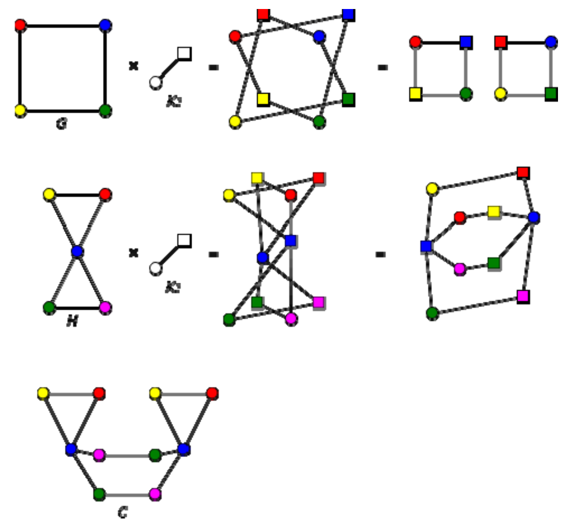
Coverage

- Literature of software testing is primarily concerned with various notions of *coverage*
- Four basic kinds of coverage:
  - **Graph coverage**
  - **Logic coverage**
  - **Input space partitioning**
  - **Syntax-based coverage**



$$\neg(p \wedge \neg p)$$

- Two purposes: to know what we have & haven’t tested, and to know when we can “safely” stop testing

