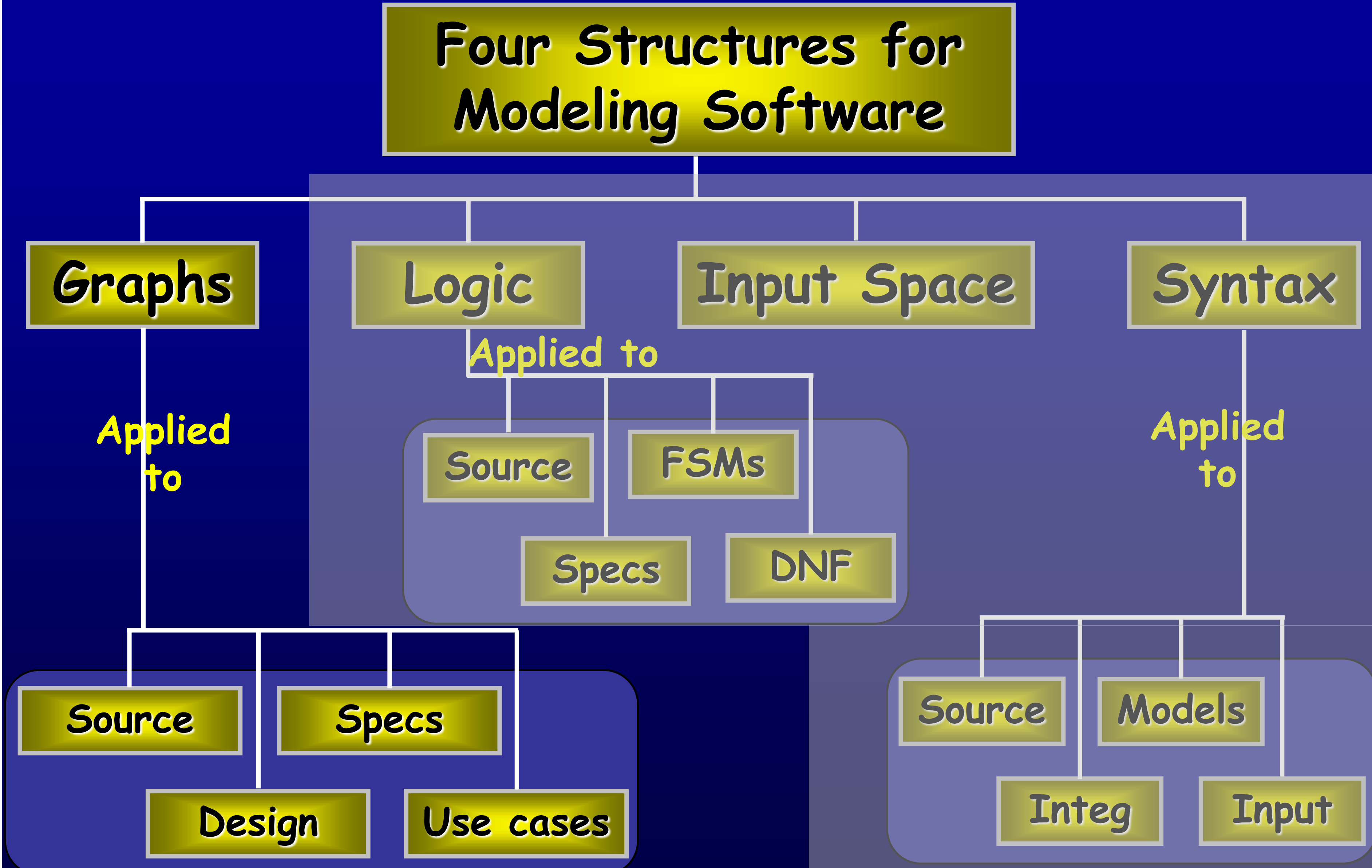
Introduction to Software Testing
Chapter 2.1, 2.2
Overview Graph Coverage Criteria

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

Ch. 2 : Graph Coverage



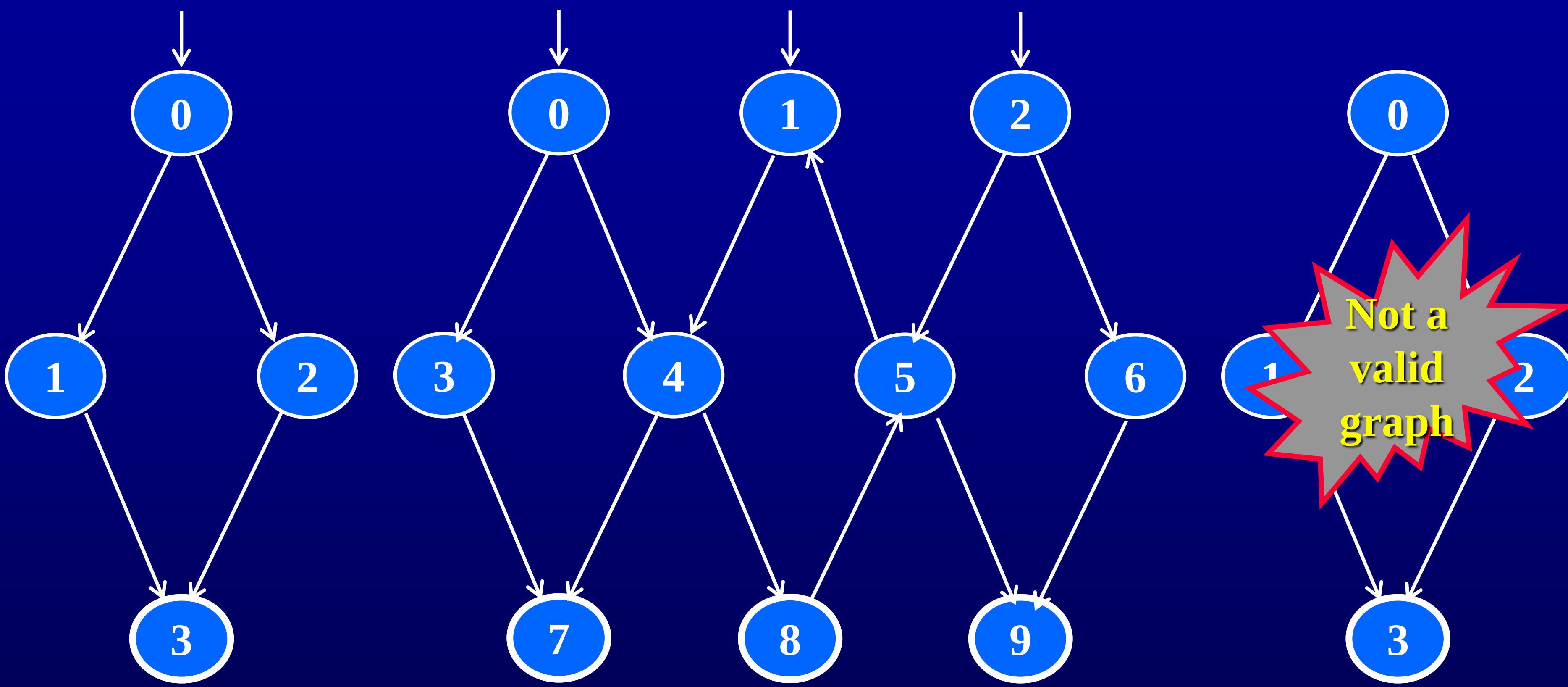
Covering Graphs (2.1)

- **Graphs are the most commonly used structure for testing**
- **Graphs can come from many sources**
  - **Control flow graphs**
  - **Design structure**
  - **FSMs and statecharts**
  - **Use cases**
- **Tests usually are intended to “cover” the graph in some way**

Definition of a Graph

- **A set N of <u>nodes</u>, N is not empty**
- **A set N_0 of <u>initial nodes</u>, N_0 is not empty**
- **A set N_f of <u>final nodes</u>, N_f is not empty**
- **A set E of <u>edges</u>, each edge from one node to another**
  - **(n_i , n_j), i is predecessor, j is successor**

Three Example Graphs



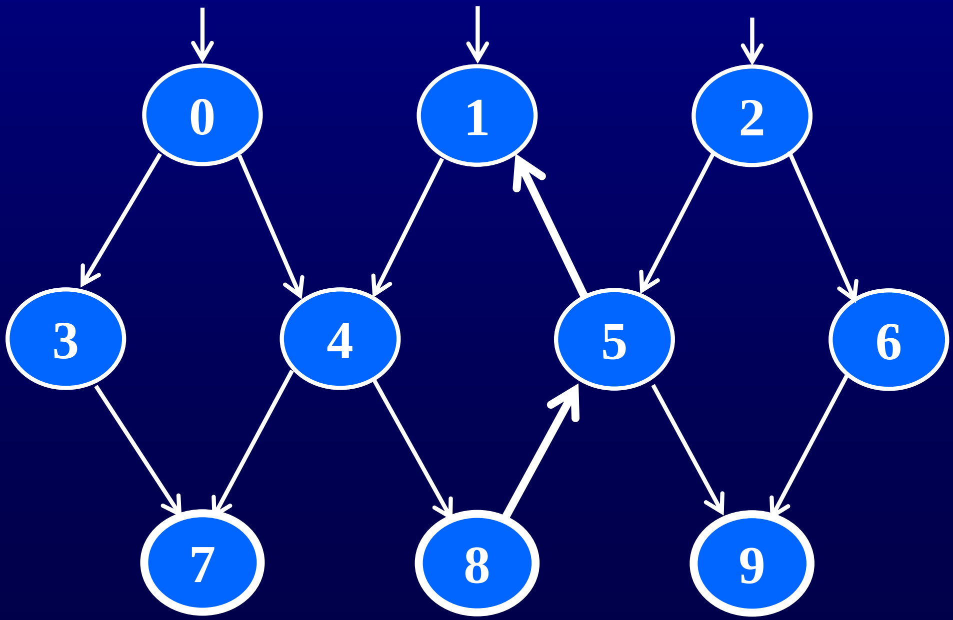
$N_0 = \{ 0 \}$ $N_f = \{ 3 \}$

$N_0 = \{ 0, 1, 2 \}$ $N_f = \{ 7, 8, 9 \}$

$N_0 = \{ \}$ $N_f = \{ 3 \}$

Paths in Graphs

- **Path** : A sequence of nodes – $[n_1, n_2, \ldots, n_M]$
  - Each pair of adjacent nodes is an edge
- **Length** : The number of edges
  - A single node is a path of length 0
- **Subpath** : A subsequence of nodes in *p* is a subpath of *p*
- **Reach** (*n*) : Subgraph that can be reached from *n*



A Few Paths

[0, 3, 7]

[1, 4, 8, 5, 1]

[2, 6, 9]

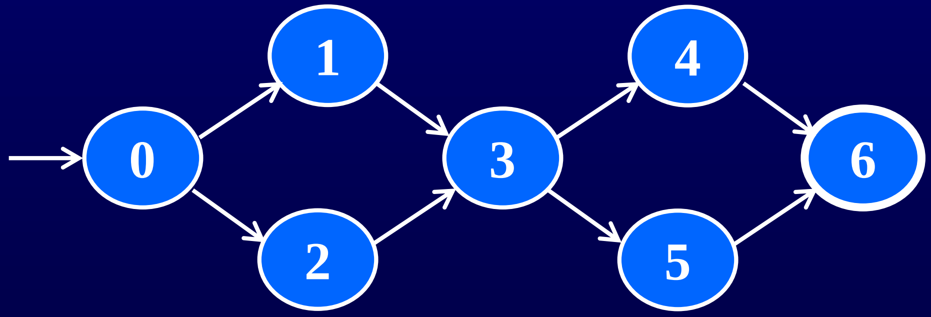
Reach (0) = { 0, 3, 4, 7, 8, 5, 1, 9 }

Reach ({0, 2}) = G

Reach([2,6]) = {2, 6, 9}

Test Paths and SESEs

- **Test Path** : A path that starts at an initial node and ends at a final node
- Test paths represent execution of test cases
  - Some test paths can be executed by many tests
  - Some test paths cannot be executed by any tests
- **SESE graphs** : All test paths start at a single node and end at another node
  - Single-entry, single-exit
  - N0 and Nf have exactly one node



Double-diamond graph

Four test paths

[0, 1, 3, 4, 6]

[0, 1, 3, 5, 6]

[0, 2, 3, 4, 6]

[0, 2, 3, 5, 6]

Visiting and Touring

- **Visit** : A test path *p* *visits* node *n* if *n* is in *p*
  A test path *p* *visits* edge *e* if *e* is in *p*
- **Tour** : A test path *p* *tours* subpath *q* if *q* is a subpath of *p*

Path [0, 1, 3, 4, 6]

Visits nodes 0, 1, 3, 4, 6

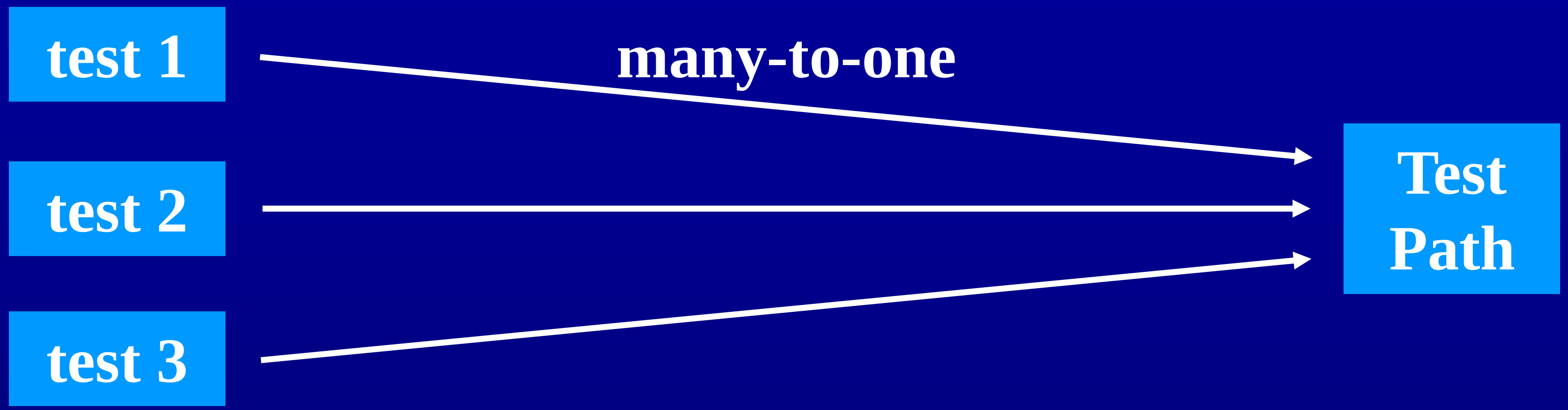
Visits edges (0, 1), (1, 3), (3, 4), (4, 6)

Tours subpaths [0, 1, 3], [1, 3, 4], [3, 4, 6], [0, 1, 3, 4], [1, 3, 4, 6]

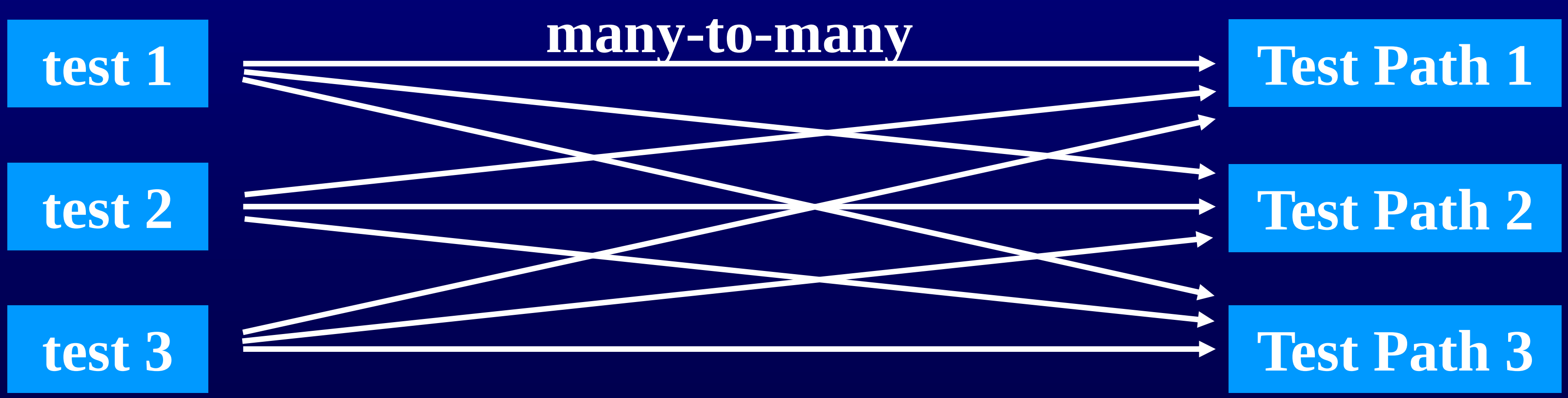
Tests and Test Paths (張智)

- **path (*t*) : The test path executed by test *t***
- **path (*T*) : The set of test paths executed by the set of tests *T***
- **Each test executes one and only one test path**
- **A location in a graph (node or edge) can be reached from another location if there is a sequence of edges from the first location to the second**
  - ***Syntactic reach* : A subpath exists in the graph**
  - ***Semantic reach* : A test exists that can execute that subpath**

Tests and Test Paths(張智)



Deterministic software – a test always executes the same test path



Non-deterministic software – a test can execute different test paths

Testing and Covering Graphs (張至潔)

- **We use graphs in testing as follows :**
  - **Developing a model of the software as a graph**
  - **Requiring tests to visit or tour specific sets of nodes, edges or subpaths**
- **Test Requirements (TR) : Describe properties of test paths**
- **Test Criterion : Rules that define test requirements**
- **Satisfaction :** ***Given a set TR of test requirements for a criterion C, a set of tests T satisfies C on a graph if and only if for every test requirement in TR, there is a test path in path(T) that meets the test requirement tr***
- **Structural Coverage Criteria : Defined on a graph just in terms of nodes and edges**
- **Data Flow Coverage Criteria : Requires a graph to be annotated with references to variables**

Node and Edge Coverage (林修博)

- The first (and simplest) two criteria require that each node and edge in a graph be executed

Node Coverage (NC) : Test set *T* satisfies node coverage on graph *G* iff for every syntactically reachable node *n* in *N*, there is some path *p* in *path(T)* such that *p* visits *n*.

- This statement is a bit cumbersome, so we abbreviate it in terms of the set of test requirements

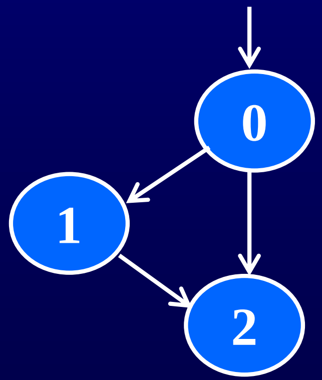
Node Coverage (NC) : TR contains each reachable node in G.

Node and Edge Coverage

- Edge coverage is slightly stronger than node coverage

<u>Edge Coverage (EC)</u> : TR contains each reachable path of length up to 1, inclusive, in G.

- The phrase "*length up to 1*" allows for graphs with one node and no edges
- NC and EC are only different when there is an edge and another subpath between a pair of nodes (as in an "if-else" statement)

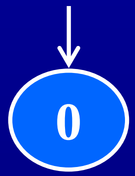


<u>Node Coverage</u> : TR = { 0, 1, 2 }
Test Path = [0, 1, 2]

<u>Edge Coverage</u> : TR = { 0,1,2,(0,1), (0, 2), (1, 2) }
Test Paths = [0, 1, 2]
[0, 2]

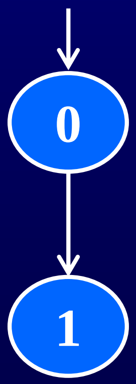
Paths of Length 1 and 0

- A graph with **only one node** will not have any edges



- It may be seem trivial, but formally, Edge Coverage needs to require Node Coverage on this graph
- Otherwise, Edge Coverage will not subsume Node Coverage
  - So we define "**length up to 1**" instead of simply "length 1"

- We have the same issue with graphs that only have **one edge** – for Edge Pair Coverage …



Covering Multiple Edges

- Edge-pair coverage requires pairs of edges, or subpaths of length 2

<u>Edge-Pair Coverage (EPC)</u> : TR contains each reachable path of length up to 2, inclusive, in G.

- The phrase "length up to 2" is used to include graphs that have less than 2 edges

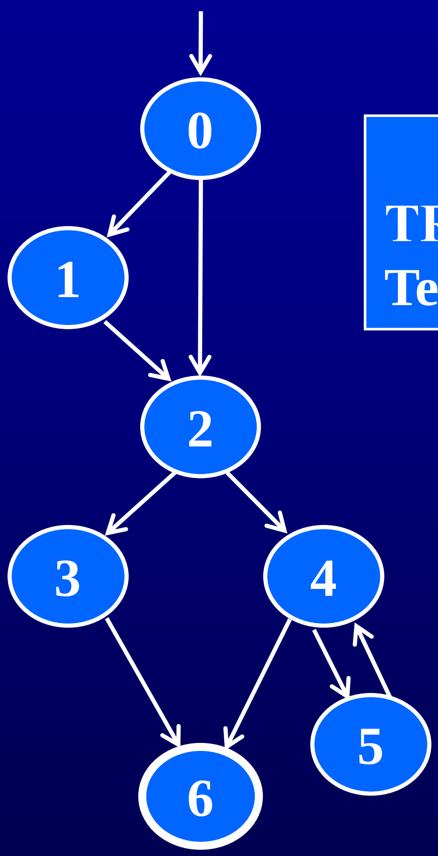
- The logical extension is to require all paths …

<u>Complete Path Coverage (CPC)</u> : TR contains all paths in G.

- Unfortunately, this is impossible if the graph has a loop, so a weak compromise is to make the tester decide which paths:

<u>Specified Path Coverage (SPC)</u> : TR contains a set S of test paths, where S is supplied as a parameter.

Structural Coverage Example



Node Coverage

TR = { 0, 1, 2, 3, 4, 5, 6 }

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 5, 4, 6]

Edge Coverage

TR = {…, (0,1), (0,2), (1,2), (2,3), (2,4), (3,6), (4,5), (4,6), (5,4) }

Test Paths: [0, 1, 2, 3, 6] [0, 2, 4, 5, 4, 6]

Edge-Pair Coverage

TR = {…, [0,1,2], [0,2,3], [0,2,4], [1,2,3], [1,2,4], [2,3,6], [2,4,5], [2,4,6], [4,5,4], [5,4,5], [5,4,6] }

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 6] [0, 2, 3, 6] [0, 2, 4, 5, 4, 5, 4, 6]

Complete Path Coverage

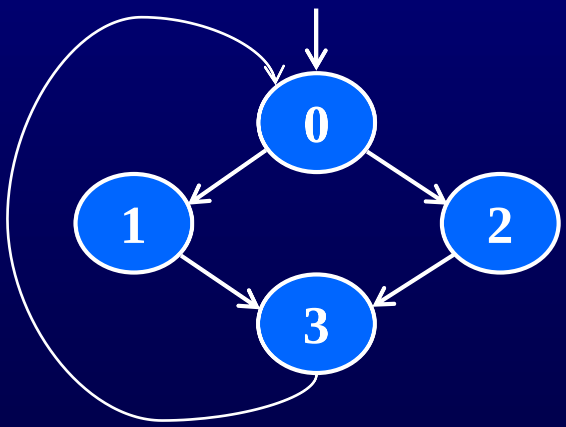
Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 6] [0, 1, 2, 4, 5, 4, 6] [0, 1, 2, 4, 5, 4, 5, 4, 6] [0, 1, 2, 4, 5, 4, 5, 4, 5, 4, 6] …

Loops in Graphs

- **If a graph contains a loop, it has an infinite number of paths**
- **Thus, CPC is not feasible**
- **SPC (specified path coverage) is not satisfactory because the results are subjective and vary with the tester**
- **Attempts to "deal with" loops:**
  - **1970s : Execute cycles once ([4, 5, 4] in previous example, informal)**
  - **1980s : Execute each loop, exactly once (formalized)**
  - **1990s : Execute loops 0 times, once, more than once (informal description)**
  - **2000s : Prime paths**

Simple Paths and Prime Paths

- **Simple Path** : ***A path from node ni to nj is simple if no node appears more than once, except possibly the first and last nodes are the same***
  - No internal loops
  - A loop is a simple path
- **Prime Path** : ***A simple path that does not appear as a proper subpath of any other simple path***



Simple Paths : [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1], [0, 1, 3], [0, 2, 3], [1, 3, 0], [2, 3, 0], [3, 0, 1], [3, 0, 2], [0, 1], [0, 2], [1, 3], [2, 3], [3, 0], [0], [1], [2], [3]

Prime Paths : [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1]

Prime Path Coverage

- A simple, elegant and finite criterion that requires loops to be executed as well as skipped

Prime Path Coverage (PPC) : TR contains each prime path in G.

