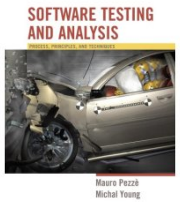
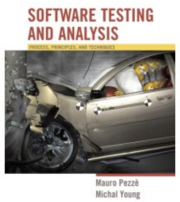
Dependence and Data Flow Models



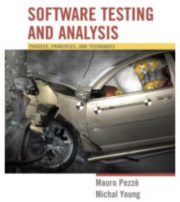
Why Data Flow Models?

- Many models emphasize control
    - Control flow graph, call graph, finite state machines
- We also need to reason about data 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 (Ch.13)



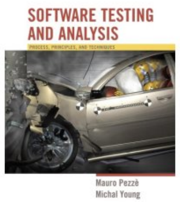
Learning objectives

- Understand basics of data-flow models and the related concepts (def-use pairs, dominators…)
- Understand some analyses that can be performed with the data-flow model of a program
  - The data flow analyses to build models
  - Analyses that use the data flow models
- Understand basic trade-offs in modeling data flow
  - variations and limitations of data-flow models and analyses, differing in precision and cost



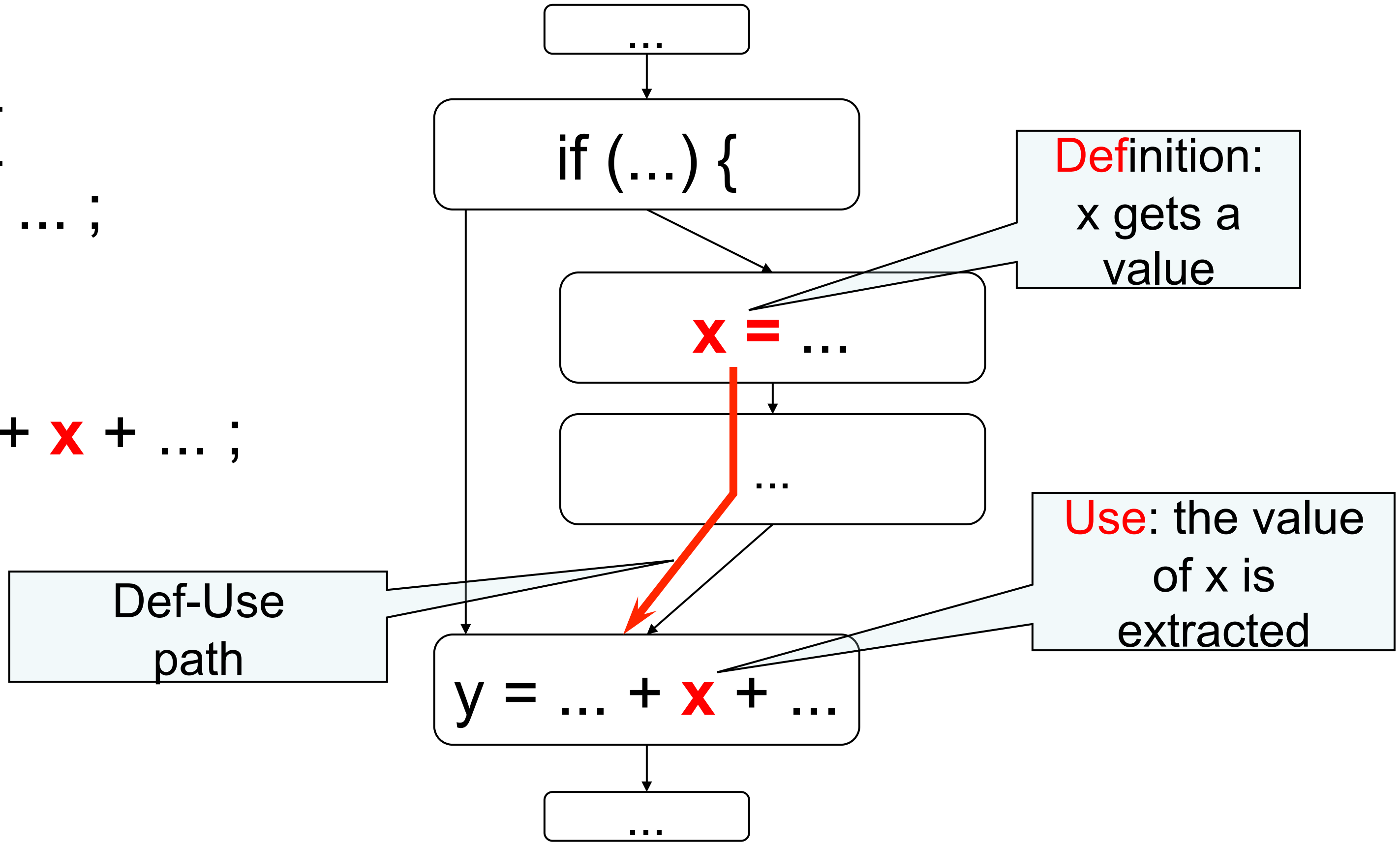
Def-Use Pairs (1)

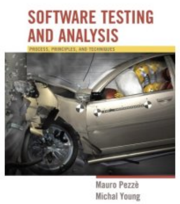
- A **def-use (du) pair** associates a point in a program where a value is produced with a point where it is used
- **Definition**: where a variable gets a value
  - Variable declaration (often the special value “uninitialized”)
  - Variable initialization
  - Assignment
  - Values received by a parameter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



Def-Use Pairs

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





Def-Use Pairs (2)

```
/**  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
  }
```

