Chapter - 3

FLOW GRAPHS AND PATH TESTING

Path Testing Basics

- **Path Testing:** Path Testing is the name given to a family of test techniques based on judiciously selecting a set of test paths through the program.
- If the set of paths are properly chosen then we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement has been executed at least once.
- Path testing techniques are the oldest of all structural test techniques.

Path Testing Basics-continued.,

- Path testing is most applicable to new software for unit testing. It is a structural technique.
- It requires complete knowledge of the program's structure.
- It is most often used by programmers to unit test their own code.
- The effectiveness of path testing rapidly deteriorates as the size of the software aggregate under test increases.

The Bug Assumption

- The bug assumption for the path testing strategies is that something has gone wrong with the software that makes it take a different path than intended.
- As an example "GOTO X" where "GOTO Y" had been intended.
- Structured programming languages prevent many of the bugs targeted by path testing: as a consequence the effectiveness for path testing for these languages is reduced and for old code in COBOL, ALP, FORTRAN and Basic, the path testing is indispensable.

Control Flow Graphs

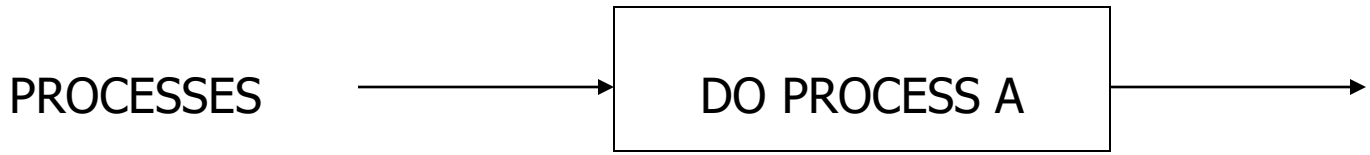
- The control flow graph is a graphical representation of a program's control structure. It uses the elements named process blocks, decisions, and junctions.
- The flow graph is similar to the earlier flowchart, with which it is not to be confused.

Flow Graph Elements

- A flow graph contains four different types of elements.
  - Process Block
  - Decisions
  - Junctions
  - Case Statements

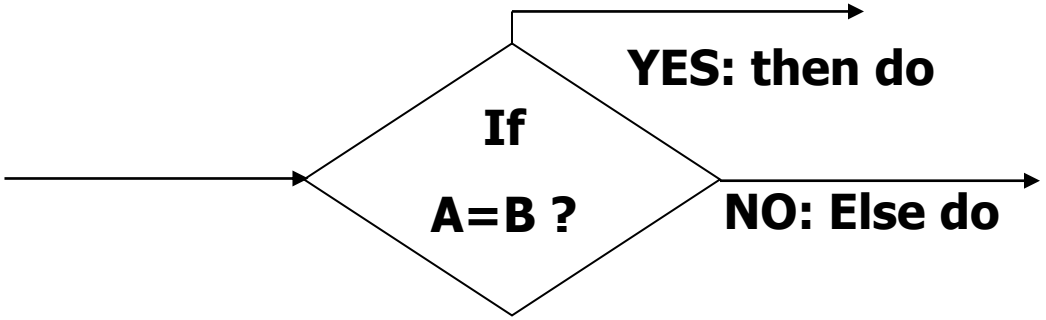
Process Block

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



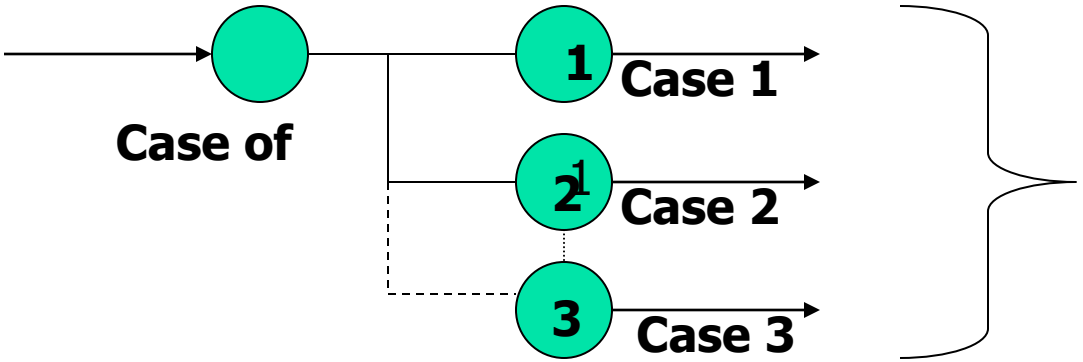
Decisions

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



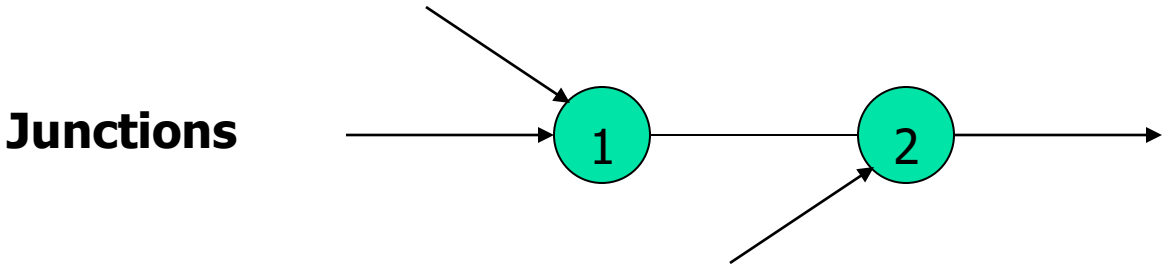
Case Statements

- A case statement is a multi-way branch or decisions.
- Examples of case statement are a jump table in assembly language, and the PASCAL case statement.
- From the point of view of test design, there are no differences between Decisions and Case Statements



Junctions

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



Control Flow Graphs Versus Flow Charts

- A program's flow chart resembles a control flow graph.
- In flow graphs, we don't show the details of what is in a process block.
- In flow charts every part of the process block is drawn.
- The flowchart focuses on process steps, where as the flow graph focuses on control flow of the program
- The act of drawing a control flow graph is a useful tool that can help us clarify the control flow and data flow issues.

