20

Path Testing:

→ **Definition:**

↳ The Program graph is a directed graph in which nodes are statement fragments & edges represent flow of control.

↳ If i and j corresponds to nodes in the Program graph, an edge exists from node i to node j if the statement fragment corresponding to node j can be executed immediately after the statement fragment corresponding to node i.

```
1. Program triangle2 'Structured Programming Version
   of Simpler Specification
2. Dim a, b, c As Integer
3. Dim ISATriangle As Boolean
   'Step 1 : Get Input
4. Output ("enter 3 integers which are sides of a triangle")
5. Input (a, b, c)
6. Output ("Side A is", a)
7. Output ("Side B is", b)
8. Output ("Side C is", c)
   'Step 2 : Is a Triangle ?
9. If (a < b + c) AND (b < a + c) AND (c < a + b)
10.    Then IsATriangle = True
11.    Else ISATriangle = False
12. EndIf
   'Step 3 : Determine Tle type
13. If ISATriangle
14.    Then If (a = b) AND (b = c)
15.            Then Output ("equilateral")
16.            Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
17.                 Then Output ("Scalene")
```

18. Else If Output ("Isosceles")
19. EndIf
20. EndIf
21. Else Output ("Not a Δle")
22. EndIf
23. End Δle

↳ A program graph of this Program is given be

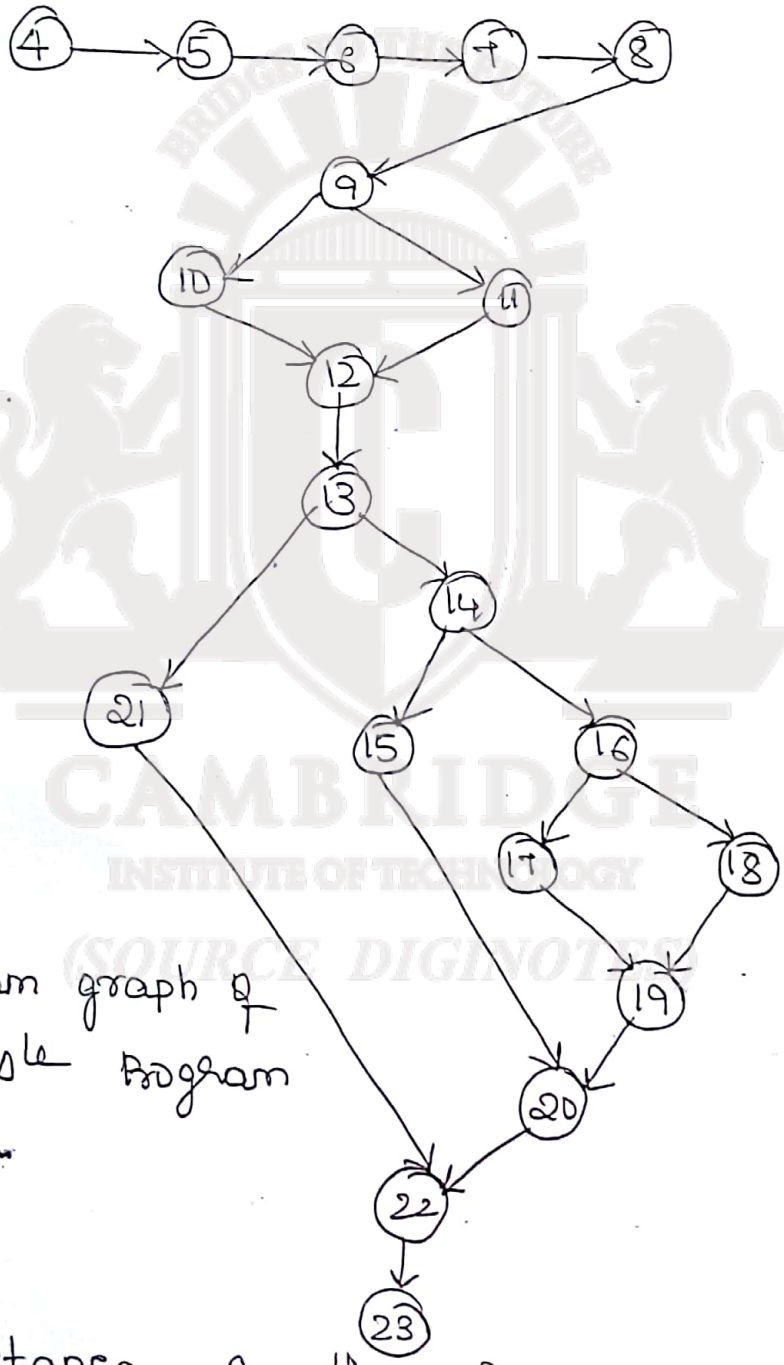


Fig:- Program graph of the Δle Program

↳ The importance of the Program graph is that Program executions correspond to paths from the

Source to Sink nodes. Because testcases force the execution of some such Program Paths, We now have a very explicit description of the relation -ship between a testcase and the part of the Program it exercises.

→ DD-Paths :- [Decision to Decision Path]

Definition:- A DD-path is a sequence of nodes in a Program graph such that:

case 1: It consists of single node with indeg=0.

case 2: " " " " outdeg=0

case 3: " " " indeg ≥ 2 OR outdeg ≥ 2

case 4: " " " indeg=1 and outdeg=1

case 5: ~~////////~~ It is a Maximal chain of length ≥ 1.

↳ Case 1 & 2 establishes the unique Source & Sink nodes of the Program graph of a Structured Program as Initial and final DD paths.

↳ case 3 deals with Complex nodes; it ensures that no node is contained in more than one DD path.

↳ Case 4 is needed for short branches.

↳ case 5 is the "normal" case in which a DD path is a Single entry, Single exit sequence of nodes.

→ Applying the above definitions to the ← Prev Program graph, Node 4 is a case 1 DD path, it is called first; and Node 23 is a case 2 DD path, it is called last.

↳ Nodes 5 through 8 are case 5 DD paths. Node 8 is the last node in this DD path, if we go beyond node 8 to include node 9, we violate the Indegree = Outdegree = 1 criteria of a chain. If we stop at 7, we violate the "Maximal" criteria.

↳ Nodes 10, 11, 15, 17, 18 & 21 are case 4 DD paths

↳ Nodes 9, 12, 13, 14, 16, 19, 20 & 22 are case 3 " "

↳ finally node 23 is a case 2 DD path.

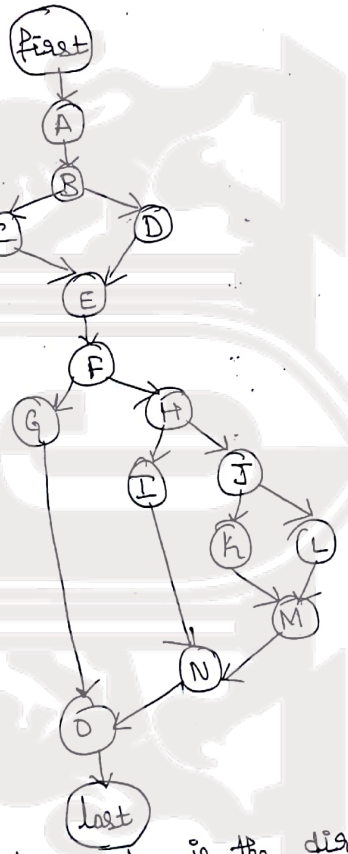
⇒ The foll table summarizes the paths, where the DD path names correspond to node names in the DD path graph in foll figure.

Table: Types of DD paths of fig (Prev)

Program Graph Nodes	DD-path Names	case Definition.
4	first	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3

21	G	4
22	D	3
23	last	2

fig :- DD-path graph for the Δ^le Program



Definition :- DD-path graph is the directed graph. in which nodes are DD-paths of its Program graph. & edges represent control flow blw successor DD-paths

↳ High quality commercial tools are available, which generate the DD-path graph of a given Program.

→ The following figure shows a graph of a simple program. It shows the impossibility of completely testing even simple program.

In this Program, 5 paths lead from node B to node F in the interior of the loop. If the loop may have up to 18 repetitions, some 4.77 trillion distinct Program execution path exists.

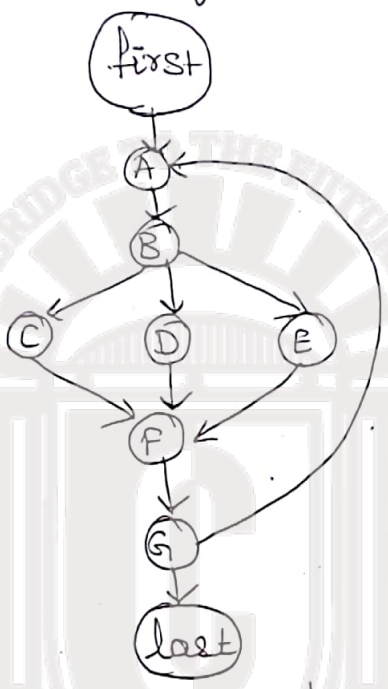


Fig :- Trillions of Paths.

→ DD paths are defined in terms of path & nodes in a directed graph. These might be called path of chains, where a chain is a path in which initial & terminal nodes are distinct and every interior node has indegree = 1 & outdegree = 1.

∴ The length (num of edges) of the chain in the following figure is 6.

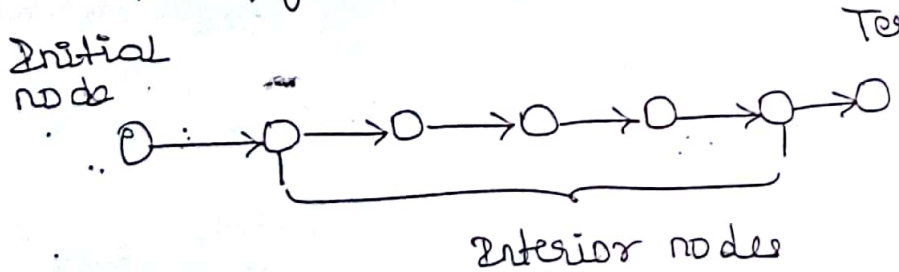


Fig :- A chain of nodes in a directed graph.

→ Test Coverage Metrics:

Test Coverage metrics are a device to measure the extent to which a set of testcases covers a Program.

↳ Several widely accepted test coverage metrics are defined in most of them are due to the early work of E.F. Miller.

↳ Most organizations now expect the C1 Metric (DD Path coverage) as the minimum acceptable level of test coverage.

↳ E.F. Miller observes that when DD path coverage is attained by a set of testcases, roughly 85% of all faults are revealed.