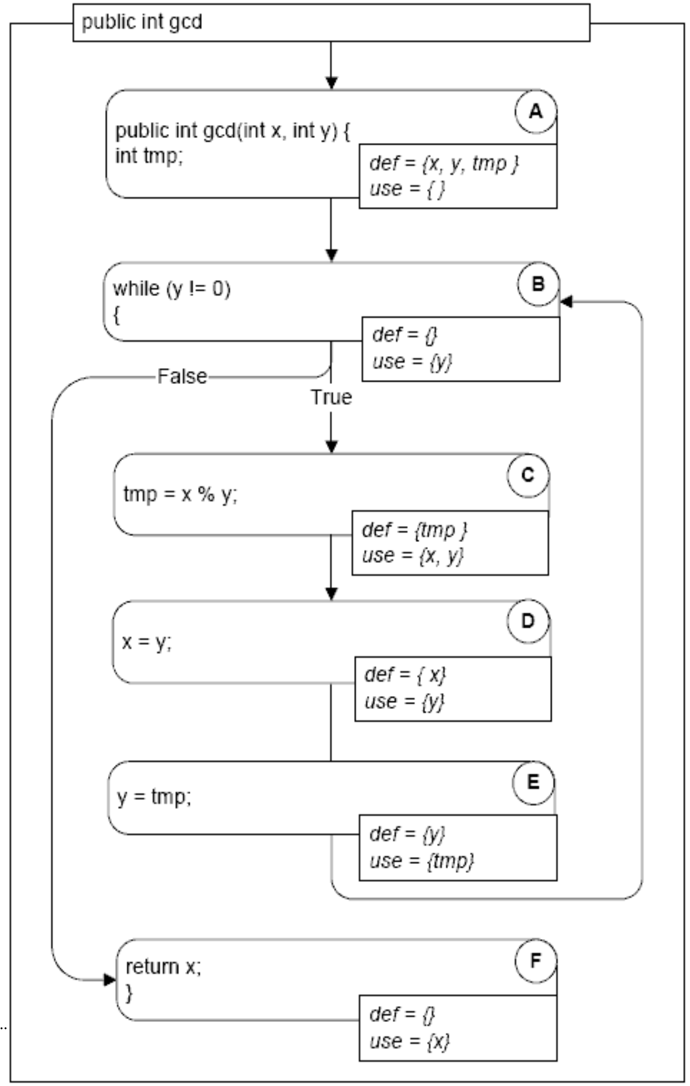
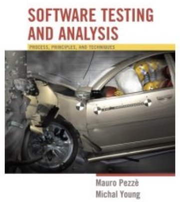


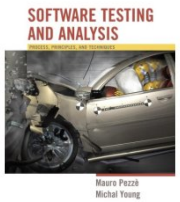
Figure 6.2, page 79



Question for class

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

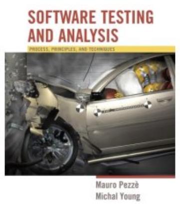
What are the def-use pairs involving x in this program fragment?



Def-Use Pairs (3)

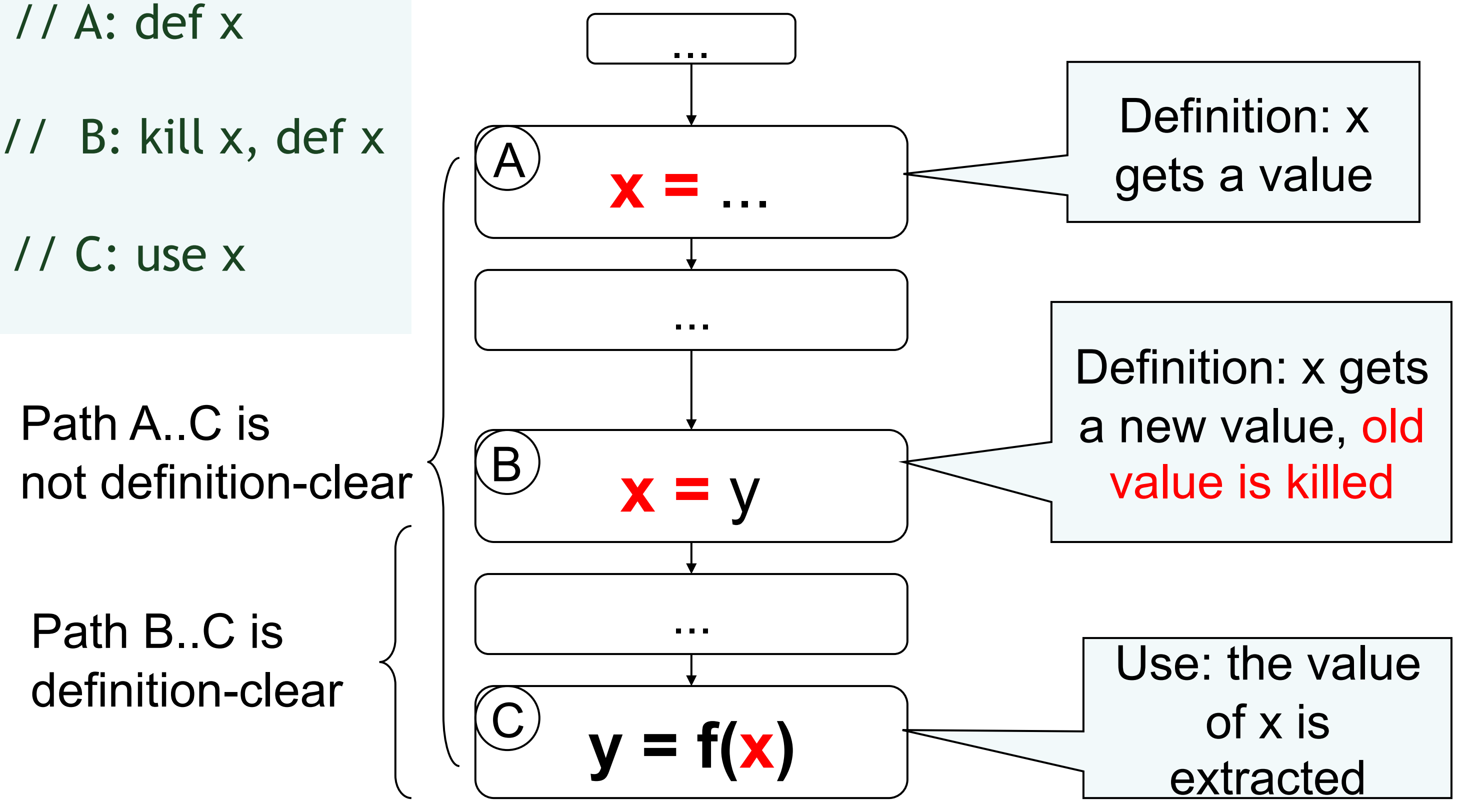
- A **definition-clear** path is a path along the CFG from a definition to a use of the same variable without* another definition of the variable between
  - If, instead, another definition is present on the path, then the latter definition **kills** the former
- A def-use pair is formed if and only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

