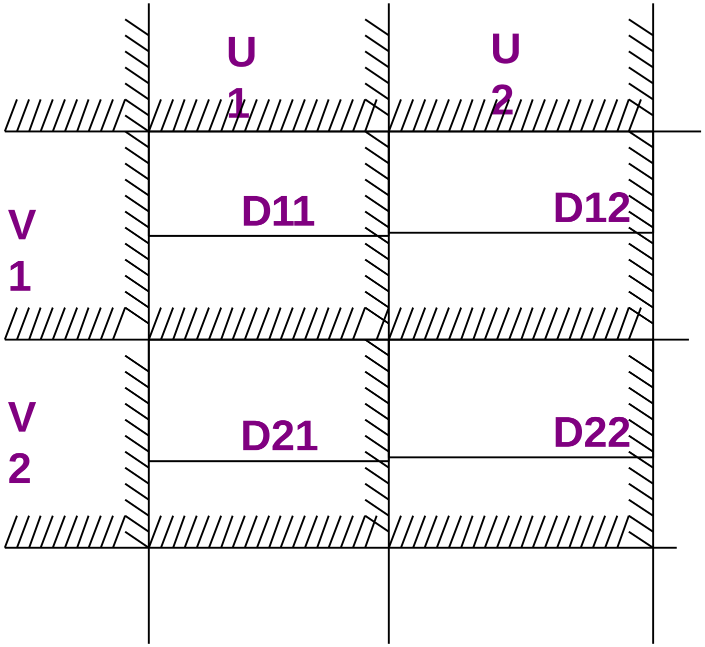
Before DT

- Analyze specs
- Make the Boundary Specs Consistent & Complete

Nice domains

Nice Domains

- Linear, Complete, Systematic, Orthogonal, Consistently Closed, & Convex



Nice domains

Nice Domains:

Boundaries are

1. Linear
2. Complete
3. System
4. Orthogonal
5. Convex
6. Closure consistency
7. Simply connected

Domains & Paths – Nice domains

- Linear Boundaries

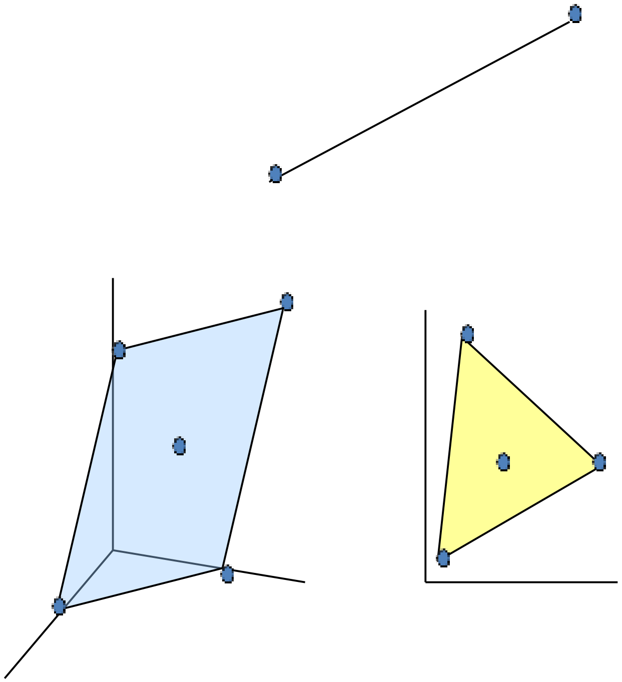
  Interpreted linear inequalities n-dim Hyperplane:

  n+1 Points

  n+1 + 1 Test Cases

- Non-Linear

- Transform



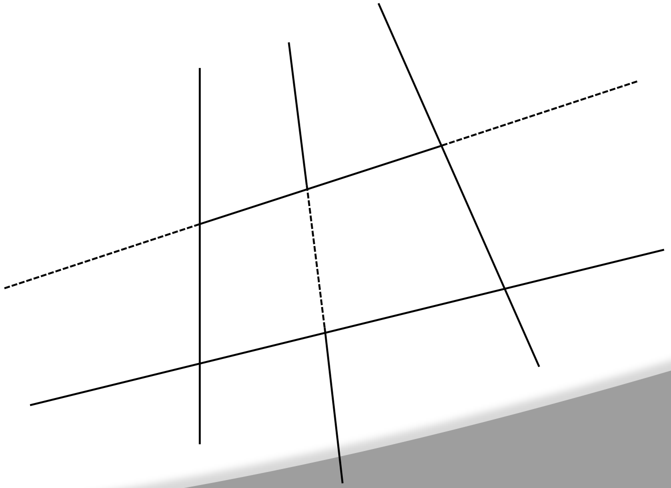
Domain Testing - Domains & Paths – Nice domains

2. Complete Boundaries

- Span the total number space (-∞,+ ∞)
  - One set of Tests

Incomplete…

- Reasons
- Tests



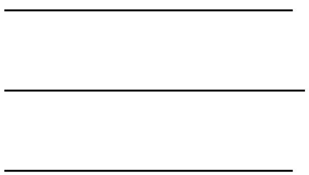
Nice domains

3. Systematic Boundaries

- Linear Inequalities differing by a constant

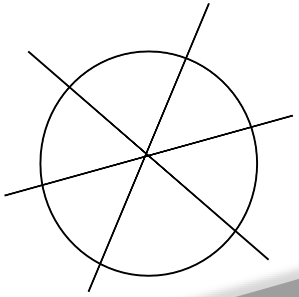
$f_j(X) \geq k_j$ or, $f_j(X) \geq g(j, c)$ $g(j, c) = j + k * c$

- Parallel lines
- Identical Sectors in a Circle



- DT

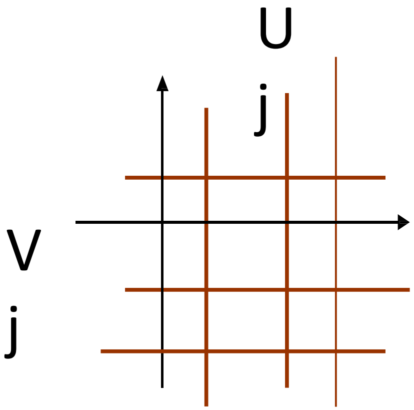
- Test a domain Tests for other Domains



Orthogonal Boundaries

4. Orthogonal Boundaries

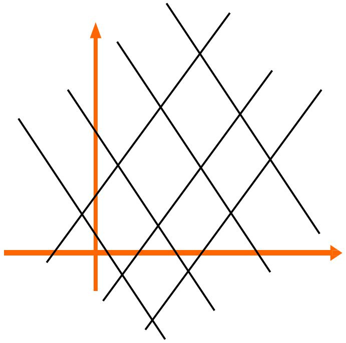
- Two boundaries or, boundary sets
- Parallel to axes
- DT
  - Each Set Independently
  - # Tests ⍰ O (n)



Orthogonal Boundaries

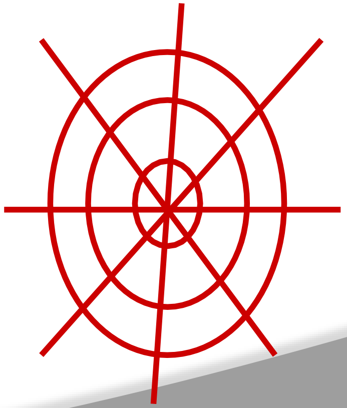
Orthogonal Boundaries

- Tilted sets
  - transformation
  - Test cases: O(n)



- Concentric circles with radial lines
  - Rectangular coordinates
  - Polar

    $r \geq a_j$ .AND. $r < a_{j+1}$



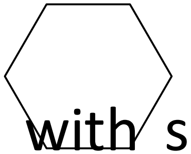
Closure Consistency

5. Closure Consistency
    - A Simple pattern in all boundary closures
    - Example
    - Same relational operator for systematic boundaries

Convex Domain

6. Convex Domain

- Line joining any two points lies with in the domain
- DT
- Non-points &1 off-pt Concave
- “ But, However, Except, Or …“in Specs Handle with special care



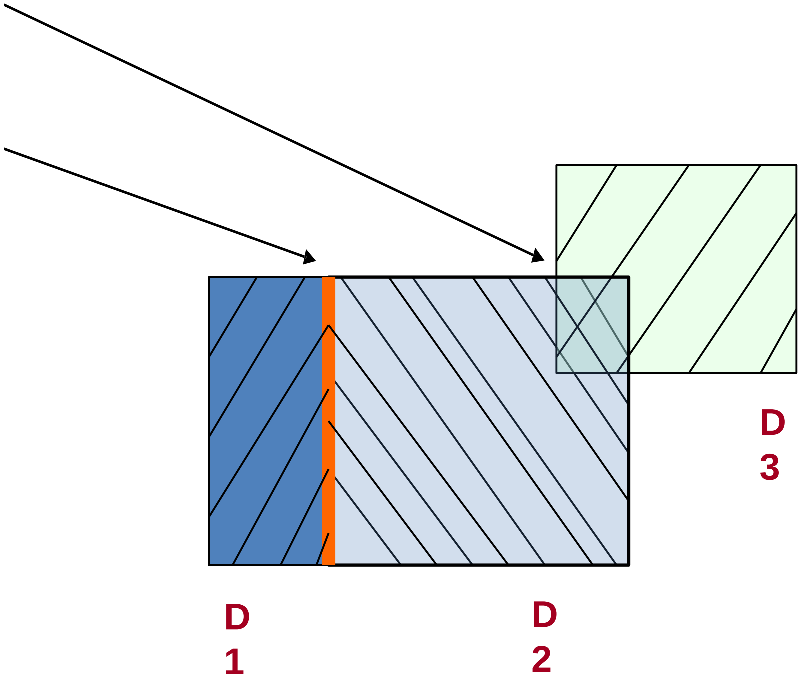
Ambiguities & Contradictions

- Holes in input vector space.
- Missing boundary
- Detected by Specification languages & tools.

Contradictions

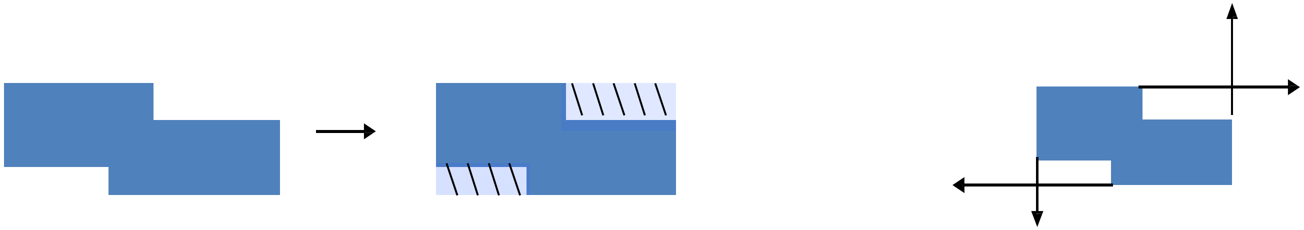
2. Contradictions..