Table: Structural Test Coverage Metrics

Metric	Description of coverage
C0	Every statement
C1	every DD path (Predicate outcome)
CP	every Predicate to each outcome
C2	C1 coverage + loop coverage
Cd	C1 coverage + every dependent pair of DD-Paths
CMCC	Multiple condition coverage
CIK	Every Program Path the contain upto k repetitions of a loop
Cstat	Statistically significant fraction of paths
CP	All possible execution Paths.

Metric - Based Testing:

1. Statement and Predicate Testing:

↳ Our formulation allows statement fragments to be individual nodes. The statement & Predicate levels (C0 and C1) collapse into one consideration.

↳ In our Δle Problem, nodes 9, 10, 11 & 12 are a complete if-then-else statement.

* If we required nodes to correspond to full statements we would execute just one of the decision alternatives & satisfy the statement coverage criteria

* Because we allow statement fragments, it is natural to divide such a statement into three node.
∴ Doing so results in Predicate outcome Coverage.

2. DD path Testing:

↳ When every DD path is traversed (the C1 Metric), each predicate outcome has been executed; this amounts to traversing every edge in the DD path graph as opposed to only every node.

↳ For If-then & if-then-else statements, both true & false branches are covered (C1P coverage). for CASE Statements each clause is covered.

3. Dependent Pairs of DD-paths:

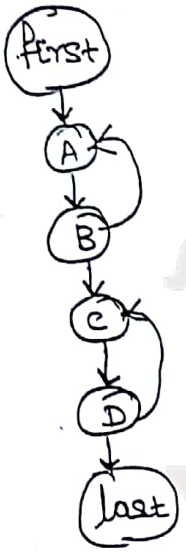
↳ The metric The most common dependency among pairs of DD paths is the define/reference relationship in which a variable is defined in one DD path and is referenced in another DD path.

↳ They are closely related to the Problem of infeasible Paths.

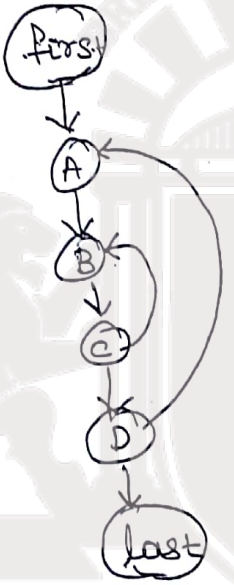
4. Multiple Condition Coverage:

↳ Instead of simply traversing compound conditions in DD paths, we should investigate diff ways that each outcome can occur.
One possibility is to make a truth table: a compound condition of 3 simple conditions would have 8 rows yielding 8 testcases.

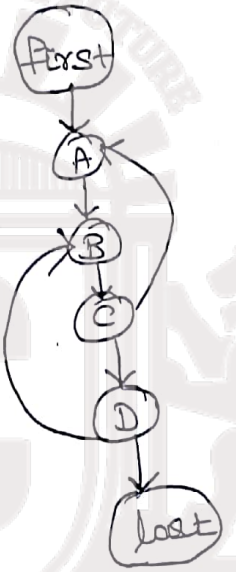
5. loop coverage:



Concatenated loops



Nested loops



Knotted loops.

↳ Concatenated loops are simply a sequence of disjoint loops. While nested loops are one is contained inside another. Knotted loops occur when it is possible to branch into the middle of a loop & the branches are internal to other loops.

↳ The simple view of loop testing is that every loop involves a decision & we need to test both outcomes of the decision: one is to traverse the loop and other is to exit the loop.

↳ If the body of a simple loop is a DD path that performs a complex calculation, this should also be tested. Once a loop has been tested, the tester condenses it into a single node.

↳ If loops are nested, this process is repeated starting with the innermost loop and working outward.

↳ If loops are knotted it will be necessary to carefully analyze them in terms of the dataflow methods.

→ Test Coverage Analyzers:

↳ Coverage Analyzers are a class of test tools that offer automated support for testing management. With a coverage Analyzer, the tester runs a set of testcases on a program that has been "instrumented" by the coverage analyzer.

↳ The analyzer then uses info produced by the instrumentation code to generate a coverage report.

Ex:- The instrumentation identifies & labels all DD paths in an original program. When the instrumented program is executed with testcases, the analyzer tabulates the DD paths traversed by each testcase.

→ Basis Path Testing:

↳ Mathematicians usually define a basis in terms of a structure called a vector space, which is a set of elements as well as operations that correspond to multiplication & addition defined for the vectors.

↳ All Vector Space have a basis.

↳ The basis of Vector Space is a set Vectors that are independent of each other and span the entire Vector Space in the sense that any other Vector in the Space can be expressed in terms of the basis Vectors.

↳ If one basis element is ~~lost~~ deleted; this spanning property is lost.

If a Program is viewed as a vector Space, then the basis for such a Space would be a very intresting set of elements to test. If the basis is OK, We could hope that everything that can be expressed in terms of the basis is also okay.

→ Mc Cabe's Basis Path Method.

↳ Mathematician define a basis in terms of a structure called a Vector Space, which is a set of elements as well as operations that correspond to Multiplicati and addition defined for Vectors.

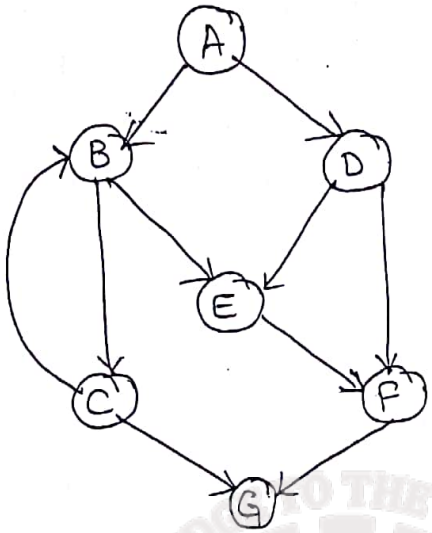
↳ The basis of a Vector Space is a set of Vectors that are independent of each other & span the entire Vector Space, in the sense that any other Vector in the Space can be expressed in terms of basis Vectors.

↳ The potential Value of this theory for testing is that if We can View a Program as a Vector Space, then the basis for such a Space would be Very intresting set of elements to test.

↳ If the basis is OK, We could hope that everything that can be expressed in terms of basis is also OK.

→ McCabe's Basis Path Method-

Fig:- McCabe's Control g



→ The above figure is a directed graph. The program does have a single entry (A) and a single exit (G).

→ McCabe based his view of testing on a major result from graph theory, which states that the cyclomatic number of a strongly connected graph is the number of linearly independent circuits in the graph.

→ A strongly connected graph can be created by adding an edge from source to sink node.

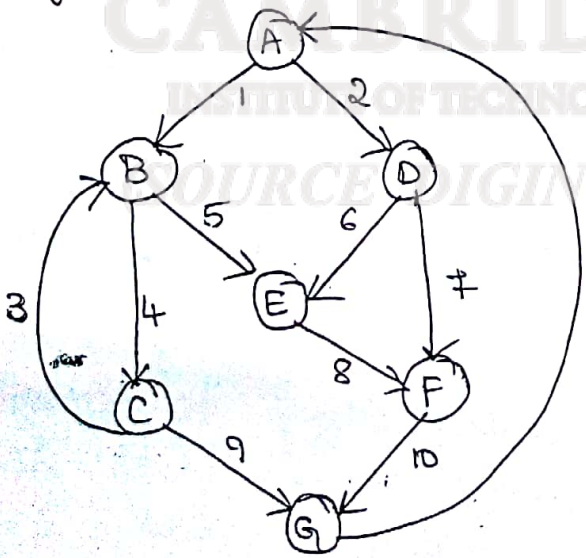


fig *

Fig:- McCabe's derived strongly connected graph

→ There exists a confusion in the literature regarding the correct formula for cyclomatic complexity

$V(G) = e - n + P$ $\quad$ $V(G) = e - n + 2P$

e - number of edges n - number of nodes P - number of connected regions

The confusion is due to the transformation from arbitrary directed graph to strongly connected graph.

$V(G) = e - n + 2P = 10 - 7 + 2(1) = 5$

$V(G) = e - n + P = 11 - 7 + 1 = 5$

Thus there are 5 linearly independent circuits.

The 5 linearly independent paths are:

P1 : A, B, C, G

P2 : A, B, C, B, C, G

P3 : A, B, E, F, G

P4 : A, D, E, F, G

P5 : A, D, F, G

→ This can be made to look like vector space by defining notions of additions & scalar Multiplication.

Addition – one path followed by another path

Multiplication – repetition of a path.

→ According to McCabes illustration of the basis part.

basis sum – P2 + P3 - P1

linear combination – 2P2 - P1

This can be seen from the incidence matrix in the following table.

Table: Path / edge Traversal

Path / edges Traversed	1	2	3	4	5	6	7	8	9	10
P1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
P2: A, B, C, B, C, G	1	0	(1)	2	0	0	0	0	1	0
P3: A, B, E, F, G	1	0	0	0	(1)	0	0	1	0	1
P4: A, D, E, F, G	0	1	0	0	0	(1)	0	1	0	1
P5: A, D, F, G	0	1	0	0	0	0	(1)	0	0	1
ex1: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
ex2: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

→ The independence of paths P1 - P5 by examining the first 5 rows of this incidence matrix. The circled entries show edges that appear in exactly one path so paths P2 - P5 must be independent.

These 5 paths span the set of all paths from node A to node G.

→ McCabes develops an algorithmic procedure to determi a set of basis paths.
The method begins by choosing a path with as many decision nodes as possible. Next the baseline pat is retraced, and in turn each decision is "flipped" that is when a node of outdegree ≥ 2 is reached a different edge must be taken.

→ According to Mc Cabe's example, he postulates the Path through nodes A, B, C, B, E, F, G as the baseline. The first decision node (outdegree ≥ 2) in this path is node A; So for the next basis path, we traverse edge 2 instead of edge 1. We get the path A, D, E, F, G. Where we retrace nodes E, F, G in Path 1 to be as Minimally different as possible.

For, the next path, we can follow the second path and take the other decision outcome of node D, which gives us the path A, D, F, G. Now only decision nodes B & C have not been flipped; doing so yeilds the last 2 basis paths, A, B, E, F, G and A, B, C, G.

→ Observations on Mccabe's Basis Path method.

→ Mccabe view has 2 dot dots:

* one is that testing the set of basis paths is sufficient
* Program paths look like a vector space.

↳ linear combination $2P_2 - P_1$?

P_2 – executing path P_2 twice?