Flow graph and flowchart generation

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

Flowgraph – Program Correspondence

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

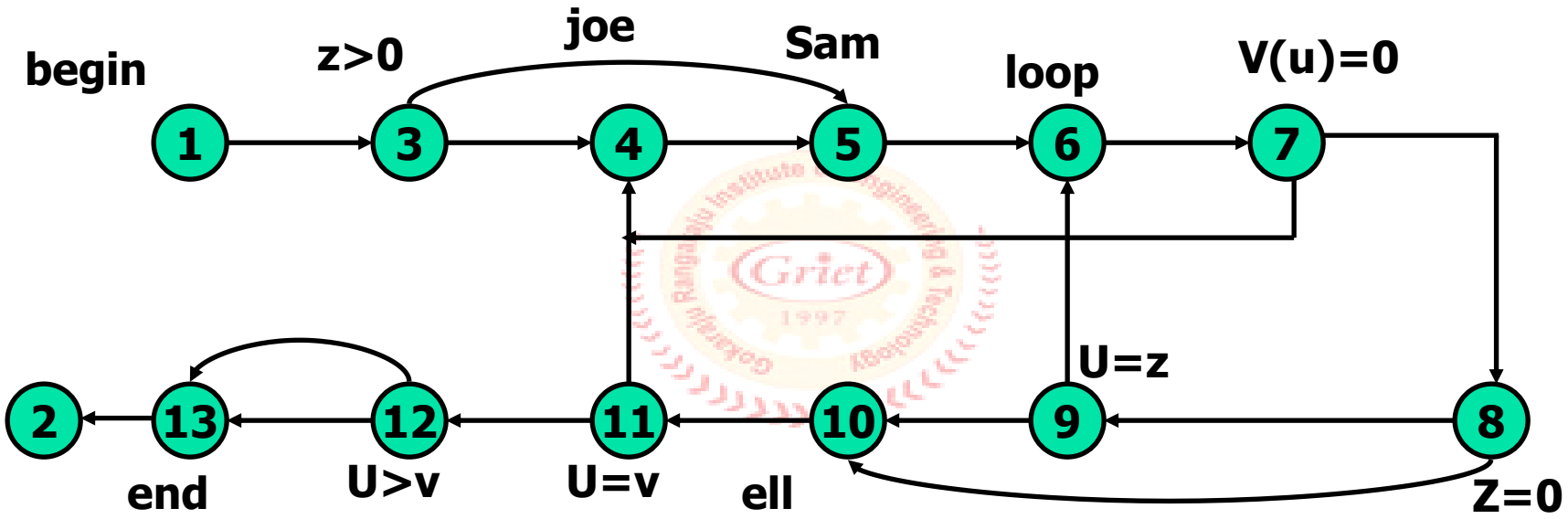
Notational Evolution

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

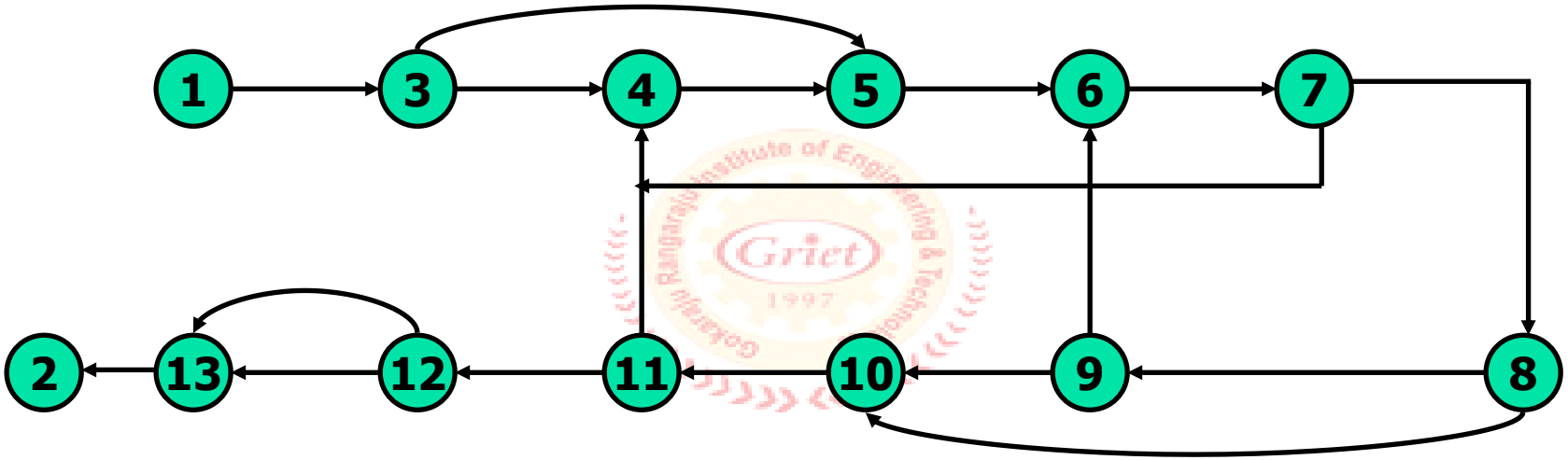
Program - Example

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

Flow Graph - Example

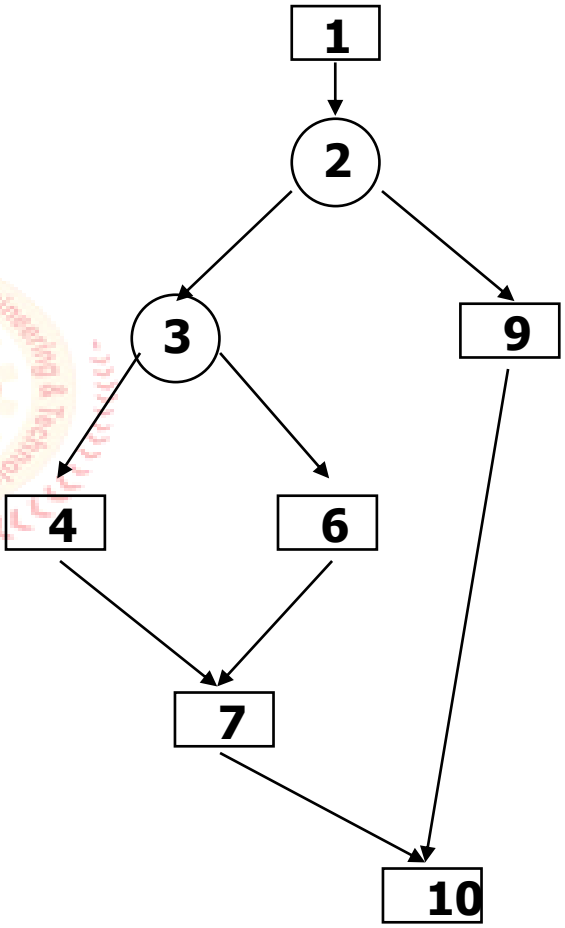


Flow Graph - Example

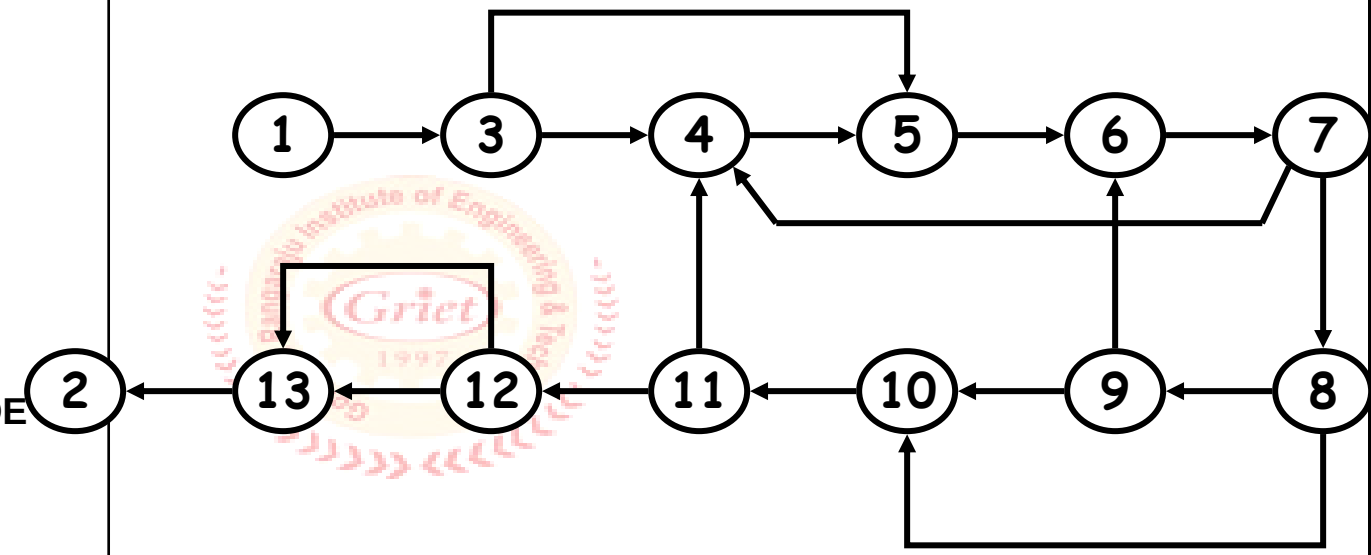


Flow Graph Example

```
procedure XYZ is
      A,B,C: INTEGER;
   begin
1.   GET(A); GET(B);
2.   if  A > 15  then
3.       if    B  < 10   then
4.                B := A + 5;
5.       else
6.                B := A  -  5;
7.       end if
8.    else
9.       A := B + 5;
10.   end if;
   end XYZ;
```

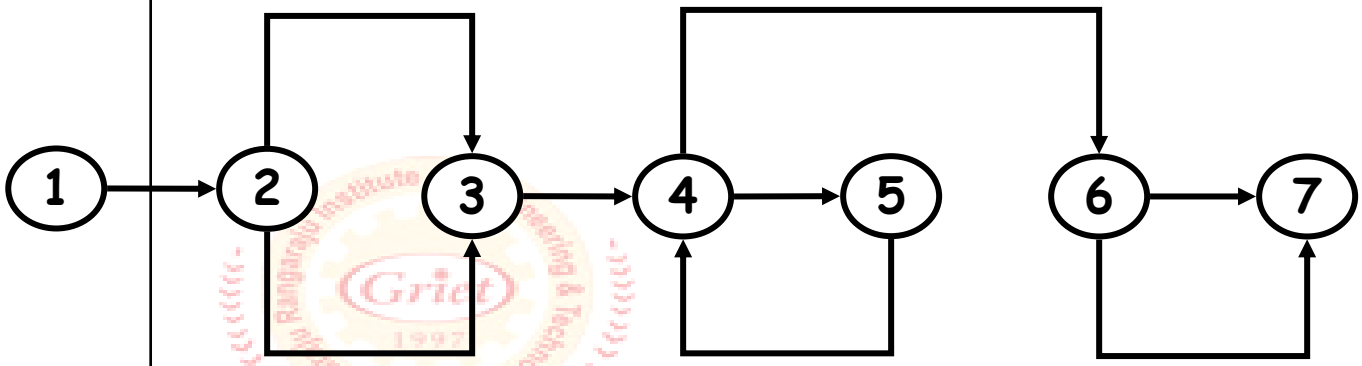


```
INPUT X,Y
          Z:=X+Y
          V:=X-Y
3         IF Z>=0 GOTO SAM
4 JOE:    Z:=Z+V
5 SAM:    Z:=Z+V
          U:=0
6         LOOP
          B(U),Q(V):=(Z+V)*U
7         IF B(U)=0 GOTO JOE
          Z:=Z-1
8         IF Z=0 GOTO ELL
          U:=U+1
9         UNTIL U=Z
          B(U-1):=B(U+1)+Q(V-1)
10  ELL:B(U+Q(V)):=U+V
    IF U=V GOTO JOE
12          IF U>V THEN U := Z
13  YY:   Z:=U
 2          END
```