- Overlapping of
  - Domain Specs
  - Closure Specs

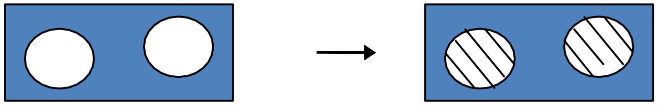


Domain Testing-Domains & Paths – Ugly Domains

Simplifying the Topology - Smoothing out concavity



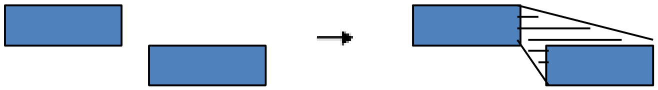
- Filling in Holes

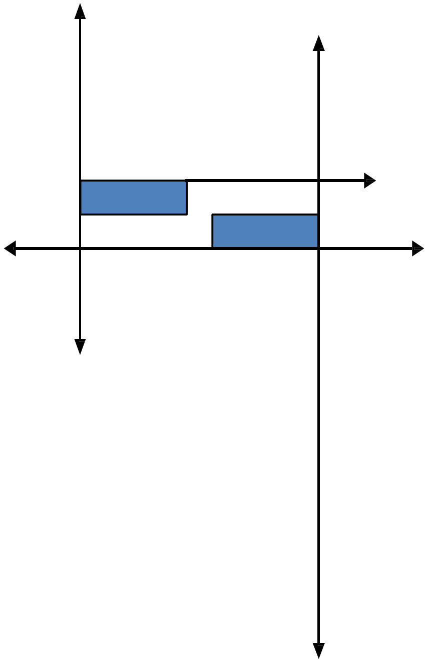


Domain Testing-Domains & Paths – Ugly Domains

3. Simplifying the Topology….

- Joining the pieces





Correct:

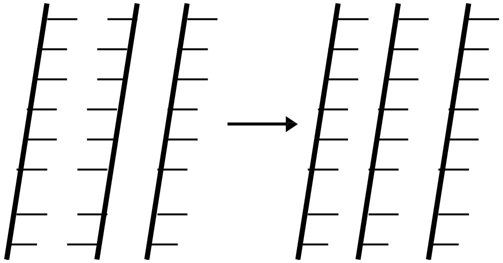
- Connect disconnected boundary segments

- Extend boundaries to infinity

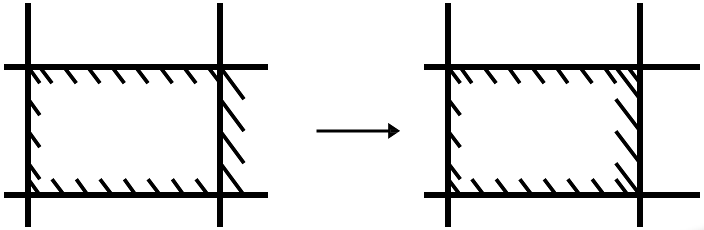
Domain Testing-Domains & Paths – Ugly Domains

4. Rectifying Boundary Closures.
   - make closures in one direction for parallel boundaries with closures in both directions

   - Force a Bounding Hyperplane to belong to the Domain.



Consistent Direction



Inclusion / Exclusion Consistency

Domain Testing - Domains & Paths

General DT Strategy

1. Select test points near the boundaries.
2. Define test strategy for each possible bug related to boundary
3. Test points for a domain useful to test its adjacent domain.
4. Run the tests. By post test analysis determine if any boundaries are faulty & if so how?
5. Run enough tests to verify every boundary of every domain

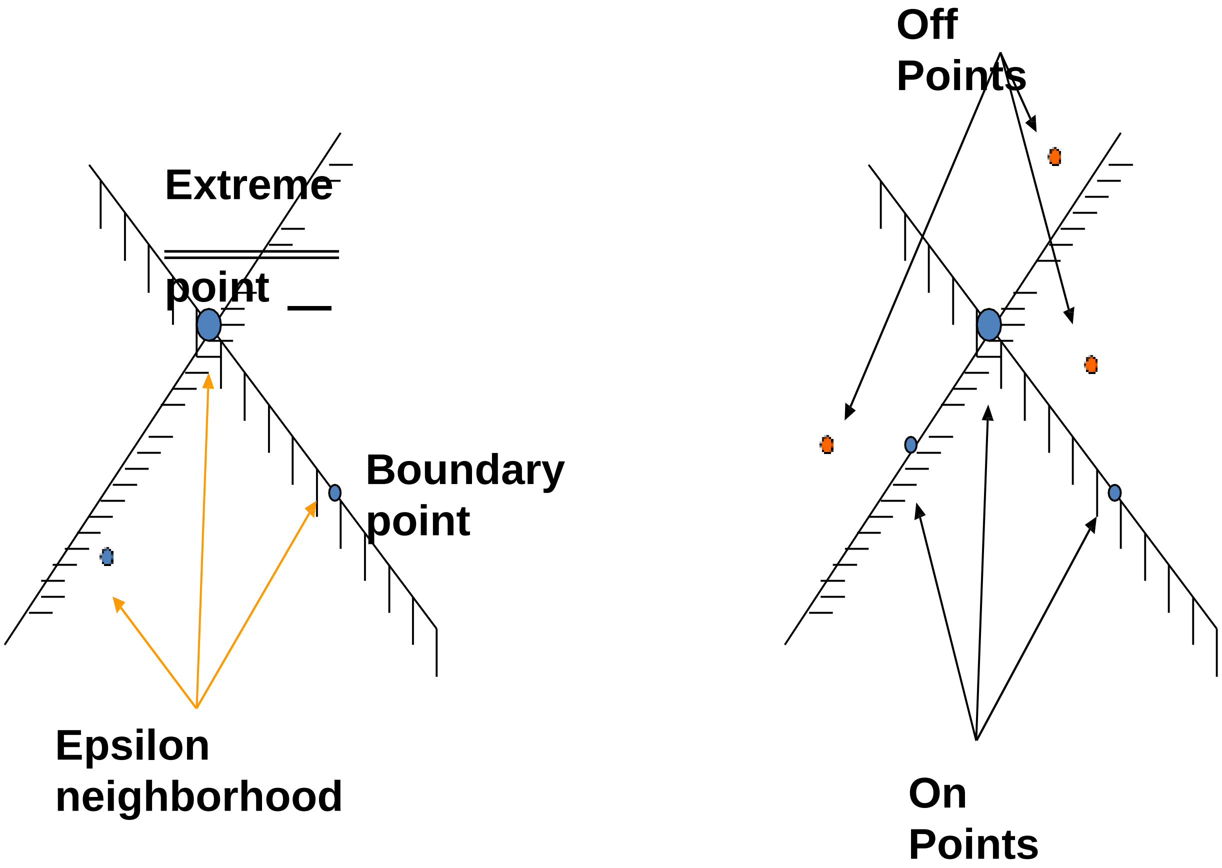
Domain Testing

DT for Specific Domain Bugs

Generally,

- Interior point
- Exterior point
- Epsilon neighborhood
- Extreme point
- On point

Domain Testing Specific Domain Bugs



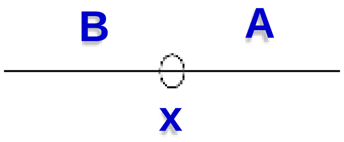
Domain D1

Domain Testing-Domains & Paths

1. 1-d Domains
2. 2-d Domains
3. Equality & inequality Predicates
4. Random Testing
5. Testing n-dimensional Domains

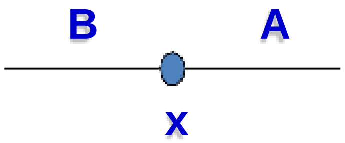
Domain Testing

Testing 1-d Domains :

