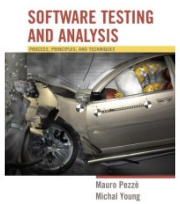
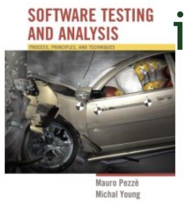
Finite Models



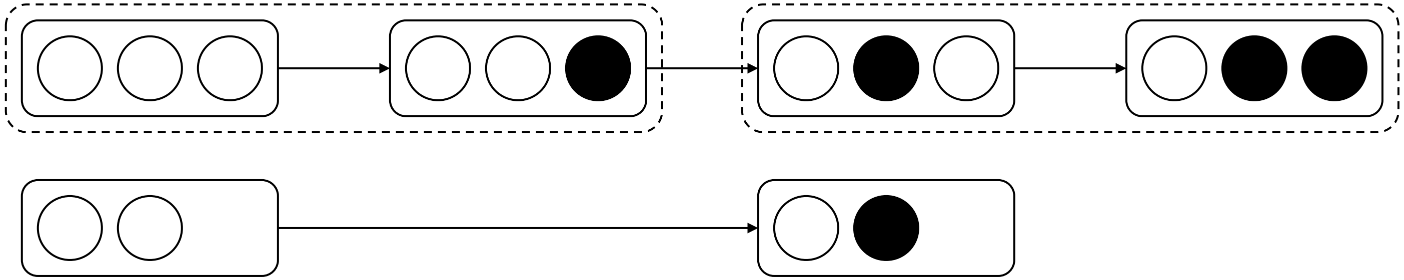
Properties of Models

- **Compact:** representable and manipulable in a reasonably compact form
  – What is *reasonably compact* depends largely on how the model will be used
- **Predictive:** must represent some salient characteristics of the modeled artifact well enough to distinguish between *good* and *bad* outcomes of analysis
  – no single model represents all characteristics well enough to be useful for all kinds of analysis
- **Semantically meaningful:** it is usually necessary to interpret analysis results in a way that permits diagnosis of the causes of failure
- **Sufficiently general:** models intended for analysis of some important characteristic must be general enough for practical use in the intended domain of application



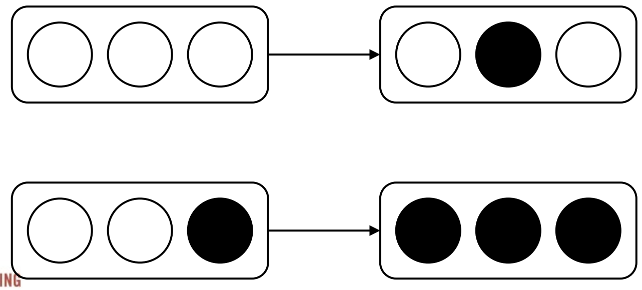
Finite Abstraction of Behavior

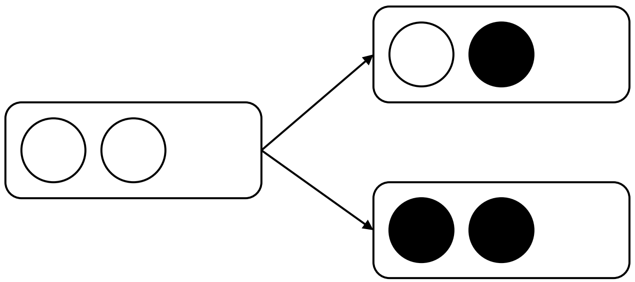
an abstraction function suppresses some details of program execution

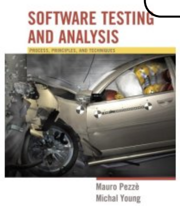


⇒

it lumps together execution states that differ with respect to the suppressed details but are otherwise identical



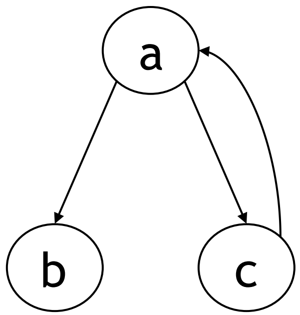


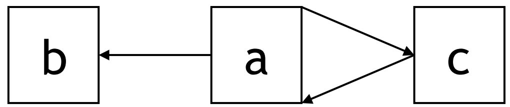


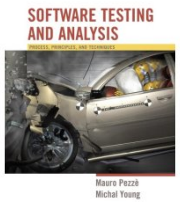
Graph Representations: directed graphs

- Directed graph:
  - N (set of nodes)
  - E (relation on the set of nodes) edges

Nodes: {a, b, c}
Edges: {(a,b), (a, c), (c, a)}

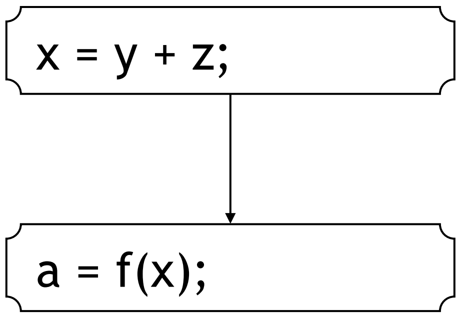






Graph Representations: labels and code

- We can label nodes with the names or descriptions of the entities they represent.
  - If nodes a and b represent program regions containing assignment statements, we might draw the two nodes and an edge (a,b) connecting them in this way:

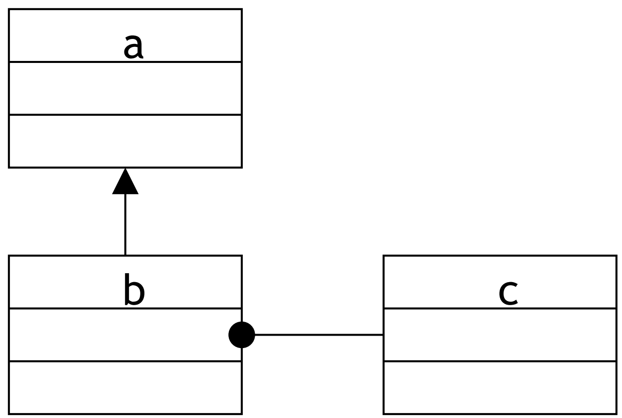


Multidimensional Graph Representations

- Sometimes we draw a single diagram to represent more than one directed graph, drawing the shared nodes only once
  - class B extends (is a subclass of) class A
  - class B has a field that is an object of type C

