SOFTWARE TESTING METHODOLOGIES

Purpose of testing

- Productivity and Quality in Software
- Goal for Testing
  1. Bug prevention
  2. Bug Discovery.

Phase in Tester Mental Life

- Phase 0: There's no difference between testing and debugging
- Phase 1: The purpose of testing is to show that the software works
- Phase 2: The purpose of testing is to show that the software doesn't works
- Phase 3: Reduce the risk
- Phase 4: 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 modelling. 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 a important one to get the system without any bugs

Testing is not Everything……

- Inspection Methods
- Design Style
- Static Analysis Methods
- Languages
- Design Methodologies and Development Environment

1. Pesticide Paradox
2. Complexity Barrier

Some Dichotomies

- Testing Vs Debugging
- Function Vs Structure
- Black Box Testing
- White box testing
- Designer Vs the Tester
- Modularity Vs Efficiency
- Small Vs Large
- The Builder Vs Buyer

A Model for Testing

The Project

- Application
- Staff
- Schedule
- Specification
- Acceptance test
- Personnel
- Standards
- Objectives
- Source
- History

Environment

- Hardware and software required to make it run.

Bugs

- ***Benign Bug Hypothesis***
- ***Bug Locality Hypothesis***
- ***Control Bug Dominance***
- ***Code/Data Separation***
- ***Lingua Salvator Est***
- ***Corrections Abide***
- ***Silver Bullets***
- ***Sadism Suffices***
- ***Angelic Testers***

Testing and Levels

- Unit Testing
- Component Testing
- Integration Testing
- System Testing

Playing pool

- Kiddie Testing
- Real Testing

Oracle

- 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
- The most common oracle is input/output oracle.

Sources of Oracles

- Kiddie Testing
- Regression test suites
- Purchased suites and oracles
- Existing programs (Hosting)

Complete Testing possible?

- Functional Testing
- Structural Testing
- Correctness Proof

- If the objective of testing were to *prove that a program is free of bugs, then testing not only would be* practically impossible

The Taxonomy of Bugs

Consequences of Bugs

- Frequency
- Correction Cost(discovery & correction)
- Installation Cost
- Consequences

Importance($)= frequency * (correctionCost+installationCost+ConsequenceCost)

How Bug Affect Us- Consequences

- Mild
- Moderate
- Annoying
- Disturbing
- Serious
- Very Serious
- Extreme
- Intolerable
- Catastrophic
- Infectious

Nightmare list and when to stop testing?

- List your worst software nightmares. State them in terms of the symptoms they produce and how your user will react to those symptoms.
- Convert the consequences of each nightmare into a cost. Usually, this is a labor cost for correcting the nightmare.
- Order the list from the costliest to the cheapest and then discard the low-concern nightmares with which you can live.
- 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.

- 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.
- Rank the bug types in order of decreasing importance to you.
- 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.
- If a test is passed, then some nightmares or parts of them go away. If a test is failed, then a nightmare is possible, but upon correcting the bug, it too goes away. Testing, then, gives you information you can use to revise your estimated nightmare probabilities.
- Stop testing when the probability of all nightmares has been shown to be inconsequential as a result of hard evidence produced by testing

A Taxonomy for Bugs

- Requirements, Features and Functionality Bugs.
- Specification and feature bug remedies.

Testing Techniques

Structural Bugs

1. Control and sequence bugs
2. Logic Bugs
3. Processing Bugs
4. Initialization Bugs

Data flow Bugs and Anamolies

- Data Bugs
- Dynamic Vs Static
- Coding Bugs

Interface, Integration and System Bugs

- External Interface
- Internal Interface
- Hardware Architecture
- Operating System
- Software Architecture (Interactive)
- Control and Sequence Bugs
- Integration Bugs
- System Bugs

Test and Test Design Bugs

- Testing
- Testing Criteria

Remedies

1. Test Debugging
2. Test Quality Assurance
3. Test Execution Automation
4. Test Design Automation

Unit-I

Sub Topic No's	Sub Topic name	Lecturer No	Slide No's
1	Introduction	L1	3
2	Concepts of path testing	L2	12
3	Predicates and Path predicates	L3	32
4	Achievable paths	L4	41
5	Path Sensitizing	L5	42
6	Path Instrumentation	L6	50
7	Applications of path testing	L7	60

Introduction

Path Testing Definition

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

Path Testing contd..

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

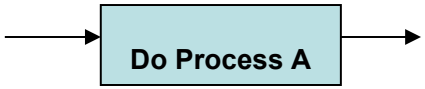
Observations

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

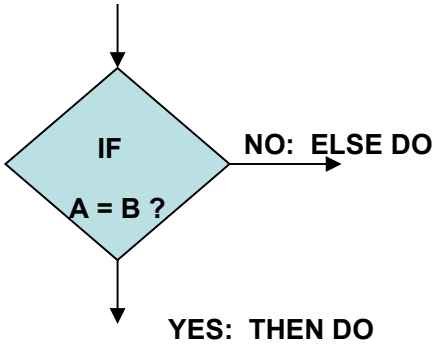
Control Flow Graph

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

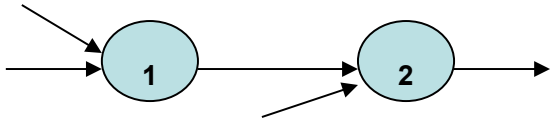
Process Block



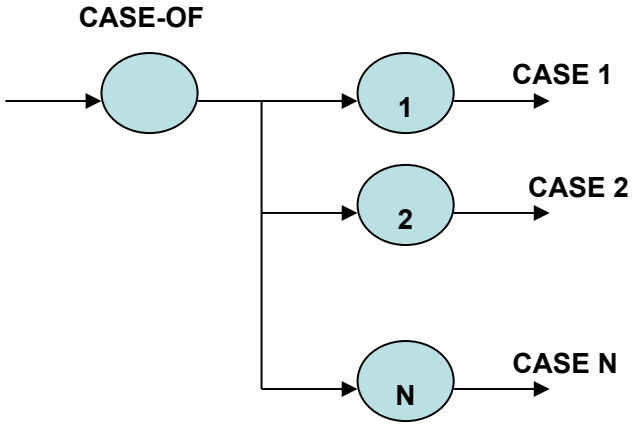
Decisions



Junctions



Case Statement



Control Flow Graph Elements

Control Flow Graph Elements:

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.
- Examples: In assembly language: jump addresses table, Multiple GOTOs, Case/Switch

- For test design, Case statement and decision are similar.