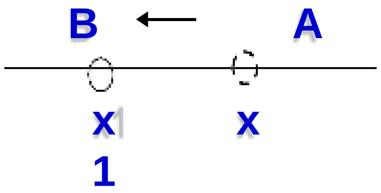


Bugs with open boundaries

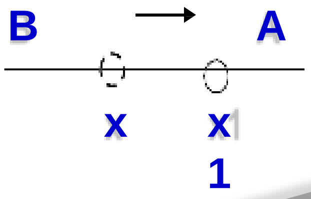
Closure Bug



Shift left Bug



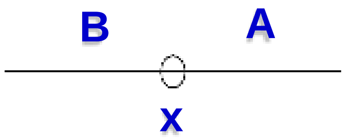
Shift Right Bug



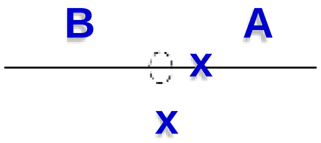
Domain Testing - Domains & Paths

Testing 1-d Domains : boundaries

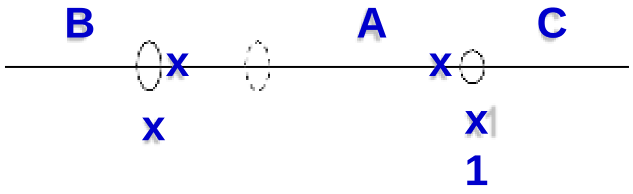
Bugs with open



Missing Boundary



Extra Boundary



Bugs with Closed Boundaries

Similar to the above

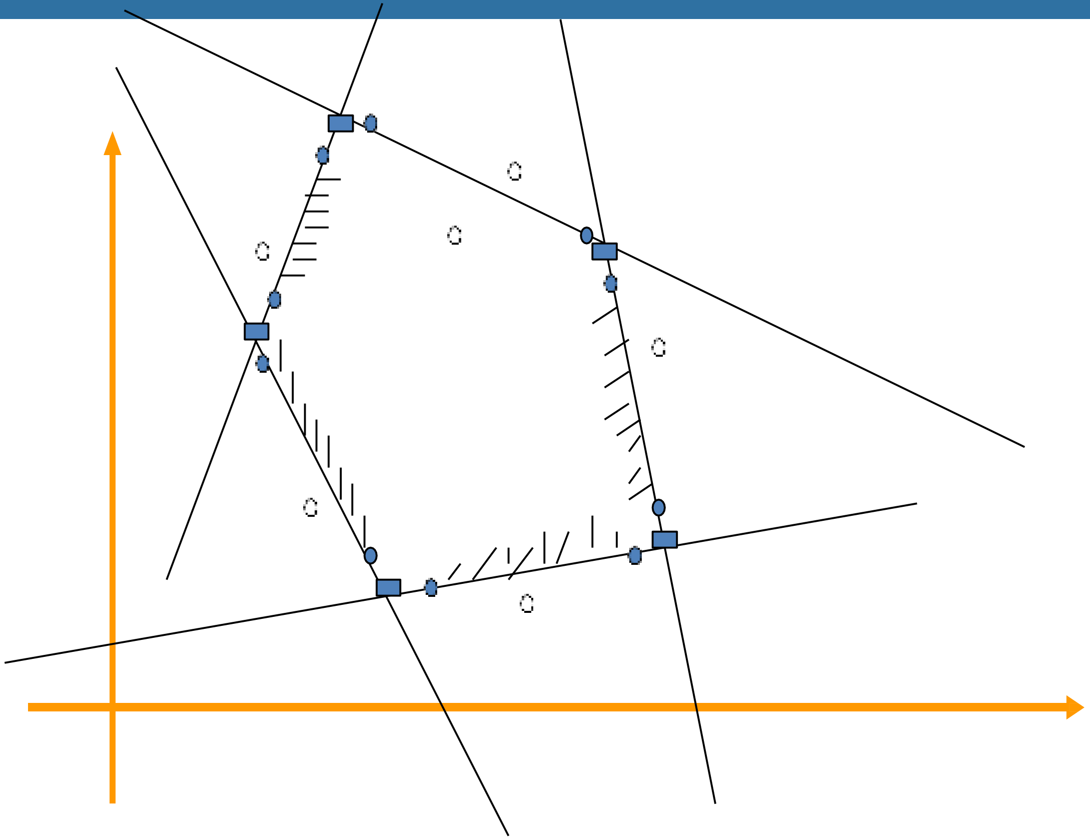
Testing 2-d Domains

Closure bug

Boundary Shift: up / down

- Tilted Boundary
- Extra Boundary
- Missing Boundary

Domains & Paths – Domain Testing



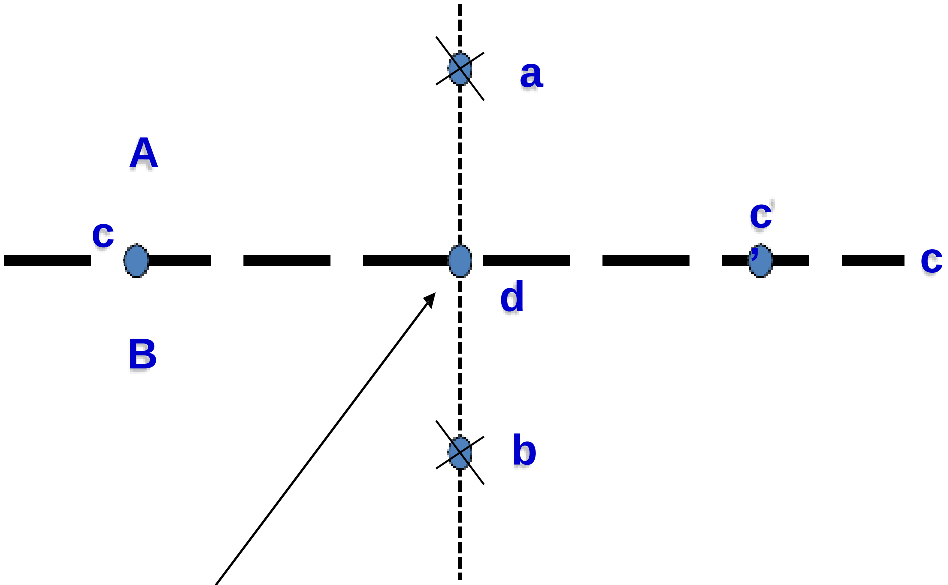
2 n : Domains share tests

3 n : no sharing of tests by domains

4 Strategy for domain testing in 2-dim

Domains & Paths – Domain Testing

3. Equality & Inequality Predicates
   - An Equality predicate defines a line in 2-d

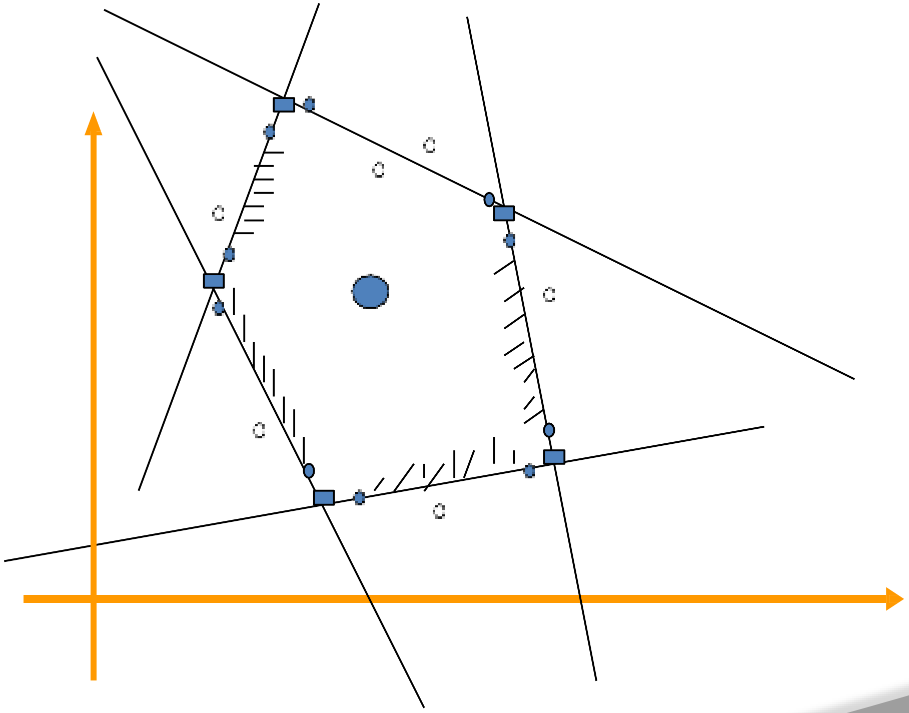


To avoid bugs

Domains & Paths – Domain Testing

Random Testing

- A Point in the center : verifies computation



Domains & Paths – Domain Testing

Testing n-Dimensional Domains (strategy)

d. imensions, p boundary segments

- (n+1)*p test cases : n on points & 1 off point
- Extreme pt shared : 2 * p points
- Equalities over m-dimensions create a subspace of n-m dimensions
- Orthogonal domains with consistent boundary closures, orthogonal to the axes & complete boundaries

Procedure for solving for values

Procedure for solving for values

Simple procedure. Need tools.

1. Identify input variables.
2. Identify variables which appear in domain-defining predicates, such as control-flow predicates.

Procedure

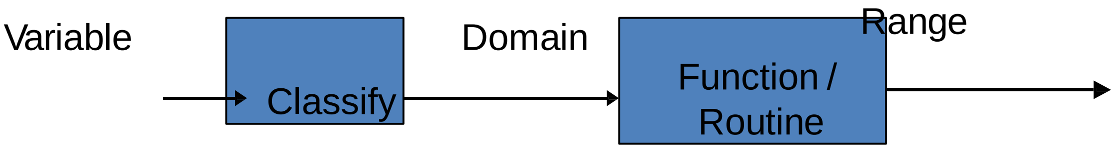
Procedure

1. Interpret all domain predicates in terms of input variables:
    1. Transform non-linear to linear
    2. Find data flow path
2. Predicate expression with p # predicates. Find # domains:< 2 ^{p}
3. Solve inequalities for extreme points
4. Use extreme points to solve for nearby on points …

Domain Testing

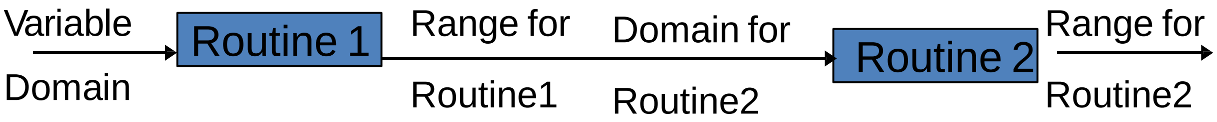
Domains & Interface Testing

- Domain
- Range

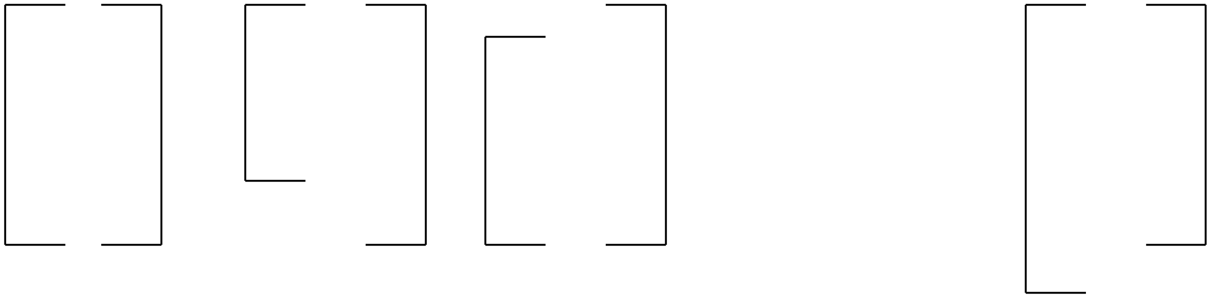


Domain Testing

Domains & Interface Testing



- Span compatibility



Logic based testing: Overview

- Logic is used in a program by programmers. Boolean algebra is the way to work with logic – simplification & calculation.
- Hardware logic testing – hardware logic test design tools and methods use logic & Boolean algebra. Hardware design language compilers/translators use logic & Boolean algebra.
- Impact of errors in specifications of a software is high as these are first in and last out. So, higher level language for specs is desired to reduce the number of errors. Higher order logic systems are used for formal specifications. The tools to simplify, transform and check specs use Boolean algebra.

Logic based testing : Overview

Knowledge based systems:

- Knowledge based systems and artificial intelligence systems use high level logic languages which are based on rule bases consisting of rules.
- Rules are predicate expressions containing domain knowledge related elements combined with logical connectives. The answers to queries (problems) are derived based on Boolean algebraic operations performed on the rule bases. Such programs are called inference engines.

