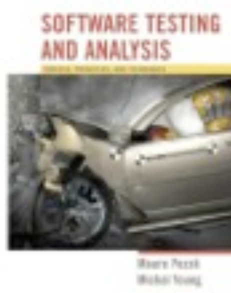
Structural and dataflow testing (cont.)

Automated testing and verification

J.P. Galeotti - Alessandra Gorla

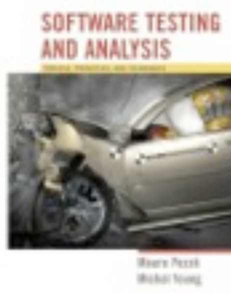
Structural testing

- Judging test suite thoroughness based on the *structure* of the program itself
- Also known as “white-box”, “glass-box”, or “code-based” testing
- To distinguish from functional (requirements-based, “black-box” testing)
- “Structural” testing is still testing product functionality against its specification. Only the measure of thoroughness has changed.



Why structural (code-based) testing?

- One way of answering the question “What is *missing* in our test suite?”
  - If part of a program is not executed by any test case in the suite, faults in that part cannot be exposed
  - But what’s a “part”?
    - Typically, a control flow element or combination:
    - Statements (or CFG nodes), Branches (or CFG edges)
    - Fragments and combinations: Conditions, paths
- Complements functional testing: Another way to recognize cases that are treated differently
  - Recall fundamental rationale: Prefer test cases that are treated *differently* over cases treated the same



No guarantees

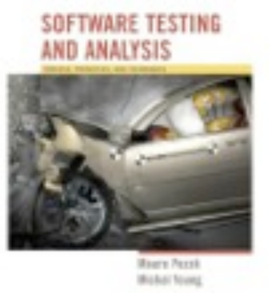
- Executing all control flow elements does not guarantee finding all faults
  - Execution of a faulty statement may not always result in a failure
    - The state may not be corrupted when the statement is executed with some data values
    - Corrupt state may not propagate through execution to eventually lead to failure
- What is the value of structural coverage?
  - Increases confidence in thoroughness of testing
    - Removes some obvious *inadequacies*

Structural testing **complements** functional testing

- Control flow testing includes cases that may not be identified from specifications alone
  - Typical case: implementation of a single item of the specification by multiple parts of the program
  - Example: hash table collision (invisible in interface spec)
- Test suites that satisfy control flow adequacy criteria could fail in revealing faults that can be caught with functional criteria
  - Typical case: missing path faults

Structural testing in practice

- Create functional test suite first, then measure structural coverage to see what is missing
- Interpret unexecuted elements
  - may be due to natural differences between specification and implementation
  - or may reveal flaws of the software or its development process
    - inadequacy of specifications that do not include cases present in the implementation
    - coding practice that radically diverges from the specification
    - inadequate functional test suites
- Attractive because automated
  - coverage measurements are convenient progress indicators
  - sometimes used as a criterion of completion
    - use with caution: does not ensure *effective* test suites

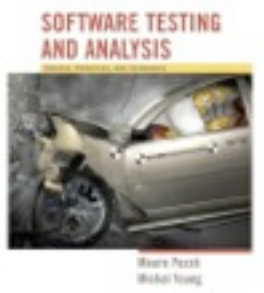


Statement testing

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once
- Coverage:

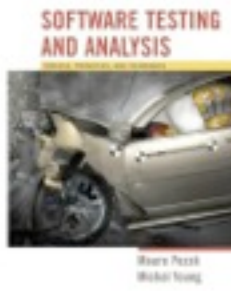
$$\frac{\text{\# executed statements}}{\text{\# statements}}$$

- **Rationale**: a fault in a statement can only be revealed by executing the faulty statement



Statements or blocks?

- Nodes in a control flow graph often represent basic blocks of multiple statements
  - Some standards refer to *basic block* coverage or *node coverage*
  - Difference in granularity, not in concept
- No essential difference
  - 100% node coverage <-> 100% statement coverage
    - but levels will differ below 100%
  - A test case that improves one will improve the other
    - though not by the same amount, in general

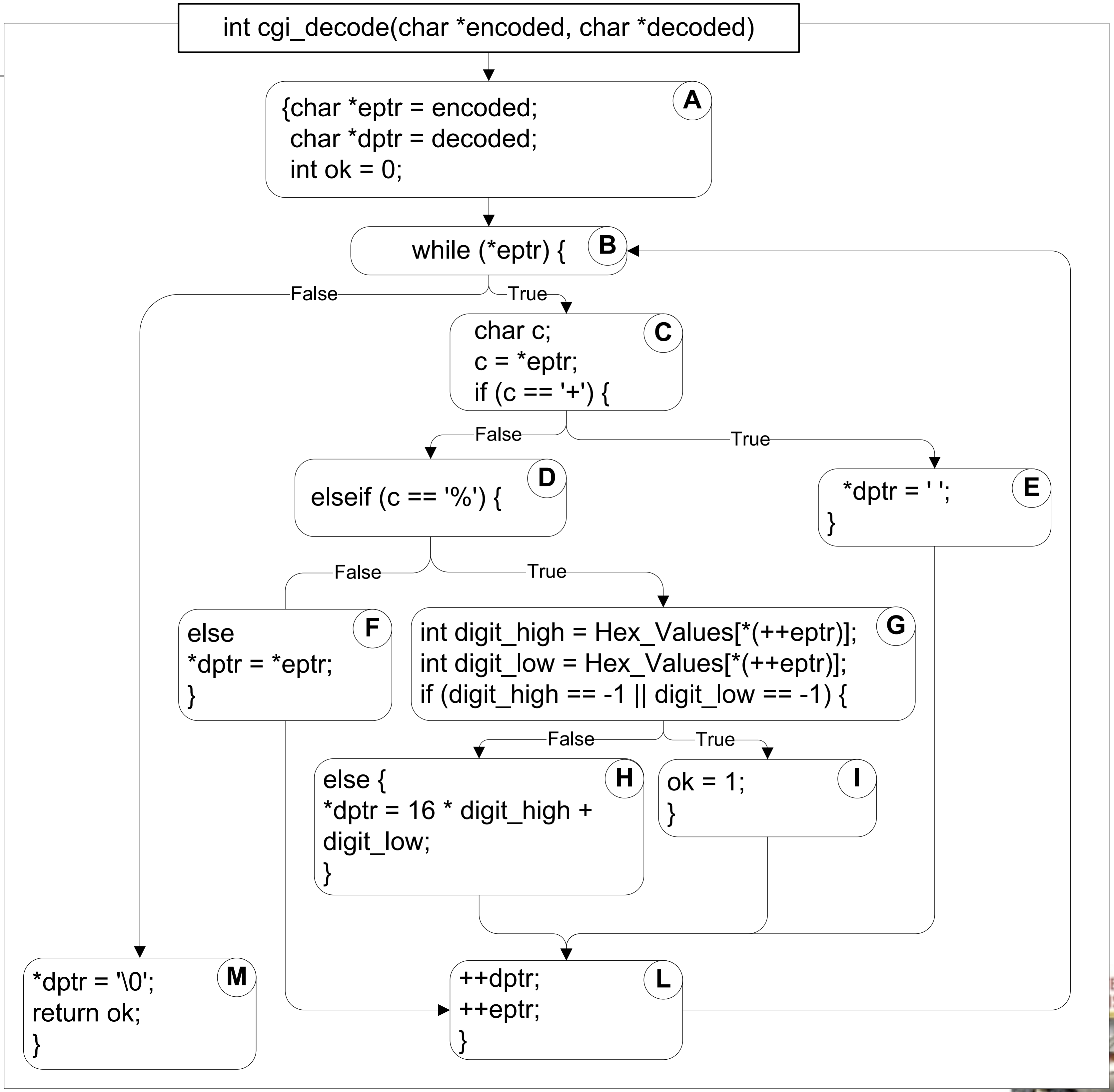


Example

T_0 =
{“”, “test”,
“test+case%1Dadequacy”}
17/18 = 94% Stmt Cov.

T_1 =
{“adequate+test
%0Dexecution%7U”}
18/18 = 100% Stmt Cov.

T_2 =
{“%3D”, “%A”, “a+b”,
“test”}
18/18 = 100% Stmt Cov.



Thursday, January 17, 13

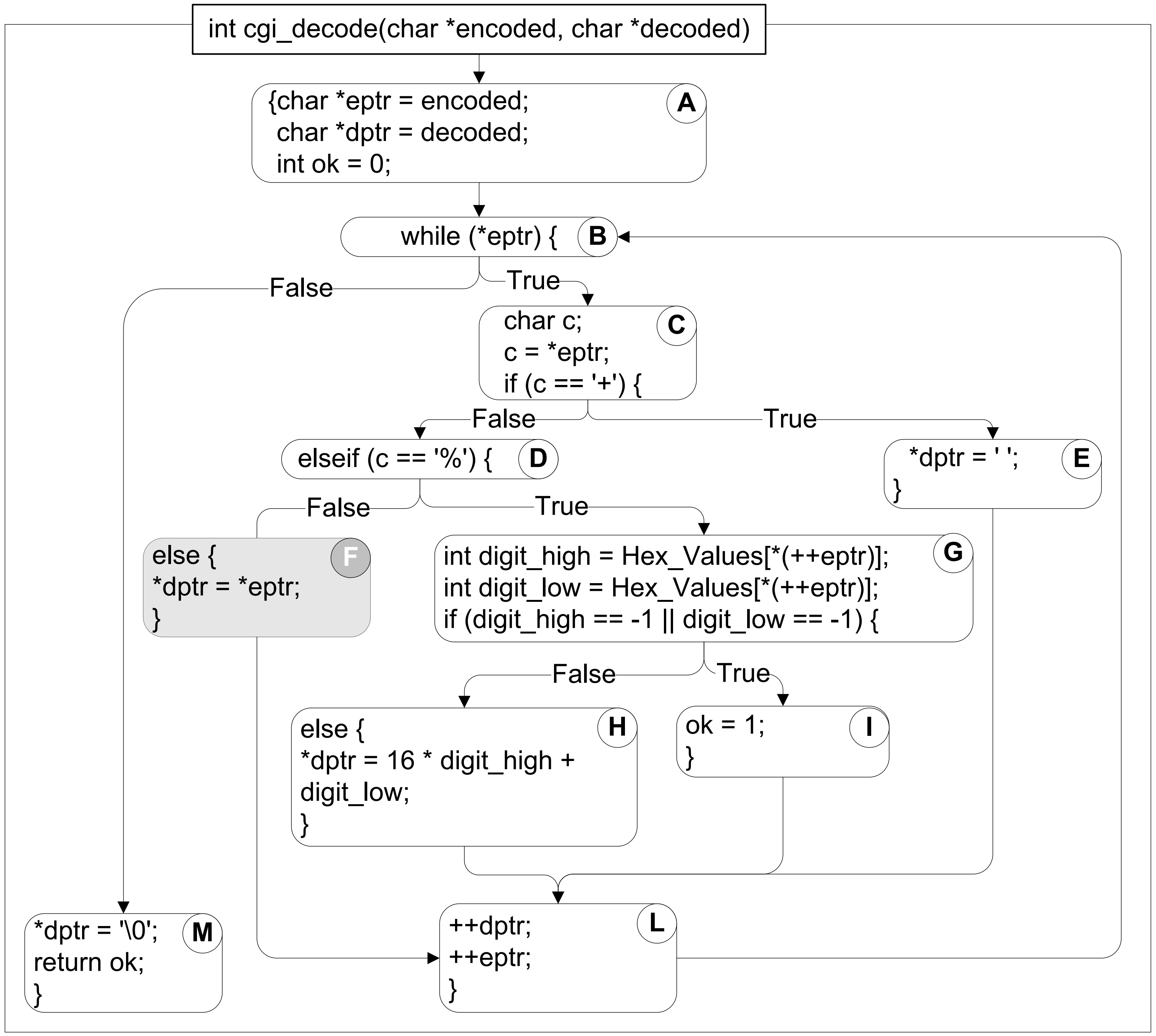
Coverage is not size

- Coverage does not depend on the number of test cases
  - $T_0, T_1 : T_1 >_{coverage} T_0 \qquad T_1 <_{cardinality} T_0$
  - $T_1, T_2 : T_2 =_{coverage} T_1 \qquad T_2 >_{cardinality} T_1$
- Minimizing test suite size is seldom the goal
  - small test cases make failure diagnosis easier
  - a failing test case in T_2 gives more information for fault localization than a failing test case in T_1

“All statements” can miss some cases

- Complete statement coverage may not imply executing all branches in a program
- Example:
  - Suppose block F were missing
  - Statement adequacy would not require *false* branch from D to L

T_3 =
{“”, “+%0D+%4J”}
100% Stmt Cov.
No *false* branch from D



int cgi_decode(char *encoded, char *decoded)
{char *eptr = encoded; char *dptr = decoded; int ok = 0;
A
while (*eptr) {
B
False
True
char c; c = *eptr; if (c == '+') {
C
False
True
elseif (c == '%') {
D
*dptr = ' '; }
E
False
True
else { *dptr = *eptr; }
F
int digit_high = Hex_Values[*(++eptr)]; int digit_low = Hex_Values[*(++eptr)]; if (digit_high == -1 || digit_low == -1) {
G
False
True
else { *dptr = 16 * digit_high + digit_low; }
H
ok = 1; }
I
*dptr = '\0'; return ok; }
M
++dptr; ++eptr; }
L

Branch testing

- Adequacy criterion: each branch (edge in the CFG) must be executed at least once
- Coverage:

$$\frac{\#\ \text{executed branches}}{\#\ \text{branches}}$$

T_3 = {“”, “+%0D+%4J”}

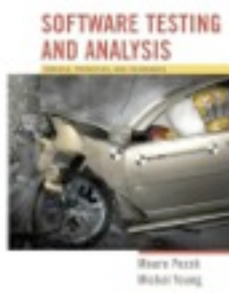
100% Stmt Cov.   88% Branch Cov. (7/8 branches)

T_2 = {“%3D”, “%A”, “a+b”, “test”}

100% Stmt Cov.   100% Branch Cov. (8/8 branches)

Statements vs branches

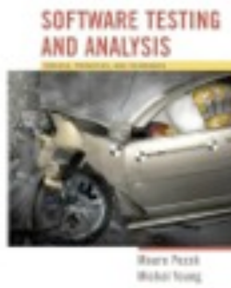
- Traversing all edges of a graph causes all nodes to be visited
  - So test suites that satisfy the branch adequacy criterion for a program P also satisfy the statement adequacy criterion for the same program
- The converse is not true (see T_3)
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)



Subsume relation

- Branch criterion subsumes statement criterion

Does this mean that if it is possible to find a fault with a test suite that satisfies statement criterion then the same fault will be discovered by any other test suite satisfying branch criterion?

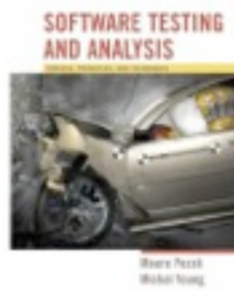


Subsume relation

- Branch criterion subsumes statement criterion

Does this mean that if it is possible to find a fault with a test suite that satisfies statement criterion then the same fault will be discovered by any other test suite satisfying branch criterion?

NO!