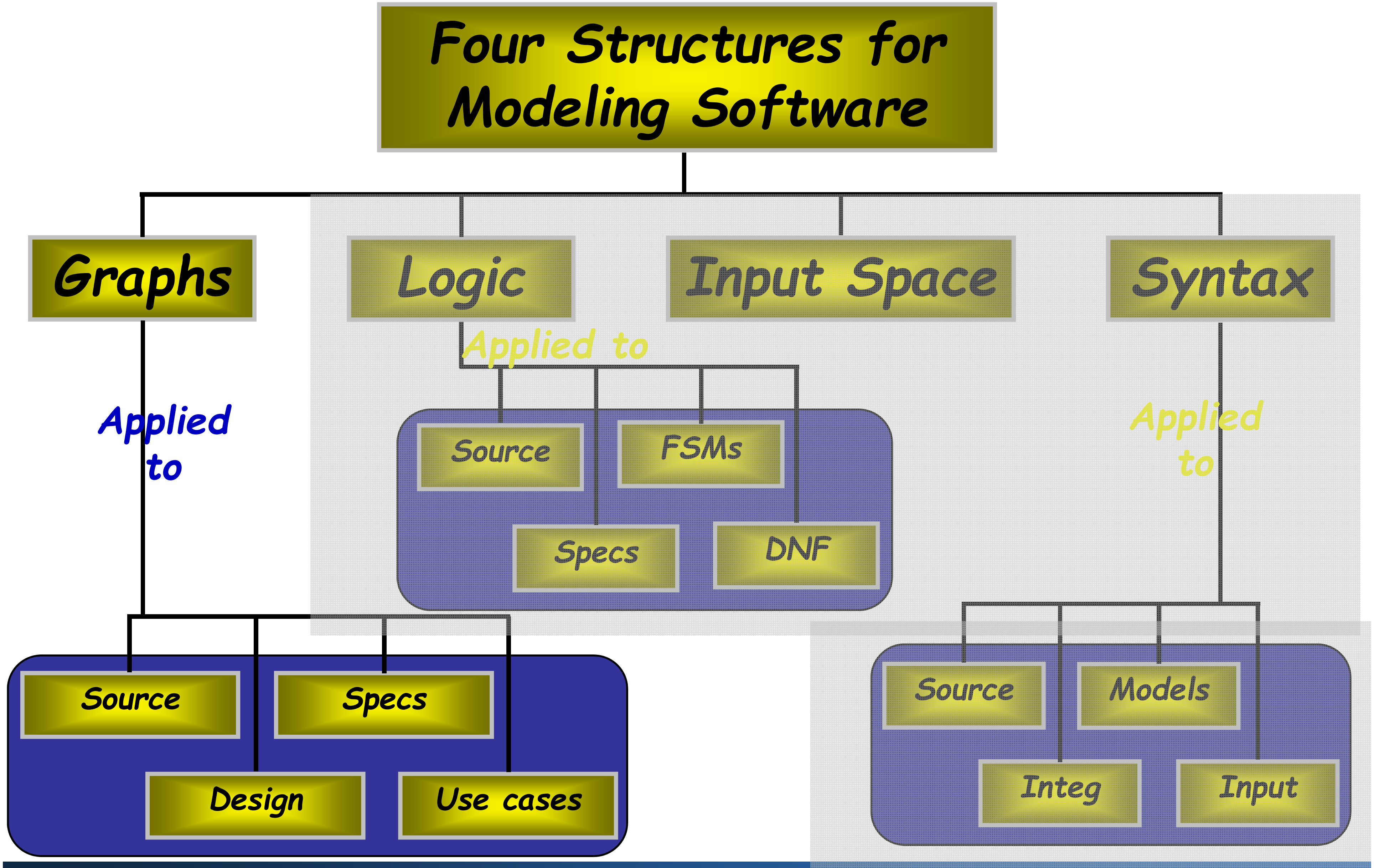
Need to Abstract Testing

- As we have seen, we can't try all possible executions of a program

- How can we *measure* "how much testing" we have done and look for more things to test?
  - Could talk about modules we have and have not tested, or use cases explored
  - Could also talk *structurally* – what aspects of the *source code* have we tested?

Graph Coverage

- *Cover all the nodes, edges, or paths of some graph related to the program*
- *Examples:*
  - *Statement coverage*
  - *Branch coverage*
  - *Path coverage*
  - *Data flow (def-use) coverage*
  - *Model-based testing coverage*
  - *Many more – most common kind of coverage, by far*