Modeling Logic with Decision Tables

- Consists of three parts
  - Condition stubs
    - Lists condition relevant to decision
  - Action stubs
    - Actions that result from a given set of conditions
  - Rules
    - Specify which actions are to be followed for a given set of conditions

Modeling Logic with Decision Tables

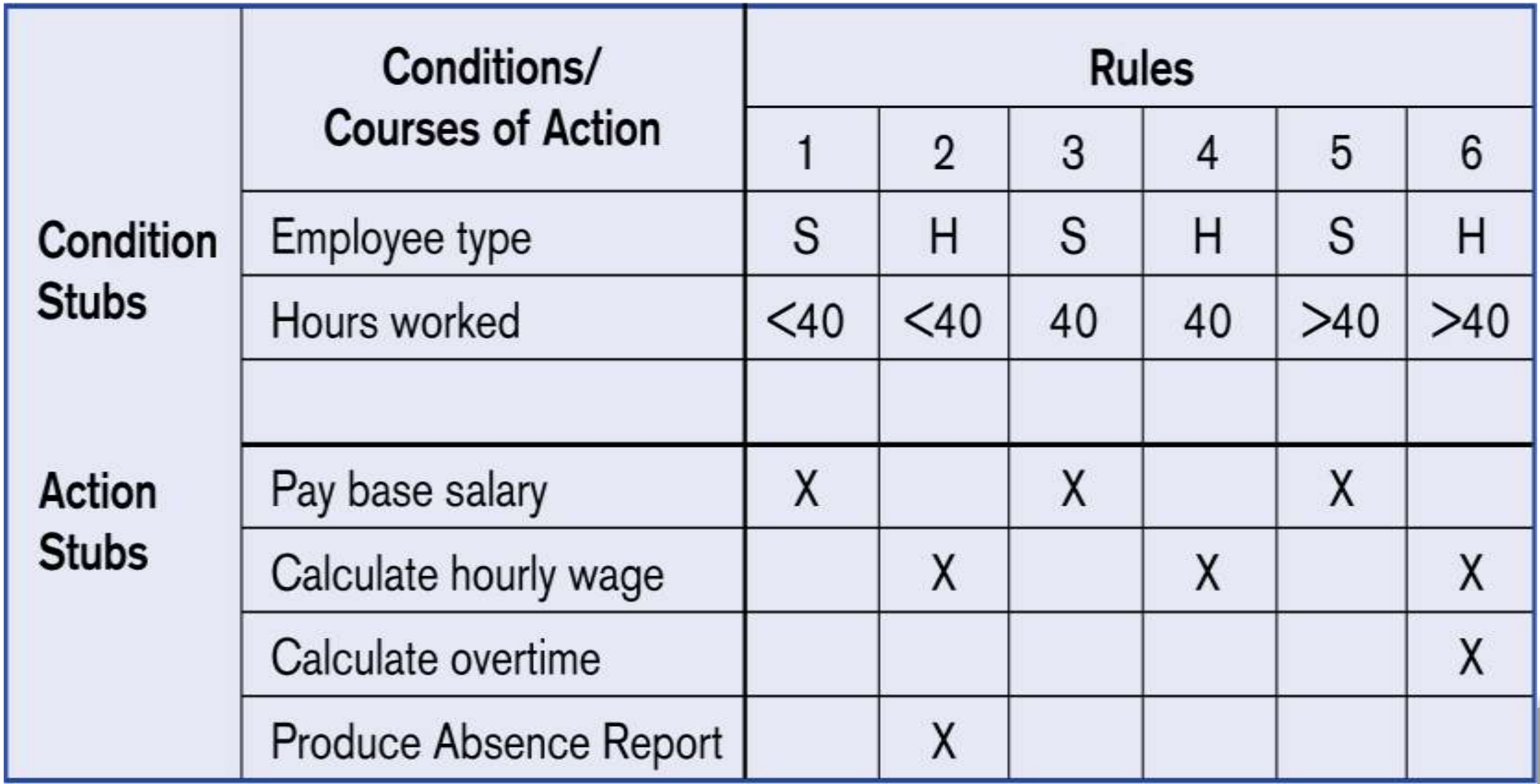
- Indifferent Condition
  - Condition whose value does not affect which action is taken for two or more rules
- Standard procedure for creating decision tables
  - Name the condition and values each condition can assume
  - Name all possible actions that can occur
  - List all rules
  - Define the actions for each rule
  - Simplify the table

Modeling Logic with Decision Tables

- Indifferent Condition
  - Condition whose value does not affect which action is taken for two or more rules
- Standard procedure for creating decision tables
  - Name the condition and values each condition can assume
  - Name all possible actions that can occur
  - List all rules
  - Define the actions for each rule
  - Simplify the table

Complete decision table for payroll system example

Complete decision table for payroll system example



	Conditions/ Courses of Action	Rules					
		1	2	3	4	5	6
Condition Stubs	Employee type	S	H	S	H	S	H
	Hours worked	<40	<40	40	40	>40	>40
Action Stubs	Pay base salary	X		X		X	
	Calculate hourly wage		X		X		X
	Calculate overtime						X
	Produce Absence Report		X				

Constructing a Decision Table

- PART 1. FRAME THE PROBLEM.
  - Identify the conditions (decision criteria).These are the factors that will influence the decision.
    - E.g., We want to know the total cost of a student’s tuition. What factors are important?
  - Identify the range of values for each condition or criteria.
    - E.g. What are they for each factor identified above?
  - Identify all possible actions that can occur.
    - E.g. What types of calculations would be necessary?

Constructing a Decision Table

- PART 2. CREATE THE TABLE.
  - Create a table with 4 quadrants.
    - Put the conditions in the upper left quadrant.One row per condition.
    - Put the actions in the lower left quadrant.One row per action.
  - List all possible rules.
    - Alternate values for first condition.Repeat for all values of second condition.Keep repeating this process for all conditions.
    - Put the rules in the upper right quadrant.
  - Enter actions for each rule
    - In the lower right quadrant, determine what, if any, appropriate actions should be taken for each rule.
  - Reduce table as necessary.

Example

- Calculate the total cost of your tuition this quarter.
  - What do you need to know?
    - Level. (Undergrad or graduate)
    - School. (CTI, Law, etc.)
    - Status. (Full or part time)
    - Number of hours
  - Actions?

Actions

- Actions?
  - Consider CTI only (to make the problem smaller):
    - U/G
      - Part Time (1 to 11 hrs.): $335.00/per hour
      - Full Time (12 to 18 hrs.): $17,820.00
      - * Credit hours over 18 are charged at the part-time rate
    - Graduate:
      - Part time (1 to 7 hrs.): $520.00/per hour
      - Full time (>= 8 hrs.): $520.00/per hour
- Create a decision table for this problem.In my solution I was able to reduce the number of rules from 16 to 4.

Boolean Algebra

A Boolean algebra consists of:

- a set B={0, 1},
- 2 binary operations on B (denoted by + & ×),
- a unary operation on B (denoted by '), such that :

$0+0=0$	$0 \times 0 = 0$
$1+0=1$	$0 \times 1 = 0$
$0+1=1$	$1 \times 0 = 0$
$1+1=1$	$1 \times 1 = 1$

$0'=1$ and $1'=0$.

Rules of a Boolean Algebra

The following axioms (‘rules’) are satisfied for all elements x, y& z of B:

(1) $x + y = y + x$ $x \times y = y \times x$(commutative axioms)

(2) $x + (y + z) = (x + y) + z$ (associative axioms)

$x \times (y \times z) = (x \times y) \times z$

(3) $x \times (y + z) = (x \times y) + (x \times z)$

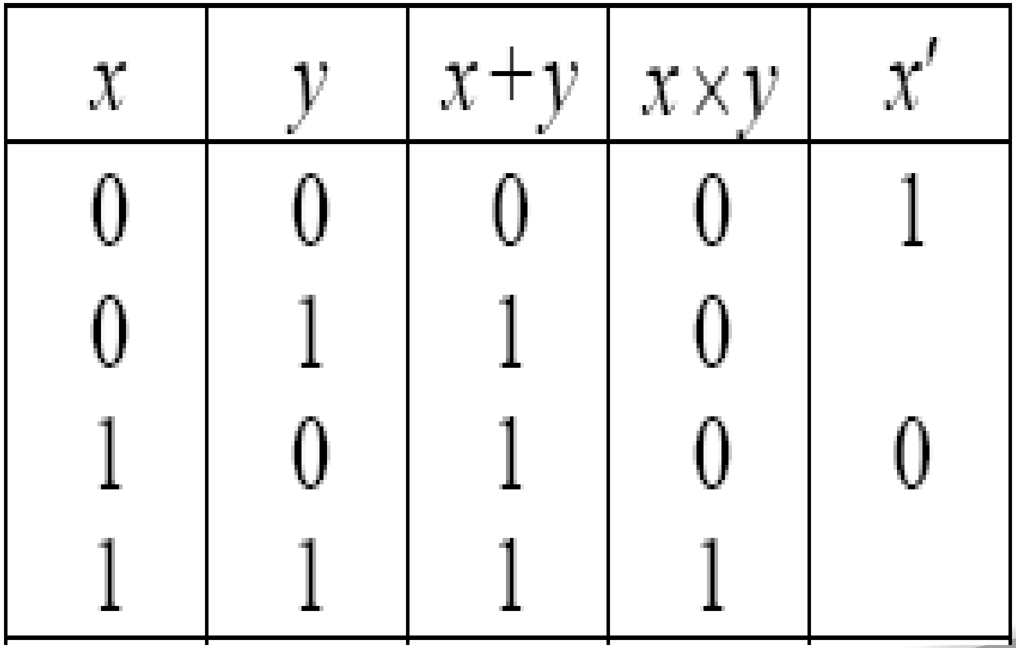
$x + (y \times z) = (x + y) \times (x + z)$ (distributive axioms)

(4) $x + 0 = x$ $x \times 1 = x$ (identity axioms)

(5) $x + x' = 1$ $x \times x' = 0$ (inverse axioms)

Boolean Notation

- This means that in effect we’ll be employing Boolean Algebra notation.
- The truth tables can be rewritten as



x	y	$x+y$	$x \times y$	x'
0	0	0	0	1
0	1	1	0	
1	0	1	0	0
1	1	1	1	

Notational Short-cuts

We will employ short-cuts in notation:

(1) In ‘multiplication’ we’ll omit the symbol ×, & write xy for x × y (just as in ordinary algebra)

(2) The associative law says that

$$x + (y + z) = (x + y) + z$$

So we’ll write this as simply $x + y + z$, because the brackets aren’t necessary.

Notational Short-cuts

Similarly, write the product of 3 terms as xyz

(3) In ordinary algebra, the expression

x + y × z means x + (y × z), because of the convention that multiplication takes precedence over addition.

e.g. x + yz means x + (y × z), and not (x + y) × z

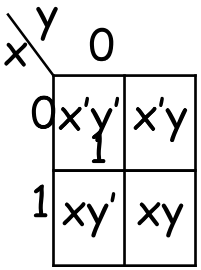
Similarly, ab + cd means (a × b) + (c × d)

Reducing Boolean Expressions

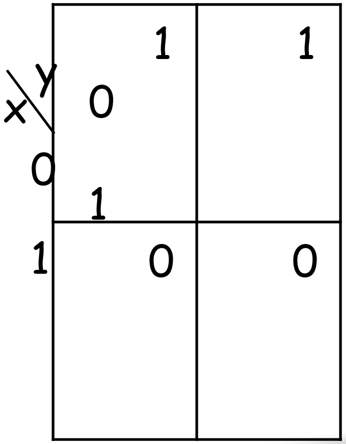
- Is this the smallest possible implementation of this expression? No! G = xyz + xyz’ + x’yz
- Use Boolean Algebra rules to reduce complexity while preserving functionality.
- Step 1: Use idempotent law (a + a = a). So xyz + xyz’ + x’yz = xyz + xyz + xyz’ + x’yz
- Step 2: Use distributive law a(b + c) = ab + ac. So xyz + xyz + xyz’ + x’yz = xy(z + z’) + yz(x + x’)
- Step 3: Use Inverse law (a + a’ = 1). So xy(z + z’) + yz(x + x’) = xy.1 + yz.1
- Step 4: Use Identity law (a . 1 = a). So xy + yz = xy.1 + yz.1 = xyz + xyz’ + x’yz

Karnaugh maps

- Alternate way of representing Boolean function
  - All rows of truth table represented with a square
  - Each square represents a minterm



x \ y	0	1
0	x'y'	x'y
1	xy'	xy



x \ y	0	1
0	1	1
1	0	0

x	y	F
0	0	1
0	1	1
1	0	0
1	1	0

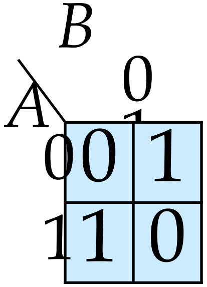
Karnaugh maps

- Easy to convert between truth table, K-map, and SOP.
  - Un optimized form: number of 1’s in K-map equals number of minterms (products) in SOP.
  - Optimized form: reduced number of minterms

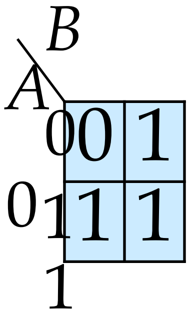
$$F(x,y) = x'y + x'y' = x'$$

Karnaugh Maps

- A Karnaugh map is a graphical tool for assisting in the general simplification procedure.
- Two variable maps.



