BASICS OF PATH TESTING:

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

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

- **CONTROL FLOW GRAPHS:**
  - The control flow graph is a graphical representation of a program's control structure. It uses the elements named process blocks, decisions, and junctions.
  - The flow graph is similar to the earlier flowchart, with which it is not to be confused.
  - **Flow Graph Elements:**A flow graph contains four different types of elements.
    (1) Process Block (2) Decisions (3) Junctions (4) Case Statements

1. **Process Block:**
    - A process block is a sequence of program statements uninterrupted by either decisions or junctions.

DATA FLOW TESTING

Data flow testing is a white-box testing technique that examines the data flow with respect to the variables used in the code. It examines the initialization of variables and checks their values at each instance.

White box testing is a software testing technique that examines the internal working of the software code being developed.

Types of data flow testing

There are two types of data flow testing:

- **Static data flow testing:** The declaration, usage, and deletion of the variables are examined without executing the code. A control flow graph is helpful in this.
- **Dynamic data flow testing:** The variables and data flow are examined with the execution of the code.

Advantages of data flow testing

Data flow testing helps catch different kinds of anomalies in the code. These anomalies include:

- Using a variable without declaration
- Deleting a variable without declaration
- Defining a variable two times
- Deleting a variable without using it in the code
- Deleting a variable twice
- Using a variable after deleting it
- Not using a variable after defining it

Disadvantages of data flow testing

A few disadvantages of data flow testing are:

- Good knowledge of programming is required for proper testing
- Expensive

- Time consuming

Techniques of data flow testing

Data flow testing can be done using one of the following two techniques:

- Control flow graph
- Making associations between data definition and usages

Control flow graph

A **control flow graph** is a graphical representation of the flow of control, i.e., the order of statements in which they will be executed.

Consider the following piece of pseudo-code:

```
1. input(x)
2. if(x>5)
3.    z = x + 10
4. else
5.    z = x - 5
6. print("Value of Z: ", z)
```

In the above piece of code, if the value of x entered by the user is greater than 5, then the order of execution of statements would be:

1, 2, 3, 6

If the value entered by the user in line 1 is less than or equal to 5, the order of execution of statements would be:

1, 4, 5, 6

Hence, the control flow graph of the above piece of code will be:

Using the above control flow graph and code, we can deduce the table below. This table mentions the node at which the variable was declared and the nodes at which it was used:

Variable Name	Defined At	Used At
x	1	2
z	3, 5	6

We can use the above table to ensure that no anomaly occurs in the code by ensuring multiple tests. E.g., each variable is declared before it is used.

Making associations

In this technique, we make associations between two kinds of statements:

- Where variables are defined
- Where those variables are used

An association is made with this format:

(line number where the variable is declared, line number where the variable is used, name of the variable)

For example, (1, 3, x) would mean that the variable 'x' is defined on line 1 and used on line 3.

Now, consider the following piece of pseudo-code:

```
1. input(x)
2. if(x>5)
3.    z = x + 10
4. else
5.    z = x - 5
6. print("Value of Z: ", z)
```

For the above snippet of pseudo-code, we will make the following associations:

- (1, (2,t), x): for the true case of IF statement in line 2
- (1, (2,f), x): for the false case of IF statement in line 2
- (1, 3, x): variable x is being used in line 3 to define the value of z
- (1, 5, x): variable x is being used in line 5 to define the value of z
- (3, 6, z): variable z is being used in line 6, which is defined in line 3
- (5, 6, z): variable z is being used in line 6, which is defined in line 5

The first two associations are for the IF statement on line 2. One association is made if the condition is true, and the other is for the false case.

Now, there are two types of uses of a variable:

- **predicate use**: the use of a variable is called p-use. Its value is used to decide the flow of the program, e.g., line 2.
- **computational use**: the use of a variable is called c-use when its value is used compute another variable or the output, e.g., line 3.

After the associations are made, these associations can be divided into these groups:

- All definitions coverage
- All p-use coverage
- All c-use coverage
- All p-use, some c-use coverage
- All c-use, some p-use coverage
- All uses coverage

Once the associations are divided into these groups, the tester makes test cases and examines each point.

The statements and variables which are found to be extra are removed from the code.

Figure 2.1: Flowgraph Elements

CONTROL FLOW GRAPHS Vs FLOWCHARTS:

A program's flow chart resembles a control flow graph.

In flow graphs, we don't show the details of what is in a process block.

In flow charts every part of the process block is drawn.

The flowchart focuses on process steps, where as the flow graph focuses on control flow of the program.

The act of drawing a control flow graph is a useful tool that can help us clarify the control flow and data flow issues.

NOTATIONAL EVOULTION:

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.
- To understand this, we will go through an example (Figure 2.2) written in a FORTRAN like programming language called **Programming Design Language (PDL)**. The program's corresponding flowchart (Figure 2.3) and flowgraph (Figure 2.4) were also provided below for better understanding.
- The first step in translating the program to a flowchart is shown in Figure 2.3, where we have the typical one-for-one classical flowchart. Note that complexity has increased, clarity has decreased, and that we had to add auxiliary labels (LOOP, XX, and YY), which have no actual program counterpart. In Figure 2.4 we merged the process steps and replaced them with the single process box. We now have a control flowgraph. But this representation is still too busy. We simplify the notation further to achieve Figure 2.5, where for the first time we can really see what the control flow looks like.

CODE* (PDL)

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

* A contrived horror

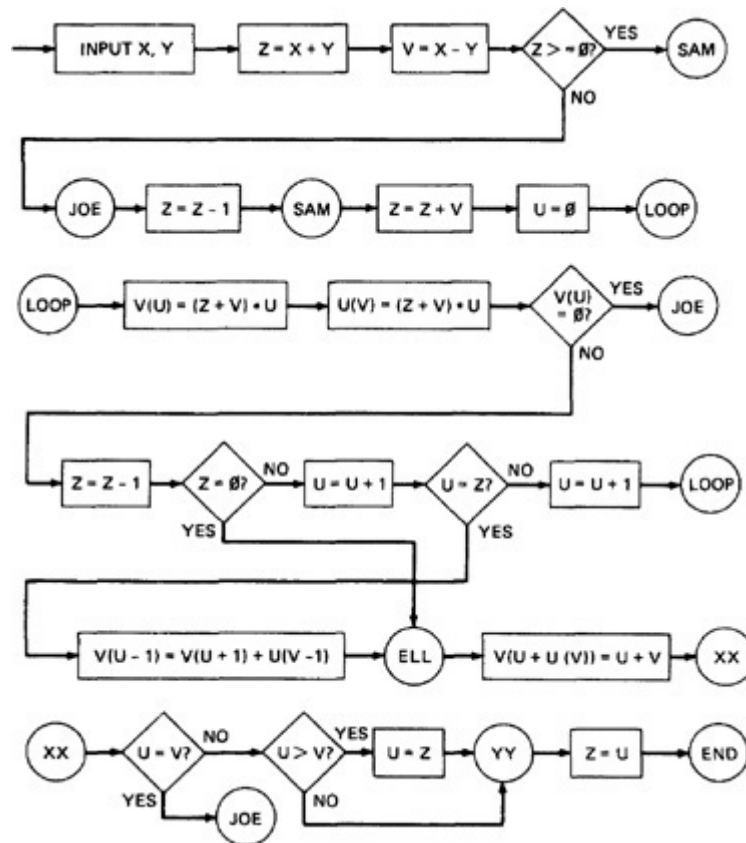*Figure 2.2: Program Example (PDL)*

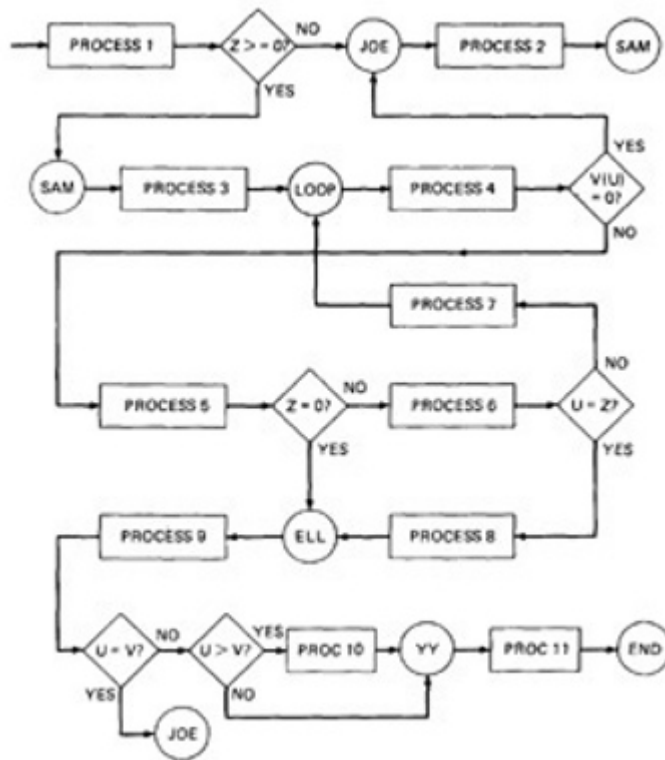*Figure 2.3: One-to-one flowchart for
example program in Figure 2.2*

