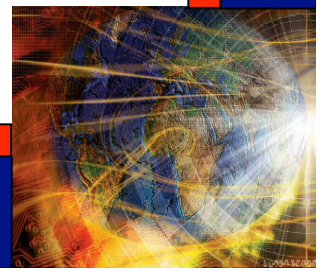
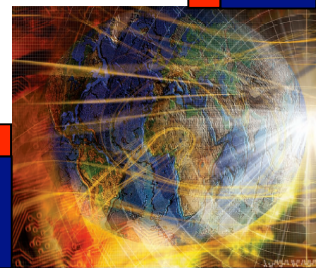
Chapter 10

Data Flow Testing
Slice Testing



Data Flow Testing

- Often confused with "dataflow diagrams“.
- Main concern: places in a program where data values are defined and used.
- Static (compile time) and dynamic (execution time) versions.
- Static: Define/Reference Anomalies on a variable that
  - is defined but never used (referenced)
  - is used but never defined
  - is defined more than once
- Starting point is a program, P, with program graph G(P), and the set V of variables in program P.
- "Interesting" data flows are then tested as mini-functions.

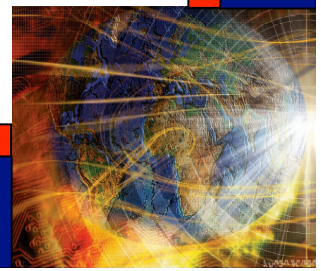


Definitions

- Node n ∈ G(P) is a *defining node of the variable* v ∈ V, written as DEF(v, n), iff the value of the variable v is defined at the statement fragment corresponding to node n.

- Node n ∈ G(P) is a *usage node of the variable* v ∈ V, written as USE(v, n), iff the value of the variable v is used at the statement fragment corresponding to node n.

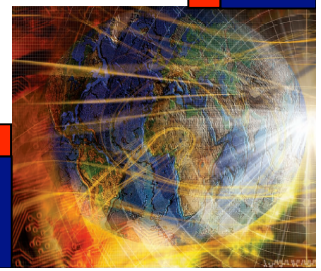
- A usage node USE(v, n) is a *predicate use* (denoted as P-use) iff the statement n is a predicate statement; otherwise, USE(v, n) is a *computation use* (denoted C-use).



More Definitions

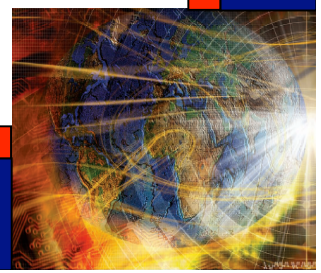
- A *definition-use path with respect to a variable* v (denoted du-path) is a path in PATHS(P) such that for some v ∈ V, there are define and usage nodes DEF(v, m) and USE(v, n) such that m and n are the initial and final nodes of the path.

- A *definition-clear path with respect to a variable* v (denoted dc-path) is a definition-use path in PATHS(P) with initial and final nodes DEF (v, m) and USE (v, n) such that no other node in the path is a defining node of v.



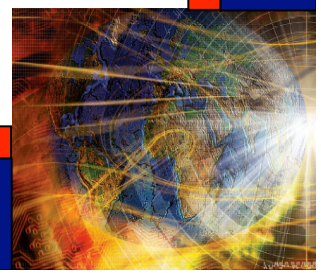
Example: first part of the Commission Program

```
1.  Program Commission (INPUT,OUTPUT)
2.  Dim locks, stocks, barrels As Integer
3.  Dim lockPrice, stockPrice, barrelPrice As Real
4.  Dim totalLocks, totalStocks, totalBarrels As Integer
5.  Dim lockSales, stockSales, barrelSales As Real
6.  Dim sales, commission As Real
7.  lockPrice = 45.0
8.  stockPrice = 30.0
9.  barrelPrice = 25.0
10. totalLocks = 0
11. totalStocks = 0
12. totalBarrels = 0
13. Input(locks)
14. While NOT(locks = -1)
15.     Input(stocks, barrels)
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 sold: ", totalStocks)
23.  Output("Barrels sold: ", totalBarrels)
```