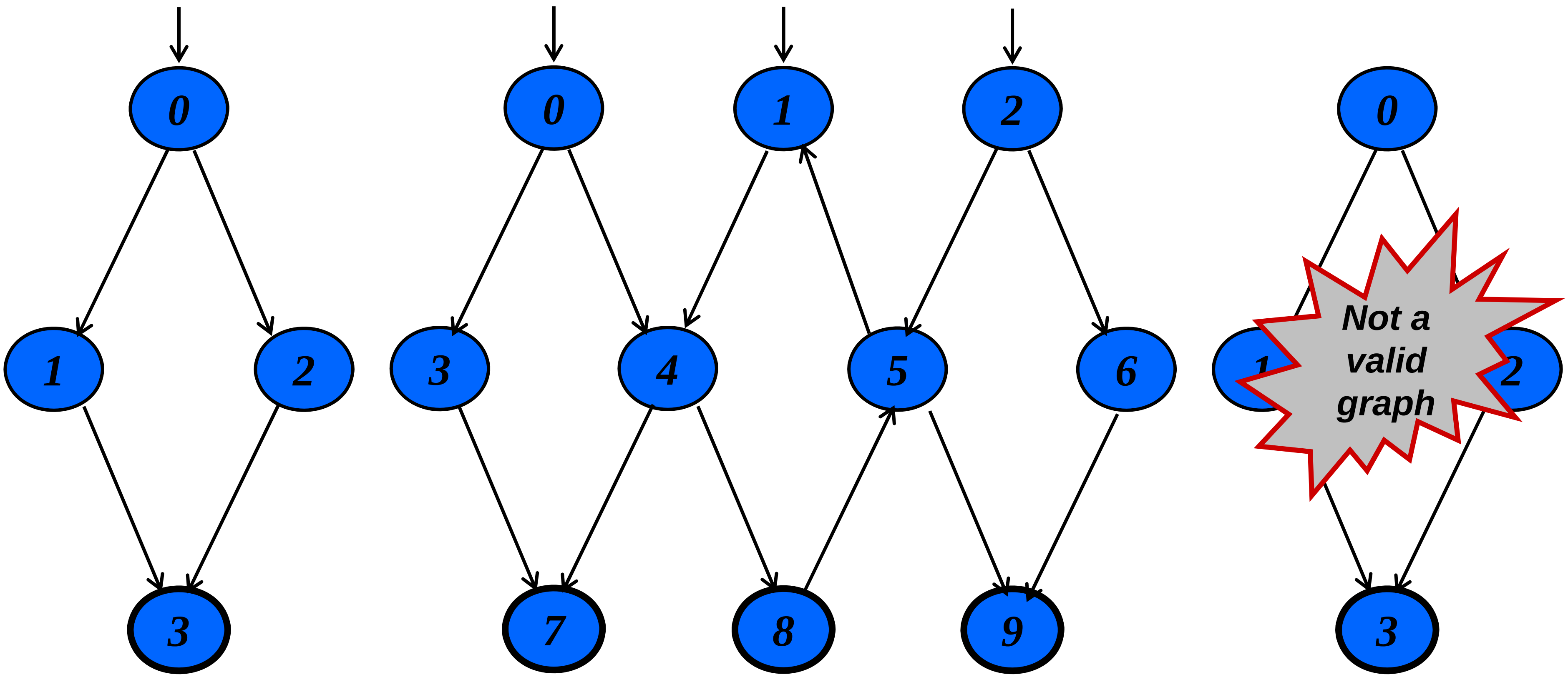

Four Structures for Modeling Software
Graphs
Logic
Input Space
Syntax
Applied to
Source
FSMs
Specs
DNF
Applied to
Applied to
Source
Specs
Design
Use cases
Source
Models
Integ
Input

- **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



$N_0 = \{ 0 \}$

$N_f = \{ 3 \}$

$N_0 = \{ 0, 1, 2 \}$

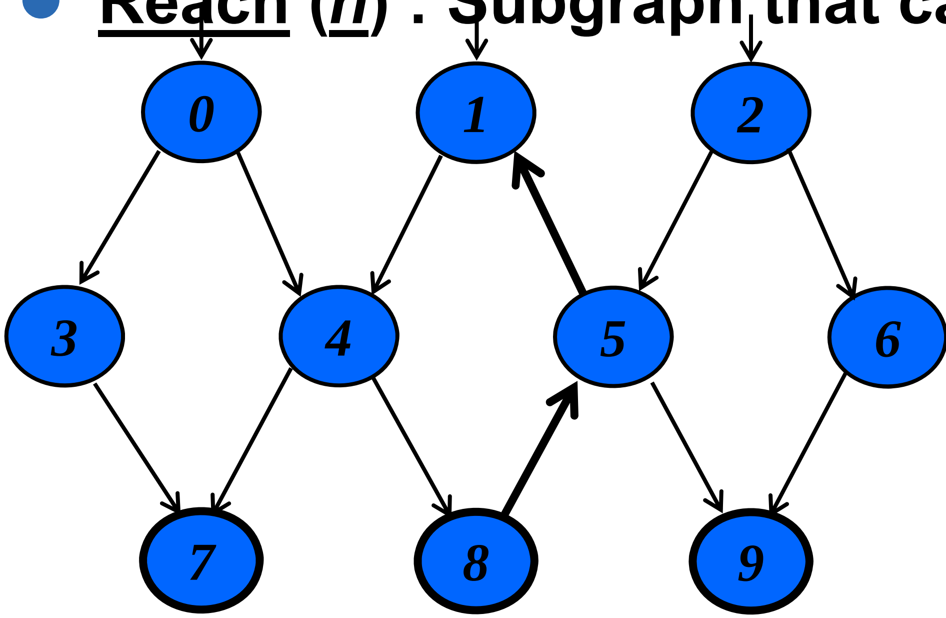
$N_f = \{ 7, 8, 9 \}$

$N_0 = \{ \}$

$N_f = \{ 3 \}$

Paths in Graphs

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



<u>A Few Paths</u>

[0, 3, 7]

[1, 4, 8, 5, 1]