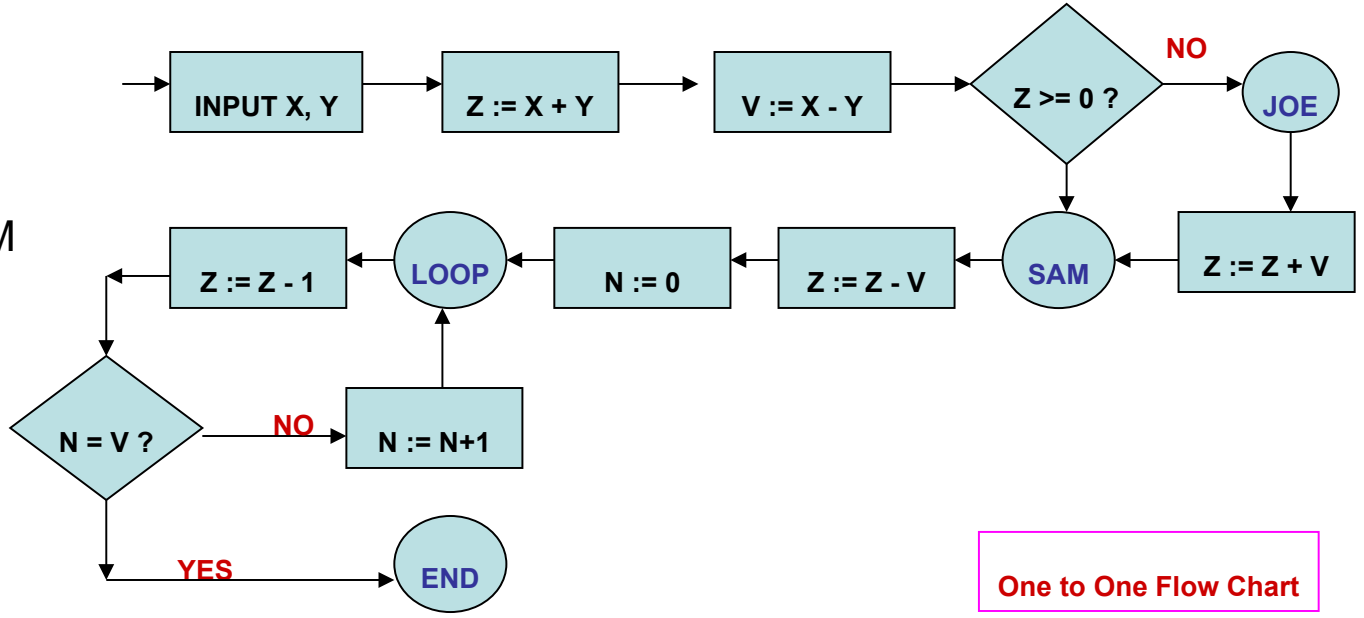
Control Flow Graph Vs Flow Charts

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

Creation of Control Flow Graph from a program

- 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**

```
        INPUT X, Y
        Z := X + Y
        V := X - Y
        IF Z >= 0  GOTO SAM
JOE:    Z := Z + V
SAM:    Z := Z - V
        FOR N = 0 TO V
        Z := Z - 1
        NEXT N
        END
```

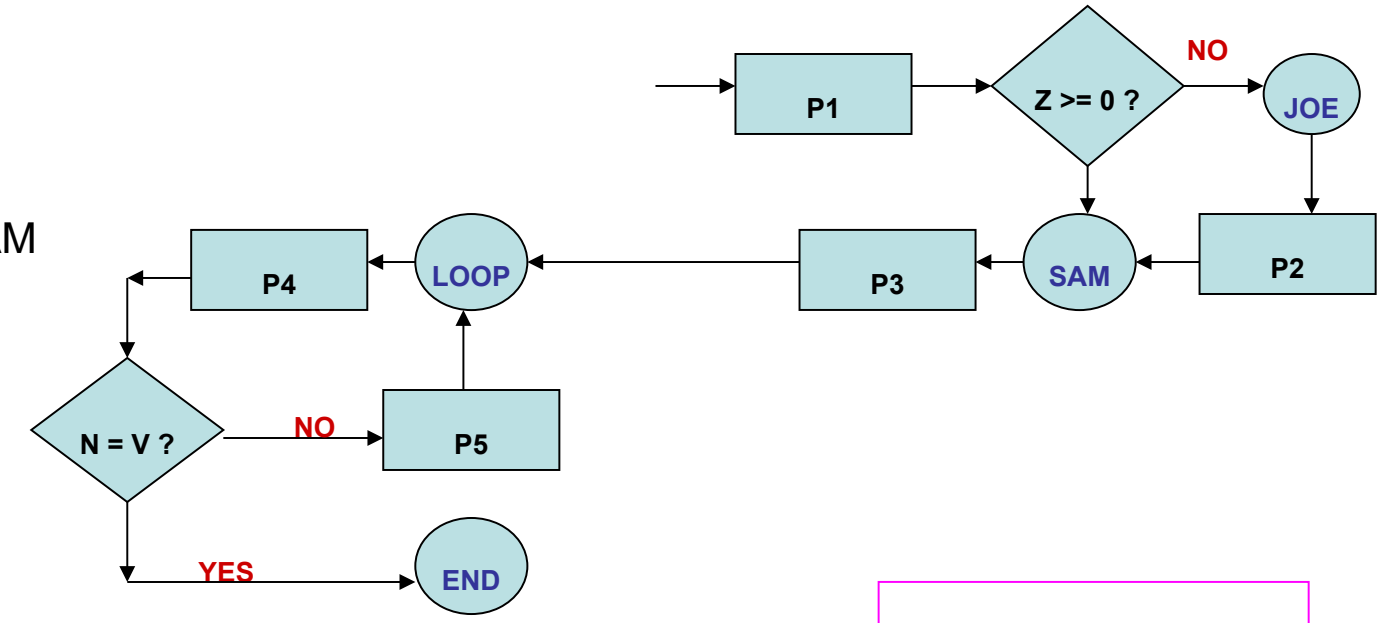


One to One Flow Chart

Creation of Control Flow Graph from a program

- 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**

```
        INPUT X, Y
        Z := X + Y
        V := X - Y
        IF Z >= 0  GOTO SAM
JOE:    Z := Z + V
SAM:    Z := Z - V
        FOR N = 0 TO V
        Z := Z - 1
        NEXT N
        END
```