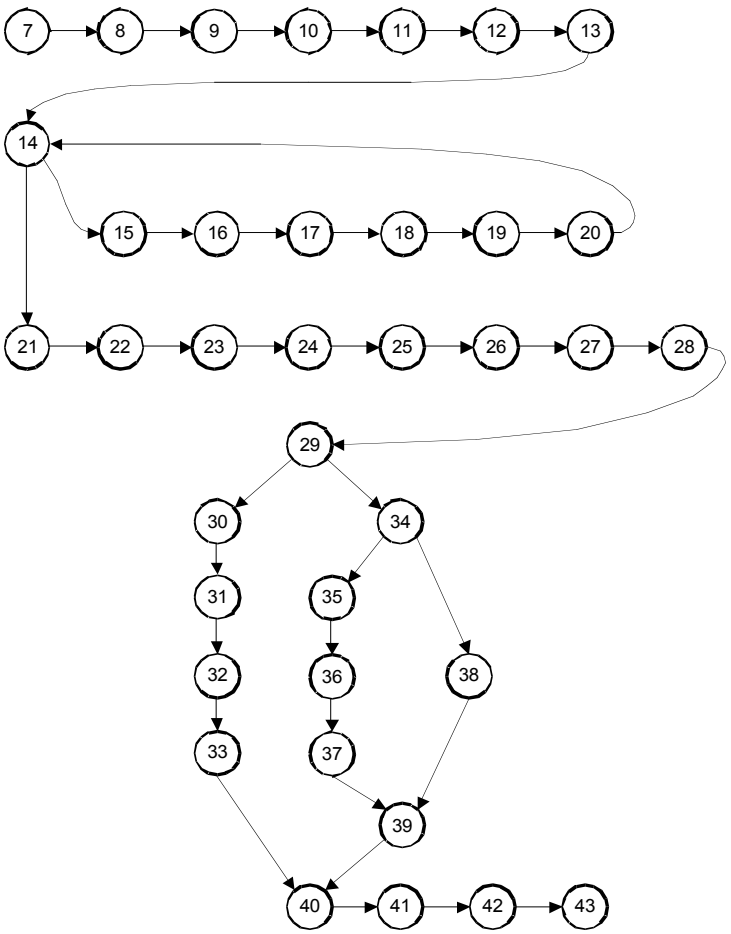


Rest of Commission Problem

```
23. Output("Barrels sold: ", totalBarrels)
24. lockSales = lockPrice * totalLocks
25. stockSales = stockPrice * totalStocks
26. barrelSales = barrelPrice * totalBarrels
27. sales = lockSales + stockSales + barrelSales
28. Output("Total sales: ", sales)
29. If (sales > 1800.0)
30.   Then
31.     commission = 0.10 * 1000.0
32.     commission = commission + 0.15 * 800.0
33.     commission = commission + 0.20 *(sales-1800.0)
34.   Else If (sales > 1000.0)
35.             Then
36.               commission = 0.10 * 1000.0
37.               commission = commission + 0.15 *(sales-1000.0)
38.             Else
39.               commission = 0.10 * sales
40.           EndIf
41.  EndIf
42. Output("Commission is $", commission)
43. End Commission
```

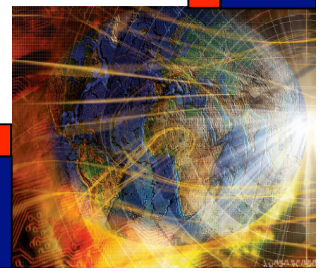


Program Graph of Commission Problem



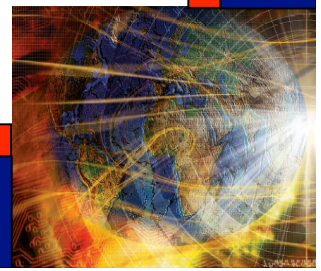
Define/Use Test Cases

- **Technique: for a particular variable,**
  - **find all its definition and usage nodes, then**
  - **find the du-paths and dc-paths among these.**
  - **for each path, devise a "suitable" (functional?) set of test cases.**
- **Note: du-paths and dc-paths have both static and dynamic interpretations**
  - **Static: just as seen in the source code**
  - **Dynamic: must consider execution-time flow (particularly for loops)**
- **Definition clear paths are easier to test**
  - **No need to check each definition node, as is necessary for du-paths**