$-P_1$ – executing Path P_1 backward / undo recent execution of P_1?

→ To get a better understanding, we take a problem.

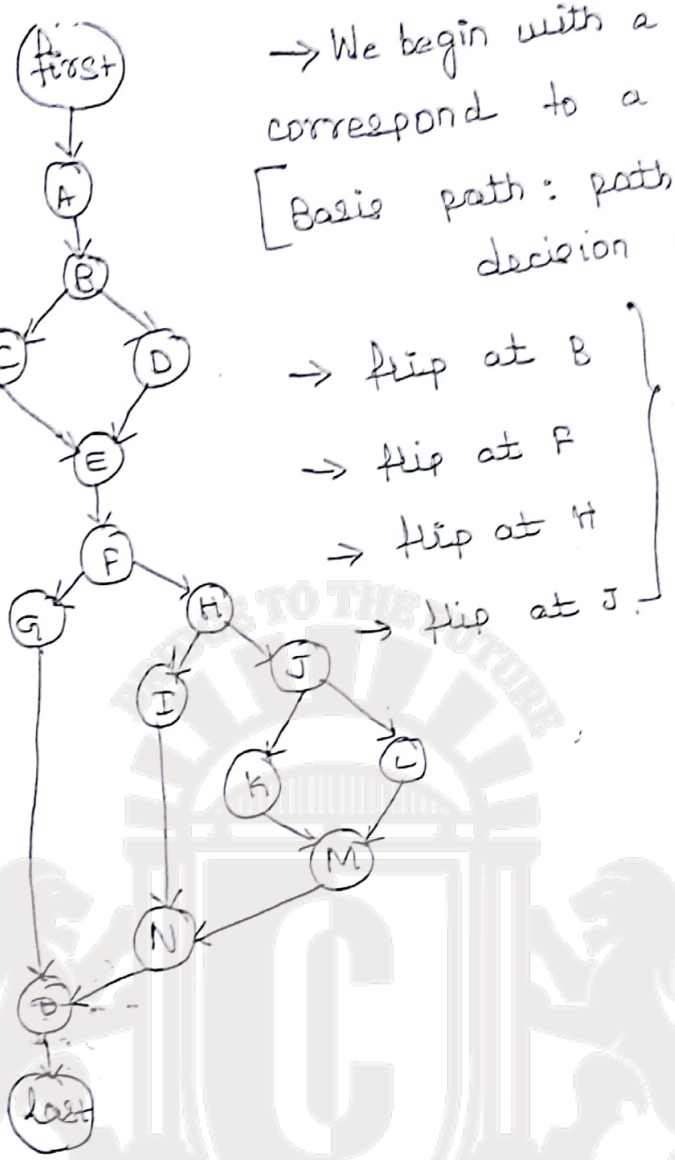


Fig:- DD path graph of Dle Program.

→ We begin with a baseline path, correspond to a decisions dle.

[Basis path : path with highest ↑ decision nodes]

→ flip at B
→ flip at F
→ flip at H
→ flip at J.

} flip at nodes with outdegree = 2.

Table 9.4 Basis paths in figure ↑

Original	P1 : A-B-C-E-F-H-J-K-M-N-O-last	decisions
flip P1 at B	P2: ABDEFHJKMNO last	infeasible
flip P1 at F	P3: ABCEFGDlast	"
" " H	P4 : ABCEFHINO last	equilateral
" J	P5 : ABCEFHJLMNO last	isoceles

→ If we follow Path P2 & P3 they are both infeasible. Path P2 is infeasible, because passing through node D means the sides are not a Dle ; so the outcome of the decision at node F must be node G.

→ Similary in P8 - passing through node C, means the sides do form a Δle ; So node G cant be traversed Path P4 & P5 are both feasible & correspond to equilateral & isoceles Δle.

→ Here, we are dealing with code-level dependencies. Which are absolutely incompatible with the latent assumption that basis paths are independent. McCabe's procedure successfully identifies basis paths that are topologi - cally independent, but when these contradict semantic dependencies, topologically possible paths are seen to be logically infeasible.

→ One solution to this problem, is to always require that flipping a decision results in a semantic - ally feasible path.
Another is to reason about logical dependencies.

→ 2 rules can be identified
* If node E is traversed, then we must traverse node H.
* If node D is traversed, then we must traverse node G.

→ following are feasible basis path set.
P1 : A - B - C - E - F - H - J - K - M - N - O - last Scalene
P2 : A - B - D - E - F - G - O - last Not a Δle
P4 : A - B - C - E - F - H - I - N - O last equilateral
P5 : A - B - C - E - F - H - J - L - M - N - O - last Isoscales.

→ The Δle program has 8 topologically possible paths of ~~these~~ listed above only the 4 basis paths are feasible.

→ Basis Path Coverage guarantees DD path coverage the Process of flipping decisions guarantees every decision Outcome is traversed, which is same as DD-Path coverage.

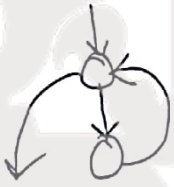
→ Essential Complexity:

→ This topic concentrates on the elegant blend of graph theory, Structured Programming, & the implications these have for testing.

→ Condensation graph - Are a way of simplifying an existing graph. Here, We condense around the Structured Programming constructs which are repeated as foll figure.

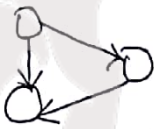
Sequence

Pretest loop

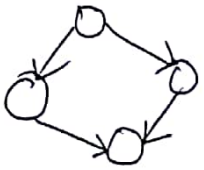


Posttest loop

ifthen



ifthenelse



case

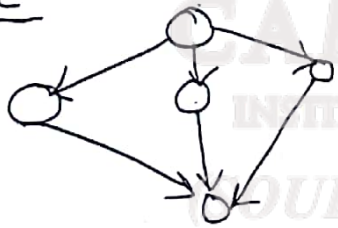


fig :- Structured Programming constructs.

→ The basic idea is to look for the graph of one of the Structured Programming constructs, collapse it into a single node, & repeat until no more Structured Programming constructs can be found.

→ This Process is followed in the foll figures, which starts with the DD path graph of the Pseudocode Δ^le Problem.

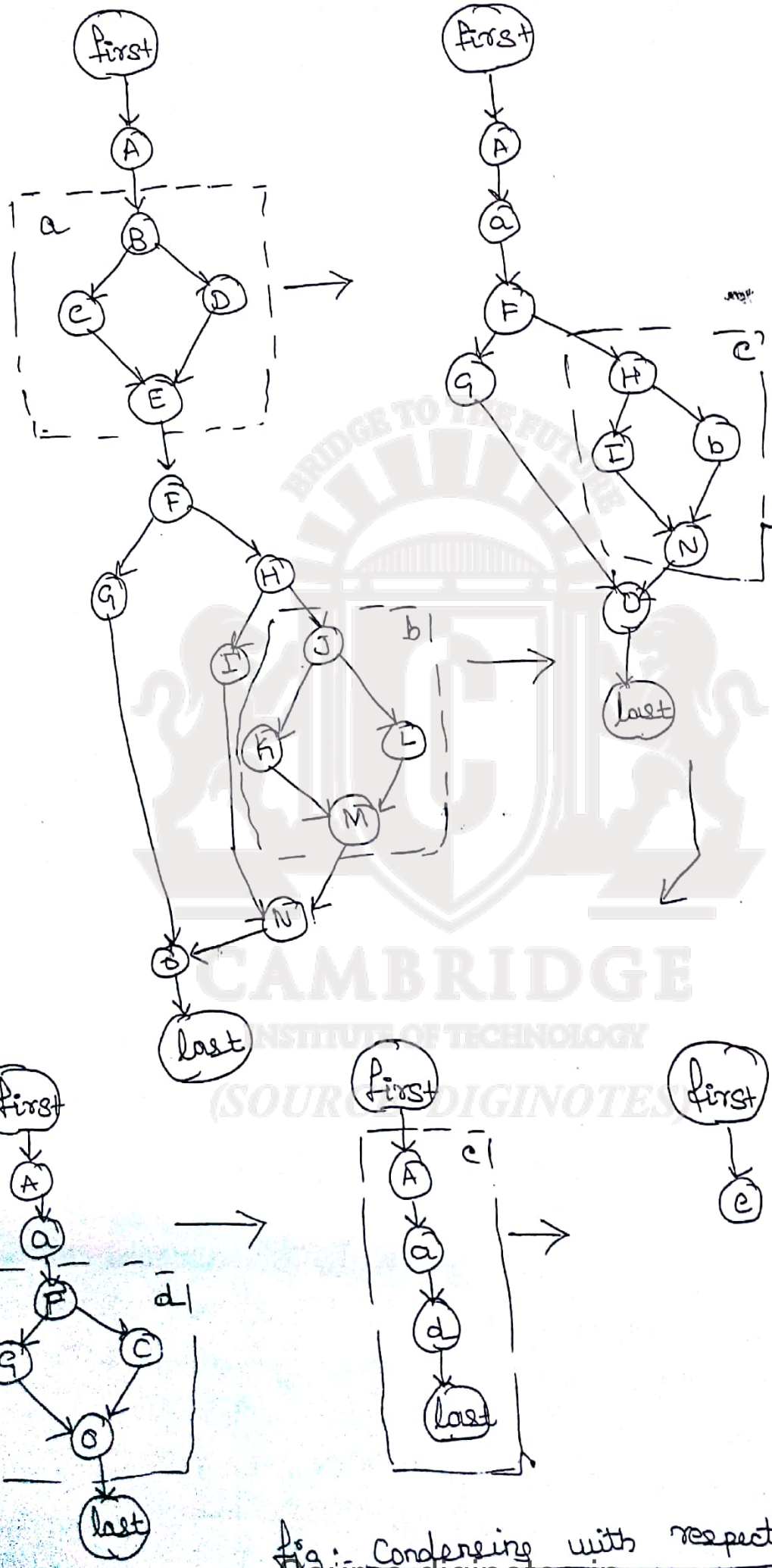


Fig: Condensing with respect to the structured Programming

→ The if-then-else construct involving nodes B, C, D, is condensed into node a, & then the 3 if-then constructs are condensed onto nodes b, c & d. The remaining if-then-else is condensed int node e resulting in a condensed graph with cyclomatic complexity $V(G) = 1$.

→ The graph in (figure *) can't be reduced this way.

* The loop with nodes B & C can't be condensed because of the edge from B to E. Similarly, nodes D, E & F look like an if-then construct, but the edge from B to E violates the structure.

→ Each of these unstructures contains 3 distinct paths as opposed to the 2 paths present in the corresponding structured programming constructs; So one conclusion is that such violations increase cyclomatic complexity.