- Will tour all paths of length 0, 1, …
- That is, it subsumes node and edge coverage
- Note : The book has a mistake, PPC does NOT subsume EPC
  - If a node *n* has an edge to itself, EPC will require [*n*, *n*, *m*]
  - [*n*, *n*, *m*] is not prime

Round Trips

- **Round-Trip Path** : ***A prime path that starts and ends at the same node***

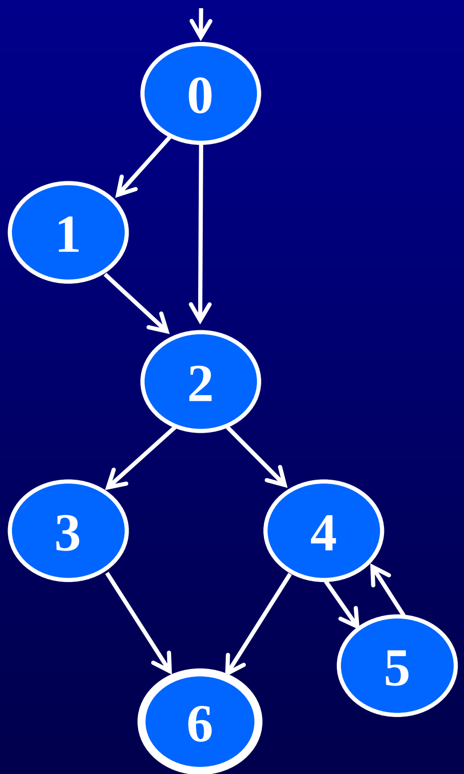
Simple Round Trip Coverage (SRTC) : TR contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.

Complete Round Trip Coverage (CRTC) : TR contains all round-trip paths for each reachable node in G.

- **These criteria omit nodes and edges that are not in round trips**
- **That is, they do not subsume edge-pair, edge, or node coverage**

Prime Path Example

- The previous example has 38 simple paths
- Only nine *prime paths*



Prime Paths

[0, 1, 2, 3, 6]
[0, 1, 2, 4, 5]
[0, 1, 2, 4, 6]
[0, 2, 3, 6]
[0, 2, 4, 5]
[0, 2, 4, 6]
[5, 4, 6]
[4, 5, 4]
[5, 4, 5]

Execute loop 0 times

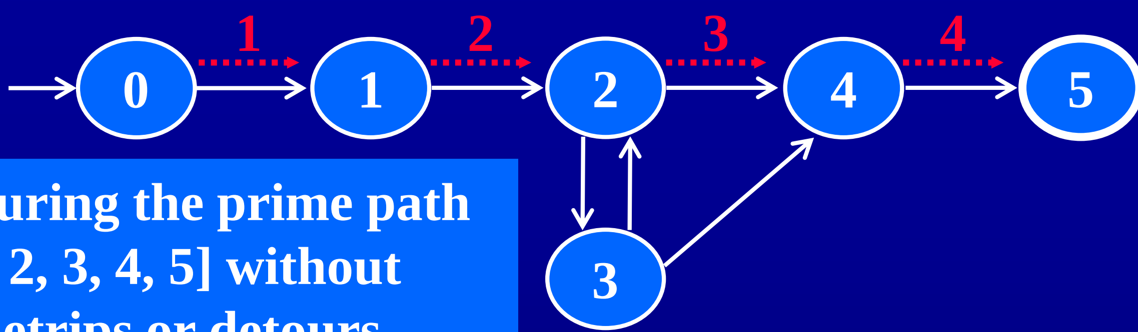
Execute loop once

Execute loop more than once

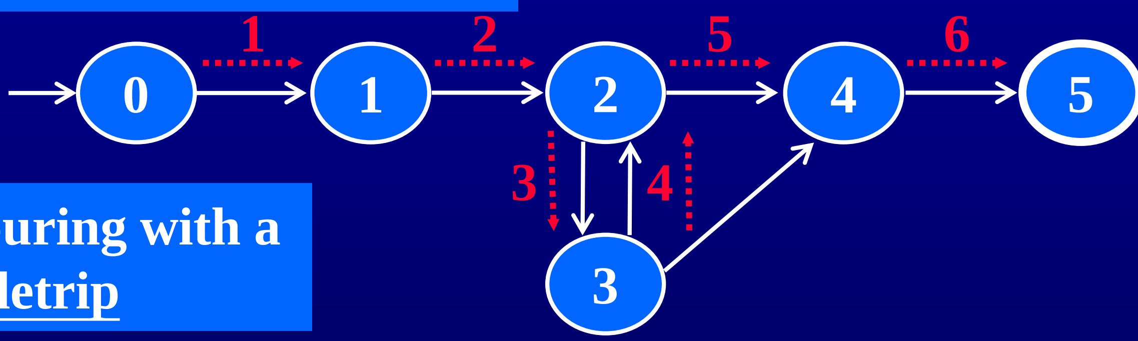
Touring, Sidetrips and Detours

- **Prime paths do not have internal loops … test paths <u>might</u>**
- **<u>Tour</u> : *A test path p tours subpath q if q is a subpath of p***
- **<u>Tour With Sidetrips</u> : *A test path p tours subpath q with <u>sidetrips</u> iff every <u>edge</u> in q is also in p in <u>the same order</u>***
  - **The tour can include a sidetrip, as long as it comes back to the same node**
- **<u>Tour With Detours</u> : *A test path p tours subpath q with <u>detours</u> iff every <u>node</u> in q is also in p in <u>the same order</u>***
  - **The tour can include a detour from node *ni*, as long as it comes back to the prime path at a successor of *ni***

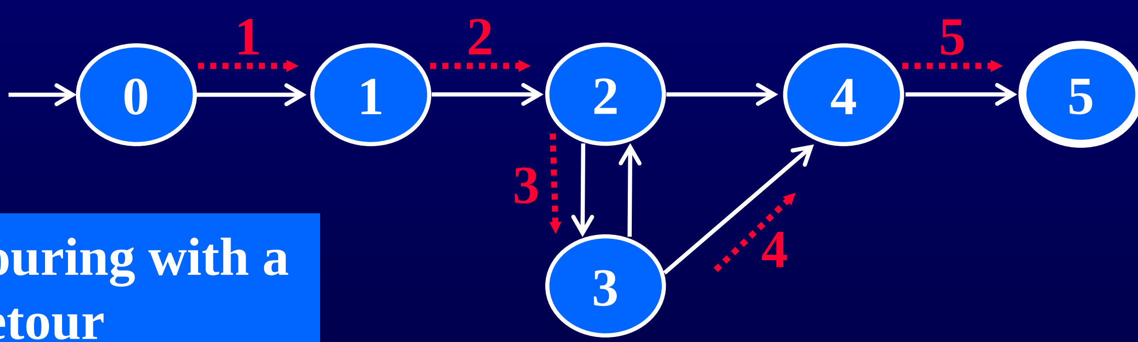
Sidetrips and Detours Example



Touring the prime path [1, 2, 3, 4, 5] without <u>sidetrips or detours</u>



Touring with a <u>sidetrip</u>



Touring with a <u>detour</u>

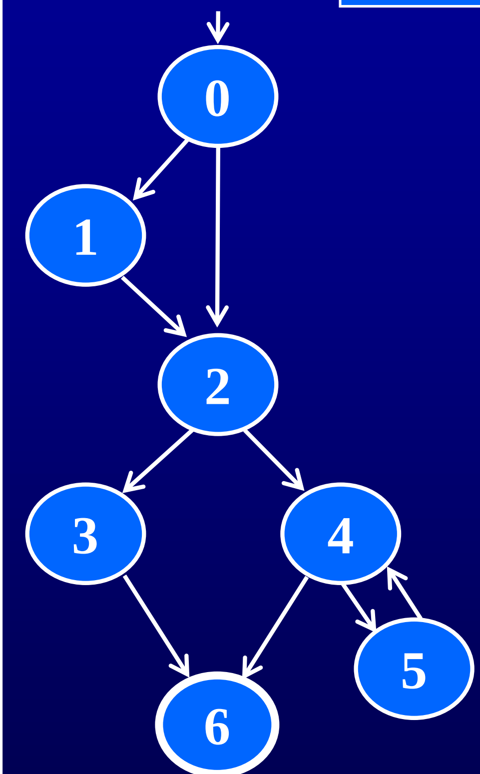
Infeasible Test Requirements

- An infeasible test requirement cannot be satisfied
  - Unreachable statement (dead code)
  - A subpath that can only be executed if a contradiction occurs ($X > 0$ and $X < 0$)
- Most test criteria have some infeasible test requirements
- It is usually undecidable whether all test requirements are feasible
- When sidetrips are not allowed, many structural criteria have more infeasible test requirements
- However, always allowing sidetrips weakens the test criteria

Practical recommendation – Best Effort Touring

- Satisfy as many test requirements as possible without sidetrips
- Allow sidetrips to try to satisfy unsatisfied test requirements

Simple & Prime Path Example



Simple paths

'!' means path
es

s path
les

Len 0
[0]
[1]
[2]
[3]
[4]
[5]
[6] !

Len 1
[0, 1]
[0, 2]
[1, 2]
[2, 3]
[2, 4]
[3, 6] !
[4, 6] !
[4, 5]
[5, 4]

Len 2
[0, 1, 2]
[0, 2, 3]
[0, 2, 4]
[1, 2, 3]
[1, 2, 4]
[2, 3, 6] !
[2, 4, 6] !
[2, 4, 5]
[4, 5, 4] *
[5, 4, 6] !
[5, 4, 5] *

Len 3
[0, 1, 2, 3]
[0, 1, 2, 4]
[0, 2, 3, 6] !
[0, 2, 4, 6] !
[0, 2, 4, 5]
[1, 2, 3, 6] !
[1, 2, 4, 5]
[1, 2, 4, 6] !

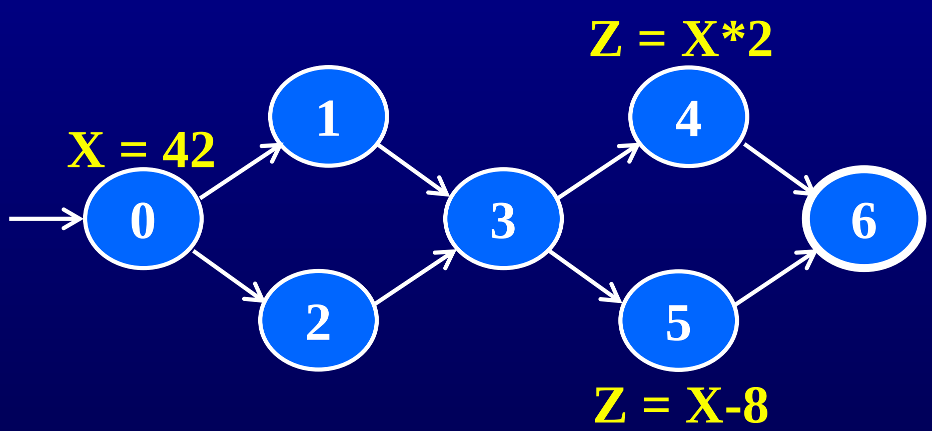
Len 4
[0, 1, 2, 3, 6] !
[0, 1, 2, 4, 6] !
[0, 1, 2, 4, 5]

Prime Paths

Data Flow Criteria

Goal: Try to ensure that values are computed and used correctly

- **Definition (def) : A location where a value for a variable is stored into memory**
- **Use : A location where a variable's value is accessed**



Defs: def (0) = {X}
def (4) = {Z}
def (5) = {Z}
Uses: use (4) = {X}
use (5) = {X}

The values given in defs should reach at least one, some, or all possible uses

DU Pairs and DU Paths

- **def (n) or def (e)** : The set of variables that are defined by node n or edge e
- **use (n) or use (e)** : The set of variables that are used by node n or edge e

- **DU pair** : A pair of locations (l_i, l_j) such that a variable v is defined at l_i and used at l_j

DU Pairs and DU Paths

- **Def-clear** : A path from l_i to l_j is *def-clear* with respect to variable v if v is not given another value on any of the nodes or edges in the path
- **Reach** : If there is a def-clear path from l_i to l_j with respect to v, the def of v at l_i reaches the use at l_j

- **du-path** : A simple subpath that is def-clear with respect to v from a def of v to a use of v
- **du** (n_i, n_j, v) – the set of du-paths from n_i to n_j
- **du** (n_i, v) – the set of du-paths that start at n_i

Touring DU-Paths

- A test path *p* ***du-tours*** du-path *d* with respect to *v* if *p* tours *d* and the subpath taken is def-clear with respect to *v*
- **Sidetrips** can be used, just as with previous touring
- Three criteria
  - Use every def
  - Get to every use
  - Follow all du-paths

Data Flow Test Criteria

- First, we make sure **every def** reaches **a use**

> <u>All-defs coverage (ADC)</u> : For each set of du-paths $S = du(n, v)$, TR contains at least one path d in S.

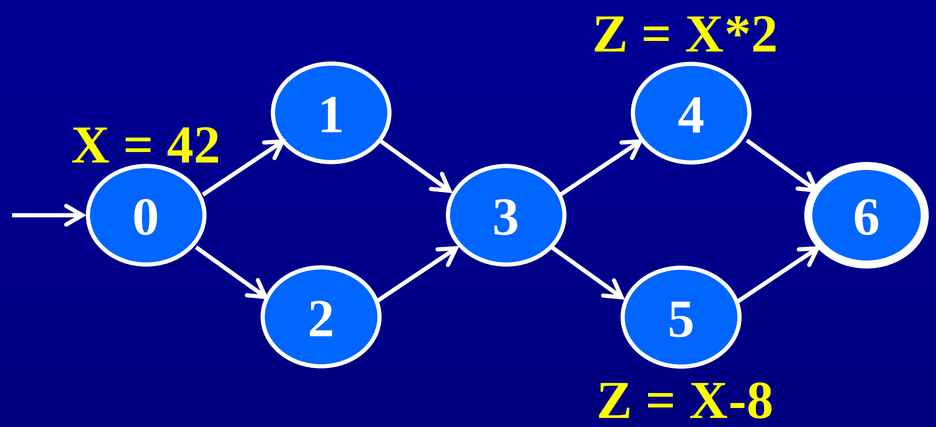
- Then we make sure that **every def** reaches **all** possible **uses**

> <u>All-uses coverage (AUC)</u> : For each set of du-paths to uses $S = du(n_i, n_j, v)$, TR contains at least one path d in S.

- Finally, we cover **all the paths** between defs and uses

> <u>All-du-paths coverage (ADUPC)</u> : For each set $S = du(ni, nj, v)$, TR contains every path in S.

Data Flow Testing Example

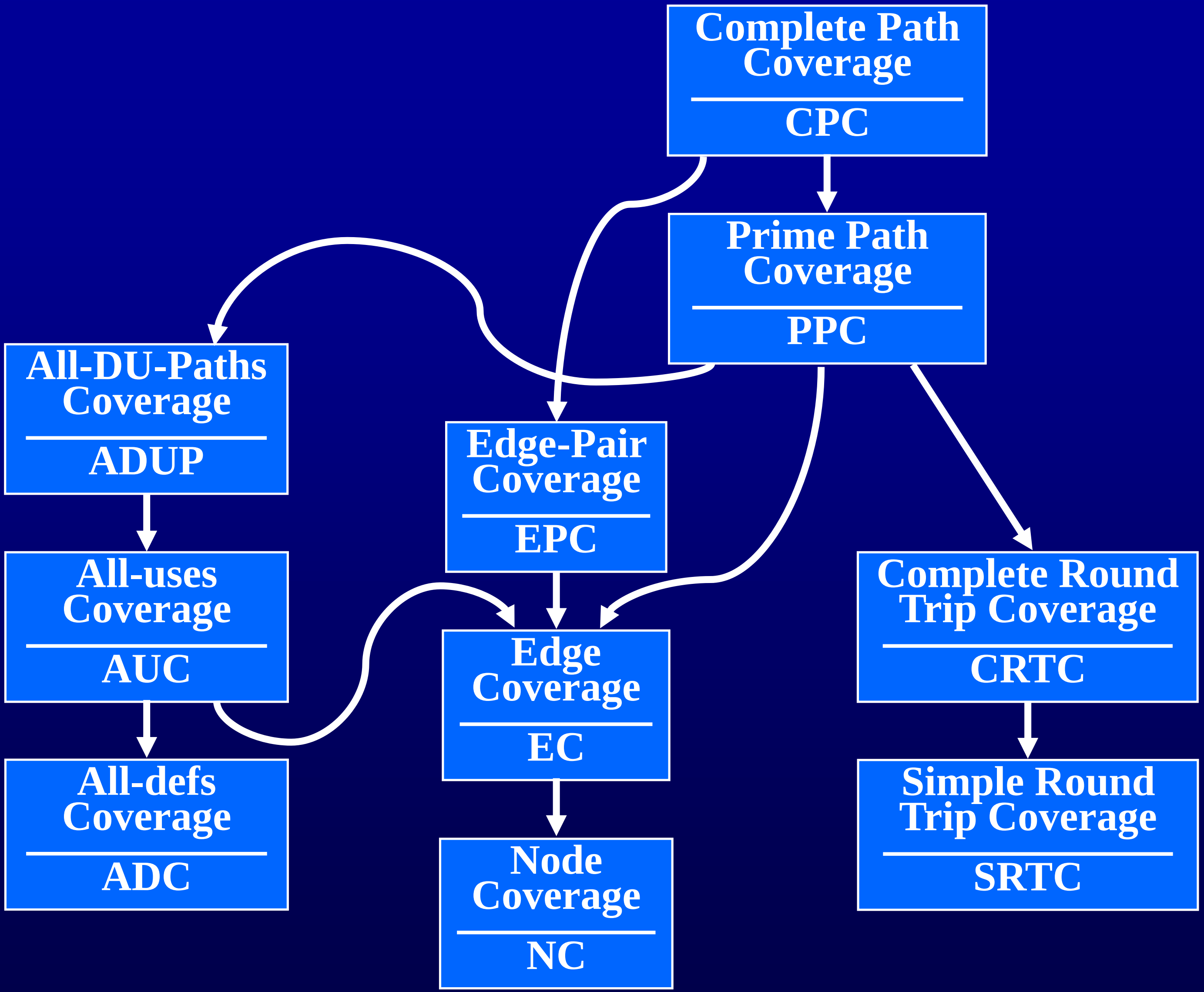


All-defs for *X*
[0, 1, 3, 4]

All-uses for *X*
[0, 1, 3, 4]
[0, 1, 3, 5]

All-du-paths for *X*
[0, 1, 3, 4]
[0, 2, 3, 4]
[0, 1, 3, 5]
[0, 2, 3, 5]

Graph Coverage Criteria Subsumption



Introduction to Software Testing
Chapter 2.3
Graph Coverage for Source Code

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

Overview

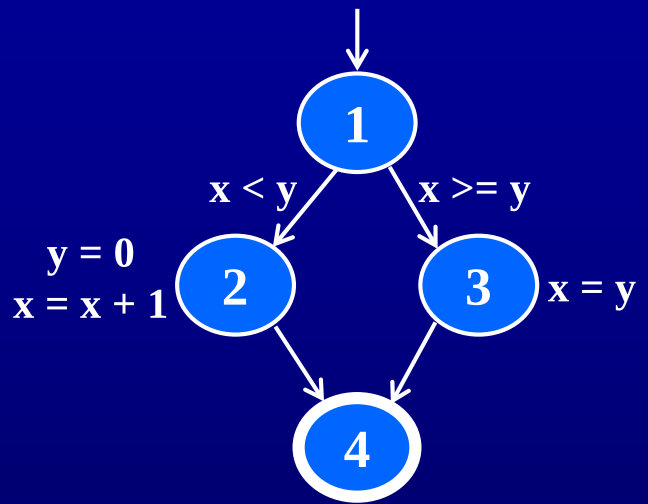
- The most common application of graph criteria is to program source
- Graph : Usually the control flow graph (CFG)
- Node coverage : Execute every statement
- Edge coverage : Execute every branch
- Loops : Looping structures such as for loops, while loops, etc.
- Data flow coverage : Augment the CFG
  - defs are statements that assign values to variables
  - uses are statements that use variables

Control Flow Graphs

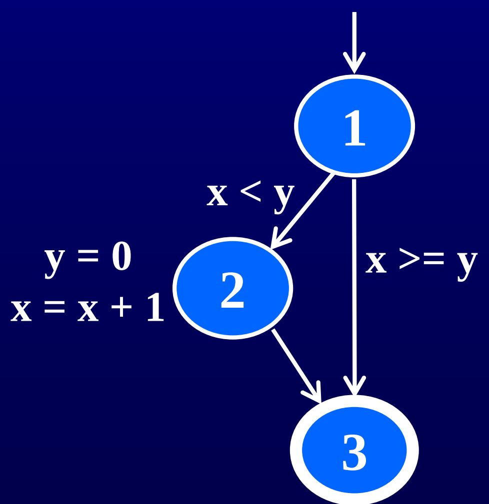
- **A CFG models all executions of a method by describing control structures**
- **Nodes : Statements or sequences of statements (basic blocks)**
  - **Basic Block : A sequence of statements such that if the first statement is executed, all statements will be (no branches)**
- **Edges : Transfers of control**
- **CFGs are sometimes annotated with extra information**
  - **branch predicates**
  - **defs**
  - **uses**
- **Rules for translating statements into graphs …**

CFG : The *if* Statement

```
if (x < y) {
   y = 0;
   x = x + 1;
}
else {
   x = y;
}
```

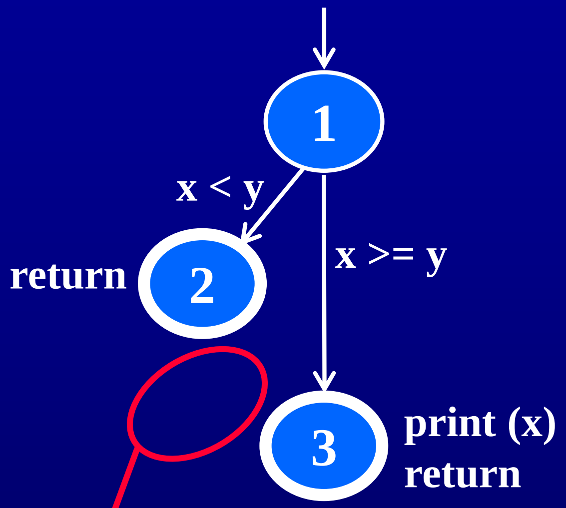



```
if (x < y) {
   y = 0;
   x = x + 1;
}
```



CFG : The *if-return* Statement

```
if (x < y) {
   return;
}
print (x);
return;
```



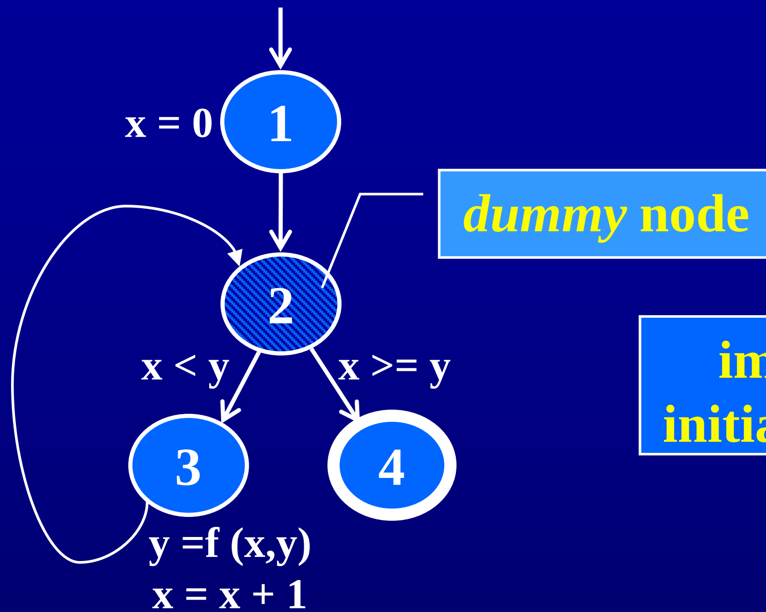
No edge from node 2 to 3.
The return nodes must be distinct.