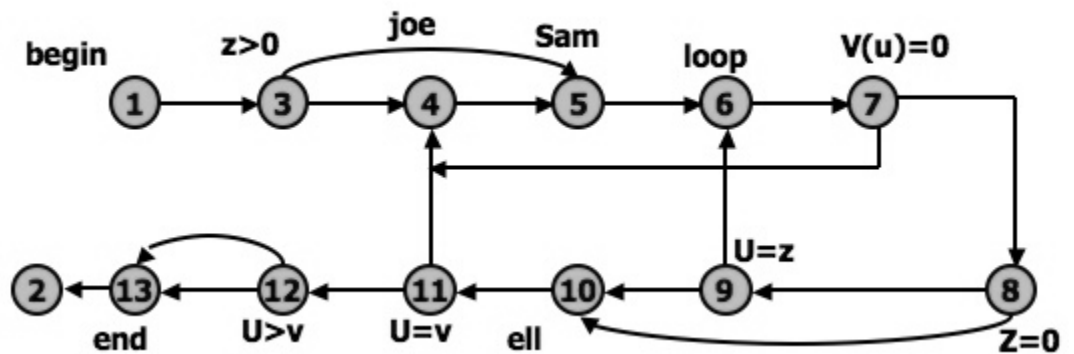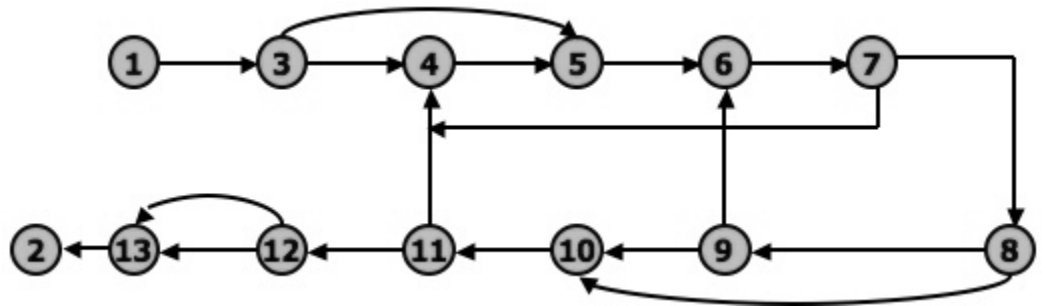**Figure 2.4: Control Flowgraph for example in Figure 2.2**

Figure 2.5: Simplified Flowgraph Notation

Figure 2.6: Even Simplified Flowgraph
Notation

The final transformation is shown in Figure 2.6, where we've dropped the node numbers to achieve an even simpler representation. The way to work with control flowgraphs is to use the simplest possible representation - that is, no more information than you need to correlate back to the source program or PDL.

LINKED LIST REPRESENTATION:

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.

Linked List representation of Flow Graph:

```
1 (BEGIN)    : 3
2 (END)      :              Exit, no outlink
3 (Z>Ø?)     : 4 (FALSE)
             : 5 (TRUE)
4 (JOE)      : 5
5 (SAM)      : 6
6 (LOOP)     : 7
7 (V(U)=Ø?)  : 4 (TRUE)
             : 8 (FALSE)
8 (Z=Ø?)     : 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)
```

Figure 2.7: Linked List Control Flowgraph Notation

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. (See Figure 2.8)
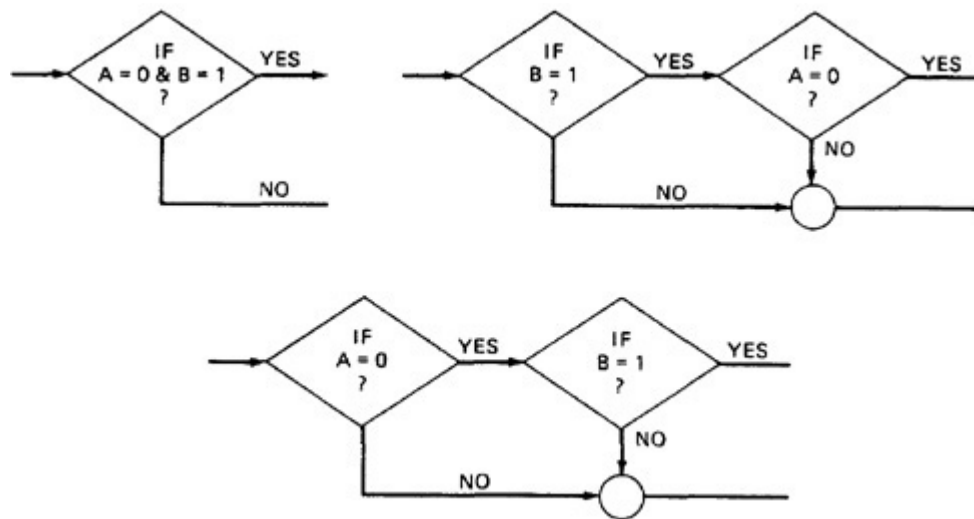
Figure 2.8: Alternative Flowgraphs for same logic
(Statement "IF (A=0) AND (B=1) THEN . . .").

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.

FLOWGRAPH AND FLOWCHART GENERATION:

Flowcharts can be

1. Handwritten by the programmer.
2. Automatically produced by a flowcharting program based on a mechanical analysis of the source code.
3. Semi automatically produced by a flow charting program based in part on structural analysis of the source code and in part on directions given by the programmer.

There are relatively few control flow graph generators.

PATH TESTING - PATHS, NODES AND LINKS:

Path:a path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same junction, decision, or exit.

A path may go through several junctions, processes, or decisions, one or more times.

Paths consists of segments.

The segment is a link - a single process that lies between two nodes.

A path segment is succession of consecutive links that belongs to some path.

The length of path measured by the number of links in it and not by the number of the instructions or statements executed along that path.

The name of a path is the name of the nodes along the path.

FUNDAMENTAL PATH SELECTION CRITERIA:

There are many paths between the entry and exit of a typical routine.

Every decision doubles the number of potential paths. And every loop multiplies the number of potential paths by the number of different iteration values possible for the loop.

Defining complete testing:

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

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.

*EXAMPLE:*Here is the correct version.
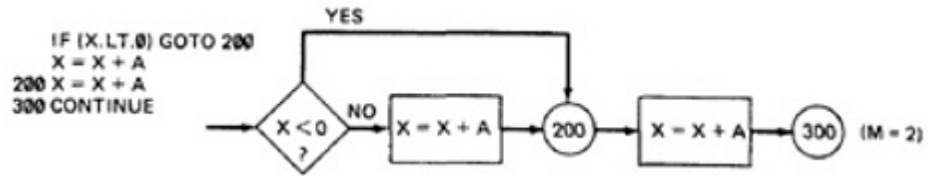
For X negative, the output is X + A, while for X greater than or equal to zero, the output is X + 2A. Following prescription 2 and executing every statement, but not every branch, would not reveal the bug in the following incorrect version:
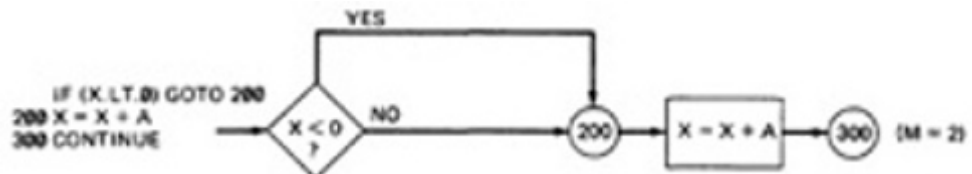
A negative value produces the correct answer. Every statement can be executed, but if the test cases do not force each branch to be taken, the bug can remain hidden. The next example uses a test based on executing each branch but does not force the execution of all statements:
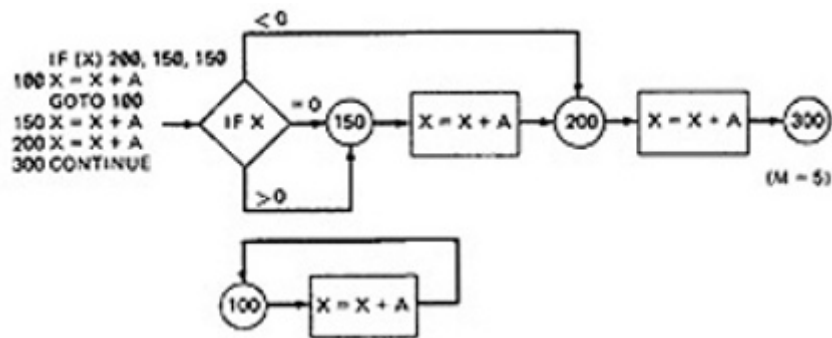
The hidden loop around label 100 is not revealed by tests based on prescription 3 alone because no test forces the execution of statement 100 and the following GOTO statement. Furthermore, label 100 is not flagged by the compiler as an unreferenced label and the subsequent GOTO does not refer to an undefined label.

A **Static Analysis** (that is, an analysis based on examining the source code or structure) cannot determine whether a piece of code is or is not reachable. There could be subroutine calls with parameters that are subroutine labels, or in the above example there

could be a GOTO that targeted label 100 but could never achieve a value that would send the program to that label.

Only a **Dynamic Analysis** (that is, an analysis based on the code's behavior while running - which is to say, to all intents and purposes, testing) can determine whether code is reachable or not and therefore distinguish between the ideal structure we think we have and the actual, buggy structure.

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 (P_{inf}):**
   - 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.

2. **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 (unprincipled or can not be accepted) and should be criminalized.

3. **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.