Exponentiation Algorithm

```
1   scanf("%d %d",&x,
&y);
2   if (y < 0)
        pow = -y;
    else
        pow = y;
3   z = 1.0;
4   while (pow != 0) {
        z = z * x;
        pow = pow - 1;
5       }
6   if (y < 0)
        z = 1.0 / z;
7   printf ("%f",z);
```



Bubble Sort Algorithm

```
1  for (j=1; j<N; j++) {
       last = N - j + 1;
2      for (k=1; k<last; k++) {
3          if (list[k] > list[k+1]) {
               temp = list[k];
               list[k] = list[k+1];
               list[k+1] = temp;
4          }
5      }
6 }
7 print("Done\n");
```

Linked List representation of flow graphs

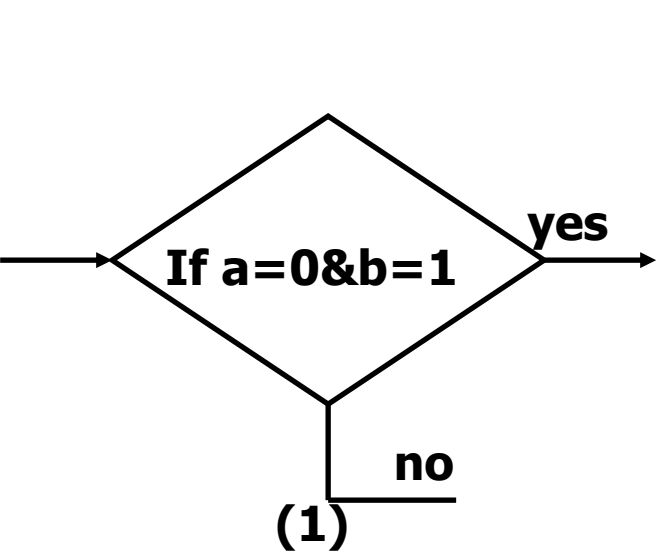
- Although graphical representations of flowgraphs are revealing the details of the control flow inside a program they are often inconvenient.
- In linked list representation, each node has a name and there is an entry on the list for each link in the flow graph. only the information pertinent to the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

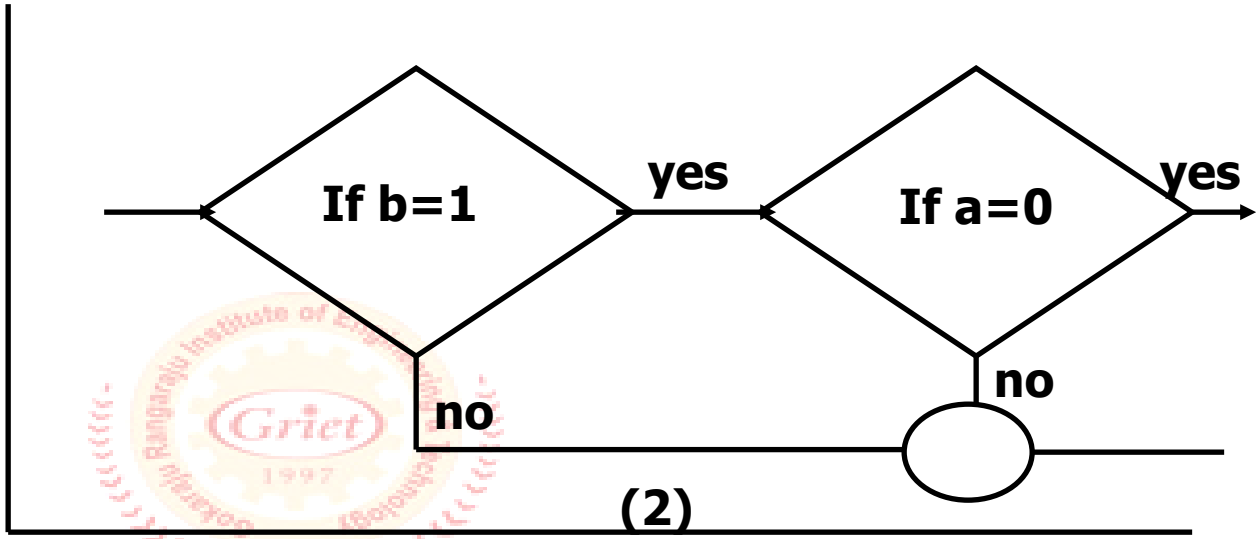
Example - Linked list representation

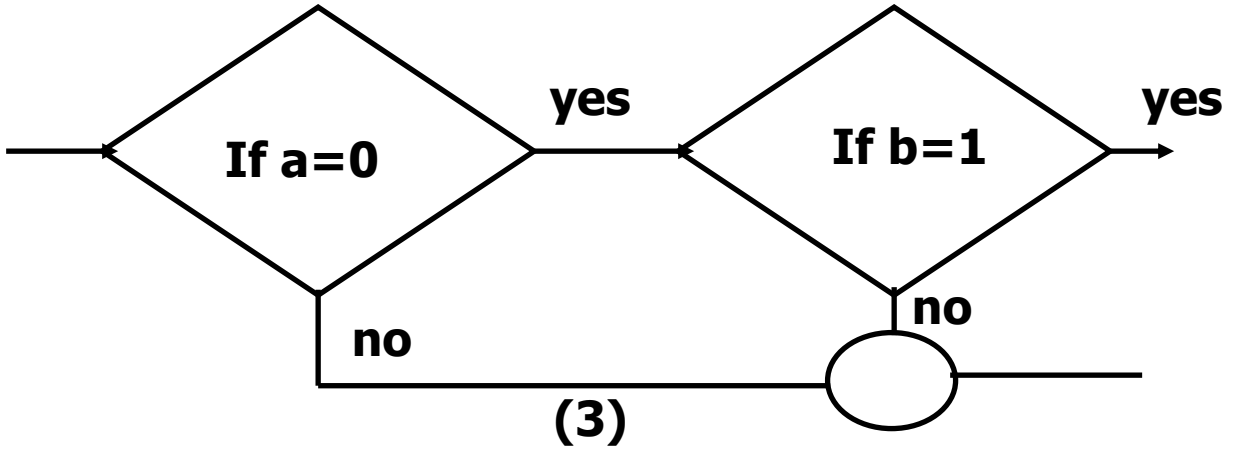
1	(begin)	: 3
2	(end)	: exit, no outlink
3	(z>0)	: 4 (false)
		: 5 (true)
4	(joe)	: 5
5	(sam)	: 6
6	(loop)	: 7
7	(v(u)=0?))	: 4 (true)
		: 8 (false)

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

Alternative flow graphs for the same logic







Path Testing- paths, nodes and links

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

Fundamental Path Selection Criteria

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

Path selection criteria - example

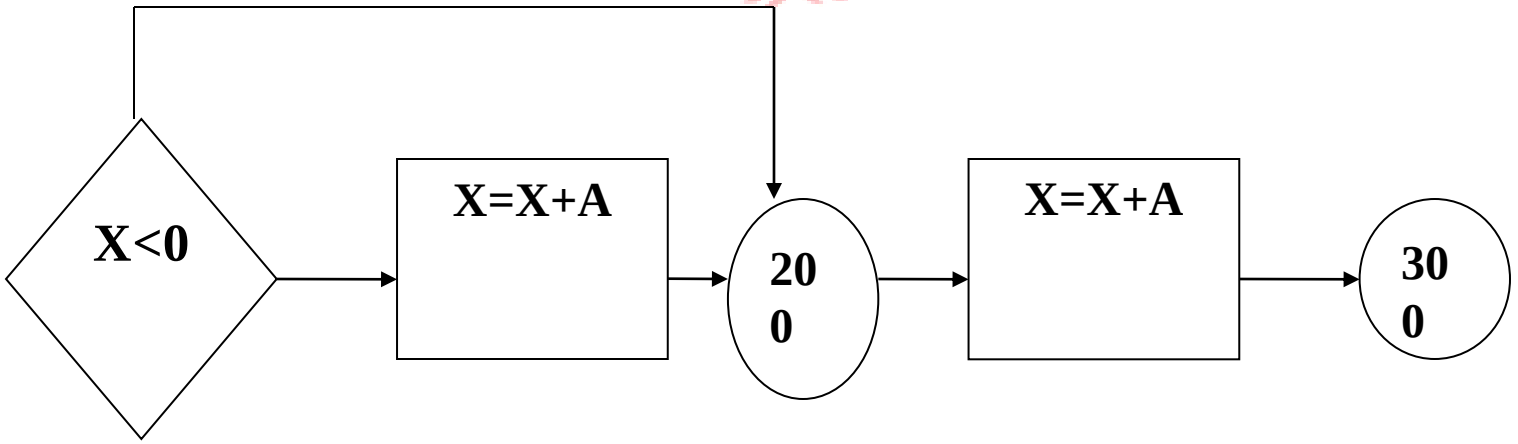
PRESCRIPTION 1:

If (X, LT,0) goto200

X=X+A

200 X=X+A

300 CONTINUE



- PRESCRIPTION 2: Executing every statement but not every branch would not reveal the bug in the incorrect version.