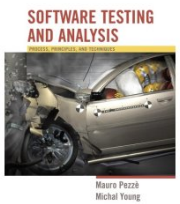
extends relation
NODES = {A, B, C}
EDGES = {(A,B)}

includes relation
NODES = {A, B, C}
EDGES = {(B,C)}



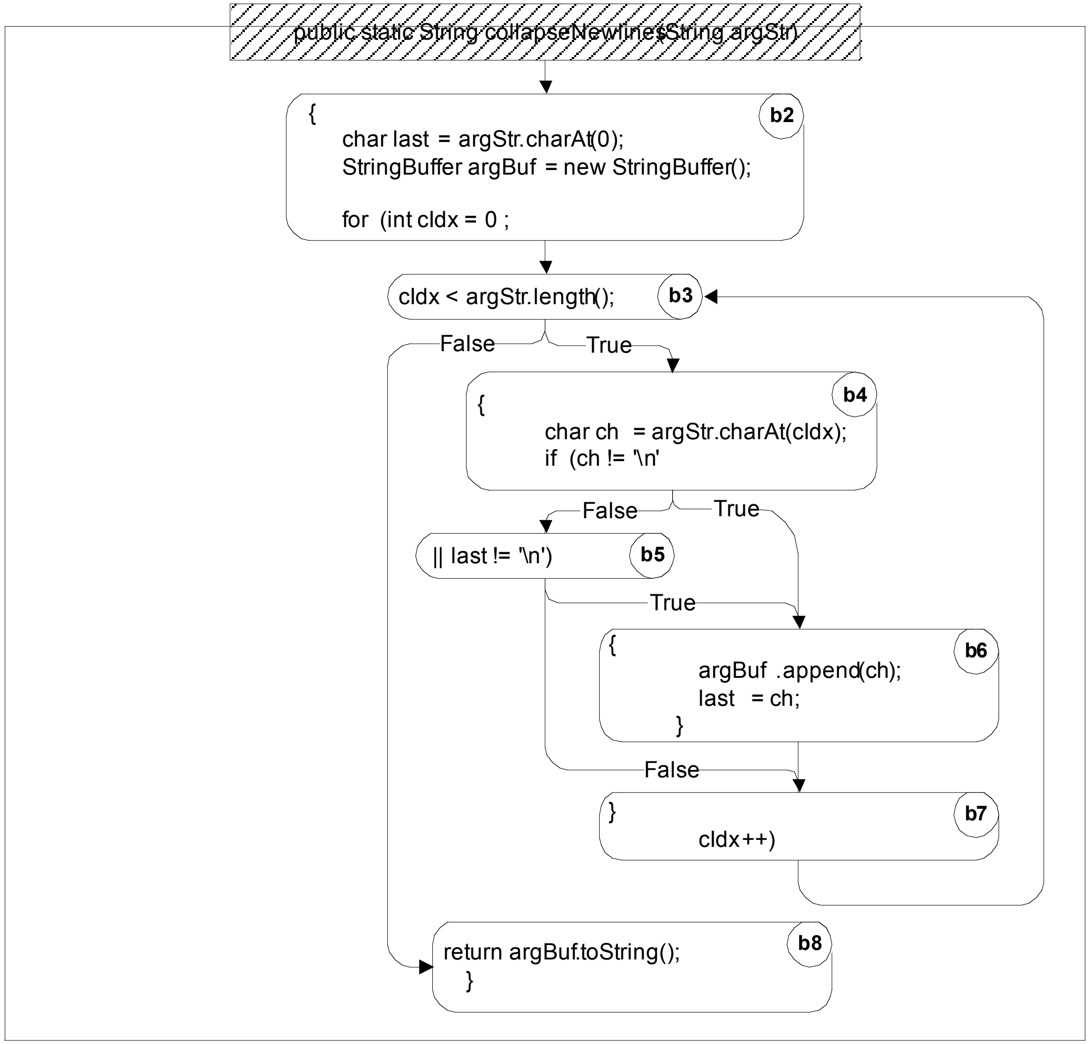
(Intraprocedural) Control Flow Graph

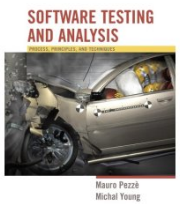
- nodes = regions of source code (basic blocks)
  - Basic block = maximal program region with a single entry and single exit point
  - Often statements are grouped in single regions to get a compact model
  - Sometime single statements are broken into more than one node to model control flow within the statement
- directed edges = possibility that program execution proceeds from the end of one region directly to the beginning of another



Example of Control Flow Graph

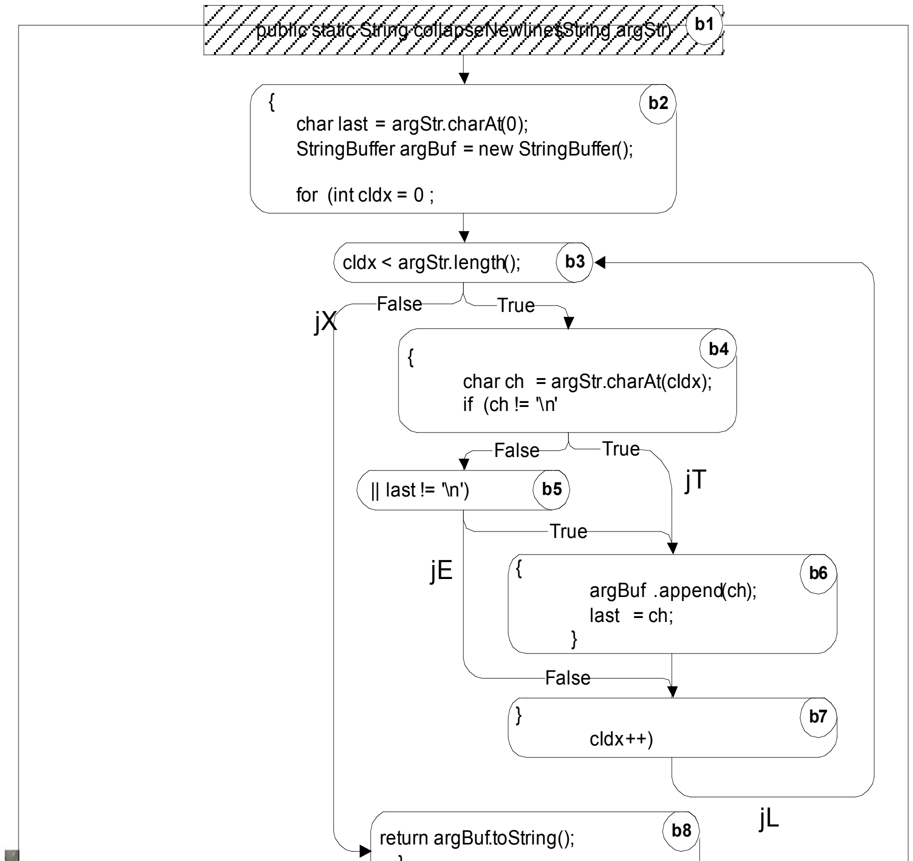
```
public static String collapseNewlines(String argStr)
  {
      char last = argStr.charAt(0);
      StringBuffer argBuf = new StringBuffer();

      for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++)
      {
         char ch = argStr.charAt(cIdx);
         if (ch != '\n' || last != '\n')
         {
            argBuf.append(ch);
            last = ch;
         }
      }

      return argBuf.toString();
  }
```





Linear Code Sequence and Jump (LCSJ)

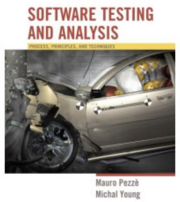
Essentially subpaths of the control flow graph from one branch to another



From	Sequence of basic blocs	To
Entry	b1 b2 b3	jX
Entry	b1 b2 b3 b4	jT
Entry	b1 b2 b3 b4 b5	jE
Entry	b1 b2 b3 b4 b5 b6 b7	jL
jX	b8	ret
jL	b3 b4	jT
jL	b3 b4 b5	jE
jL	b3 b4 b5 b6 b7	jL