Commonsense 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 is 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 (practical) design of tests. It is better to make many simple paths than a few complicated paths.
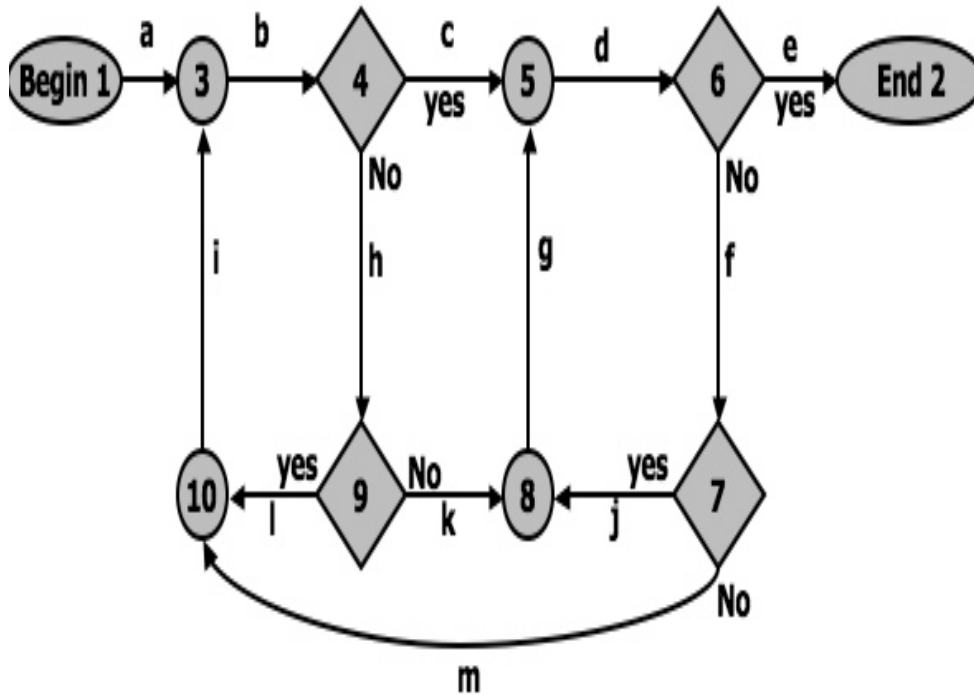
▪ **Path Selection Example:**

Figure 2.9: An example flowgraph to explain path selection

▪ **Practical Suggestions in Path Testing:**
1. Draw the control flow graph on a single sheet of paper.
2. Make several copies - as many as you will need for coverage (C1+C2) and several more.
3. Use a yellow highlighting marker to trace paths. Copy the paths onto a master sheets.
4. Continue tracing paths until all lines on the master sheet are covered, indicating that you appear to have achieved C1+C2.
5. As you trace the paths, create a table that shows the paths, the coverage status of each process, and each decision.
6. The above paths lead to the following table considering Figure 2.9:

PATHS	DECISIONS				PROCESS–LINK												
	4	6	7	9	a	b	c	d	e	f	g	h	i	j	k	l	m
abcde	YES	YES			✓	✓	✓	✓	✓								
abhkgde	NO	YES	NO		✓	✓		✓	✓		✓	✓			✓		
abhlibcde	NO,YES	YES		YES	✓	✓	✓	✓	✓			✓	✓			✓	
abcdfjgde	YES	NO,YES	YES		✓	✓	✓	✓	✓	✓	✓			✓			
abcdfmibcde	YES	NO,YES	NO		✓	✓	✓	✓	✓	✓			✓				✓

After you have traced a a covering path set on the master sheet and filled in the table for every path, check the following:

1. Does every decision have a YES and a NO in its column? (C2)
2. Has every case of all case statements been marked? (C2)
3. Is every three - way branch (less, equal, greater) covered? (C2)
4. Is every link (process) covered at least once? (C1)

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

7. **LOOPS:**
   - **Cases for a single loop:** A Single loop can be covered with two cases: Looping and Not looping. But, experience shows that many loop-related bugs are not discovered by C1+C2. Bugs hide themselves in corners and congregate at boundaries - in the cases of loops, at or around the minimum or maximum number of times the loop can be iterated. The minimum number of iterations is often zero, but it need not be.

   CASE 1: Single loop, Zero minimum, N maximum, No excluded values

   1. Try bypassing the loop (zero iterations). If you can't, you either have a bug, or zero is not the minimum and you have the wrong case.
   2. Could the loop-control variable be negative? Could it appear to specify a negative number of iterations? What happens to such a value?
   3. One pass through the loop.
   4. Two passes through the loop.
   5. A typical number of iterations, unless covered by a previous test.
   6. One less than the maximum number of iterations.
   7. The maximum number of iterations.
   8. Attempt one more than the maximum number of iterations. What prevents the loop-control variable from having this value? What will happen with this value if it is forced?

   CASE 2: Single loop, Non-zero minimum, No excluded values

   9. Try one less than the expected minimum. What happens if the loop control variable's value is less than

the minimum? What prevents the value from being less than the minimum?

10. The minimum number of iterations.
11. One more than the minimum number of iterations.
12. Once, unless covered by a previous test.
13. Twice, unless covered by a previous test.
14. A typical value.
15. One less than the maximum value.
16. The maximum number of iterations.
17. Attempt one more than the maximum number of iterations.

CASE 3: Single loops with excluded values

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

- **Kinds of Loops:** There are only three kinds of loops with respect to path testing:
  - **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 used to discard some of these values:
      1. Start at the inner most loop. Set all the outer loops to their minimum values.
      2. 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.
      3. If you've done the outmost loop, GOTO step 5, else move out one loop and set it up as in step 2 with all other loops set to typical values.
      4. Continue outward in this manner until all loops have been covered.
      5. Do all the 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.

a, b) Nested Loops

c) Concatenated Loops

d, e, f) Horrible Loops
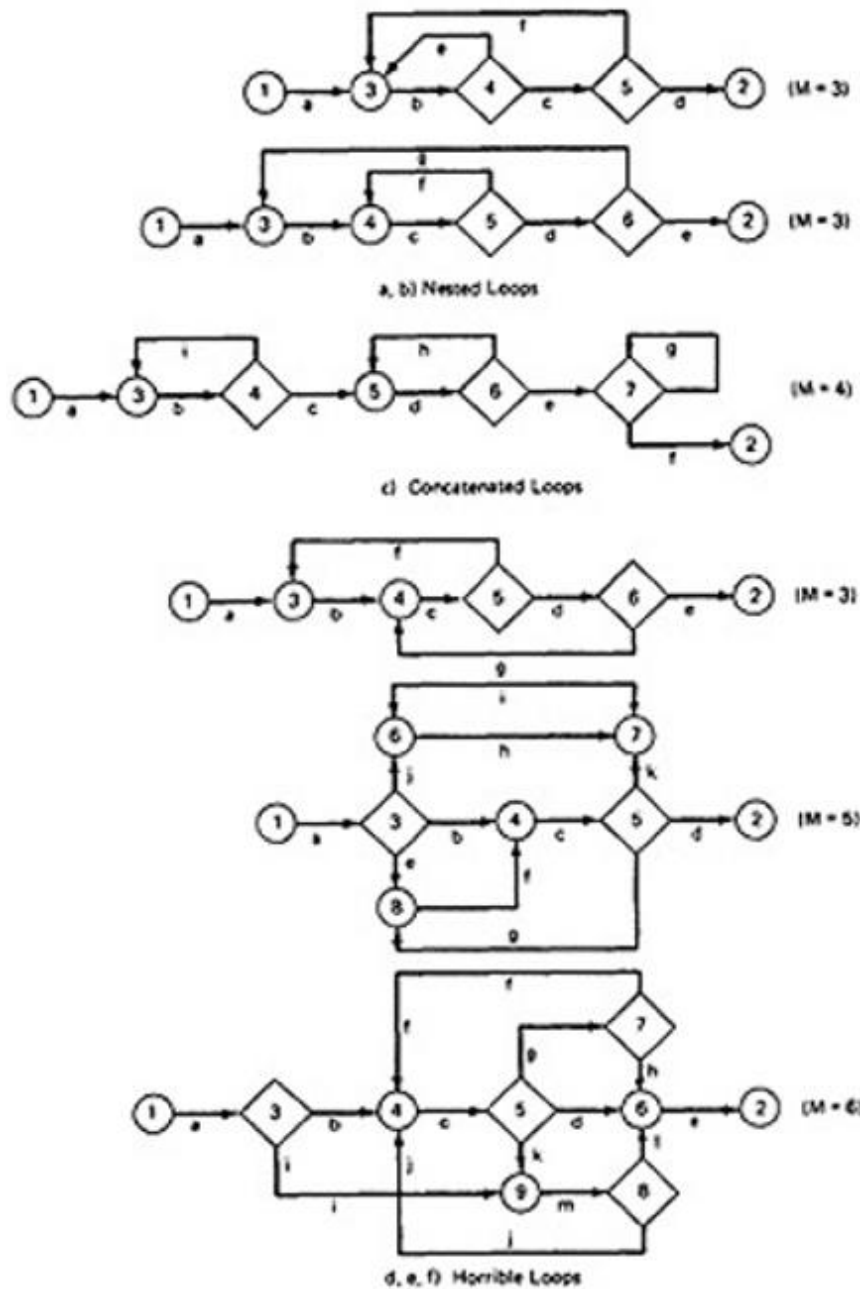
Figure 2.10: Example of Loop types

- **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, PATH PREDICATES AND ACHIEVABLE PATHS:

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

PATH PREDICATE: 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 (practical approach) 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 INTERPRETATION:
- The simplest predicate depends only on input variables.
- For example if x_1,x_2 are inputs, the predicate might be $x_1+x_2>=7$, given the values of x_1 and x_2 the direction taken through the decision is based on the predicate is determined at input time and does not depend on processing.
- Another example, assume a predicate $x_1+y>=0$ that along a path prior to reaching this predicate we had the assignment statement $y=x_2+7$. although our predicate depends on processing, we can substitute the symbolic expression for y to obtain an equivalent predicate $x_1+x_2+7>=0$.

- The act of symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called **predicate interpretation.**
- Some times the interpretation may depend on the path; for example,
- `INPUT X`
- `ON X GOTO A, B, C, ...`
- `A: Z := 7 @ GOTO HEM`
- `B: Z := -7 @ GOTO HEM`
- `C: Z := 0 @ GOTO HEM`
- `.........`
- `HEM: DO SOMETHING`
- `.........`
- `HEN: IF Y + Z > 0 GOTO ELL ELSE GOTO EMM`

The predicate interpretation at HEN depends on the path we took through the first multiway branch. It yields for the three cases respectively, if Y+7>0, Y-7>0, Y>0.

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

INDEPENDENCE OF VARIABLES AND PREDICATES:

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

CORRELATION OF VARIABLES AND PREDICATES:

- Two variables are correlated if every combination of their values cannot be independently specified.
- Variables whose values can be specified independently without restriction are called uncorrelated.
- A pair of predicates whose outcomes depend on one or more variables in common are said to be correlated predicates. For example, the predicate X==Y is followed by another predicate X+Y == 8. If we select X and Y values to satisfy the first predicate, we might have forced the 2nd predicate's truth value to change.
- Every path through a routine is achievable only if all the predicates in that routine are uncorrelated.