Define and Use Nodes

Variable	*Defined at Node*	*Used at Node*
locks	13, 19	14, 16
stocks	15	17
barrels	15	18
totalLocks	10, 16	16, 21, 24
totalStocks	11, 17	17, 22, 25
totalBarrels	12, 18	18, 23, 26

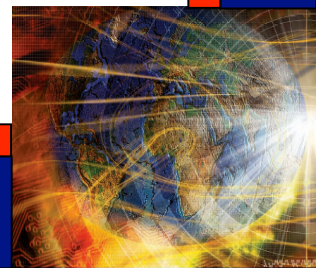


Example (continued)

```
13. Input(locks)
14. While NOT(locks = -1)
15.     Input(stocks, barrels)
16.     totalLocks = totalLocks + locks
17.     totalStocks = totalStocks + stocks
18.     totalBarrels = totalBarrels + barrels
19.     Input(locks)
20.  EndWhile
```

We have

- DEF (locks, 13), DEF (locks, 19)
- USE (locks, 14), a predicate use
- USE (locks, 16). A computation use
- du-paths for locks are the node sequences <13, 14> (a dc-path), <13, 14, 15, 16>, <19, 20, 14 >, < 19, 20, 14 , 15, 16>
- Is <13, 14, 15, 16> definition clear?
- Is < 19, 20, 14, 15, 16> definition clear? What about repetitions?

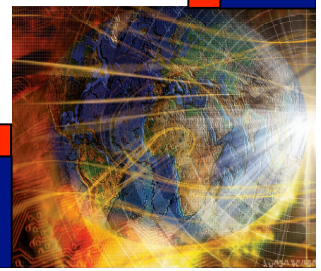


Coverage Metrics Based on du-paths

- 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.

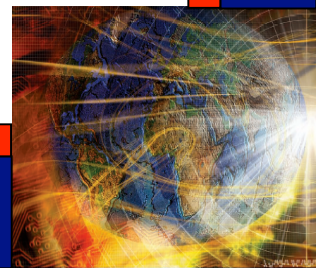
- The set T satisfies the *All-Defs criterion* for the program P iff for every variable v ∈ V, T contains definition-clear paths from every defining node of v to a use of v.

- The set T satisfies the *All-Uses criterion* for the program P iff for every variable v ∈ 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).



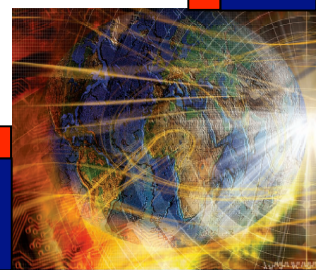
Coverage Metrics Based on du-paths
(continued)

- 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 at least one computation use.



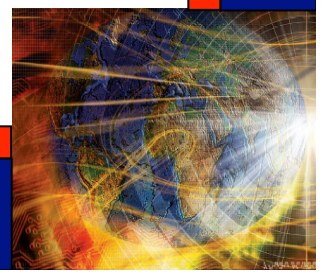
Coverage Metrics Based on du-paths
(continued)

- 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 at least one predicate use.