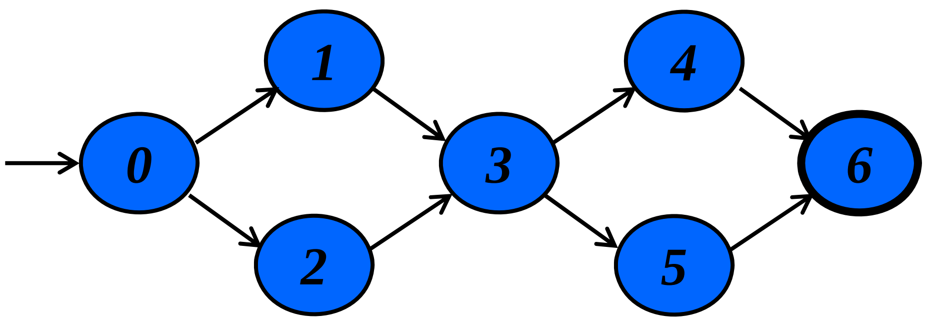
[2, 6, 9]

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

Reach ({0, 2}) = G

Test Paths and SESEs

- <u>Test Path</u> : 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 <u>any</u> tests
- <u>SESE graphs</u> : All test paths start at a single node and end at another node
  - Single-entry, single-exit
  - N0 and Nf have exactly one node



<u>Double-diamond graph</u>

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

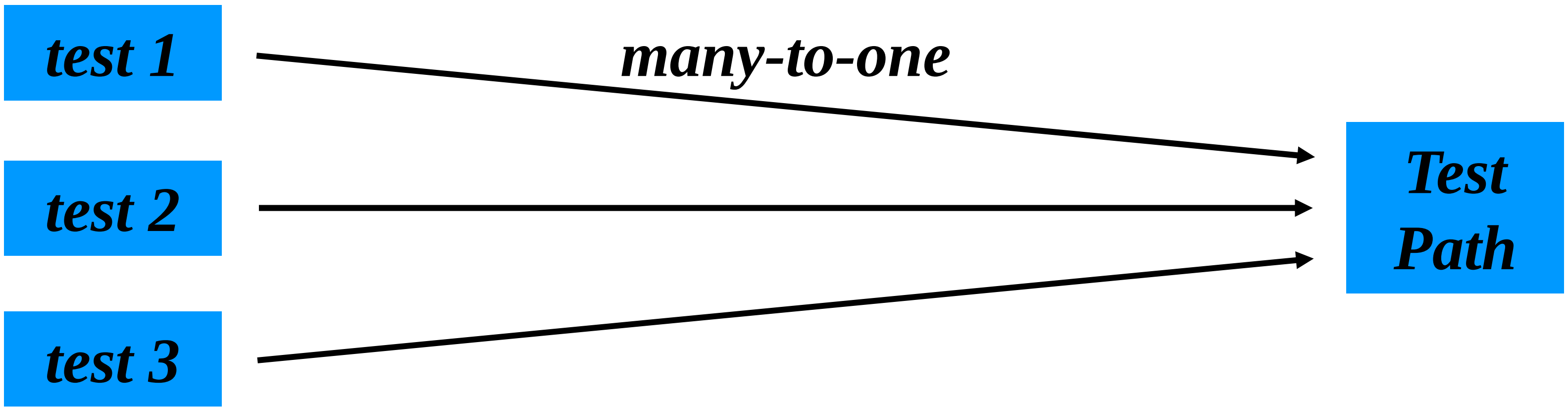
Visits edges (0, 1), (1, 3), (3, 4), (4, 6)