“All branches” can still miss conditions

- Sample fault: missing operator (negation)

```
digit_high == -1 || digit_low == 1
```

- Branch adequacy criterion can be satisfied by varying only digit_high
    - The faulty sub-expression might never determine the result
    - We might never really test the faulty condition, even though we tested both outcomes of the branch

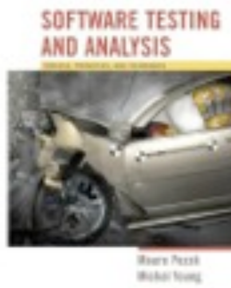
Condition testing

- Branch coverage exposes faults in how a computation has been decomposed into cases
  - intuitively attractive: check the programmer's case analysis
  - but only roughly: groups cases with the same outcome
- Condition coverage considers case analysis in more detail
  - also *individual conditions* in a compound Boolean expression
    - e.g., both parts of `digit_high == -1 || digit_low == -1`

Basic condition testing

- Adequacy criterion: each basic condition must be executed at least once
- Coverage:

$$\frac{\text{\# truth values taken by all basic conditions}}{2 * \text{\# basic conditions}}$$



Basic conditions vs branches

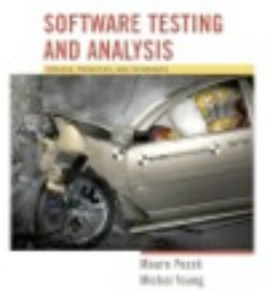
- Basic condition adequacy criterion can be satisfied without satisfying branch coverage

T4 = {“first+test%9Ktest%K9”}

satisfies basic condition adequacy

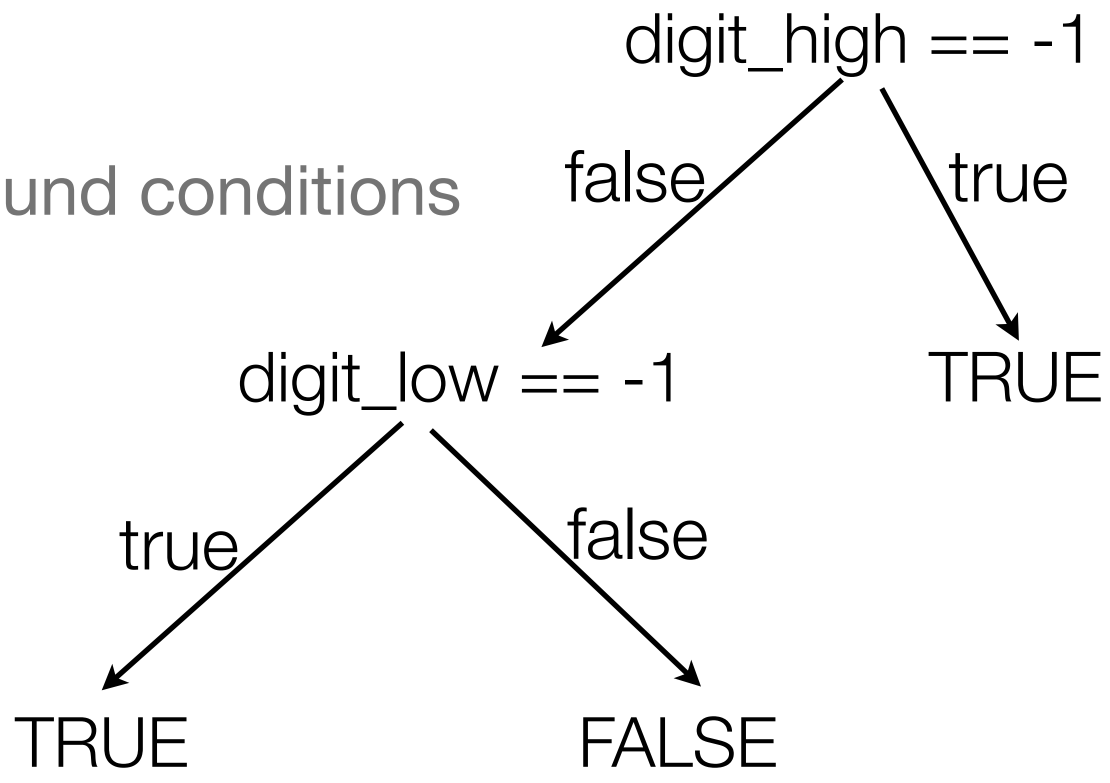
does not satisfy branch condition adequacy

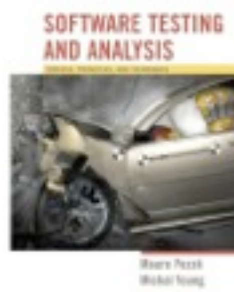
Branch and basic condition are not comparable (neither implies the other)



Covering branches and conditions

- Branch and condition adequacy:
  - cover all conditions and all decisions
- Compound condition adequacy:
  - Cover all possible evaluations of compound conditions
  - Cover all branches of a decision tree



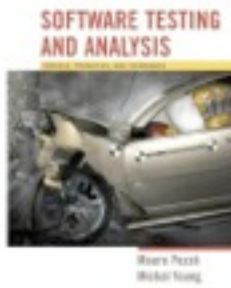


Compound conditions: Exponential complexity

`(((a || b) && c) || d) && e`

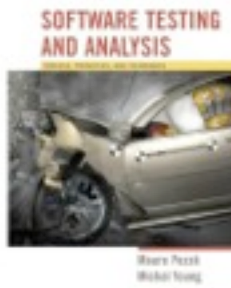
Test Case	a	b	c	d	e
(1)	T	–	T	–	T
(2)	F	T	T	–	T
(3)	T	–	F	T	T
(4)	F	T	F	T	T
(5)	F	F	–	T	T
(6)	T	–	T	–	F
(7)	F	T	T	–	F
(8)	T	–	F	T	F
(9)	F	T	F	T	F
(10)	F	F	–	T	F
(11)	T	–	F	F	–
(12)	F	T	F	F	–
(13)	F	F	–	F	–

- short-circuit evaluation often reduces this to a more manageable number, but not always



Modified condition/decision (MC/DC)

- Motivation: Effectively test important combinations of conditions, without exponential blowup in test suite size
  - "Important" combinations means: Each basic condition shown to independently affect the outcome of each decision
- Requires:
  - For each basic condition C, two test cases,
  - values of all *evaluated* conditions except C are the same
  - compound condition as a whole evaluates to *true* for one and *false* for the other



MC/DC: linear complexity

- N+1 test cases for N basic conditions

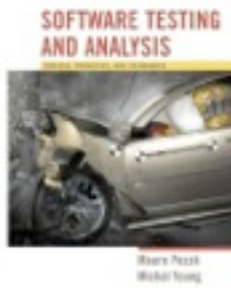
```
(((a || b) && c) || d) && e
```

Test Case	a	b	c	d	e	outcome
(1)	true	--	true	--	true	true
(2)	false	true	true	--	true	true
(3)	true	--	false	true	true	true
(6)	true	--	true	--	false	false
(11)	true	--	false	false	--	false
(13)	false	false	--	false	--	false

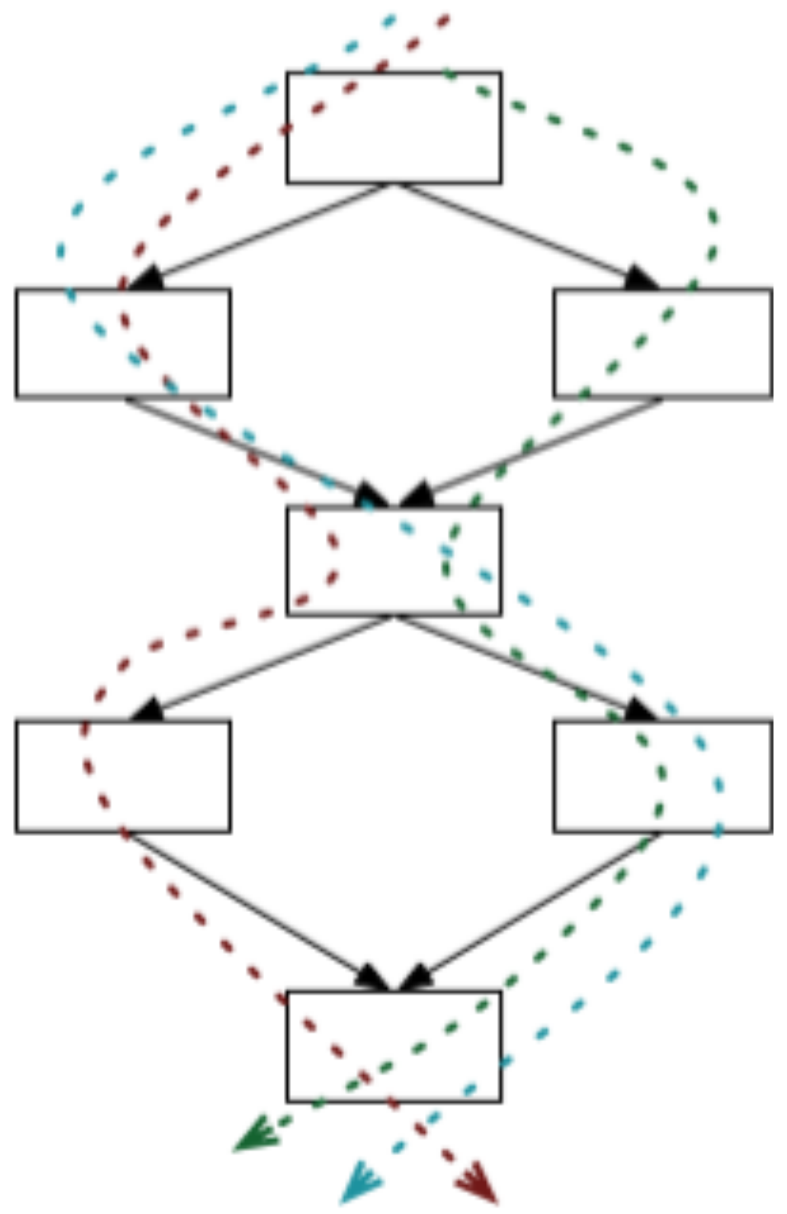
- Underlined values independently affect the output of the decision
- Required by the RTCA/DO-178B standard

Comments on MC/DC

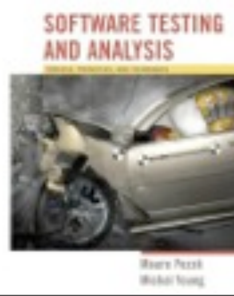
- MC/DC is
  - basic condition coverage (C)
  - branch coverage (DC)
  - plus one additional condition (M):
    every condition must *independently affect* the decision's output
- It is subsumed by compound conditions and subsumes all other criteria discussed so far
  - stronger than statement and branch coverage
- A good balance of thoroughness and test size (and therefore widely used)



Paths? (Beyond individual branches)



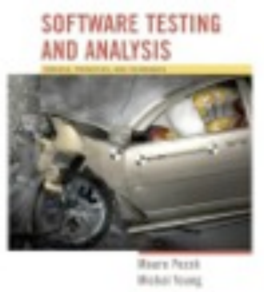
- Should we explore sequences of branches (paths) in the control flow?
- Many more paths than branches
  - A pragmatic compromise will be needed



Path adequacy

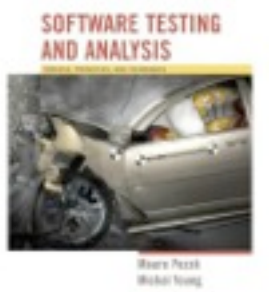
- Decision and condition adequacy criteria consider individual program decisions
- Path testing focuses consider combinations of decisions along paths
- Adequacy criterion: each path must be executed at least once
- Coverage:

$$\frac{\text{\# executed paths}}{\text{\# paths}}$$



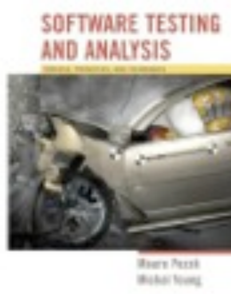
Practical path coverage criteria

- The number of paths in a program with loops is unbounded
    - the simple criterion is usually impossible to satisfy
- For a feasible criterion: Partition infinite set of paths into a finite number of classes
- Useful criteria can be obtained by limiting
    - the number of traversals of loops
    - the length of the paths to be traversed
    - the dependencies among selected paths

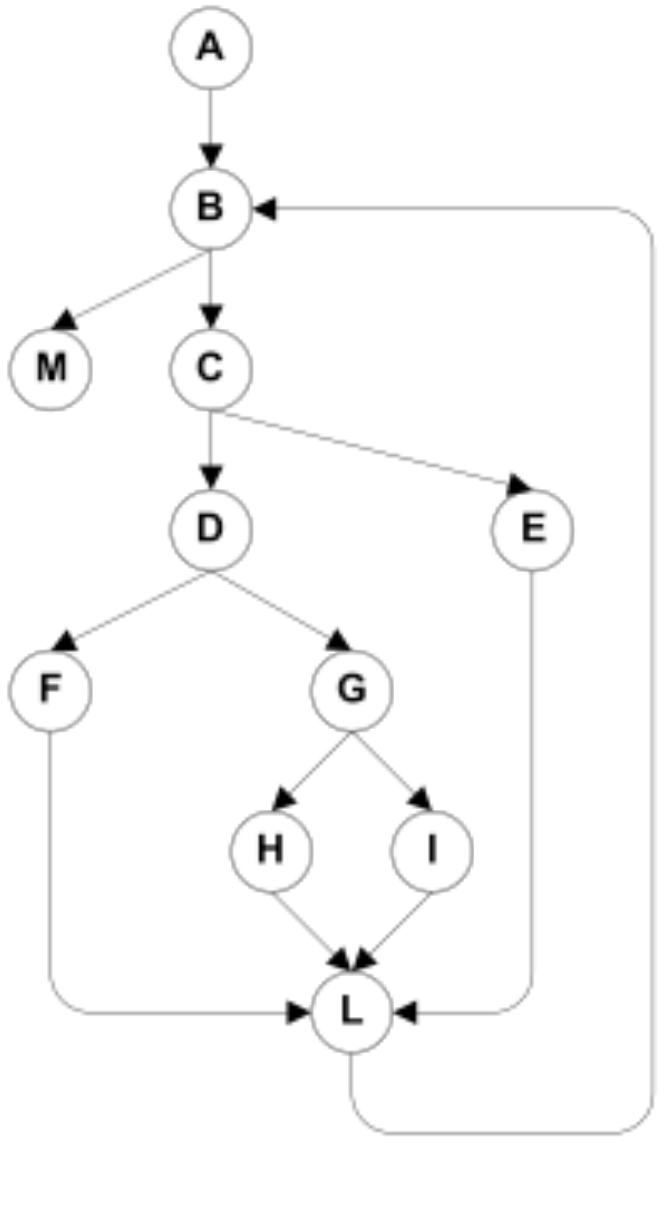


Boundary interior path testing

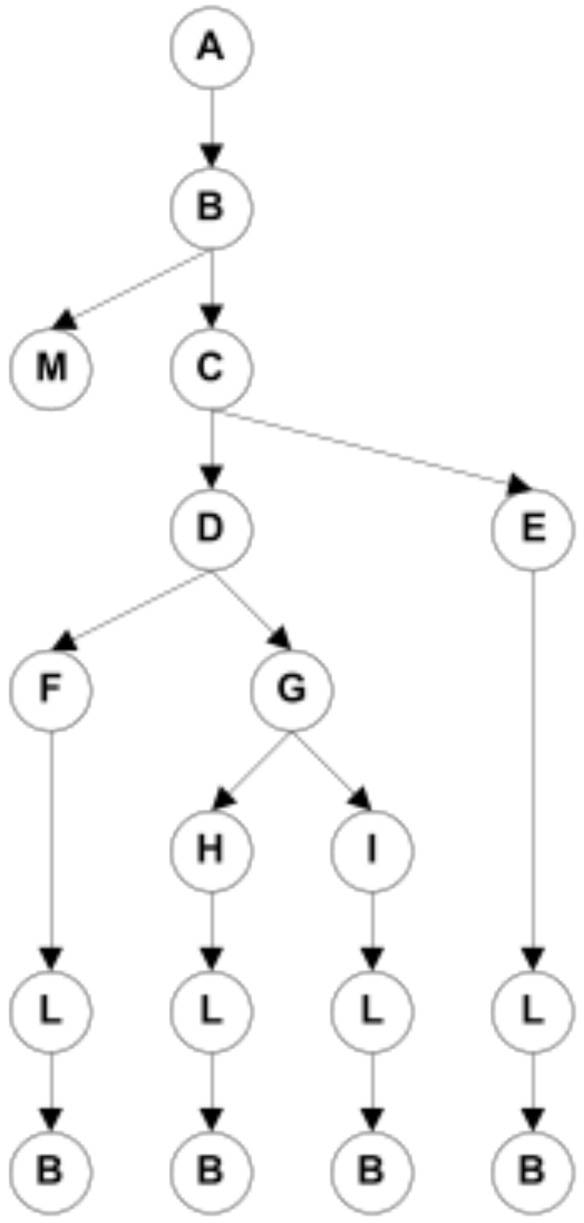
- Group together paths that differ only in the subpath they follow when repeating the body of a loop
  - Follow each path in the control flow graph up to the first repeated node
  - The set of paths from the root of the tree to each leaf is the required set of subpaths for boundary/interior coverage