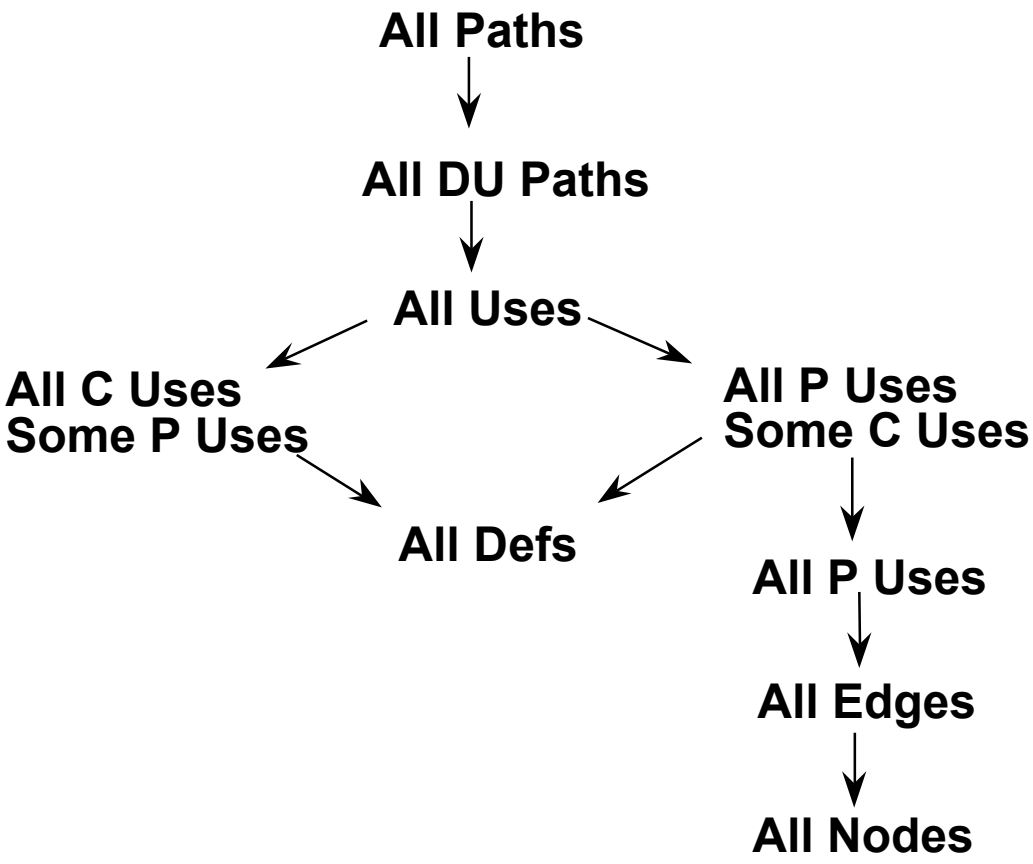


Coverage Metrics Based on du-paths
(concluded)

- The set T satisfies the *All-du-paths criterion* for the program P iff for every variable v ∈ 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), and that these paths are either single-loop traversals or cycle-free.

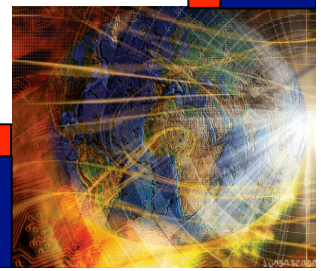


Rapps-Weyuker Coverage Subsumption



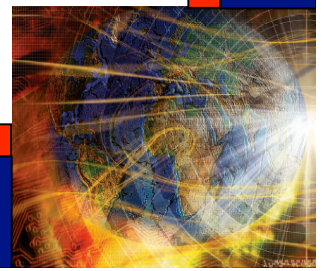
S. Rapps and E. J. Weyuker
"Selecting Software Test Data Using Data Flow Information"
IEEE Transactions of Software Engineering vol 11 no 4 IEEE Computer Society Press, Washington, D. C. , April 1985, pp 367 - 375.

Exercise: Where does the “All definition-clear paths” coverage metric fit in the Rapps-Weyuker lattice?