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

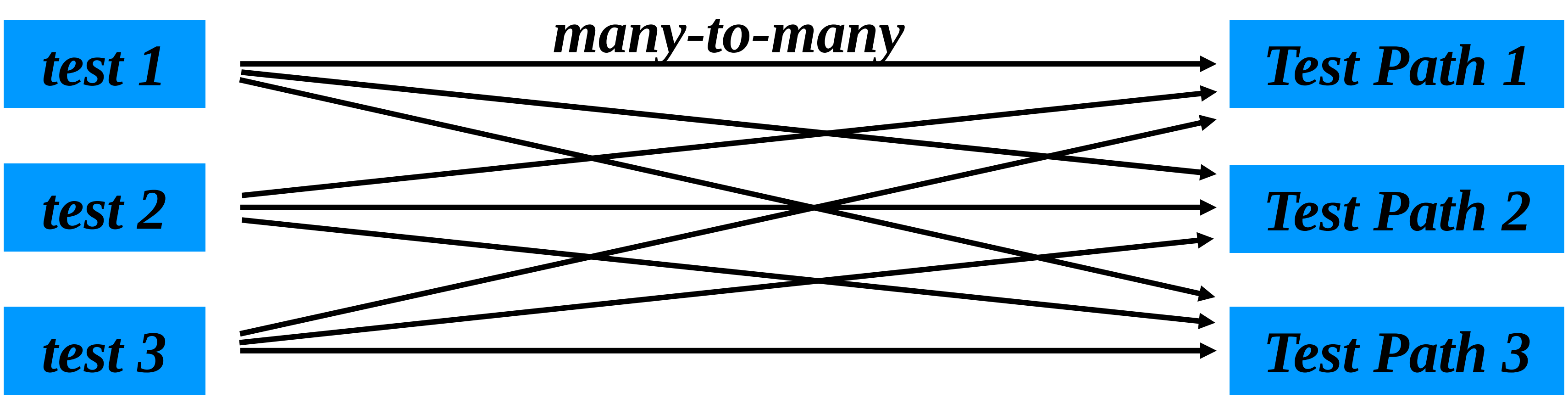
Tests and Test Paths

- **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



Deterministic software – a test always executes the same test path



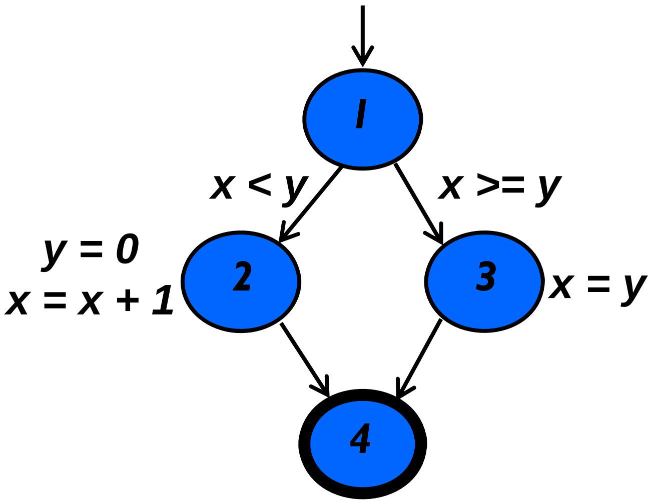
Non-deterministic software – a test can execute different test paths

Control Flow Graphs

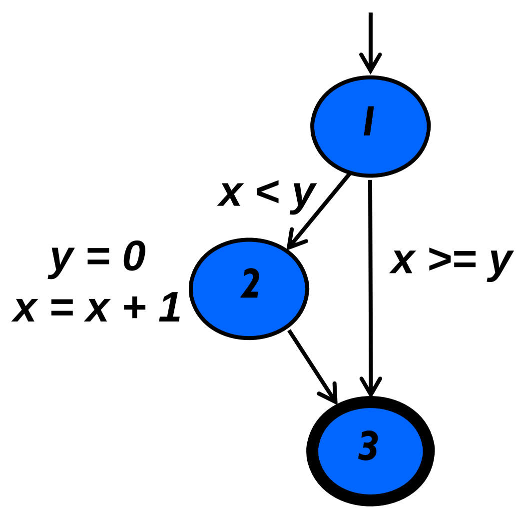
- A CFG models all executions of a method by describing control structures
- Nodes : Statements or sequences of statements (basic blocks)
- Edges : Transfers of control
- Basic Block : A sequence of statements such that if the first statement is executed, all statements will be (no branches)
- CFGs are 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;
}
```

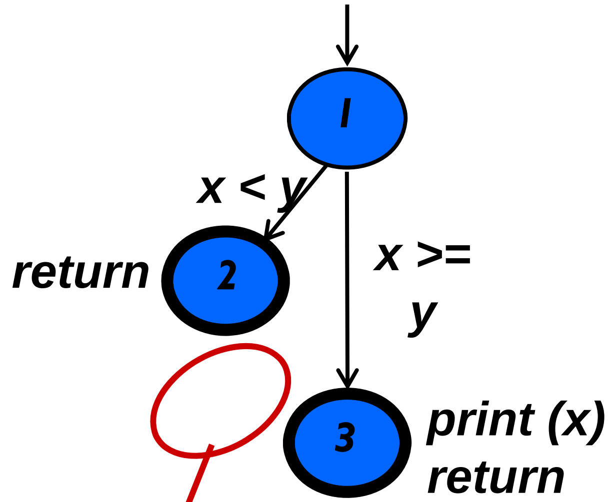