Interprocedural control flow graph

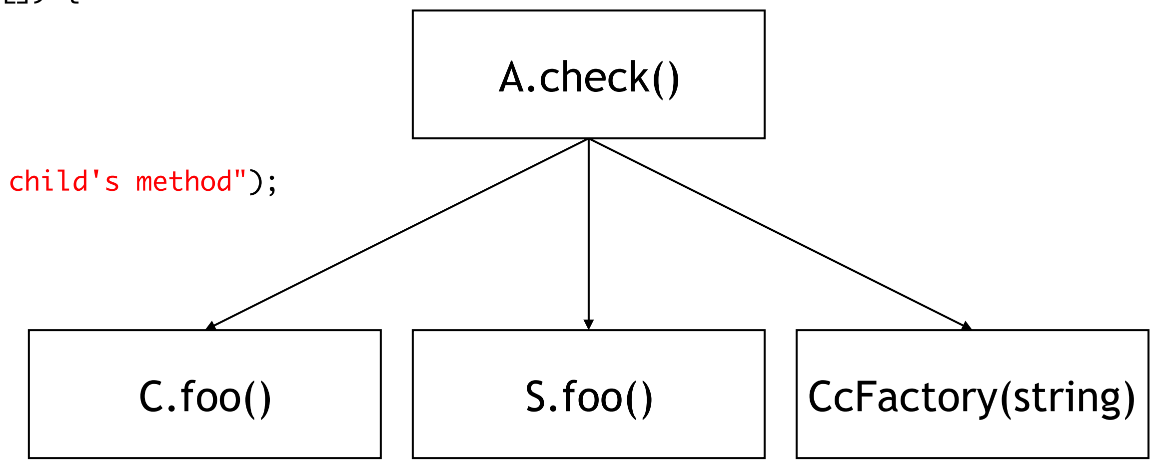
- Call graphs
  - Nodes represent procedures
    - Methods
    - C functions
    - …
  - Edges represent *calls* relation

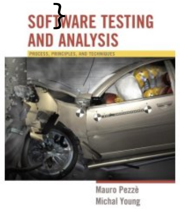


Overestimating the *calls* relation

The static call graph includes calls through dynamic bindings that never occur in execution.

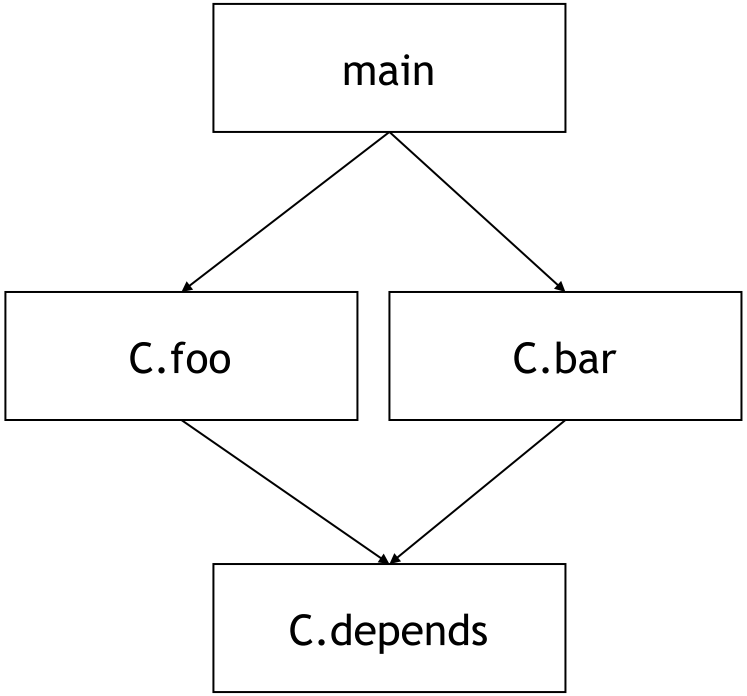
```
public class C {
    public static C cFactory(String kind) {
     if (kind == "C") return new C();
     if (kind == "S") return new S();
     return null;
    }
    void foo() {
     System.out.println("You called the parent's method");
    }
    public static void main(String args[]) {
     (new A()).check();
    }
}
class S extends C {
    void foo() {
     System.out.println("You called the child's method");
    }
}
class A {
    void check() {
     C myC = C.cFactory("S");
     myC.foo();
    }
}
```

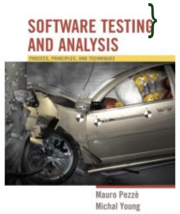




Contex Insensitive Call graphs

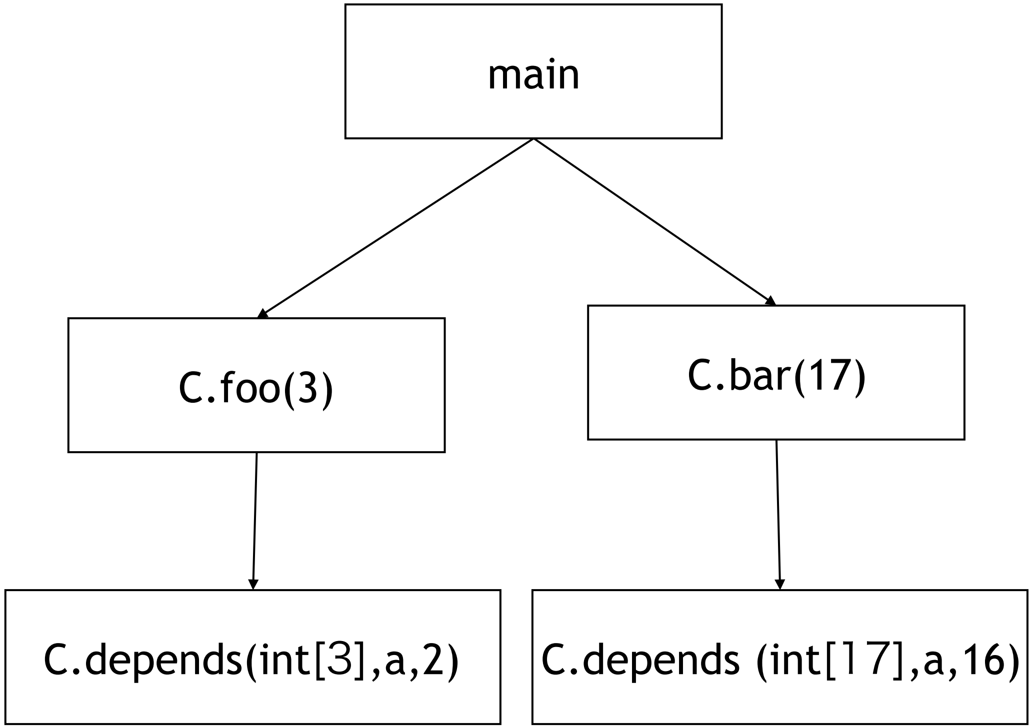
```
public class Context {
   public static void main(String args[]) {
      Context c = new Context();
      c.foo(3);
      c.bar(17);
   }

   void foo(int n) {
      int[]  myArray = new int[ n ];
      depends( myArray, 2) ;
   }

   void bar(int n) {
      int[]  myArray = new int[ n ];
      depends( myArray, 16) ;
   }

   void depends( int[] a, int n ) {
      a[n] = 42;
   }
}
```