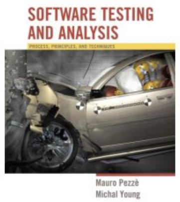
**There is an over-simplification here, which we will repair later.*



Definition-Clear or Killing

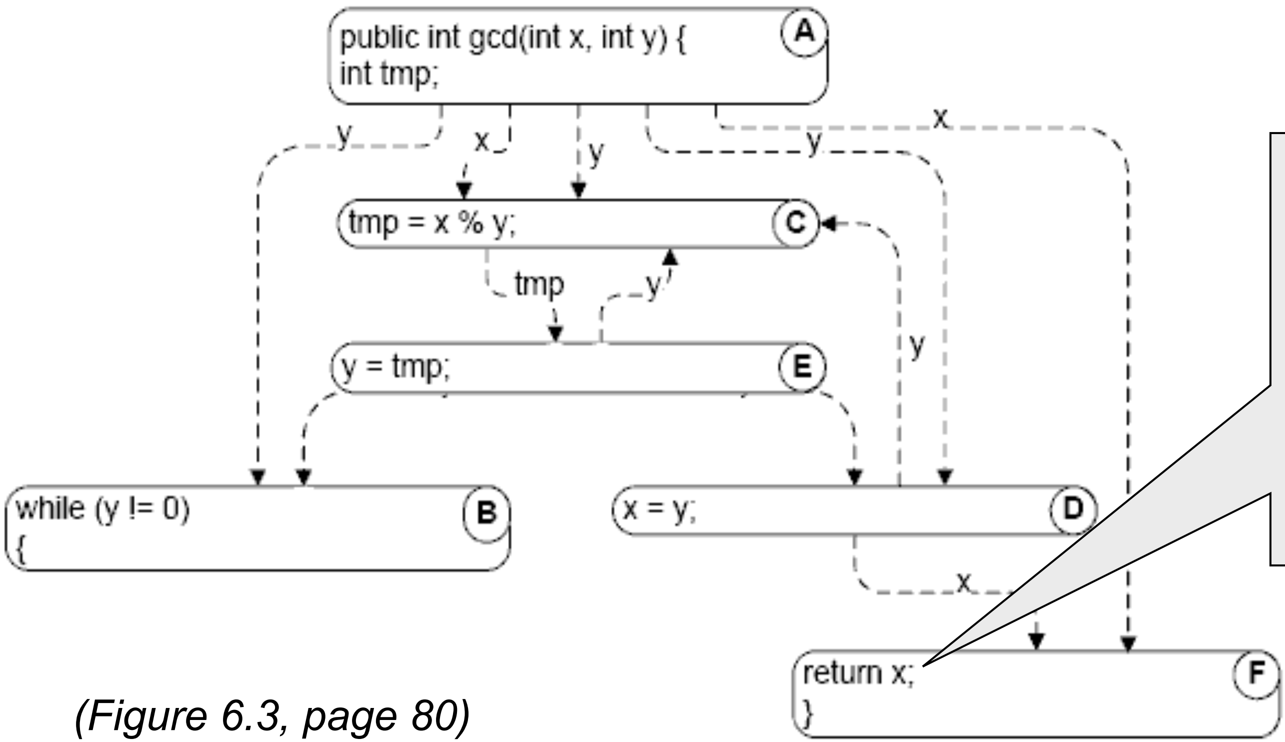
```
x = ...      // A: def x
q = ...
x = y;      //  B: kill x, def x
z = ...
y = f(x);  // C: use x
```





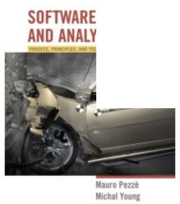
(Direct) Data Dependence Graph

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



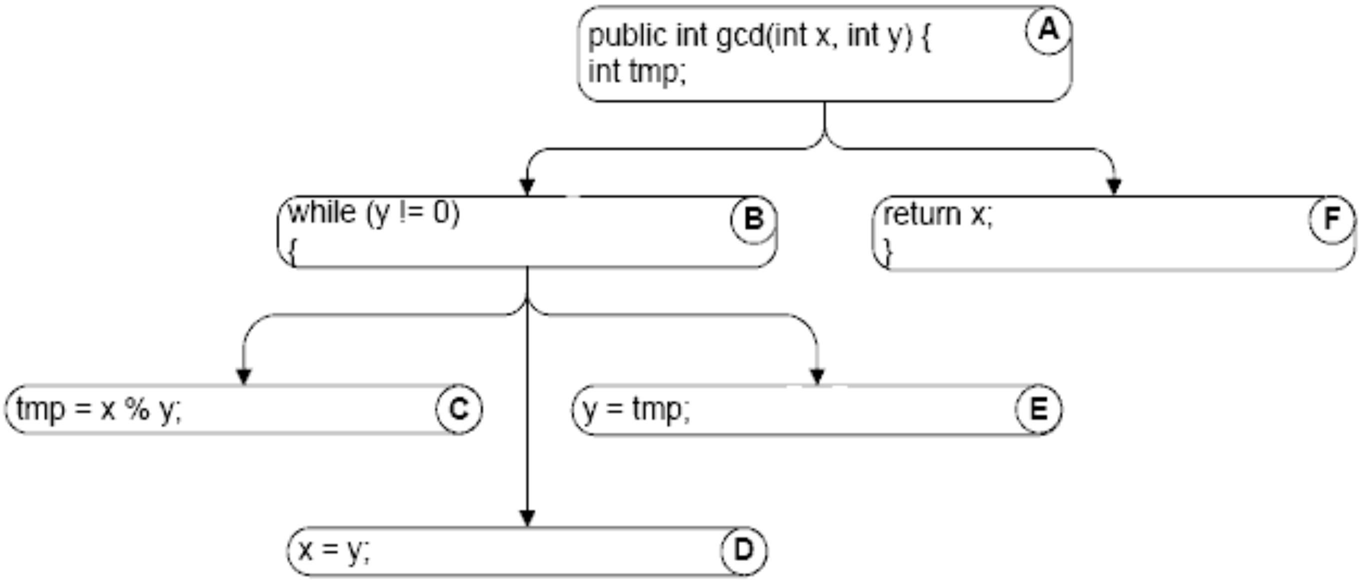
Dependence edges show this x value could be the unchanged parameter or could be set at line D