```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```



CFG : The if-Return Statement

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



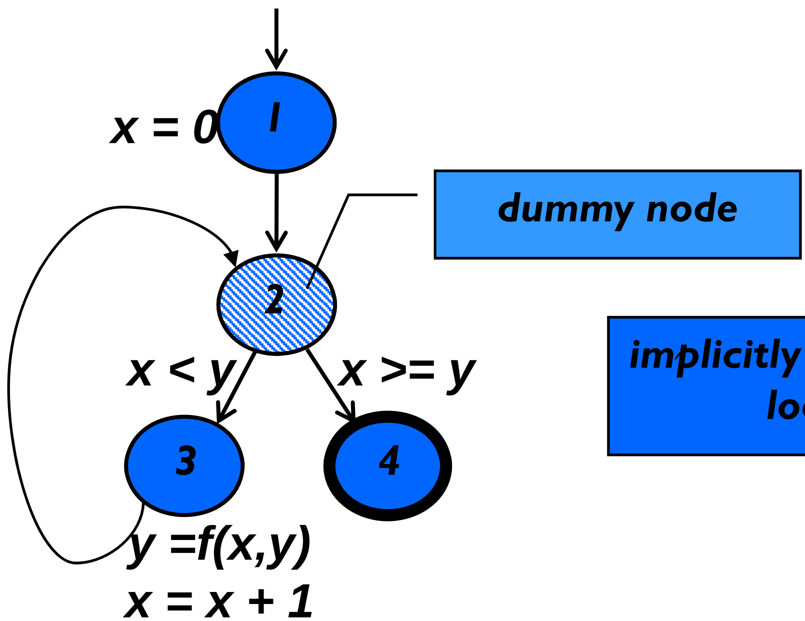
No edge from node 2 to 3.
The return nodes must be distinct.

Loops

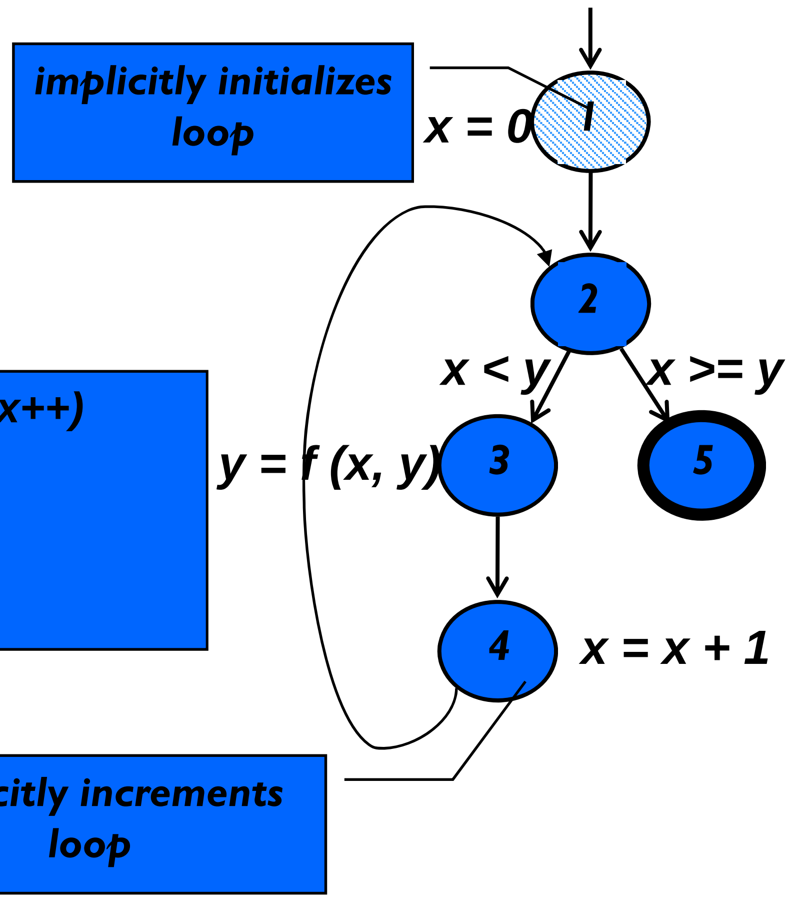
- **Loops require "*extra*" nodes to be added**
- **Nodes that do not represent statements or basic blocks**