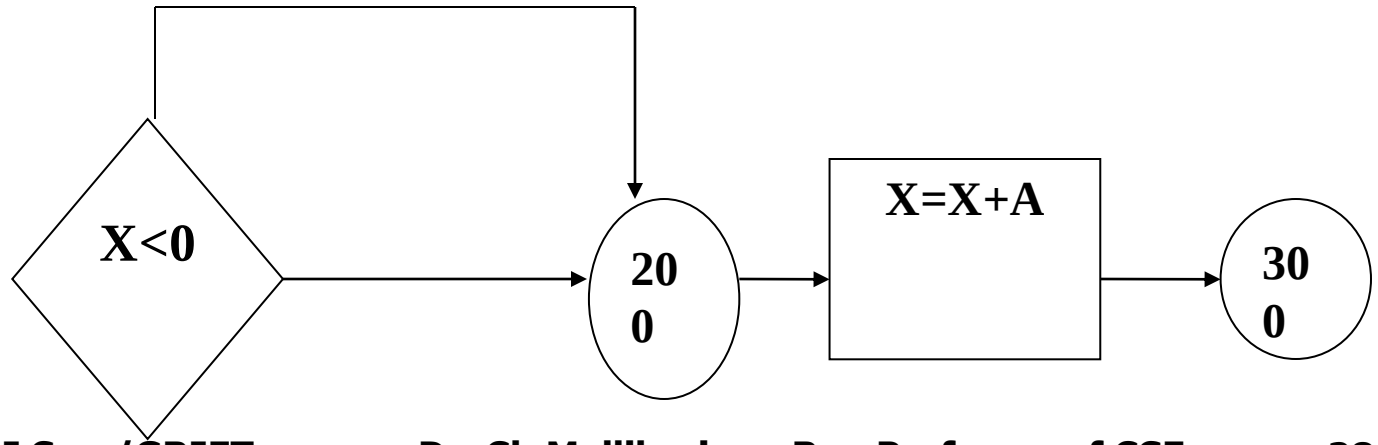
PRESCRIPTION 1:

If (X, LT,0) goto200

200 X=X+A

300 CONTINUE

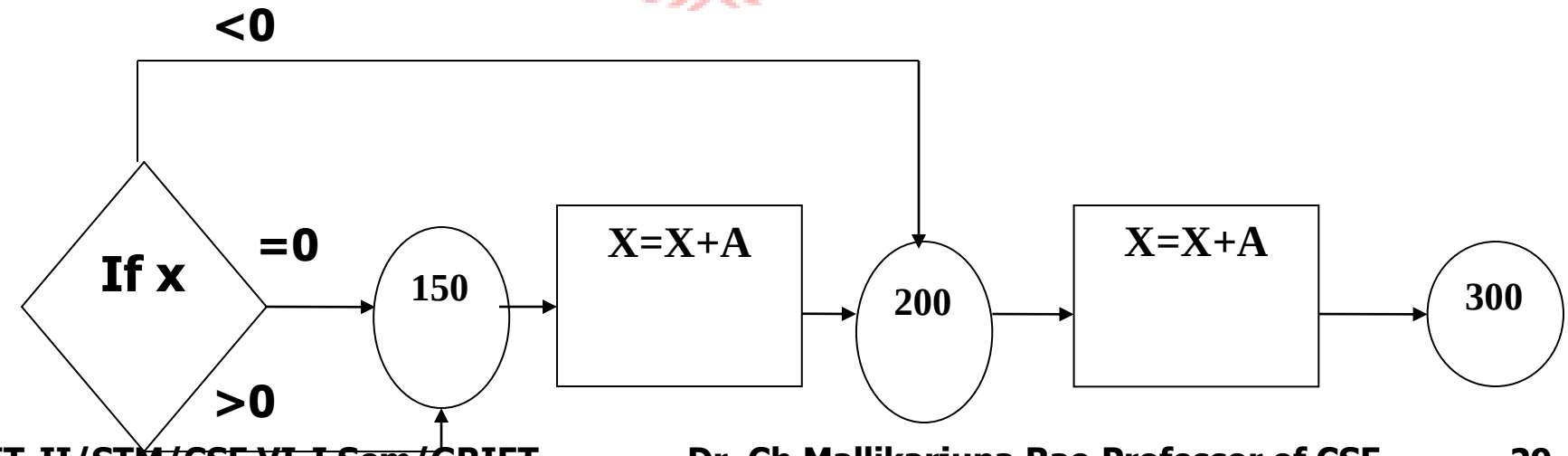
Negative value produces the correct answer. Every statement can be executed, but if the test cases do not force each branch to be taken, the bug can remain hidden.



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

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

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



Path Testing Criteria

- Any testing strategy based on paths must at least both exercise every instruction and take branches in all directions.
- A set of tests that does this is not complete in an absolute sense, but it is complete in the sense that anything less must leave something untested.
- So we have explored three different testing criteria or strategies out of a potentially infinite family of strategies.
  - 1. Path testing
  - 2. Statement testing
  - 3. Branch testing

Path Testing (P_{inf})

- Path Testing: execute all possible control flow paths through the program: typically, this is restricted to all possible entry/exit paths through the program.
- If we achieve this prescription, we are said to have achieved **100% path coverage.** This is the strongest criterion in the path testing strategy family: it is generally impossible to achieve.

Statement Testing (P_1)

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

Branch Testing (P_2)

- Execute enough tests to assure that every branch alternative has been exercised at least once under some test.
- If we do enough tests to achieve this prescription, then we have achieved **100% branch coverage.**
- An alternative characterization is to say that we have achieved **100% link coverage**.
- For structured software, branch testing and therefore branch coverage strictly includes statement coverage.
- We denote branch coverage by C2.

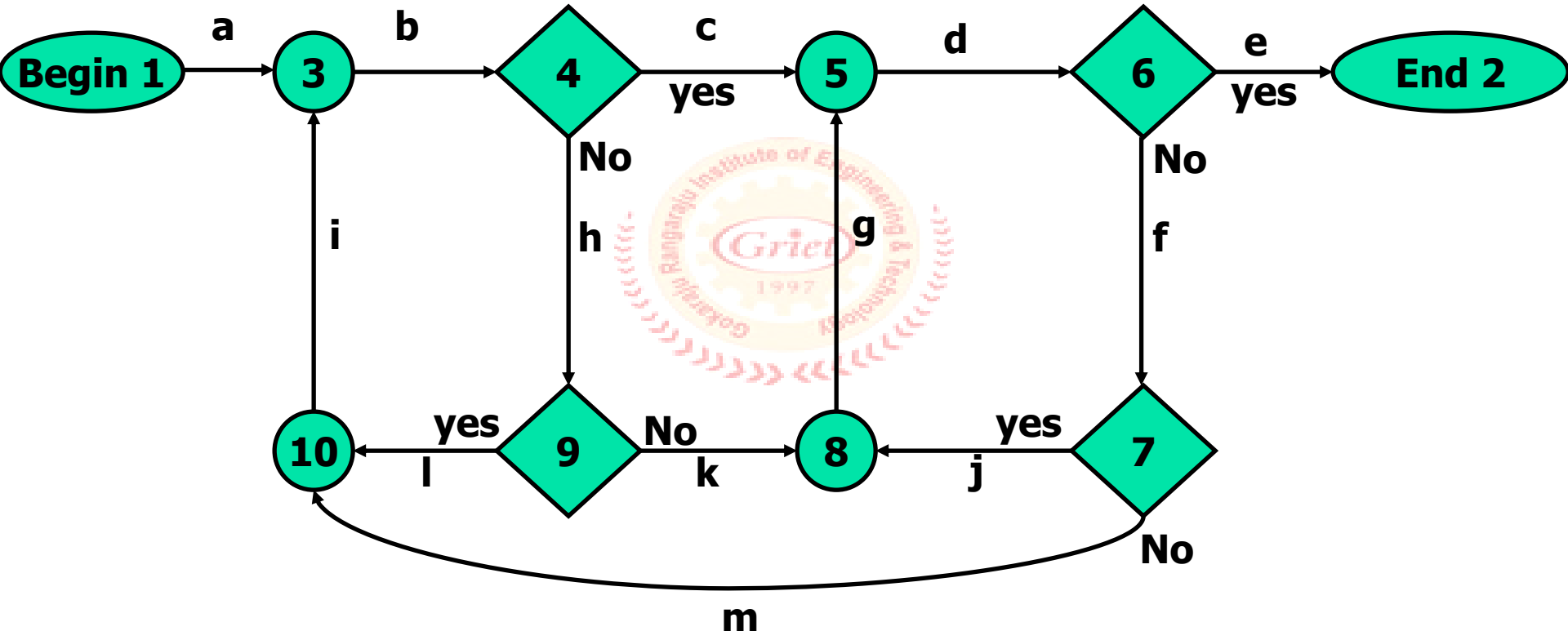
Common sense and strategies

- Branch and statement coverage are accepted today as the minimum mandatory testing requirement.
- The question " why not use a judicious sampling of paths?", what's wrong with leaving some code, untested?" is ineffectual in the view of common sense and experience since:
  - 1. not testing a piece of a code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs.
  - 2. the high probability paths are always thoroughly tested if only to demonstrate that the system works properly.