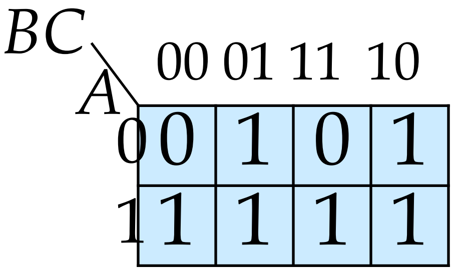
$F = AB' + A'B$



$F = AB + A'B + AB'$

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

- Three variable maps.



$F = AB'C' + AB'C + ABC + ABC' + A'B'C + A'BC'$

Rules for K-Maps

- We can reduce functions by circling 1’s in the K-map.
- Each circle represents minterm reduction.
- Following circling, we can deduce minimized and-or form.

$F(x,y) = x'y + x'y' = x'$

Rules to consider

- Every cell containing a 1 must be included at least once.
- The largest possible “power of 2 rectangle” must be enclosed

Rules for K-Maps

Rules to consider

1. Every cell containing a 1 must be included at least once.
2. The largest possible “power of 2 rectangle” must be enclosed.

Specifications

1. Rewrite the specifications using consistent terminology.
2. Identify the predicates on which the cases are based. Name them with suitable letters, such as A, B, C.
3. Rewrite the specification in English that uses only the logical connectives AND, OR, and NOT, however stilted it may seem.
4. Convert the rewritten specifications into an equivalent set of Boolean expressions.
5. Identify the default action and cases, if any are specified.

Specifications

6. Enter the Boolean expressions in a KV chart and check for consistency. If the specifications are consistent, there will be no except for the cases that result in multiple actions.
7. Enter the default cases, and check for consistency.
8. If all boxes are covered, the specification is complete.
9. If the specification is incomplete or inconsistent, translate the corresponding boxes of the KV chart back into English and get a clarification, explanation, or revision.
10. If the default cases were not specified explicitly, translate the default cases back into English and get a confirmation.

UNIT-IV
PATH PRODUCTS

Course Learning Outcomes

CLOs	Course Learning Outcomes
CLO 14	State Path products and path expression, different laws used in path testing.
CLO 15	Demonstrate Reduction procedure and applications.
CLO 16	Explain about Regular expressions
CLO 17	Demonstrate about Flow anomaly detection

Purpose & Applications

1. Check for data flow anomalies.
2. Regular expressions are applied to problems in test design & debugging.
3. Electronics engineers use flow graphs to design & analyze circuits & logic designers. Software development, testing & debugging tools use flow graph analysis tools & techniques.
4. These are helpful for test tool builders.

Path Products & expressions

Motivation:

1. Flow graph is an abstract representation of a program.
2. A question on a program can be mapped on to an equivalent question on an appropriate flow graph.
3. It will be a foundation for syntax testing & state testing

Path Products & expressions

Path Expression:

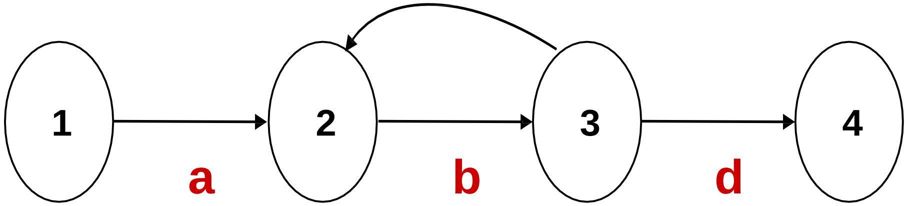
An algebraic representation of sets of paths in a flow graph.

Regular Expression:

Path expressions converted by using arithmetic laws & weights into an algebraic function.

Path Products & expressions

- Annotate each link with a name.
- The pathname as you traverse a path (segment) expressed as concatenation of the link names is the
- path product.
- Examples of path products between 1 & 4 are:

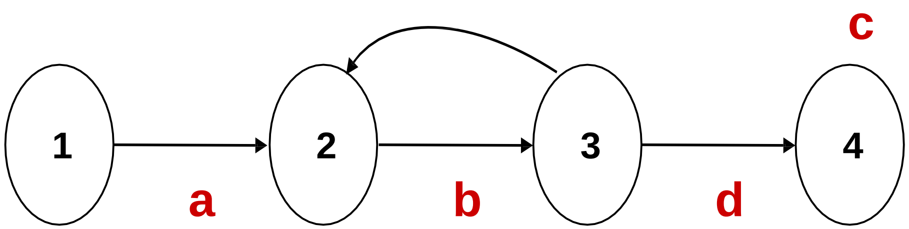


a b d a b c b a b c b c b d ....

Path Products & expressions – Path Expression

Path Expression

Simply: Derive using path products.

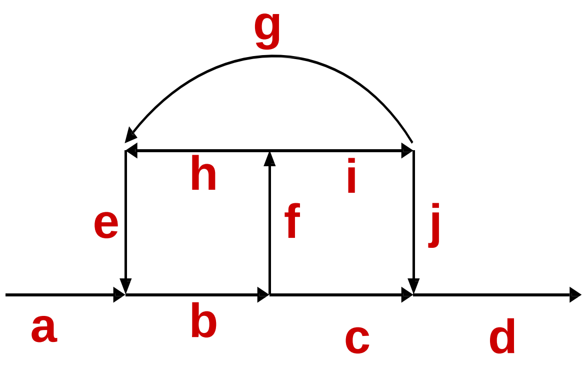


Example:

{ a b d, a b c b d, a b c b c b d , ….. } abd + abcbd + abcbcbd + ….

Path Products & expressions – Path Expression

Example:



{ abcd , abfhebcd , abfigebcd , abfijd }

abcd + abfhebcd + abfigebcd + abfijd

Path Products & expressions – Path Expression

Path name for two successive path segments is the concatenation of their path products.

X = abc	**Y = def**	**XY = abcdef**
a X = aabc	**X a = abca**	**XaX = abcaabc**
X = ab + cd	**Y = ef + gh**	**XY = abef + abgh + cdef + cdgh**

Path Products & expressions – Path Product

Path Product

- Not Commutative:

  $XY \neq YX$ in general

- Associative

A (BC) = (AB) C = ABC

Path Products & expressions – Path Product

Denotes a set of paths in parallel between two nodes.

- Commutative

X + Y = Y + X : **Rule 2**

- Associative

(X + Y) + Z =X + (Y + Z)= X + Y + Z : **Rule 3**

- Distributive

A (B + C) = A B + A C (A + B) C
= A C + B C : **Rule 4**

Path Products & expressions

- Absorption

: **Rule 5**

X + X = X
X + any subset of X = X

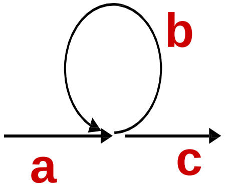
X = a + bc + abcd

X + a=X+bc + abcd=X

Path Products & expressions

- **Loop:**

An infinite set of parallel paths.



$$b^* = b^0 + b^1 + b^2 + b^3 + \ldots\ldots$$

$$X^* = X^0 + X^1 + X^2 + X^3 + \ldots\ldots$$

$$X^+ = X^1 + X^2 + X^3 + \ldots\ldots$$

- $X\ X^* = X^*\ X = X^+$

♦ a

$a = a^*\ a = a^+$

$$X^n = X^0 + X^1 + X^2 + X^3 + \ldots\ldots$$

More Rules…

$x^m + x^n = x^n$ if $n \geq m$: **Rule 6**

$= x^m$ if $n < m$

$x^m x^n = x^{m+n}$: **Rule 7**

$x^n x^* = x^* x^n = x^*$: **Rule 8**

$x^n x^+ = x^+ x^n = x^+$: **Rule 9**

$x^* x^+ = x^+ x^* = x^+$: **Rule 10**

Identity Elements

Identity Elements ..

1 : Path of Zero Length

$1 + 1 = 1$: **Rule 11**

$1X = X1 = X$: **Rule 12**

$1^n = 1^n = 1^* = 1^+ = 1$: **Rule 13**

$1^+ + 1 = 1^* = 1$: **Rule 14**

Identity Elements

Identity Elements ..