CFG : while and for Loops

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

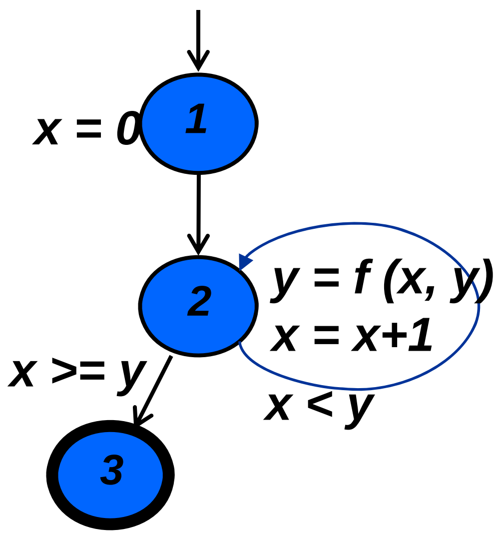



```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
```

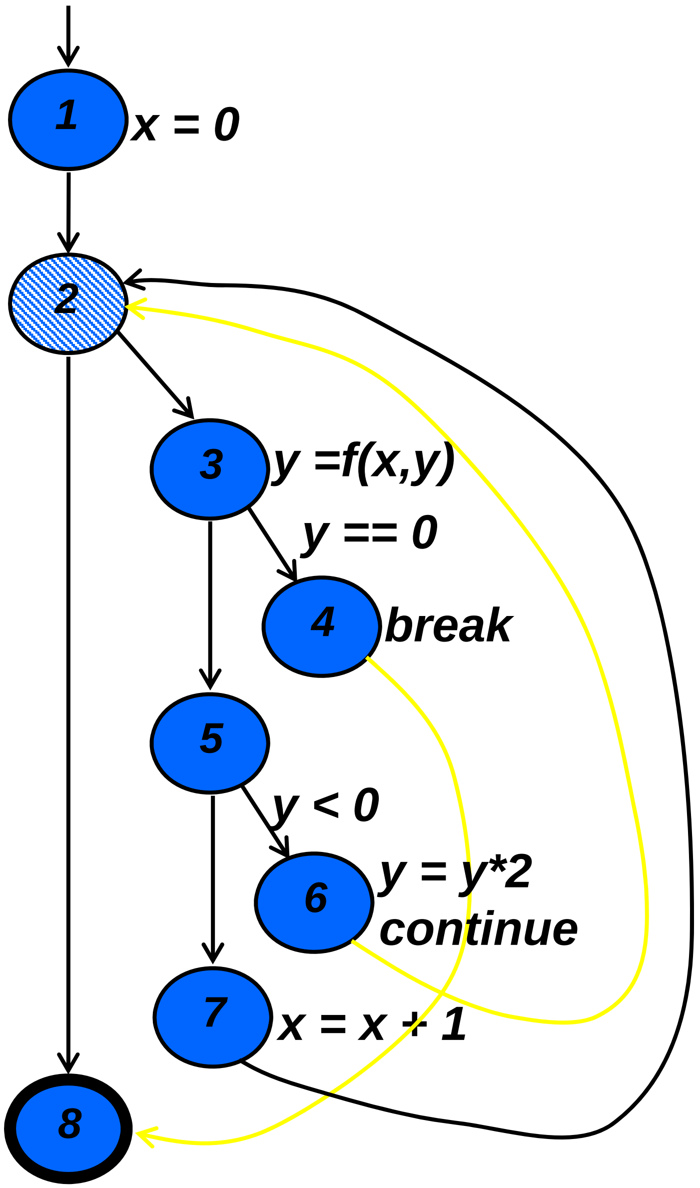


CFG : do Loop, break and continue

```
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);
```