Which paths to be tested?

- You must pick enough paths to achieve C1+C2.
- The question of what is the fewest number of such paths is interesting to the designer of test tools that help automate the path testing, but it is not crucial to the pragmatic design of tests.

Path Selection – Example



An example of path selection

- Draw the control flow graph on a single sheet of paper.
- Make several copies – as many as you will need for coverage (C1+C2) and several more.
- Use a yellow highlighting marker to trace paths. Copy the paths onto a master sheets.
- Continue tracing paths until all lines on the master sheet are covered, indicating that you appear to have achieved C1+C2.
- As you trace the paths, create a table that shows the paths, the coverage status of each process, and each decision.

Tracing table for Path Selection

Paths	Conditions				Process Links											
	4	6	7	9	A	B	C	D	E	F	G	H	I	J	K	L	M
abcde	yes	Yes	-	-	1	1	1	1	1	-	-	-	-	-	-	-	-
Abhkgde	No	Yes	-	No	1	1	-	1	1	-	1	1	-	-	1	-	-
Abhlibcde	No, yes	Yes	-	Yes	1	1	1	1	1	-	-	1	1	-	-	1	-
abcdfjgde	Yes	No, yes	Yes	-	1	1	1	1	1	1	1	-	-	1	-	-	-
Abcdfmibcde	Yes	No, yes	no	-	1	1	1	1	1	1	-	-	1	-	-	-	1

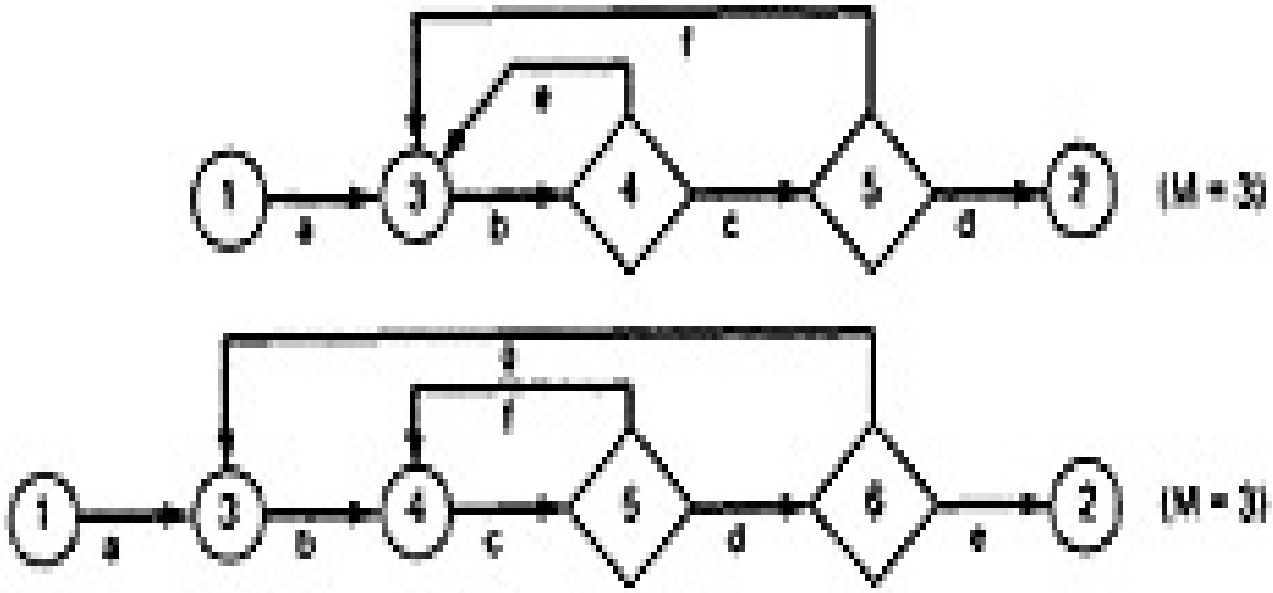
- After you have traced a covering path set on the master sheet and filled in the table for every path. Check the following:
- Does every decision have a YES and NO in its column? (C2)
- Has every case of all case statements been marked? (C2)
- Is every three way branch covered? (C2)
- Is every link covered at least once? (C1)

Revised path selection rules

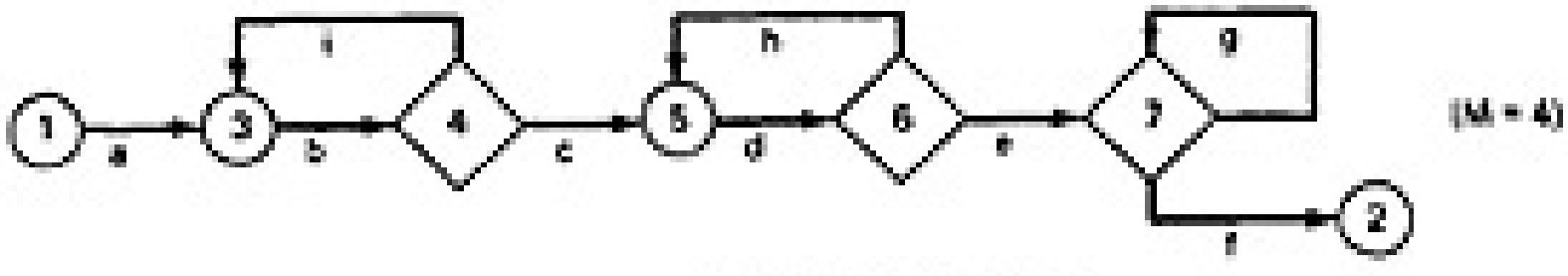
- Pick the simplest, functionally sensible entry/exit path.
- Pick additional paths as small variation from previous paths. Pick paths that do not have loops rather than paths that do. Favor short paths that make sense over paths that don't.
- Pick additional paths that have no obvious functional meaning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

Loops

- There are only three kinds of loops with respect to path testing:
  - Nested loops
  - Concatenated loops
  - Horrible loops



a, b) Nested Loops



c) Concatenated Loops

Cases for a single loop

- A single loop can be covered with two cases:
  - Looping
  - Not looping
- Experience shows that loop related bugs are not discovered by C1+c2.
- Bugs lurk in some corners and congregate at boundaries-in the case of loops at or around the minimum and maximum number of times the loop can be iterated.

Cases for a single loop-continued.,

- Case 1: single loop, Zero minimum, N maximum, No Excluded Values
  - Try bypassing the loop. If you can't you either have a bug, or zero is not the minimum and you have the wrong case.
  - Could the loop control variable be negative
  - One pass through the loop
  - A typical number of iterations, unless covered by a previous test.
  - One less than the maximum number of iterations
  - The maximum number of iterations
  - Attempt one more than the maximum number of iterations.

Cases for a single loop-continued.,

- Case 2: single loop, Non Zero Minimum, No Excluded Values
  - Try one less than the expected minimum
  - The minimum number of iterations
  - One more than the minimum number of iterations
  - Once, unless covered by a previous test
  - Twice, unless covered by a previous test
  - A typical value
  - One less than the maximum value
  - The maximum number of iterations
  - Attempt one more than the maximum number of iterations

Cases for a single loop-continued.,

- Case 3: single loops with excluded values
  - Treat single loops with excluded values as two sets of tests consisting of loops without excluded values, such as case 1 and 2 above
  - Example, the total range of the loop control variable was 1 to 20, but that values 7,8,9,10 were excluded. The two sets of tests are 1-6 and 11-20.
  - The sample test cases to attempt would be 0,1,2,4,6,7 for the first range and 10,11,15,19,20,21 for the second range.