PATH PREDICATE EXPRESSIONS:

- A path predicate expression is a set of boolean expressions, all of which must be satisfied to achieve the selected path.
- Example:
- `X1+3X2+17>=0`
- `X3=17`
- `X4-X1>=14X2`

- 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:

A: X5 > 0   E: X6 < 0
B: X1 + 3X2 + 17 >= 0 B: X1 + 3X2 + 17 >= 0

C: X3 = 17   C: X3 = 17
D: X4 - X1 >= 14X2   D: X4 - X1 >= 14X2

-
- 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 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 (harmful) situation in which the desired path is achieved for the wrong reason.
- There are three types of Testing Blindness:
  1. **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.
  2. **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.

3. **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:

REVIEW: ACHIEVABLE AND UNACHIEVABLE PATHS:
- We want to select and test enough paths to achieve a satisfactory notion of test completeness such as C1+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

$$(A+BC) (D+E) (FGH) (IJ) (K) (I) (L).$$

- Multiply out the expression to achieve a sum of products form:

$$ADFGHIJKL+AEFGHIJKL+BCDFGHIJKL+BCEFGHIJKL$$

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

- The act of finding a set of solutions to the path predicate expression is called **PATH SENSITIZATION**.

HEURISTIC PROCEDURES FOR SENSITIZING PATHS:

- This is a workable approach, instead of selecting the paths without considering how to sensitize, attempt to choose a covering path set that is easy to sensitize and pick hard to sensitize paths only as you must to achieve coverage.
- Identify all variables that affect the decision.
- Classify the predicates as dependent or independent.
- Start the path selection with un correlated, independent predicates.
- If coverage has not been achieved using independent uncorrelated predicates, extend the path set using correlated predicates.
- If coverage has not been achieved extend the cases to those that involve dependent predicates.
- Last, use correlated, dependent predicates.

PATH INSTRUMENTATION:

PATH INSTRUMENTATION:

- Path instrumentation is what we have to do to confirm that the outcome was achieved by the intended path.
- **Co-incidental Correctness:** The coincidental correctness stands for achieving the desired outcome for wrong reason.
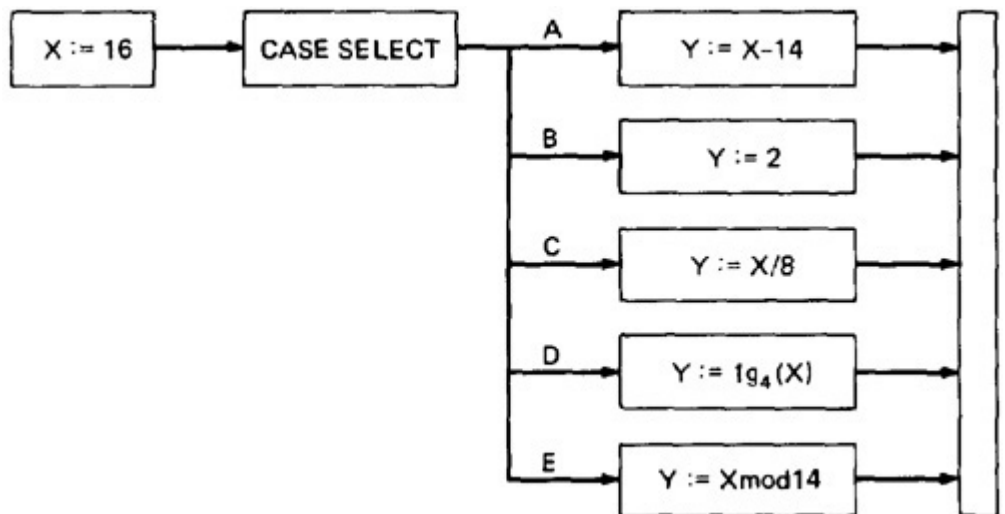
Figure 2.11: Coincidental Correctness

The above figure is an example of a routine that, for the (unfortunately) chosen input value (X = 16), yields the same outcome (Y = 2) no matter which case we select. Therefore, the tests chosen this way will not tell us whether we have achieved coverage. For example, the five cases could be totally jumbled and still the outcome would be the same. **Path Instrumentation** is what we have to do to confirm that the outcome was achieved by the intended path.

- The types of instrumentation methods include:
  1. **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.
- If we run the tested routine under a trace, then we have all the information we need to confirm the outcome and, furthermore, to confirm that it was achieved by the intended path.
- The trouble with traces is that they give us far more information than we need. In fact, the typical trace program provides so much information that confirming the path from its massive output dump is more work than simulating the computer by hand to confirm the path.
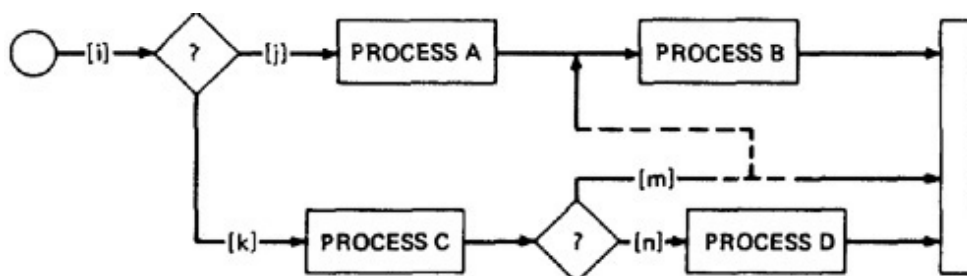
2. **Traversal Marker or Link Marker:**
    - A simple and effective form of instrumentation is called a traversal marker or link marker.
    - Name every link by a lower case letter.
    - Instrument the links so that the link's name is recorded when the link is executed.
    - The succession of letters produced in going from the routine's entry to its exit should, if there are no bugs, exactly correspond to the path name.
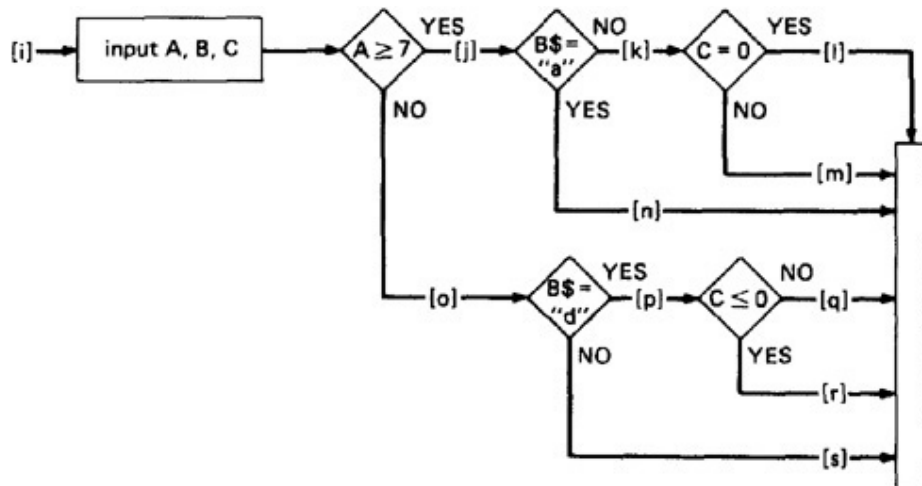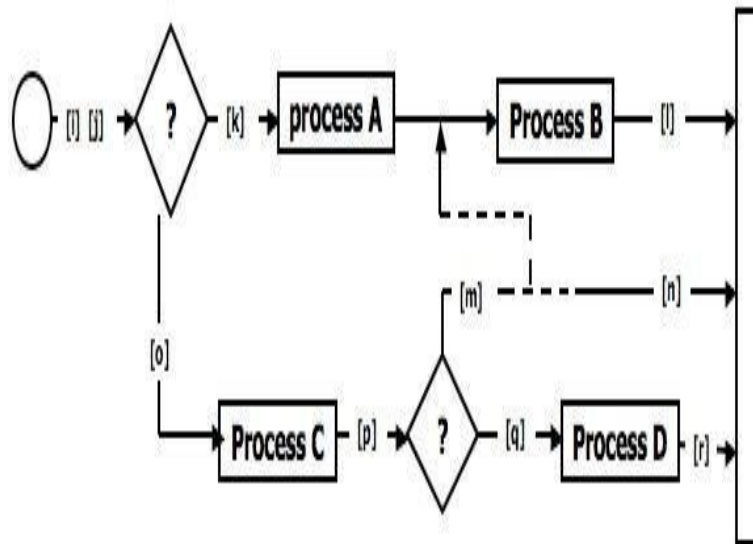

3. *Figure 2.12: Single Link Marker Instrumentation*

- **Why Single Link Markers aren't enough:** Unfortunately, a single link marker may not do the trick because links can be chewed by open bugs.

4. *Figure 2.13: Why Single Link Markers aren't enough.*

5. We intended to traverse the ikm path, but because of a rampaging GOTO in the middle of the m link, we go to process B. If coincidental correctness is against us, the outcomes will be the same and we won't know about the bug.

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

1. *Figure 2.14: Double Link Marker Instrumentation.*

2. **Link Counter:** A less disruptive (and less informative) instrumentation method is based on counters. Instead of a unique link name to be pushed into a string when the link is traversed, we simply increment a link counter. We now confirm that the path length is as expected. The same problem that led us to double link markers also leads us to double link counters.

UNIT-2(PART 2)

TRANSACTION FLOW TESTING AND DATA FLOW TESTING:

This unit gives an indepth overview of two forms of functional or system testing namely Transaction Flow Testing and Data Flow Testing.

At the end of this unit, the student will be able to:

- Understand the concept of transaction flow testing and data flow testing.
- Visualize the transaction flow and data flow in a software system.
- Understand the need and appreciate the usage of the two testing methods.
- Identify the complications in a transaction flow testing method and anomalies in data flow testing.
- Interpret the data flow anomaly state graphs and control flow grpahs and represent the state of the data objetcs.
- Understand the limitations of Static analysis in data flow testing.
- Compare and analyze various strategies of data flow testing.