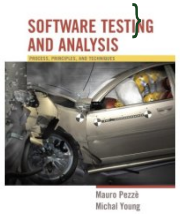




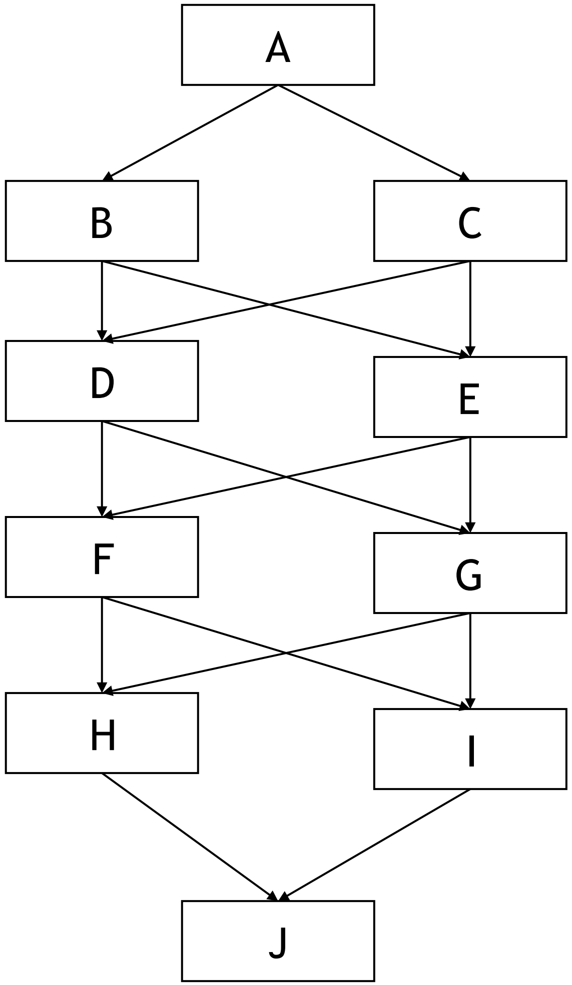
Contex Sensitive Call graphs

```
public class Context {
    public static void main(String args[]) {
        Context c = new Context();
        c.foo(3);
        c.bar(17);
    }

    void foo(int n) {
        int[]  myArray = new int[ n ];
        depends( myArray, 2) ;
    }

    void bar(int n) {
        int[]  myArray = new int[ n ];
        depends( myArray, 16) ;
    }

    void depends( int[] a, int n ) {
        a[n] = 42;
    }
}
```





Context Sensitive CFG exponential growth



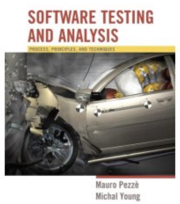
1 context A

2 contexts AB AC

4 contexts ABD ABE ACD ACE

8 contexts …

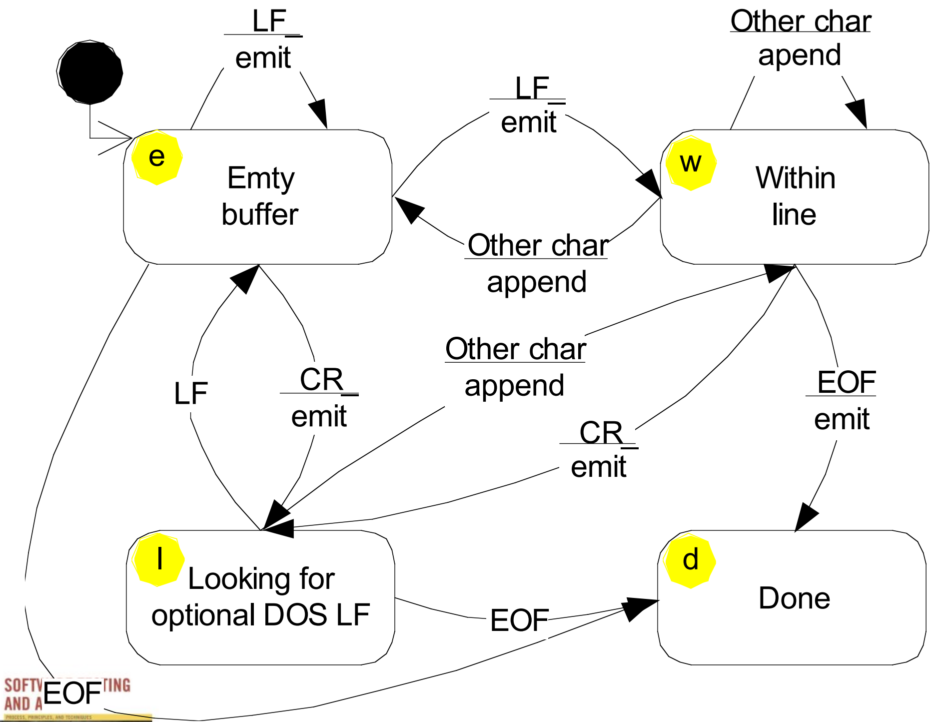
16 calling contexts …

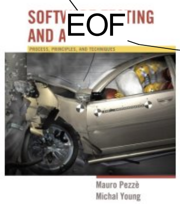


Finite state machines

- finite set of states (nodes)
- set of transitions among states (edges)

Graph representation (Mealy machine)

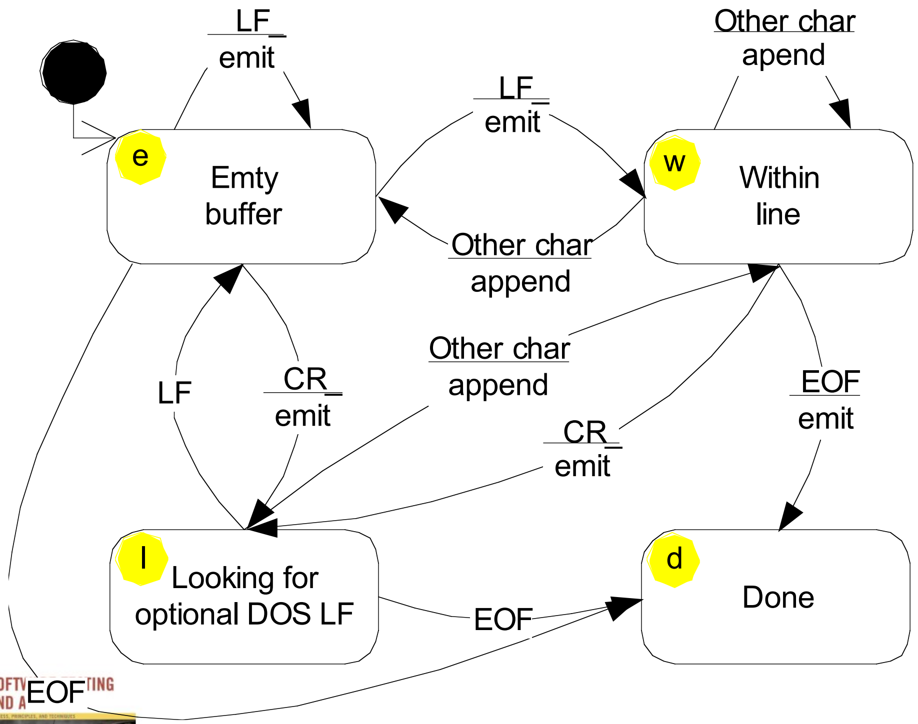




Finite state machines

- finite set of states (nodes)
- set of transitions among states (edges)