Nested Loops

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

Concatenated loops

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

Horrible loops

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

Loop Testing Time

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

Predicates, paths predicates, and achievable paths

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

Path Predicates

- A predicate associated with a path is called a Path Predicate.
- For example

  "x is greater than zero"

  "x+y>=90"

  "w is either negative or equal to 10 is true"

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

Multiway Branches

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

If case=1 DO A1 ELSE

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

Inputs

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

  Input Vector.

Predicate Expressions

Predicate Interpretation

- The simplest predicate depends only on input variables.
- For example if x1,x2 are inputs, the predicate might be x1+x2>=7, given the values of x1 and x2 the direction taken through the decision is based on the predicate is determined at input time and does not depend on processing.
- Another example, assume a predicate x1+y>=0 that along a path prior to reaching this predicate we had the assignment statement y=x2+7. although our predicate depends on processing we can substitute the symbolic expression for y to obtain an equivalent predicate x1+x2+7>=0.
- The act of symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called **predicate interpretation.**

Predicate Interpretation-continued.,

- Some times the interpretation may depend on the path; for example,
- Input x
  On x goto a,b,c
  a: z:=7 @ goto hem
  b: z:=-7 @ goto hem
  c: z:=0 @ goto hem
  ……
  HEM: do some thing
  ……
  HEN: if y+z>0 goto ELL else goto EMM
- The predicate interpretation at HEN depends on the path we took through the first multiway branch. It yields for the three cases respectively, if Y+7>0, y-7>0, y>0.
- The path predicates are the specific form of the predicates of the decisions along the selected path after interpretation.

Independence of Variables and Predicates

- The path predicates take on truth values based on the values of input variables, either directly or indirectly.
- If a variable's value does not change as a result of processing, that **variable** is **independent** of the processing.
- If the variable's value can change as a result of the processing the **variable** is **dependent**.
- A **predicate** whose truth value can change as a result of the processing is said to be **process dependent** and one whose truth value does not change as a result of the processing is **process independent.**
- Process dependence of a predicate does not always follow from dependence of the input variables on which that predicate is based.

Correlation of Variables and Predicates

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

Path Predicate Expressions

- A path predicate expression is a set of boolean expressions, all of which must be satisfied to achieve the selected path.
- Example: $X_1+3X_2+17>=0$

  $X_3=17$

  $X_4-X_1>=14\ X_2$
- Any set of input values that satisfy all of the conditions of the path predicate expression will force the routine to the path.
- Some times a predicate can have an OR in it.
  - **Example if X5>0 . OR. X6<0**
  - **This will give two set of expressions, either of which if solved forces the path.**

A: $X_5>0$ **E:$X_6<0$**

B: $X_1+3\ X_2+17>=0$ **B: $X_1+3\ X_2+17>=0$**

C:$X_3=17$ **C: $X_3=17$**

D:$X_4-X_1>=14\ X_2$ **D: $X_4-X_1>=14\ X_2$**

- Boolean algebra notation to denote the boolean expression:

ABCD+EBCD=(A+E)BCD

Predicate Coverage

- Compound Predicate: predicates of the form A OR B, A AND B and more complicated boolean expressions are called as compound predicates.
- Some times even a simple predicate becomes compound after interpretation. Example: the predicate if (x=17) whose opposite branch is if x.NE.17 which is equivalent to x>17 . Or. X<17.

Predicate Coverage

- Predicate coverage is being the achieving of all possible combinations of truth values corresponding to the selected path have been explored under some test.
- As achieving the desired direction at a given decision could still hide bugs in the associated predicates.

Testing Blindness

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

Assignment Blindness

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

Correct	buggy
X:=7	x:=7
…….	………..
If y>0 then	if x+y>0 then

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

Equality Blindness

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

Correct	buggy
If y:=2 then	if y:=2 then
…….	…….
If x+y>3 then	if x>1 then

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

Self Blindness

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

Correct	buggy
X:=a	x:=a
…….	…….
If x-1>0 then	if x+a-2>0 then

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

Path Sensitizing

- We want to select and test enough paths to achieve a satisfactory notion of test completeness such as C1 and C2.
- Extract the programs control flowgraph and select a set of tentative covering paths.
- For any path in that set, interpret the predicates along the path as needed to express them in terms of the input vector. In general individual predicates are compound or may become compound as a result of interpretation.
- Trace the path through, multiplying the individual compound predicates to achieve a boolean expression such as (A+BC) (D+E) (FGH) (IJ) (K) (L)
- Multiply out the expression to achieve a sum of products form:

  ADFGHIJKL+AEFGHIJKL+BCDFGHIJKL+BCEFGHIJKL

Path Sensitizing-continued.,

- Each product term denotes a set of inequalities that if solved will yield an input vector that will drive the routine along the designated path.
- Solve any one of the inequality sets for the chosen path and you have found a set of input values for the path.
- If you can find a solution, then the path is achievable.
- If you cant find a solution to any of the sets of inequalities, the path is un achievable.
- The act of finding a set of solutions to the path predicate expression is called PATH SENSITIZATION

Heuristic Procedures for Sensitizing Paths

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

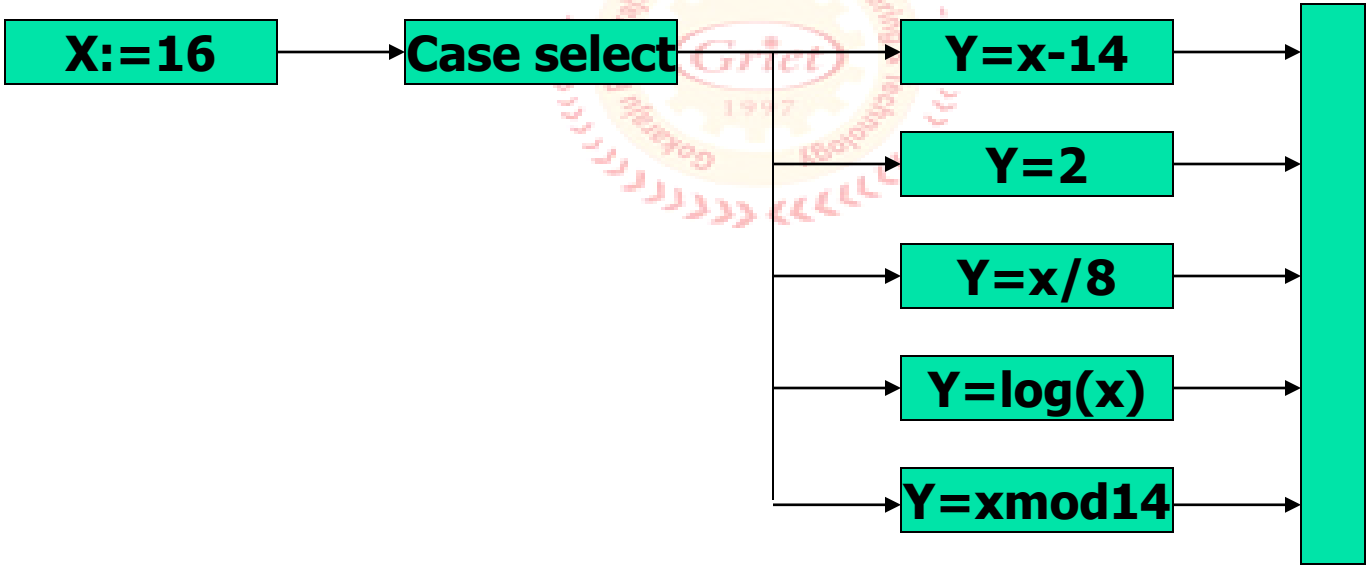
- Identify all variables that affect the decision.
- Classify the predicates as dependent or independent.
- Start the path selection with un correlated, independent predicates.
- If coverage has not been achieved using independent uncorrelated predicates, extend the path set using correlated predicates.
- If coverage has not been achieved extend the cases to those that involve dependent predicates.
- Last, use correlated, dependent predicates.