TRANSACTION FLOWS:

- ### *INTRODUCTION:*
    - 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)
- ### *TRANSACTION FLOW GRAPHS:*
    - Transaction flows are introduced as a representation of a system's processing.
    - The methods that were applied to control flow graphs are then used for functional testing.
    - Transaction flows and transaction flow testing are to the independent system tester what control flows are path testing are to the programmer.
    - 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).
    - An example of a Transaction Flow is as follows:
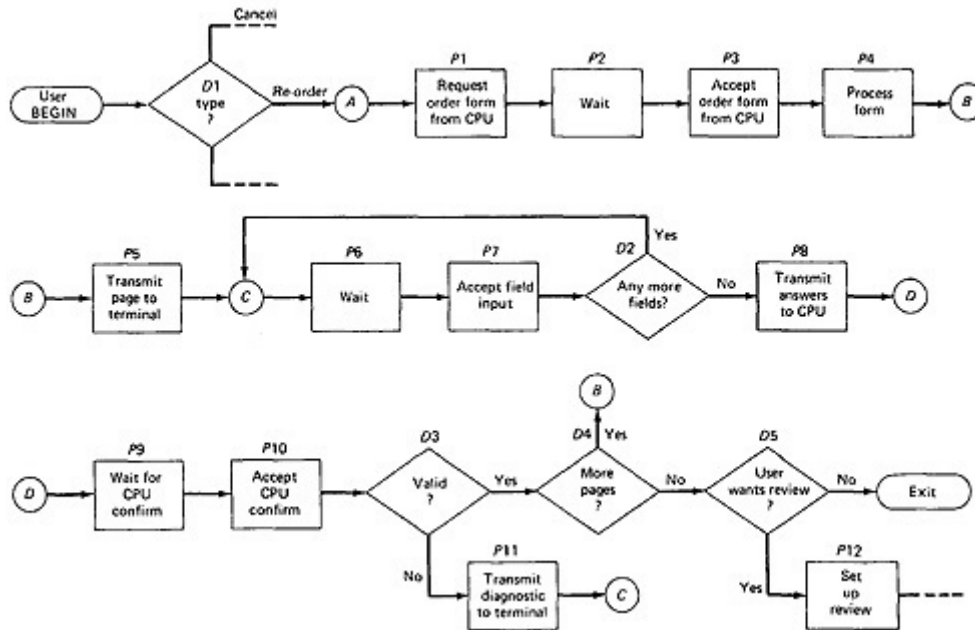
Figure 3.1: An Example of a Transaction Flow

- *USAGE:*
  - Transaction flows are indispensable for specifying requirements of complicated systems, especially online systems.
  - A big system such as an air traffic control or airline reservation system, has not hundreds, but thousands of different transaction flows.
  - The flows are represented by relatively simple flowgraphs, many of which have a single straight-through path.
  - Loops are infrequent compared to control flowgraphs.
  - The most common loop is used to request a retry after user input errors. An ATM system, for example, allows the user to try, say three times, and will take the card away the fourth time.
- *COMPLICATIONS:*
  - 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. It can be a Decision, Biosis or a Mitosis.

1.   **Decision:**Here the transaction will take one alternative or the other alternative but not both. (See Figure 3.2 (a))

2.   **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. (See Figure 3.2 (b))

3.   **Mitosis:**Here the parent transaction is destroyed and two new transactions are created.(See Figure 3.2 (c))
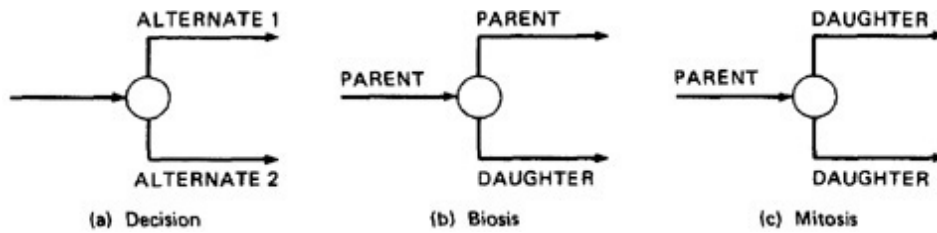
Figure 3.2: Nodes with multiple outlinks

Mergers: Transaction flow junction points are potentially as troublesome as transaction flow splits. There are three types of junctions: (1) Ordinary Junction (2) Absorption (3) Conjugation

1.   **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. (See Figure 3.3 (a))

2.   **Absorption:** In absorption case, the predator transaction absorbs prey transaction. The prey gone but the predator retains its identity. (See Figure 3.3 (b))

3.   **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.(See Figure 3.3 (c))
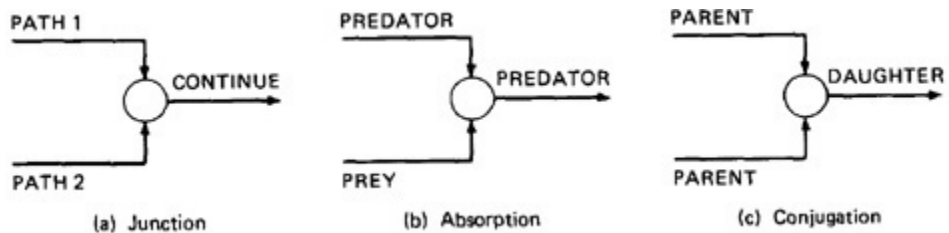
Figure 3.3: Transaction Flow Junctions and
Mergers

We have no problem with ordinary decisions and junctions. Births, absorptions, and conjugations are as problematic for the software designer as they are for the software modeler and the test designer; as a consequence, such points have more than their share of bugs. The common problems are: lost daughters, wrongful deaths, and illegitimate births.

TRANSACTION FLOW TESTING TECHNIQUES:

- ***GET THE TRANSACTIONS 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 AND WALKTHROUGHS:***
  - Transaction flows are natural agenda for system reviews or inspections.
  - In conducting the walkthroughs, you should:
    - Discuss enough transaction types to account for 98%-99% of the transaction the system is expected to process.
    - 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.
  - Make transaction flow testing the corner stone of system functional testing just as path testing is the corner stone of unit testing.
  - Select additional flow paths for loops, extreme values, and domain boundaries.
  - Design more test cases to validate all births and deaths.
  - 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.
- ***PATH 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.
- ***PATH 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.

BASICS OF DATA FLOW TESTING:

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

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

- ### *DATA FLOW MACHINES:*
  - There are two types of data flow machines with different architectures. (1) Von Neumann machnes (2) Multi-instruction, multi-data machines (MIMD).
  - ### *Von Neumann Machine Architecture:*
    - Most computers today are von-neumann machines.
    - This architecture features interchangeable storage of instructions and data in the same memory units.
    - The Von Neumann machine Architecture executes one instruction at a time in the following, micro instruction sequence:
      1. Fetch instruction from memory
      2. Interpret instruction
      3. Fetch operands
      4. Process or Execute
      5. Store result
      6. Increment program counter
      7. GOTO 1
  - ### *Multi-instruction, Multi-data machines (MIMD) Architecture:*
    - These machines can fetch several instructions and objects in parallel.
    - They can also do arithmetic and logical operations simultaneously on different data objects.
    - The decision of how to sequence them depends on the compiler.

- ### *BUG ASSUMPTION:*
  - The bug assumption for data-flow testing strategies is that control flow is generally correct and that something has gone wrong with the software so that data objects are not available when they should be, or silly things are being done to data objects.
  - Also, if there is a control-flow problem, we expect it to have symptoms that can be detected by data-flow analysis.
  - Although we'll be doing data-flow testing, we won't be using data flowgraphs as such. Rather, we'll use an ordinary control flowgraph annotated to show what happens to the data objects of interest at the moment.

- ### *DATA FLOW GRAPHS:*
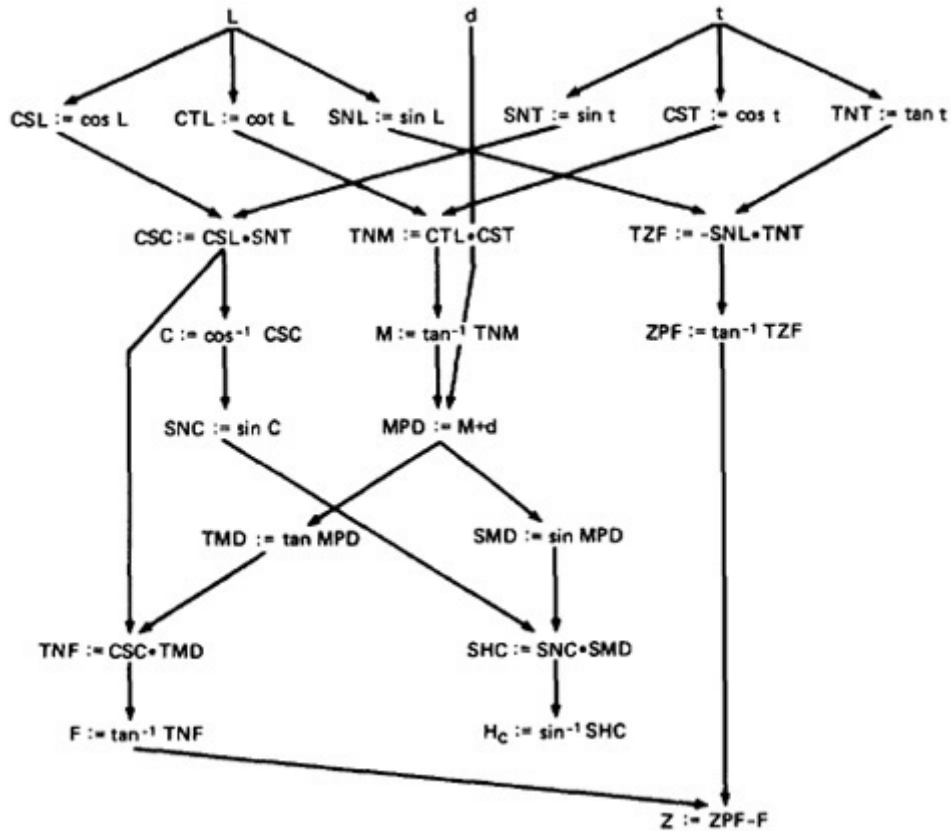 - The data flow graph is a graph consisting of nodes and directed links.