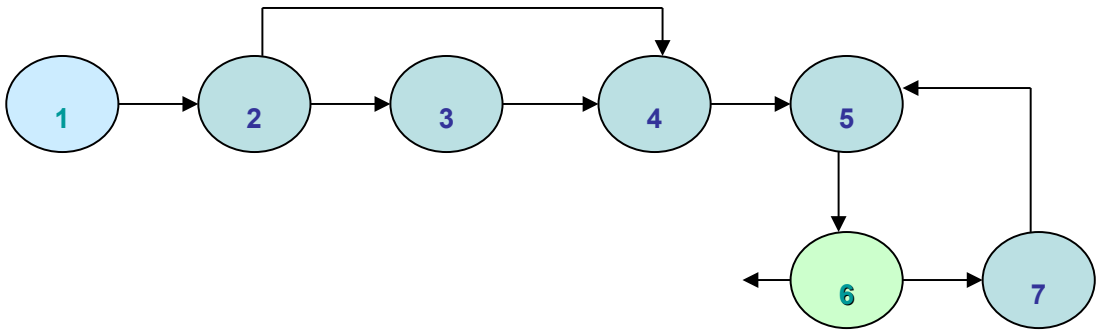


Simplified Flow Graph

Creation of Control Flow Graph from a program

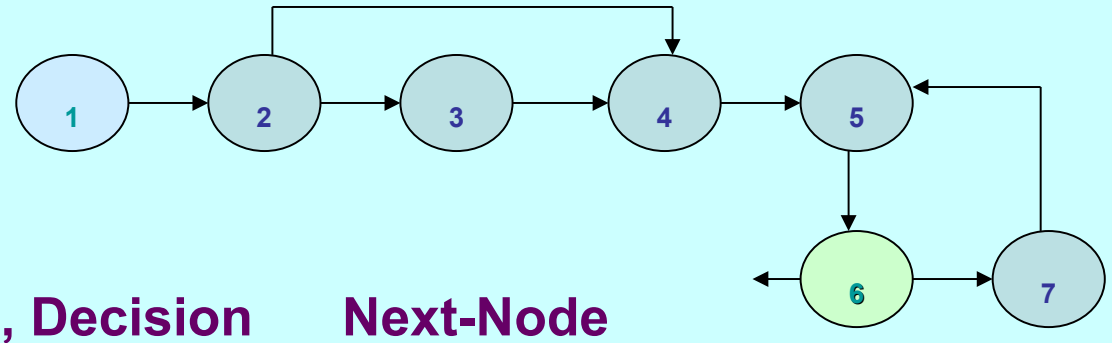
- One statement to one element translation to get a Classical Flow chart
- Add additional labels as needed
- Merge process steps
- A process box is implied on every junction and decision
- Remove External Labels
- Represent the contents of elements by numbers.
- We have now **Nodes** and **Links**

```
        INPUT X, Y
        Z := X + Y
        V := X - Y
        IF Z >= 0  GOTO SAM
JOE:    Z := Z + V
SAM:    Z := Z - V
        FOR N = 0 TO V
        Z := Z - 1
        NEXT N
        END
```



Simplified 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

Path Testing Concepts

1. **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.

   Path segment is a succession of consecutive links that belongs to the same path. (3,4,5)

   Length of a path is measured by # of links in the path or # of nodes traversed.

   Name of a path is the set of the names of the nodes along the path. (1,2,3 4,5, 6)
   (1,2,3,4, 5,6,7, 5,6,7, 5,6)

   Path-Testing Path is an “entry to exit” path through a processing block.

Path Testing Concepts..

2. 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.

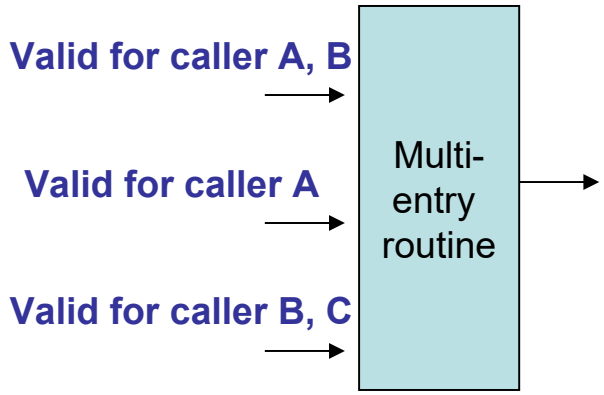
Tools could generate test cases.

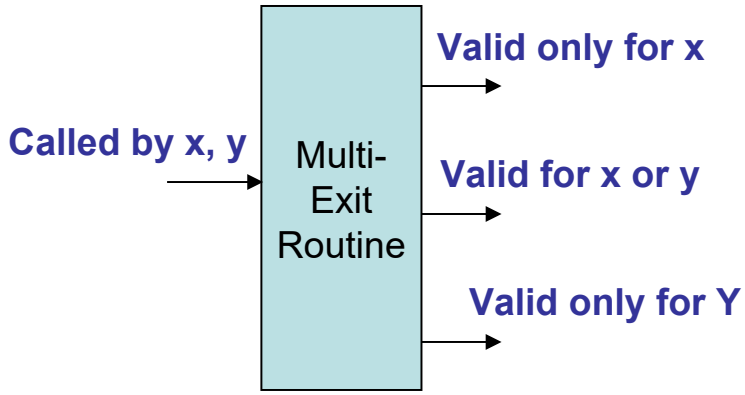
Path Testing Concepts..

Multi-entry / Multi-exit routines: (ill-formed)

- A Weak approach: Hence, convert it to single-entry / single-exit routine.
- Integration issues:

  Large # of inter-process interfaces. Creates problem in Integration.
  More # test cases and also a formal treatment is more difficult.



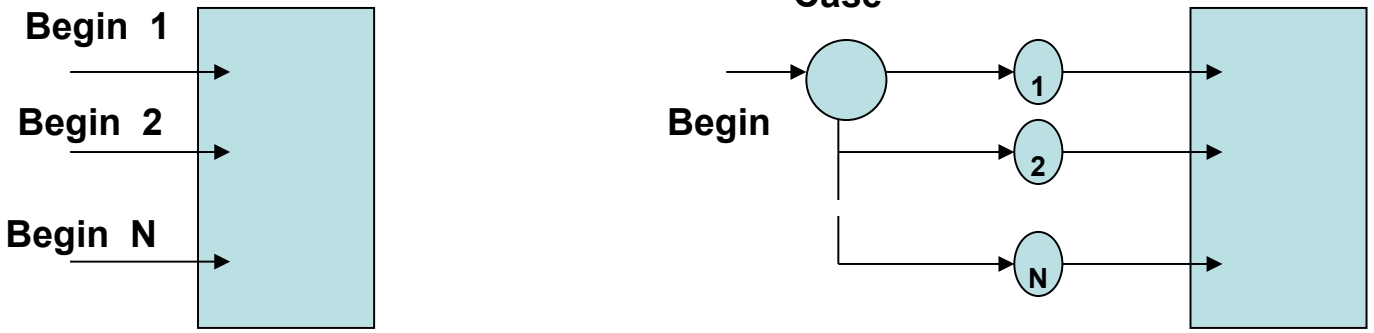


- Theoretical and tools based issues
  - A good formal basis does not exist.
  - Tools may fail to generate important test cases.

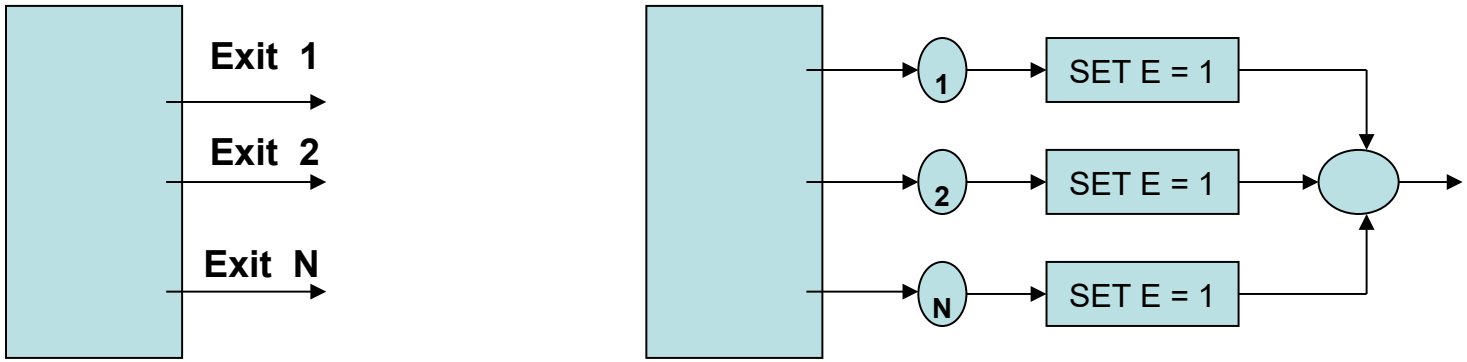
Path Testing Concepts contd..