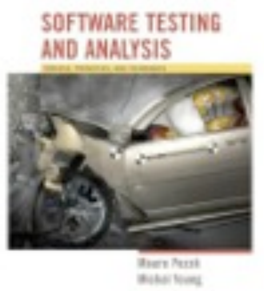
Boundary interior adequacy for cgi-decode



(i)



(ii)

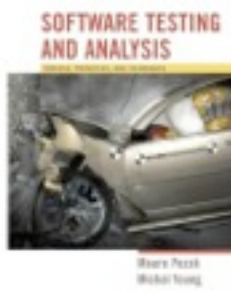


Limitations of boundary interior adequacy

- The number of paths can still grow exponentially

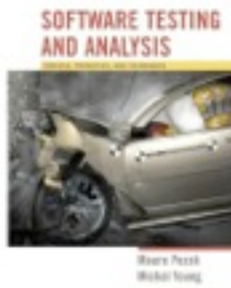
```
if (a) {
     S1;
    }
if (b) {
     S2;
    }
if (c) {
     S3;
    }
   ...
if (x) {
     Sn;
    }
```

- The subpaths through this control flow can include or exclude each of the statements Si, so that in total N branches result in 2^N paths that must be traversed
- Choosing input data to force execution of one particular path may be very difficult, or even impossible if the conditions are not independent



Loop boundary adequacy

- Variant of the boundary/interior criterion that treats loop boundaries similarly but is less stringent with respect to other differences among paths
- Criterion: A test suite satisfies the loop boundary adequacy criterion iff for every loop:
  - In at least one test case, the loop body is iterated zero times
  - In at least one test case, the loop body is iterated once
  - In at least one test case, the loop body is iterated more than once



LCSAJ adequacy

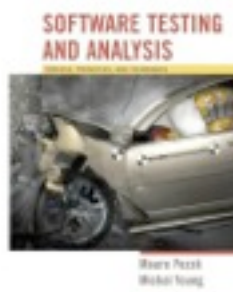
- A Linear Code Sequence and Jump is a program unit comprised of a textual code sequence that terminates in a jump to the beginning of another code sequence and jump.
- An LCSAJ is represented as a triple (X,Y,Z) where X and Y are, respectively:
  - X: location of the first statement
  - Y: location of the last statement
  - Z: location to which the statement Y jumps

LCSAJ

```
1   begin
2     int x, y, p;
3     input (x, y);
4     if(x<0)
5       p=g(y);
6     else
7       p=g(y*y);
8   end
```

LCSAJ	Start Line	End Line	Jump to
1	1	6	exit
2	1	4	7
3	7	8	exit

The last statement in as LCSAJ is a jump and Z may be the program exit. When control arrives at statement X, follows to statement Y, and then jumps to Z, we say that LCSAJ (X,Y,Z) is covered.



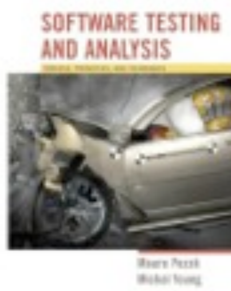
LCSAJ coverage

```
1   begin
2     int x, y, p;
3     input (x, y);
4     if(x<0)
5       p=g(y);
6     else
7       p=g(y*y);
8   end
```

LCSAJ	Start Line	End Line	Jump to
1	1	6	exit
2	1	4	7
3	7	8	exit

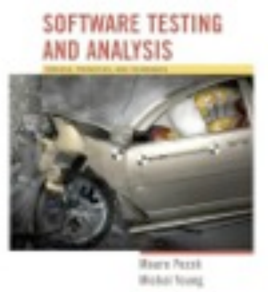
$$T = \left\{ \begin{array}{lll} t_1 : & < \mathrm{x} = -5 & \mathrm{y} = 2 > \\ t_2 : & < \mathrm{x} = 9 & \mathrm{y} = 2 > \end{array} \right\}$$

Test suite T covers all three LCSAJs



Exercise

```
begin
// Compute $x^y$ given non-negative integers x and y.
  int x, y, p;
  input (x, y);
  p=1;
  count=y;
  while(count>0){
    p=p*x;
    count=count-1;
  }
  output(p);
end
```



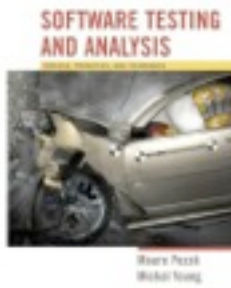
Exercise

```
1    begin
2    // Compute x^y given non-negative
3      int x, y, p;
4      input (x, y);
5      p=1;
6      count=y;
7      while(count>0){
8        p=p*x;
9        count=count-1;
10     }
11     output(p);
12   end
```

LCSAJ	Start Line	End Line	Jump to
1	1	10	7
2	7	10	7
3	7	7	11
4	1	7	11
5	11	12	exit

$$T = \left\{ \begin{array}{lll} t_1 : & < \mathrm{x} = 5 & \mathrm{y} = 0 > \\ t_2 : & < \mathrm{x} = 5 & \mathrm{y} = 2 > \end{array} \right\}$$

T covers all the LCSAJs



Cyclomatic adequacy

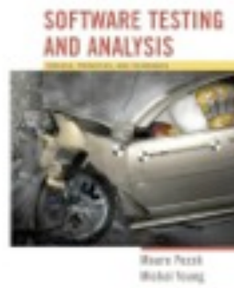
- Cyclomatic number: the minimum number of paths that can generate all paths by addition and subtraction
- Take one path from entry to exit, and count the number of alternative by flipping one condition at a time.

Cyclomatic testing -- example

```
if (a) {
      S1;
     }
if (b) {
      S2;
     }
if (c) {
      S3;
     }
    ...
if (x) {
      Sn;
     }
```

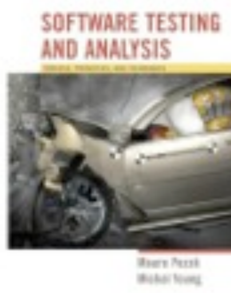
1	FALSE	FALSE	FALSE	FALSE
2	TRUE	FALSE	FALSE	FALSE
3	FALSE	TRUE	FALSE	FALSE
4	FALSE	FALSE	TRUE	TRUE
5	FALSE	FALSE	FALSE	TRUE

<true, false, true, false>
can be obtained by adding 2 and 4 and subtracting 1



Towards procedure call testing

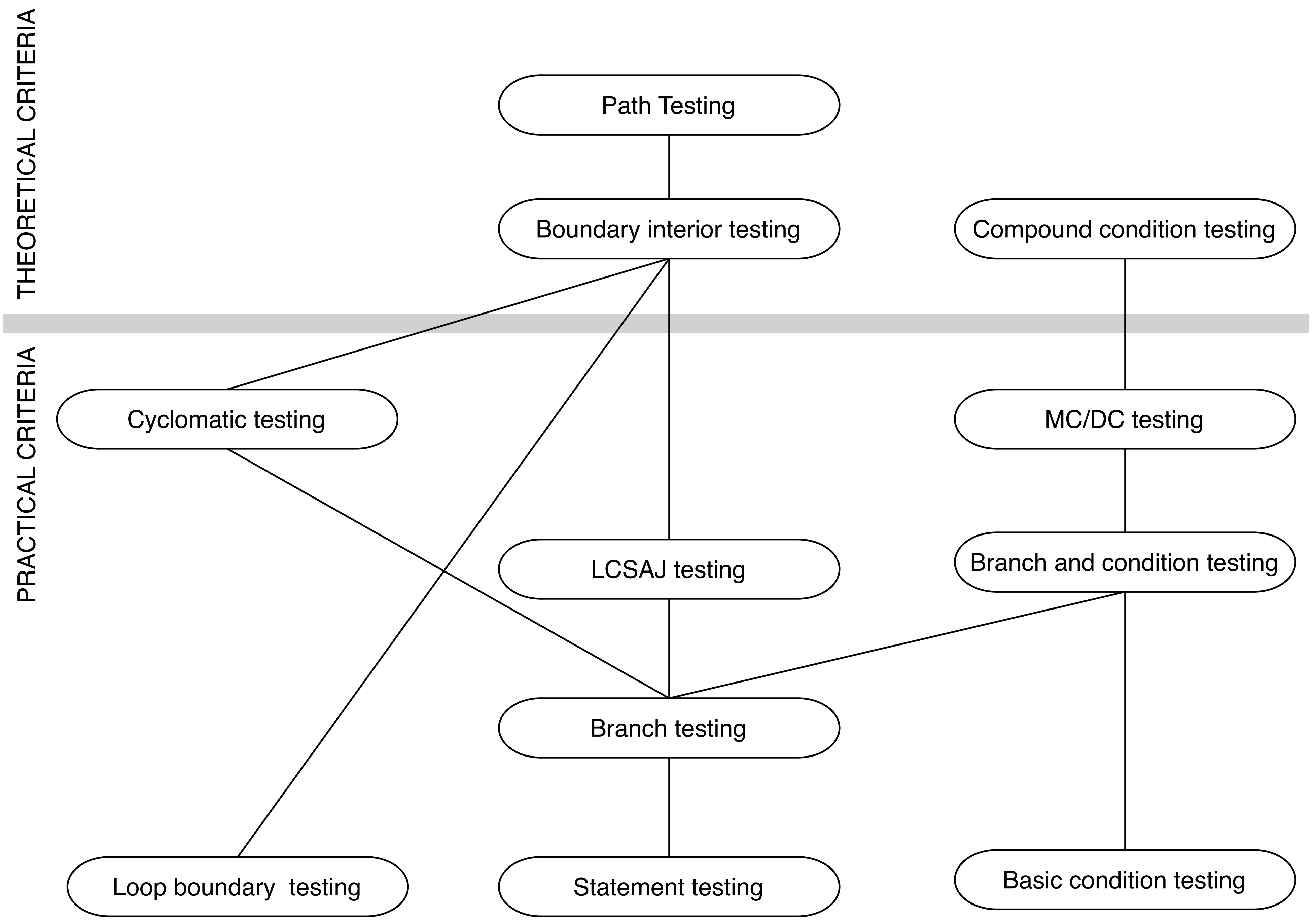
- The criteria considered to this point measure coverage of control flow within individual procedures.
  - not well suited to integration or system testing
- Choose a coverage granularity commensurate with the granularity of testing
  - if unit testing has been effective, then faults that remain to be found in integration testing will be primarily interface faults, and testing effort should focus on interfaces between units rather than their internal details

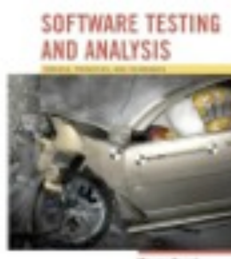


Procedure call testing

- Procedure entry and exit testing
  - procedure may have multiple entry points (e.g., Fortran) and multiple exit points
- Call coverage
  - The same entry point may be called from many points

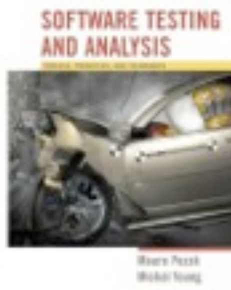
Subsumption relation





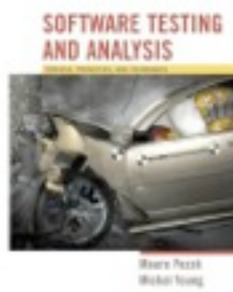
Satisfying structural criteria

- Sometimes criteria may not be satisfiable
  - The criterion requires execution of
    - statements that cannot be executed as a result of
      - defensive programming
      - code reuse (reusing code that is more general than strictly required for the application)
    - conditions that cannot be satisfied as a result of
      - interdependent conditions
    - paths that cannot be executed as a result of
      - interdependent decisions



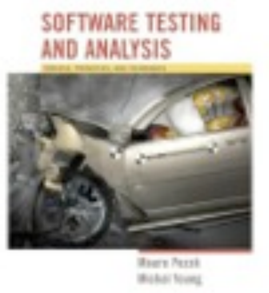
Satisfying structural criteria

- Large amounts of *fossil* code may indicate serious maintainability problems
  - But some unreachable code is common even in well-designed, well-maintained systems
- Solutions:
  - make allowances by setting a coverage goal less than 100%
  - require justification of elements left uncovered



Summary

- We defined a number of adequacy criteria
  - NOT test design techniques!
- Different criteria address different classes of errors
- Full coverage is usually unattainable
  - Remember that attainability is an undecidable problem!
- …and when attainable, “inversion” is usually hard
  - How do I find program inputs allowing to cover something buried deeply in the CFG?
  - Automated support (e.g., symbolic execution) may be necessary
- Therefore, rather than requiring full adequacy, the “degree of adequacy” of a test suite is estimated by coverage measures
  - May drive test improvement



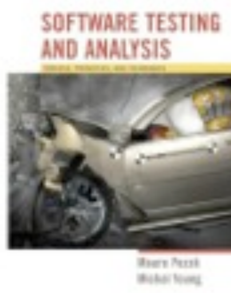
Dataflow testing

Automated testing and verification

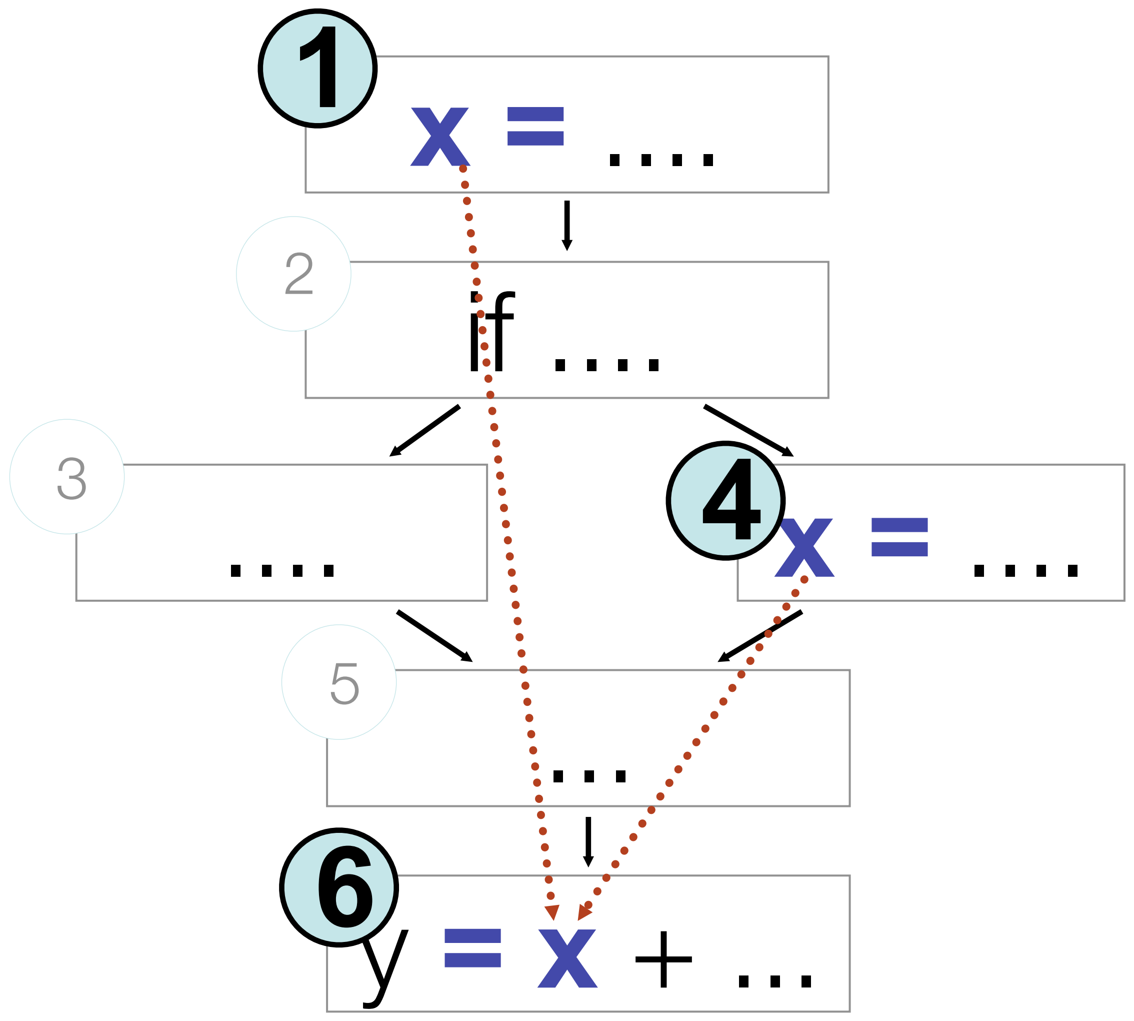
J.P. Galeotti - Alessandra Gorla

Motivation

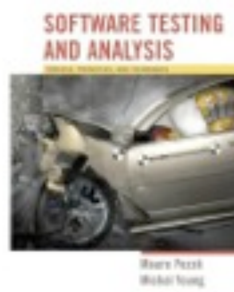
- Middle ground in structural testing
    - Node and edge coverage don't test interactions
    - Path-based criteria require impractical number of test cases
        - And only a few paths uncover additional faults, anyway
    - Need to distinguish "important" paths
- Intuition: Statements interact through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