Path Instrumentation

- Path instrumentation is what we have to do to confirm that the outcome was achieved by the intended path.
- Types of instrumentation methods are:
  - An interpretive trace program
  - Traversal marker or link marker
  - The two link marker method

Co- incidental Correctness

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



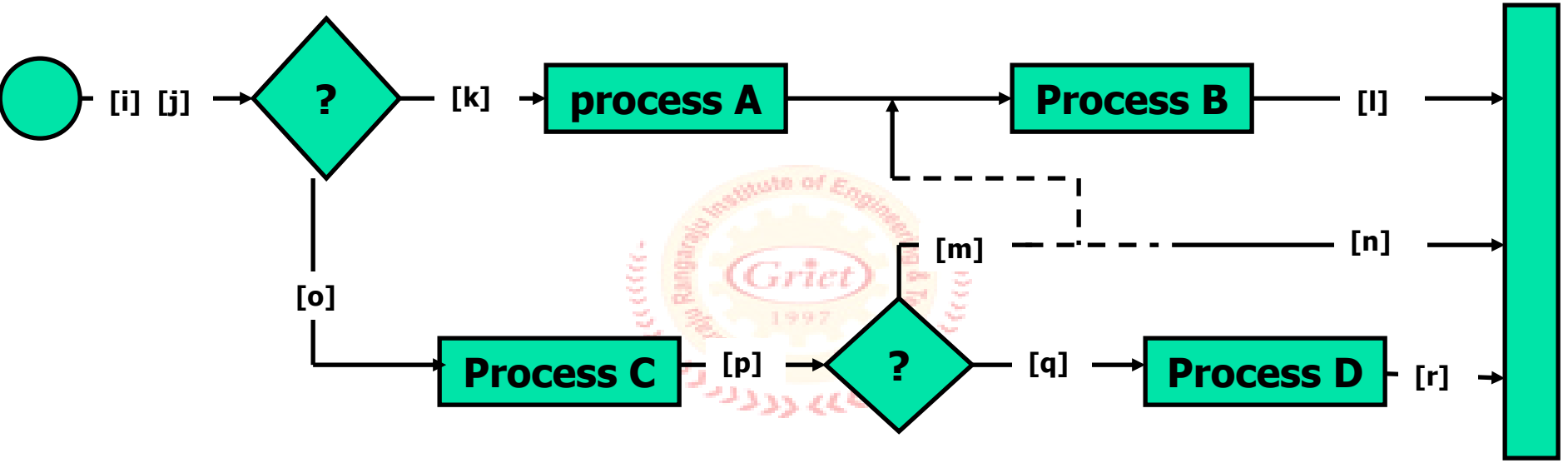
Interpretive trace program

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

Traversal marker or link marker

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

Example - Two link marker



Chapter - 4

TRANSACTION FLOW TESTING

Transaction flow Graphs

- Transaction flows are introduced as a representation of a system's processing.
- The methods that were applied to control flow graphs are then used for functional testing.
- Transaction flows and transaction flow testing are to the independent system tester what control flows and path testing are to the programmer.

Transaction flow Graphs-continued.,

- The transaction flow graph is to create a behavioral model of the program that leads to functional testing.
- The transaction flowgraph is a model of the structure of the system's behavior (functionality).

Transaction Flows

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

Example of a Transaction

- A transaction for an online information retrieval system might consist of the following steps or tasks:
  - Accept input (tentative birth)
  - Validate input (birth)
  - Transmit acknowledgement to requester
  - Do input processing
  - Search file
  - Request directions from user
  - Accept input
  - Validate input
  - Process request
  - Update file
  - Transmit output
  - Record transaction in log and clean up (death)

Complications in transaction flow graphs

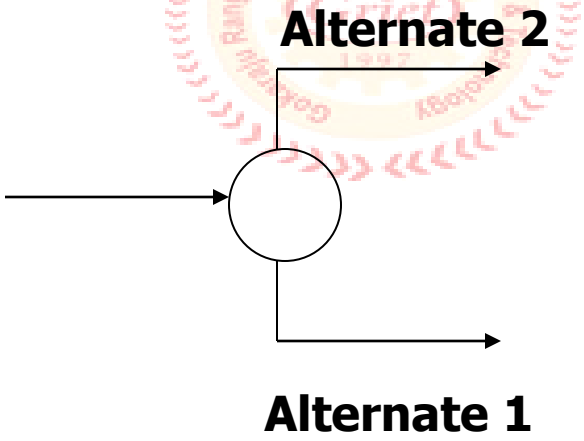
- In simple cases, the transactions have a unique identity from the time they're created to the time they're completed.
- In many systems the transactions can give birth to others, and transactions can also merge.

Births

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

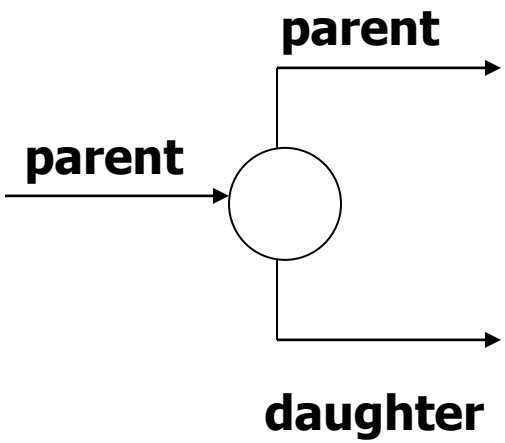
Decision

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



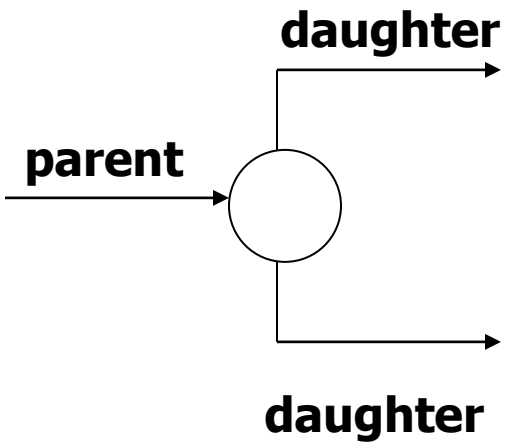
Biosis

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



Mitosis

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

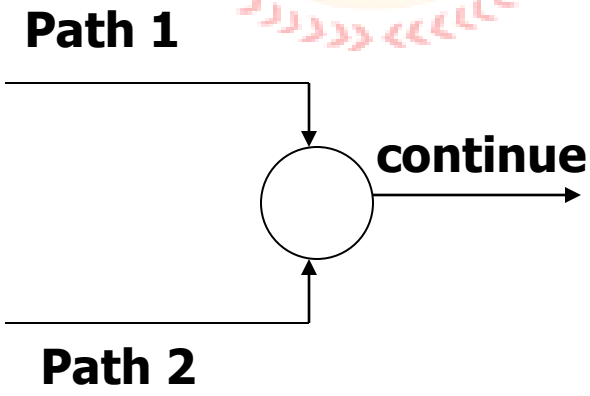


Mergers

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

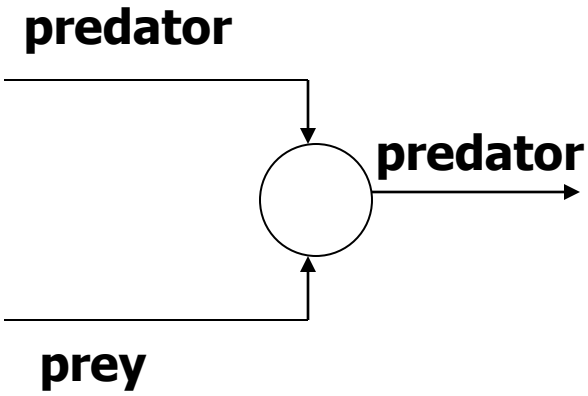
Ordinary junction

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



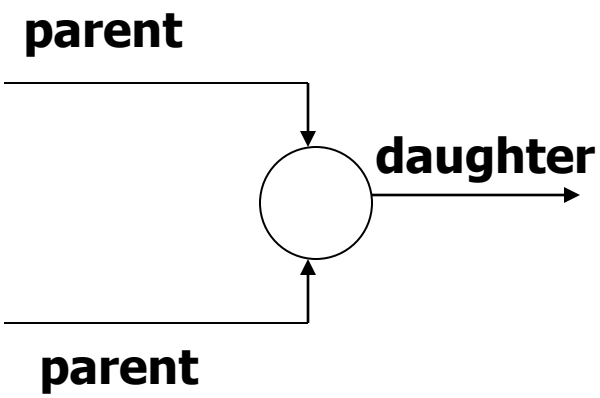
Absorption

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



Conjugation

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



Transaction flow testing techniques

- Get the transaction flows:

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

Inspections, reviews, walkthroughs

- Transaction flows are natural agenda for system reviews or inspections.
- 1. In conducting the walkthroughs, you should:
  - A. discuss enough transaction types to account for 98%-99% of the transaction the system is expected to process.
  - B. discuss paths through flows in functional rather than technical terms.
  - Ask the designers to relate every flow to the specification and to show how that transaction, directly or indirectly, follows from the requirements.

Inspections, reviews, walkthroughs-continued.,

- 2. make transaction flow testing the corner stone of system functional testing just as path testing is the corner stone of unit testing.
- 3. select additional flow paths for loops, extreme values, and domain boundaries.
- 4. design more test cases to validate all births and deaths.
- 5. publish and distribute the selected test paths through the transaction flows as early as possible so that they will exert the maximum beneficial effect on the project.