Loops

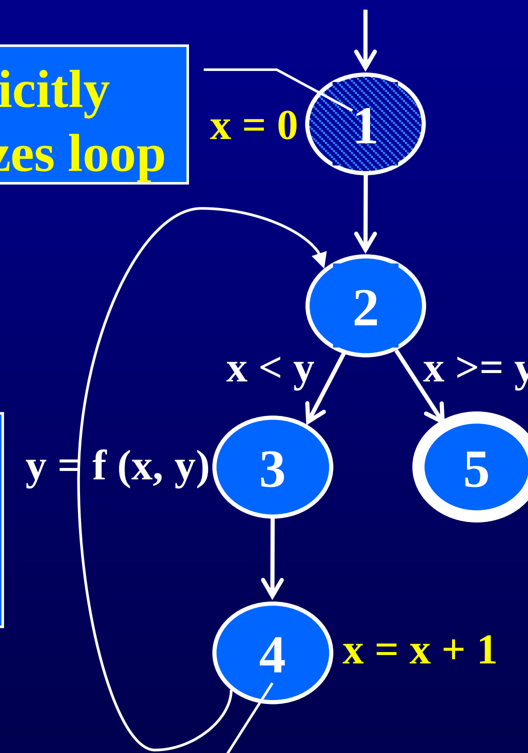
- **Loops require "*extra*" nodes to be added**
- **Nodes that <u>do not</u> represent statements or basic blocks**

CFG : *while* and *for* Loops

```
x = 0;
while (x < y) {
   y = f (x, y);
   x = x + 1;
}
```

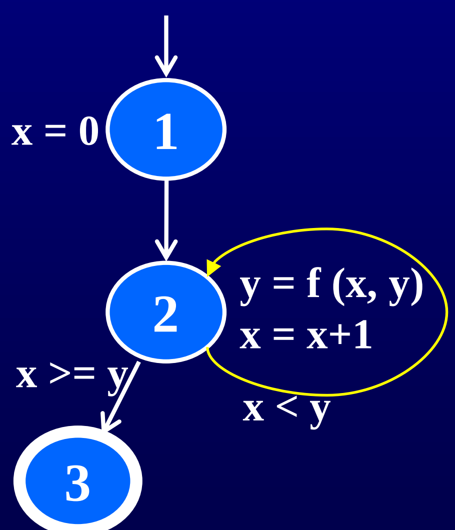



```
for (x = 0; x < y; x++) {
   y = f (x, y);
}
```

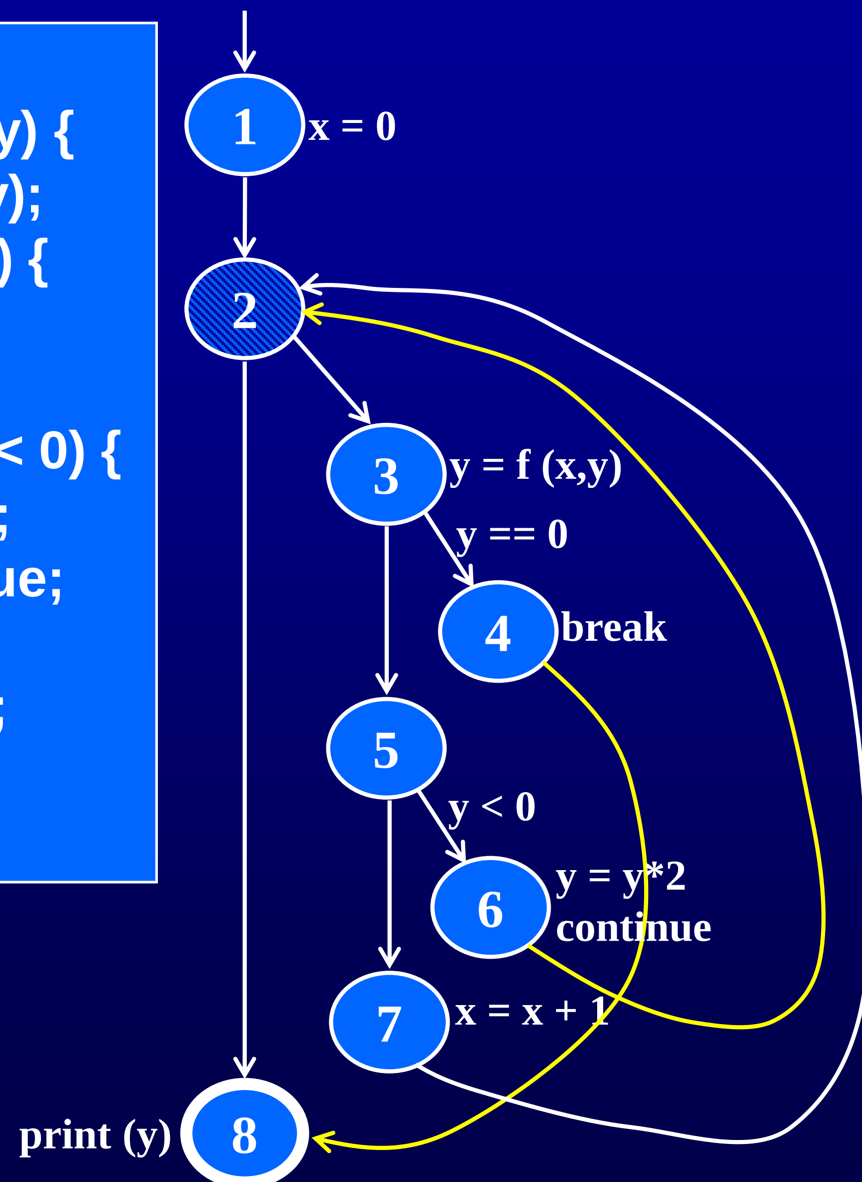


CFG : *do* Loop, *break* and *continue*

```
x = 0;
do {
  y = f (x, y);
  x = x + 1;
} while (x < y);
println (y)
```



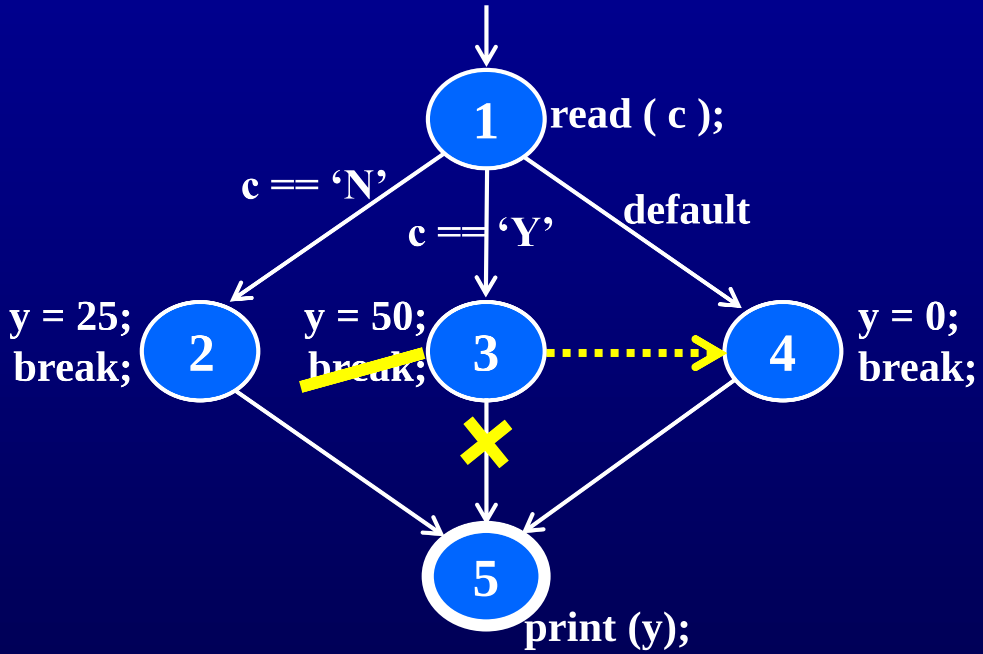

```
x = 0;
while (x < y) {
  y = f (x, y);
  if (y == 0) {
    break;
  }
  else if (y < 0) {
    y = y*2;
    continue;
  }
  x = x + 1;
}
print (y);
```



CFG : The Case (*switch*) Structure

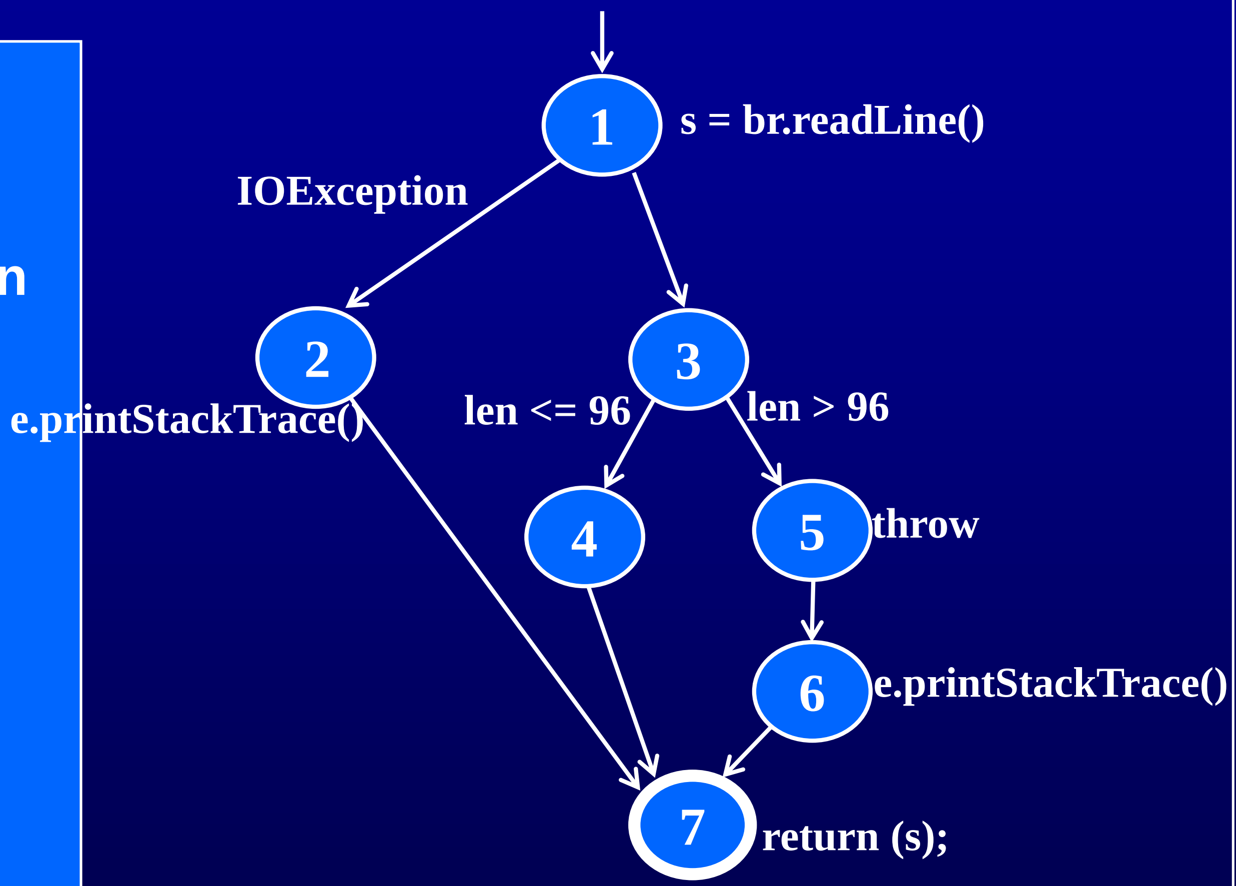
```
read ( c) ;
switch ( c ) {
  case 'N':
    y = 25;
    break;
  case 'Y':
    y = 50;
    break;
  default:
    y = 0;
    break;
}
print (y);
```

If a **break** is omitted …



CFG : The Exception (*try-catch*) Structure

```
try {
  s = br.readLine();
  if (s.length() > 96)
    throw new Exception
      ("too long");
}
catch IOException e) {
  e.printStackTrace();
}
catch Exception e) {
  e.printStackTrace();
}
return (s);
```



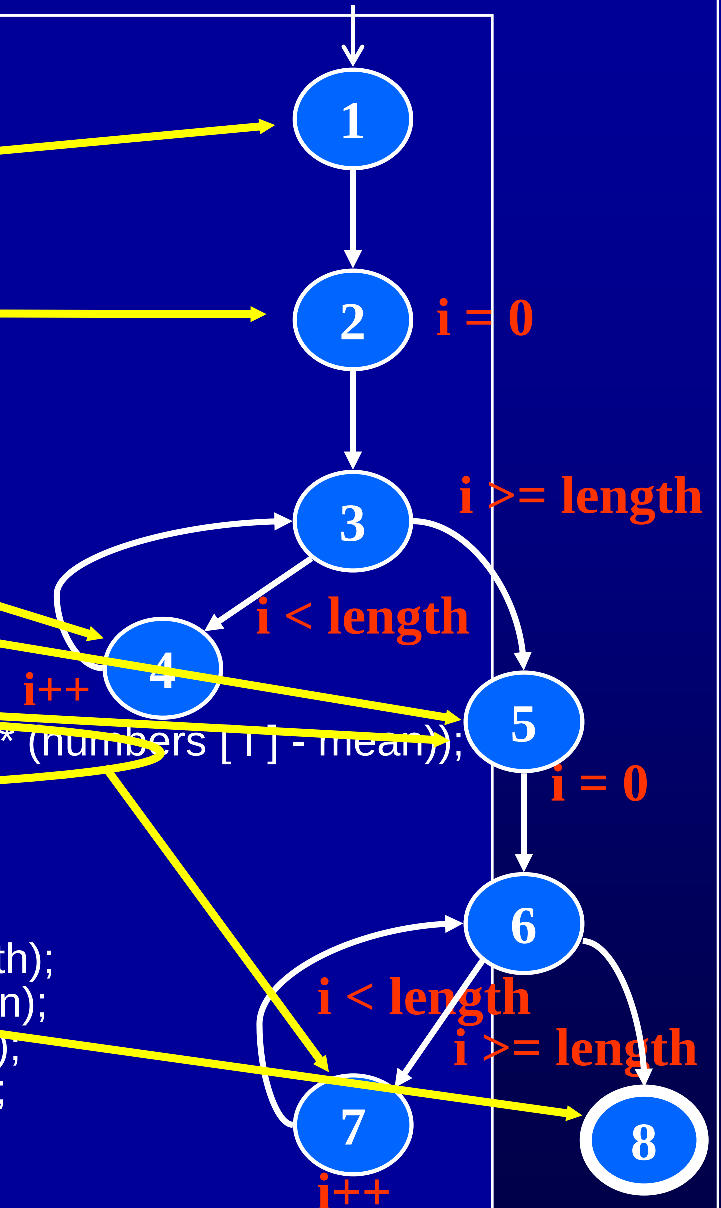
Example Control Flow – Stats

```
public static void computeStats (int [ ] numbers) {
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)   {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)  {
        varsum = varsum  + ((numbers [ I ] - mean) * (numbers [ I ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                   " + length);
    System.out.println ("mean:                     " + mean);
    System.out.println ("median:                   " + med);
    System.out.println ("variance:                 " + var);
    System.out.println ("standard deviation: " + sd);
}
```