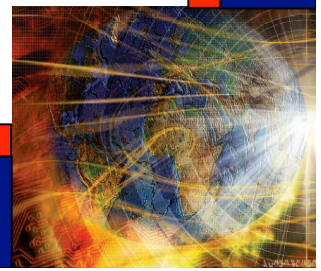
Data Flow Testing Strategies

- Data flow testing is indicated in
  - Computation-intensive applications
  - "long" programs
  - Programs with many variables
- A definition-clear du-path represents a small function that can be tested by itself.
- If a du-path is not definition-clear, it should be tested for each defining node.



Slice Testing

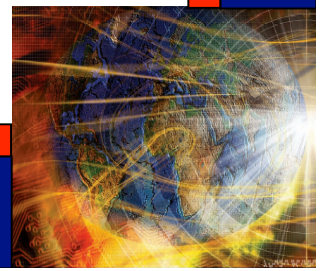
- Often confused with "module execution paths"
- Main concern: portions of a program that "contribute" to the value of a variable at some point in the program.
- Nice analogy with history -- a way to separate a complex system into "disjoint" components that interact:
  - European history
  - North American history
  - Orient history
- A dynamic construct.



Slice Testing Definitions

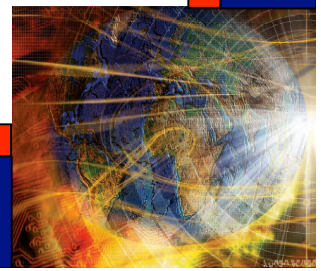
Starting point is a program, P, with program graph G(P), and the set V of variables in program P. Nodes in the program graph are numbered and correspond to statement fragments.

- Definition: 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 P prior to n that contribute to the values of variables in V at statement fragment n.
- This is actually a "backward slice".
- Exercise: define a "forward slice".



Fine Points

- "prior to" is the dynamic part of the definition.
- "contribute" is best understood by extending the Define and Use concepts:
  - P-use: used in a predicate (decision)
  - C-use: used in computation
  - O-use: used for output
  - L-use: used for location (pointers, subscripts)
  - I-use: iteration (internal counters, loop indices)
  - I-def: defined by input
  - A-def: defined by assignment
- usually, the set V of variables consists of just one element.
- can choose to define a slice as a compilable set of statement fragments -- this extends the meaning of "contribute"
- because slices are sets, we can develop a lattice based on the subset relationship.



In the program fragment

```
13. Input(locks)
14. While NOT(locks = -1)
15.     Input(stocks, barrels)
16.     totalLocks = totalLocks + locks
17.     totalStocks = totalStocks + stocks
18.     totalBarrels = totalBarrels + barrels
19.     Input(locks)
20.EndWhile
```

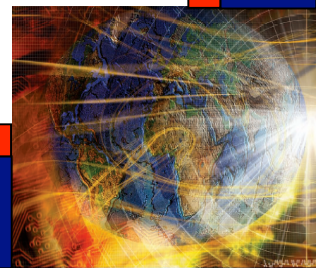
There are these slices on locks (notice that statements 15, 17, and 18 do not appear):