Convert a multi-entry / exit routine to a single entry / exit routine:

- Use an entry parameter and a case statement at the entry => single-entry



- Merge all exits to Single-exit point after setting one exit parameter to a value.



Path Testing Concepts contd..

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.

Path Testing Concepts

3. 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.

- ♦ Point 1 => point 2 and 3.
- ♦ Point 1 is impractical.
- ♦ Point 2 & 3 are not the same
- ♦ For a structured language, Point 3 => Point 2

Path Testing Concepts

Path Testing Criteria :

1. **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.

2. **Statement Testing (P_1)** :

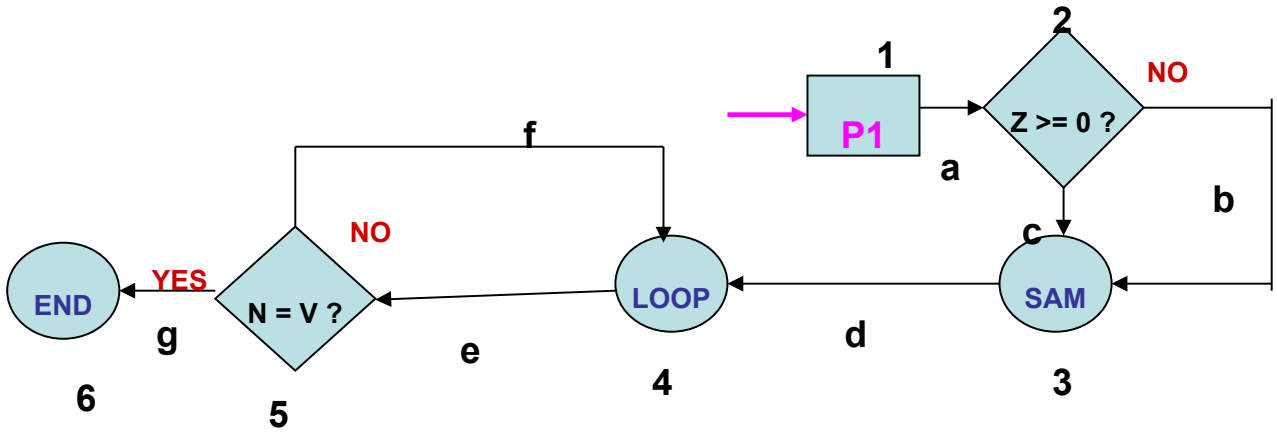
   Execute all statements in the program at least once under the some test.
   100% statement coverage => 100% node coverage.
   Denoted by **C1**
   C1 is a minimum testing requirement in the IEEE unit test standard: ANSI 87B.

3. **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

Picking enough (the fewest) paths for achieving C1+C2



1. Does every decision have Y & N (C2)?
2. Are call cases of case statement marked (C2)?
3. Is every three way branch covered (C2)?
4. Is every link covered at least once (C1)?

Make small changes in the path changing only 1 link or

Paths	Decisions		Process-link						
	2	5	a	b	c	d	e	f	g
abdeg	No	Y	Y	Y		Y	Y		Y
acdeg	Y	Y	Y		Y	Y	Y		Y
abdefeg	Y	N	Y	Y		Y	Y	Y	Y
acdefeg	Y	N	Y		Y	Y	Y	Y	Y

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

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**

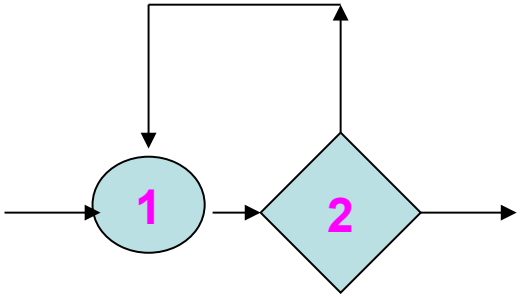
Let us denote the Minimum # of iterations by n_{min}
the Maximum # of iterations by n_{max}
the value of loop control variable by V
the #of test cases by T
the # of iterations carried out by n

- Later, we analyze the Loop-Testing times

Testing of path involving loops…

1. Testing a Single Loop Statement (three cases)

Case 1. $n_{min} = 0$, $n_{max} = N$, no excluded values

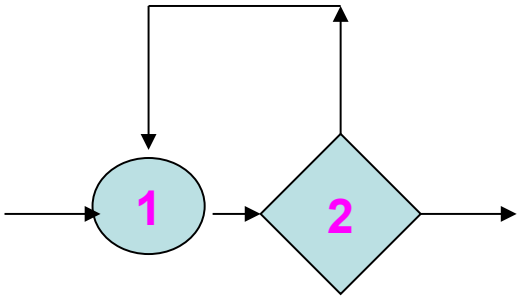


1. Bypass the loop.
   If you can't, there is a bug, $n_{min} \neq 0$ or a wrong case.
2. Could the value of loop (control) variable V be negative?
   could it appear to specify a –ve n ?
3. Try one pass through the loop statement: n = 1
4. Try two passes through the loop statement: n = 2
   To detect initialization data flow anomalies:
   Variable defined & not used in the loop, or
   Initialized in the loop & used outside the loop.
5. Try n = typical number of iterations : $n_{min} < n < n_{max}$
6. Try n = n_{max} -1

7. Try n = n_{max}
8. Try n = n_{max} + 1.
   What prevents V (& n) from having this value?
   What happens if it is forced?

Testing of path involving loops…

Case 2. n_{min} = +ve, n_{max} = N, no excluded values



1. Try n_{min} - 1
   Could the value of loop (control) variable V be < n_{min}?
   What prevents that ?
2. Try n_{min}
3. Try n_{min} + 1

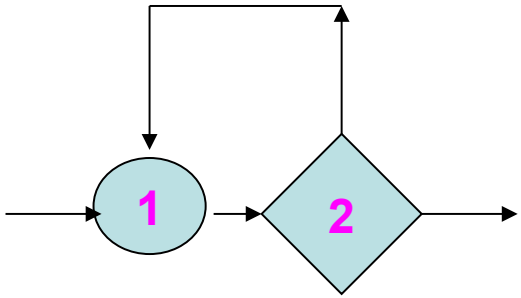
4. Once, unless covered by a previous test.
5. Twice, unless covered by a previous test.

4. Try n = typical number of iterations : n_{min} < n < n_{max}
5. Try n = n_{max} -1

6. Try n = n_{max}
7. Try n = n_{max} + 1.
   What prevents V (& n) from having this value?
   What happens if it is forced?