Figure 3.4: Example of a data flow graph

○ We will use an control graph to show what happens to data objects of interest at that moment.
○ Our objective is to expose deviations between the data flows we have and the data flows we want.
○ ***Data Object State and Usage:***
  ▪ Data Objects can be created, killed and used.
  ▪ They can be used in two distinct ways: (1) In a Calculation (2) As a part of a Control Flow Predicate.
  ▪ The following symbols denote these possibilities:
    1. **Defined:** d - defined, created, initialized etc
    2. **Killed or undefined:** k - killed, undefined, released etc
    3. **Usage:** u - used for something (c - used in Calculations, p - used in a predicate)
  ▪ ***1. Defined (d):***
    ▪ An object is defined explicitly when it appears in a data declaration.
    ▪ Or implicitly when it appears on the left hand side of the assignment.
    ▪ It is also to be used to mean that a file has been opened.
    ▪ A dynamically allocated object has been allocated.
    ▪ Something is pushed on to the stack.
    ▪ A record written.

2. Killed or Undefined (k):
    ▪ An object is killed on undefined when it is released or otherwise made unavailable.

When its contents are no longer known with certitude (with aboslute certainity / perfectness).

- Release of dynamically allocated objects back to the availability pool.
- Return of records.
- The old top of the stack after it is popped.
- An assignment statement can kill and redefine immediately. For example, if A had been previously defined and we do a new assignment such as A : = 17, we have killed A's previous value and redefined A

3. *Usage (u):*

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

DATA FLOW ANOMALIES:

An anomaly is denoted by a two-character sequence of actions.

For example, ku means that the object is killed and then used, where as dd means that the object is defined twice without an intervening usage.

What is an anomaly is depend on the application.

There are nine possible two-letter combinations for d, k and u. some are bugs, some are suspicious, and some are okay.

1.	**dd** :- probably harmless but suspicious. Why define the object twice without an intervening usage?
2.	**dk** :- probably a bug. Why define the object without using it?
3.	**du** :- the normal case. The object is defined and then used.
4.	**kd** :- normal situation. An object is killed and then redefined.
5.	**kk** :- harmless but probably buggy. Did you want to be sure it was really killed?
6.	**ku** :- a bug. the object doesnot exist.
7.	**ud** :- usually not a bug because the language permits reassignment at almost any time.
8.	**uk** :- normal situation.
9.	**uu** :- normal situation.

In addition to the two letter situations, there are six single letter situations.

We will use a leading dash to mean that nothing of interest (d,k,u) occurs prior to the action noted along the entry-exit path of interest.

A trailing dash to mean that nothing happens after the point of interest to the exit.

They possible anomalies are:

1.	**-k** :- possibly anomalous because from the entrance to this point on the path, the variable had not been defined. We are killing a variable that does not exist.
2.	**-d** :- okay. This is just the first definition along this path.
3.	**-u** :- possibly anomalous. Not anomalous if the variable is global and has been previously defined.
4.	**k-** :- not anomalous. The last thing done on this path was to kill the variable.
5.	**d-** :- possibly anomalous. The variable was defined and not used on this path. But this could be a global definition.
6.	**u-** :- not anomalous. The variable was used but not killed on this path. Although this sequence is not anomalous, it signals a frequent kind of bug. If d and k mean dynamic storage allocation and return respectively, this could be an instance in which a dynamically allocated object was not returned to the pool after use.
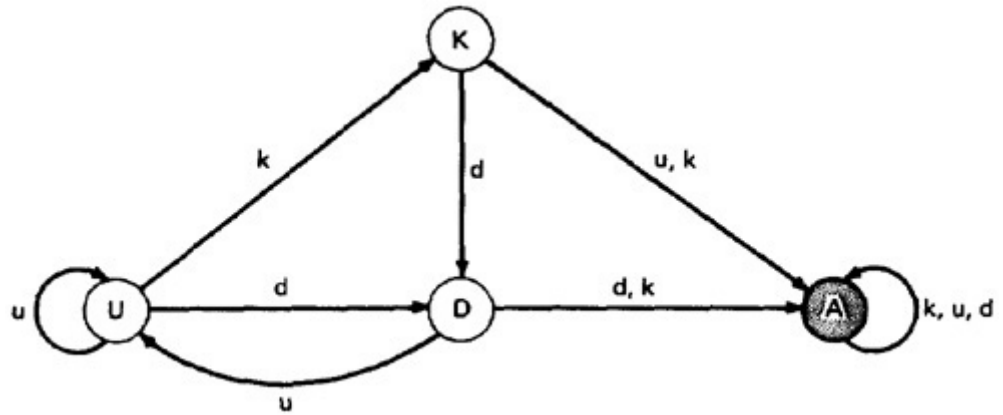
DATA FLOW ANOMALY STATE GRAPH:

Data flow anomaly model prescribes that an object can be in one of four distinct states:
1.	**K** :- undefined, previously killed, doesnot exist
2.	**D** :- defined but not yet used for anything
3.	**U** :- has been used for computation or in predicate
4.	**A** :- anomalous

These capital letters (K,D,U,A) denote the state of the variable and should not be confused with the program action, denoted by lower case letters.
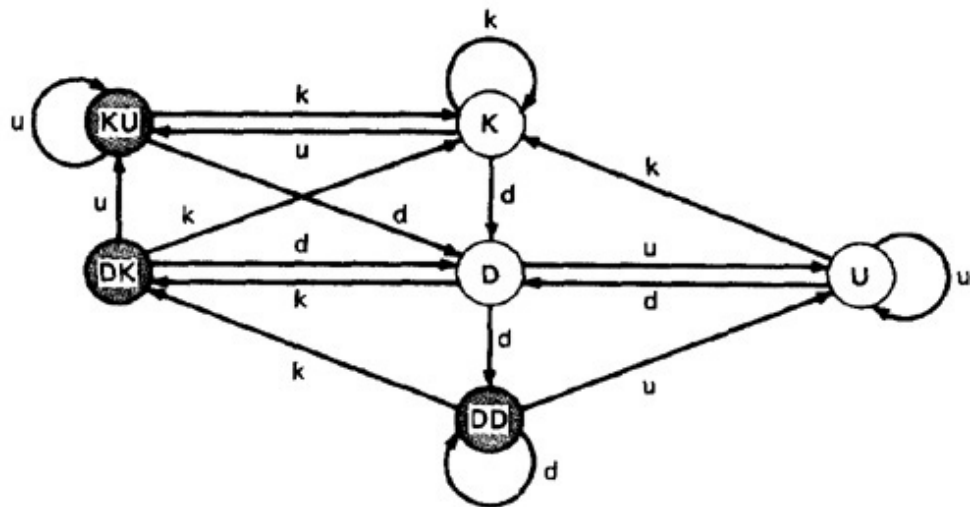
Unforgiving Data - Flow Anomaly Flow Graph: Unforgiving model, in which once a variable becomes anomalous it can never return to a state of grace.

Figure 3.5: Unforgiving Data Flow Anomaly State Graph

Assume that the variable starts in the K state - that is, it has not been defined or does not exist. If an attempt is made to use it or to kill it (e.g., say that we're talking about opening, closing, and using files and that 'killing' means closing), the object's state becomes anomalous (state A) and, once it is anomalous, no action can return the variable to a working state. If it is defined (d), it goes into the D, or defined but not yet used, state. If it has been defined (D) and redefined (d) or killed without use (k), it becomes anomalous, while usage (u) brings it to the U state. If in U, redefinition (d) brings it to D, u keeps it in U, and k kills it.

Forgiving Data - Flow Anomaly Flow Graph: Forgiving model is an alternate model where redemption (recover) from the anomalous state is possible.

Figure 3.6: Forgiving Data Flow Anomaly State Graph

This graph has three normal and three anomalous states and he considers the kk sequence not to be anomalous. The difference between this state graph and Figure 3.5 is that redemption is possible. A proper action from any of the three anomalous states returns the variable to a useful working state.

The point of showing you this alternative anomaly state graph is to demonstrate that the specifics of an anomaly depends on such things as language, application, context, or even your frame of mind. In principle, you must create a new definition of data flow anomaly (e.g., a new state graph) in each situation. You must at least verify that the anomaly definition behind the theory or imbedded in a data flow anomaly test tool is appropriate to your situation.

STATIC Vs DYNAMIC ANOMALY DETECTION:

Static analysis is analysis done on source code without actually executing it. For example: source code syntax error detection is the static analysis result.

Dynamic analysis is done on the fly as the program is being executed and is based on intermediate values that result from the program's execution. For example: a division by zero warning is the dynamic result.

If a problem, such as a data flow anomaly, can be detected by static analysis methods, then it doesnot belongs in testing - it belongs in the language processor.

There is actually a lot more static analysis for data flow analysis for data flow anomalies going on in current language processors.

For example, language processors which force variable declarations can detect (-u) and (ku) anomalies.

But still there are many things for which current notions of static analysis are INADEQUATE.