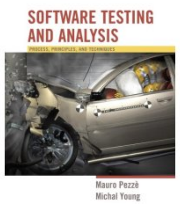
(Figure 6.3, page 80)



Control dependence (1)

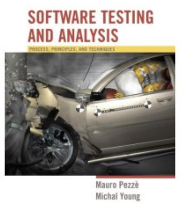
- Data dependence: Where did these values come from?
- Control dependence: Which statement controls whether this statement executes?
  - Nodes: as in the CFG
  - Edges: unlabelled, from entry/branching points to controlled blocks



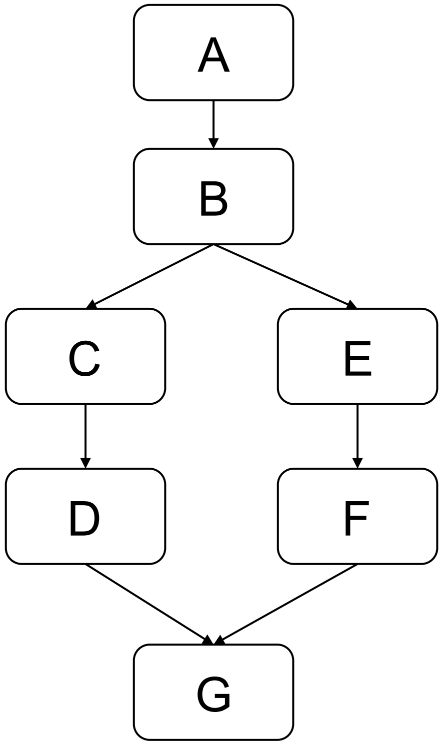


Dominators

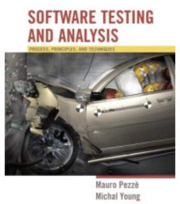
- **Pre-dominators** in a rooted, directed graph can be used to make this intuitive notion of "controlling decision" precise.
- Node M dominates node N if every path from the root to N passes through M.
  - A node will typically have many dominators, but except for the root, there is a unique **immediate 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)

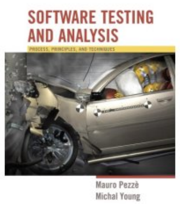


- A pre-dominates all nodes; G post-dominates all nodes
- F and G post-dominate E
- G is the immediate post-dominator of B
  - C does *not* post-dominate B
- B is the immediate pre-dominator of G
  - F does *not* pre-dominate G

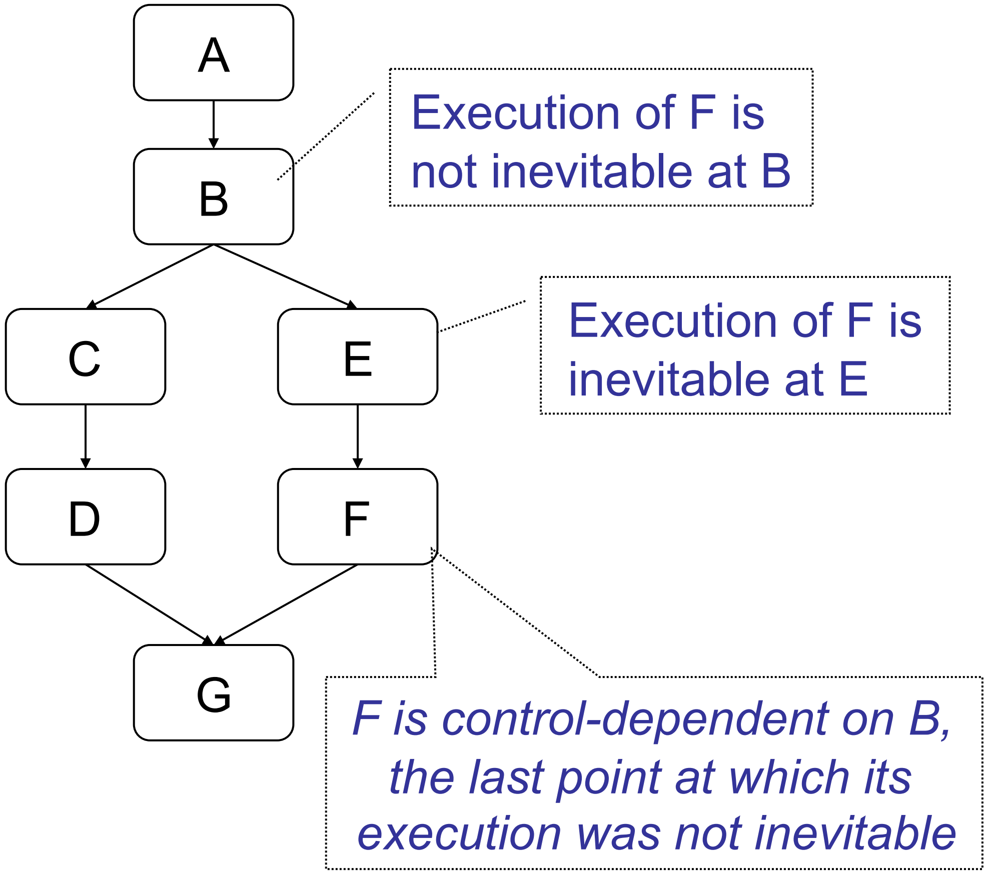


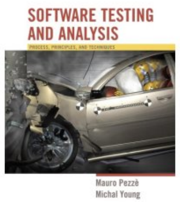
Control dependence (2)