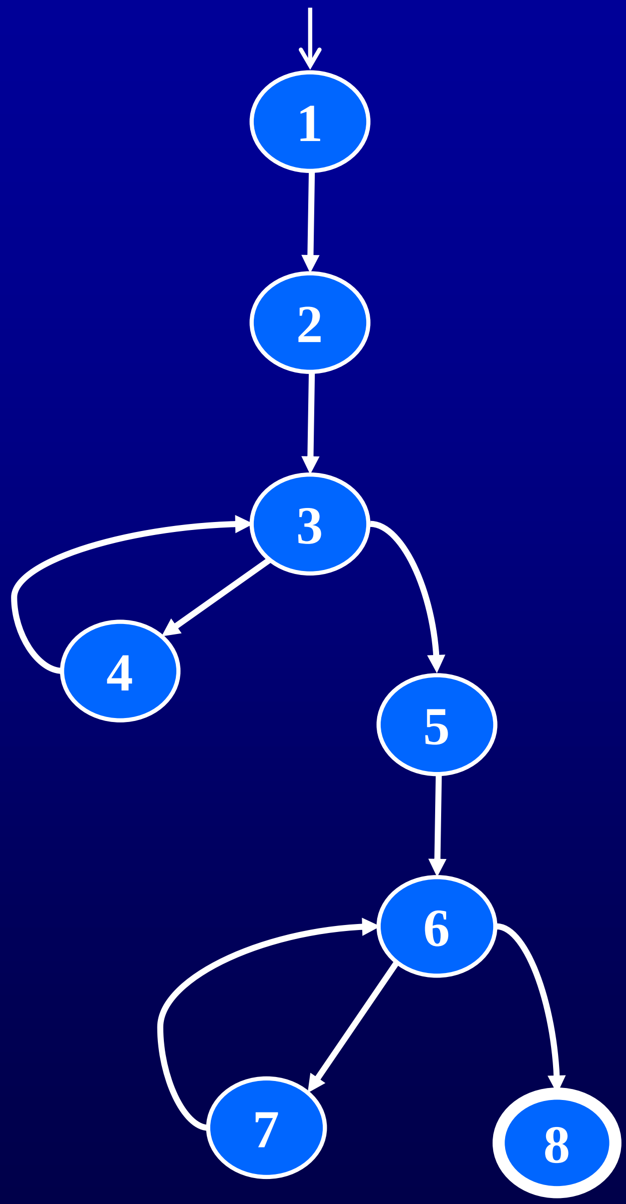

© Ammann & Offutt

Control Flow Graph for Stats

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ I ] - mean) * (numbers [ I ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                    " + length);
    System.out.println ("mean:                      " + mean);
    System.out.println ("median:                    " + med);
    System.out.println ("variance:                  " + var);
    System.out.println ("standard deviation: " + sd);
}
```

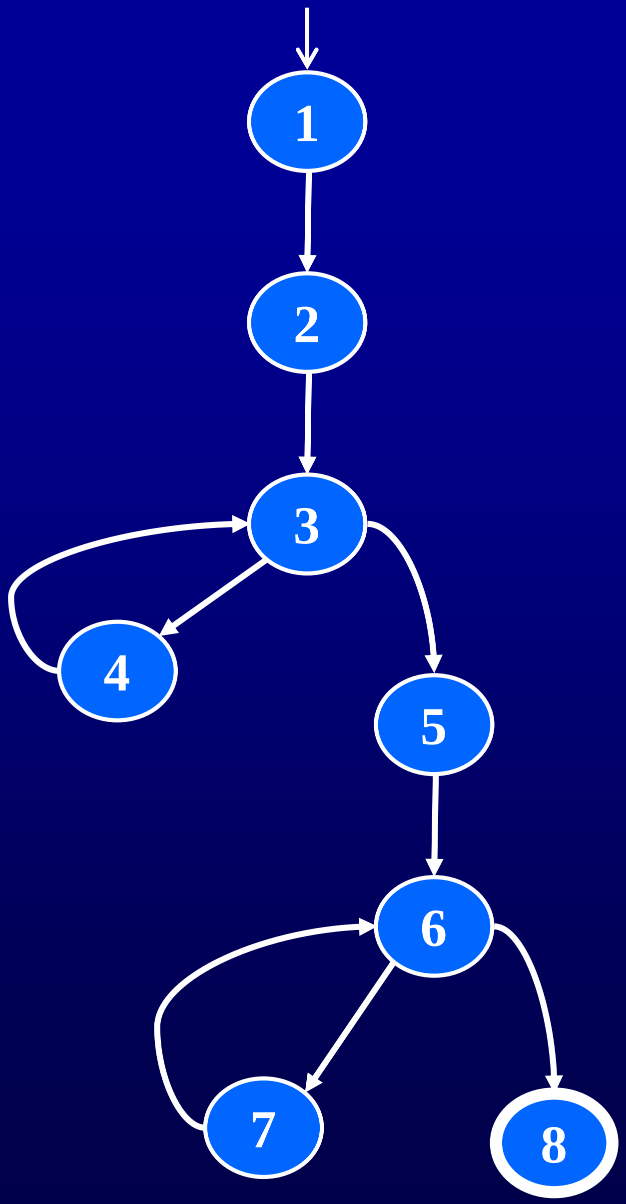


Control Flow TRs and Test Paths – EC



Edge Coverage	
TR	**Test Path**
A. [1, 2]	[1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [2, 3]	
C. [3, 4]	
D. [3, 5]	
E. [4, 3]	
F. [5, 6]	
G. [6, 7]	
H. [6, 8]	
I. [7, 6]	

Control Flow TRs and Test Paths – EPC



Edge-Pair Coverage

TR

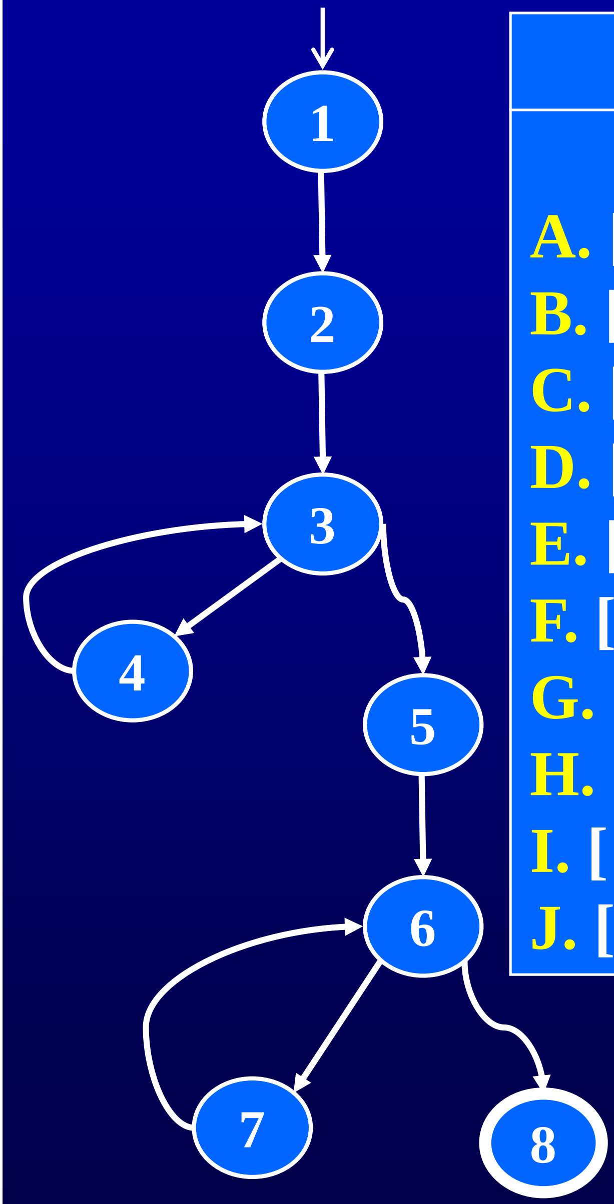
A. [1, 2, 3]
B. [2, 3, 4]
C. [2, 3, 5]
D. [3, 4, 3]
E. [3, 5, 6]
F. [4, 3, 5]
G. [5, 6, 7]
H. [5, 6, 8]
I. [6, 7, 6]
J. [7, 6, 8]
K. [4, 3, 4]
L. [7, 6, 7]

Test Paths

i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
ii. [1, 2, 3, 5, 6, 8]
iii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]

TP	TRs toured	*sidetrips*
i	A, B, D, E, F, G, I, J	C, H
ii	A, C, E, H	
iii	A, B, D, E, F, G, I, J, K, L	C, H

Control Flow TRs and Test Paths – PPC



Prime Path Coverage

TR	Test Paths
A. [3, 4, 3]	i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [4, 3, 4]	ii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]
C. [7, 6, 7]	iii. [1, 2, 3, 4, 3, 5, 6, 8]
D. [7, 6, 8]	iv. [1, 2, 3, 5, 6, 7, 6, 8]
E. [6, 7, 6]	v. [1, 2, 3, 5, 6, 8]
F. [1, 2, 3, 4]	
G. [4, 3, 5, 6, 7]	
H. [4, 3, 5, 6, 8]	
I. [1, 2, 3, 5, 6, 7]	
J. [1, 2, 3, 5, 6, 8]	

TP	TRs toured	*sidetrips*
i	A, D, E, F, G	H, I, J
ii	A, B, C, D, E, F, G,	H, I, J
iii	A, F, H	J
iv	D, E, F, I	J
v	J	

Data Flow Coverage for Source

def : a location where a value is stored into memory

- x appears on the left side of an assignment (`x = 44;`)
- x is an actual parameter in a call and the method changes its value
- x is a formal parameter of a method (implicit def when method starts)
- x is an input to a program

Data Flow Coverage for Source

use : a location where variable's value is accessed

- x appears on the right side of an assignment
- x appears in a conditional test
- x is an actual parameter to a method
- x is an output of the program
- x is an output of a method in a return statement

If a def and a use appear on the same node, then it is only a DU-pair if the def occurs after the use and the node is in a loop

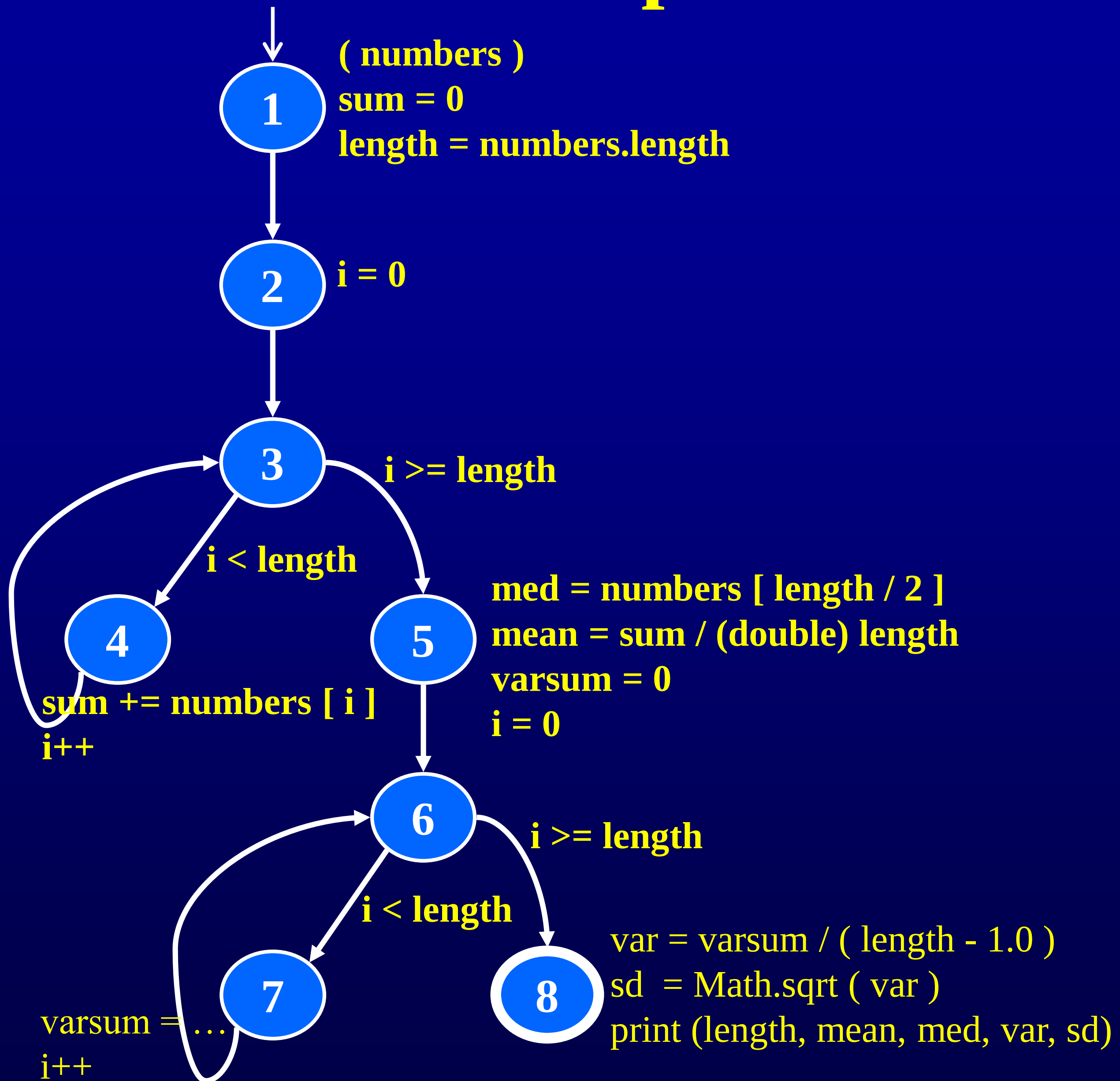
```
while (x > 0) {
    z = x+1;
    x = 2*z;
}
```

du-pair (n1,n1,x) ?

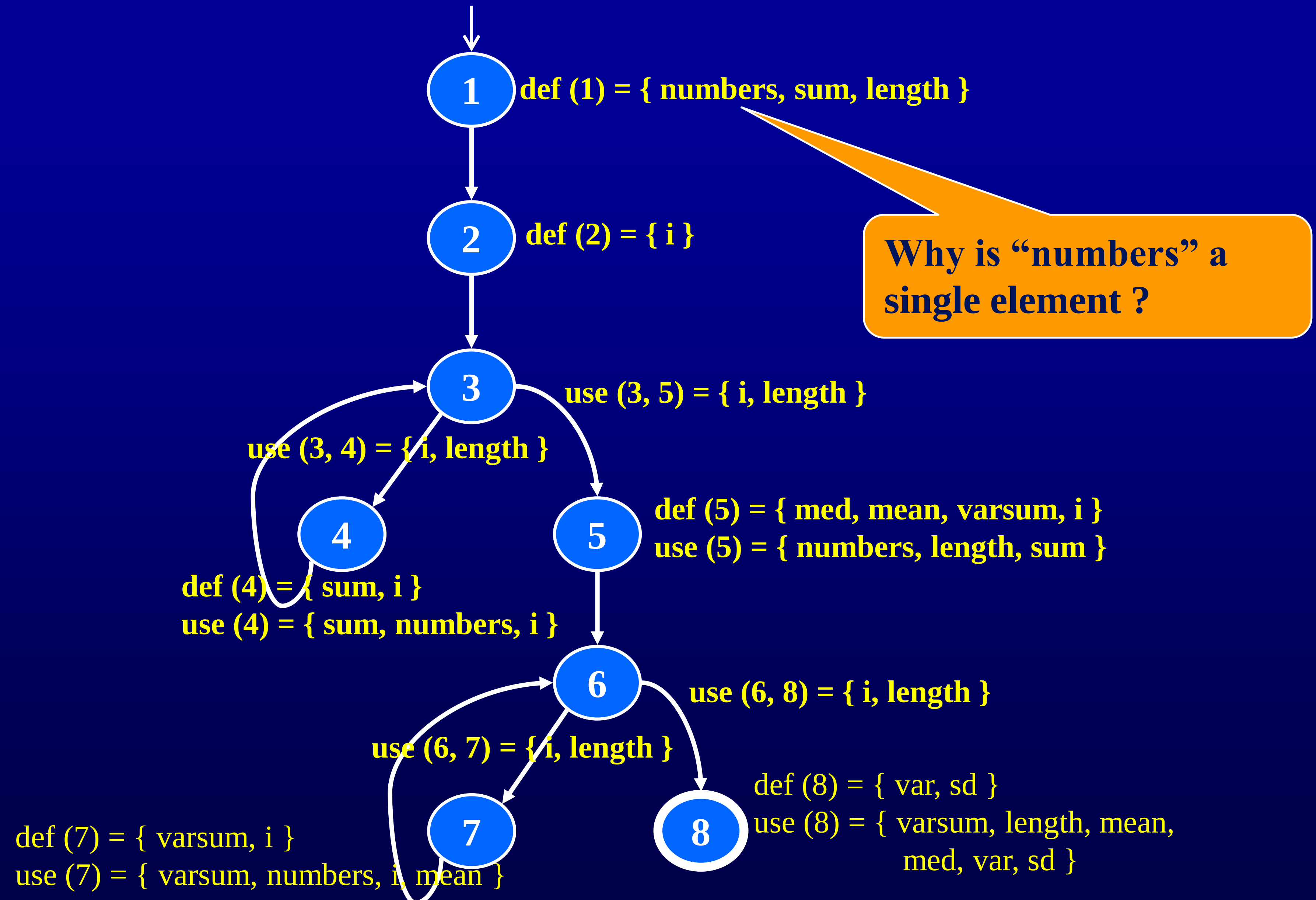
Example Data Flow – Stats

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0.0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0.o;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                    " + length);
    System.out.println ("mean:                      " + mean);
    System.out.println ("median:                    " + med);
    System.out.println ("variance:                  " + var);
    System.out.println ("standard deviation: " + sd);
}
```

Control Flow Graph for Stats



CFG for Stats – With Defs & Uses



A question of def-use analysis

Why is it difficult to analyze def-use pairs for programs with pointer variables ?

Defs and Uses Tables for Stats

Node	Def	Use
1	{ numbers, sum, length }	{ numbers }
2	{ i }	
3		
4	{ sum, i }	{ numbers, i, sum }
5	{ med, mean, varsum, i }	{ numbers, length, sum }
6		
7	{ varsum, i }	{ varsum, numbers, i, mean }
8	{ var, sd }	{ varsum, length, var, mean, med, var, sd }

Edge	Use
(1, 2)	
(2, 3)	
(3, 4)	{ i, length }
(4, 3)	
(3, 5)	{ i, length }
(5, 6)	
(6, 7)	{ i, length }
(7, 6)	
(6, 8)	{ i, length }

DU Pairs for Stats

variable	DU Pairs
numbers	(1, 4) (1, 5) (1, 7)
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))
med	(5, 8)
var	(8, 8)
sd	(8, 8)
mean	(5, 7) (5, 8)
sum	(1, 4) (1, 5) (4, 4) (4, 5)
varsum	(5, 7) (5, 8) (7, 7) (7, 8)
i	(2, 4) (2, (3,4)) (2, (3,5)) ~~(2, 7) (2, (6,7)) (2, (6,8))~~ (4, 4) (4, (3,4)) (4, (3,5)) ~~(4, 7) (4, (6,7)) (4, (6,8))~~ (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))

defs come before uses, do not count as DU pairs

defs after use in loop, these are valid DU pairs

No def-clear path … different scope for i

No path through graph from nodes 5 and 7 to 4 or 3

DU Paths for Stats

variable	DU Pairs	DU Paths
numbers	(1, 4) (1, 5) (1, 7)	[1, 2, 3, 4] [1, 2, 3, 5] [1, 2, 3, 5, 6, 7]
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))	[1, 2, 3, 5] [1, 2, 3, 5, 6, 8] [1, 2, 3, 4] [1, 2, 3, 5] [1, 2, 3, 5, 6, 7] [1, 2, 3, 5, 6, 8]
med	(5, 8)	[5, 6, 8]
var	(8, 8)	*No path needed*
sd	(8, 8)	*No path needed*
sum	(1, 4) (1, 5) (4, 4) (4, 5)	[1, 2, 3, 4] [1, 2, 3, 5] [4, 3, 4] [4, 3, 5]

variable	DU Pairs	DU Paths
mean	(5, 7) (5, 8)	[5, 6, 7] [5, 6, 8]
varsum	(5, 7) (5, 8) (7, 7) (7, 8)	[5, 6, 7] [5, 6, 8] [7, 6, 7] [7, 6, 8]
i	(2, 4) (2, (3,4)) (2, (3,5)) (4, 4) (4, (3,4)) (4, (3,5)) (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))	[2, 3, 4] [2, 3, 4] [2, 3, 5] [4, 3, 4] [4, 3, 4] [4, 3, 5] [5, 6, 7] [5, 6, 7] [5, 6, 8] [7, 6, 7] [7, 6, 7] [7, 6, 8]

DU Paths for Stats – No Duplicates

There are 38 DU paths for Stats, but only 12 unique

[1, 2, 3, 4]	[4, 3, 4]
[1, 2, 3, 5]	[4, 3, 5]
[1, 2, 3, 5, 6, 7]	[5, 6, 7]
[1, 2, 3, 5, 6, 8]	[5, 6, 8]
[2, 3, 4]	[7, 6, 7]
[2, 3, 5]	[7, 6, 8]

4 expect a loop not to be "entered"

6 require at least one iteration of a loop

2 require at least <u>two</u> iterations of a loop

Test Cases and Test Paths

Test Case : numbers = (44) ; length = 1

Test Path : [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]

Additional DU Paths covered (no sidetrips)

[1, 2, 3, 4]   [2, 3, 4]   [4, 3, 5]   [5, 6, 7]   [7, 6, 8]

The five stars that require at least one iteration of a loop

Test Case : numbers = (2, 10, 15) ; length = 3

Test Path : [1, 2, 3, 4, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 7, 6, 8]

DU Paths covered (no sidetrips)

[4, 3, 4]   [7, 6, 7]

The two stars that require at least two iterations of a loop

Other DU paths require arrays with length 0 to skip loops

But the method fails with index out of bounds exception…

med = numbers [length / 2];

A fault was found

Summary

- Applying the graph test criteria to control flow graphs is relatively straightforward
  - Most of the developmental research work was done with CFGs
- A few subtle decisions must be made to translate control structures into the graph
- Some tools will assign each statement to a unique node
  - These slides and the book uses basic blocks
  - Coverage is the same, although the bookkeeping will differ

Introduction to Software Testing
Chapter 2.4
Graph Coverage for Design Elements

Paul Ammann & Jeff Offutt

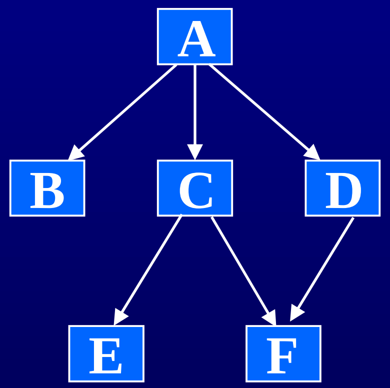
http://www.cs.gmu.edu/~offutt/softwaretest/

OO Software and Designs

- Emphasis on modularity and reuse puts complexity in the design connections
- Testing design relationships is more important than before
- Graphs are based on the connections among the software components
  - Connections are dependency relations, also called couplings

Call Graph

- The most common graph for structural design testing
- <u>Nodes</u> : Units (in Java – methods)
- <u>Edges</u> : Calls to units



Example call graph

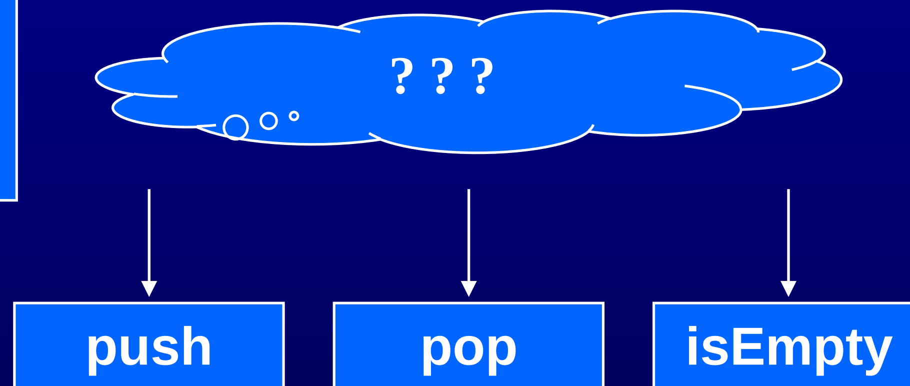
<u>Node coverage</u> : call every unit at least once (method coverage)

<u>Edge coverage</u> : execute every call at least once (call coverage)

Call Graphs on Classes

- **Node and edge coverage of class call graphs often do not work very well**
- **Individual methods might not call each other at all!**

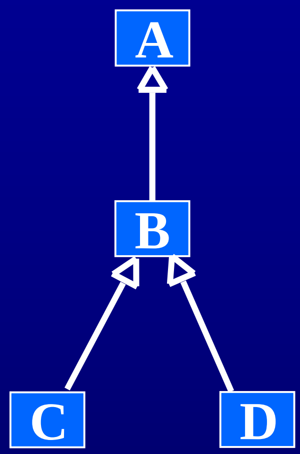
<u>Class stack</u>
public void push (Object o)
public Object pop ( )
public boolean isEmpty (Object o)



Other types of testing are needed – do <u>not</u> use graph criteria

Inheritance & Polymorphism

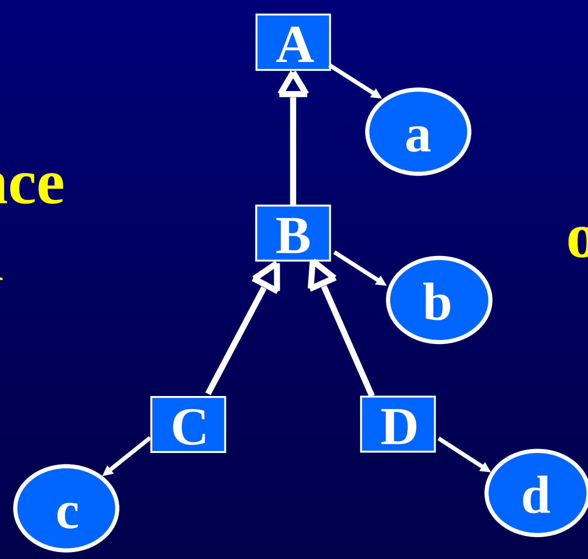
Caution : Ideas are preliminary and not widely used



Example inheritance hierarchy graph

Classes are not executable, so this graph is not directly testable

We need objects



objects

What is coverage on this graph ?

Coverage on Inheritance Graph

- Create an object for each class ?
  - This seems weak because there is no execution
- Create an object for each class and apply call coverage?

OO Call Coverage : TR contains each reachable node in the call graph of an object instantiated for each class in the class hierarchy.

OO Object Call Coverage : TR contains each reachable node in the call graph of every object instantiated for each class in the class hierarchy.

- Data flow is probably more appropriate …

Data Flow at the Design Level

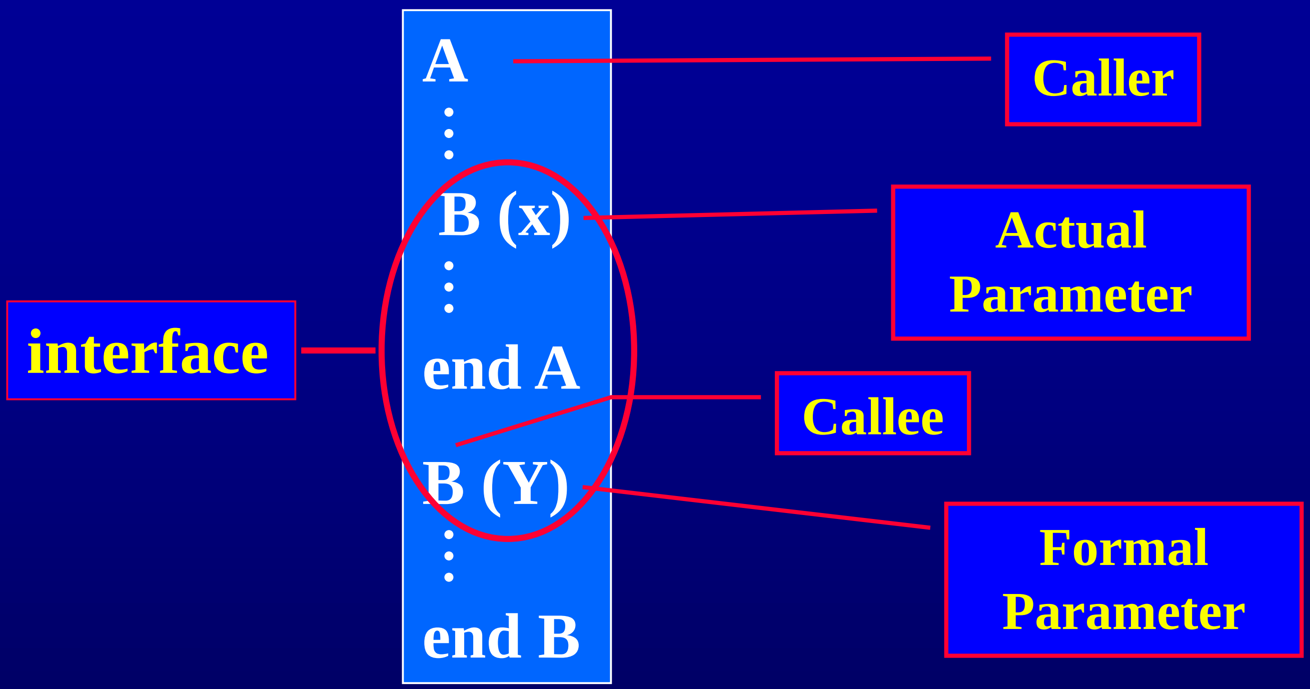
- **Data flow couplings among units and classes are <u>more complicated</u> than control flow couplings**
  - **When values are passed, they "<u>change names</u>"**
  - **Many different ways to share data**
  - **Finding defs and uses can be difficult – finding which uses a def can reach is very difficult**

Data Flow at the Design Level

- **When software gets complicated … testers should get interested**
  - **That's where the faults are!**

- **Caller** : A unit that invokes another unit
- **Callee** : The unit that is called
- **Callsite** : Statement or node where the call appears
- **Actual parameter** : Variable in the caller
- **Formal parameter** : Variable in the callee

Example Call Site



- Applying data flow criteria to def-use pairs between units is <u>too expensive</u>
- Too many possibilities
- But this is integration testing, and we really only care about the <u>interface</u> …

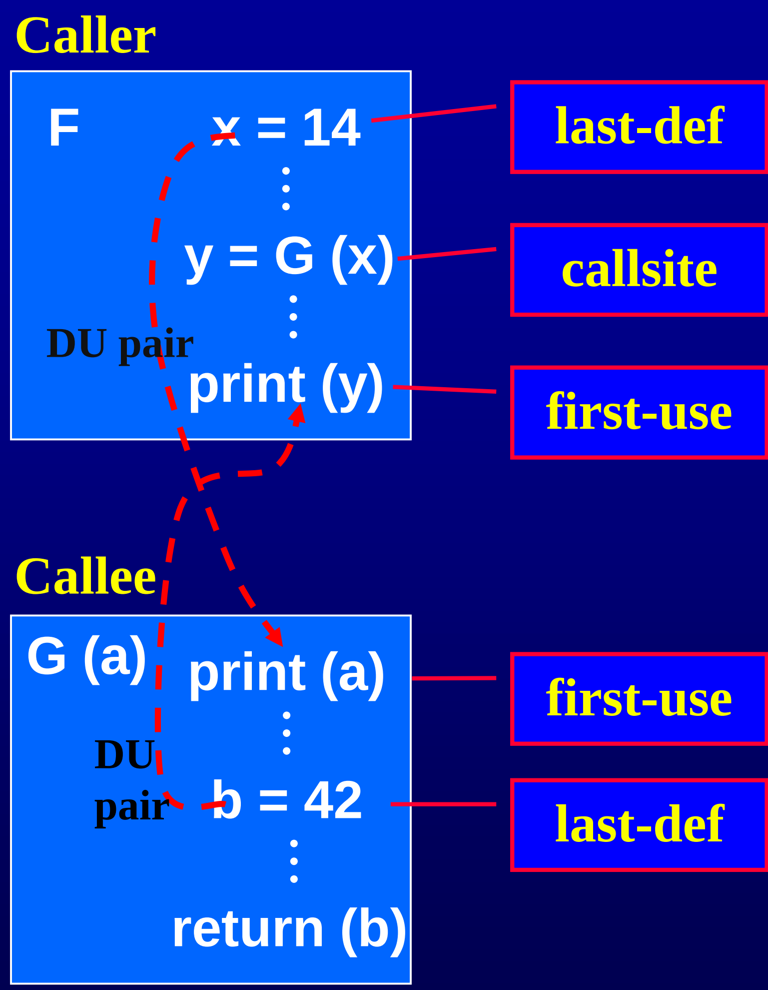
Inter-procedural DU Pairs

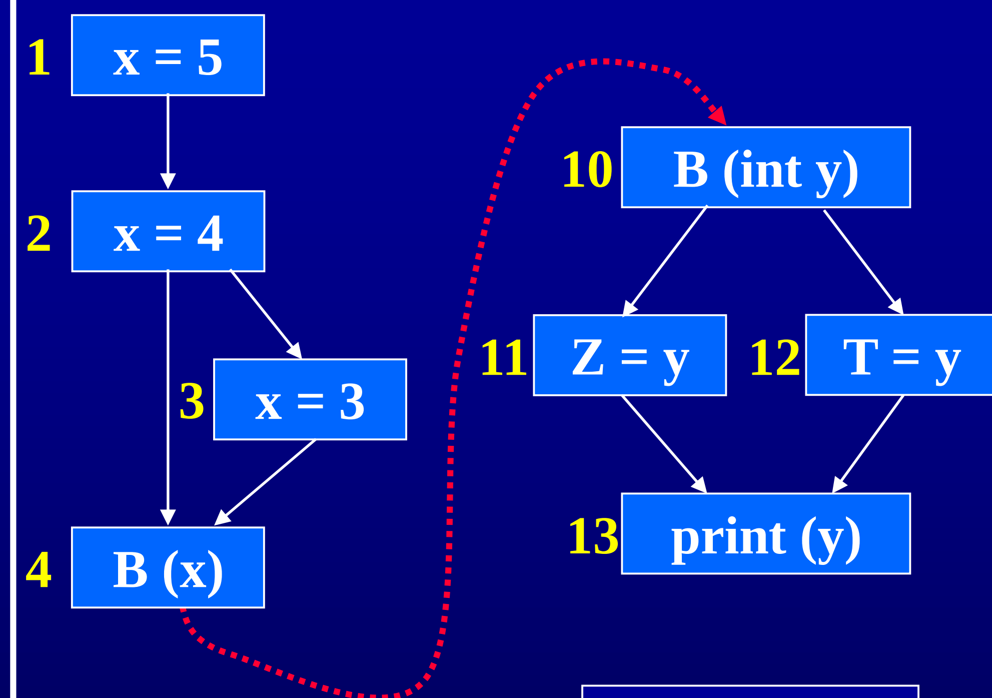
- If we focus on the interface, then we just need to consider the <u>last definitions</u> of variables before calls and returns and <u>first uses</u> inside units and after calls