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

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

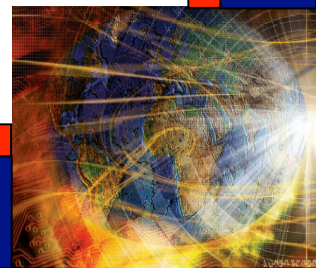
S3: S(locks, 16) = {13, 14, 19, 20}

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

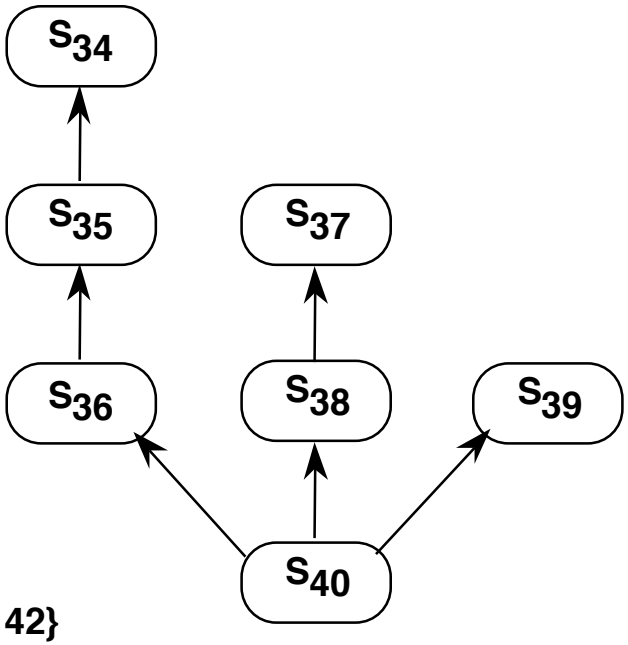


Lattice of Slices

- Because a slice is a set of statement fragment numbers, we can find slices that are subsets of other slices.
- This allows us to “work backwards” from points in a program, presumably where a fault is suspected.
- The statements leading to the value of commission when it is output are an excellent example of this pattern.
- Some researchers propose that this is the way good programmers think when they debug code.



Example Lattice of Slices



S34: S(commission, 41) = {41}

S35: S(commission, 42) = {41, 42}

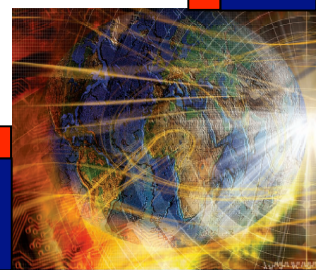
S36: S(commission, 43) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 36, 41, 42, 43}

S37: S(commission, 47) = {47}

S38: S(commission, 48) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36, 47, 48}

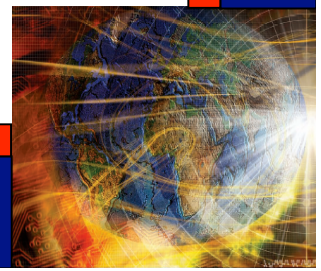
S39: S(commission, 50) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36, 50}

S40: S(commission, 51) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36, 41, 42, 43, 47, 48, 50}



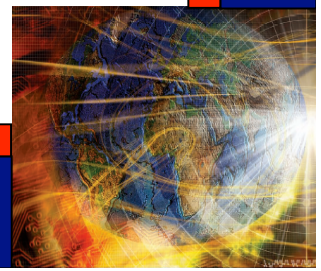
Diagnostic Testing with Slices

- Relative complements of slices yield a "diagnostic" capability. The relative complement of a set B with respect to another set A is the set of all elements of A that are not elements of B. It is denoted as A - B.
- Consider the relative complement set S(commission, 48) - S(sales, 35):
  - S(commission, 48) = {3, 4, 5,36,18,19, 20, 23, 24, 25, 26, 27, 34, 38, 39, 40, 44,45,47}
  - S(sales, 35) = {3, 4, 5, 36, 18, 19, 20, 23, 24, 25, 26, 27}
  - S(commission, 48) - S(sales, 35) = {34, 38, 39, 40, 44,45,47}
- If there is a problem with commission at line 48, we can divide the program into two parts, the computation of sales at line 34, and the computation of commission between lines 35 and 48. If sales is OK at line 34, the problem must lie in the relative complement; if not, the problem may be in either portion.



Programming with Slices

- One researcher suggests the possibility of "slice splicing":
  - Code a slice, compile and test it.
  - Code another slide, compile and test it, then splice the two slices.
  - Continue until the whole program is complete.
- Exercise: in what ways is slice splicing distinct from agile (bottom up) programming?



Exercise/Discussion: When should testing stop?

- when you run out of time?
- when continued testing causes no new failures?
- when continued testing reveals no new faults?
- when you can't think of any new test cases?
- when you reach a point of diminishing returns?
- when mandated coverage has been attained?
- when all faults have been removed?