→ Programs with high cyclomatic complexity require more testing.

→ Guidelines and Observations:

1) McCabes was partly right when he observed. "It is important to understand that these are purely criteria that measure the quality of testing and not a procedure to identify testcases.

DD path coverage Metric ≅ (Predicate outcome metric)
cyclomatic complexity metric ≅ (requires atleast the cyclomatic number of distint program paths be traversed).

* Basis path testing therefore gives us a lower boundary on how much testing is necessary.

→ Path-based testing also provides us with a set of metrics that act as cross-checks on functional testing.
These metrics can be used to resolve the gaps & redundancies. When a DD path coverage is failed to attain it implies there are gaps in the functional test cases.

→ The best view of structural testing is that use the properties of the source code to identify appropriate coverage metrics, & then use these to cross check on functional testcases.

→ The following Venn diagram shows the relationship among specified behaviours (set S), Programmed behaviours (set P) and topologically feasible paths in a program (set T).

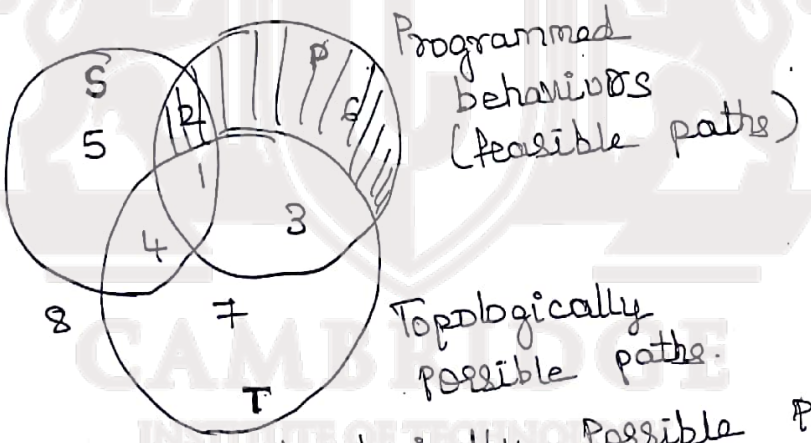


fig:- feasible & topologically possible paths.

→ Region 1 is the most desirable, it contains specified behaviors that are implemented by feasible paths.

→ By defn, all feasible paths are topologically possible, so the shaded portions 2, 6 must be empty

→ Region 3 contains feasible paths that correspond to unspecified behaviors. [such extra functionality needs to be examined].

→ Regions 4 and 7 contains infeasible paths.
Region 4 refers to specified behaviors that have almost been implemented – topologically possible yet infeasible paths. This region very likely corresponds to coding errors

→ Region 5 corresponds to specified behaviors that have not been implemented.

→ Region 7 – unspecified, infeasible, yet topologically possible paths.

---

chapter - 3

DataFlow Testing:

→ Refers to forms of testing that focus on the points at which variables receive values and the points at which these values are used.

→ **Define / use Testing:**
The following definitions refer to Program P that has a Program graph G(P) and a set of Program variable V. The Program graph G(P) is constructed with statement fragment as nodes and edges that correspond to node sequence.
G(P) has a single entry node and a single exit node.

1) Node $n \in G(P)$ is a defining node of the variable $V \in V$, written as DEF(V, n), iff the value of the variable V is defined at the statement fragment corresponding to node n.

[i/p, assignment, loop control, Procedure calls are all examples of statements that are defining nodes].

→ Node $n \in G(P)$ is a <u>usage node</u> of the variable $\in V$, written as USE (v, n) iff the value of the variable v is used at the statement fragment corresponding to node n.

[o/p statements, assignment, conditional, loop control & Procedure calls]

→ A usage node USE (v, n) is a <u>Predicate use</u> (denoted as P-use) iff the statement n is a Predicate statement; otherwise USE (v, n) is a <u>computation use</u> (C-use).

[Predicates - outdegree ≥ 2, computation - outdegree ≤ 1]

→ A <u>definition use path</u> with respect to a variable v (denoted du Path) is a path in PATHS (P) such that for some $v \in V$ these are define & usage nodes DEF (v, m) and USE (v, n) such that m & n are initial & final nodes of the Path.

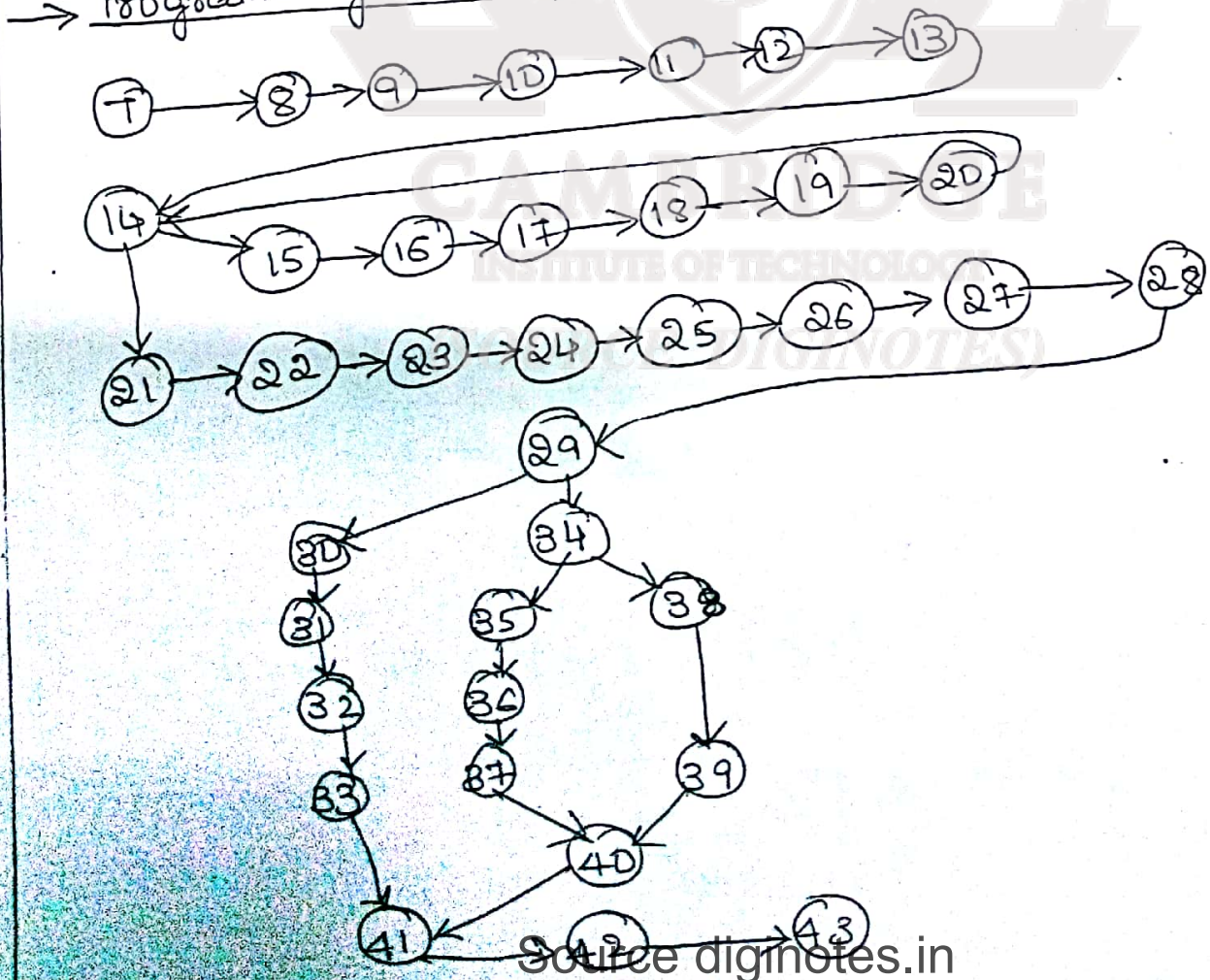
→ A <u>definition - clear Path</u> with respect to a variable v (denothe dc Path) is a definition - use Path in PATHS (P) with initial & final nodes DEF (v, m) & USE (v, n) such that no other node in the Path is a defining node of v.

→ Example [Commission Problem]

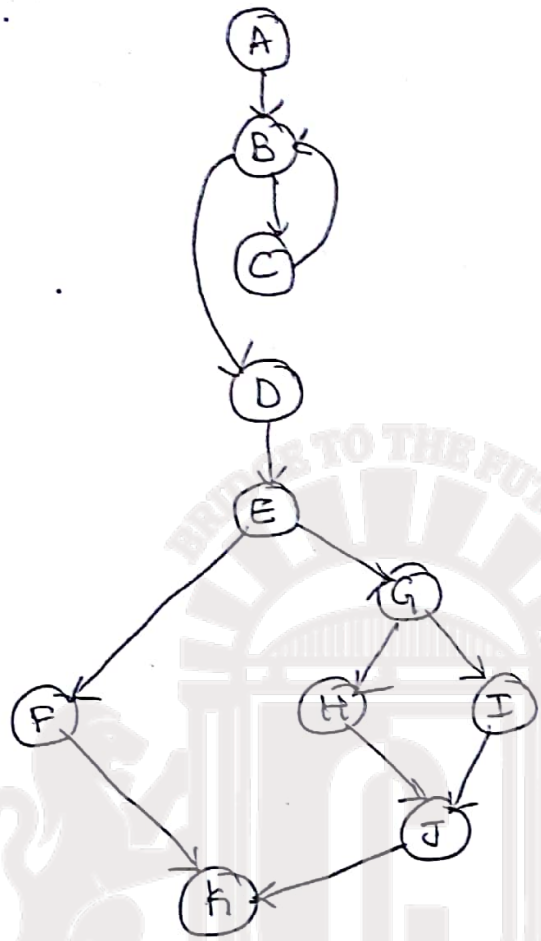
```
1.  Program Commission (Input, output)
2.  Dim locks, stocks, barrels As Integer
3.  Dim lockPrice, stockPrice, barrelPrice As Real
4.  Dim locksales, stocksales, barrelsales As Real
5.  Dim totalLocks, totalStocks, total barrel As Integer
6.  Dim Sales, Commission As Real
7.  lockPrice = 45.0
8.  StockPrice = 30.0
9.  BarrelPrice = 25.0
10. totalLocks = 0
11.   "   Stocks = 0
12.   "   Barrels = 0
13. Input (locks)
14. While NOT (locks = -1)
    Indicate end of data. → Input (stocks, Barrels)
15
16. totalLocks = totalLocks + locks
17. totalStocks = totalStocks + stocks
18. totalBarrels = totalBarrels + barrels.
19. Input (locks)
20. EndWhile
21. Output ("locks Sold", totalLocks)
22. Output ("Stocks
23. Output (" Barrels
24. locksales = lockPrice * totalLocks.
25. Stocksales = StockPrice * totalStocks
26. Barrelsales = BarrelPrice * totalBarrels
27. Sales = locksales + StockSales + Barrelsales
28. Output (" Totalsales : ", Sales)
29. If Sales > 1800.0)
```

30. Then
31. commission = 0.10 * 1000.0
32. commission = commission + 0.15 * 800.0
33 " " " + 0.20 * (Sales - 1800.0)
34. Else If (Sales > 1000.0)
35. Then
36. commission = 0.10 * 1000.0
37. " = commission + 0.15 * (Sales - 1000.0)
38. else
39. commission = 0.10 * Sales
40. End If
41. End If
42. output (" commission is $", commission)
43. End commission.