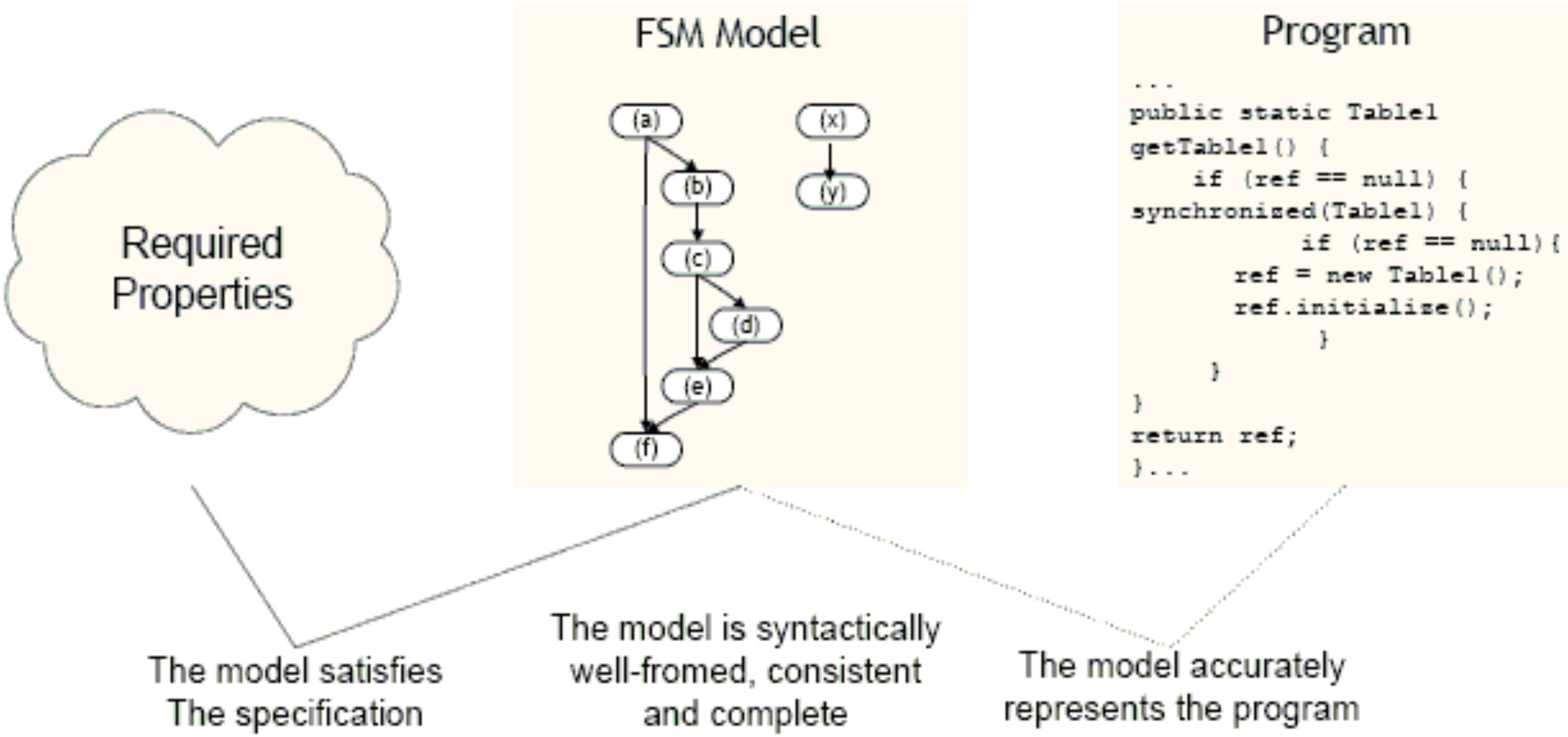
Graph representation (Mealy machine)

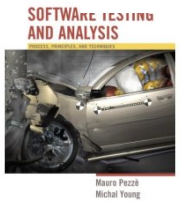


Tabular representation

	LF	CR	EOF	other
e	e/emit	l/emit	d/-	w/append
w	e/emit	l/emit	d/emit	w/append
l	e/-		d/-	w/append

Using Models to Reason about System Properties





```
1   /** Convert each line from standard input */
2   void transduce() {
3
4     #define BUFLEN 1000
5     char buf[BUFLEN];   /* Accumulate line into this buffer  */
6     int   pos = 0;          /* Index for next character in buffer */
7
8     char inChar; /* Next character from input */
9
10    int atCR = 0; /* 0="within line", 1="optional DOS LF" */
11
12    while ((inChar = getchar()) != EOF ) {
13      switch (inChar) {
14      case LF:
15        if (atCR) {    /* Optional DOS LF */
16          atCR = 0;
17        } else {         /* Encountered CR within line */
18          emit(buf, pos);
19          pos = 0;
20        }
21        break;
22      case CR:
23        emit(buf, pos);
24        pos = 0;
25        atCR = 1;
26        break;
27      default:
28        if (pos >= BUFLEN-2) fail("Buffer overflow");
29        buf[pos++] = inChar;
30      } /* switch */
31    }
32    if (pos > 0) {
33      emit(buf, pos);
34    }
35  }
```

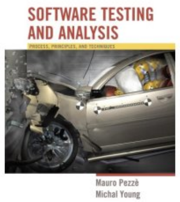
Abstraction Function

Abstract state	Concrete state		
	Lines	atCR	pos
e (Empty buffer)	3 – 13	0	0
w (Within line)	13	0	> 0
l (Looking for LF)	13	1	0
d (Done)	36	–	–

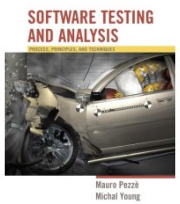
	LF	CR	EOF	other
e	e / emit	l / emit	d / –	w / append
w	e / emit	l / emit	d / emit	w / append
l	e / –	l / emit	d / –	w / append

Summary

- Models must be much simpler than the artifact they describe to be understandable and analyzable
- Must also be sufficiently detailed to be useful
- CFG are built from software
- FSM can be built before software to documentintended behavior

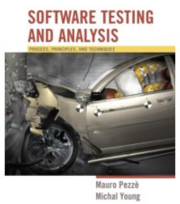


Dependence and Data Flow Models



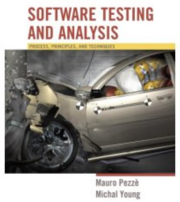
Why Data Flow Models?