Dataflow 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



Terms

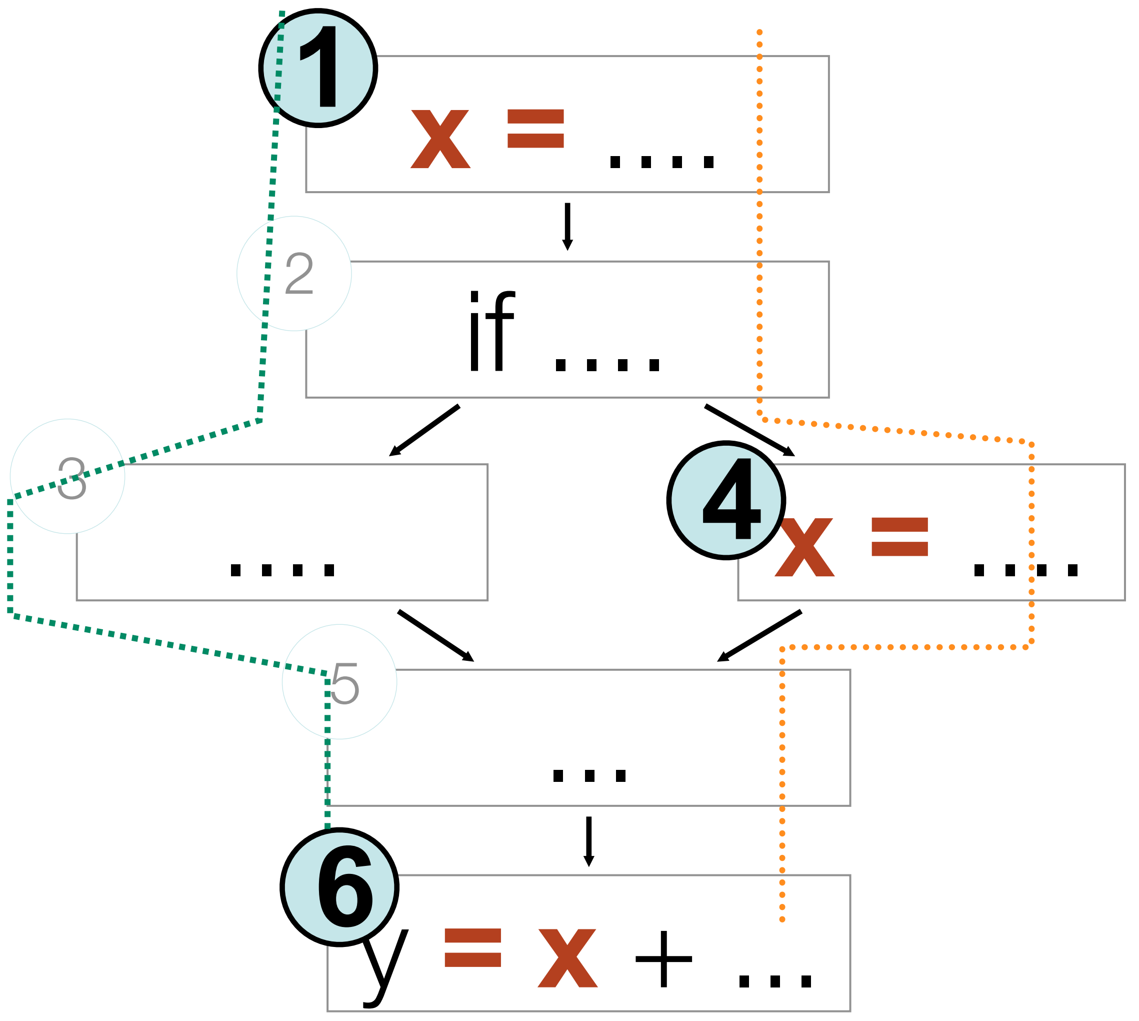
- DU pair: a pair of *definition* and *use* for some variable, such that at least one DU path exists from the definition to the use

    x = ... is a *definition* of x

    = ... x ... is a *use* of x

- DU path: a definition-clear path on the CFG starting from a definition to a use of a same variable
    - Definition clear: Value is not replaced on path
    - Note – loops could create infinite DU paths between a def and a use

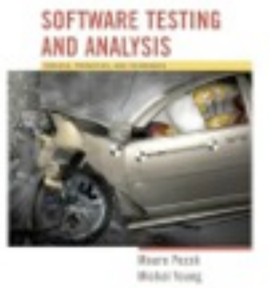
Definition-clear path



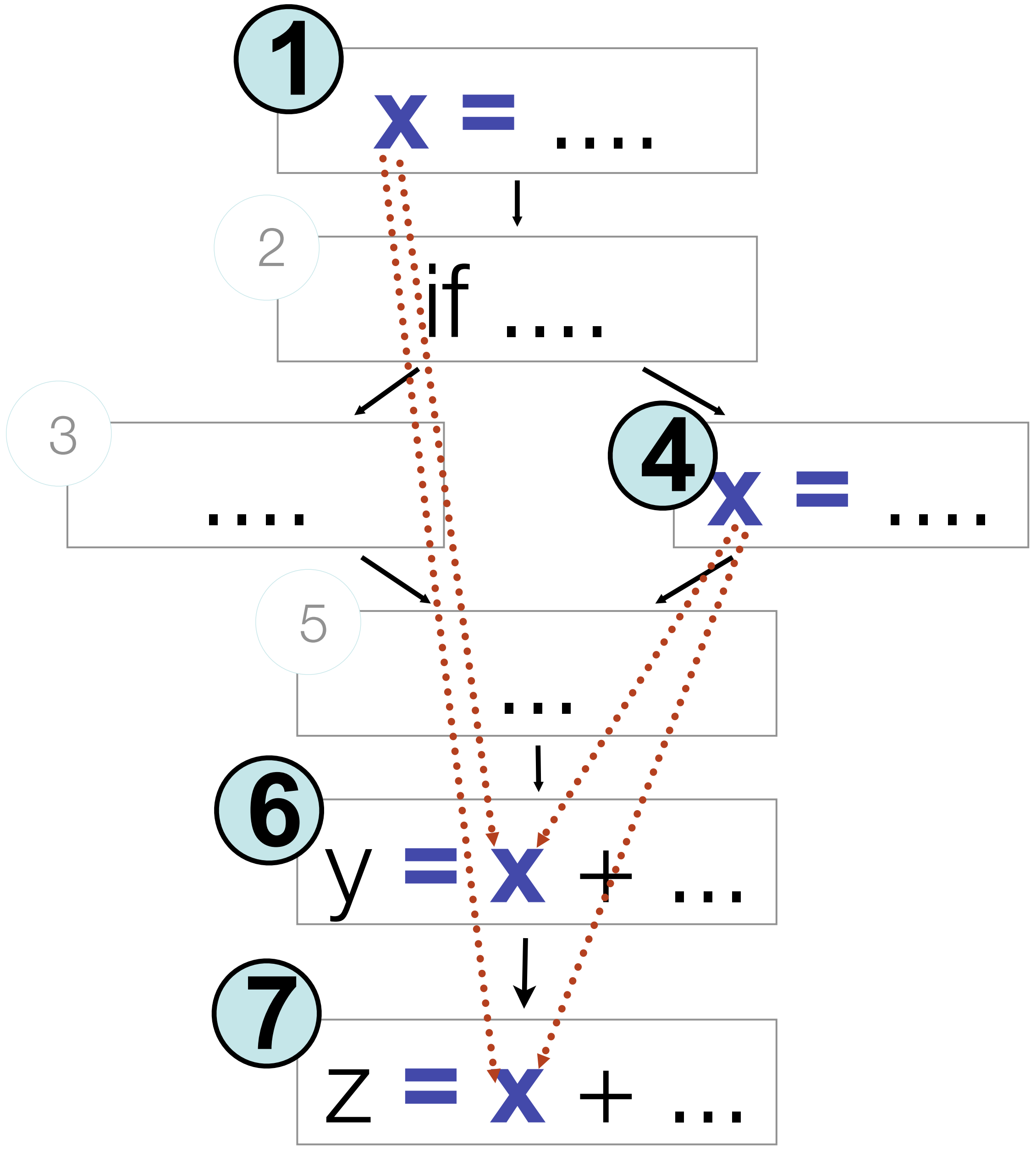
- 1,2,3,5,6 is a definition-clear path from 1 to 6
  - x is not re-assigned between 1 and 6
- 1,2,4,5,6 is not a definition-clear path from 1 to 6
  - the value of x is "killed" (reassigned) at node 4
- (1,6) is a DU pair because 1,2,3,5,6 is a definition-clear path

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)

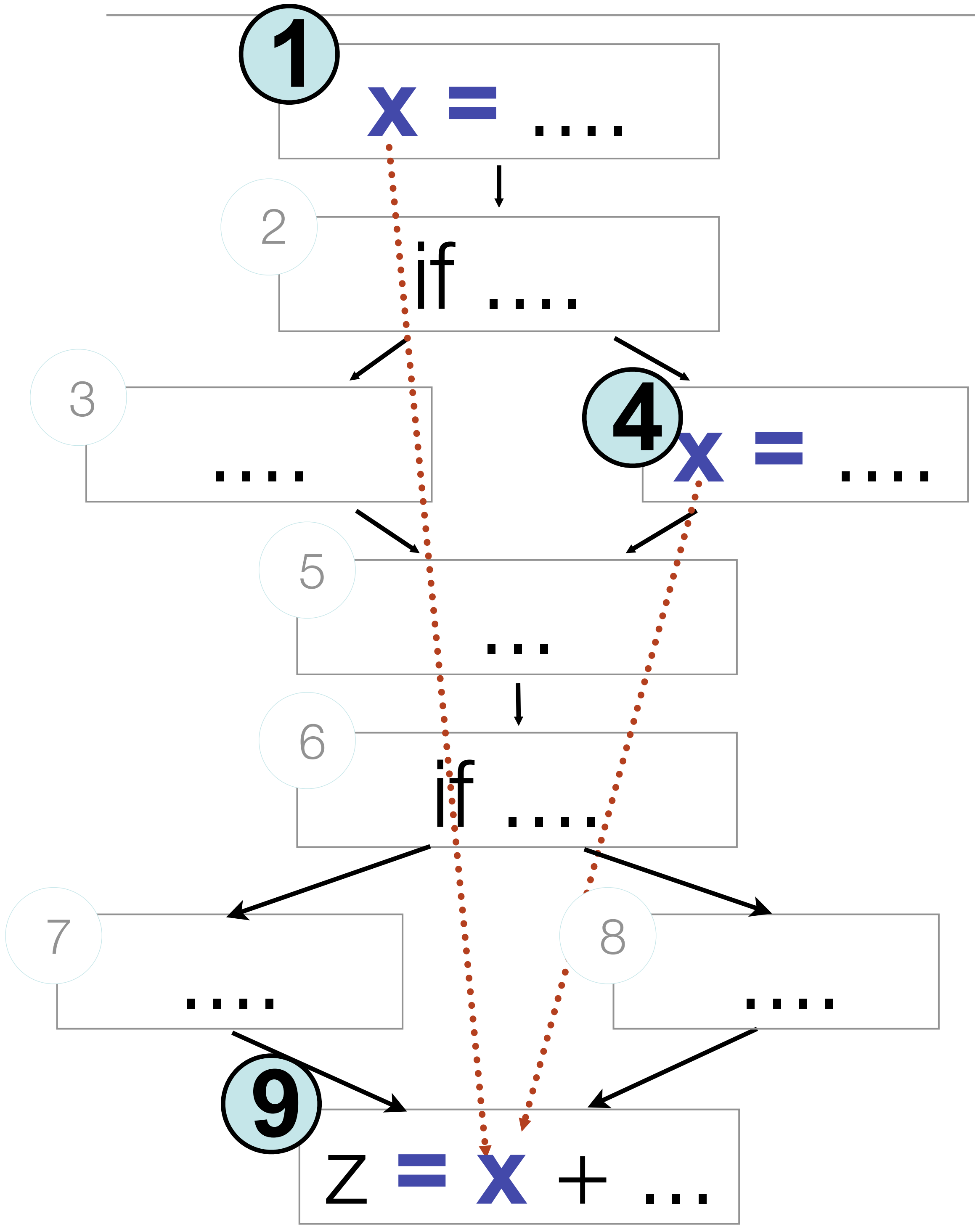


All du pairs (all-uses)



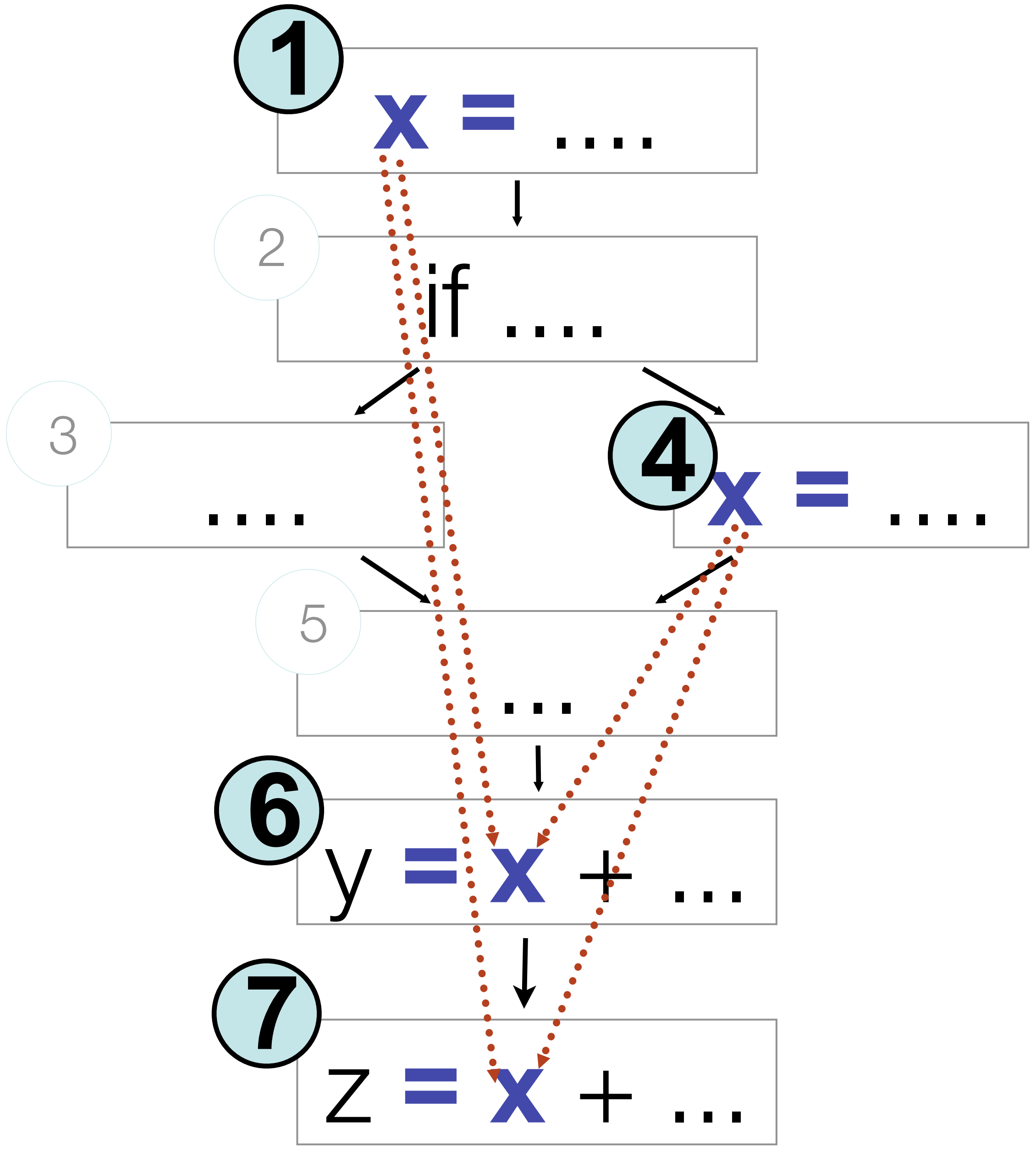
- Requires to cover all the following pairs:
  - def at 1 - use at 6
  - def at 1 - use at 7
  - def at 4 - use at 6
  - def at 4 - use at 7