Path Testing Concepts…

Case 3. Single loop with excluded values



1. Treat this as single loops with excluded values as two sets.

2. Example:

V = 1 to 20 excluding 7,8,9 and10

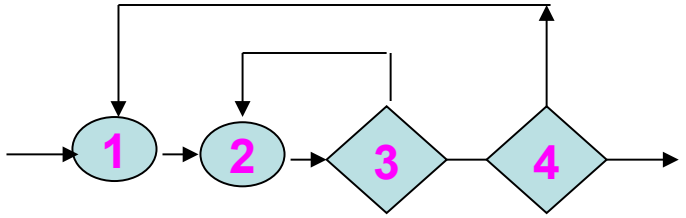
Test cases to attempt are for:

V = 0,1,2,4,6,7 and V = 10,11,15,19,20,21

(underlined cased are not supposed to work)

Testing of path involving loops…

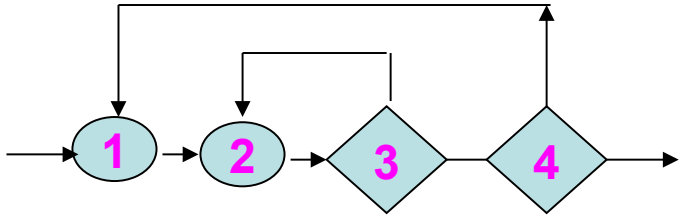
2. Testing a Nested Loop Statement



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

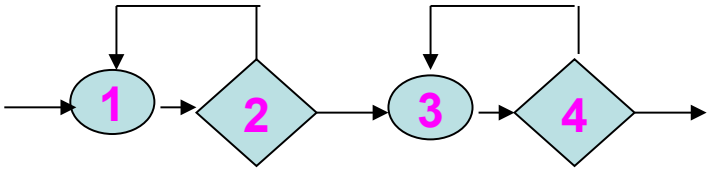
Testing of path involving loops…



- Compromise on # test cases for processing time.
- Expand tests for solving potential problems associated with initialization of variables and with excluded combinations and ranges.
- catch data initialization problems.

Testing of path involving loops…

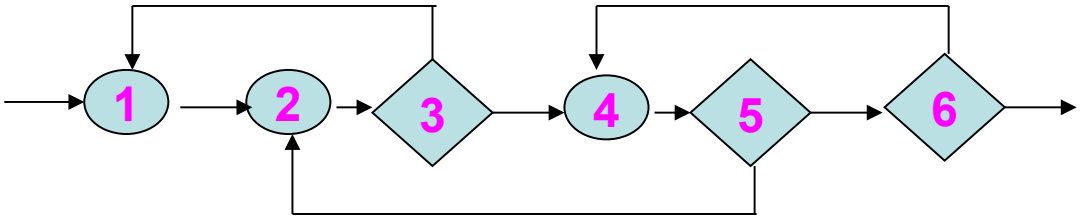
3. Testing Concatenated Loop Statements



- 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.

Testing of path involving loops…

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.

Testing of path involving loops…

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.
  - Test with the extreme-value combinations, but use different numbers.

- The test time problem is solved by rescaling the test limit values.

  - Can be achieved through a separate compile, by patching, by setting parameter values etc..

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.
  - Not all initialization errors are caught by path testing.
  - Specification errors cannot be caught.

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.
- More complicated path testing techniques than P1 & P2
  - Between P_2 & P_α
    - Complicated & impractical
  - Weaker than P1 or P2.
    - For regression (incremental) testing, it's cost-effective

Predicates, Predicate Expressions

Path

- A sequence of process links (& nodes)

Predicate

- The logical function evaluated at a decision : True or False. *(Binary , boolean)*

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

Predicates, Predicate Expressions…

Predicate Interpretation

- 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                                  INPUT X
        ON X GOTO A, B, C                           IF X < 0 THEN Y:= 2
           A:  Z := 7  @  GOTO H                       ELSE  Y := 1
     B:  Z := -7  @ GOTO H                  IF X + Y*Y > 0 THEN …
           C:  Z := 0  @ GOTO H