- From models emphasizing control…
    - Control flow graph, call graph, finite state machines
- … to those enable reasoning about dependence
    - Where does this value of x come from?
    - What would be affected by changing this?
    - …
- Many program analyses and test design techniques use data flow information
  - Often in combination with control flow
    - Example: “Taint” analysis to prevent SQL injection attacks
    - Example: Dataflow test criteria (later)



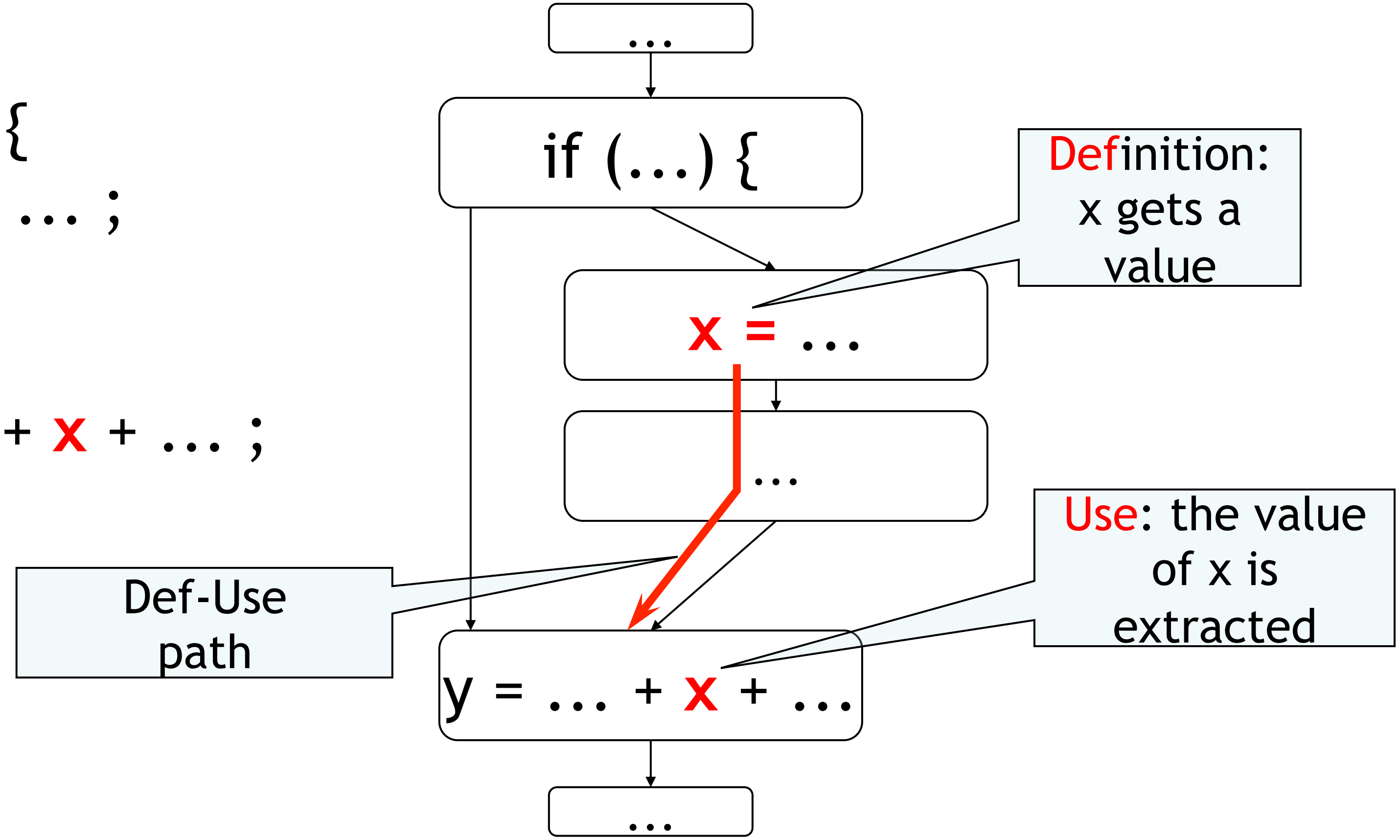
Def-Use Pairs (1)

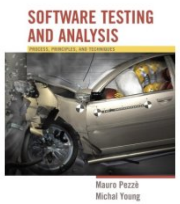
- A **def-use (du) pair** associates a point in a program where a value is produced with a point where it is used
- **Definition**: where a variable gets a value
  - Variable declaration (often the special value “uninitialized”)
  - Variable initialization
  - Assignment
  - Values received by a parameter
- **Use**: extraction of a value from a variable
  - Expressions
  - Conditional statements
  - Parameter passing
  - Returns



Def-Use Pairs (2)

```
...
if (...) {
    x = ... ;
...
}
y = ... + x + ... ;
```





Def-Use Pairs (3)

```
/**  Euclid's algorithm */
public class GCD
{
public int gcd(int x, int y) {
    int tmp;               // A: def x, y, tmp
    while (y != 0) {       // B: use y
        tmp = x % y;       // C: def tmp; use x, y
        x = y;             // D: def x; use y
        y = tmp;           // E: def y; use tmp
    }
    return x;              // F: use x
  }
```

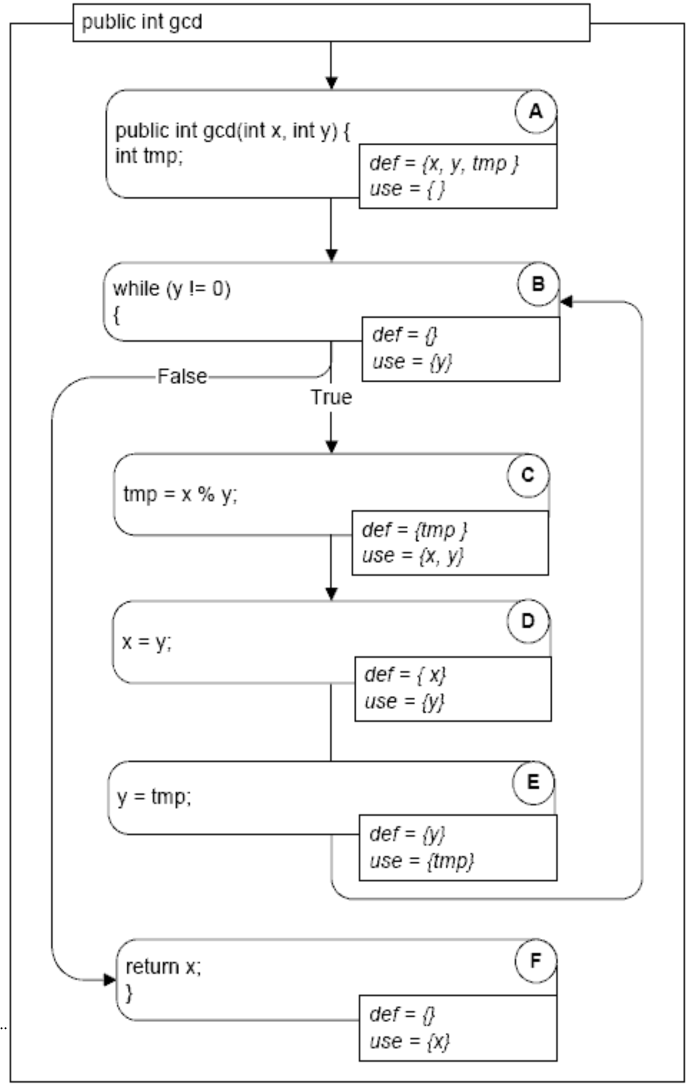
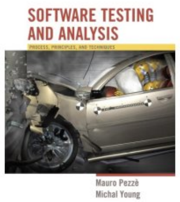
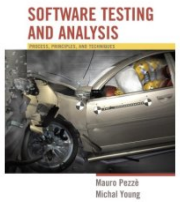