→ Program graph of commission Problem :-



Decision to decision Path graph of Program



The foll table details the statement fragments associated with DD Paths

DD path	Nodes		
A	7, 8, 9, 10, 11, 12, 13	I	38, 39
B	14	J	40
C	15, 16, 17, 18, 19, 20	K	41, 42, 43
D	21, 22, 23, 24, 25, 26, 27, 28		
E	29		
F	30, 31, 32, 33		
G	34		
H	35, 36, 37		

The foll table lists the define and usage nodes for the variables in the commission problem.

Variable	Defined at Node	Used at Node
→ lockPrice	7	24
→ stockPrice	8	25
→ BarrelPrice	9	26
→ totalLocks	10, 16	16, 21, 24
→ totalStocks	11, 17	17, 22, 25
→ totalBarrels	12, 18	18, 23, 26
→ locks	13, 19	14, 16
→ stocks	15	17
→ Barrels	15	18
→ lockSales	24	27
→ stockSales	25	27
→ BarrelSales	26	27
→ Sales	27	28, 29, 33, 34, 37, 39
→ commission	31, 32, 33, 36, 37, 39	32, 33, 37, 42

→ The foll table 10.3 ↓ Presents some of the du-paths in the commission Problem; they are named by their beginning and ending nodes. The third colum indicates whether the du-paths are definition clear

→ The table only shows the details for the totalStocks Variable. The initial value definition for totalStocks occurs at node 11, and its first used at node 17. Thus the Path (11, 17). which consists of node sequen <11, 12, 13, 14, 15, 16, 17> is definition clear.

→ The path (11, 22) consisting of node sequence <11, 12, 13 (14, 15, 16, 17, 18, 19, 20)*, 21, 22> is not definition-clea because values of totalStocks are defined at node 11 & node 17.

Table: Selected Define / use Paths:

Variable	Path (Beginning, end nodes)	Definition-use (Yes/No)
lockPrice	(7, 24)	Yes
StockPrice	(8, 25)	Yes
barrelPrice	(9, 26)	Yes
totalStocks	(11, 17)	Yes
"	11, 22	No
"	11, 25	No
"	17, 17	Yes
"	17, 22	No
"	17, 25	No
locks	13, 14	Yes
locks	13, 16	Yes
locks	19, 14	Yes
locks	19, 16	Yes
Sales	27, 28	Yes
"	27, 29	Yes
"	27, 33	Yes
"	27, 34	Yes
"	27, 37	Yes
"	27, 39	Yes.

1) du- Paths for Stocks:

→ for Stocks.
DEF (Stocks, 15) and USE (Stocks, 17)
Path <15, 17> is a du-path with respect to Stocks.
No other defining nodes are used for Stocks. therefore this Path is def clear.

2) du- Paths for locks:

→ 2 defining & 2 usage nodes.
DEF (locks, 13) DEF (locks, 19) USE (locks, 14)
USE (locks, 16)

4 du- Paths: P_1 = <13, 14>
P_2 = <13, 14, 15, 16>
P_3 = <19, 20, 14>
P_4 = <19, 20, 14, 15, 16>

↳ Du-paths P_1 & P_2 refer to the Priming value of locks, which is read at node 13: locks has a Predicate use in the while statement & if the cond? is true, a computation use at statement 16. The other 2 du- Paths start near the end of the while loop & occur when the loop repeats.

3) du- Paths for totalLocks:

→ 2 defining nodes DEF (totalLocks, 10) & DEF (totalLocks, 16)
3 usage nodes USE (totalLocks, 16), USE (totalLocks, 21),
USE (totalLocks, 24)

Path P_5 = <10, 11, 12, 13, 14, 15, 16> is a du path in which the Initial value of totalLocks (10) has a computation use.
This path is Definition clear.

Path P_6 = <10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 14, 21>
P_6 ignores the possible repetition of the while loop.

the subpath <16, 17, 18, 19, 20, 14, 15> Might be tra... several times. This du-path is not definition cle...

Path P_7 = <10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 14, 21, 22, 23, 24>

$P_{7.}$ = <P6, 22, 23, 24>

Dupath P_7 is also not definition-clear because it includes node 16.

P_8 = <16, 17, 18, 19, 20, 14, 21>

P_9 = <16, 17, 18, 19, 20, 14, 21, 22, 23, 24>

Both are definition clear.

4) du-Paths for sales.

→ Only one defining node is used; therefore all the du-paths with respect to sales must be definition clear. They also illustrate Predicate & computation uses.

P_{10} = <27, 28>

P_{11} = <27, 28, 29>

P_{12} = <27, 28, 29, 30, 31, 32, 33>

P_{13} = <27, 28, 29, 34>

P_{14} = <27, 28, 29, 34, 35, 36, 37>

P_{15} = <27, 28, 29, 34, 38, 39>

5) du-paths for commission:

→ In Statements 29. through 41, the calculation of Commission is controlled by ranges of the variable sales.

→ Statements 31 to 33 built up the value of commission by using the memory location to hold intermediate values.

The "built up" version uses intermediate values, & these will appear as define & usage nodes in the du-path analysis.

Hence disallow du-paths from assignment statements like 31 & 32 and consider the paths that begin with a "real" defining nodes:

DEF (commission, 33), DEF (commission, 37) and DEF (" , 38).

Only one usage node: USE (commission, 42).

→ du-path Test Coverage Metrics:

→ Analyzing a Program with definition-use paths defines a set of test coverage metrics known as the Rapps-Weyuker dataflow metrics. The first 3 of these are equivalent to 3 of EF. Miller's metrics, All paths, edges and nodes. The others presume that define & usage nodes have been identified for all Program variable and that du-paths have been identified with respect to each variable.

→ In the following definitions, T is a set of paths in the Program graph G(P) of a Program P, with the set V of variables.

1) Definition: The set T satisfies the All-Defs criterion for the Program P iff for every variable $v \in V$, T contains definition-clear paths from every defining node of v to use of v.

2) Definition:

This set of T satisfies the All-uses criterion for Program P iff for every variable $v \in V$, T contains definition-clear paths from every defining node of v to every use of v, and to the successor node of each USE (v, n).

3) Definition:

The set T satisfies the All-P-uses/some C-uses criterion for the program P iff for every variable $v \in V$, T contains definition-clear paths from every defining node of v to every predicate use of v. If a definition of v has no P-uses, a definition-clear path leads to atleast one computation use.

4) Definition:

The set T satisfies the All C-uses/some P-uses criterion for the Program P iff for every variable $v \in V$, T contains definition-clear paths from every defining node of v to every computation use of v: If a definition of v has no C-uses, a definition clear path leads to atleast one predicate use.

5) Definition:

The set T satisfies the All-du-paths criterion for the Program P iff for every variable $v \in V$, T contains definition-clear paths from every defining node of v to every use of v & to the successor node of each USE (v, n) & that these paths are either single loop traversals or cycle-free.

→ These test coverage metrics have several set theory based relationships reported to as "subsumption" in Rapps and Weyuker (1985). The relationships are shown below.

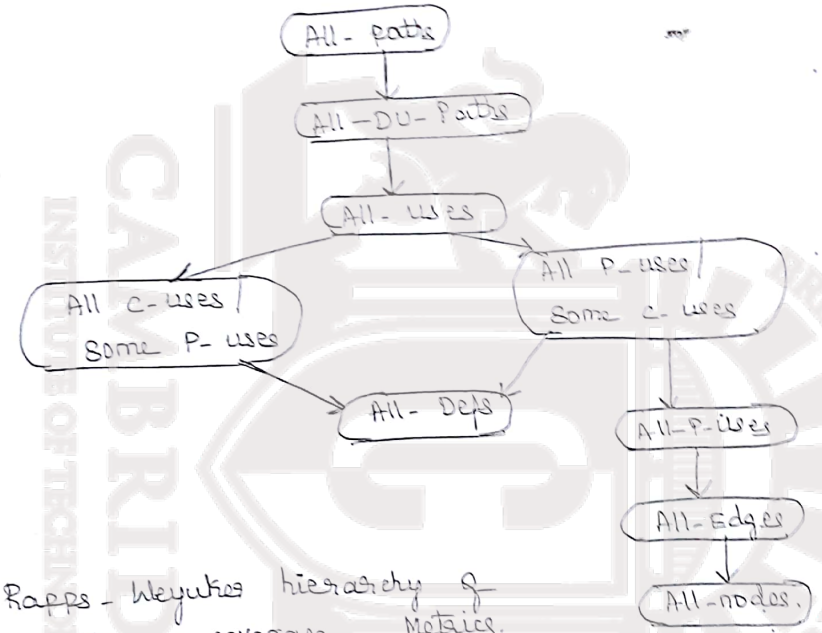


Fig:- Rapps - Weyuker hierarchy of dataflow coverage Metrics.

→ Slice - Based Testing:

1) Definition: Given a Program P and a set of Variables V in P, a slice on the Variable set V at statement n, written S(V, n) is the set of all statements in P prior to node n that contribute to the values of Variables in V at node n.

2) Definition: Given a Program P and a Program graph G(P) in which statements and statement fragments are numbered and set V of Variables in P, the slice on the Variable set V at statement fragment n, written S(V, n), is the set of node.

numbers of all statement fragments in ? prior and including n that contribute to the values variables in v at statement fragment n.

→ The idea of slice is to separate a Program into components that have some useful meanings.

2 parts of definition

"Prior to" " contribute"

Prior to:- a slice captures the execution time behaviour of a program with respect to the variable (s) in the slice. Eventually, we will develop a lattice of slices, in which nodes are slice and edges correspond to the subset relationship.