- 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



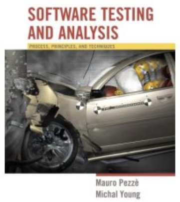
Control Dependence





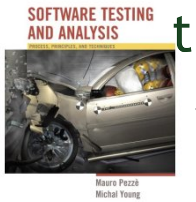
Data Flow Analysis

Computing data flow information

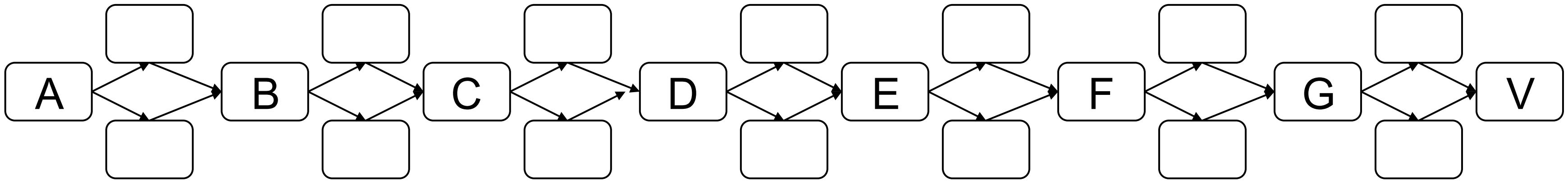


Calculating def-use pairs

- Definition-use pairs can be defined in terms of paths in the program control flow graph:
  - There is an association (d,u) between a definition of variable v at d and a use of variable v at u iff
    - there is at least one control flow path from d to u
    - with no intervening definition of v.
  - v_d **reaches** u (v_d is a **reaching definition** at u).
  - If a control flow path passes through another definition e of the same variable v, v_e **kills** v_d at that point.
- Even if we consider only loop-free paths, the number of paths in a graph can be exponentially larger than the number of nodes and edges.
- Practical algorithms therefore do not search every individual path. Instead, they summarize the reaching definitions at a node over all the paths reaching that node.



Exponential paths
(even without loops)



2 paths from A to B

4 from A to C

8 from A to D

16 from A to E

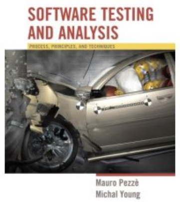
...

128 paths from A to V

Tracing each path is not efficient, and we can do much better.

DF Algorithm

- An efficient algorithm for computing reaching definitions (and several other properties) is based on the way reaching definitions at one node are related to the reaching definitions at an adjacent node.
- Suppose we are calculating the reaching definitions of node n, and there is an edge (p,n) from an immediate predecessor node p.
  - If the predecessor node p can assign a value to variable v, then the definition v_p reaches n. We say the definition v_p is generated at p.
  - If a definition v_p of variable v reaches a predecessor node p, and if v is not redefined at that node (in which case we say the v_p is killed at that point), then the definition is propagated on from p to n.



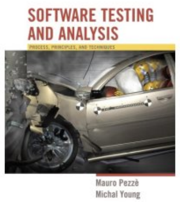
Equations of node E (y = tmp)

Calculate reaching definitions at E in terms of its immediate predecessor D

```
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
  }
```

$$Reach(E) = ReachOut(D)$$

$$ReachOut(E) = (Reach(E) \setminus \{y_A\}) \cup \{y_E\}$$

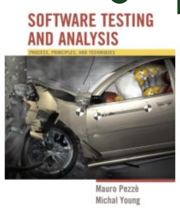


Equations of node B (while (y != 0))

This line has two predecessors: Before the loop, end of the loop

```
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
   }
```

- $\text{Reach}(B) = \text{ReachOut}(A) \cup \text{ReachOut}(E)$
- $\text{ReachOut}(A) = \text{gen}(A) = \{x_A, y_A, tmp_A\}$
- $\text{ReachOut}(E) = (\text{Reach}(E) \setminus \{y_A\}) \cup \{y_E\}$



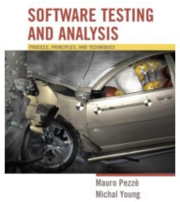
General equations for Reach analysis

$$\text{Reach}(n) = \bigcup_{m \in \text{pred}(n)} \text{ReachOut}(m)$$

$$\text{ReachOut}(n) = (\text{Reach}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$$\text{gen}(n) = \{ v_n \mid v \text{ is defined or modified at } n \}$$

$$\text{kill}(n) = \{ v_x \mid v \text{ is defined or modified at } x, x \neq n \text{ AND } v \text{ is defined or modified at } n \}$$



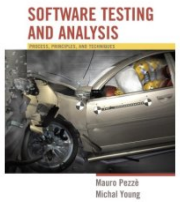
Avail equations

$$\text{Avail}(n) = \bigcap_{m \in \text{pred}(n)} \text{AvailOut}(m)$$

$$\text{AvailOut}(n) = (\text{Avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

gen(n) = { exp | exp is computed at n }

kill(n) = { exp | exp has variables assigned at n }



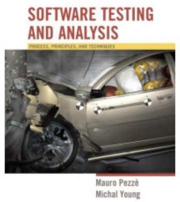
Live variable equations

$$\text{Live}(n) = \bigcup_{m \in \text{succ}(n)} \text{LiveOut}(m)$$

$$\text{LiveOut}(n) = (\text{Live}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$$\text{gen}(n) = \{ v \mid v \text{ is used at } n \}$$