- <u>Last-def</u> : The set of nodes that define a variable *x* and has a def-clear path from the node through a callsite to a use in the other unit
  - Can be from caller to callee (parameter or shared variable) or from callee to caller as a return value

- <u>First-use</u> : The set of nodes that have uses of a variable *y* and for which there is a def-clear and use-clear path from the callsite to the nodes

Example Inter-procedural DU Pairs





Last Defs
2, 3

First Uses
11, 12

Example – Quadratic

```
// Program to compute the quadratic root for two numbers
import java.lang.Math;

class Quadratic
{
 private static float Root1, Root2;

 public static void main (String[] argv)
 {
      int X, Y, Z;
     boolean ok;
     int controlFlag = Integer.parseInt (argv[0]);
     if (controlFlag == 1)
     {
          X = Integer.parseInt (argv[1]);
          Y = Integer.parseInt (argv[2]);
          Z = Integer.parseInt (argv[3]);
      }
     else
     {
           X = 10;
           Y = 9;
           Z = 12;
      }
           ok = Root (X, Y, Z);
           if (ok)
            System.out.println
                 ("Quadratic: " + Root1 + Root2);
           else
              System.out.println ("No Solution.");
 }

// Three positive integers, finds quadratic root
 private static boolean Root (int A, int B, int C)
 {
     float D;
     boolean Result;
     D = (float) Math.pow ((double)B,
            (double2-4.0)*A*C );
     if (D < 0.0)
     {
         Result = false;
         return (Result);
     }
     Root1 = (float) ((-B + Math.sqrt(D))/(2.0*A));
     Root2 = (float) ((-B – Math.sqrt(D))/(2.0*A));
     Result = true;
     return (Result);
 } / /End method Root

 } // End class Quadratic
```

```
// Program to compute the quadratic root for two numbers
import java.lang.Math;

class Quadratic
{
 private static float Root1, Root2;

 public static void main (String[] argv)
 {
     int X, Y, Z;
     boolean ok;
     int controlFlag = Integer.parseInt (argv[0]);
     if (controlFlag == 1)
     {
        X = Integer.parseInt (argv[1]);
        Y = Integer.parseInt (argv[2]);
        Z = Integer.parseInt (argv[3]);
     }
     else
     {
        X = 10;
        Y = 9;
        Z = 12;
     }
```

shared variables

last-defs

first-use

first-use

last-def

last-defs

```
            ok = Root (X, Y, Z);
            if (ok)
              System.out.println
                  ("Quadratic: " + Root1 + Root2);
            else
               System.out.println ("No Solution.");
      }

      // Three positive integers, finds the quadratic root
      private static boolean Root (int A, int B, int C)
      {
         float D;
         boolean Result;
         D = (float) Math.pow ((double)B, (double2-4.0)*A*C);
         if (D < 0.0)
         {
            Result = false;

            return (Result);
         }
         Root1 = (float) ((-B + Math.sqrt(D)) / (2.0*A));
         Root2 = (float) ((-B – Math.sqrt(D)) / (2.0*A));
         Result = true;
         return (Result);
      } / /End method Root

   } // End class Quadratic
```

Quadratic – Coupling DU-pairs

Pairs of locations: <u>method</u> name, <u>variable</u> name, <u>statement</u>

(main (), X, 15) – (Root (), A, 38)

(main (), Y, 16) – (Root (), B, 38)

(main (), Z, 17) – (Root (), C, 38)

(main (), X, 21) – (Root (), A, 38)

(main (), Y, 22) – (Root (), B, 38)

(main (), Z, 23) – (Root (), C, 38)

(Root (), Root1, 44) – (main (), Root1, 28)

(Root (), Root2, 45) – (main (), Root2, 28)

(Root (), Result, 41) – (main (), ok, 26)

(Root (), Result, 46) – (main (), ok, 26)

Coupling Data Flow Notes

- Only variables that are <u>used or defined</u> in the callee
- <u>Implicit initializations</u> of class and global variables
- <u>Transitive</u> DU-pairs are too expensive to handle
  - A calls B, B calls C, and there is a variable defined in A and used in C
- <u>Arrays</u> : a reference to one element is considered to be a reference to all elements

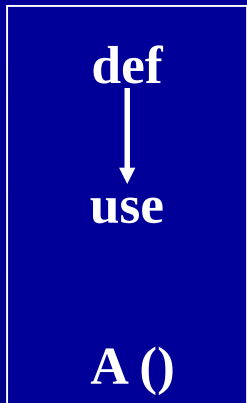
Inheritance, Polymorphism & Dynamic Binding

- Additional control and data connections make data flow analysis more complex
- The defining and using units may be in different call hierarchies
- When inheritance hierarchies are used, a def in one unit could reach uses in any class in the inheritance hierarchy
- With dynamic binding, the same location can reach different uses depending on the current type of the using object
- The same location can have different definitions or uses at different points in the execution !

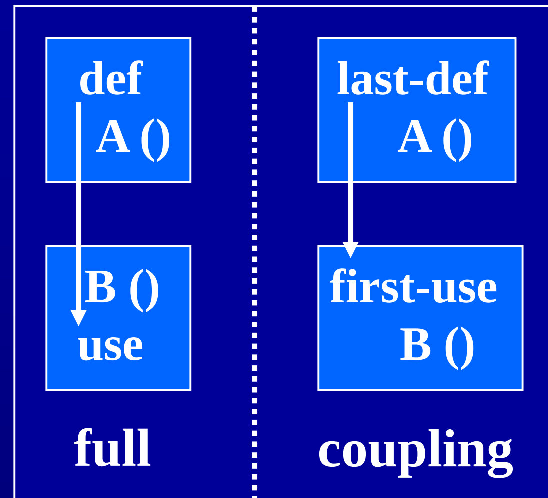
Additional Definitions

- <u>**Inheritance**</u> **: If class B *inherits* from class A, then all variables and methods in A are implicitly in B, and B can add more**
  - **A is the *parent* or *ancestor***
  - **B is the *child* or *descendent***
- **An object reference <u>*obj*</u> that is declared to be of type *A* can be assigned an object of either type *A*, *B*, or any of *B*'s descendents**
  - <u>**Declared type**</u> **: The type used in the declaration: *A obj;***
  - <u>**Actual type**</u> **: The type used in the object assignment: *obj = new B();***
- <u>**Class (State) Variables**</u> **: The variables declared at the class level, often private**

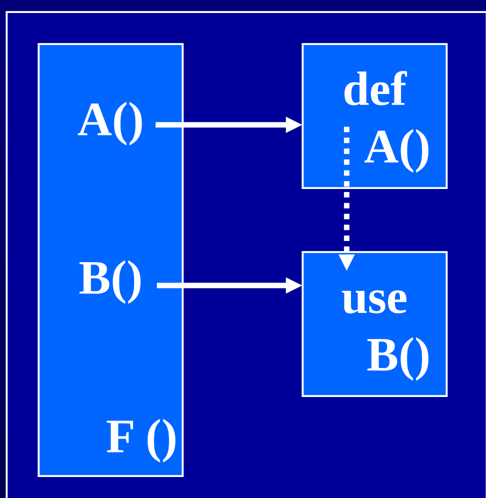
Types of Def-Use Pairs



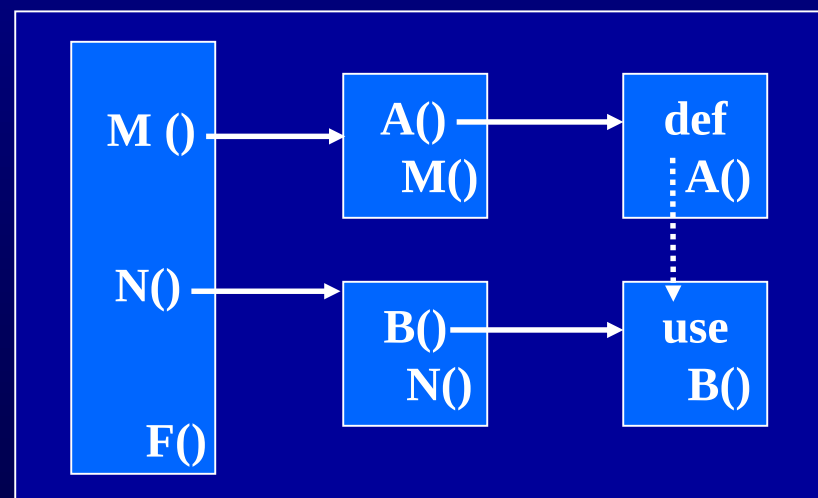
intra-procedural data flow
(within the same unit)



inter-procedural
data flow



object-oriented direct
coupling data flow

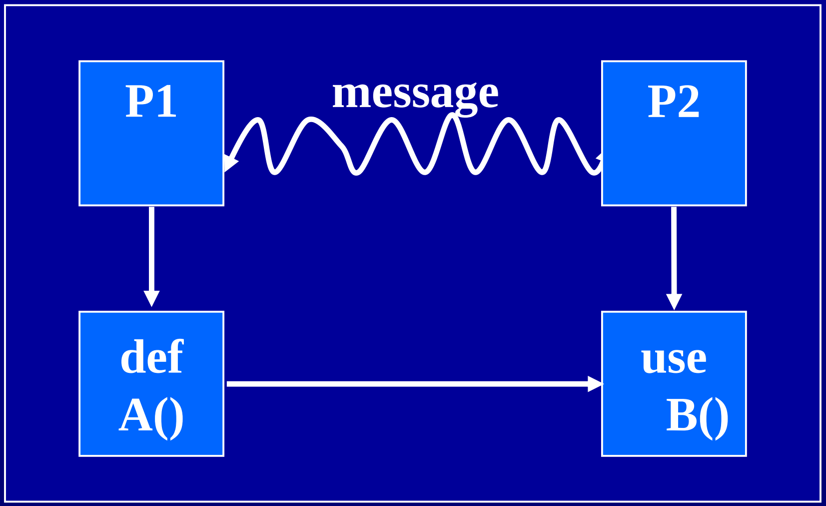


object-oriented indirect
coupling data flow

OO Data Flow Summary

- The defs and uses could be in the same class, or different classes
- Researchers have applied data flow testing to the direct coupling OO situation
  - Has not been used in practice
  - No tools available
- Indirect coupling data flow testing has not been tried either in research or in practice
  - Analysis cost may be prohibitive

Web Applications and Other Distributed Software



distributed software data flow

- "message" could be HTTP, RMI, or other mechanism
- A() and B() could be in the same class or accessing a persistent variable such as in a web session
- Beyond current technologies

Summary—What Works?

- **Call graphs** are common and very useful ways to design integration tests
- **Inter-procedural data flow** is relatively easy to compute and results in effective integration tests
- The ideas for **OO software** and **web applications** are preliminary and have not been used much in practice

Introduction to Software Testing
Chapter 2.5
Graph Coverage for Specifications

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

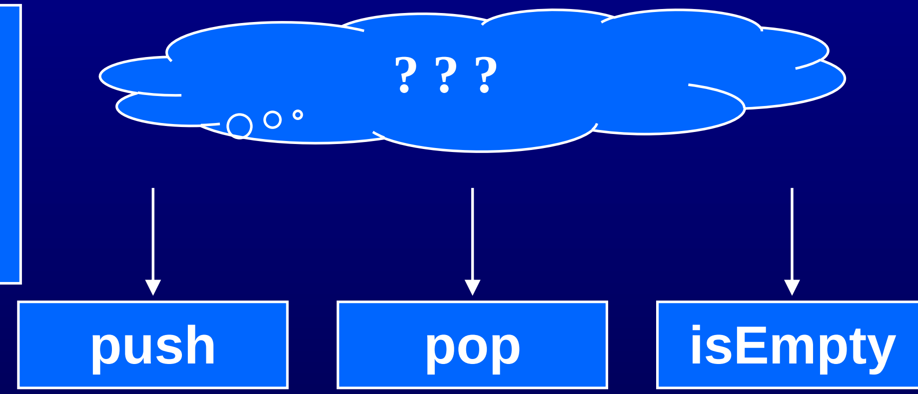
Design Specifications

- A design specification describes aspects of <u>what</u> behavior software should exhibit
- A design specification may or may not reflect the implementation
  - <u>More accurately</u> – the implementation may not exactly reflect the spec
  - Design specifications are often called models of the software
- Two types of descriptions are used in this chapter
  1. Sequencing constraints on class methods
  2. State behavior descriptions of software

Sequencing Constraints

- Sequencing constraints are rules that impose constraints on the order in which methods may be called
- They can be encoded as preconditions or other specifications
- Section 2.4 said that classes often have methods that do not call each other

Class stack
public void push (Object o)
public Object pop ( )
public boolean isEmpty ( )



- Tests can be created for these classes as sequences of method calls
- Sequencing constraints give an easy and effective way to choose which sequences to use

Sequencing Constraints Overview

- Sequencing constraints might be
  - Expressed explicitly
  - Expressed implicitly
  - Not expressed at all
- Testers should derive them if they do not exist
  - Look at existing design documents
  - Look at requirements documents
  - Ask the developers
  - Last choice : Look at the implementation
- If they don't exist, expect to find more faults !
- Share with designers before designing tests
- Sequencing constraints do not capture all behavior

Queue Example

```
public int DeQueue()
{
  // Pre: At least one element must be on the queue.
  … …

public EnQueue (int e)
{
  // Post: e is on the end of the queue.
```

- Sequencing constraints are implicitly embedded in the pre and postconditions
  - EnQueue () must be called before DeQueue ()
- Does not include the requirement that we must have at least as many Enqueue () calls as DeQueue () calls
  - Can be handled by state behavior techniques

File ADT Example

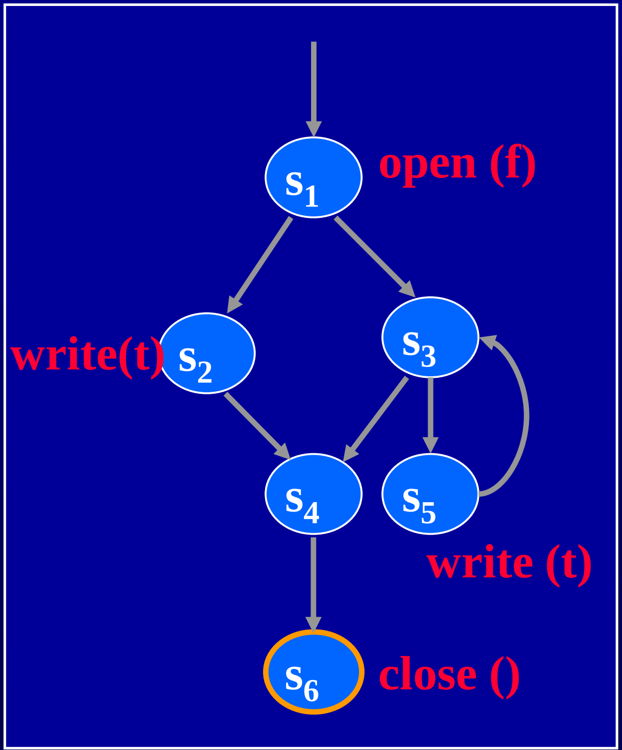
abstract data type

class FileADT has three methods:

- **open (String fName)** // Opens file with name fName
- **close ()** // Closes the file and makes it unavailable
- **write (String textLine)** // Writes a line of text to the file

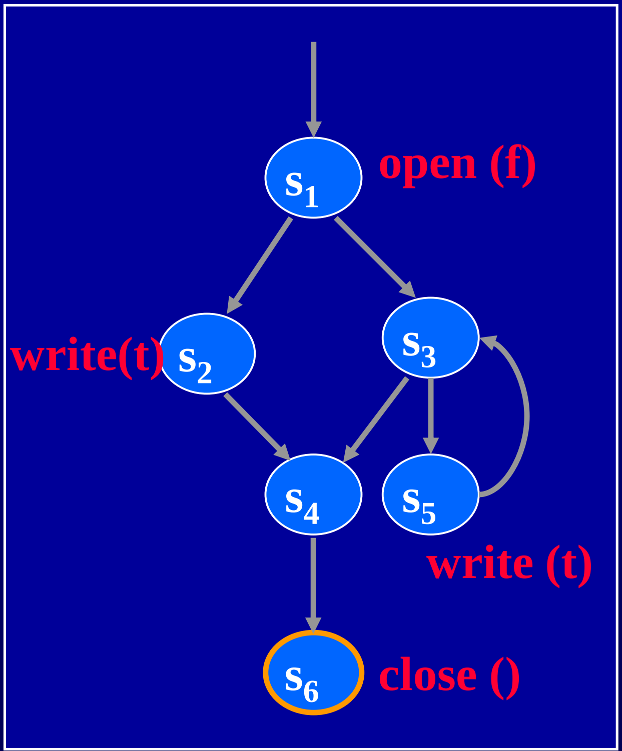
Valid sequencing constraints on FileADT:

1. An open (f) must be executed before every write (t)
2. An open (f) must be executed before every close ()
3. A write (f) may not be executed after a close () unless there is an open (f) in between
4. A write (t) should be executed before every close ()



Static Checking

Is there a path that violates any of the sequencing constraints ?

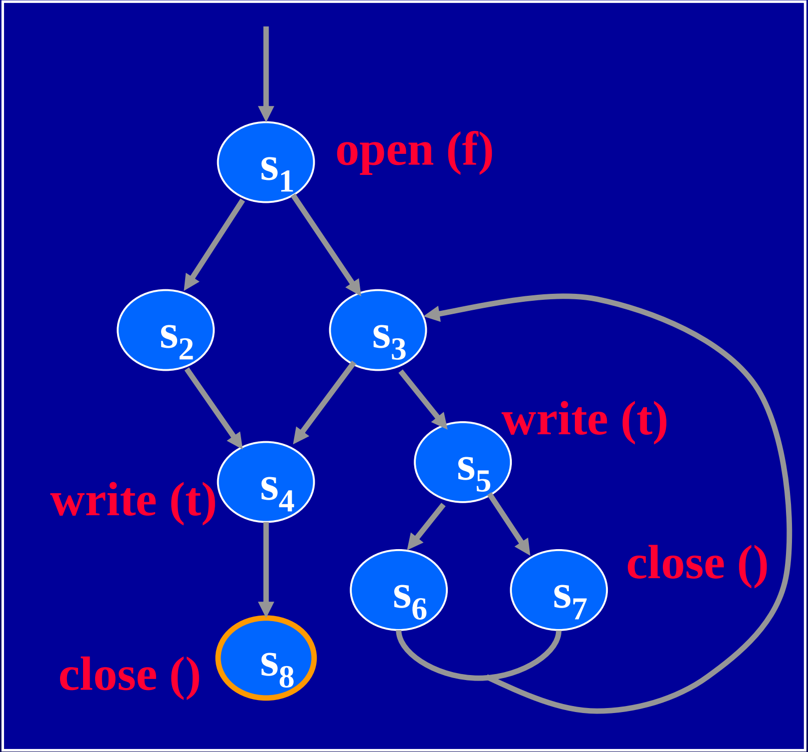


- Is there a path to a write() that does not go through an open() ?
- Is there a path to a close() that does not go through an open() ?
- Is there a path from a close() to a write()?
- Is there a path from an open() to a close() that does not go through a write() ? ("write-clear" path)

[1, 3, 4, 6] – ADT use anomaly!

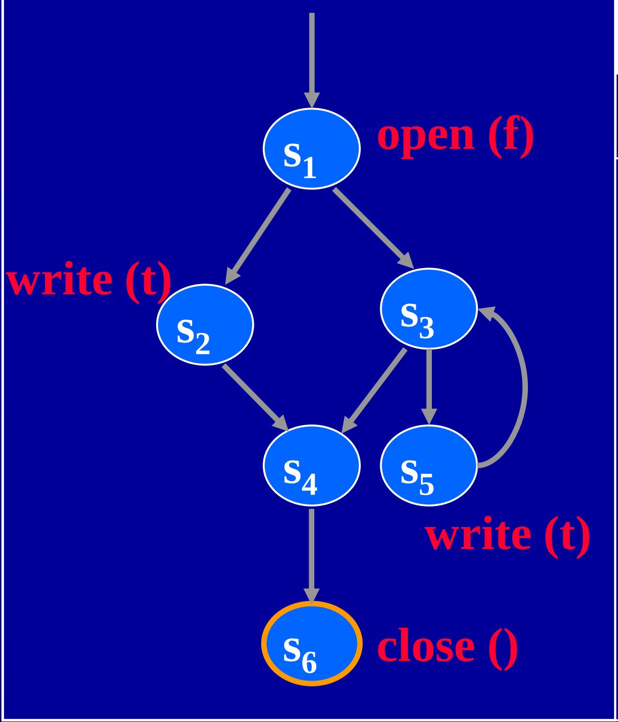
Static Checking

Consider the following graph :



[7, 3, 4] – close () before write () !

Generating Test Requirements



[1, 3, 4, 6] – ADT use anomaly!