Figure 6.2, page 79



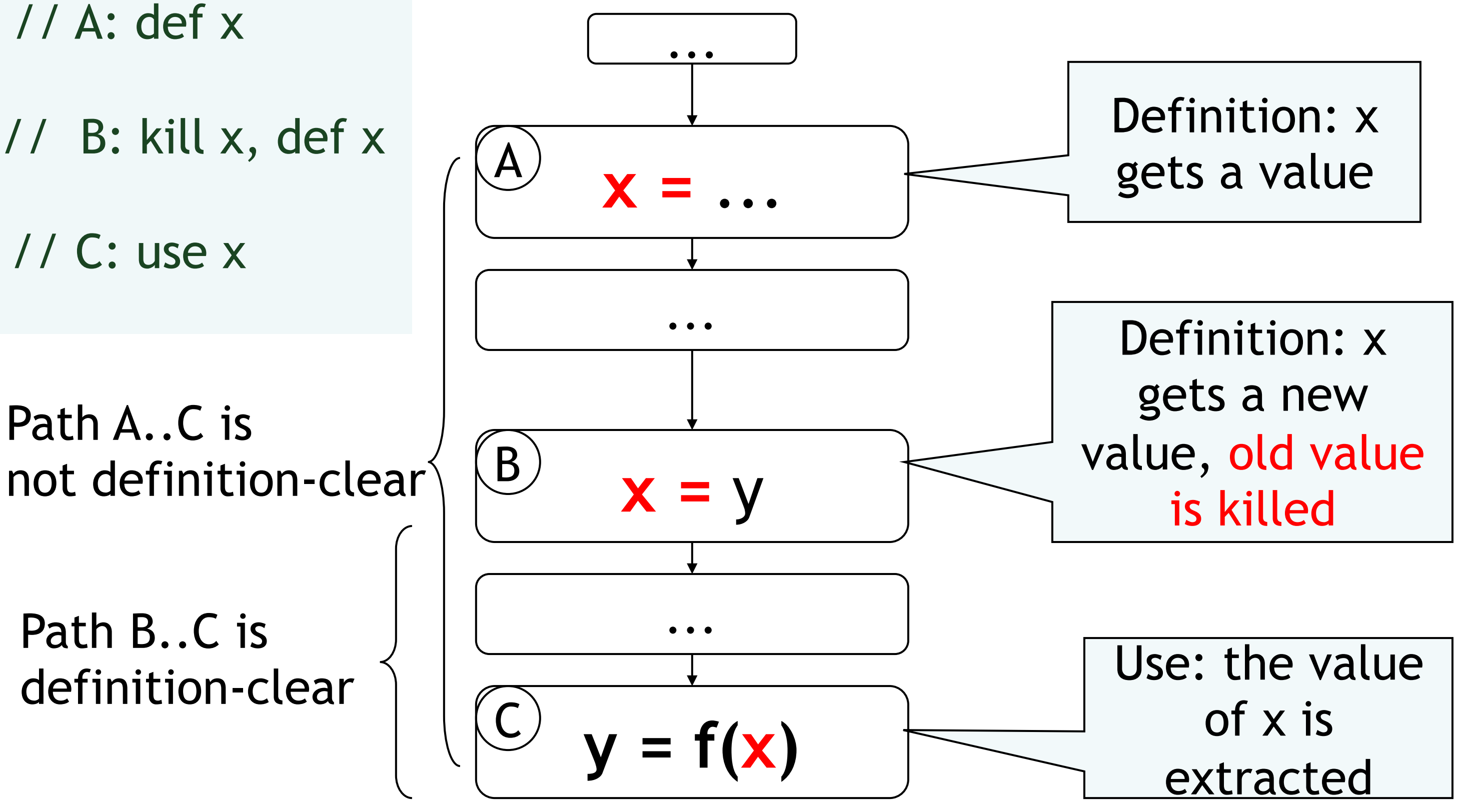
Def-Use Pairs (4)

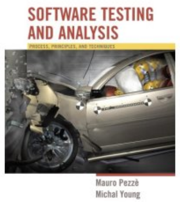
- A **definition-clear** path is a path along the CFG from a definition to a use of the same variable without another definition of the variable between
  - If, instead, another definition is present on the path, then the latter definition **kills** the former
- A def-use pair is formed if and only if there is a definition-clear path between the definition and the use



Definition-Clear or Killing

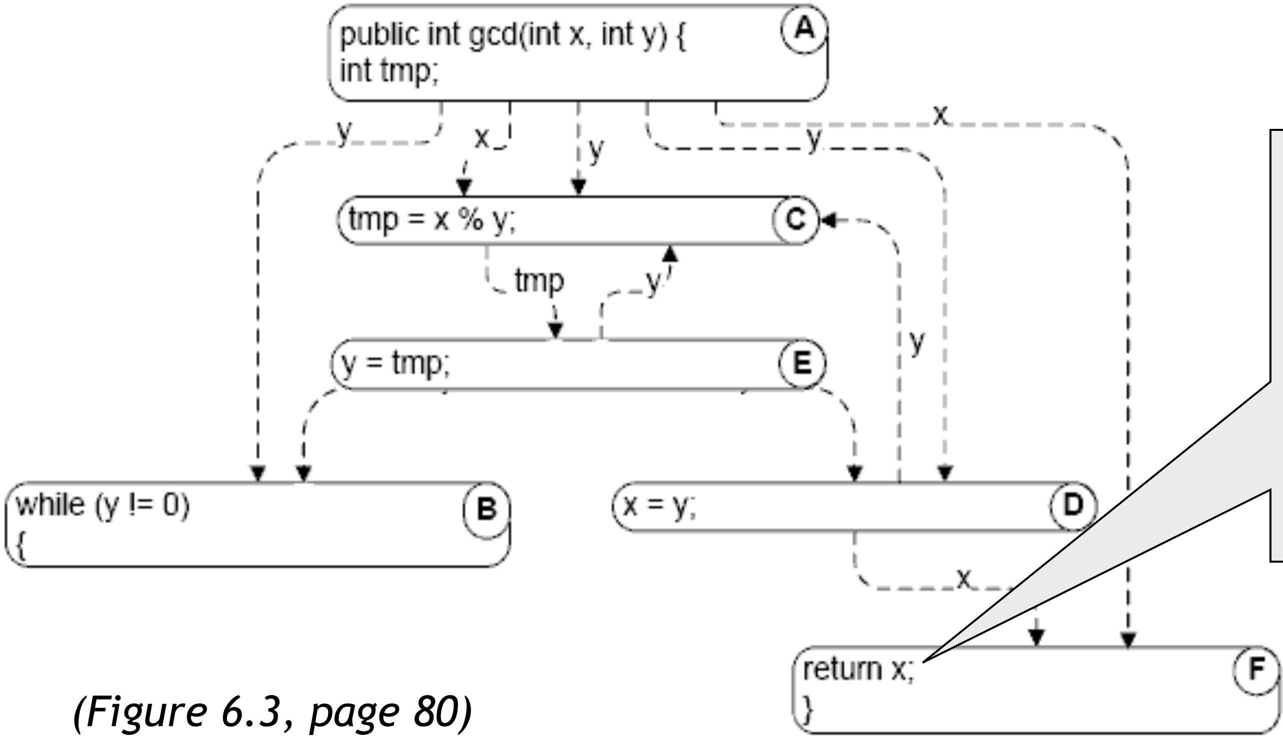
```
x = ...       // A: def x
q = ...
x = y;       //  B: kill x, def x
z = ...
y = f(x);   // C: use x
```