All du paths



- Requires to cover all the following pairs:
  - def at 1 - use at 9 (through 7)
  - def at 1 - use at 9 (through 8)
  - def at 4 - use at 9 (through 7)
  - def at 4 - use at 9 (through 8)

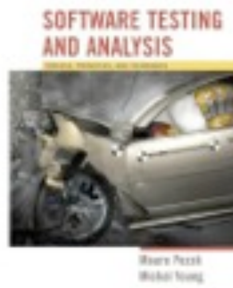
All definitions



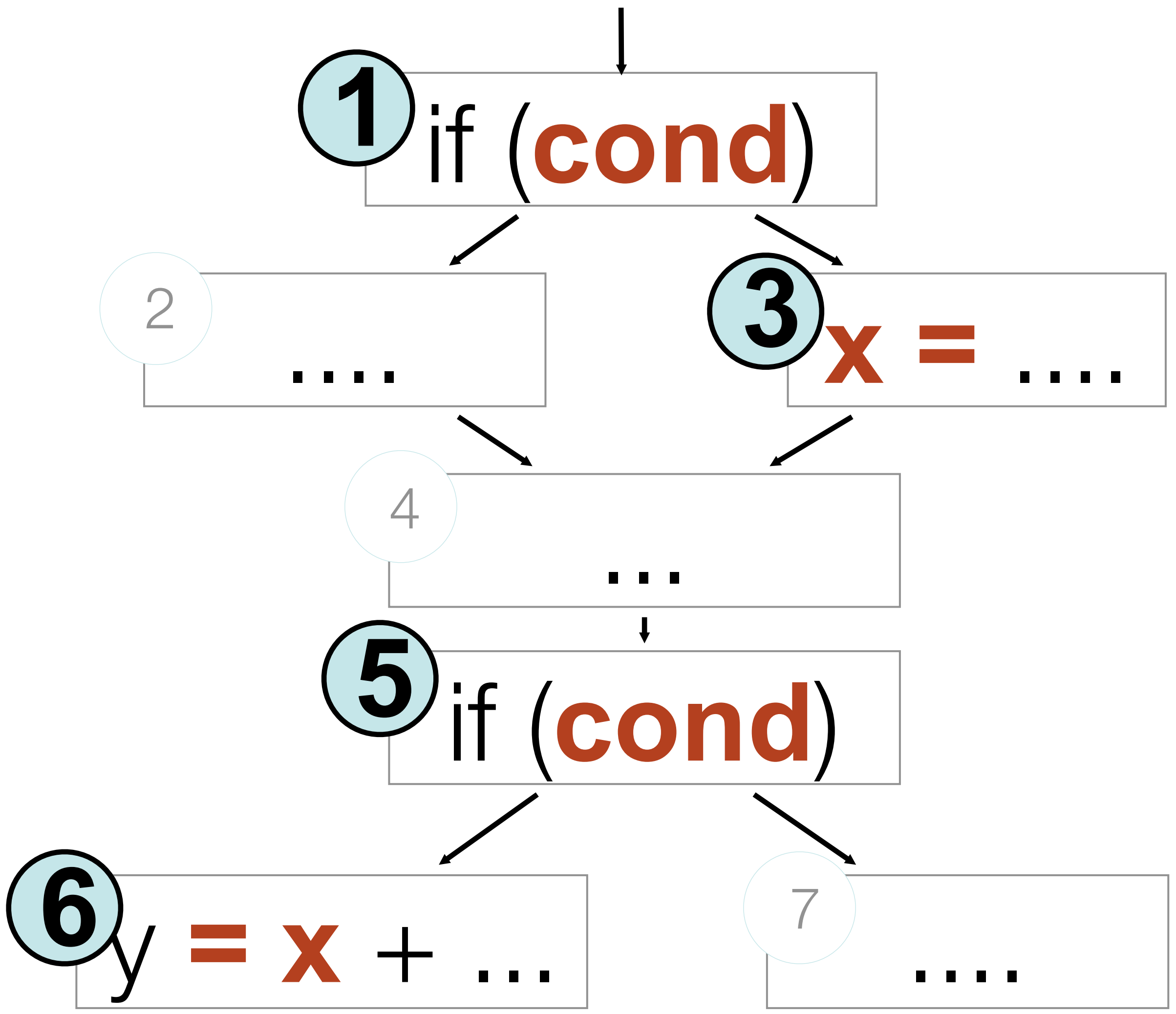
- Requires to cover 2 pairs:
  - def at 1 - use at 6

    OR
  - def at 1 - use at 7
  - def at 4 - use at 6

    OR
  - def at 4 - use at 7



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

Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

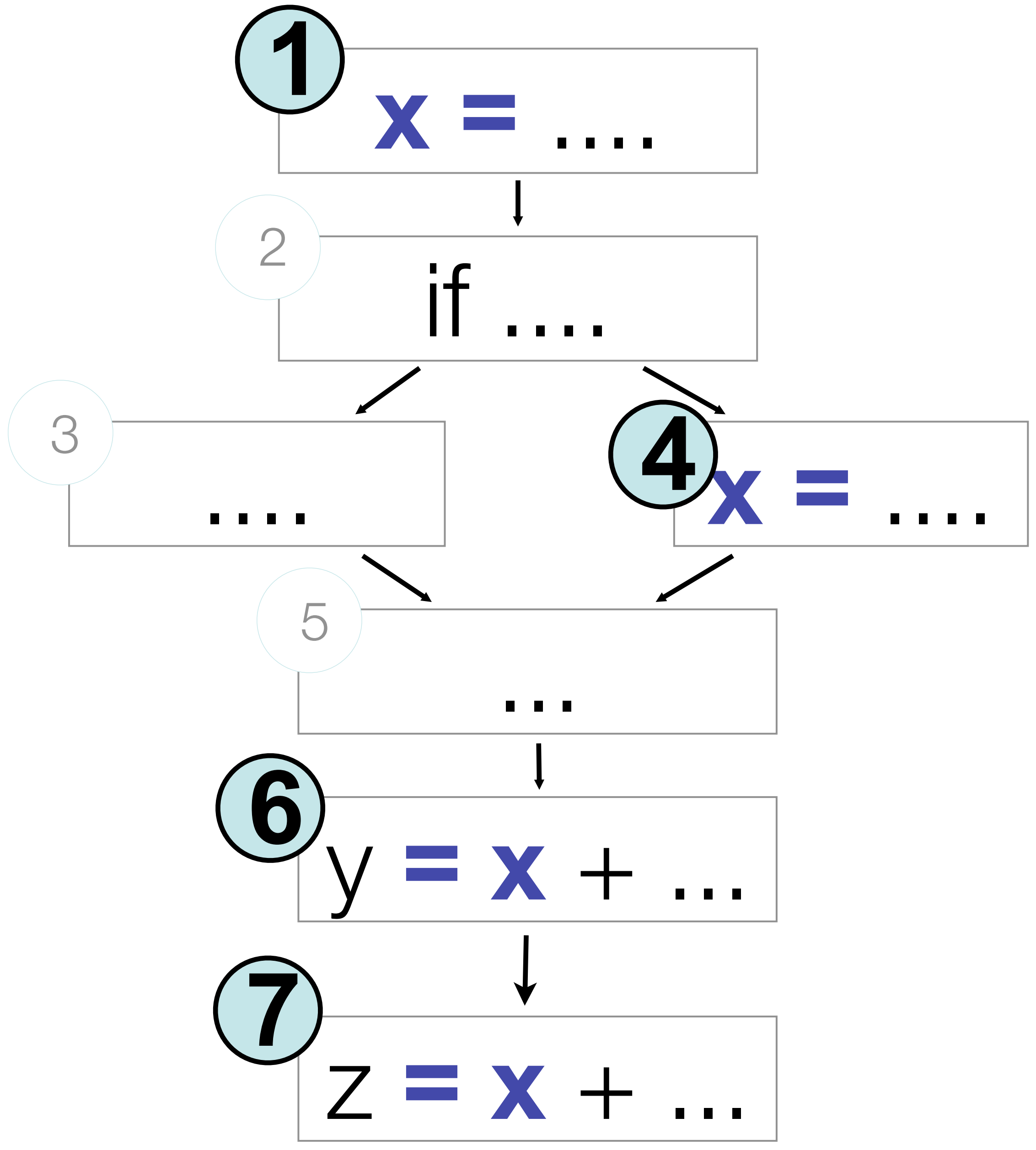
gen(n) = vn where var v is defined at node n

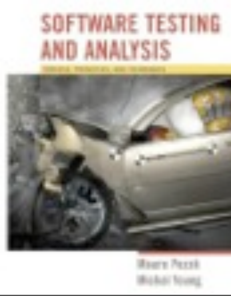
kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in*[n] :=* ∪ *{* ***out****[m]* | *m* ***pred****(n) }*

out[n] := **gen**(n) ∪ (***in****[n]* - ***kill****[n])*





Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

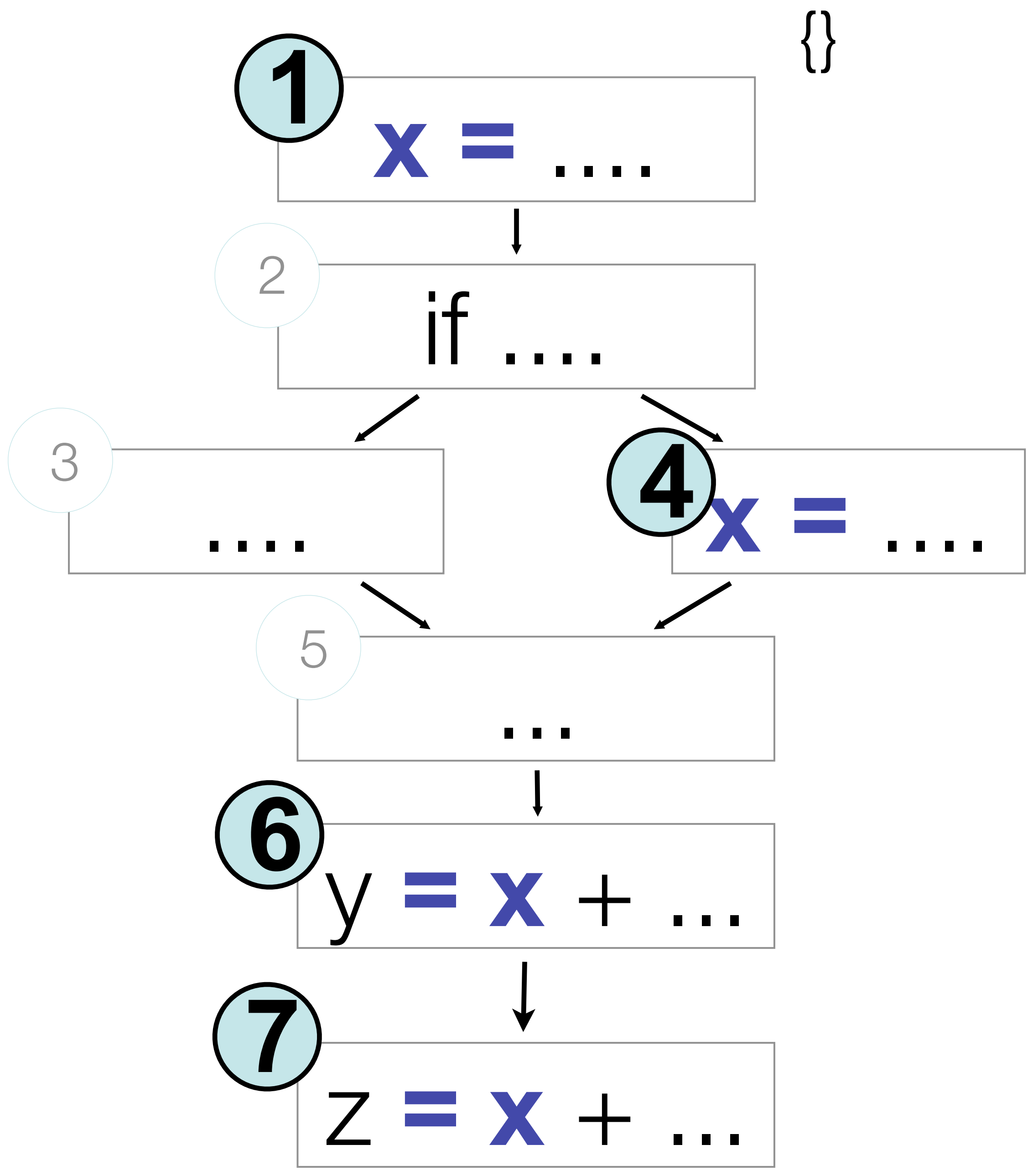
gen(n) = vn where var v is defined at node n