- But it is possible that the logic of the program does not allow the pair of edges [1, 3, 4]
- That is – the loop body must be taken at least once
- Determining this is undecidable – so static methods are not enough

- Use the sequencing constraints to generate test requirements
- The goal is to violate every sequencing constraint

Test Requirements for FileADT

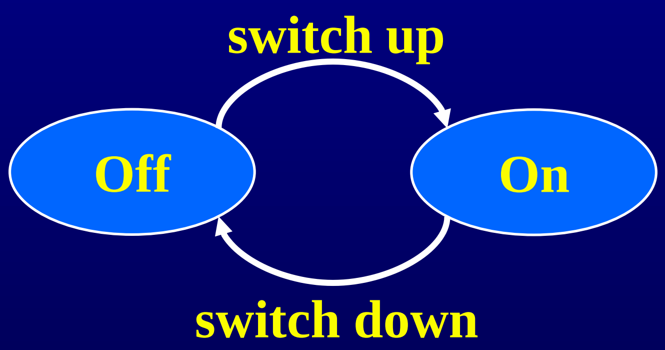
Apply to all programs that use FileADT

- Cover every path from the start node to every node that contains a write() such that the path does not go through a node containing an open()
- Cover every path from the start node to every node that contains a close() such that the path does not go through a node containing an open()
- Cover every path from every node that contains a close() to every node that contains a write()
- Cover every path from every node that contains an open() to every node that contains a close() such that the path does not go through a node containing a write()

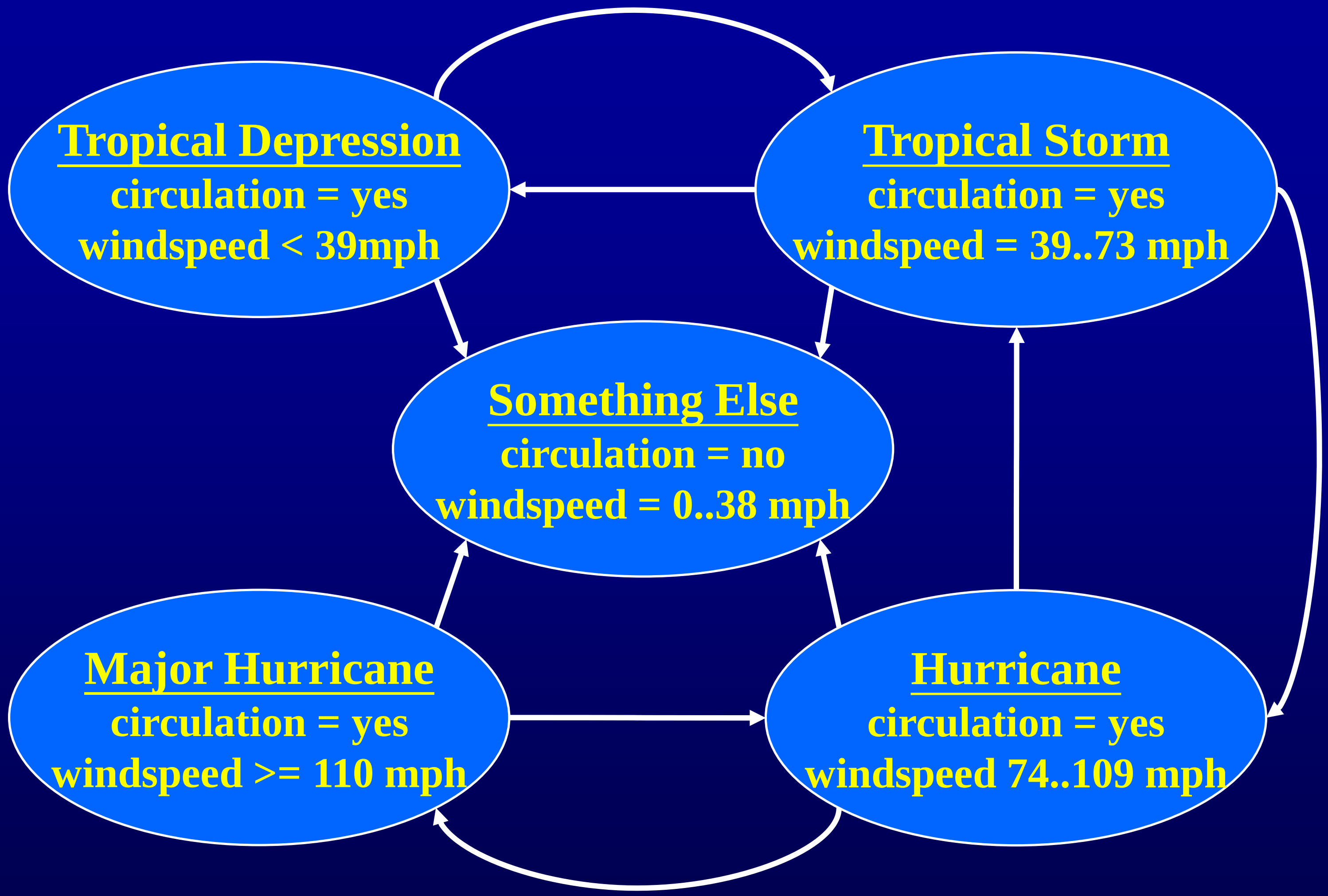
- If program is correct, all test requirements will be infeasible
- Any tests created will almost definitely find faults

Testing State Behavior

- A finite state machine (FSM) is a graph that describes how software variables are modified during execution
- Nodes : States, representing sets of values for key variables
- Edges : Transitions, possible changes in the state



Finite State Machine – Two Variables



Other variables may exist but not be part of state

Finite State Machines are Common (1/2)

- FSMs can accurately model many kinds of software
  - Embedded and control software (think electronic gadgets)
  - Abstract data types
  - Compilers and operating systems
  - Web applications
- Creating FSMs can help find software problems

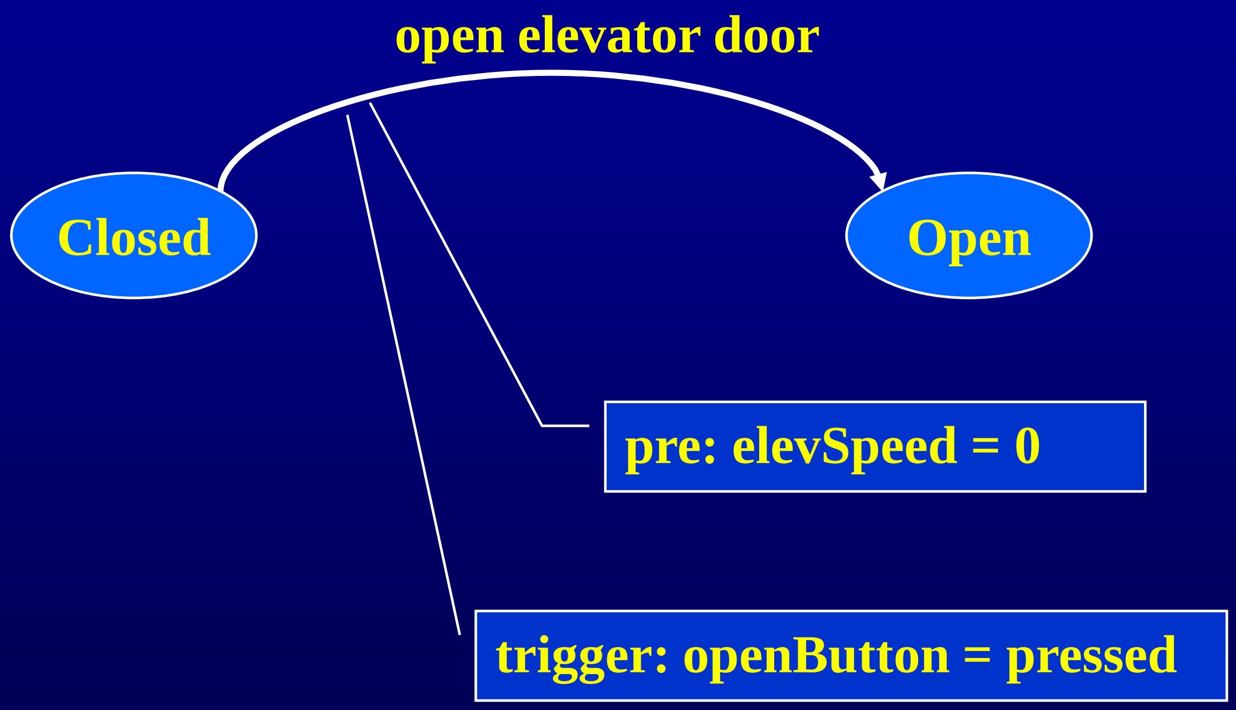
Finite State Machines are Common (2/2)

- Numerous languages for expressing FSMs
  - UML statecharts
  - Automata
  - State tables (SCR)
  - Petri nets
- Limitation : FSMs are not always practical for programs that have lots of states (for example, GUIs)

Annotations on FSMs

- FSMs can be annotated with different types of actions
  - Actions on transitions
  - Entry actions to nodes
  - Exit actions on nodes
- Actions can express changes to variables or conditions on variables
- These slides use the basics:
  - Preconditions (guards) : conditions that must be true for transitions to be taken
  - Triggering events : changes to variables that cause transitions to be taken
- This is close to the UML Statecharts, but not exactly the same

Example Annotations



Covering FSMs

- Node coverage : execute every state (*state coverage*)
- Edge coverage : execute every transition (*transition coverage*)
- Edge-pair coverage : execute pairs of transitions (*transition-pair*)
- Data flow:
  - Nodes often do not include defs or uses of variables
  - Defs of variables in triggers are used immediately (the next state)
  - Defs and uses are usually computed for guards, or states are extended
  - FSMs typically only model a subset of the variables
- Generating FSMs is often harder than covering them …

Deriving FSMs

- With some projects, an FSM (such as a statechart) was created during design
  - Tester should check to see if the FSM is still current with respect to the implementation
- If not, it is very helpful for the tester to derive the FSM
- Strategies for deriving FSMs from a program:
  1. Combining control flow graphs
  2. Using the software structure
  3. Modeling state variables
  4. Using implicit or explicit specifications
- Example based on a digital watch …
  - Class Watch uses class Time

Class Watch

Ask students to explain!

```
class Watch
// Constant values for the button (inputs)
private static final int NEXT = 0;
private static final int UP   = 1;
private static final int DOWN = 2;
// Constant values for the state
private static final int TIME      = 5;
private static final int STOPWATCH = 6;
private static final int ALARM     = 7;
// Primary state variable
private int mode = TIME;
// Three separate times, one for each state
private Time watch, stopwatch, alarm;

public Watch () // Constructor
public void doTransition (int button) // Handles inputs
public String toString ()  // Converts values
```

```
class Time   ( inner class )
private int hour   = 0;
private int minute = 0;

public void changeTime (int button)
public String toString ()
```

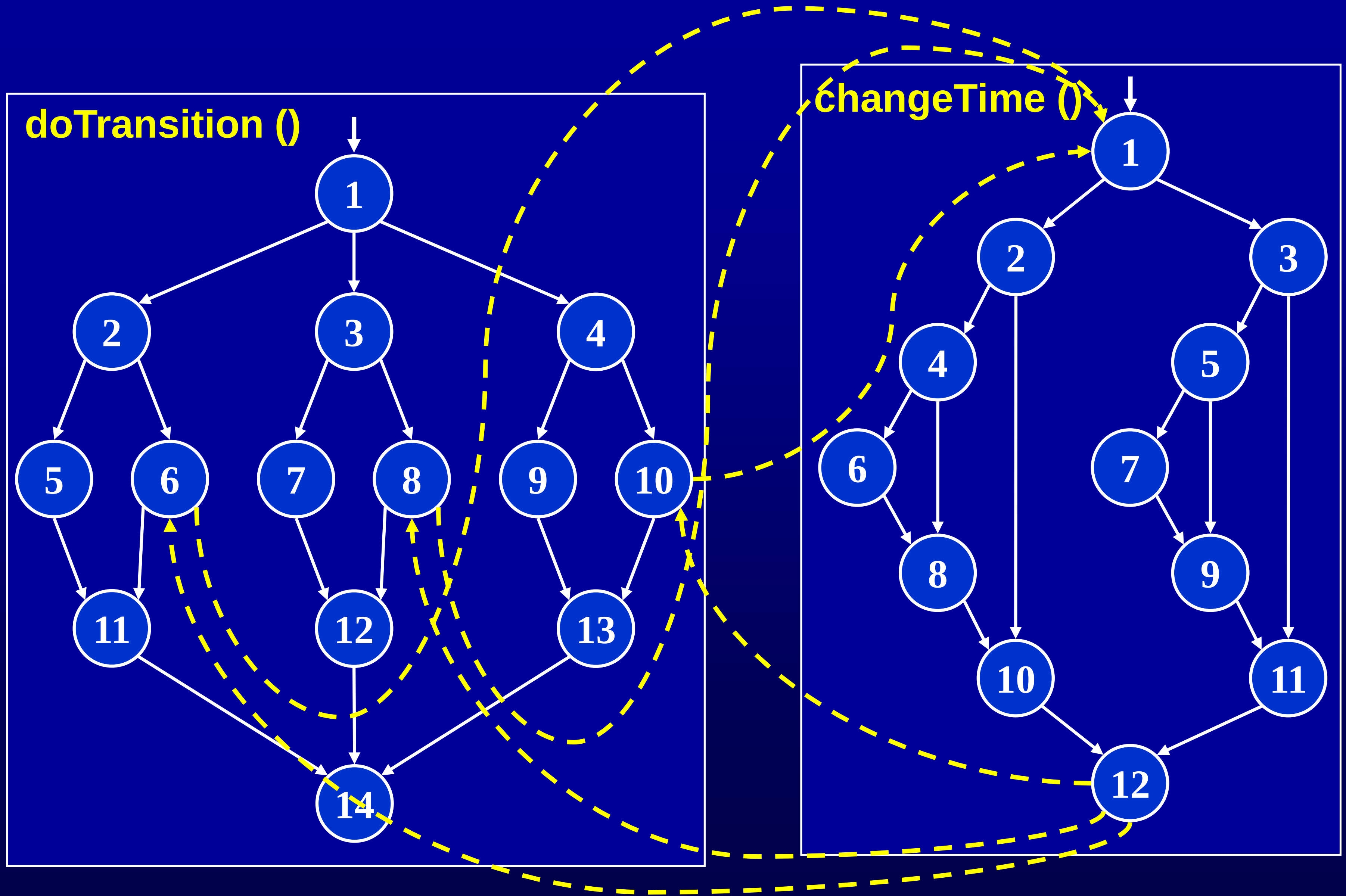
```java
// Takes the appropriate transition when a button is pushed.
public void doTransition (int button)
{
   switch ( mode )
   {
      case TIME:
         if (button == NEXT)
            mode = STOPWATCH;
         else
            watch.changeTime (button);
         break;
      case STOPWATCH:
         if (button == NEXT)
            mode = ALARM;
         else
            stopwatch.changeTime (button);
         break;
      case ALARM:
         if (button == NEXT)
            mode = TIME;
         else
            alarm.changeTime (button);
         break;
      default:
         break;
   }
}  // end doTransition()
```

```java
// Increases or decreases the time.
// Rolls around when necessary.
public void changeTime (int button)
{
   if (button == UP)
   {
      minute += 1;
      if (minute >= 60)
      {
         minute = 0;
         hour += 1;
         if (hour >= 12)
            hour = 0;
      }
   }
   else if (button == DOWN)
   {
      minute -= 1;
      if (minute < 0)
      {
         minute = 59;
         hour -= 1;
         if (hour <= 0)
            hour = 12;
      }
   }
}  // end changeTime()
```

1. Combining Control Flow Graphs

- The first instinct for inexperienced developers is to draw CFGs and link them together
- This is really not a FSM
- Several problems
  - Methods must return to correct callsites – built-in nondeterminism
  - Implementation must be available before graph can be built
  - This graph does not scale up
- Watch example …

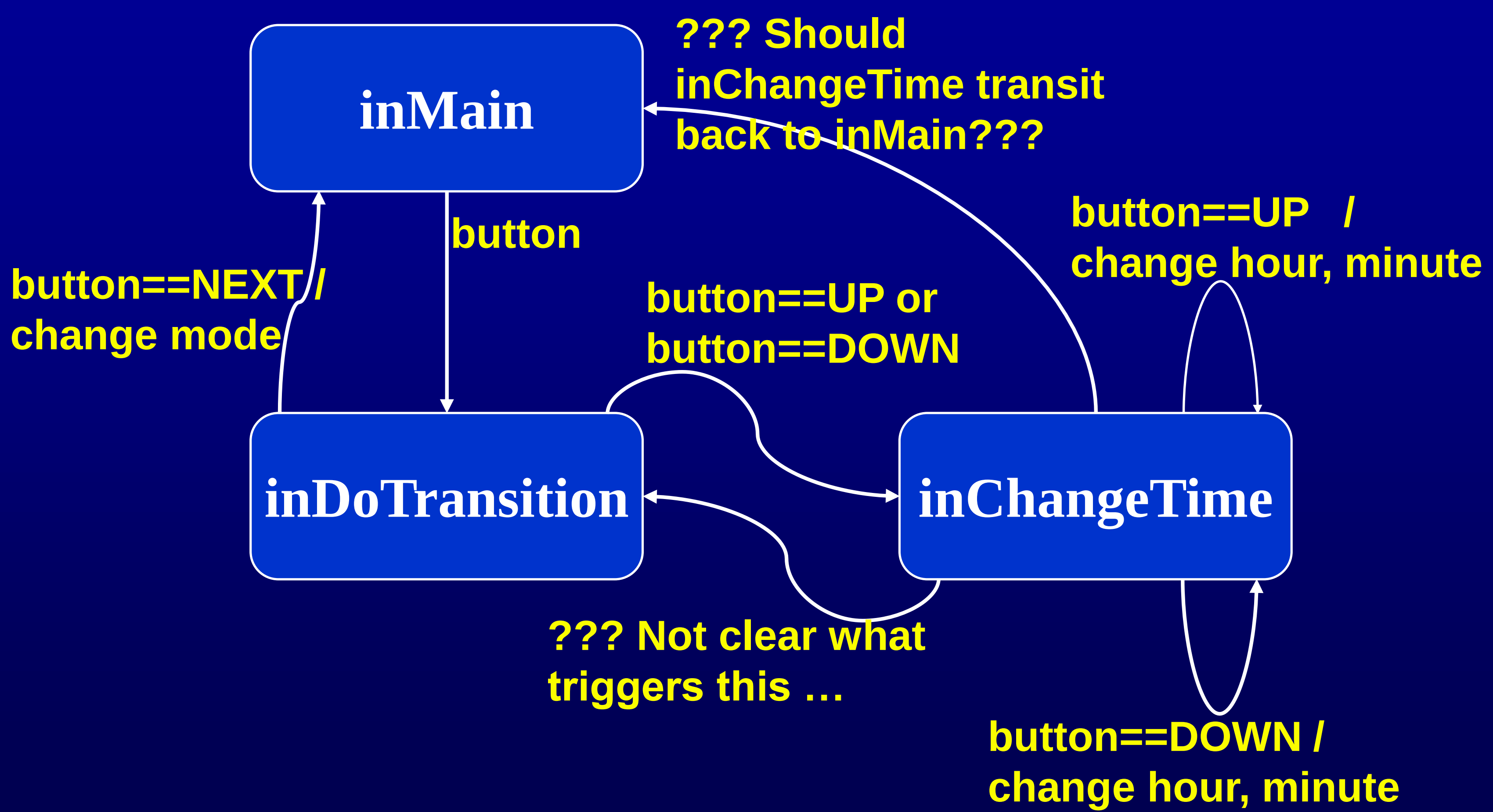
CFGs for Watch



2. Using the Software Structure

- A more experienced programmer may map methods to states
- These are really not states
- Problems
  - Subjective – different testers get different graphs
  - Requires in-depth knowledge of implementation
  - Detailed design must be present
- Watch example …

Software Structure for Watch



3. Modeling State Variables

- More mechanical
- State variables are usually defined early
- First identify all state variables, then choose which are relevant
- In theory, every combination of values for the state variables defines a different state
- In practice, we must identify ranges, or sets of values, that are all in one state
- Some states may not be feasible

State Variables in Watch

Constants

- ~~NEXT, UP, DOWN~~
- ~~TIME, STOPWATCH, ALARM~~

Not relevant, really just values

Non-constants

- int mode
- Time watch, stopwatch, alarm

Time class variables

- int hour
- int minute

Merge into the three Time variables

State Variables and Values

Relevant State Variables

- mode : TIME, STOPWATCH, ALARM
- watch : 12:00, 12:01..12:59, 01:00..11:59
- stopwatch : 12:00, 12:01..12:59, 01:00..11:59
- alarm : 12:00, 12:01..12:59, 01:00..11:59

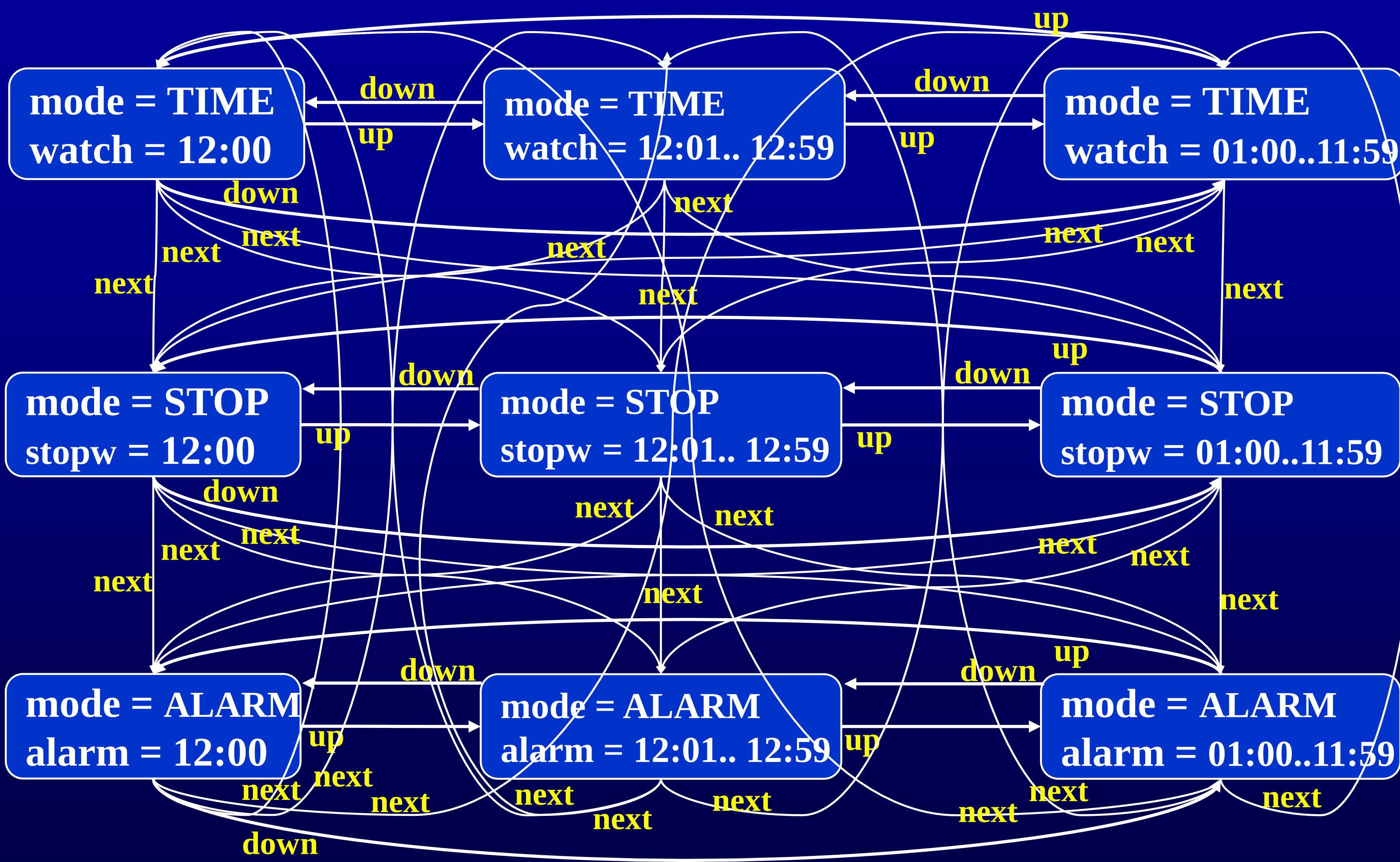
These three ranges actually represent quite a bit of thought and semantic domain knowledge of the program

Total 3*3*3*3 = 81 states …

But the three watches are independent, so we only care about 3+3+3 = 9 states

(still a messy graph …)

State Variable Model for Watch



NonDeterminism in the State Variable Model

- Each state has three outgoing transitions on *next*
- This is a form of non-determinism, but it is not reflected in the implementation
- Which transition is taken depends on the current state of the other watch
- The 81-state model would not have this non-determinism
- This situation can also be handled by a hierarchy of FSMs, where each watch is in a separate FSM and they are organized together

4. Using Implicit or Explicit Specifications

- Relies on explicit requirements or formal specifications that describe software behavior
- These could be derived by the tester
- These FSMs will sometimes look much like the implementation-based FSM, and sometimes much like the state-variable model
  - For watch, the specification-based FSM looks just like the state-variable FSM, so is not shown
- The disadvantage of FSM testing is that some implementation decisions are not modeled in the FSM

Summary–Tradeoffs in Applying Graph Coverage Criteria to FSMs

- Two advantages
  1. Tests can be designed before implementation
  2. Analyzing FSMs is much easier than analyzing source

- Three disadvantages
  1. Some implementation decisions are not modeled in the FSM
  2. There is some variation in the results because of the subjective nature of deriving FSMs
  3. Tests have to “mapped” to actual inputs to the program – the names that appear in the FSM may not be the same as the names in the program

Introduction to Software Testing
Chapter 2.6
Graph Coverage for Use Cases

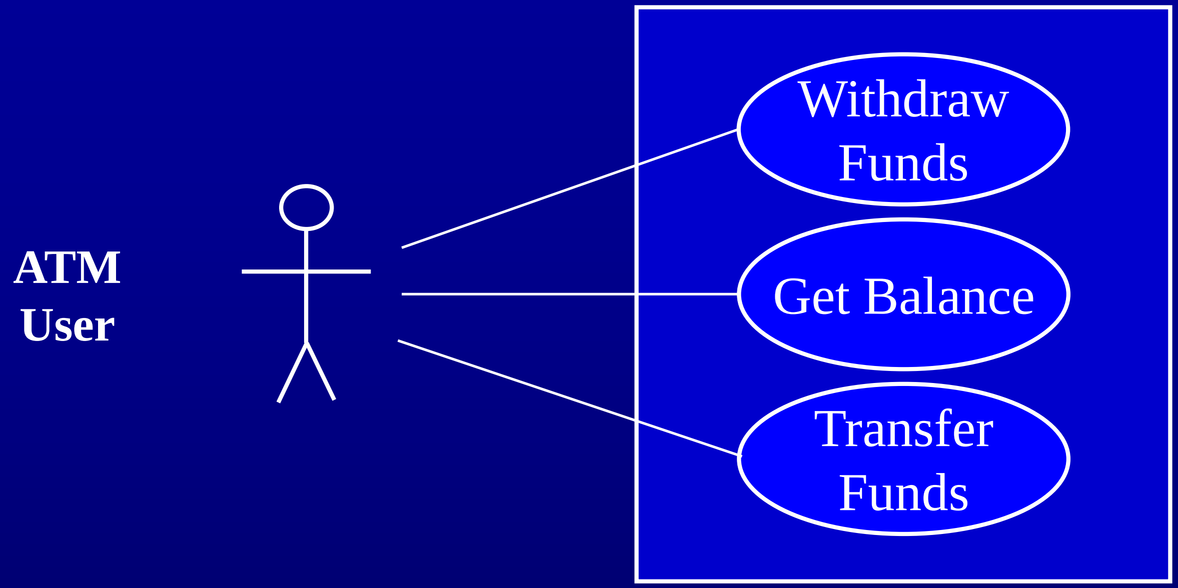
Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

UML Use Cases

- UML use cases are often used to express **software requirements**
- They help express computer application **workflow**
- We won’t teach use cases, but show **examples**

Simple Use Case Example



- **Actors** : Humans or software components that use the software being modeled
- **Use cases** : Shown as circles or ovals
- **Node Coverage** : Try each use case once …

Use Case graphs, by themselves, are not useful for testing

Elaboration

- Use cases are commonly **elaborated** (or **documented**)
- Elaboration is first written **textually**
  - **Details** of operation
  - **Alternatives** model choices and conditions during execution

2013/11/21 stopped here.