0 : empty set of paths

$X + 0 = 0 + X = X$: **Rule 15**

$X0 = 0X = 0$: **Rule 16**

$0^* = 1 + 0 + 0^2 + 0^3 + \ldots = 1$: **Rule 17**

Initialization Steps

- Combine all serial links by multiplying their path expressions.
- Combine all parallel links by adding their path expressions.
- Remove all self-loops-replace with links of the form X*
- **Reduction Procedure:**

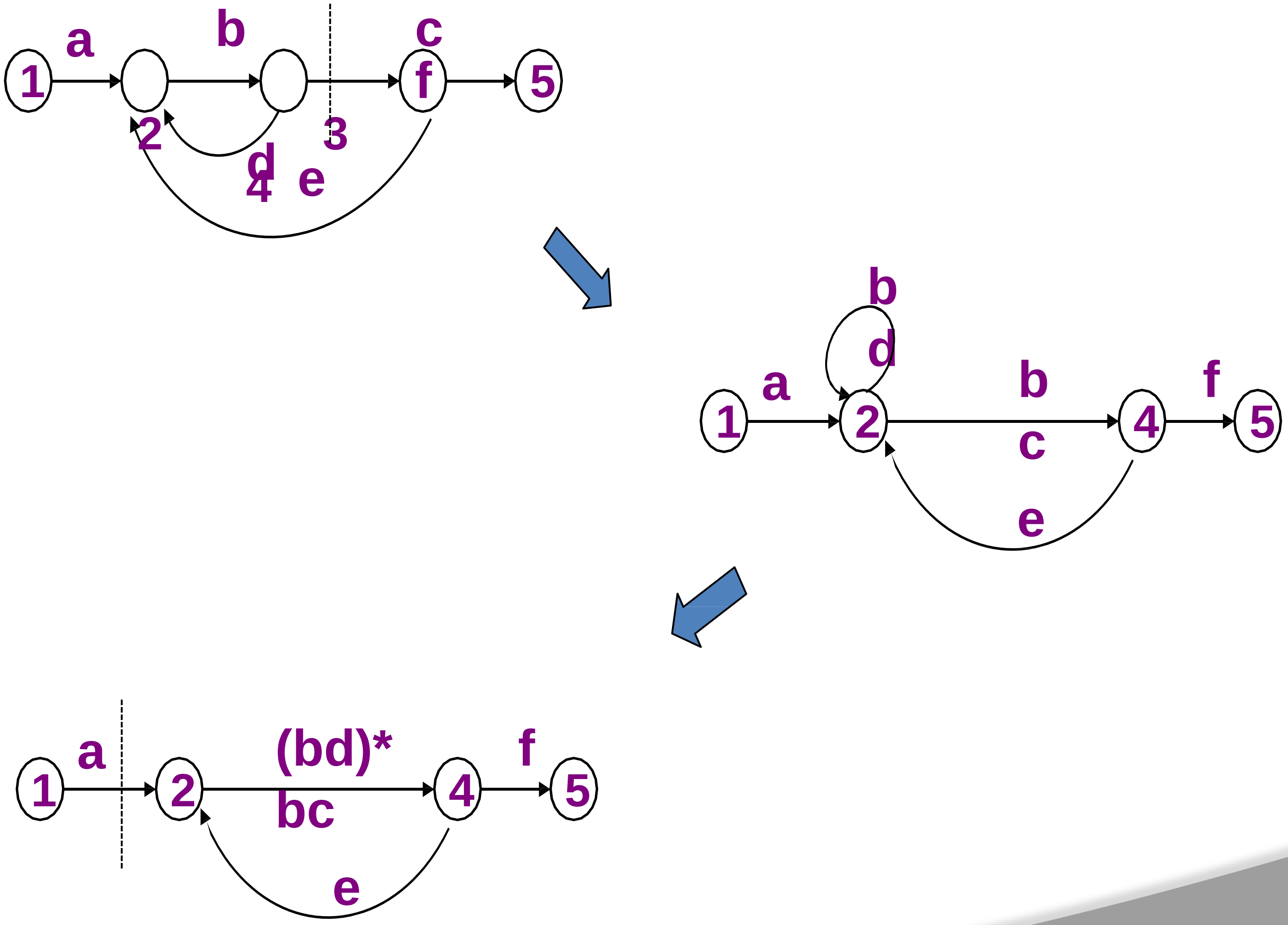
  To convert a flow graph into a path expression that denotes the set of all entry/exit paths.

  Node by Node Reduction Procedure
- **Cross-Term Step(Step 4 of the Algorithm)**
- Fundamental step.

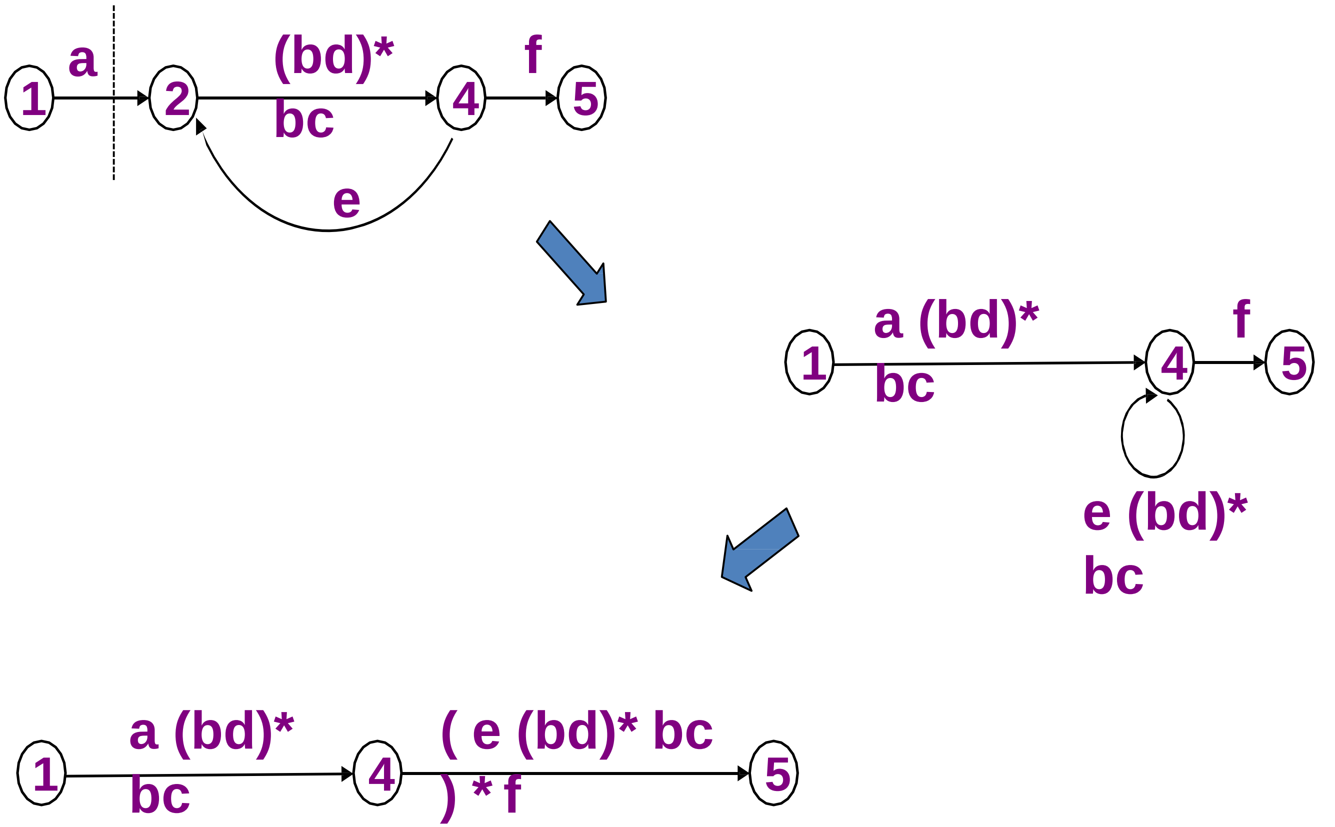
  Removes nodes one by one till there's one entry & one exit node.
- Replace the node by path products of all in-links with all out-links and interconnecting its immediate neighbors.

Processing of Loop Terms

Processing of Loop Terms:

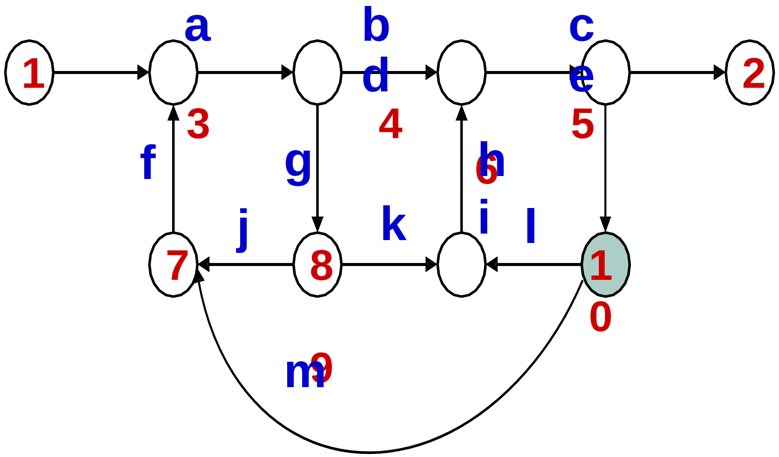


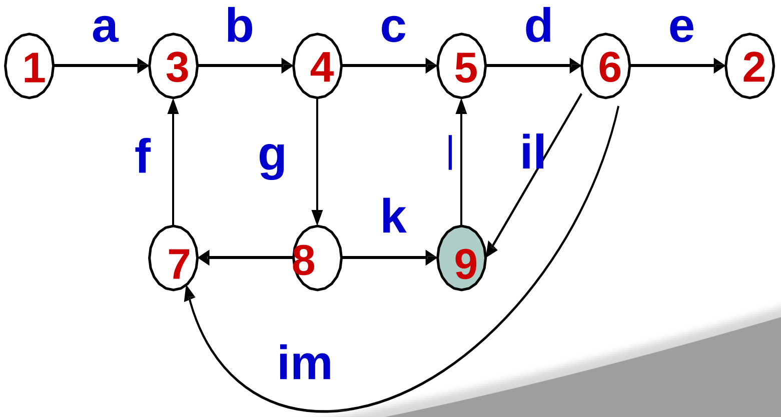
Processing of Loop Terms



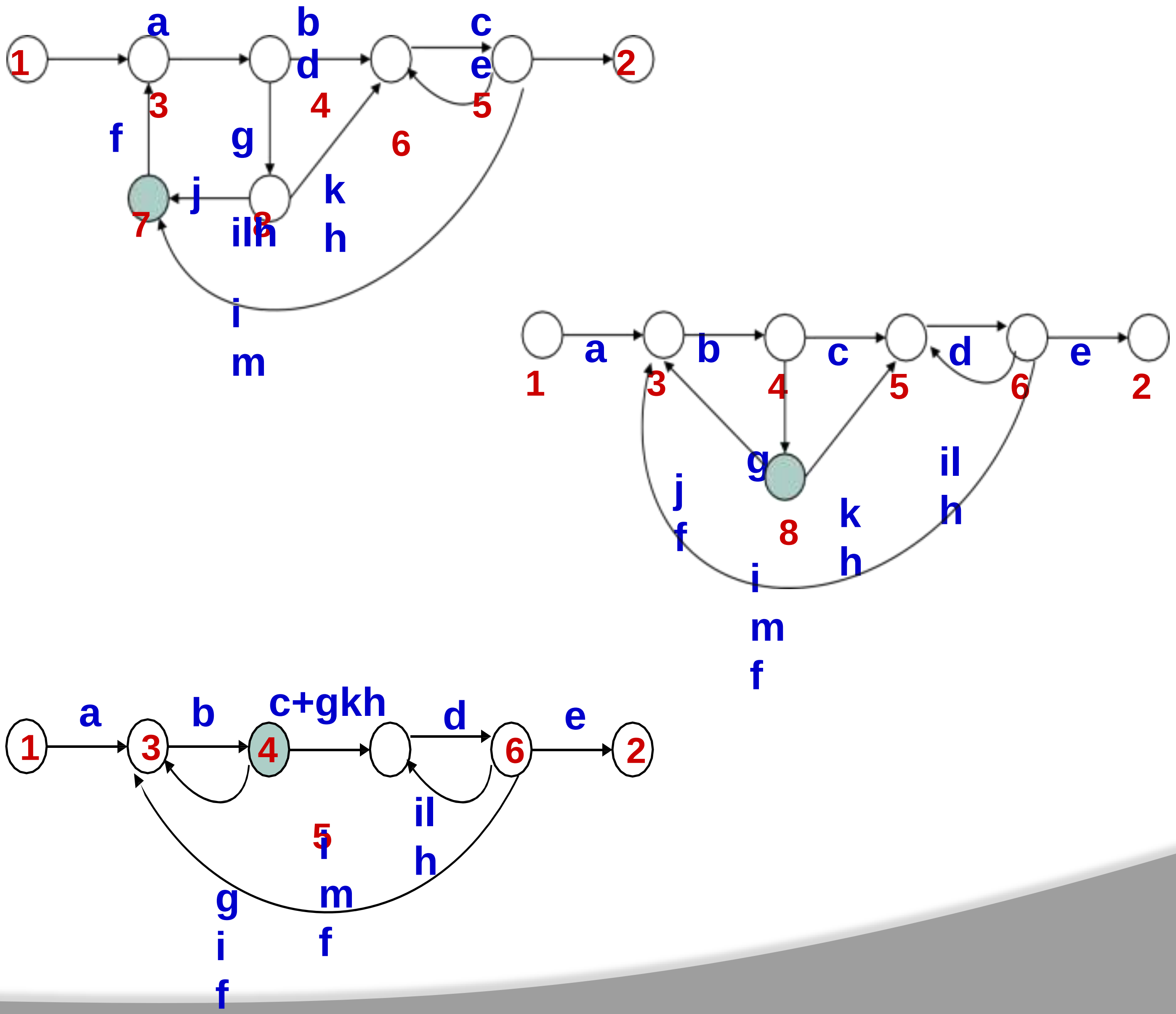
Example

Example:

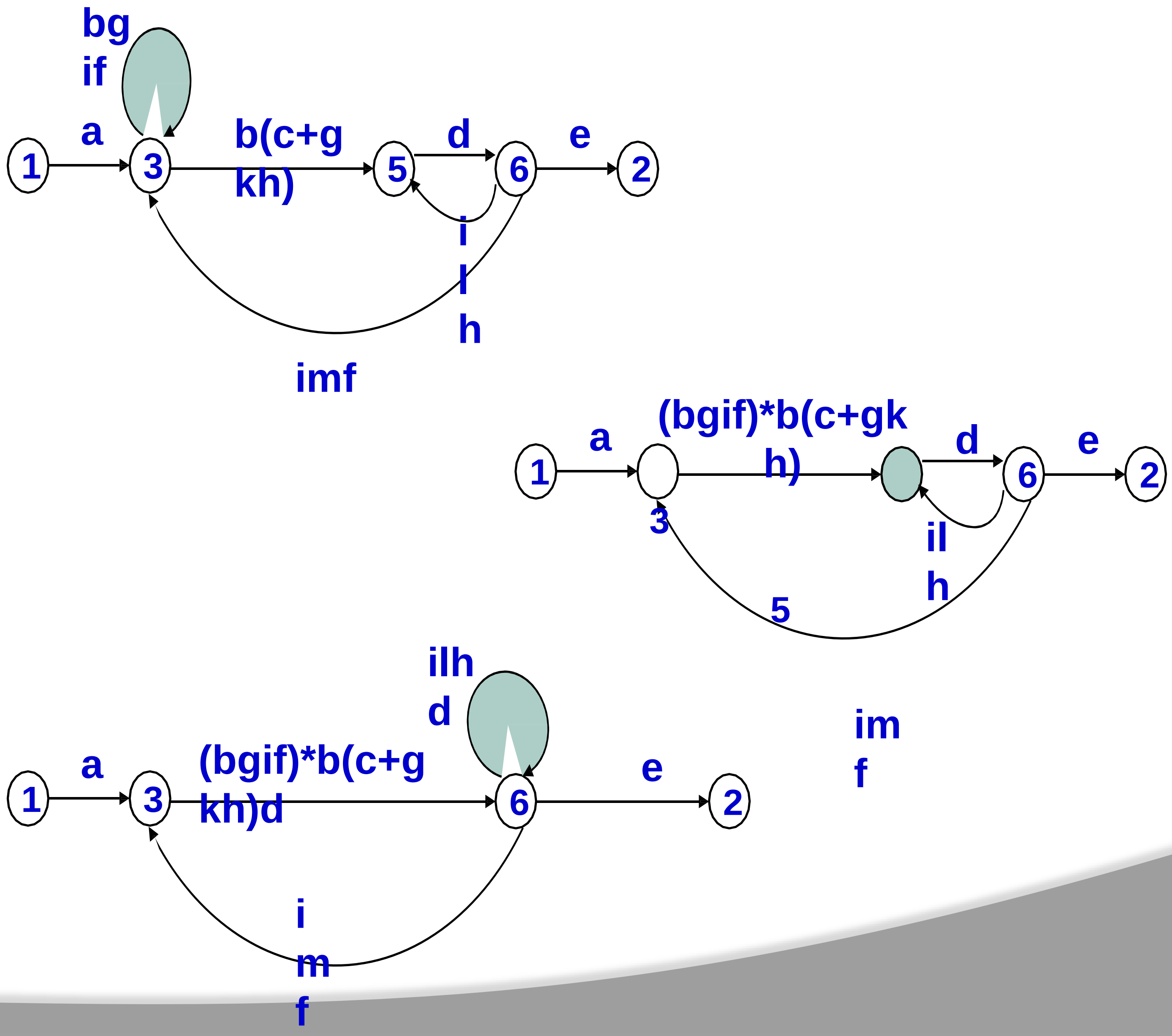




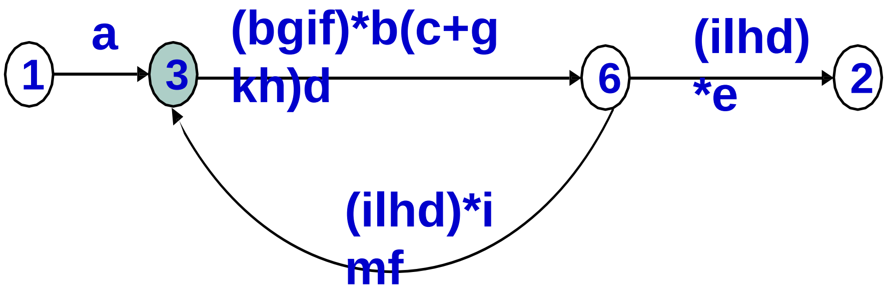
Processing of Loop Terms

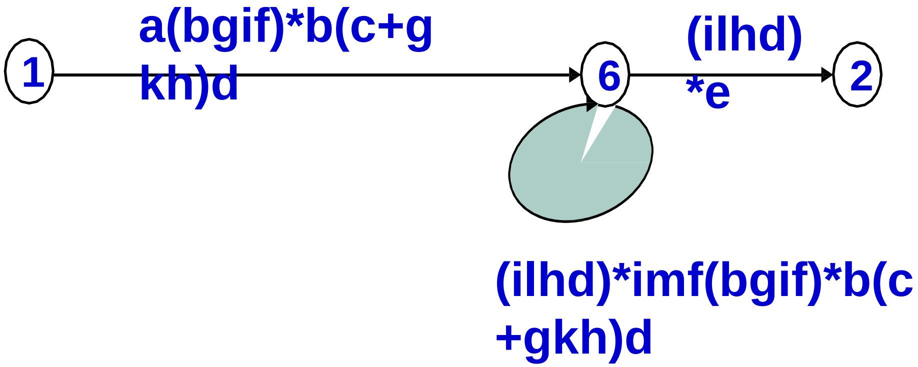


Processing of Loop Terms

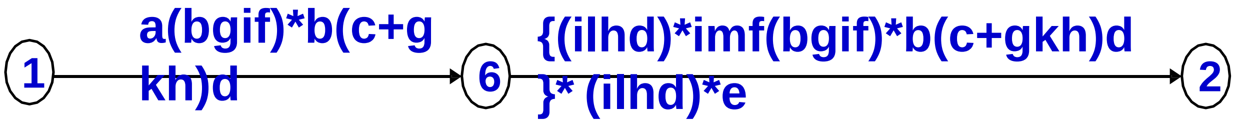


Processing of Loop Terms





Processing of Loop Terms



Flow Graph Path Expression :