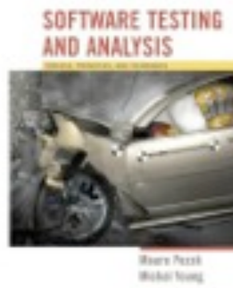
kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in*[n] :=* ∪ *{* ***out****[m] | m* ***pred****(n) }*

out[n] := **gen**(n) ∪ (***in****[n]* - ***kill****[n])*





Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

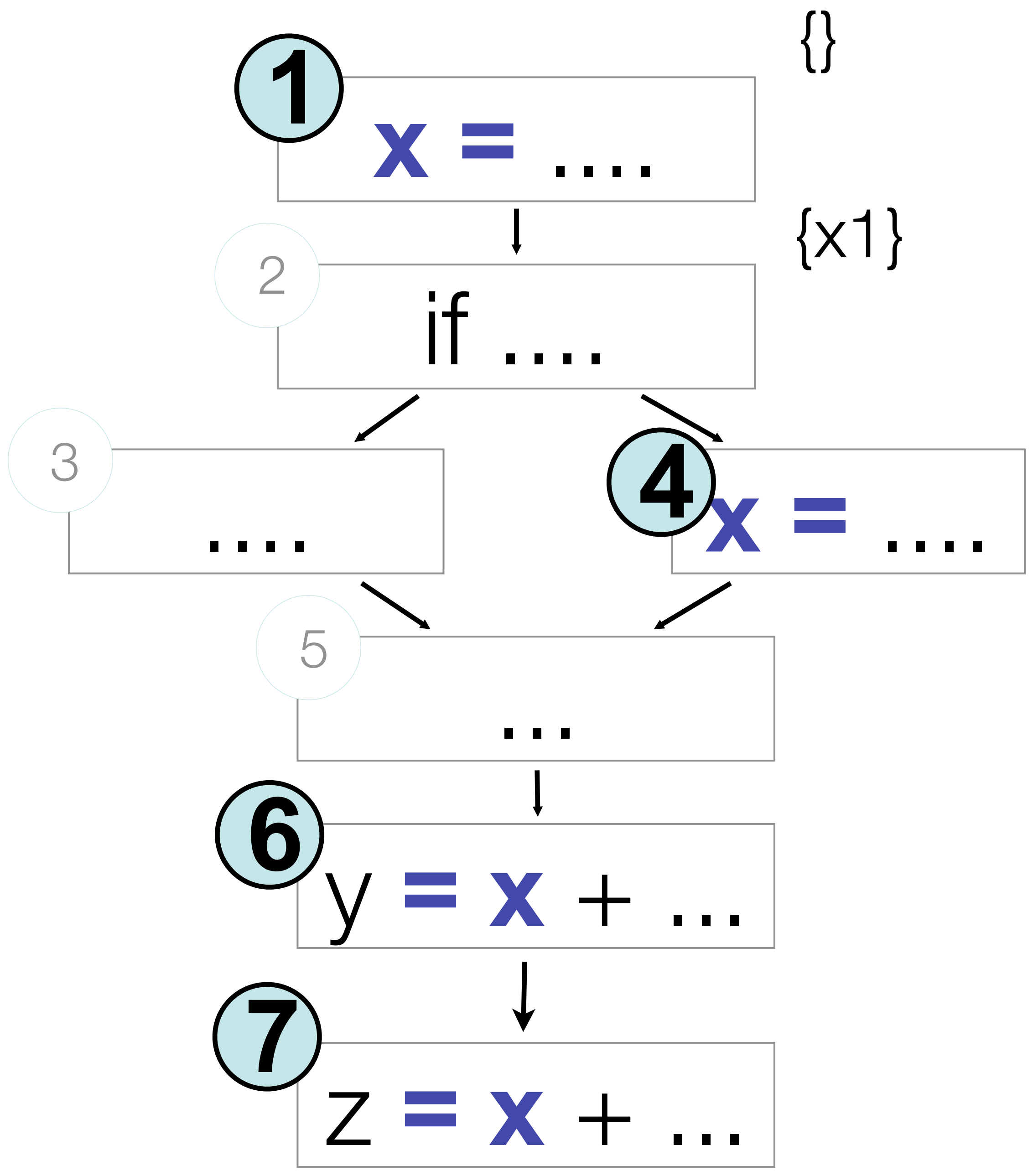
gen(n) = vn where var v is defined at node n

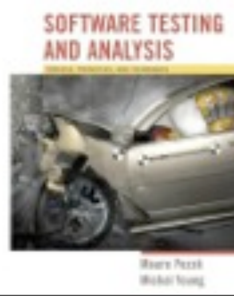
kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in[n] := ∪ { ***out***[m] | m ***pred***(n) }

out[n] := **gen**(n) ∪ (***in***[n] - ***kill***[n])





Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

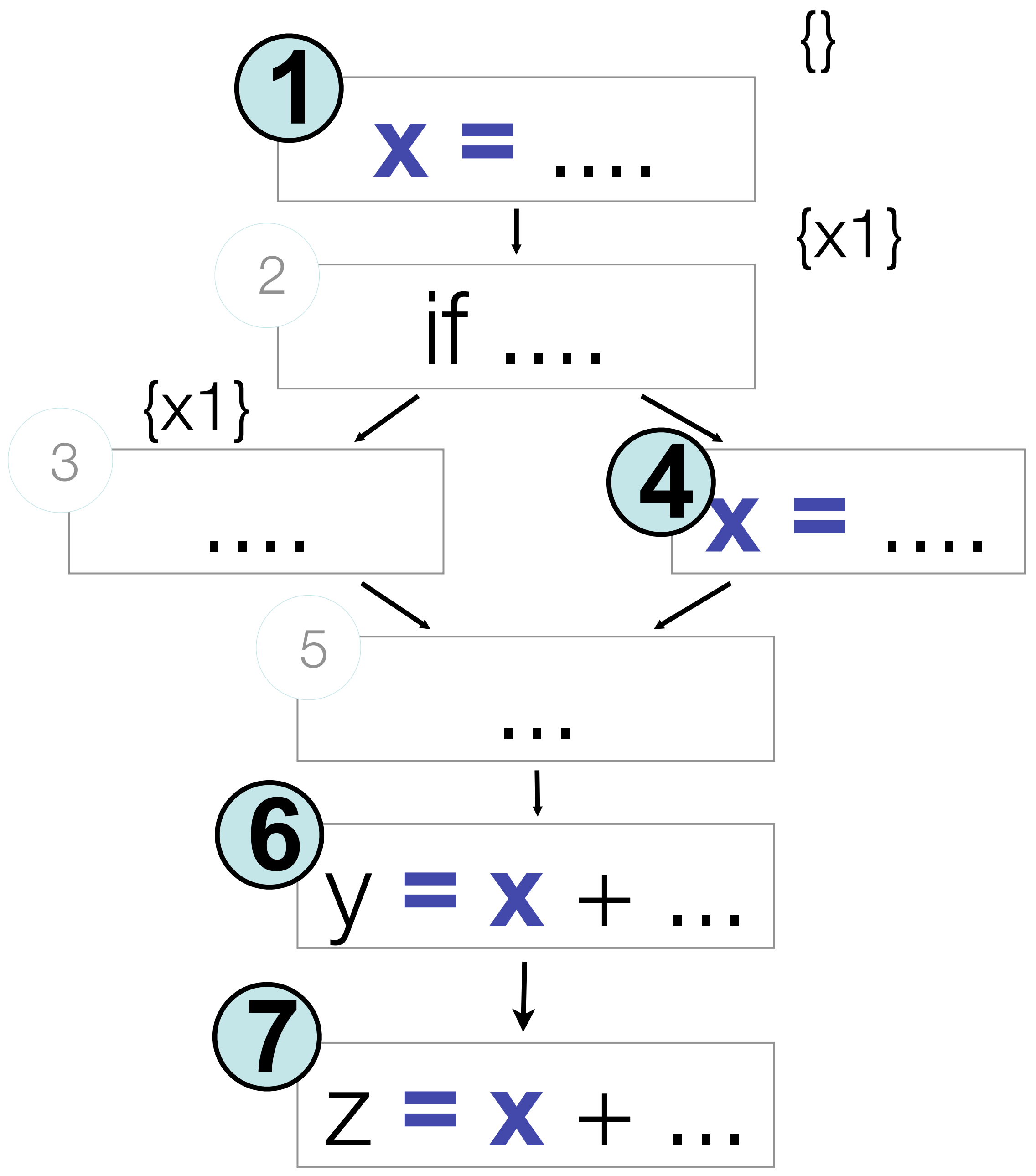
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in*[n] :=* ∪ *{* ***out****[m]* | *m* ***pred****(n) }*

out[n] := **gen**(n) ∪ (***in****[n]* - ***kill****[n])*



Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

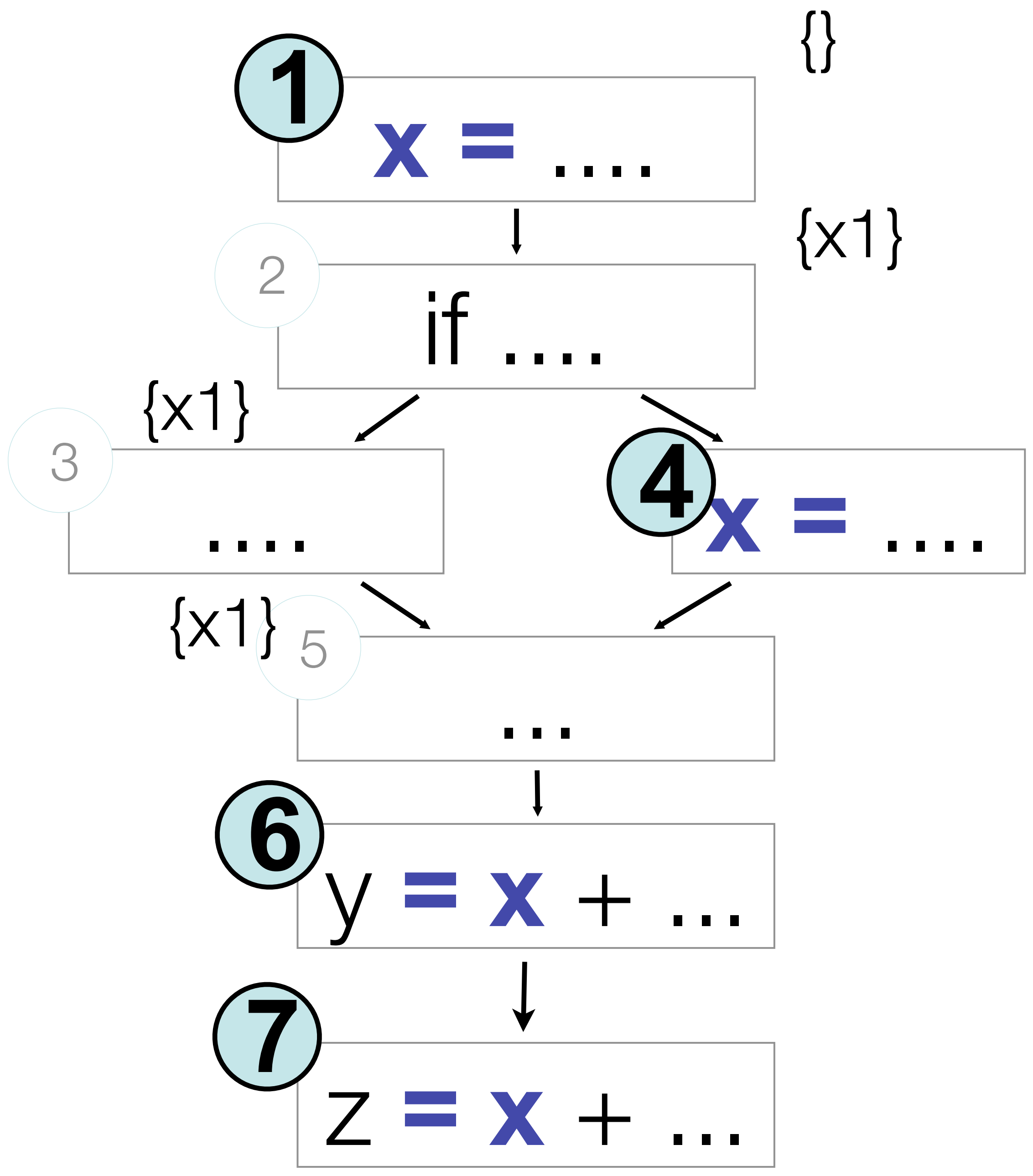
gen(n) = vn where var v is defined at node n

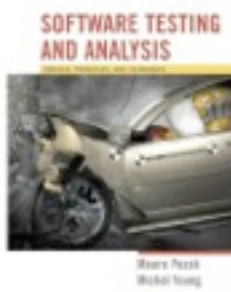
kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

***in**[n] :=* $\bigcup$ *{ **out**[m] | m **pred**(n) }*

out**[n] :=* **gen**(n) ∪ (in**[n] - **kill**[n])*





Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

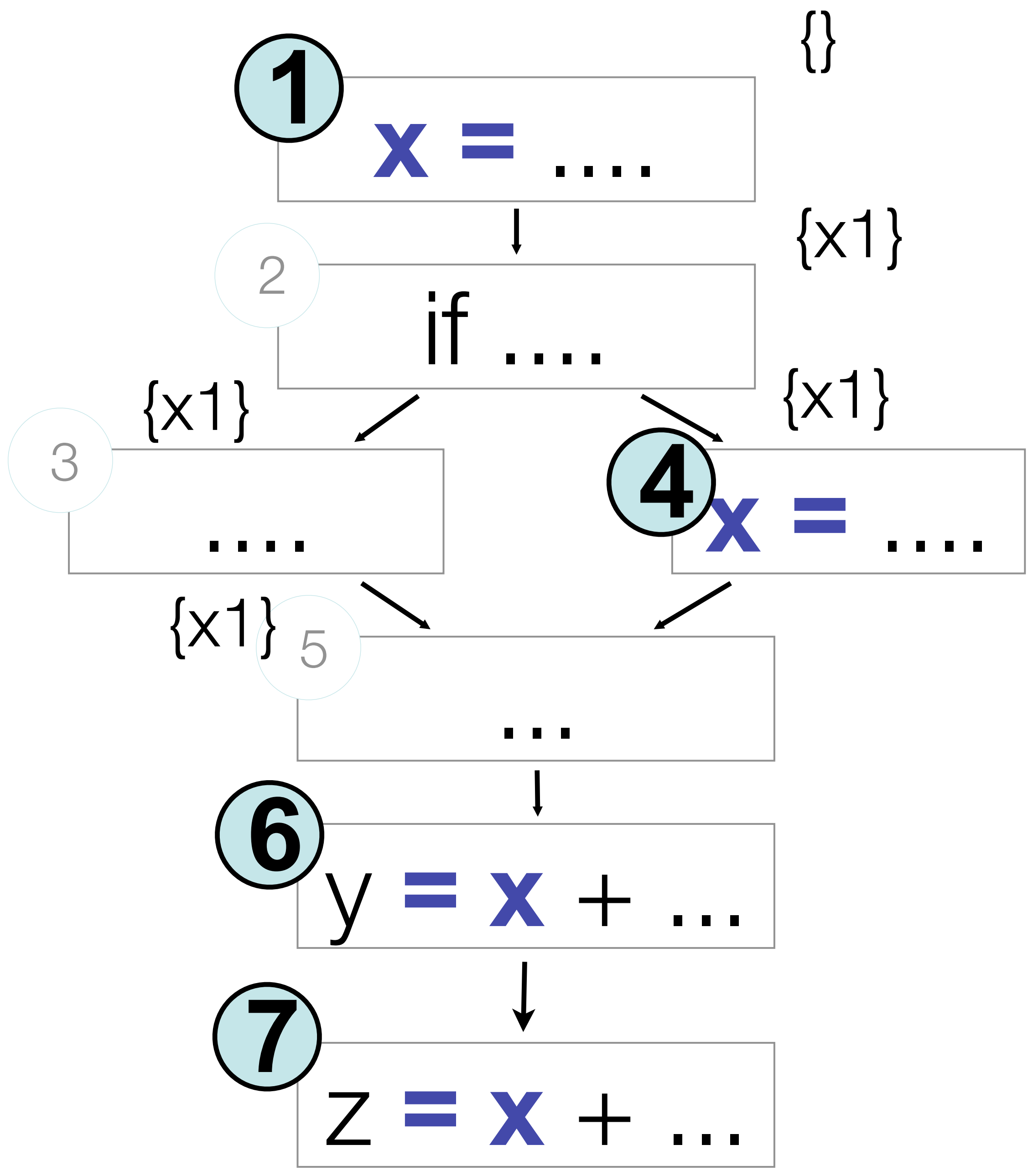
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in[*n*] := ∪ { ***out***[*m*] | *m* ***pred***(*n*) }

out[n] := **gen**(n) ∪ (***in***[*n*] - ***kill***[*n*])



Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

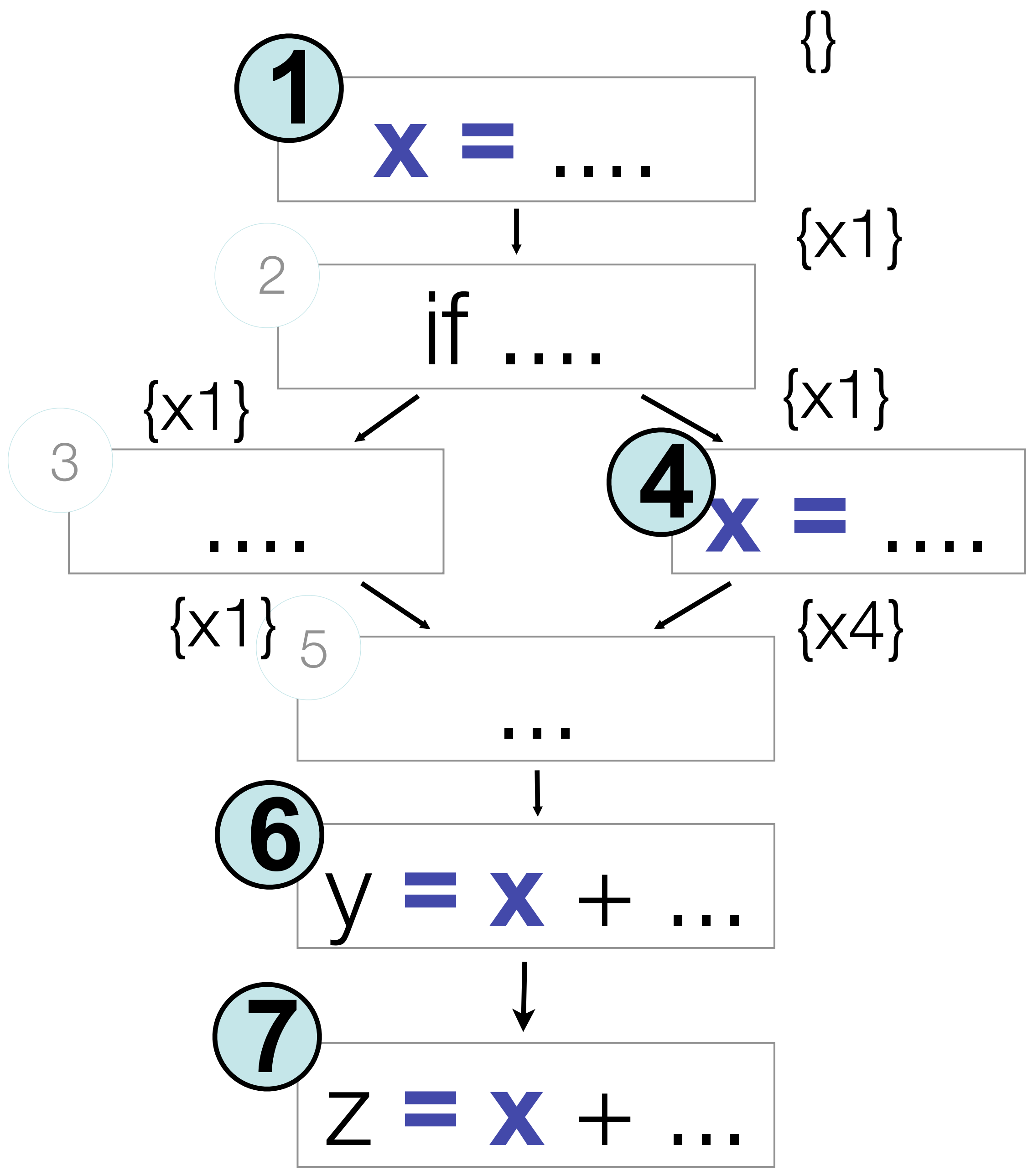
gen(n) = vn where var v is defined at node n

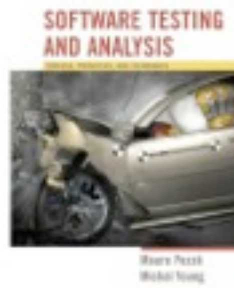
kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in[n] := ∪ { ***out***[m] | m ***pred***(n) }

out[n] := **gen**(n) ∪ (***in***[n] - ***kill***[n])





Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

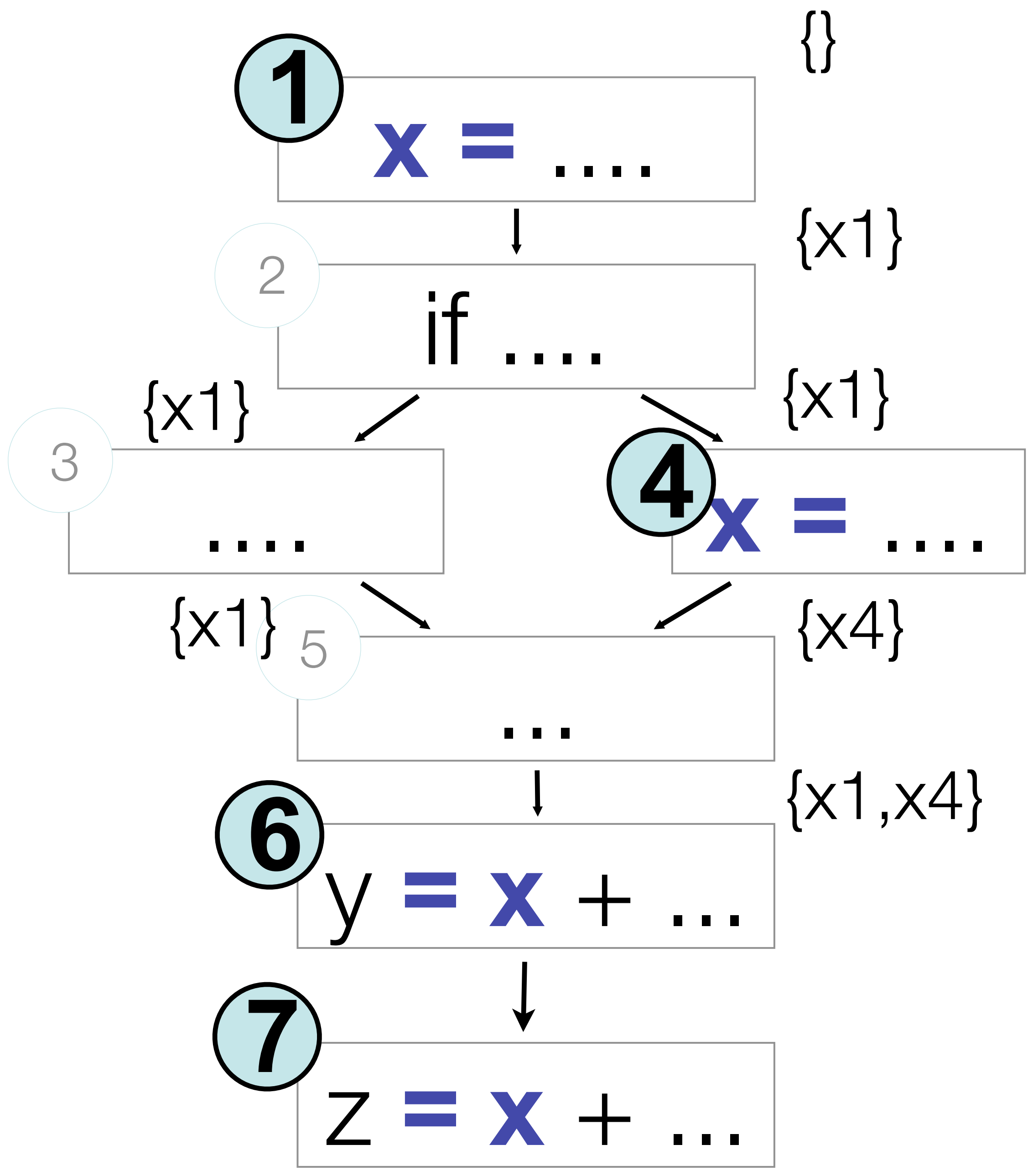
  gen(n) = vn where var v is defined at node n

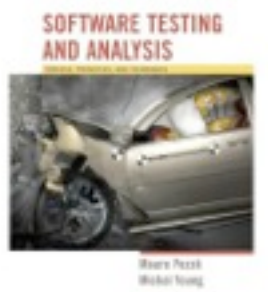
  kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in*[n] :=* ∪ *{* ***out****[m]* | *m* ***pred****(n) }*

out[n] := **gen**(n) ∪ (***in****[n]* - ***kill****[n])*





Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

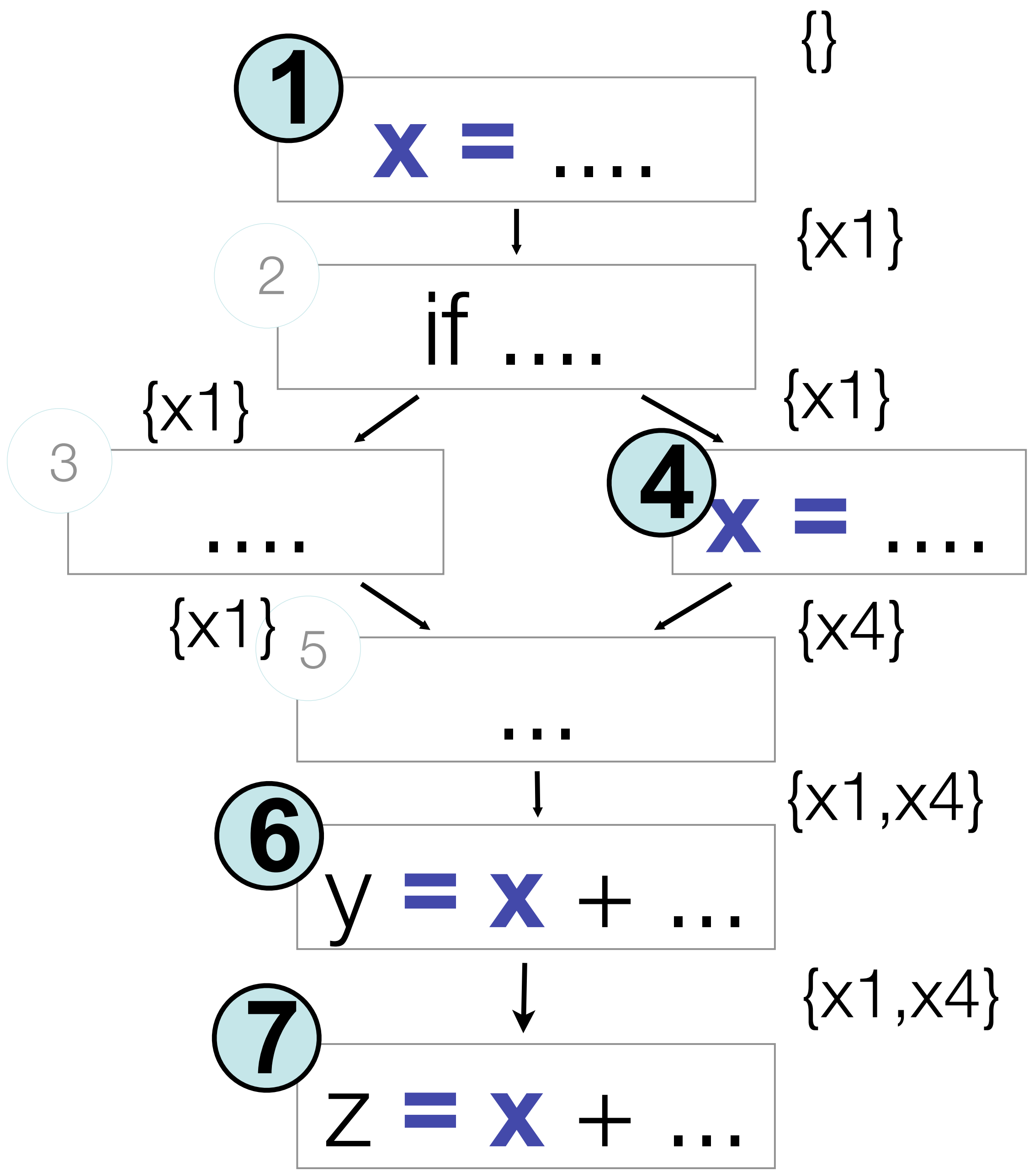
gen(n) = vn where var v is defined at node n

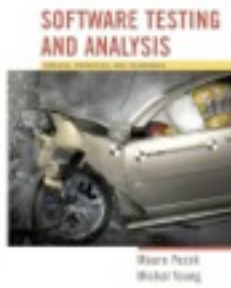
kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in*[n] :=* ∪ *{* ***out****[m] | m* ***pred****(n) }*

out[n] := **gen**(n) ∪ (***in****[n] -* ***kill****[n])*





Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

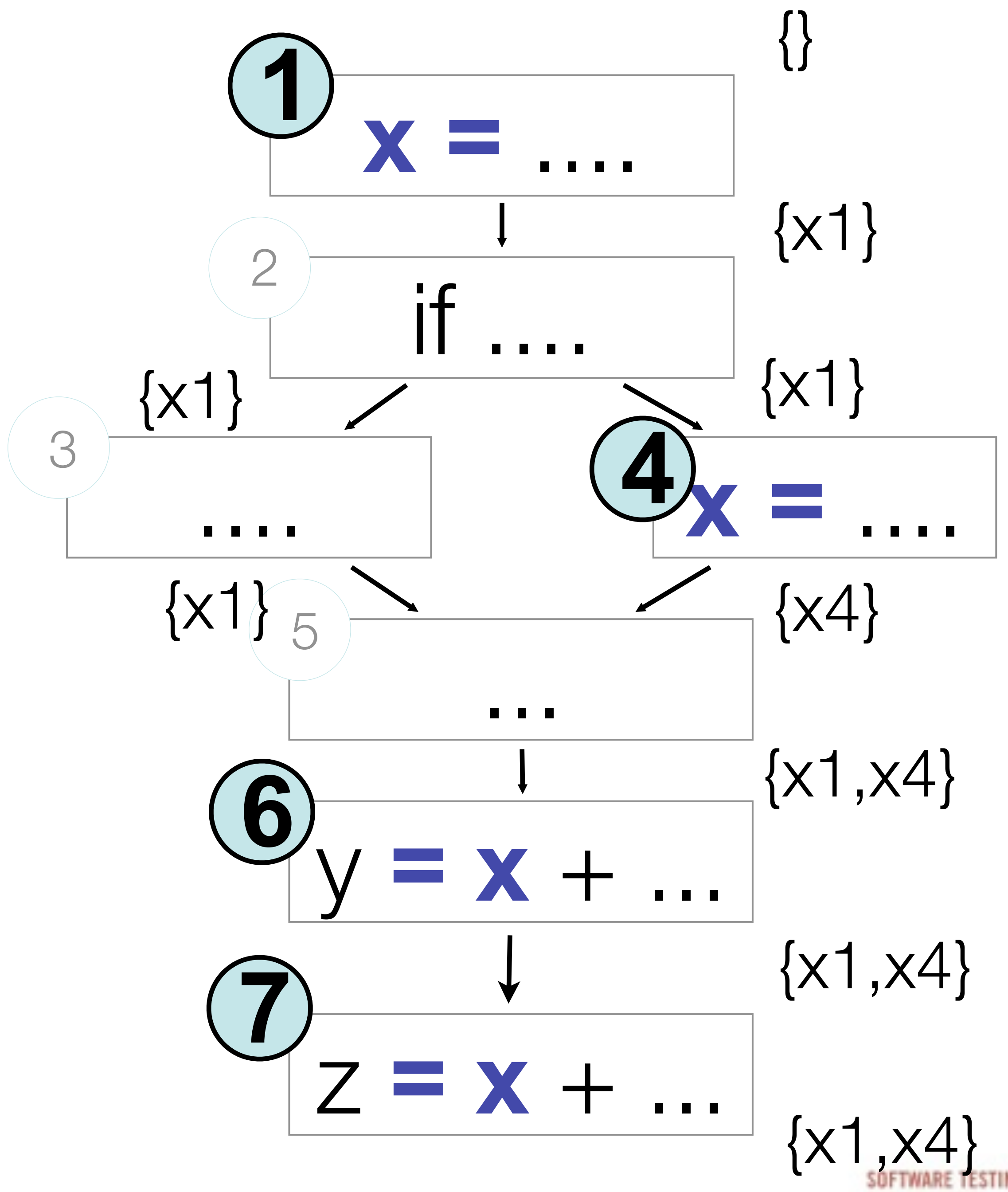
gen(n) = vn where var v is defined at node n