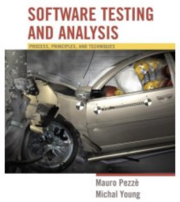
$$\text{kill}(n) = \{ v \mid v \text{ is modified at } n \}$$



Classification of analyses

- Forward/backward: a node's set depends on that of its predecessors/successors
- Any-path/all-path: a node's set contains a value iff it is coming from any/all of its inputs

	Any-path (∪)	All-paths (∩)
Forward (pred)	Reach	Avail
Backward (succ)	Live	"inevitable"

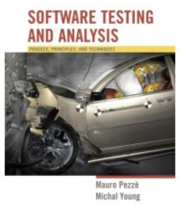


Iterative Solution of Dataflow Equations

[G. KilDall, POPL'73]

- Initialize values (first estimate of answer)
  - For "any path" problems, first guess is "nothing" (empty set) at each node
  - For "all paths" problems, first guess is "everything" (set of all possible values = union of all "gen" sets)
- Repeat until nothing changes
  - Pick some node and recalculate (new estimate)

This will converge on a "fixed point" solution where every new calculation produces the same value as the previous guess.



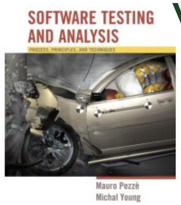
Worklist Algorithm for Data Flow

See figures 6.6, 6.7 on pages 84, 86 of Pezzè & Young

One way to iterate to a fixed point solution.

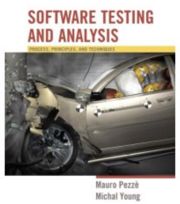
General idea:

- Initially all nodes are on the work list, and have default values
  - Default for "any-path" problem is the empty set, default for "all-path" problem is the set of all possibilities (union of all gen sets)
- While the work list is not empty
  - Pick any node n on work list; remove it from the list
  - Apply the data flow equations for that node to get new values
  - If the new value is changed (from the old value at that node), then
    - Add successors (for forward analysis) or predecessors (for backward analysis) on the work list
- Eventually the work list will be empty (because new computed values = old values for each node) and the algorithm stops.



Cooking your own: From Execution to Conservative Flow Analysis

- We can use the same data flow algorithms to approximate other dynamic properties
  - Gen set will be "facts that become true here"
  - Kill set will be "facts that are no longer true here"
  - Flow equations will describe propagation
- Example: Taintedness (in web form processing)
  - "Taint": a user-supplied value (e.g., from web form) that has not been validated
  - Gen: we get this value from an untrusted source here
  - Kill: we validated to make sure the value is proper

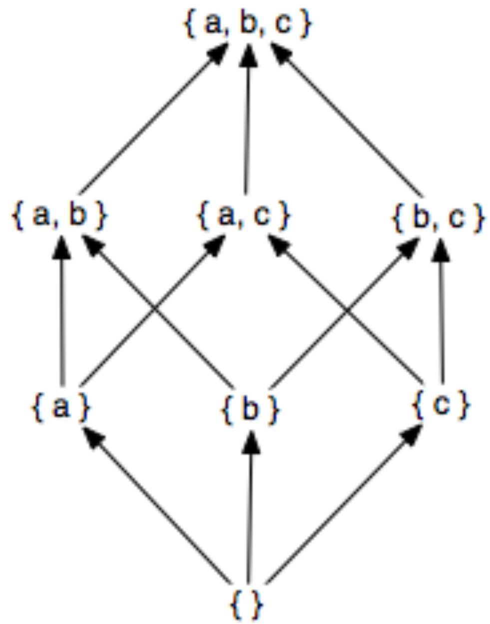


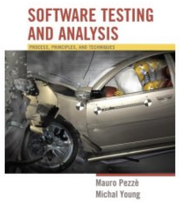
Cooking your own analysis (2)

- Flow equations must be monotonic
  - Initialize to the bottom element of a lattice of approximations
  - Each new value that changes must move up the lattice
- Typically: Powerset lattice
  - Bottom is empty set, top is universe
  - Or empty at top for all-paths analysis

Monotonic: $y > x$ implies $f(y) \geq f(x)$

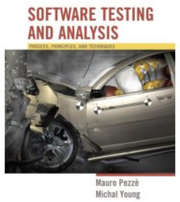
(where f is application of the flow equations on values from successor or predecessor nodes, and ">" is movement up the lattice)





Data flow analysis with arrays and pointers

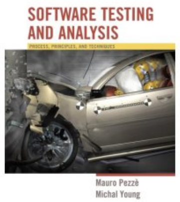
- Arrays and pointers introduce uncertainty: Do different expressions access the same storage?
  - a[i] same as a[k] when i = k
  - a[i] same as b[i] when a = b (**aliasing**)
- The uncertainty is accomodated depending to the kind of analysis
  - Any-path: gen sets should include all potential aliases and kill set should include only what is definitely modified
  - All-path: vice versa