Path Selection

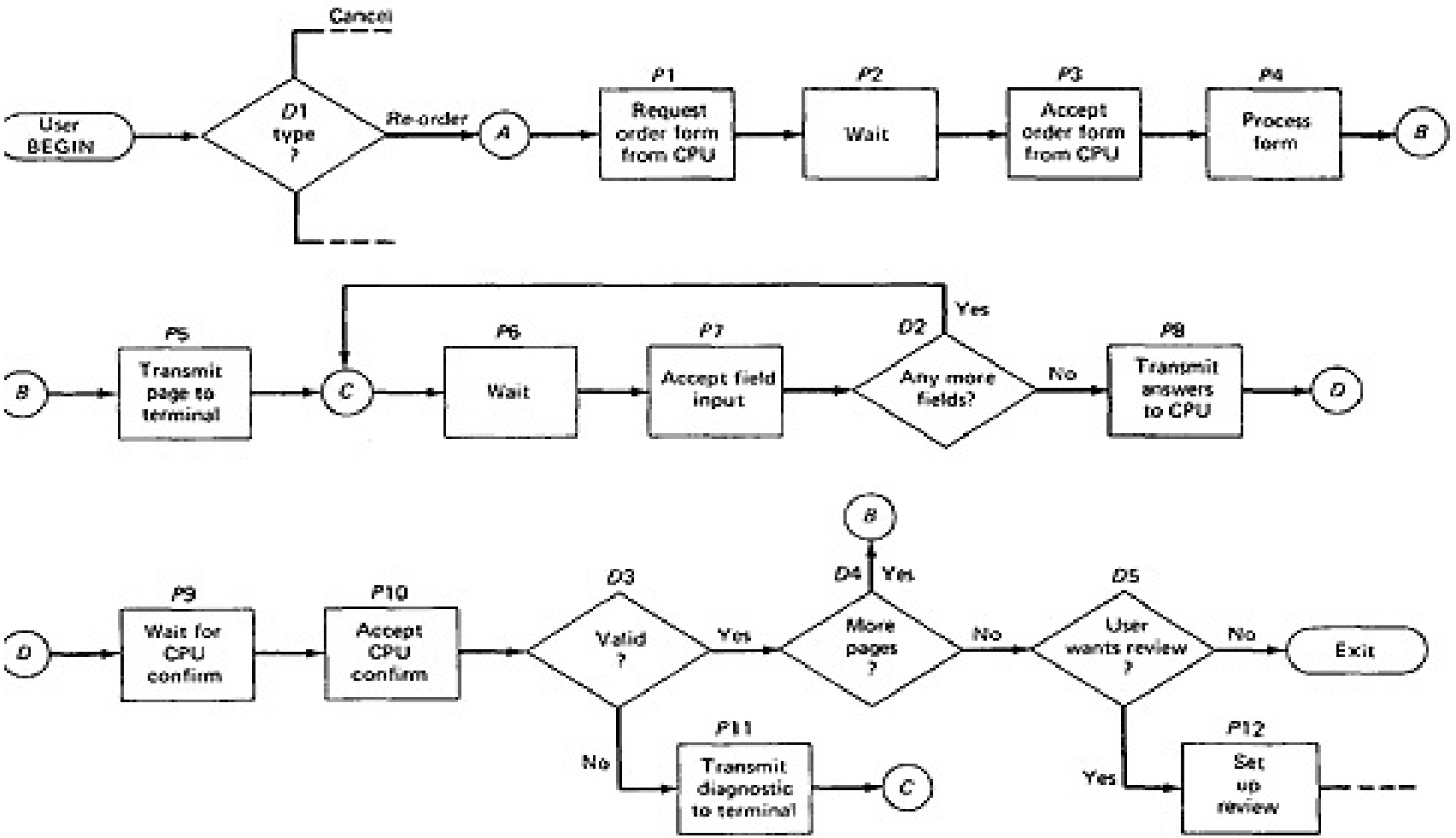
- Select a set of covering paths (c1+c2) using the analogous criteria you used for structural path testing.
- Select a covering set of paths based on functionally sensible transactions as you would for control flow graphs.
- Try to find the most tortuous, longest, strangest path from the entry to the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

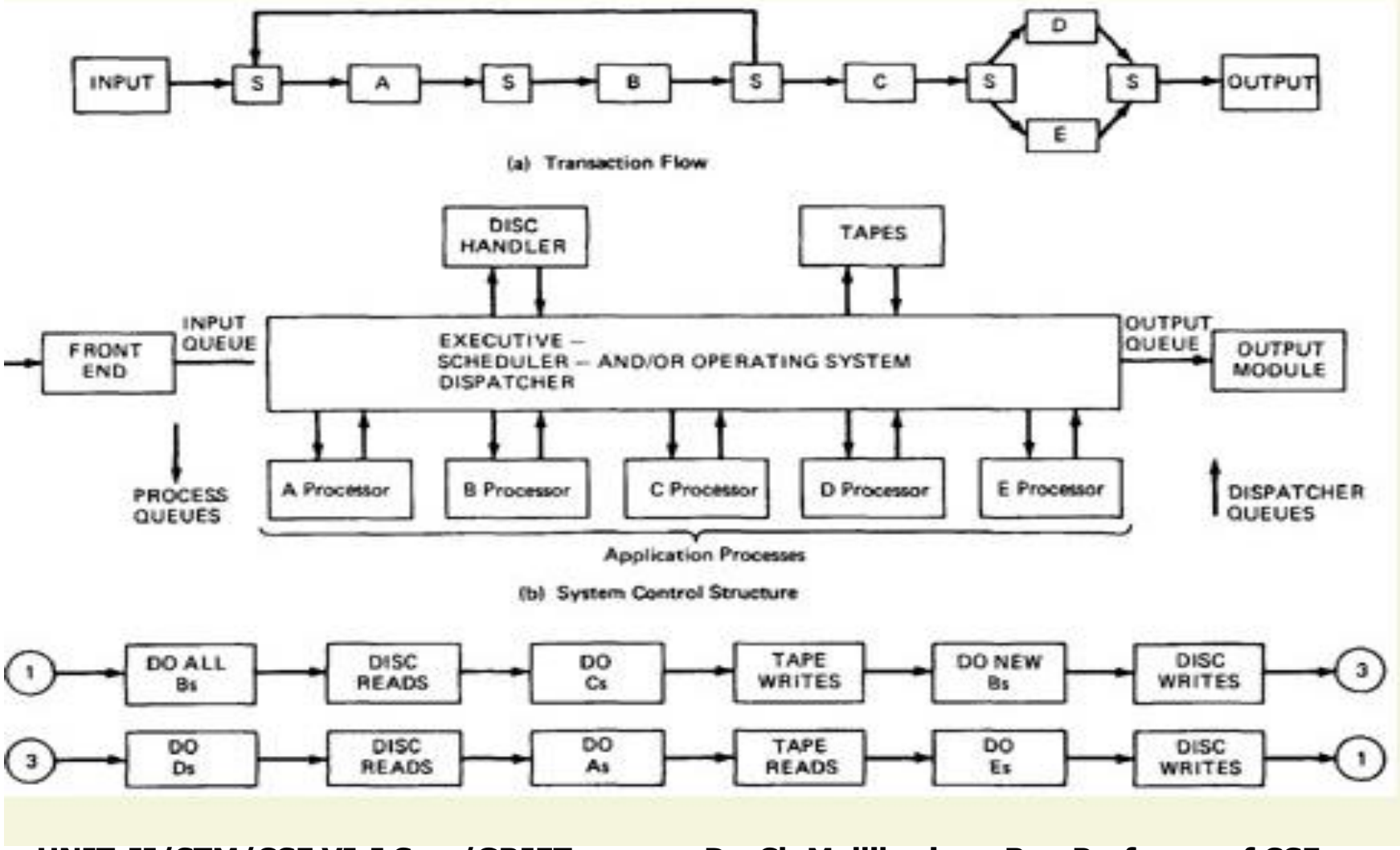
Sensitization

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

Instrumentation

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


Cancel
User BEGIN
D1 type ?
Re-order
A
P1
Request order form from CPU
P2
Wait
P3
Accept order form from CPU
P4
Process form
B
B
P5
Transmit page to terminal
C
P6
Wait
P7
Accept field input
D2
Any more fields?
Yes
No
P8
Transmit answers to CPU
D
D
P9
Wait for CPU confirm
P10
Accept CPU confirm
D3
Valid ?
Yes
No
B
D4
More pages ?
Yes
No
D5
User wants review ?
No
Exit
Yes
P11
Transmit diagnostic to terminal
C
P12
Set up review


INPUT
S
A
S
B
S
C
S
D
E
S
OUTPUT
(a) Transaction Flow
DISC HANDLER
TAPES
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
DISPATCHER QUEUES
Application Processes
(b) System Control Structure
1
DO ALL Bs
DISC READS
DO Cs
TAPE WRITES
DO NEW Bs
DISC WRITES
3
3
DO Ds
DISC READS
DO As
TAPE READS
DO Es
DISC WRITES
1