Contribute:- data declaration statements have an effect on the value of a variable. The notion of contribution is partially classified by the Predicate (P-use) and computation use (C-use) & distinctions of Rapps and Weyuker (1985). Specifically, the use relationship pertains to 5 forms of usage.

* P-use used in Predicate (decision)
* C-use used in computation
* O-use used for o/p
* L-use used for location (Pointers, subscripts)
* I-use Iteration (internal counters, loop indices)

→ Identify 2 forms of definition nodes:

* I-def Defined by input
* A-def Defined by assignment.

→ Presume that the slice S(v,n) is a slice on one variable; that is the set v consists of a single variable. V.

If statement fragment n is a defining node for v, then n is included in the slice. If statement fragment n is a usage node for v, then n is not included in the slice.

P-uses and C-uses of other variables are included to the extent that their execution affects the value of the variable V.

L-use and I-use variables are typically invisible outside their modules, hence are excluded from slices.

→ Example:

These examples are followed by looking at the source code for the commission problem that we used to analyze in terms of define/use paths.

Slices on the lock variable show why it is potentially fault-prone. It has a P-use at node 14 and a C-use at node 16 and has 2 definitions. the I-defs at nodes 13 and 19.

S1 : S(locks, 13) = {13}

S2 : S(locks, 14) = {13, 14, 19, 20}

S3 : S(locks, 16) = {13, 14, 19, 20}

S4 : S(locks, 19) = {19}

→ Slices for locks are short-definition clear paths contained entirely within a loop, so they are not affected by iterations of the loop.

S5 : S (Stocks, 15) = {13, 14, 15, 19, 20}

S6 : S (Stocks, 17) = {13, 14, 15, 19, 20}

S7 : S (Barrels, 15) = {13, 14, 15, 19, 20}

S8 : S (barrels, 18) = {13, 14, 15, 19, 20}

→ The next three slices illustrate how repetition appears in slice. Node 10 is an A-def for total lock & node 16 contains both an A-def and a C-use. The remaining nodes in S10 (13, 14, 19, 20) pertain to the while loop controlled by locks. Slices S10 & S11 are equal because 21 & 24 nodes are on O-use & C-use of totalLocks, respec.

S9 : S (total locks, 10) = {10}

S10 : S (" , 16) = {10, 13, 14, 16, 19, 20}

S11 : S (" , 21) = {10, 13, 14, 16, 19, 20}.

→ The slices on totalStocks & totalBarrels are initialized by A-defs at nodes 11 & 12 & then are defined by A-defs at node 17 & 18.

S12 : S (totalstocks, 11) = {11}

S13 : S (" , 17) = {11, 13, 14, 15, 17, 19, 20}

S14 : S (" , 22) = {11, 13, 14, 15, 17, 19, 20}

S15 : S (total Barrels, 12) = {12}

S16 : S (" , 18) = {12, 13, 14, 15, 18, 19, 20}

S17 : S (" , 23) = {12, 13, 14, 15, 18, 19, 20}

→ The next slices demonstrate convention regarding values defined by assignment statements (A-defs).

S_{18} : S (lockprice, 24) = {7}

S_{19} : S (stockprice, 25) = {8}

S_{20} : S (barrelprice, 26) = {9}

S_{21} : S (locksales, 24) = {7, 10, 13, 14, 16, 19, 20, 24}

S_{22} : S (stocksales, 25) = {8, 11, 13, 14, 15, 17, 19, 20, 25}

S_{23} : S (barrelSales, 26) = {9, 12, 13, 14, 15, 18, 19, 20, 24}

→ The slices on Sales and Commission, only one defining node exists for Sales, the A-def at node 27. The remaining slices on sales show the P_uses, C-uses and the O-use in definition — clear paths.

S_{24} : S (Sales, 27) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

S_{25} : S (Sales, 28) = { }

S_{26} : S (Sales, 29) = { }

S_{27} : S (Sales, 33) = { }

S_{28} : S (Sales, 34) = { }

S_{29} : S (Sales, 37) = { }

S_{30} : S (Sales, 39) = { }

→ S_{24} can be written as $S_{24} = \{S_{10} \cup S_{13} \cup S_{16} \cup S_{21} \cup S_{22} \cup S_{23} \cup \{27\}$.

→ Everything comes together with slices on commission. Six A-def nodes are used for commission. 3 computation of commission are controlled by P-uses of Sales in the IF, ELSE IF logic.

This yeilds 8 paths of slices that Compute Com

S81 : S (commission, 81) = {81}

S82 : S (commission, 82) = {81, 82}

S83 : S (commission, 83) = {7, 8 ——— 20, 24, 25, 24, 27, 29, 30, 31, 32, 33}

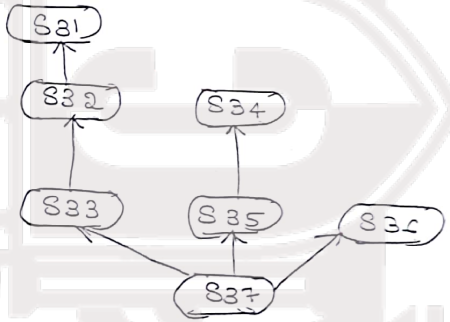
S84 : S (commission, 86) = {86}

S85 : S (commission, 37) = {7 ——— 20, 24 — 27, 36, 37}

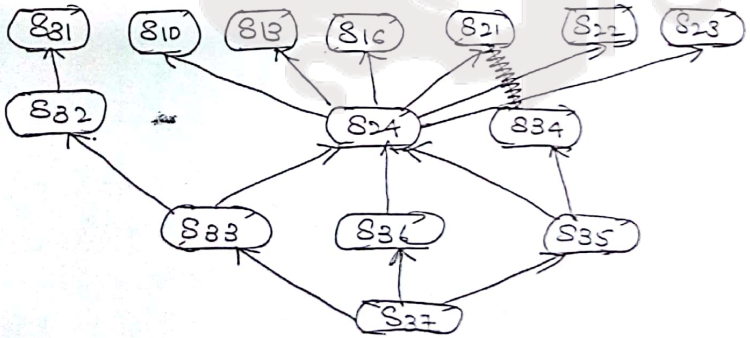
S86 : S (commission, 39) = {7 — 20, 24 — 27, 29, 34, 38, 39}

S87 : S (commission, 41) = {7 — 20, 24, 27, 29, 30 — 39}

lattice of slices on commission:



lattice on sales & commission:



→ Style and Technique:

1. Never make a slice $S(V,n)$ for which Variables $v \notin V$ donot appear in statement fragment n.
2. Make slices on one variable. The set v in slice $S(V,n)$ can contain several variables. The slice $S(V,24)$ where $V = \{$ locksales, stocksales, barrelSales $\}$ contains all the elements of the slice $S(\{$ sales $\},27)$ except statement 27. Those & ~~Slices~~ are
3. Make slices for all A-def nodes.
4. Make slices for P-use nodes.
5. Slices on non-P use usage nodes are not very interesting
6. Consider making slices compatible.

→ Guidelines and Observations:

1. Slices donot map nicely into testcases. On the other hand, they Provide a handy way to eliminate interaction among variables.
2. Relative complements of slices yield a diagnostic capability. Relative complements $(A-B)$.
3. A many to many relationship exists b/w slices and DD paths: statements in one slice may be in several DD paths and statements in one DD path may be in several slices.
4. If we develop a lattice of slices. It is convenient to postulate a slice on the very first statement.
5. Slices exhibit define/reference information. When slices are equal, the corresponding paths are definition clear.

Chapter - 3

Test Execution:

The purpose of run-time support for testing is to enable frequent hands-free execution of a test suite. A large suite of testdata may be generated automatically from a more compact & abstract set of testcase Specifications. for unit & integration testing, and sometimes for system testing as well. the s/w under test may be combined with additional "scaffolding" code to provide a suitable test environment, which might, for example, including simulations of other s/w and h/w resources.

↳ The test environment often includes additional support for selecting testcases and for summarizing & reporting results.

→ From Test case Specifications to Testcases:

↳ If the testcase Specifications Produced in testcase design already include concrete input values & expected results, as for example in the category-Partition method, then Producing a complete testcase may be as simple as filling a template with those values.

→ A more general testcase Specification may designate many Possible concrete testcases and it may be desirable to generate just one instance or many.

There is no clear, sharp line b/w testcase design and testcase generation.

→ Automatic generation of concrete testcases from more abstract testcase Spec' reduces the impact of small interface changes in the course of development: corresponding changes to the testsuite are still required with every Program change but changes

to testcase spec are likely to be smaller and more localized than changes to the concrete testcases.

→ Initiating testcases that satisfy several constraints may be simple if the constraints are independent but becomes more difficult to automate when multiple constraints apply to the same item.

→ General testcase specifications that may require considerable computation to produce test data often arise in model-based testing.

→ fortunately, model-based testing is closely tied to model analysis techniques that can be adapted as test data generation methods.

→ Scaffolding:

↳ Code developed to facilitate testing is called Scaffolding by analogy to the temporary structures erected around a building during construction or maintenance. Scaffolding may include test drivers, test harnesses and stubs in addition to program instrumentation and support for recording & managing test execution.

↳ A common estimate is that half of the code developed in a sw project is scaffolding of some kind, but the amount of scaffolding that must be constructed with sw project can vary widely, and depends both on the appln domain and the architectural design and build plan. Which can reduce cost by exposing appropriate interfaces and providing necessary functionality in a rational order

↳ The purposes of Scaffolding are to Provide Controllability to execute testcases and Observability to judge the outcome of test execution.

↳ Scaffolding is required to simply make a module executable, but even in incremental development with immediate integration of each module, Scaffolding for controllability and observability may be required because the external interfaces of the system may not Provide sufficient control to drive the module under test through testcases. or Sufficient Observability of the effect.

↳ It may be desirable to substitute a seperate test "driver" program for the full system, in order to Provide more direct control of an interface or to remove dependence on other subsystems.

↳ When testability is considered in s/w architectural design, it often happens that interfaces exposed for use in Scaffolding have other uses.

→ Generic Versus Specific Scaffolding:

↳ The simplest form of Scaffolding is a driver Program that runs a single specific test case. If, for example, a testcase specification calls for executing Method calls in a particular sequence, this is easy to accomplish by writing the code to make the method calls in that sequence.

↳ Writing drivers, on the other hand, may be cumbersome and a disincentive to through testing. At the very least one will want to factor out some of the common drives code into reusable modules. Sometimes it is worthwhile to write more generic test drivers that essentially interpret testcase specifications.

↳ generic scaffolding support can be used across a wide class of applications. Such support typically includes, in addition to a standard interface for executing a set of testcases, basic support for logging test execution and results.