Why Static Analysis isn't enough? There are many things for which current notions of static analysis are inadequate. They are:

- **Dead Variables:**Although it is often possible to prove that a variable is dead or alive at a given point in the program, the general problem is unsolvable.
- **Arrays:**Arrays are problematic in that the array is defined or killed as a single object, but reference is to specific locations within the array. Array pointers are usually dynamically calculated, so there's no way to do a static analysis to validate the pointer value. In many languages, dynamically allocated arrays contain garbage unless explicitly initialized and therefore, -u anomalies are possible.
- **Records and Pointers:**The array problem and the difficulty with pointers is a special case of multipart data structures. We have the same problem with records and the pointers to them. Also, in many applications we create files and their names dynamically and there's no way to determine, without execution, whether such objects are in the proper state on a given path or, for that matter, whether they exist at all.
- **Dynamic Subroutine and Function Names in a Call:**subroutine or function name is a dynamic variable in a call. What is passed, or a combination of subroutine names and data objects, is constructed on a specific path. There's no way, without executing the path, to determine whether the call is correct or not.
- **False Anomalies:**Anomalies are specific to paths. Even a "clear bug" such as ku may not be a bug if the path along which the anomaly exist is unachievable. Such "anomalies" are false anomalies. Unfortunately, the problem of determining whether a path is or is not achievable is unsolvable.

- **Recoverable Anomalies and Alternate State Graphs:**What constitutes an anomaly depends on context, application, and semantics. How does the compiler know which model I have in mind? It can't because the definition of "anomaly" is not fundamental. The language processor must have a built-in anomaly definition with which you may or may not (with good reason) agree.
- **Concurrency, Interrupts, System Issues:**As soon as we get away from the simple single-task uniprocessor environment and start thinking in terms of systems, most anomaly issues become vastly more complicated. How often do we define or create data objects at an interrupt level so that they can be processed by a lower-priority routine? Interrupts can make the "correct" anomalous and the "anomalous" correct. True concurrency (as in an MIMD machine) and pseudoconcurrency (as in multiprocessing) systems can do the same to us. Much of integration and system testing is aimed at detecting data-flow anomalies that cannot be detected in the context of a single routine.

Although static analysis methods have limits, they are worth using and a continuing trend in language processor design has been better static analysis methods, especially for data flow anomaly detection. That's good because it means there's less for us to do as testers and we have far too much to do as it is.

DATA FLOW MODEL:

The data flow model is based on the program's control flow graph - Don't confuse that with the program's data flowgraph..

Here we annotate each link with symbols (for example, d, k, u, c, p) or sequences of symbols (for example, dd, du, ddd) that denote the sequence of data operations on that link with respect to the variable of interest. Such annotations are called link weights.

The control flow graph structure is same for every variable: it is the weights that change.   *Components of the model:*

1. To every statement there is a node, whose name is unique. Every node has at least one outlink and at least one inlink except for exit nodes and entry nodes.
2. Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statements (e.g., END, RETURN), to complete the graph. Similarly, entry nodes are dummy nodes placed at entry statements (e.g., BEGIN) for the same reason.
3. The outlink of simple statements (statements with only one outlink) are weighted by the proper sequence of data-flow actions for that statement. Note that the sequence can consist of more than one letter. For example, the assignment statement A:= A + B in most languages is weighted by cd or possibly ckd for variable A. Languages that permit multiple simultaneous assignments and/or compound statements can have anomalies within the statement. The sequence must correspond to the order in which the object code will be executed for that variable.
4. Predicate nodes (e.g., IF-THEN-ELSE, DO WHILE, CASE) are weighted with the p - use(s) on every outlink, appropriate to that outlink.
5. Every sequence of simple statements (e.g., a sequence of nodes with one inlink and one outlink) can be replaced by a pair of nodes that has, as weights on the link between them, the concatenation of link weights.
6. If there are several data-flow actions on a given link for a given variable, then the weight of the link is denoted by the sequence of actions on that link for that variable.
7. Conversely, a link with several data-flow actions on it can be replaced by a succession of equivalent links, each of which has at most one data-flow action for any variable.

Let us consider the example:

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

* A contrived horror

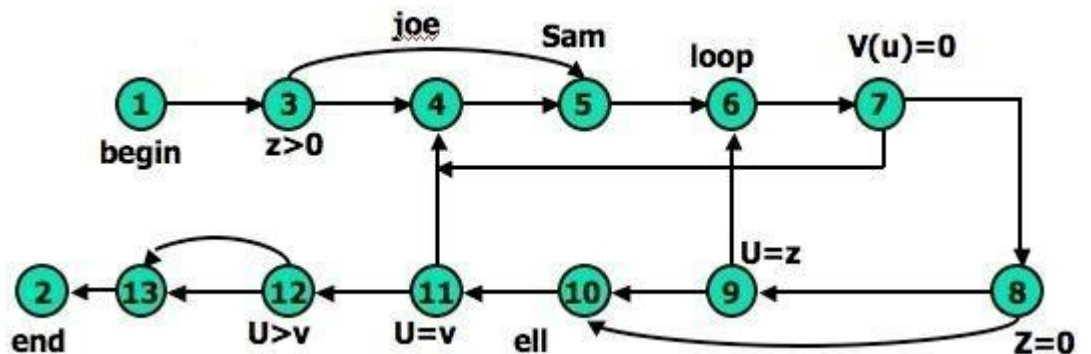**Figure 3.7: Program Example (PDL)**

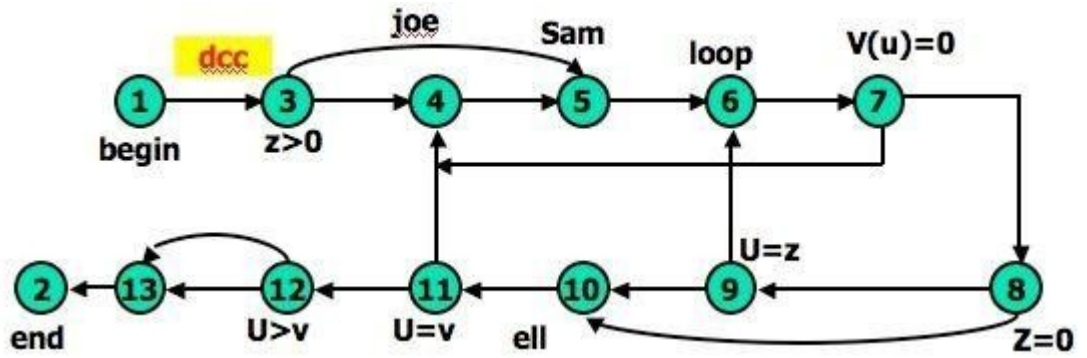**Figure 3.8: Unannotated flowgraph for example program in Figure 3.7**

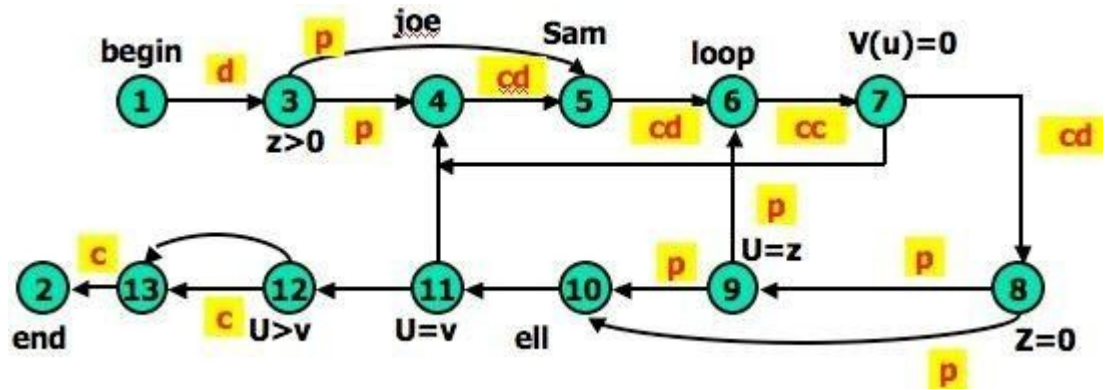*Figure 3.9: Control flowgraph annotated for X and Y data flows.*

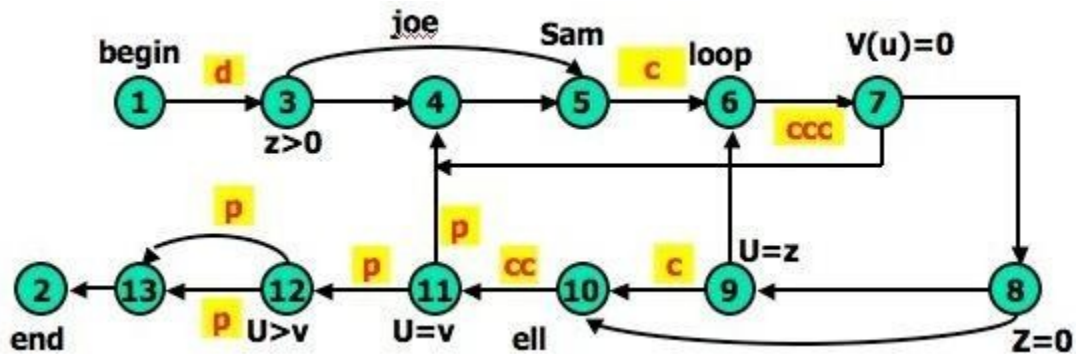*Figure 3.10: Control flowgraph annotated for Z data flow.*

Figure 3.11: Control flowgraph annotated for V data flow.

STRATEGIES OF DATA FLOW TESTING:

- ***INTRODUCTION:***
  - Data Flow Testing Strategies are structural strategies.
  - In contrast to the path-testing strategies, data-flow strategies take into account what happens to data objects on the links in addition to the raw connectivity of the graph.
  - In other words, data flow strategies require data-flow link weights (d,k,u,c,p).
  - Data Flow Testing Strategies are based on selecting test path segments (also called **sub paths**) that satisfy some characteristic of data flows for all data objects.
  - For example, all subpaths that contain a d (or u, k, du, dk).
  - A strategy X is **stronger** than another strategy Y if all test cases produced under Y are included in those produced under X - conversely for **weaker**.
- ***TERMINOLOGY:***

1. **Definition-Clear Path Segment**, with respect to variable X, is a connected sequence of links such that X is (possibly) defined on the first link and not redefined or killed on any subsequent link of that path segment. ll paths in Figure 3.9 are definition clear because variables X and Y are defined only on the first link (1,3) and not thereafter. In Figure 3.10, we have a more complicated situation. The following path segments are definition-clear: (1,3,4), (1,3,5), (5,6,7,4), (7,8,9,6,7), (7,8,9,10), (7,8,10), (7,8,10,11). Subpath (1,3,4,5) is not definition-clear because the variable is defined on (1,3) and again on (4,5). For practice, try finding all the definition-clear subpaths for this routine (i.e., for all variables).