Elaboration of ATM Use Case （張唯霖講）

- Use Case Name : Withdraw Funds
- Summary : Customer uses a valid card to withdraw funds from a valid bank account.
- Actor : ATM Customer
- Precondition : ATM is displaying the idle welcome message
- Description :
  - Customer inserts an ATM Card into the ATM Card Reader.
  - If the system can recognize the card, it reads the card number.
  - System prompts the customer for a PIN.
  - Customer enters PIN.
  - System checks the card's expiration date and whether the card has been stolen or lost.
  - If the card is valid, the system checks if the entered PIN matches the card PIN.
  - If the PINs match, the system finds out what accounts the card can access.
  - System displays customer accounts and prompts the customer to choose a type of transaction. There are three types of transactions, Withdraw Funds, Get Balance and Transfer Funds. (The previous eight steps are part of all three use cases; the following steps are unique to the Withdraw Funds use case.)

Elaboration of ATM Use Case–(2/3) （吳嘉峰講）

- **Description (continued) :**
  - Customer selects Withdraw Funds, selects the account number, and enters the amount.
  - System checks that the account is valid, makes sure that customer has enough funds in the account, makes sure that the daily limit has not been exceeded, and checks that the ATM has enough funds.
  - If all four checks are successful, the system dispenses the cash.
  - System prints a receipt with a transaction number, the transaction type, the amount withdrawn, and the new account balance.
  - System ejects card.
  - System displays the idle welcome message.

Elaboration of ATM Use Case–(3/3) （吳庭蓁講）

- <u>Alternatives</u> :
  - If the system cannot recognize the card, it is ejected and the welcome message is displayed.
  - If the current date is past the card's expiration date, the card is confiscated and the welcome message is displayed.
  - If the card has been reported lost or stolen, it is confiscated and the welcome message is displayed.
  - If the customer entered PIN does not match the PIN for the card, the system prompts for a new PIN.
  - If the customer enters an incorrect PIN three times, the card is confiscated and the welcome message is displayed.
  - If the account number entered by the user is invalid, the system displays an error message, ejects the card and the welcome message is displayed.
  - If the request for withdraw exceeds the maximum allowable daily withdrawal amount, the system displays an apology message, ejects the card and the welcome message is displayed.
  - If the request for withdraw exceeds the amount of funds in the ATM, the system displays an apology message, ejects the card and the welcome message is displayed.
  - If the customer enters Cancel, the system cancels the transaction, ejects the card and the welcome message is displayed.
- <u>Postcondition</u> :
  - Funds have been withdrawn from the customer’s account.

Wait A Minute …

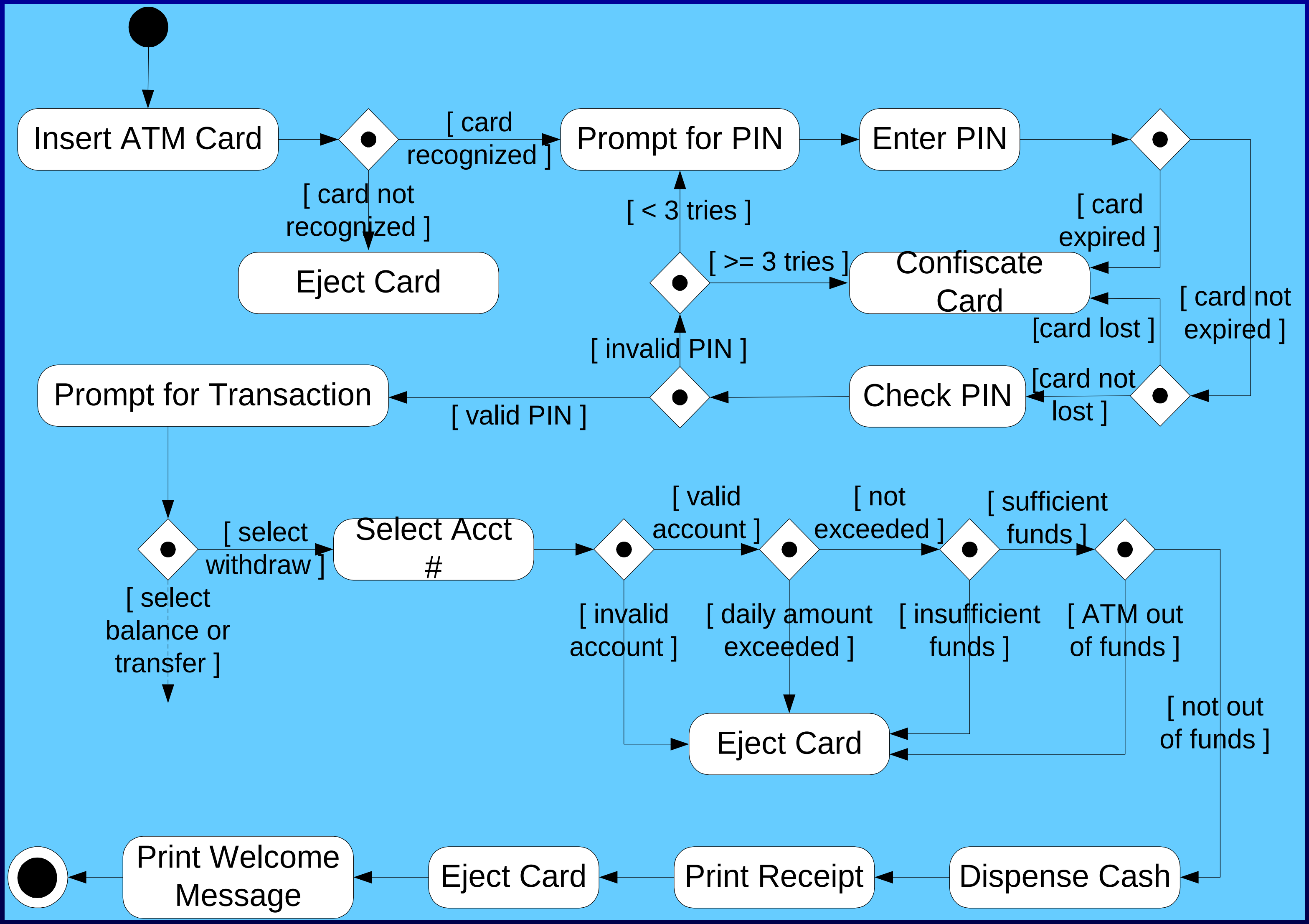
- What does this have to do with **testing** ?
- Specifically, what does this have to do with **graphs** ???
- Remember our admonition : **Find** a graph, then cover it!
- Beizer suggested "**Transaction Flow Graphs**" in his book
- UML has something very similar :

Activity Diagrams

Use Cases to Activity Diagrams

- Activity diagrams indicate flow among activities
- Activities should model user level steps
- Two kinds of nodes:
  - Action states
  - Sequential branches
- Use case descriptions become action state nodes in the activity diagram
- Alternatives are sequential branch nodes
- Flow among steps are edges
- Activity diagrams usually have some helpful characteristics:
  - few loops, simple predicates, no obvious DU pairs

ATM Withdraw Activity Graph



Covering Activity Graphs (1/4)

- **Node Coverage**
  - **Inputs to the software are derived from labels on nodes and predicates**
  - **Used to form test case values**
- **Edge Coverage**
- **Data flow techniques do not apply**

Covering Activity Graphs (2/4)

- **Scenario Testing**
  - **Scenario : A complete path through a use case activity graph**
  - **Should make semantic sense to the users**
  - **Scenario 1, Soap Opera:**

1. Wife bursted into bedroom. Husband lied in bed.
2. Wife: *Why were you not by my side ?*
3. Husband: *I didn't want to hurt my mom.*
4. Wife: *Don't you love me any more ?*
5. Husband: *Would you stop being childish again!*
6. Wife slapped her husband, ran out of room into rain in tears.

Covering Activity Graphs (3/4)

- **Scenario Testing**
  - **Scenario : A complete path through a use case activity graph**
  - **Should make semantic sense to the users**
  - **Scenario 2, Soap opera:**

1. Wife bursted into bedroom. Husband lied in bed.
2. Wife: *Why were you not by my side ?*
3. Husband: *I didn't want to hurt my mom.*
4. Wife: *Don't you love me any more ?*
5. Husband hugged wife fiercely. They fell into bed.

Covering Activity Graphs (4/4)

- Scenario Testing
  - Scenario : A complete path through a use case activity graph
  - Should make semantic sense to the users
  - Number of paths often finite
    - If not, scenarios defined based on domain knowledge
  - Use "specified path coverage", where the set S of paths is the set of scenarios
  - Note that specified path coverage does not necessarily subsume edge coverage, but scenarios <u>should be</u> defined so that it does

Summary of Use Case Testing

- Use cases are defined at the requirements level
- Can be very high level
- UML Activity Diagrams encode use cases in graphs
  - Graphs usually have a fairly simple structure
- Requirements-based testing can use graph coverage
  - Straightforward to do by hand
  - Specified path coverage makes sense for these graphs

Introduction to Software Testing
Chapter 2.7
Representing Graphs Algebraically

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

Graphs – Circles and Arrows

- It is usually easier for humans to understand graphs by viewing nodes as circles and edges as arrows
- Sometimes other forms are used to manipulate the graphs
  - Standard data structures methods are appropriate for tools
  - An algebraic representation (*regular expression*) allows certain useful mathematical operations to be performed

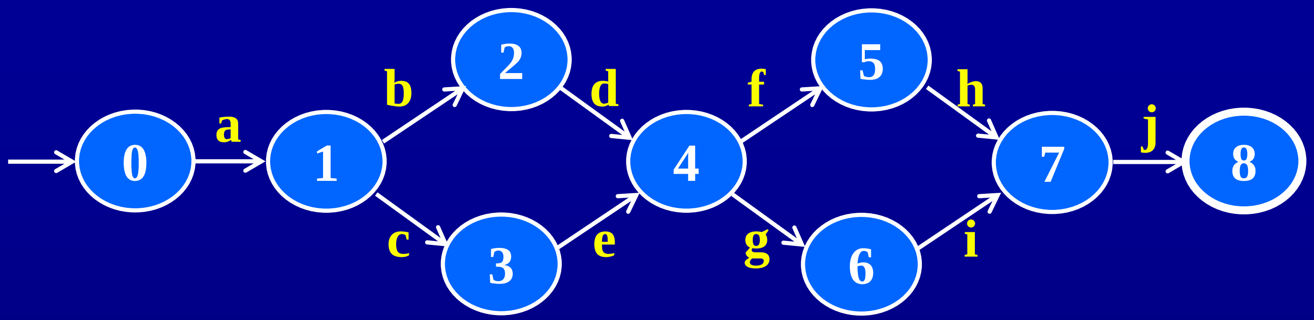
Representing Graphs Algebraically

- Assign a unique label to each edge
  - Could represent semantic information, such as from an activity graph or an FSM
  - Could be arbitrary – we use lower case letters as an abstraction
- Operators:
  - Concatenation (multiplicative) : if edge *a* is followed by edge *b*, they are concatenated as "$a * b$", or more usually, "ab"
  - Selection (additive) : if either edge *a* or *b* can be taken, they are summed as "$a + b$"

Representing Graphs Algebraically

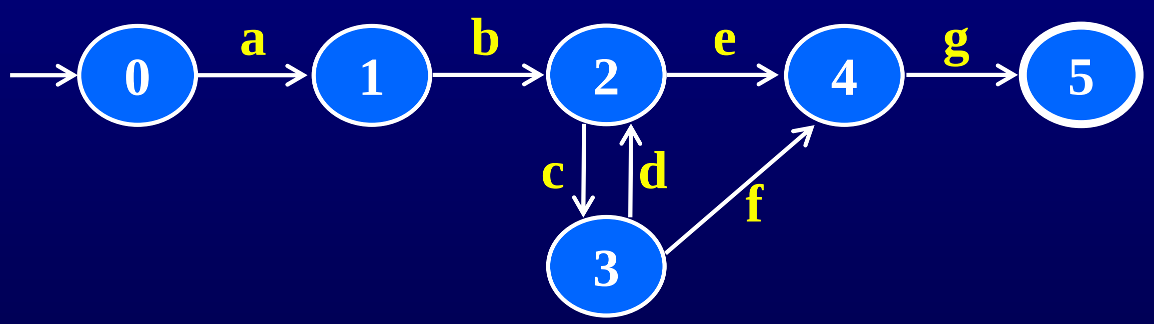
- **<u>Path Product</u> : A sequence of edges "multiplied" together is called a product of edges, or a path product**
- **<u>Path Expression</u> : Path products and possibly '+' operators**
  - $A = ab + cd$
- **<u>Loops</u> : represented as exponents**
  - $A = (ab+c)^*$
  - $A = (ab+c)^3$

Examples



Double-diamond graph

Path Expression = abdfhj + abdgij + acegij + acefhj



Path Products : A = abeg, B = (cd)*, C = cf

Path Expression = ab (cd)* (e + cf) g

Let's Look at the Math

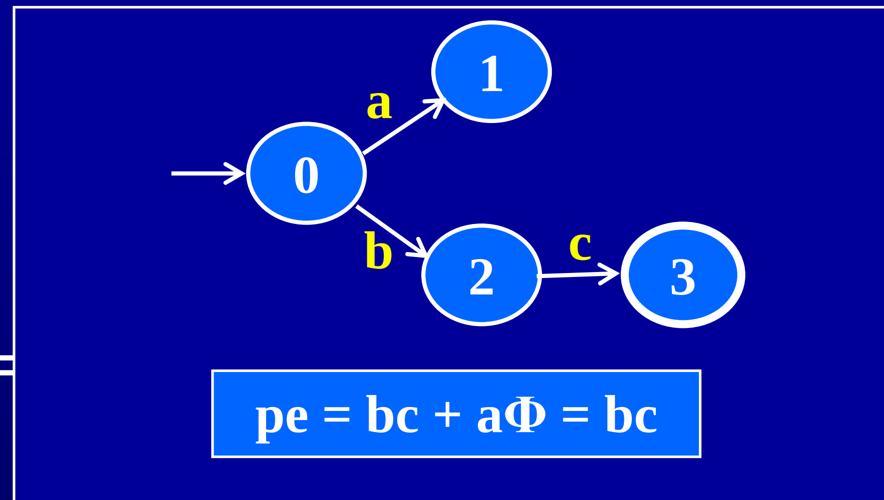
- **Path Products**
  - **Commutative** : Path product is not ($AB \mathrel{!=} BA$)
  - **Associative** : Path product is ($A(BC) = (AB)C = ABC$)
- **Path Summation**
  - **Commutative** : Summation is ($A + B = B + A$)
  - **Associative** : ($(A+B)+C = A+(B+C) = A + B + C$)
- **These are close to the "usual" arithmetic addition and multiplication, so most of the standard algebraic laws can be used**
  - **In normal math, multiplication is commutative**
- **(Don't worry … we'll get back to testing soon …)**

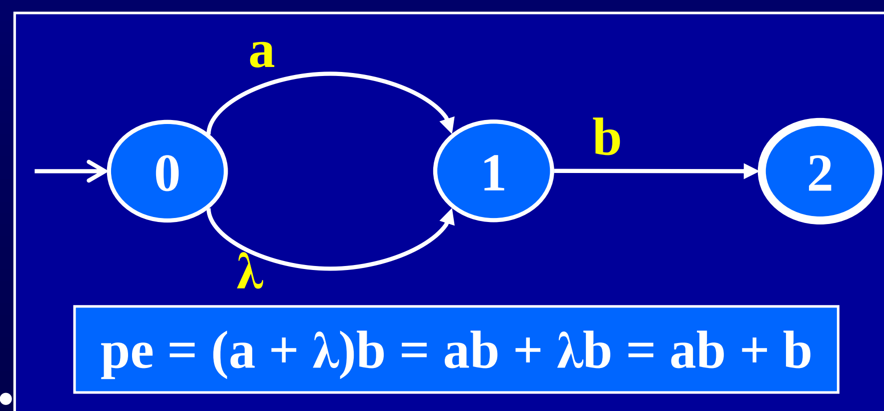
Algebraic Laws on Graph Expressions

- **Distributive** : A (B + C) = AB + AC
- **Distributive** : (B + C) D = BD + CD
- **Absorption rule** : A + A = A
- Shortcut notation for loops
  - At least one iteration : $A^{+} = AA^{*}$
  - Bounds on iteration : $A^{\underline{3}} = A^{0} + A^{1} + A^{2} + A^{3}$
    - More generally : $A^{\underline{n}} = A^{0} + A^{1} + \ldots + A^{n}$
- Absorbing exponents
  - $A^{\underline{n}} + A^{\underline{m}} = A^{\underline{max(n,m)}}$
  - $A^{\underline{n}}A^{\underline{m}} = A^{\underline{n+m}}$
  - $A^{\underline{n}} A^{*} = A^{*}A^{\underline{n}} = A^{*}$
  - $A^{\underline{n}} A^{+} = A^{+}A^{\underline{n}} = A^{+}$
  - $A^{*} A^{+} = A^{+}A^{*} = A^{+}$

Identity Operators

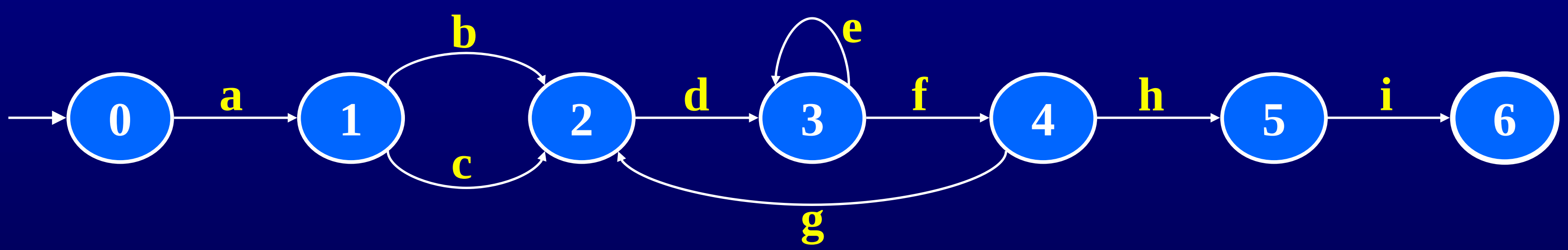
- **Multiplicative** : λ (an empty path)
- **Additive** : Φ (a null path – set of paths that has no paths)
- **Multiplicative identity laws**
  - $\lambda + \lambda = \lambda$
  - $\lambda A = A\lambda = A$
  - $\lambda^n = \lambda^{\underline{n}} = \lambda^* = \lambda^+ =$
  - $\lambda^+ + \lambda = \lambda^* = \lambda$
- **Additive identity laws**
  - $A + \Phi = \Phi + A = A$
  - $A \Phi = \Phi A = \Phi$
  - $\Phi^* = \lambda + \Phi + \Phi^2 + \ldots$





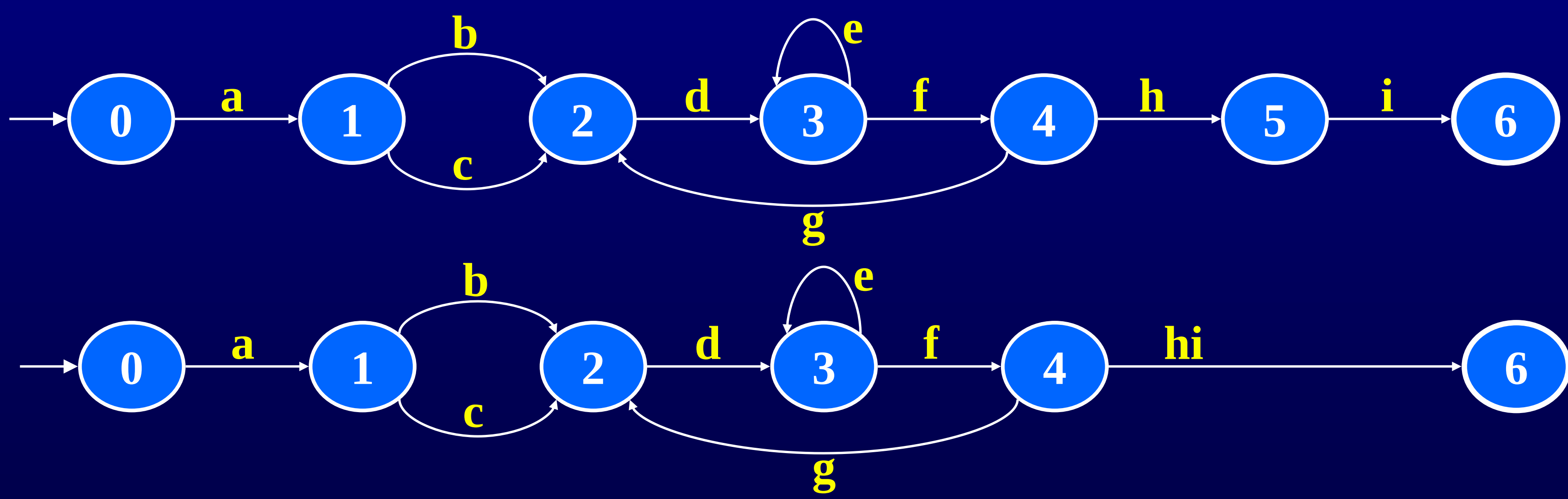
Graphs to Path Expressions

- General algorithms for reducing finite state machines to regular expressions are widely known
- This lecture presents a four step, iterative, special case process
- The four steps are illustrated on a simple graph