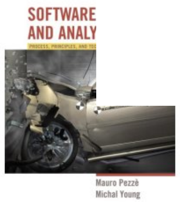


(Direct) Data Dependence Graph

- A direct data dependence graph is:
  - Nodes: as in the control flow graph (CFG)
  - Edges: def-use (du) pairs, labelled with the variable name



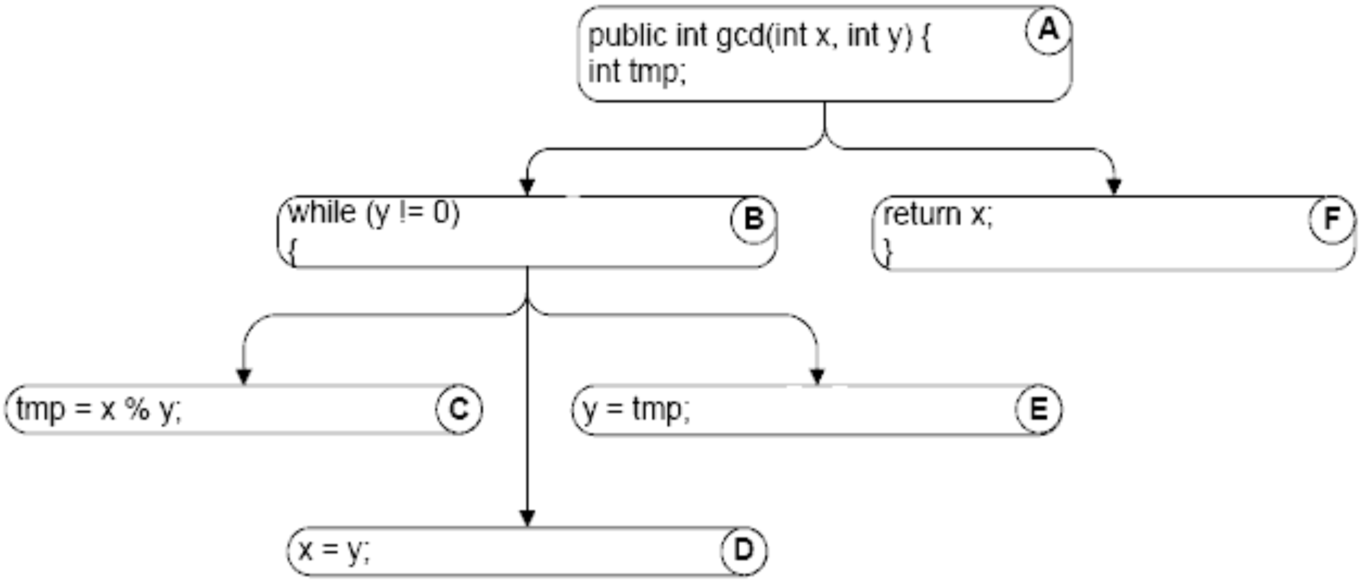
Dependence edges show this x value could be the unchanged parameter or could be set at line D

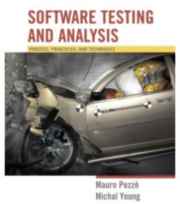


(Figure 6.3, page 80)

Control dependence (1)

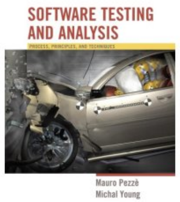
- Data dependence: Where did these values come from?
- Control dependence: Which statement controls whether this statement executes?
  - Nodes: as in the CFG
  - Edges: unlabelled, from entry/branching points to controlled blocks



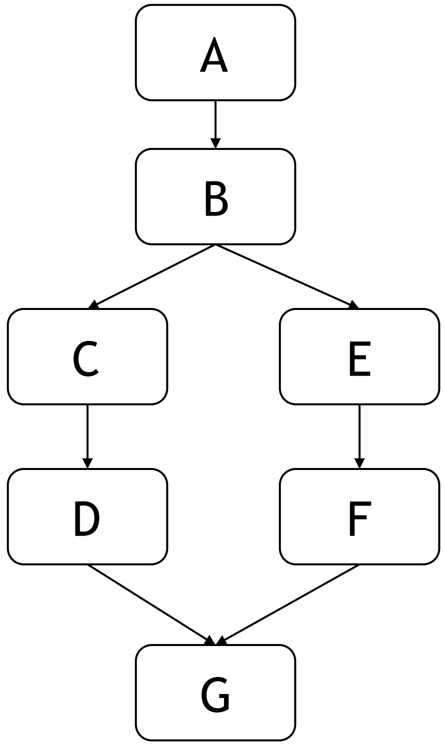


Dominators

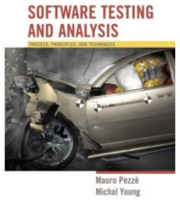
- **Pre-dominators** in a rooted, directed graph can be used to make this intuitive notion of "controlling decision" precise.
- Node M dominates node N if every path from the root to N passes through M.
  - A node will typically have many dominators, but except for the root, there is a unique **immediate dominator** of node N which is closest to N on any path from the root, and which is in turn dominated by all the other dominators of N.
  - Because each node (except the root) has a unique immediate dominator, the immediate dominator relation forms a tree.
- **Post-dominators**: Calculated in the reverse of the control flow graph, using a special "exit" node as the root.



Dominators (example)

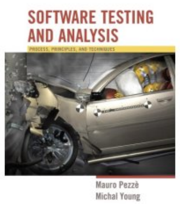


- A pre-dominates all nodes; G post-dominates all nodes
- F and G post-dominate E
- G is the immediate post-dominator of B
  - C does *not* post-dominate B
- B is the immediate pre-dominator of G
  - F does *not* pre-dominate G

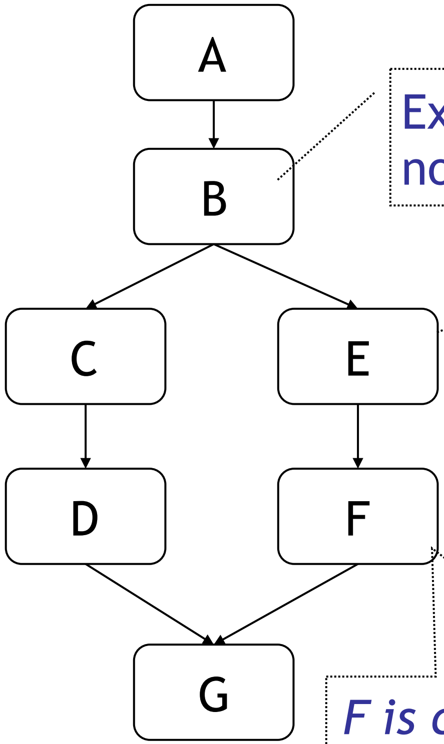


Control dependence (2)

- We can use post-dominators to give a more precise definition of control dependence:
  - Consider again a node N that is reached on some but not all execution paths.
  - There must be some node C with the following property:
    - C has at least two successors in the control flow graph (i.e., it represents a control flow decision);
    - C is not post-dominated by N
    - there is a successor of C in the control flow graph that is post-dominated by N.
  - When these conditions are true, we say node N is control-dependent on node C.
    - Intuitively: C was the last decision that controlled whether N executed



Control Dependence



Execution of F is not inevitable at B

Execution of F is inevitable at E

F is control-dependent on B, the last point at which its execution was not inevitable