           H:  DO SOMETHING

           K:  IF X + Z > 0  GOTO  GOOD  ELSE  GOTO  BETTER
```

- Predicate interpretation may or may not depend on the path.

- Path predicates are the specific form of the predicates of the decisions along the selected path after interpretation.

Predicates, Predicate Expressions…

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
- Examples: X + Y = 10 Increment X & Decrement Y.
  X is odd Add an even # to X
- If all the input variables (on which a predicate is based) are process independent, then **predicate is process independent**.

Predicates, Predicate Expressions…

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. Otherwise, there is an input value which the routine loops indefinitely.

Predicates, Predicate Expressions…

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.
- Example (inputs being numerical values):

If $X_5 > 0$ .OR. $X_6 < 0$ then

$X_1 + 3X_2 + 17 >= 0$

$X_3 = 17$

$X_4 - X_1 >= 14\ X_2$

Predicates, Predicate Expressions…

A: $X_5 > 0$
B: $X_1 + 3X_2 + 17 >= 0$
C: $X_3 = 17$
D: $X_4 – X_1 >= 14\ X_2$

E: $X_6 < 0$
F: $X_1 + 3X_2 + 17 >= 0$
G: $X_3 = 17$
H: $X_4 – X_1 >= 14\ X_2$

Converting into the predicate expression form:

$$A\ B\ C\ D + E\ B\ C\ D \Rightarrow (\ A + E\ )\ B\ C\ D$$

If we take the alternative path for the expression: D then

$$(A + E\ \overline{)}\ B\ C\ D$$

Predicates, Predicate Expressions…

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:**
  - 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.

Testing blindness

- **coming to the right path** – even thru a wrong decision (at a predicate). Due to the interaction of some statements makes a buggy predicate work, and the bug is not detected by the selected input values.

- **calculating wrong number of tests** at a predicate by ignoring the # of paths to arrive at it.

- **Cannot be detected by path testing and need other strategies**

Testing blindness

- **Assignment blinding**: A buggy Predicate seems to work correctly as the specific value chosen in an assignment statement works with both the correct & buggy predicate.

Correct	Buggy	
X := 7	X := 7	
IF Y > 0 THEN …	IF X + Y > 0 THEN …	**(check for Y=1)**

- **Equality blinding**:
  - When the path selected by a prior predicate results in a value that works both for the correct & buggy predicate.

Correct	Buggy	
IF Y = 2 THEN …	IF Y = 2 THEN ..	
IF X + Y > 3 THEN …	IF X > 1 THEN …	**(check for any X>1)**

- **Self-blinding**
  - When a buggy predicate is a multiple of the correct one and the result is indistinguishable along that path.

Correct	Buggy	
X := A	X := A	
IF X - 1 > 0 THEN …	IF X + A -2 > 0 THEN …	**(check for any X,A)**

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. Example: (A + BC) (D + E)

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.

7. A set of input values for that path is found when any of the inequality sets is solved.

   A solution found => **path is achievable**. Otherwise the path is **unachievable.**

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.

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.

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… Heuristic procedures: contd..

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.

7. 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.