a(bgif)*b(c+gkh)d

{(ilhd)*imf(bgif)*b(c+gkh)d}* (ilhd)*e

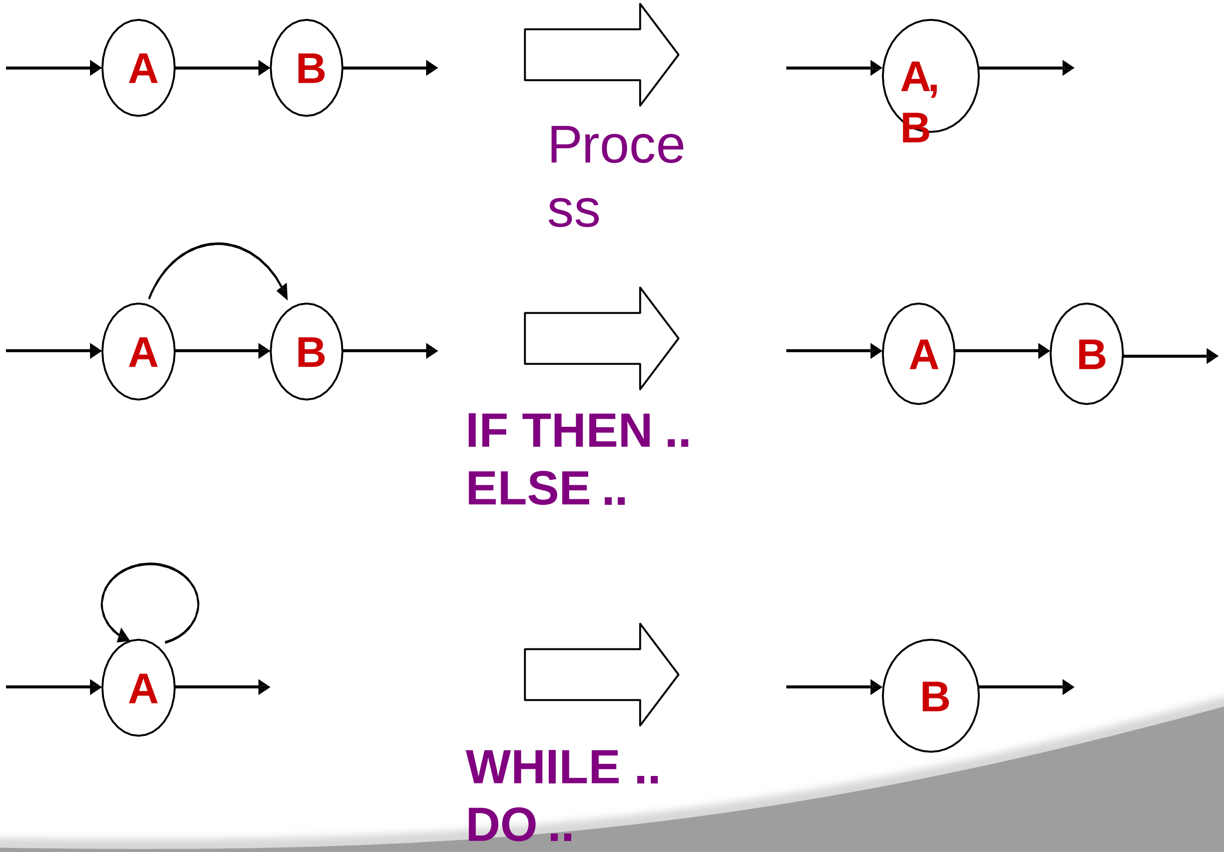
Path Products & expressions

$$
\begin{aligned}
(A + B)^* &= (A^* + B^*)^* && : I1 \\
&= (A^* \; B^*)^* && : I2 \\
&= (A^* B)^* \; A^* && : I3 \\
&= (B^* A)^* \; B^* && : I4 \\
&= (A^* B + A)^* && : I5 \\
&= (B^* A + B)^* && : I6 \\
(A + B + C + . . .)^* &= (A^* + B^* + C^* + . . .)^* && : I7 \\
&= (A^* B^* C^* . . .)^* && : I8
\end{aligned}
$$

Derived by removing nodes in different orders & applying the series-parallel-loop rules.

Path Products & expressions

Reducible to a single link by successive application of the transformations shown below.

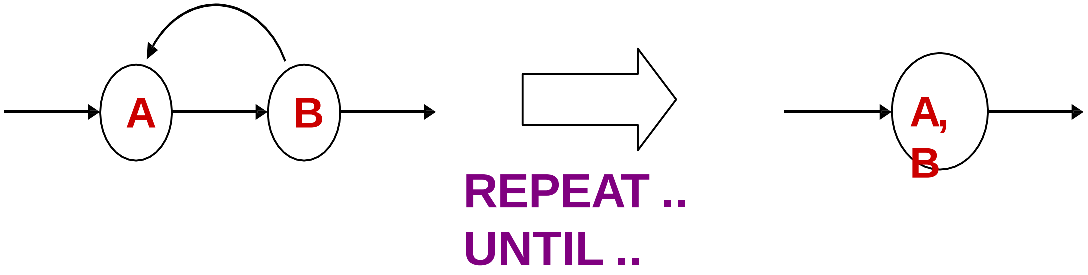


Process

IF THEN .. ELSE ..

WHILE .. DO ..

Path Products & expressions

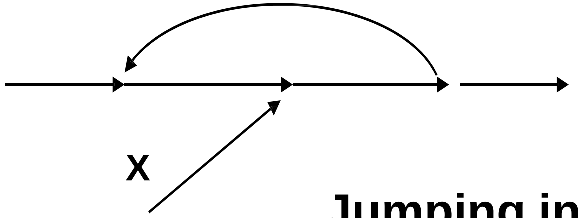


Properties:

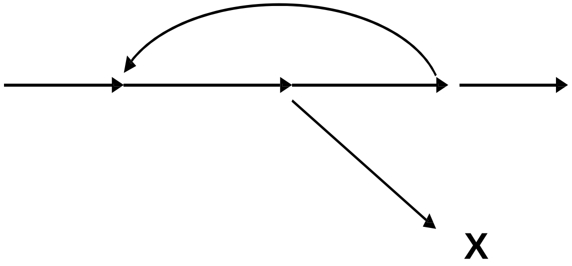
- No cross-term transformation.
- No GOTOs.
- No entry into or exit from the middle of a loop.

Unstructured Flow Graphs/Code

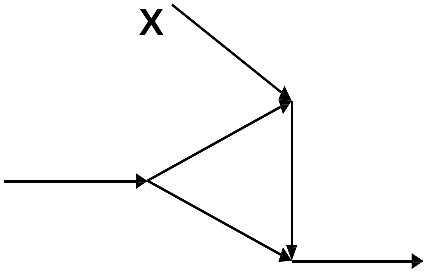
Some examples – unstructured flow graphs/code:



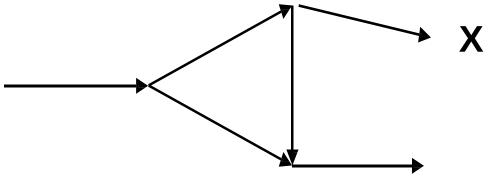
Jumping into loops



Jumping out of loops



Branching into Decisions



Branching out of Decisions

2000

IARE

EDUCATION FOR LIBERATION

UNIT-V
TRANSITION TESTING

Course Learning Outcomes

CLOs	Course Learning Outcomes
CLO 18	Explain state graphs and state testing
CLO 19	Demonstrate about the testability tips
CLO 20	Explain state behavior in state graphs

State graph

- A state graph is a graphical representation of the program (its FSM) in terms of states, transitions, inputs and outputs It has one start state and usually, an end/destination/exit state.
- Note => In the exam you may draw only 3 state graph for simplicity.
- State graph in the above example is used to model the behavior of the program that recognizes a string occurrence at the input. It can be used to design, implement and the testing of the program

A Property of a state graph

- State graphs are not dependent on time or temporal behavior or the program. (Temporal behavior is represented by some time sequence diagrams etc..) The system changes state only when an event (with an input sequence occurs or an epsilon symbol representing no event appears at the input of a transition).
- State graphs (FSM) are implemented as state tables which are represented in software with definite data structures and associated operations.

State table

Very big state graphs are difficult to follow as the diagrams get complicated and links get entwined. It is more convenient to represent the state graph as a table called state table or state transition table.

Each row represents the transitions from the originating state. There is one column for each input symbol (erroneous input or normal input). The entry in the table represents the new state to which the system transits to on this transition and the output it prints on the target printer device or on the output side.

A Property of a state graph

- State graphs are not dependent on time or temporal behavior or the program. (Temporal behavior is represented by some time sequence diagrams etc..) The system changes state only when an event (with an input sequence occurs or an epsilon symbol representing no event appears at the input of a transition).
- State graphs (FSM) are implemented as state tables which are represented in software with definite data structures and associated operations.

Software implementation of state table

There are four tables that are needed.

1. A table or a 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. (TRANISITION_TABLE)
3. A table or case statement that specifies the output (or output code) associated with every state-input combination (OUTPUT_TABLE)
4. A table that stores the present state of each device or process or component or system that uses the same state table. (DEVICE_TABLE)

Software implementation of state table

1. The present state is fetched from the memory (from DEVICE_TABLE).
2. the present input value (symbol) is fetched from the environment. It is encoded if it is non-numerical by using the INPUT_CODE_TABLE.
3. The present state and the input code are concatenated to give a pointer (row,column) into a cell of the TRANSITION_TABLE.
4. The OUTPUT_TABLE contains a pointer to the routine to be executed when that state-input combination occurs.
5. The same pointer value is used to fetch the new state value, which is then stored in DEVICE_TABLE

Principles of state testing

Asit is not possible to test every path the stategraph, use the notion of coverage. We assume that the graph is strongly connected.

1.It is not useful or practical to plan an entire grand tour of the states for testing initially as it does not work out due to possibility of bugs.

2. During the maintenance phase only few transitions and states need to be tested which are affected.

3.For very long test input symbol sequences it is difficult to test the system.

Uses/Advantages of state testing

- State testing can find bugs which are not possible to be found with other types of testing. Normally most of systems can be modeled as state graphs.
- It can find if the specifications are complete and ambiguous. This is seen clearly if the state table is filled with multiple entries in some cells or some cells are empty. IT can also tell if some default transitions or transitions on erroneous inputs are missing.
- State testing can identify the system's seemingly impossible states and checks if there are transitions from these states to other states are defined in the specifications or not. That is, the error recovery processes are defined for such impossible states.

Uses/Advantages of state testing

- State testing can simplify the design of the program / system by identifying some equivalent states and then merging these states. Also, state testing using FSM can allow design/test design in a hierarchical manner if the state tables are so designed.
- The state testing can identify if the system reaches a dead state / unreachable states and allow one to correct the program specifications and make the system complete, robust and consistent.
- The bugs in the functional behavior can be caught earlier and will be less expensive if state testing is done earlier than the structural (white box) testing.

Disadvantages of state testing

- State transition testing does not guarantee the complete testing of the program. How much of testing with how many combinations of input symbol sequences constitutes sufficient number of tests is not clear/known. It is not practical to test thru every path in the state graph.
- Functional behavior is tested and structural bugs are not tested for. There could be difficulty if those bugs are found and mixed up with behavioral bugs.
- We assume that the state graph is strongly connected that is every node is connected to every other node thru a path.

Application areas for state testing

Any program that processes input as a sequence of events/symbols and produces output such as detection of specified input symbol combinations, sequential format verification, parsing, etc.

Communication Protocols: processing depends on current state of the protocol stack, OS, network environment and the message received

- concurrent systems,
- system failures and the corresponding recovery systems,
- distributed data bases,
- device drivers – processing depends on the state of the device

Application areas for state testing

- Operation requested by the user or system
  - –multi-tasking systems,
  - –human computer interactive systems,
  - –resource management systems – processing depends on availability
- levels and states of resources
  - –Processing of hierarchical pop-up menus on windows based software systems – letting the user navigate thru menus
  - –the web based application software, embedded systems and other systems also use this model for design and testing.

Good-state graphs and Bad state graph

The principles of judging whether a state graph is good or bad are:

- the total number of states is equal to the product of possibilities of factors that make up the state. (ie., number of permutations of all values of all attributes/properties of the system/component)
- For every state and every input there is exactly one transition specified to exactly one, possibly the same, state.
- For every transition there is one output action specified. That output could be trivial (epsilon), but at least one output does some thing sensible.
- For every state there is a sequence of inputs that drives the system to the starting (same) state.
- A good state graph has at least two input symbols. With one symbol only a limited number of useful graphs are possible.

Principles of State Testing

- The strategy for state testing is analogous to that used for path testing flow graphs.
- Just as it's impractical to go through every possible path in a flow graph, it's impractical to go through every path in a state graph.
- The notion of coverage is identical to that used for flow graphs. Even though more state testing is done as a single case in a grand tour, it's   impractical to do it that way for several reasons

References

1. Boris Beizer “Software Testing Techniques“, Dreamtech Press 2nd Edition, 2003.

2. P C Jorgenson “Software Testing : Craft of Software Testing “, Auerback Publications 3rd Edition, 2013.