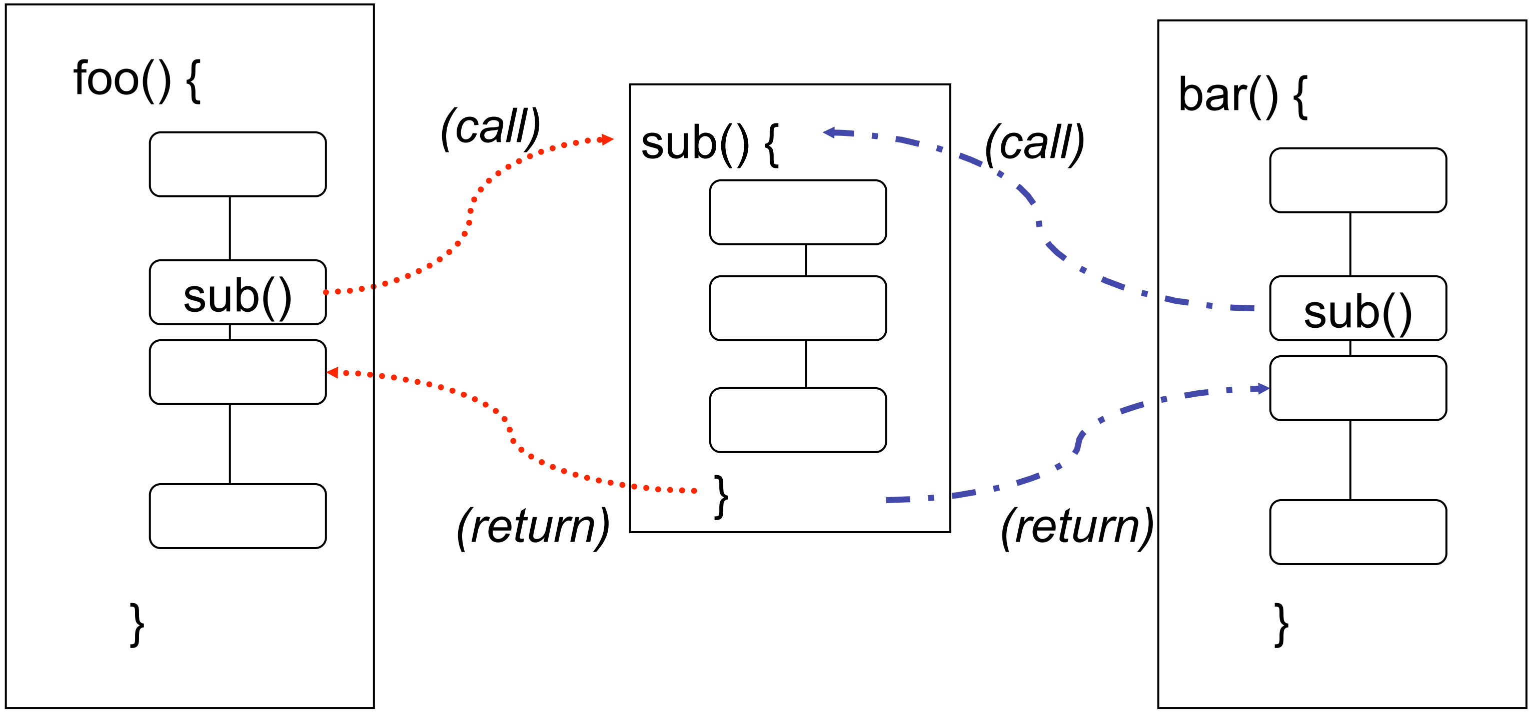
Scope of Data Flow Analysis

- Intraprocedural
  - Within a single method or procedure
    - as described so far
- Interprocedural
  - Across several methods (and classes) or procedures

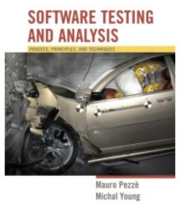
- Cost/Precision trade-offs for interprocedural analysis are critical, and difficult
  - context sensitivity
  - flow-sensitivity



Context Sensitivity

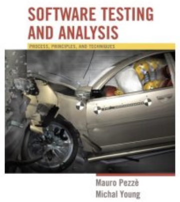


A **context-sensitive** (interprocedural) analysis distinguishes sub() called from foo() from sub() called from bar();
A **context-insensitive** (interprocedural) analysis does not separate them, as if foo() could call sub() and sub() could then return to bar()



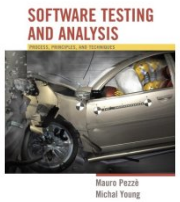
Flow Sensitivity

- Reach, Avail, etc. were flow-sensitive, intraprocedural analyses
  - They considered ordering and control flow decisions
  - Within a single procedure or method, this is (fairly) cheap — $O(n^3)$ for n CFG nodes
- Many interprocedural flow analyses are flow-insensitive
  - $O(n^3)$ would not be acceptable for all the statements in a program!
    - Though $O(n^3)$ on each individual procedure might be ok
  - Often flow-insensitive analysis is good enough ... consider type checking as an example

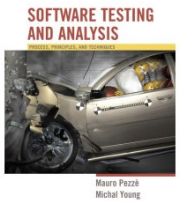


Summary

- Data flow models detect patterns on CFGs:
  - Nodes initiating the pattern
  - Nodes terminating it
  - Nodes that may interrupt it
- Often, but not always, about flow of information (dependence)
- Pros:
  - Can be implemented by efficient iterative algorithms
  - Widely applicable (not just for classic “data flow” properties)
- Limitations:
  - Unable to distinguish feasible from infeasible paths
  - Analyses spanning whole programs (e.g., alias analysis) must trade off precision against computational cost