2. **Loop-Free Path Segment** is a path segment for which every node in it is visited atmost once. For Example, path (4,5,6,7,8,10) in Figure 3.10 is loop free, but path (10,11,4,5,6,7,8,10,11,12) is not because nodes 10 and 11 are each visited twice.

3. **Simple path segment** is a path segment in which at most one node is visited twice. For example, in Figure 3.10, (7,4,5,6,7) is a simple path segment. A simple path segment is either loop-free or if there is a loop, only one node is involved.

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

STRATEGIES: The structural test strategies discussed below are based on the program's control flowgraph. They differ in the extent to which predicate uses and/or computational uses of variables are included in the test set. Various types of data flow testing strategies in decreasing order of their effectiveness are:

1. **All - du Paths (ADUP):** The all-du-paths (ADUP) strategy is the strongest data-flow testing strategy discussed here. It requires that every du path from every definition of every variable to every use of that definition be exercised under some test.

For variable X and Y: In Figure 3.9, because variables X and Y are used only on link (1,3), any test that starts at the entry satisfies this criterion (for variables X and Y, but not for all variables as required by the strategy).

For variable Z: The situation for variable Z (Figure 3.10) is more complicated because the variable is redefined in many places. For the definition on link (1,3) we must exercise paths that include subpaths (1,3,4) and (1,3,5). The definition on link (4,5) is covered by any path that includes (5,6), such as subpath (1,3,4,5,6, ...). The (5,6) definition requires paths that include subpaths (5,6,7,4) and (5,6,7,8).

For variable V: Variable V (Figure 3.11) is defined only once on link (1,3). Because V has a predicate use at node 12 and the subsequent path to the end must be forced for both directions at node 12, the all-du-paths strategy for this variable requires that we exercise all loop-free entry/exit paths and at least one path that includes the loop caused by (11,4). Note that we must test paths that include both subpaths (3,4,5) and (3,5) even though neither of these has V definitions. They must be included because they provide alternate du paths to the V use on link (5,6). Although (7,4) is not used in the test set for variable V, it will be included in the test set that covers the predicate uses of array variable V() and U.

The all-du-paths strategy is a strong criterion, but it does not take as many tests as it might seem at first because any one test simultaneously satisfies the criterion for several definitions and uses of several different variables.

2.   **All Uses Startegy (AU):** The all uses strategy is that at least one definition clear path from every definition of every variable to every use of that definition be exercised under some test. Just as we reduced our ambitions by stepping down from all paths (P) to branch coverage (C2), say, we can reduce the number of test cases by asking that the test set should include at least one path segment from every definition to every use that can be reached by that definition.

For variable V: In Figure 3.11, ADUP requires that we include subpaths (3,4,5) and (3,5) in some test because subsequent uses of V, such as on link (5,6), can be reached by either alternative. In AU either (3,4,5) or (3,5) can be used to start paths, but we don't have to use both. Similarly, we can skip the (8,10) link if we've included the (8,9,10) subpath. Note the hole. We must include (8,9,10) in some test cases because that's the only way to reach the c use at link (9,10) - but suppose our bug for variable V is on link (8,10) after all? Find a covering set of paths under AU for Figure 3.11.

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

For variable Z: In Figure 3.10, for APU+C we can select paths that all take the upper link (12,13) and therefore we do not cover the c-use of Z: but that's okay according to the strategy's definition because every definition is covered. Links (1,3), (4,5), (5,6), and (7,8) must be included because they contain definitions for variable Z. Links (3,4), (3,5), (8,9), (8,10), (9,6), and (9,10) must be included because they contain predicate uses of Z. Find a covering set of test cases under APU+C for all variables in this example - it only takes two tests.

For variable V: In Figure 3.11, APU+C is achieved for V by (1,3,5,6,7,8,10,11,4,5,6,7,8,10,11,12[upper], 13,2) and (1,3,5,6,7,8,10,11,12[lower], 13,2). Note that the c-use at (9,10) need not be included under the APU+C criterion.

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

For variable Z: In Figure 3.10, ACU+P coverage is achieved for Z by path (1,3,4,5,6,7,8,10, 11,12,13[lower], 2), but the predicate uses of several definitions are not covered. Specifically, the (1,3) definition is not covered for the (3,5) p-use, the (7,8) definition is not covered for the (8,9), (9,6) and (9, 10) p-uses.

> *The above examples imply that APU+C is stronger than branch coverage but ACU+P may be weaker than, or incomparable to, branch coverage.*

5.   **All Definitions Strategy (AD) :** The all definitions strategy asks only every definition of every variable be covered by atleast one use of that variable, be that use a computational use or a predicate use.

For variable Z: Path (1,3,4,5,6,7,8, . . .) satisfies this criterion for variable Z, whereas any entry/exit path satisfies it for variable V.
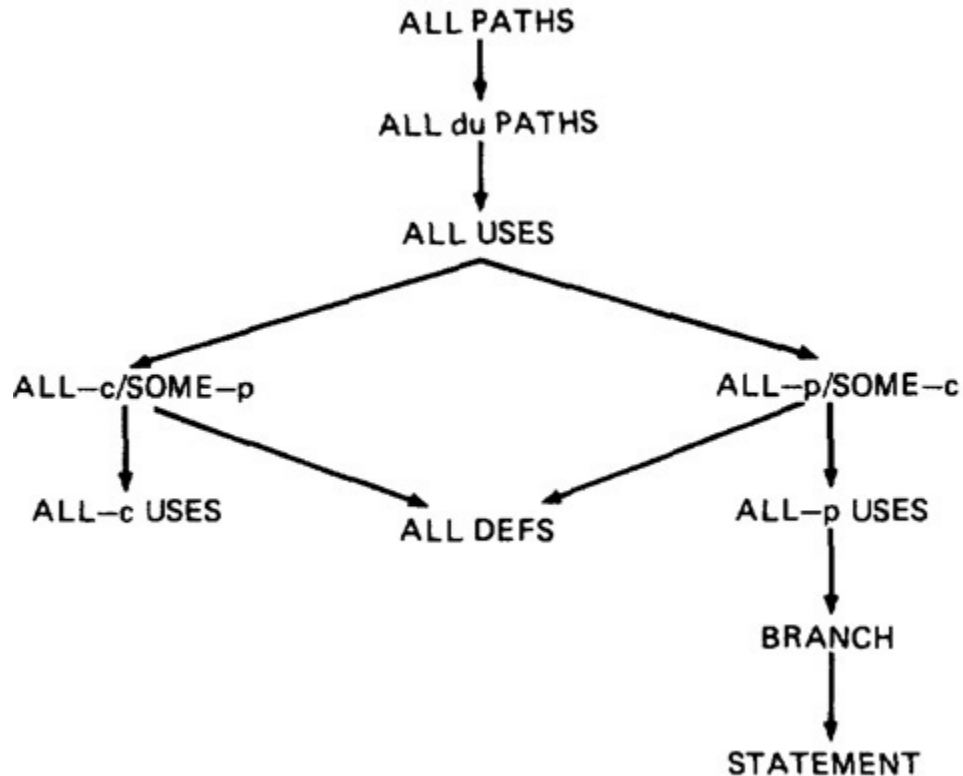
> *From the definition of this strategy we would expect it to be weaker than both ACU+P and APU+C.*

6.   **All Predicate Uses (APU), All Computational Uses (ACU) Strategies :** The all predicate uses strategy is derived from APU+C strategy by dropping the requirement that we include a c-use for the variable if there are no p-uses for the variable. The all computational uses strategy is derived from ACU+P strategy by dropping the requirement that we include a p-use for the variable if there are no c-uses for the variable.

> *It is intuitively obvious that ACU should be weaker than ACU+P and that APU should be weaker than APU+C.*

ORDERING THE STRATEGIES:

○   Figure 3.12 compares path-flow and data-flow testing strategies. The arrows denote that the strategy at the arrow's tail is stronger than the strategy at the arrow's head.

ALL PATHS

↓

ALL du PATHS

↓

ALL USES

ALL−c/SOME−p   ALL−p/SOME−c

ALL−c USES   ALL DEFS   ALL−p USES

BRANCH

↓

STATEMENT

Figure 3.12: Relative Strength of Structural Test Strategies.

○ The right-hand side of this graph, along the path from "all paths" to "all statements" is the more interesting hierarchy for practical applications.
○ Note that although ACU+P is stronger than ACU, both are incomparable to the predicate-biased strategies. Note also that "all definitions" is not comparable to ACU or APU.

SLICING AND DICING:

○ A (static) program **slice** is a part of a program (e.g., a selected set of statements) defined with respect to a given variable X (where X is a simple variable or a data vector) and a statement i: it is the set of all statements that could (potentially, under static analysis) affect the value of X at statement i - where the influence of a faulty statement could result from an improper computational use or predicate use of some other variables at prior statements.
○ If X is incorrect at statement i, it follows that the bug must be in the program slice for X with respect to i
○ A program **dice** is a part of a slice in which all statements which are known to be correct have been removed.
○ In other words, a dice is obtained from a slice by incorporating information obtained through testing or experiment (e.g., debugging).
○ The debugger first limits her scope to those prior statements that could have caused the faulty value at statement i (the slice) and then eliminates from further consideration those statements that testing has shown to be correct.
○ Debugging can be modeled as an iterative procedure in which slices are further refined by dicing, where the dicing information is obtained from ad hoc tests aimed primarily at eliminating possibilities. Debugging ends when the dice has been reduced to the one faulty statement.

- ○ **Dynamic slicing** is a refinement of static slicing in which only statements on achievable paths to the statement in question are included.