8. 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

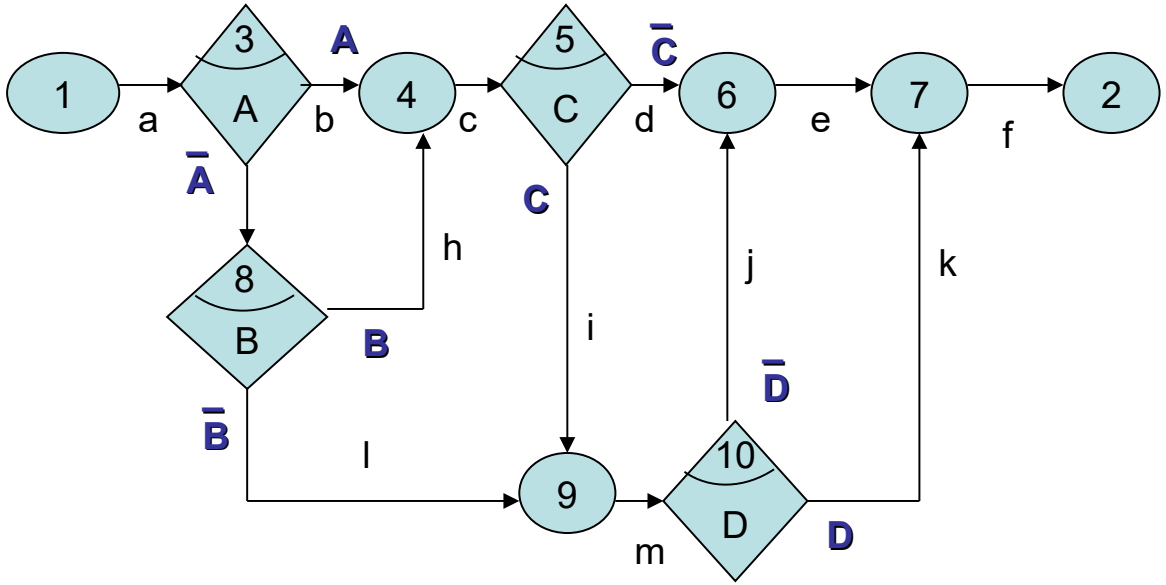
2. Independent Correlated Predicates

3. Dependent Predicates

4. Generic

Examples for Path Sensitization..

1. Simple Independent Uncorrelated Predicates



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

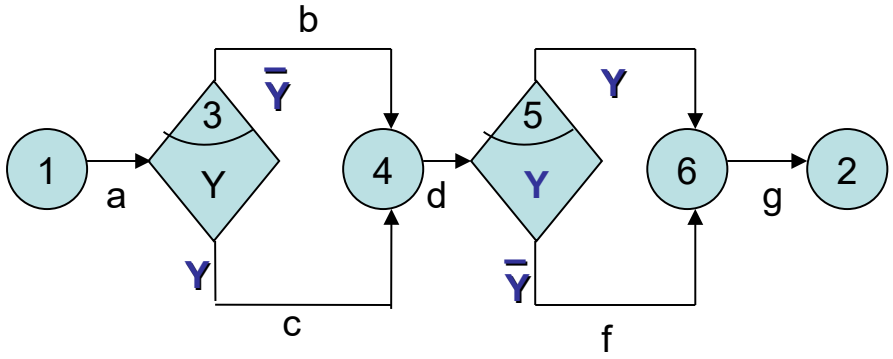
Path	Predicate Values
abcdef	A $\bar{C}$
aghcimkf	$\bar{A}$ B C D
aglmjef	$\bar{A}$ $\bar{B}$ $\bar{D}$

Path	Predicate Values
abcdef	A $\bar{C}$
abcimjef	A C $\bar{D}$
abcimkf	A C D
aghcdef	$\bar{A}$ B $\bar{C}$
aglmkf	$\bar{A}$ $\bar{B}$ $\bar{D}$

A Simple case of solving inequalities. (obtained by the procedure for finding a covering set of paths)

Examples for Path Sensitization …

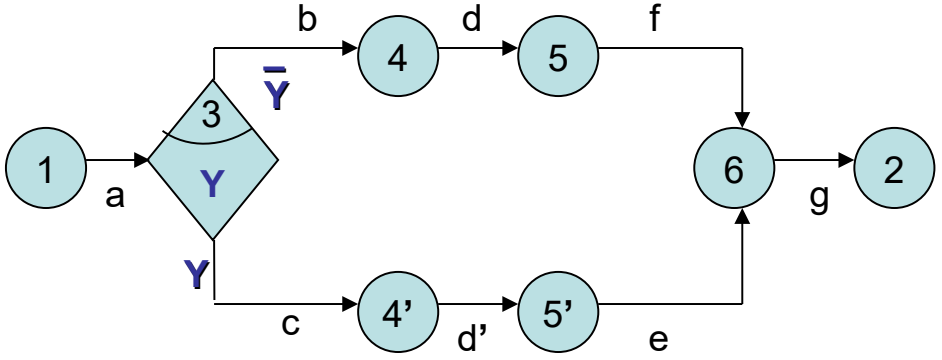
2. Correlated Independent Predicates



Correlated paths => **some** paths are unachievable
ie., redundant paths.
ie., n decisions but # paths are < 2^n

Due to practice of saving code which makes the code very difficult to maintain.

Eliminate the correlated decisions.
Reproduce common code.



If a chosen sensible path is not achievable,

- there's a bug.
- design can be simplified.
- get better understanding of correlated decisions

Correlated decision removed & CFG simplified

3. 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, and then work backward to determine the value of input variables (input vector).

4. 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.

5. Continue until the entrance & therefore have established a set of input conditions for the path.

If the solution is not found, path is not achievable, *it could be a bug.*

4. The General Case contd..

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.

Advantages & Disadvantages of the two approaches:

The forward method is usually less work.

you do not know where you are going as you are tracing the graph.

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:

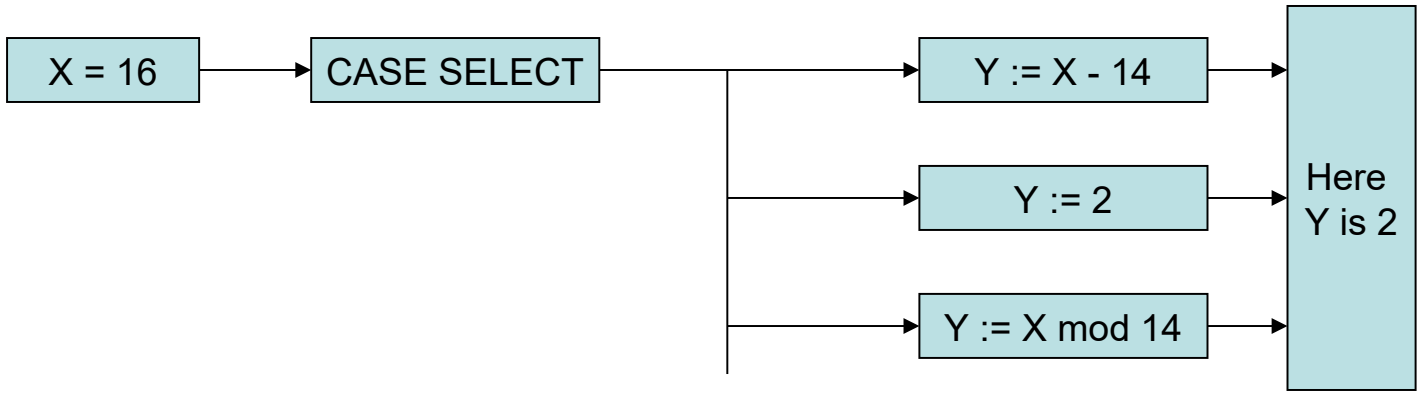
Observed outcome

PATH INSTRUMENTATION

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.

PATH INSTRUMENTATION METHODS

1. General strategy:

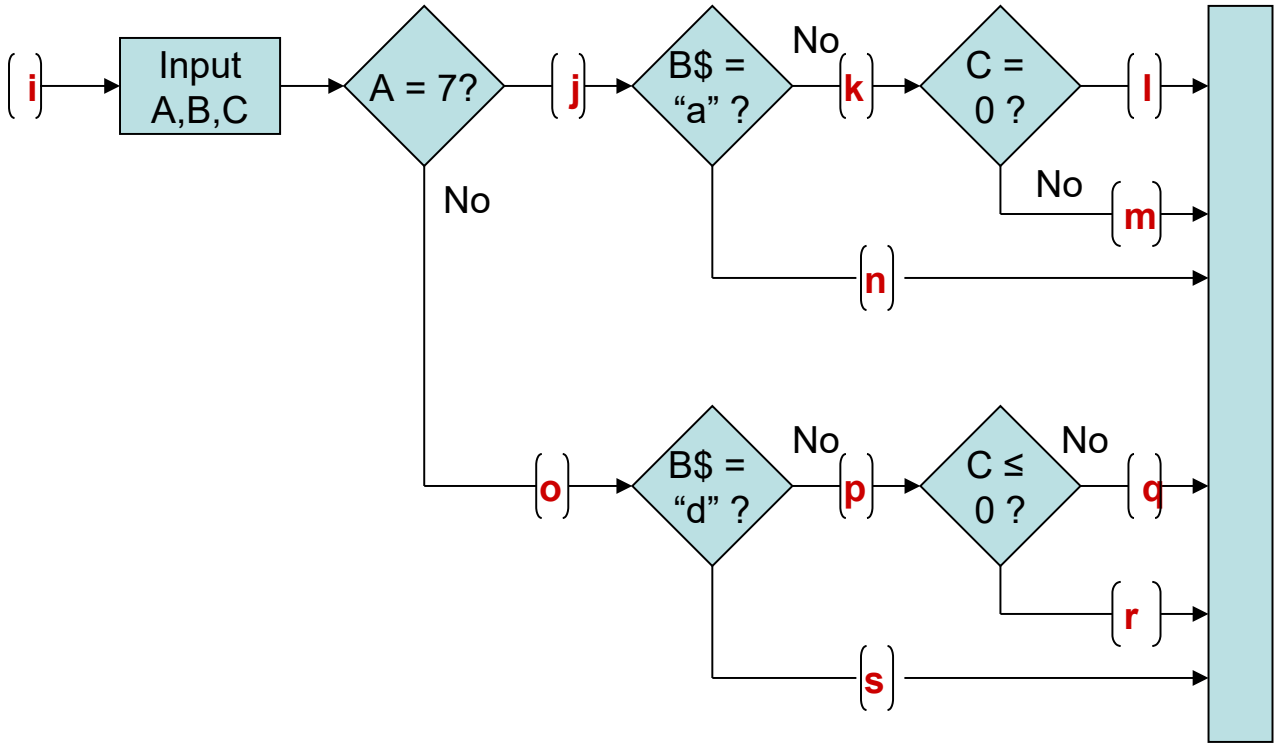
1. Based on Interpretive tracing & use interpreting trace program.
2. A trace confirms the expected outcome is or isn't obtained along the intended path.
3. Computer trace may be too massive. Hand tracing may be simpler.

2. Traversal or Link Makers:

Simple and effective

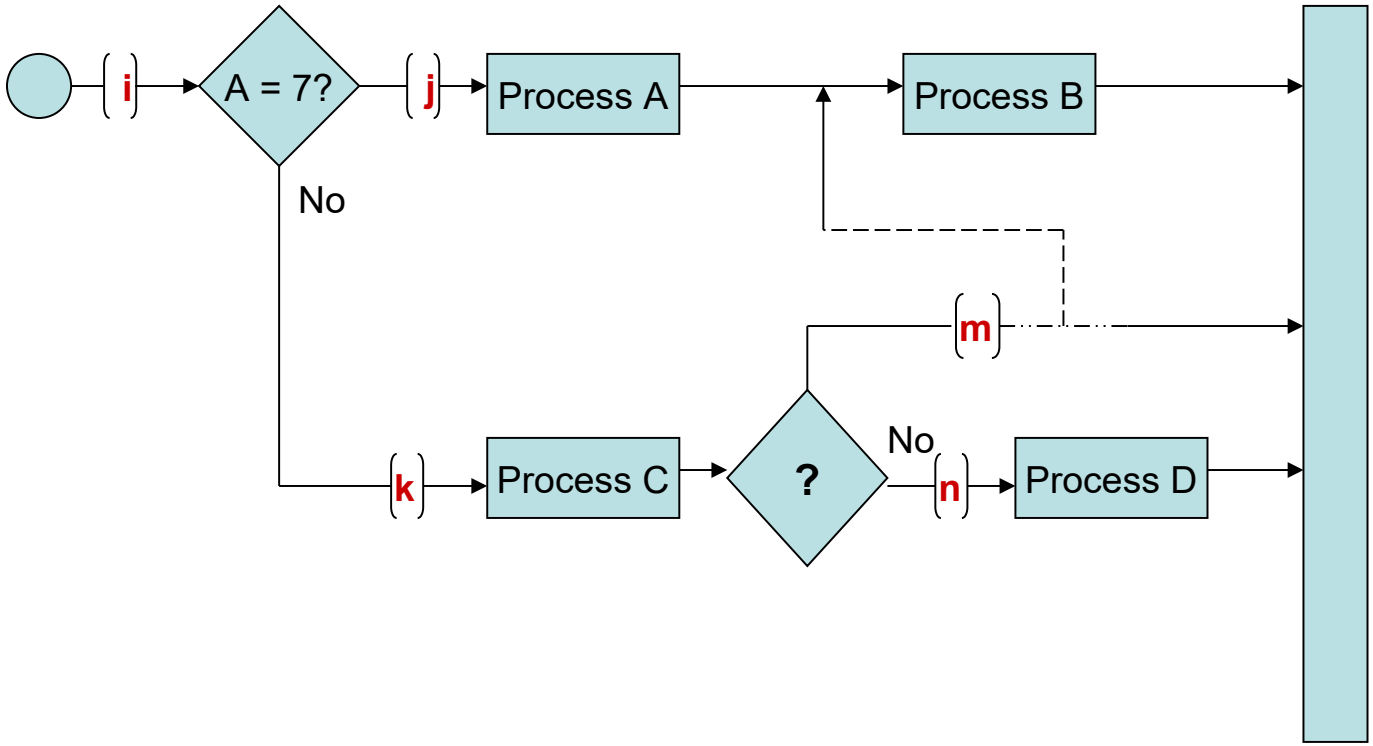
1. Name every link.
2. Instrument the links so that the link is recorded when it is executed (during the test)
3. The succession of letters from a routine's entry to exit corresponds to the pathname.

Single Link Marker Instrumentation: An example



Sample path: i j n

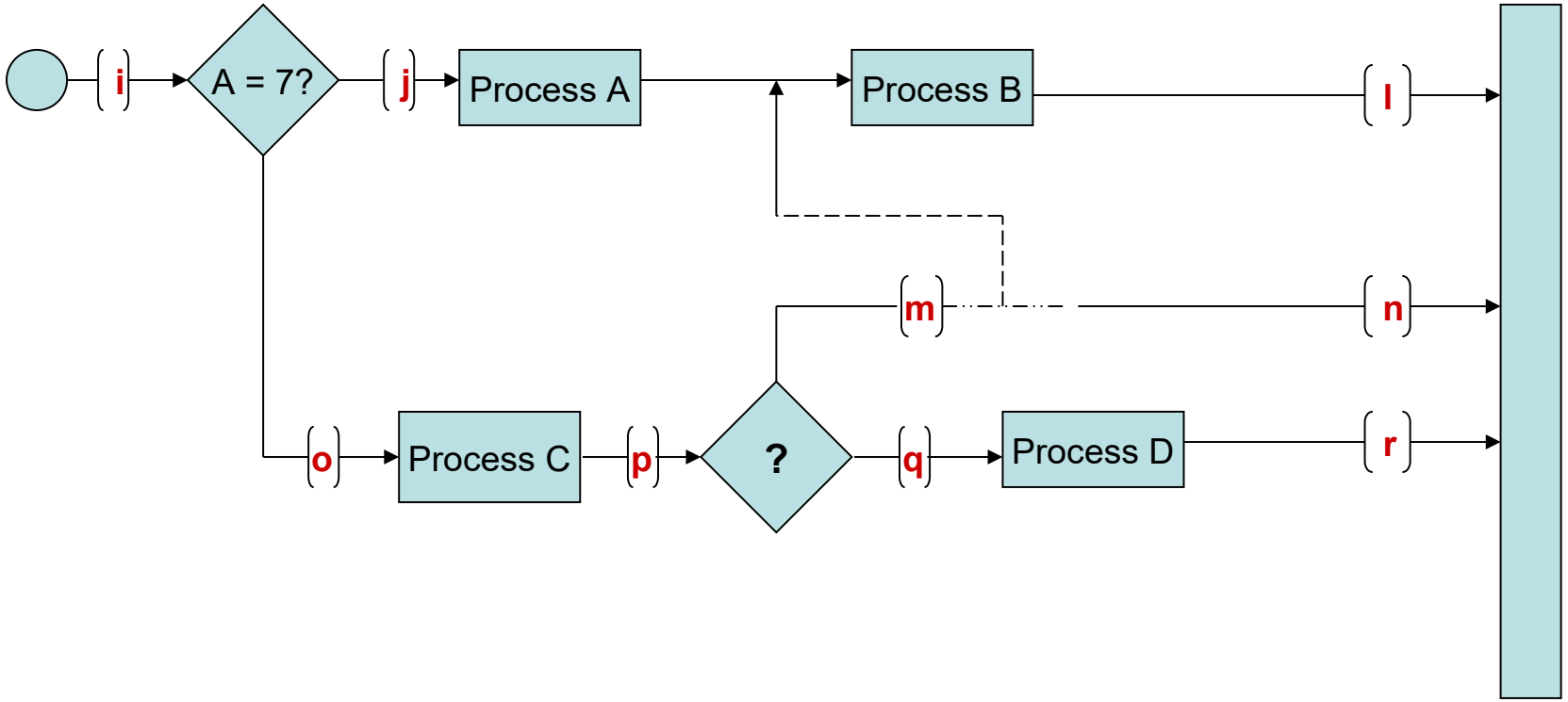
Single Link Marker Instrumentation: Not good enough



Problem:

Processing in the links may be chewed open by bugs. Possibly due to GOTO statements, control takes a different path, yet resulting in the intended path again.

Double Link Marker Instrumentation:



The problem is solved.

Two link markers specify the path name and both the beginning & end of the link.

3. Link Counters Technique:

- 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.
- 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.

PATH INTRUMENTATION techniques...