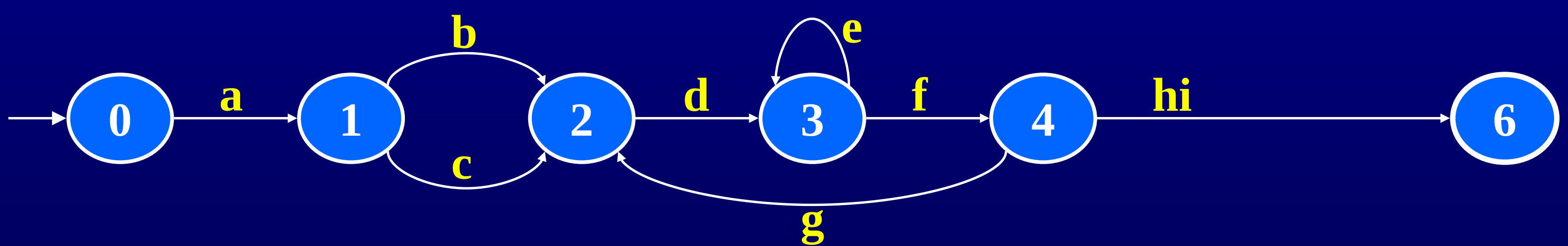
Step 1 : Sequential Edges

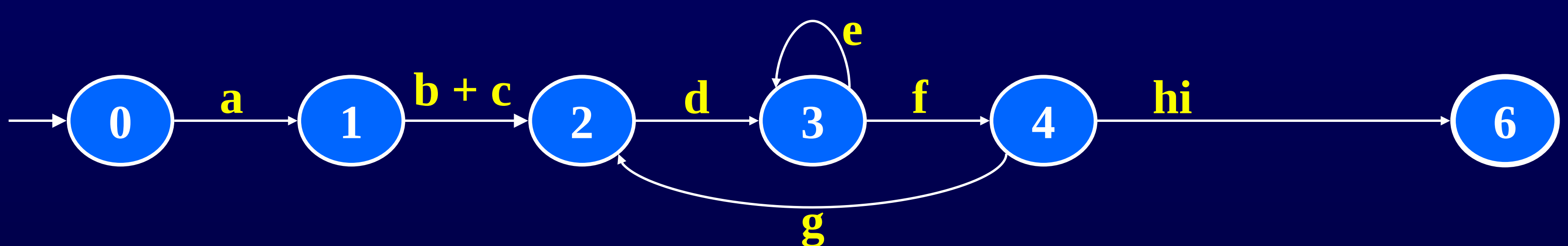
- Combine all sequential edges
- Multiply the edge labels
- Precisely: For any node that has only one incoming and one outgoing edge, eliminate the node, combine the two edges, and multiply their path expressions
- Example: Combine edges h and i



Step 2 : Parallel Edges

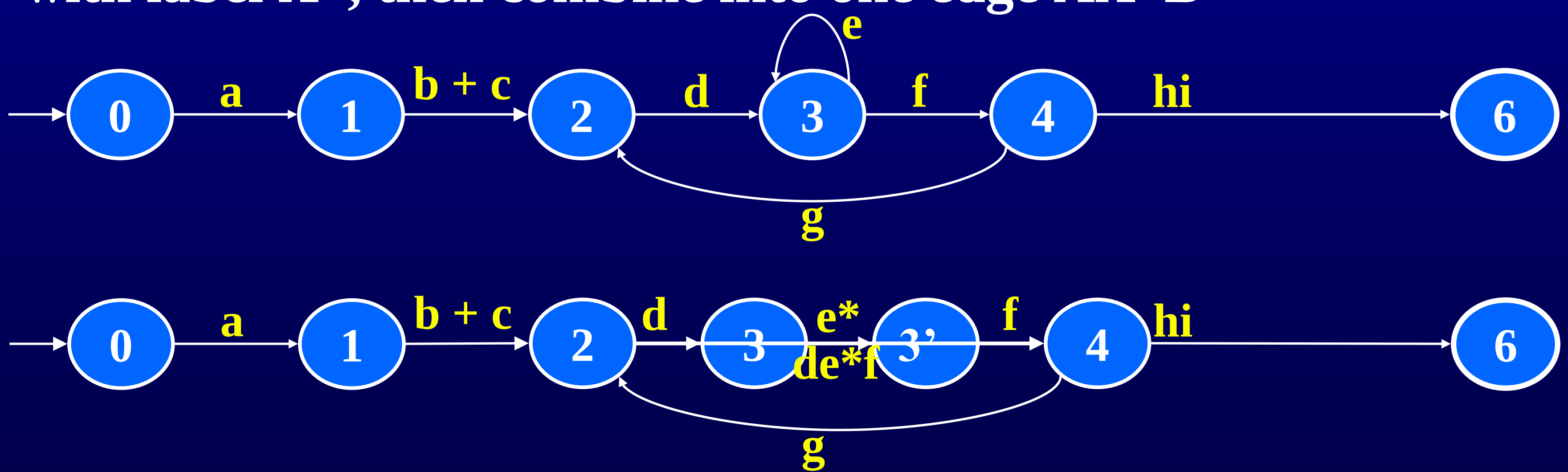
- Combine all parallel edges
- Add the edge labels
- Precisely: For any pair of edges with the same source and target nodes, combine the edges and add their path expressions
- Example : Combine edges b and c





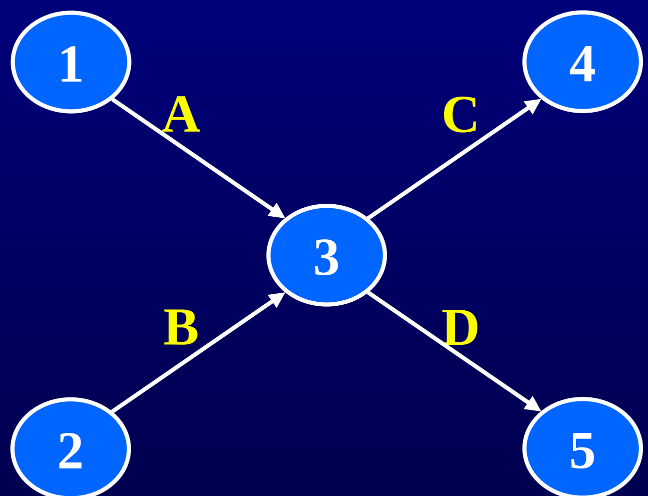
Step 3 : Self-Loops

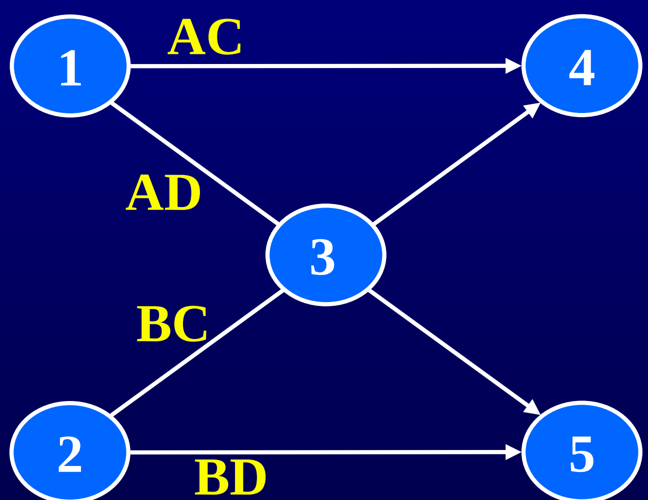
- Combine all self-loops (loops from a node to itself)
- Add a new "dummy" node
- An incoming edge with exponent
- Merge the three resulting sequential nodes with multiplication
- Precisely: For any node n1 with a self-loop X, incoming edge A and outgoing edge B, remove X, add a new node n1' and edge with label X*, then combine into one edge AX*B



Step 4 : Remove Tester-Chosen Nodes

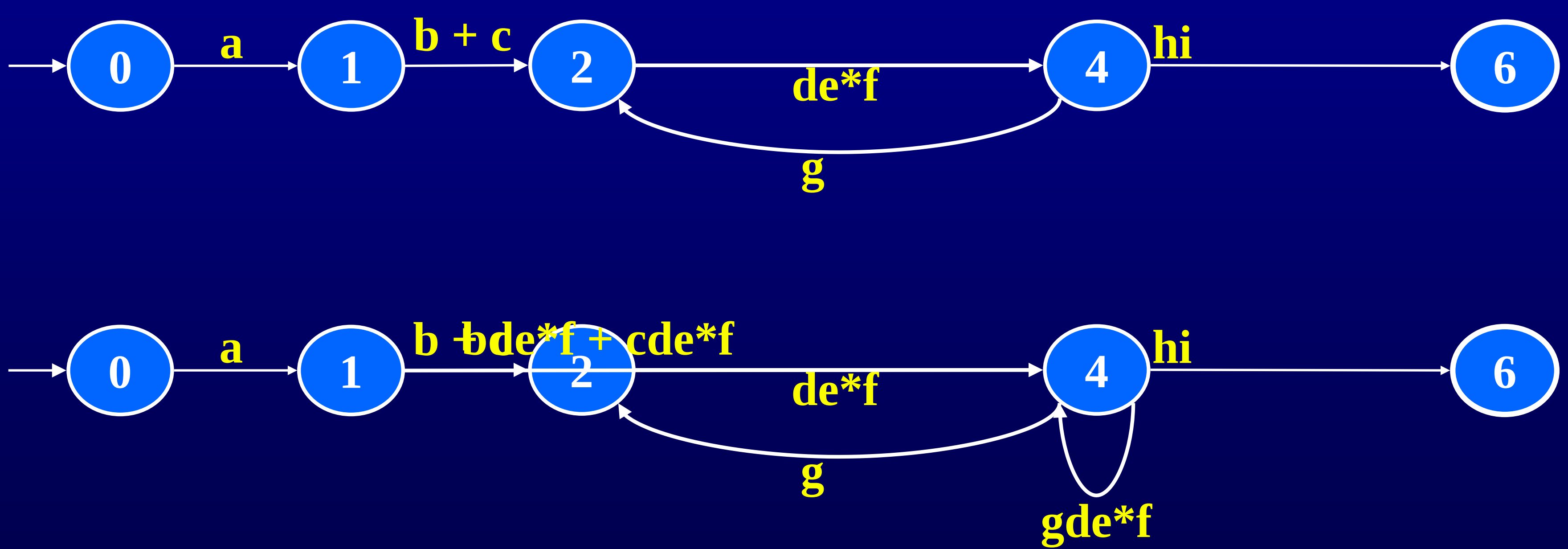
- Use for "other" special cases, when steps 1-3 do not apply
- Choose a node that is not initial or final
- Replace it by inserting edges from all predecessors to all successors
- Multiply path expressions from all incoming with all outgoing edges





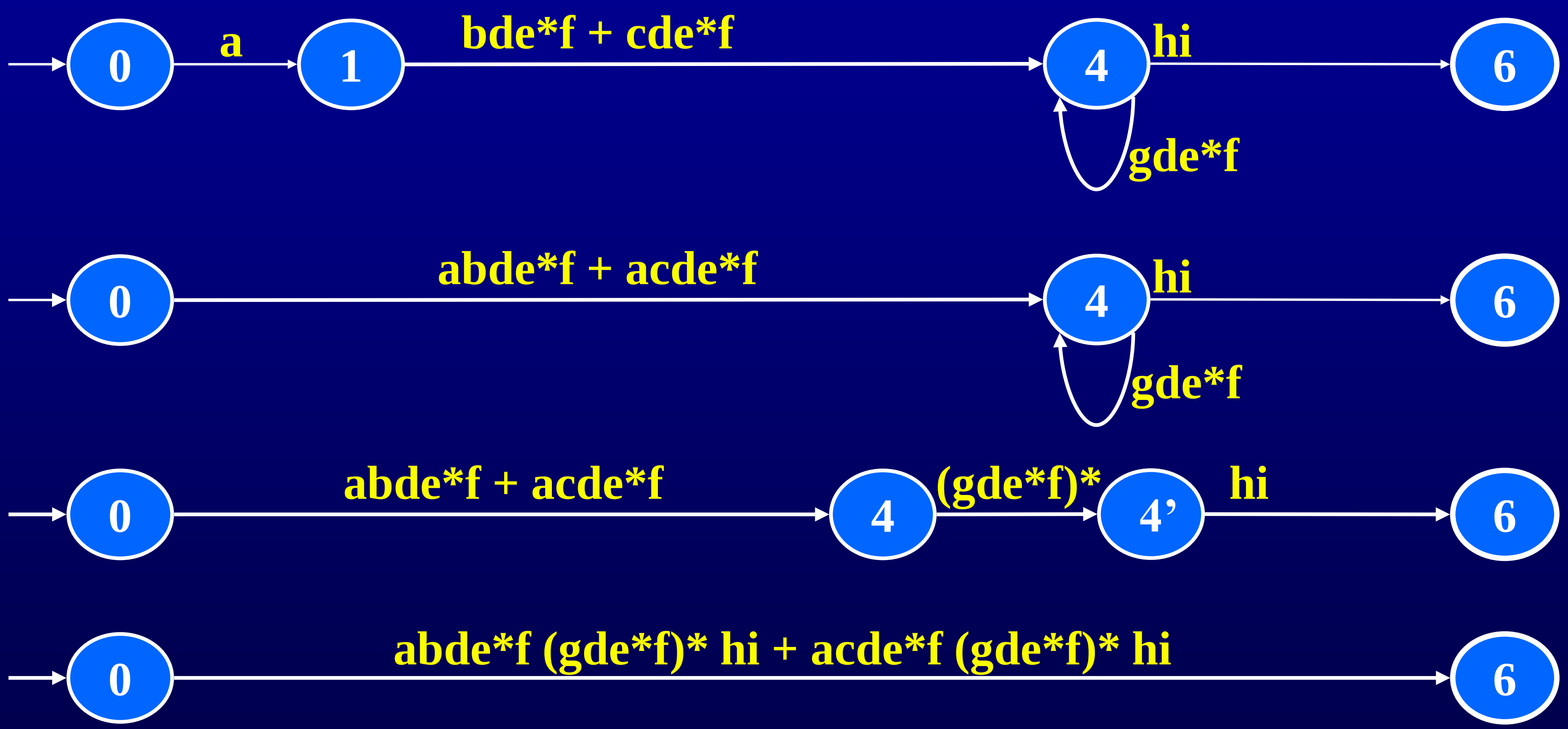
Step 4 : Eliminate Node 2

- Remove node 2
- Edges (1, 2) and (2, 4) become one edge
- Edges (4, 2) and (2, 4) become a self-loop



Repeat Steps 1 Through 4

Continue until only one edge is left …



Applications of Path Expressions

1. Deriving test inputs
2. Counting paths in a flow graph
3. Minimum number of paths to satisfy All Edges
4. Complementary operations analysis

1. Deriving Test Inputs

- Very simple … find a graph and cover it
- Cover regular expressions by covering each separate path product
- This is a form of the specified path coverage criterion
- Loops are represented by exponents – if an unbounded exponent appears, replace with a constant value based on domain knowledge of the program
- Test requirements for running example:
  - Final path expression:

    $$abde^*f\ (gde^*f)^*\ hi + acde^*f\ (gde^*f)^*\ hi$$

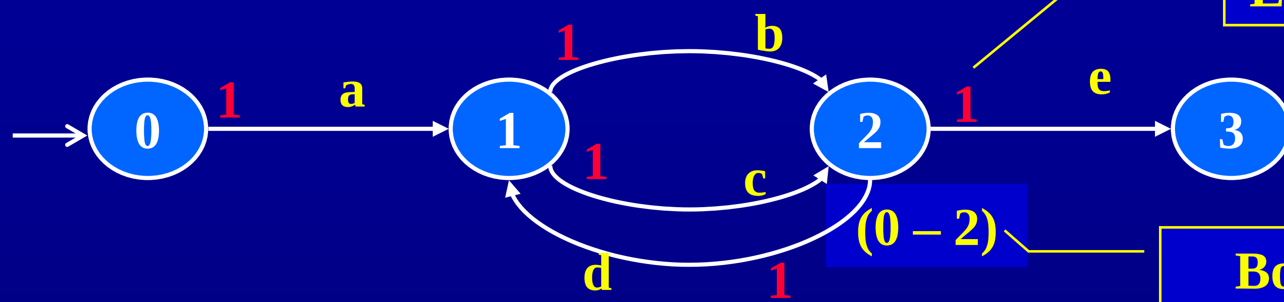
  - Test Requirements

    $abde^5f\ (gde^5f)^5\ hi$, $acde^5f\ (gde^5f)^5\ hi$

2. Counting Paths in a Flow Graph

- It is sometimes useful to know the number of paths in a graph
- The path expressions allow this computation with straightforward arithmetic
- Cycles mean we have an infinite number of paths, so we have to make assumptions to approximate
- Put a reasonable bound on the number of iterations by replacing the '*' with an integer value
  - The bound may be a true maximum number of iterations
  - The bound may represent a tester's assumption that executing the loop '*N times*' is enough

2. Counting Paths in a Flow Graph Example



Edge weights

Bounds the loop – max 2, min 0 iterations

$$pe = a\,(b + c)\,(d\,(b + c)\,)^{\underline{2}}\,e$$

$$= 1 * (1 + 1) * (1 * (1 + 1)\,)^{\underline{2}} * 1$$

$$= 1 * 2 * 2^{\underline{2}} * 1$$

$$= 2 * (\,\Sigma^{2}_{i=0}\, 2^{i}\,) * 1$$

$$= 2 * (\,2^{0} + 2^{1} + 2^{2}\,) * 1$$

$$= 2 * (\,1 + 2 + 4\,) * 1$$

$$= 2 * 7 * 1$$

$$= 14$$

2. Counting Costs of Executing Paths Edge Weights (1/2)

- It sometimes helps to have the number of paths include a measure of *cost* for executing each path
  - Number of paths in a function call
  - Expensive operations
  - An operation our intuition tells the tester to avoid or encourage
- Edge weights default to 1 – otherwise marked by tester
- Loop weight: the maximum number of iterations allowed
  - Only mark one edge per cycle !

2. Counting Costs of Executing Paths
Edge Weights (2/2)

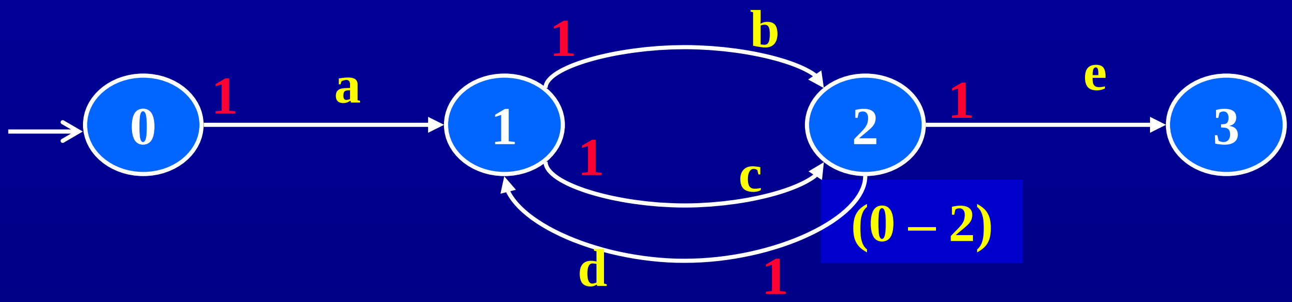
- Compute path expression and substitute weights for edges
  - Path expression :

    A+B becomes *weight (A) + weight (B)*
  - Path product : *AB* becomes *weight (A) * weight (B)*
  - Loop is sum of the weight of all iterations
- The result of this computation is the estimated total cost of executing all paths

3. Minimum Number of Paths to Satisfy All Edges

- If we wish to satisfy All Edges, how many paths are needed?
- How many tests will be needed?
- Similar to computing the maximum number of paths
  - Different computation
- Specifically:
  - Path expression: *A+B* becomes *weight (A)+weight (B)*
  - Path product: *AB* becomes *max(weight(A), weight(B))*
  - Loop : A^n is either *1* or *weight (A)*
  - Judgment of tester – if all paths in the loop can be taken in one test, use *1*, else use *weight(A)*

3. Min Number of Paths to Satisfy All Edges Example



$$pe = a\ (b + c)\ (d\ (b + c)\ )^{\underline{2}}\ e$$

<u>Conservatively</u> assume the same edge from 1 to 2 must be taken every iteration through the loop –use the edge weight …

$$= 1 * 2 * (1 * (2))^{\underline{2}} * 1$$

$$= 1 * 2 * (1 * 2) * 1$$

$$= \max\ (1, 2, 1, 2, 1)$$

$$= 2$$

4. Complementary Operations Analysis (1/2)

- **A method for finding potential anomalies**
  - **Def-use anomalies (use before a def)**
  - **FileADT example (closing before writing)**
- **A pair of operations are *complementary* if their behaviors negate each other, or one must be done before the other**
  - **push & pop**
  - **enqueue & dequeue**
  - **getting memory & disposing of memory**
  - **open & close**

4. Complementary Operations Analysis (2/2)

- **Edge weights are replaced with one of three labels**
  - ***C*** **– Creator operation**
  - ***D*** **– Destructor operation**
  - ***1*** **– Neither**

4. Complementary Operations Analysis Arithmetic Operations

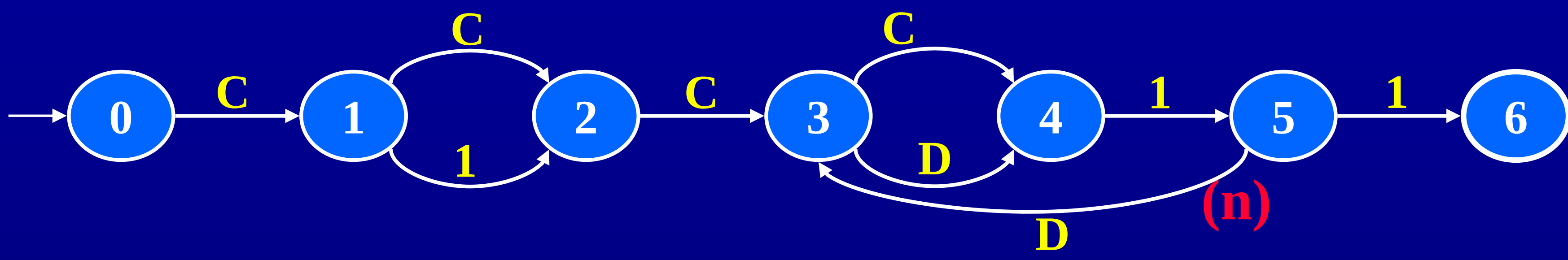
- The addition and multiplication operators are replaced with two tables

*	C	D	1
C	C^2	1	C
D	DC	D^2	D
1	C	D	1

+	C	D	1
C	C	C+D	C+1
D	D+C	D	D+1
1	C+1	D+1	1

- Not the same as usual algebra
  - C*D = 1, C+C = C, D+D = D
  - Multiplication is not commutative because the operations are not equivalent … a destructor cancels a creator, but not vice versa

4. Complementary Operations Analysis Example



$$pe = C\ (C+1)\ C\ (C+D)\ 1\ (D\ (C+D)\ 1)^n\ 1$$

$$\text{reduced } pe = (CCCC + CC\cancel{CD} + CCC + C\cancel{CD})\ (DC + DD)^n$$

C*D = 1, canceling out

$$\text{final } pe = (CCCC + CC + CCC + C)\ (DC + DD)^n$$

4. Using Complementary Operations Analysis

final pe = (CCCC + CC + CCC + C) (DC + DD)^n

- **Ask questions:**
  - **Can we have more destructors than creators?**
    - **CCCD (DD)^n, n > 1**
    - **CCCD (DD)^n, n > 0**
    - **CCC (DDDCDD)**
  - **Can we have more creators than destructors?**
    - **CCCC**
    - **CCD (DC)^n, forall n**
- **Each "yes" response represents a specification for a test that <u>might</u> cause anomalous behavior**

Summary of Path Expressions

- **Having an algebraic representation of a graph can be very useful**
- **Most techniques involve some human input and subjectivity**
- **The techniques have not been sufficiently quantified**
- **We know of no commercial tools to support this analysis**