↳ The foll figure illustrates use of generic test scaffolding in the JFlex lexical analyzer generator.

```
1. Public final class IntCharSet {
:15 . . .
16  Public Void add (Interval . interval) {
156 . . .
157 }
```

```
1  Package JFlex.tests;
2
3  import JFlex.IntCharSet;
4  import JFlex.Interval;
5  import junit.framework.Testcase;
11 . . .
12 Public class charClassesTest extends Testcase {
25 . . .
26 Public Void testAdd1 () {
27 IntCharSet Set = new IntCharSet (new Interval ('a', 'h'));
28 Set.add (new Interval ('0', '2'));
29 Set.add (new Interval ('A', 'Z'));
30 Set.add (new Interval ('h', '0'));
31 assertEquals ("{['A'-'Z']['a'-'z']}", Set.toString());
32 }
33
34 Public Void testAdd2 () {
35 IntCharSet Set = new IntCharSet (new Interval ('a', 'h'));
36 Set.add (new Interval ('0', '2'));
```

```
37.  Set.add(new Interval('A','Z'));
38.  Set.add(new Interval('i','n'));
39.  assertEquals("{['A'-'Z']['a'-'z']}",set.toString());
40.  }
     99 ...
     100 }
```

↳ fully generic Scaffolding may Suffice for Small numbers of hand-written testcases.
A large Suite of automatically generated testcases and a Smaller set of hand-written testcases can Share the Same underlying generic test Scaffolding.

↳ The Simplest kind of Stub, Sometimes called a mock can be generated automatically by analysis of the Source code. A mock is limited to checking expected invocations and Producing Precomputed results that are part of the testcase Specification or were recorded in a Prior execution.

↳ The balance of quality, Scope and cost for a Substantial piece of Scaffolding Sw. Say a nlw traffic generator for a distributed System or a test harness for a Compiler - is essentially similar to the development of any other Substantial piece of Sw, including Similar considerations regarding Specialization to a Single Project

↳ The balance is altered in favour of Simplicity and quick construction for the many Small piece of Scaffolding that are typically produced during development to Support unit and Small Scale integration-testing.

→ Test Oracles:

↳ S/w that applies a Pass/fail criterion to a Program execution is called a test Oracle, often shortened to Oracle. In addition to rapidly classifying a large number of testcase executions, automated test Oracles make it possible to classify behaviours that exceed human capacity in other ways, such as checking real-time response against latency requirements or dealing with voluminous o/p data in a machine readable rather than human-readable form

↳ One approach to Judging Correctness - but not the only one - Compares the actual o/p or behaviour of a Program with Predicted o/p or behaviour. A testcase with a Comparison-based Oracle relies on Predicted o/p or that is either Precomputed as part of the testcase specification or can be derived in some way independent of the Program under test.

↳ Precomputing expected test results is reasonable for a small number of results because the expense of producing (and debugging) Predicted results is incurred. Once and amortized over many executions of the testcase.

↳ A harness typically takes 2 i/p 1) The i/p to the Program under test and 2) the Predicted o/p. Frameworks for writing testcases as program code likewise provide support for comparison-based Oracles. The assertEquals method of Junit illustrated in the following figure is a simple example of comparison based Oracle-support.

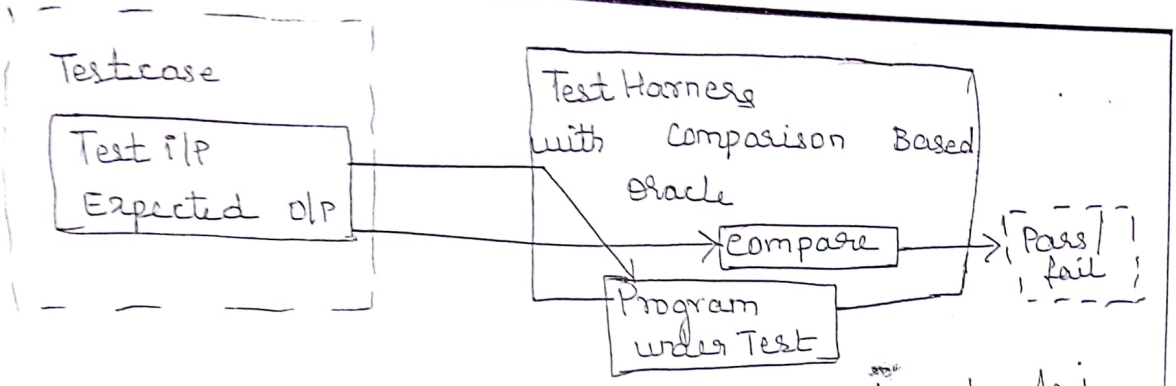


Fig:- A test harness with a comparison based test Oracle processes test cases consisting of (Prog i/p, Predicted o/p) pairs.

↳ A common misperception is that a test Oracle always requires Predicted Program o/p to compare to the o/p produced in a test execution. In fact it is often possible to judge o/p or behaviour without Predicting it.

↳ Oracles that check results without reference to a predicted o/p are often Partial, in the sense that they can detect some violations of the actual Specification but not others. They check necessary but not sufficient conditions for correctness.

↳ Similarly checking that a sort routine Produces Sorted o/p is simple & cheap, but it is only a Partial Oracle because the o/p is also required to be a Permutation of the i/p. A cheap partial Oracle that can be used for a large number of testcases is often combined with a more expensive Comparision-based Oracle that can be used with a smaller set of testcases for which Predicted o/p has been Obtained.

→ Self-checks as Oracles:

↳ The oracle can be incorporated into the Program under test so that it checks its own work

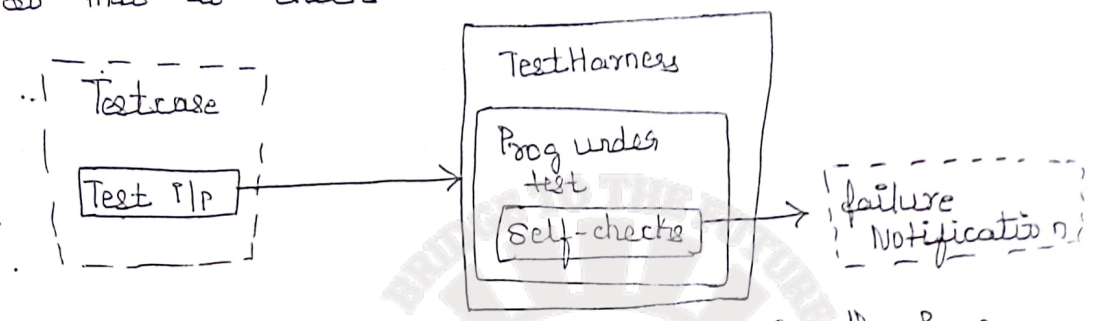


figure :- When Self-checks are embedded in the Program, testcases need not include Predicate outcomes.

↳ Self-checks assertions may be left in the Production Version of a System. Where they Provide much better diagnostic information than the uncontrolled applⁿ crash the outcomes may otherwise report.

↳ Side-effects free assertions are essential when assertions may be deactivated. because otherwise Suppressing assertions checking can introduce Program failures that appear only when one is not testing.

↳ Self-checks in the form of assertions embedded in Program code are useful Primarily for checking module and Subsystem-level Specifications, rather than overall Program behaviour.

↳ Devising Program assertions that correspond in a natural way to Specifications Poses 2 main challenges: bridging the gap b/w concrete execution values and abstractions used in Specifications and dealing in a reasonable way with quantification over collections of values.

→ The intended effect of an operation is described in terms of a Precondition and Postcondition, relating the concrete state to the abstract model. Consider again a specification of the get method of java.util.Map, with Pre & Post conditions expressed as the Hoare triple.

$$(|<k,v> \in \phi(dict)|)$$

$$D = dict.get(k)$$

$$(|D = v|)$$

ϕ is an abstraction function that constructs the abstract model type from the concrete data structure.

→ In addition to an abstraction function, reasoning about the correctness of internal structures usually involves structural invariants, that is, Properties of the data structure that are Preserved by all operations. structural invariants are good candidates for self checks implemented as assertions. They pertain directly to the concrete datastructure implementation and can be implemented within the module that encapsulates that data structure.

→ The doll figure illustrates an invariant check found in the source code of the eclipse Programming invariants

```
1. Package org.eclipse.jdt.internal.ui.text;
2. import java.text.CharacterIterator;
3. import org.eclipse.jface.text.Assert;
4 /*
5 * A <code> charsequence </code> based implementation R
6 * <code> characterIterator </code>.
7 * @ since 3.0
8 */
9 Public class sequenceCharacterIterator implements charac
      terIterator {
13....
```

```
14. Private void Invariant (){
15.  Assert.isTrue (fIndex >= fFirst);
16.  Assert.isTrue (fIndex <= fLast);
17. }
49. ....
50.  Public SequenceCharacterIterator (CharSequence Sequence,
                                      int first, int last)
51.  throws IllegalArgumentException {
52.  If (Sequence == null)
53.  throw new NullPointerException();
54.  if (first <0 || first > last)
55.  throw new IllegalArgumentException();
56.  If (last > Sequence.length())
57.  throw new IllegalArgumentException()
58.  fSequence = Sequence;
59.  fFirst = first;
60.  fLast = last;
61.  fIndex= first;
62.  invariant();
63.  }
148. ....
144.  Public class setIndex(int Position){
145.  if (Position >= getBeginIndex() && position <=
                                      getEndIndex())
146.  fIndex = Position;
147.  else
148.  throw new IllegalArgumentException();
149.
150.  invariant();
151   return current();
152  }
263 ....
264 }
```

→ It is sometimes straightforward to translate quantification in a specification statement into iterat -ion in a program assertion. Some run-time assertion checking systems provide quantifiers that are simply interpreted as loops. This approach can work when collections are small and quantifiers are not too deeply nested.

→ Treating quantifiers as loops doesnot scale well to large collections and cannot be applied at all when a specification quantifies over an infinite collection.

→ The problem of quantification over large sets of values is a variation on the basic problem of program testing, which is that we cant exhaustively check all program behaviors. Instead, we select a tiny fraction of possible program behaviors or I/P representatives. The same tactic is applicable to quantification in spec?

→ A final implementation problem for self-checks is that asserted properties sometimes involve values that are either not kept in the program at all or values that have been replaced. A spec of noninterference may use ghost variable to track entry and exit of threads in a concurrent program from a critical section.