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.

PATH INTRUMENTATION - IMPLEMENTATION

For Unit testing : Implementation may be provided by a comprehensive test tool.

For higher level testing or for testing an unsupported language:

- Introduction of probes could introduce bugs.
- Instrumentation is more important for higher level of program structure like transaction flow
- At higher levels, the discrepancies in the structure are more possible & overhead of instrumentation may be less.

For Languages supporting conditional assembly or **compilation**:

- Probes are written in the source code & tagged into categories.
- Counters & traversal markers can be implemented.
- Can selectively activate the desired probes.

For language not supporting conditional assembly / compilation:

- Use macros or function calls for each category of probes. This may have less bugs.
- A general purpose routine may be written.

In general:

- Plan instrumentation with probes in levels of increasing detail.

Implementation & Application of Path Testing

1. Integration, Coverage, and Paths in Called Components

- Mainly used in Unit testing, especially new software.
- integrating one component at a time.
- **In reality,** integration proceeds in associated blocks of components.

To achieve C1 or C2 coverage:

- Predicate interpretation may require us to treat a subroutine as an in-line-code.
- Sensitization becomes more difficult.
- Selected path may be unachievable as the called components' processing may block it.

Weaknesses of Path testing:

- It assumes that effective testing can be done one level at a time without bothering what happens at lower levels.
- predicate coverage problems & blinding.

Implementation & Application of Path Testing

2. 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.

Implementation & Application of Path Testing

3. Application of path testing to **Maintenance**

- Path testing is applied first to the modified component.
- Select paths to achieve C2 over the changed code.
- Newer and more effective strategies could emerge to provide coverage in maintenance phase.

4. 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.

Implementation & Application of Path Testing

Application of path testing to **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.)

Control Flow Graphs and Path Testing – Questions from previous exams

For reference – just to see that we have covered these:

Q. Define Path Testing. Explain three path testing criteria.

Q. Illustrate with an example, how statement and branch coverage can be achieved during path selection. Write all steps involved in it.

Q. Write the effectiveness and limitations of path testing.

Q. Define Path Sensitization. Explain heuristic procedure for sensitizing paths with the help of an example.

Q. How can a program control structure be represented graphically? Explain with the help of required diagrams.

Q. How is a flowchart differed from a control flow chart?

Q. Explain about Multi entry and Multi exit routines & fundamental path selection criteria

Q. Explain about path instrumentation. How are link counters useful in Path Instrumentation method?

Q. Write about implementation of path testing. What are the various applications of path testing?

Q. Categorize different kinds of loops and explain.