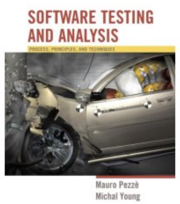


Data flow testing

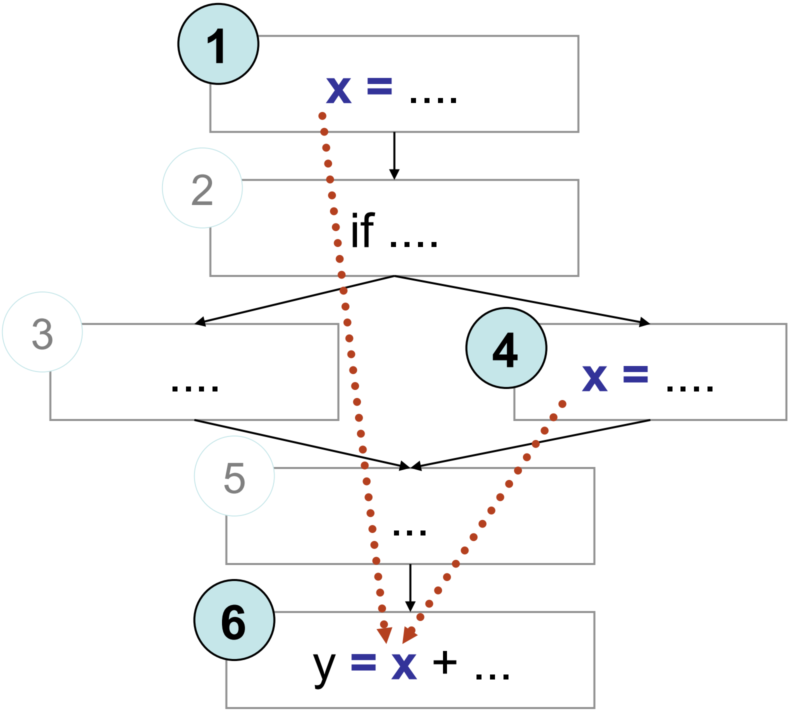


Motivation

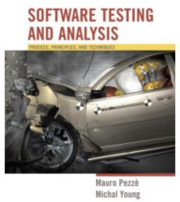
- Middle ground in structural testing
  - Node and edge coverage don't test interactions
  - Path-based criteria require impractical number of test cases
    - And only a few paths uncover additional faults, anyway
  - Need to distinguish "important" paths
- Intuition: Statements interact through *data flow*
  - Value computed in one statement, used in another
  - Bad value computation revealed only when it is used



Data flow concept



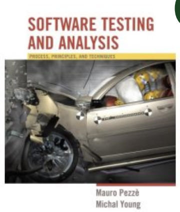
- Value of x at 6 could be computed at 1 or at 4
- Bad computation at 1 or 4 could be revealed only if they are used at 6
- (1,6) and (4,6) are *def-use (DU) pairs*
  - defs at 1,4
  - use at 6



Adequacy criteria

- All DU pairs: Each DU pair is exercised by at least one test case
- All DU paths: Each *simple* (non looping) DU path is exercised by at least one test case
- All definitions: For each definition, there is at least one test case which exercises a DU pair containing it
  - (Every computed value is used somewhere)

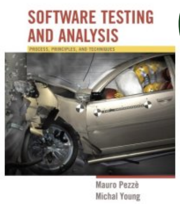
Corresponding coverage fractions can also be defined



Difficult cases

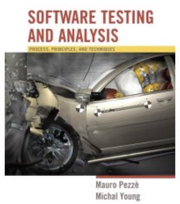
- x[i] = ... ; ... ; y = x[j]
  - DU pair (only) if i==j
- p = &x ; ... ; *p = 99 ; ... ; q = x
  - *p is an alias of x
- m.putFoo(...); ... ; y=n.getFoo(...);
  - Are m and n the same object?
  - Do m and n share a “foo” field?

- Problem of *aliases*: Which references are (always or sometimes) the same?

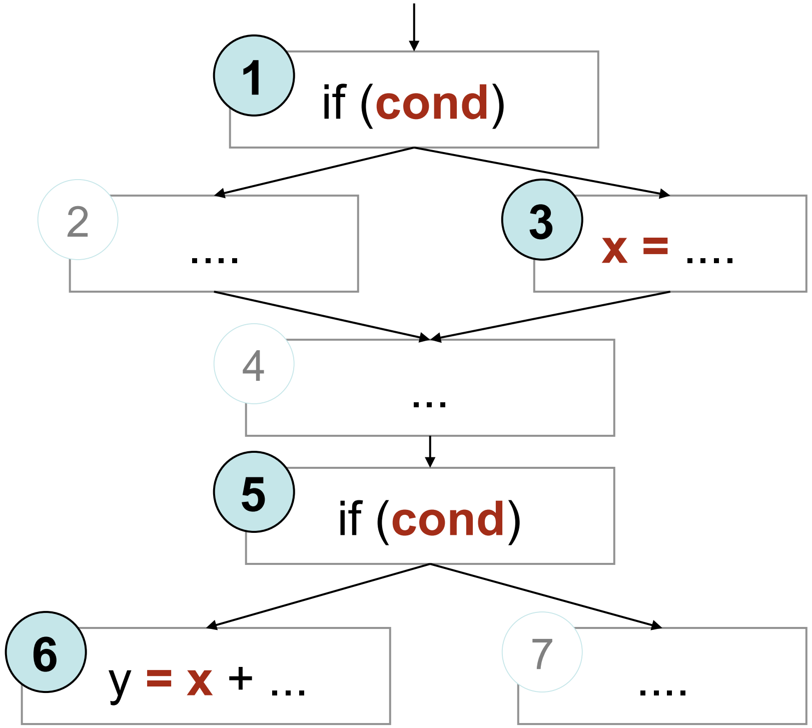


Data flow coverage with complex structures

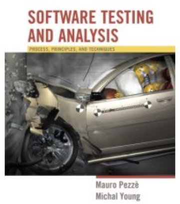
- Arrays and pointers are critical for data flow analysis
  - Under-estimation of aliases may fail to include some DU pairs
  - Over-estimation, on the other hand, may introduce unfeasible test obligations
- For testing, it may be preferrable to accept under-estimation of alias set rather than over-estimation or expensive analysis
  - Controversial: In other applications (e.g., compilers), a *conservative* over-estimation of aliases is usually required
  - Alias analysis may rely on external guidance or other global analysis to calculate good estimates
  - Undisciplined use of dynamic storage, pointer arithmetic, etc. may make the whole analysis infeasible



Infeasibility



- Suppose *cond* has not changed between 1 and 5
  - Or the conditions could be different, but the first implies the second
- Then (3,5) is not a (feasible) DU pair
  - But it is difficult or impossible to determine which pairs are infeasible
- Infeasible test obligations are a problem
  - No test case can cover them



Infeasibility

- The path-oriented nature of data flow analysis makes the infeasibility problem especially relevant
  - Combinations of elements matter!
  - Impossible to (infallibly) distinguish feasible from infeasible paths. More paths = more work to check manually.
- In practice, reasonable coverage is (often, not always) achievable
  - Number of paths is exponential in worst case, but often linear
  - All DU *paths* is more often impractical