→ The post condition of an in-place sort operation will state that the new value is sorted and a permutation relation refers to both the "before" and "after" values of the object to be sorted. A run-time assertion system must manage ghost variables and retained "before" values and must ensure that they have no sideeffects outside assertion checking.

→ Capture and Replay :

If one cannot completly avoid human involvement in testcase execution, one can atleast avoid unnec -ary repetition of this cost & opportunity for error

↳ The principle is simple. The first time such a test case is executed, the oracle function is carried out by a human, and the interaction sequence is captured. Provided the execution was judged to be correct, the captured log now forms an pair. for subsequent automated retesting.

↳ The savings from automated retesting with a captured log depends on how many build and test cycles we can continue to use it in, before it is invalidated by some change to the program.

↳ Mapping from concrete state to an abstract model of interaction sequences is sometimes possible but is generally quite limited. A more fruitful approach is capturing i/p and o/p behaviours at multiple levels of abstraction within the implementation.

↳ further amplification of the value of a captured log can be obtained by varying the logged events to obtain additional testcases.

Chapter-1

Structural

State

Chapter - 1

Structural Testing:

→ Statement and Block coverage:

Statement coverage: The statement coverage of T WRT (P, R) is computed as

$$|Sc| / (|Se| - |Si|).$$

Sc - set of statements covered
Si - " " unreachable statements
Se - " " statements in the program, that is the coverage domain. T is considered adequate WRT the statement coverage criterion if the statement coverage of T WRT (P,R) is 1.

Block coverage: The Block coverage of T WRT (P, R) is computed as

$$|Bc| / (|Be| - |Bi|).$$

Bc - set of blocks covered
Bi - set of unreachable blocks
Be - blocks in the Program, that is the block coverage domain. T is considered adequate WRT to the block coverage criterion if the block coverage of T WRT (P, R) is 1.

[unreachable - Infeasible path].

Example :-

```
1. begin
2. Int x, y;
3. Int z;
4. Input (x, y); z = 0;
5. if (x<0 and y<0) {
6.    z = x * x;
7.    if (y ≥ 0) z = z+1;
8. }
9. else z = x * x * x;
10. output (z);
11. }
12. end
```

→ Coverage domain corresponding to statement coverage for the Program above is given as

$S_c = \{2, 3, 4, 5, 6, 7, 7b, 9, 10\}$

[z = z+1] → 7b

Consider a test-case set T_1 that consists of 2 test cases against which Prog ↑ has been executed

$T_1 = \{ t_1 : <x = -1, y = -1>, t_2 : <x = 1, y = 1> \}$

Statements 2, 3, 4, 5, 6, 7 & 10 are covered upon the execution of P against t_1.

Similarly the execution against t_2 covers 2, 3, 4, 5, 9 & 10. Neither of the 2 testcases covers Statement 7b that is unreachable. as ~~one can conclude~~

Thus we obtain $|S_c| = 8$, $|S_1| = 1$, $|S_e| = 9$.

Statement Coverage for T is $8/(9-1) = 1$.

Hence Adequate for (P, R).

→ The 5 blocks in Prev Program are shown ↓.
The Coverage domain for the block coverage Criterion $B_c = \{1, 2, 3, 4, 5\}$.
Consider now a test set T_2 containing 3 tests against which Program ← Prev has been executed.

$$T_2 \begin{cases} t_1 < x = -1, y = -1 > \\ t_2 < x = -3, y = -1 > \\ t_3 < x = -1, y = -3 > \end{cases}$$

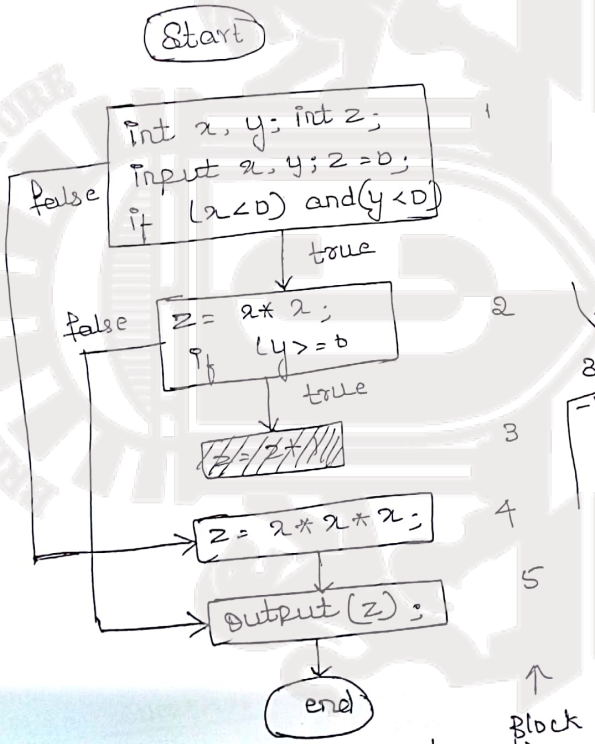


fig:- CFG of Program ←

The shaded block 3 is infeasible because the cond^n in block 2 will never be true.

Blocks 1, 2, 5 are covered when the Program is executed against test t_1. Test t_2 & t_3 also executed exactly the same set of blocks. for T_2 and Program ←, we obtain $|B_e| = 5$, $|B_c| = 3$ & $|B_i| = 1$.
The block coverage can now be computed as
$3/(5-1) = 0.75 < 1$ → not adequate.

→ Conditions and Decisions:

↳ An expression that evaluates to true / false constitutes a condition. Also known as Predicate.

A, x > y. A
B. A and (x < y)
(A and B) or (A and D) and (D)
(A xor B) and (x ≥ y)

A, B, D - Boolean Variables
x, y - integers.

xor, or, and — Boolean / logical operators.

1) Simple and compound conditions:

Simple:- does not use any boolean operators except ~~not~~ (¬) operators.
Made up of variable & one operator fro the set {<, ≤, >, ≥, ==, ≠}.

Compound:- Made up of 2/more simple cond^n joined by one / more Boolean operators

2) Conditions as decisions:

If, While & switch statements serve as contexts for decisions. If & While — one decisions
switch — contains more.

decisons outcomes
true, false, undefined

Ex:- 1. bool foo (int a_parameter)
2. { while (true) { //infinite loop.
3. a_parameter = 0;
4. }
5. }
:
6. if (x < y and foo (y)) { // undefined
7. Compute (x, y);

8) Coupled Conditions:-

C' = (A and B) or (Card A) — Compound Condition.
The first occurrence of A is said to be coupled to its second occurrence.

4) Conditions with assignments:

1. A = x < y ; // A simple cond' assigned to a Boolean Variable.
2. x = P or S ||
3. x = y + z * 8 ; if (x)
4. A = x < y ; x = A * B

↳ Condition evaluated before used as decision in a selection / loop statements.

→ Decision Coverage:

↳ also known as a branch decision coverage.
↳ A decision is covered if the flow of control has been diverted to all possible destinations that correspond to this decision. That is all outcomes of the decision have been taken.

Example:- Need for decision Coverage.

Consider the below Program, below. This Program inputs an integer x and if necessary transforms it into a +ve Value before invoking function foo-1 to compute the output z.

However as indicated, this Program has an error. As per its requirements, the Program is supposed to compute z using foo-2 when $x \geq 0$.

Consider the Test set T

$$T = \{ t_i \doteq \langle x = -5 \rangle \}$$

Program :-

```
1. begin
2. int x, z
3. input (x);
4. if (x<0)
5.    z = -x
6. z = foo-1 (x);
7. output (z); ← There should have been an else
                 clause before this statement.
8. end.
```

→ The above Program is adequate WRT Block and Statement coverage Criteria. But the execution doesn't force the condition inside ~~foo~~ this if to be evaluated to false thus avoiding the need to compute z using foo-2. Hence T doesn't reveal the error in this Program.

Program transforms function

→ Suppose a testcase is added to T to obtain an enhanced test set T',

$$T' = \{ t_1 : \langle x = -5 \rangle \; t_2 : \langle x = 3 \rangle \}$$

When the Program is executed against all tests in T', all statements and blocks in the Program are covered. The decision is also covered because $x < 0$ evaluates to true when the Program is executed against t_1 and to false when executed against t_2.

→ Control is directed to the statement at line 6 without executing line 5. This causes the value of z to be computed using f_{DD-1} and not f_{DD-2} as required. If $f_{DD-1}(3) \neq f_{DD-2}(3)$ then the program will give an incorrect o/p when executed against test t_2.

Decision Coverage:

The decision coverage of T WRT (P, R) is computed as $|Dc| / (|De| - |Di|)$,

Dc - set of decision covered

Di - set of infeasible decisions

De - set of decisions in the Program

T is considered adequate WRT the decision coverage criteria if the decision coverage of T WRT (P, R) is 1.

In the above Programe $|De| = 1$.

→ Condition Coverage:

↳ A decision can be composed of a simple condition such as $(x<0)$ or of a more complex condition such as $((x<0)$ and $(y<0))$ or $(P \geq q)$.

↳ Simple condition is covered if it evaluates to true and false, in one or more executions

Compound condition is covered if each simple condition is also covered.

Ex:- 1. if $(x<0$ and $y<0)$ {
2. $z = foo(x, y)$;

The condition coverage of T WRT (P, R) is computed as

$$|cc| = (|Ce| - |Ci|)$$

Cc - set of simple condition covered.
Ci - set of infeasible simple conditions
Ce - set of simple conditions in the Program

T is considered adequate WRT to the condition coverage criterion if the condition coverage of T WRT (P, R) is 1.

Alternate formula :-

$$\frac{|Cc|}{2 \times (|Ce| - |Ci|)}$$

Example :- Consider the foll Program that inputs values of x and y and computes the output z using functions fool and foo 2.

Partial Specifications for this Program are given in the foll table.

Table :- Truth table for the computation of z in Program ↓

x < 0	y < 0	Output (z)
true	true	fOO1 (x, y)
true	false	(fOO 2 (x, y))
false	true	fOO 2 (x, y)
false	false	fOO 2 (x, y).

Program :-

```
1. begin
2. int x, y, z;
3. input (x, y);
4. if (x<0 and y<0)
5.    z = fOO1 (x, y);
6. else
7.    z = fOO 2 (x, y);
8. output (z);
9. end.
```

→ The above table reveals that for x≥0 and y≥0 the program incorrectly computes z as fOO 2 (x, y).

T is adequate WRT the statement coverage & block coverage and the decision coverage criteria.

→ Note that Ce = {x<0, y<0}. Tests in T cover only the second of the 2 elements in Ce. As both conditions in Ce are feasible, |Ci| = 0.

condition coverage = 1/(2-0) = 0.5.
for T