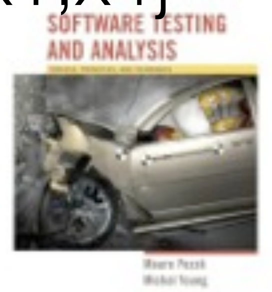
kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in*[n]* := ∪ *{* ***out****[m]* | *m* ***pred****(n) }*

out[n] := **gen**(n) ∪ (***in****[n]* - ***kill****[n])*





Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

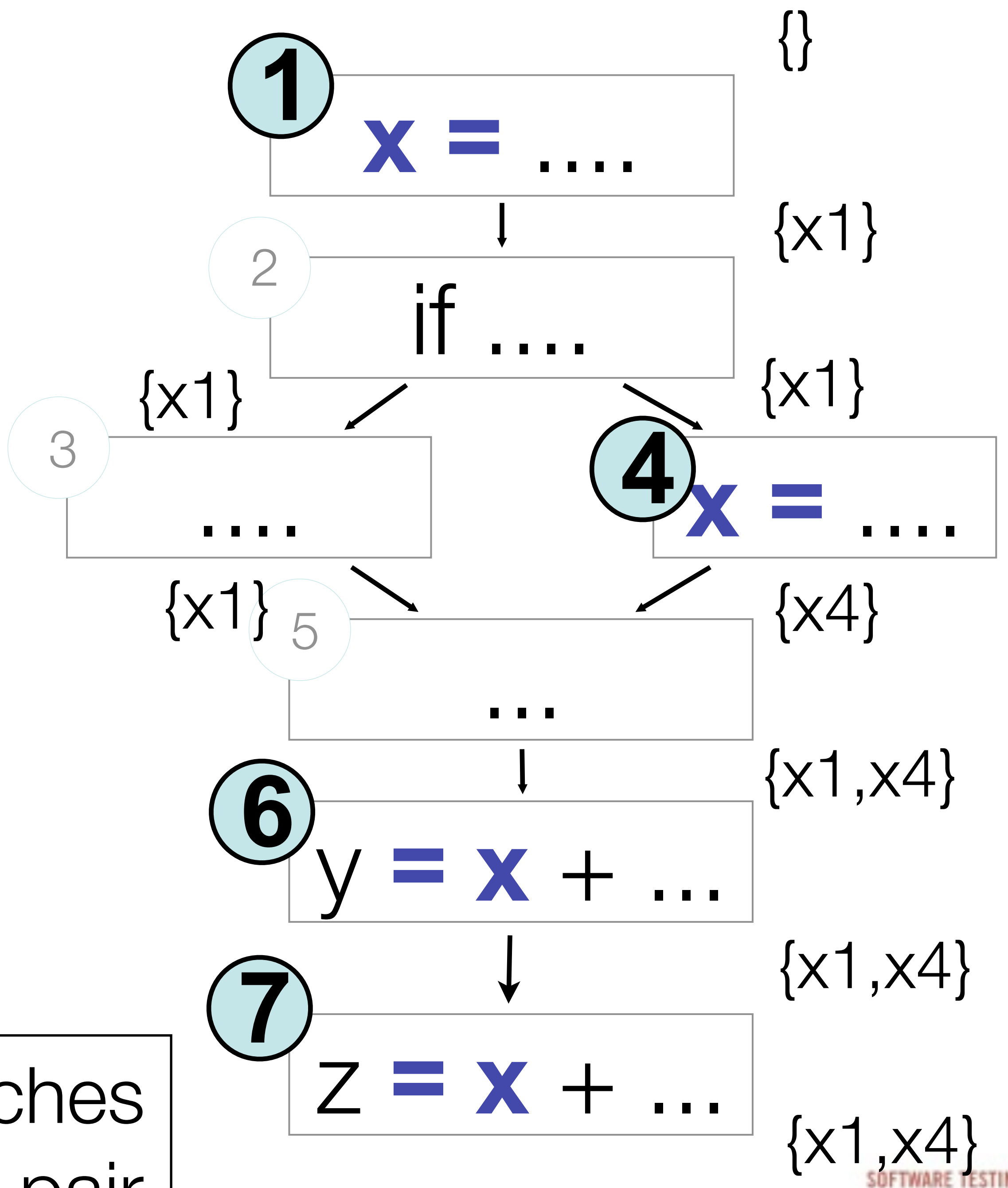
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in[*n*] := ∪ { ***out***[*m*] | *m* ***pred***(*n*) }

out[n] := **gen**(n) ∪ (***in***[*n*] - ***kill***[*n*])



Every time a definition of variable x reaches a use of variable x we found a new DU pair

Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

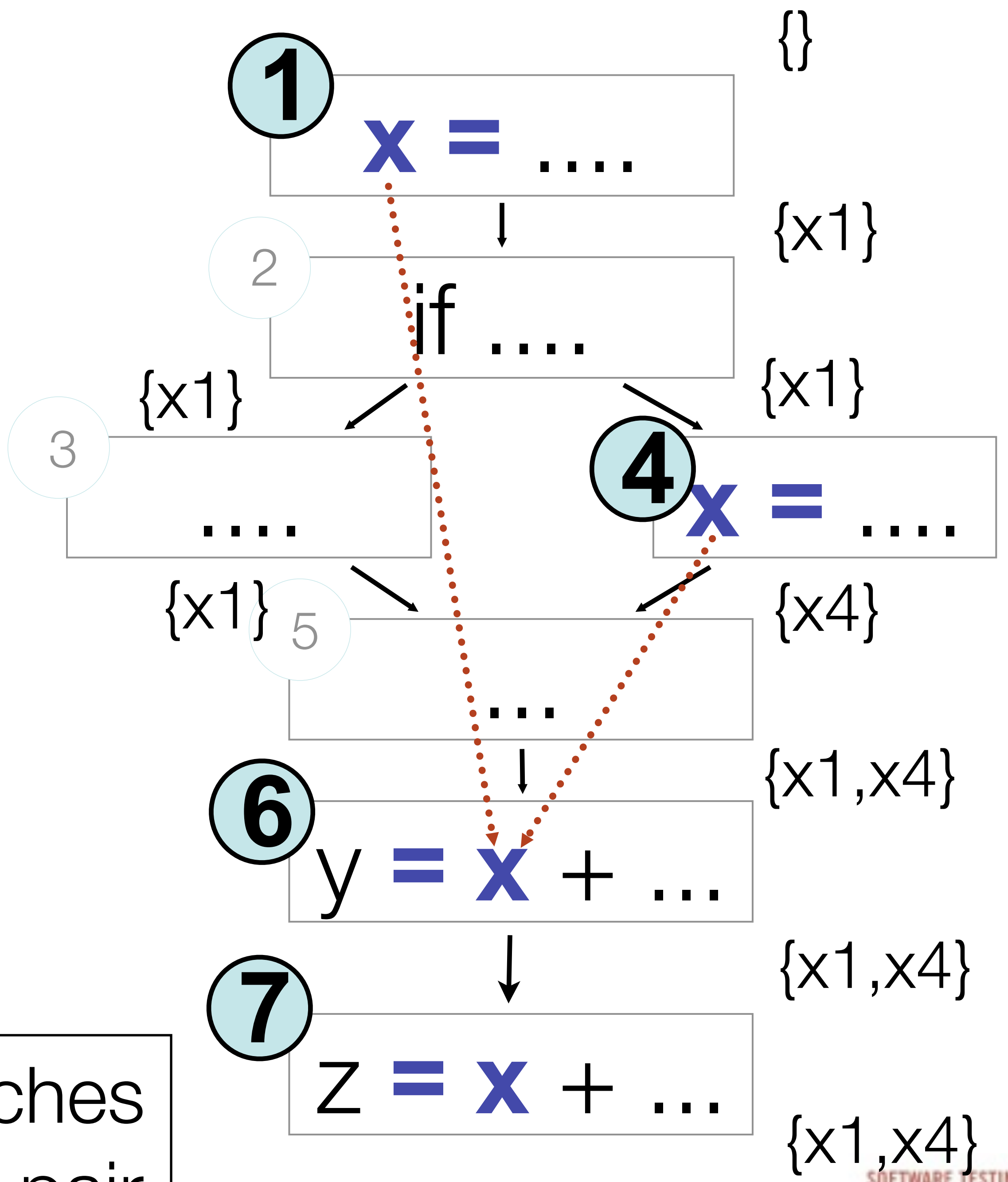
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in[*n*] := ∪ { ***out***[*m*] | *m* ***pred***(*n*) }

out[n] := **gen**(n) ∪ (***in***[*n*] - ***kill***[*n*])



Every time a definition of variable x reaches a use of variable x we found a new DU pair

Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

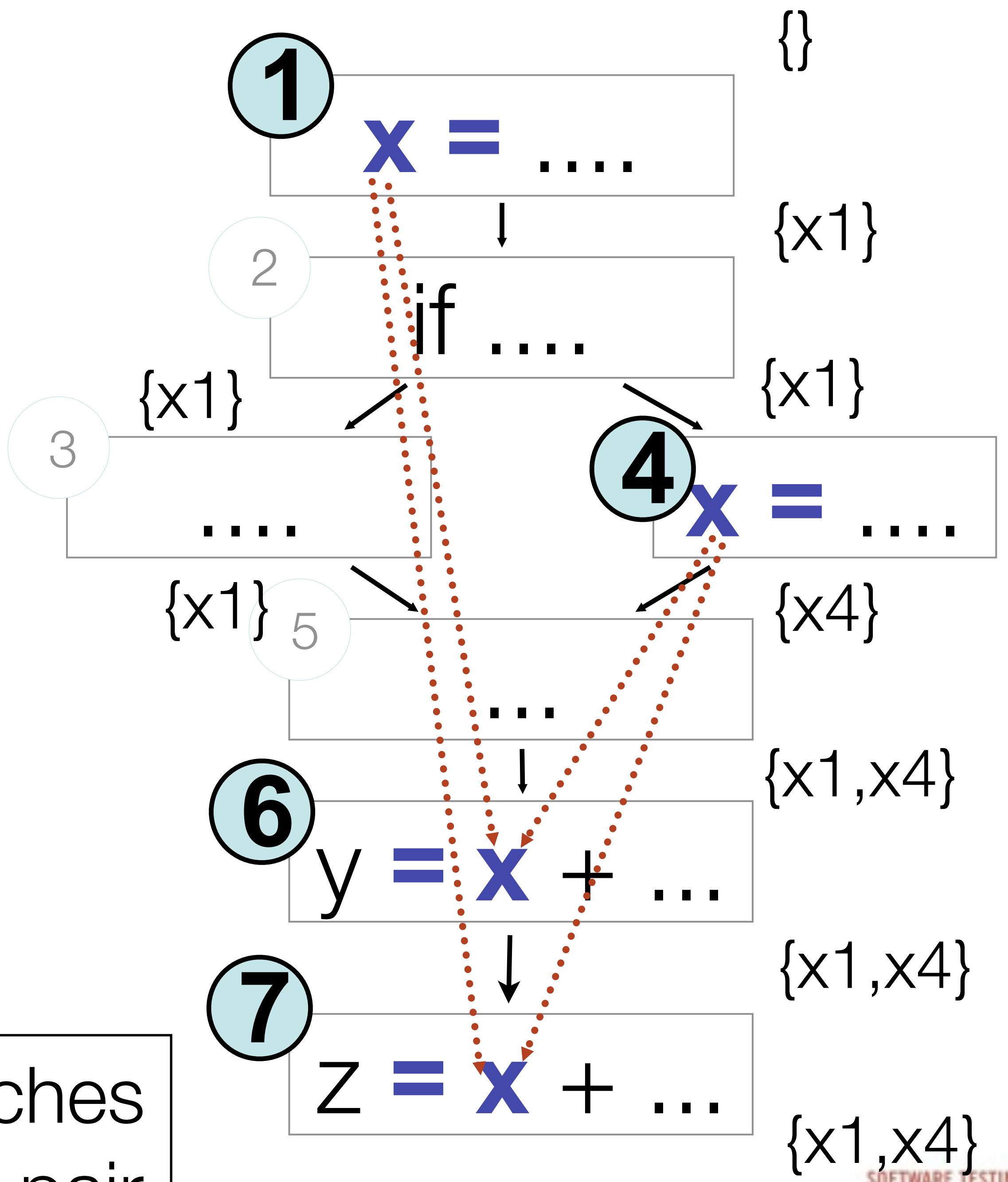
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in[*n*] := ∪ { ***out***[*m*] | *m* ***pred***(*n*) }

out[n] := **gen**(n) ∪ (***in***[*n*] - ***kill***[*n*])

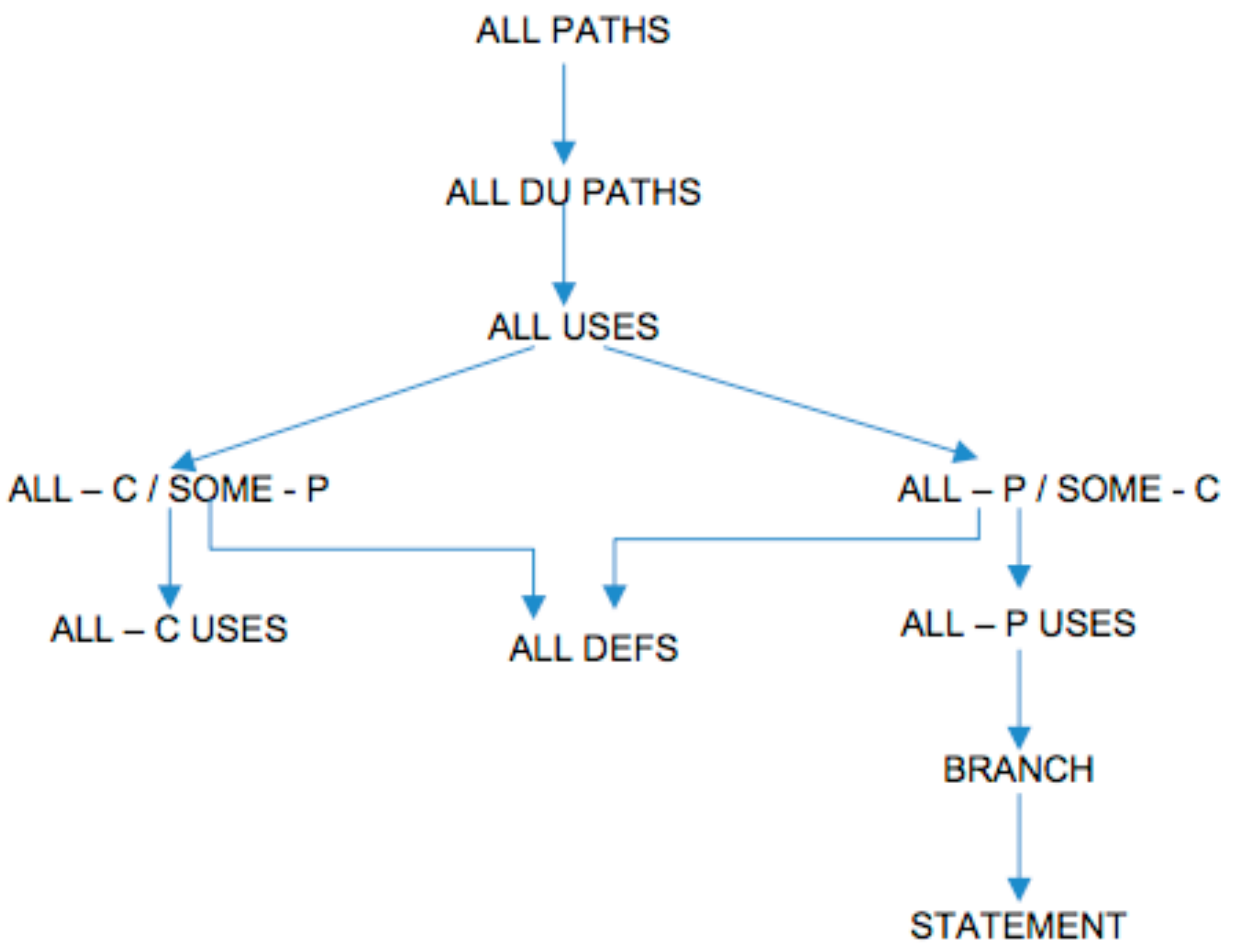


Every time a definition of variable x reaches a use of variable x we found a new DU pair

Other data flow criteria

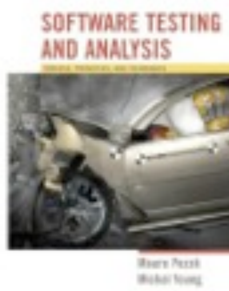
- Some criteria distinguish between computational use (c-use) and predicate use (p-use)
- **All c-uses:** at least one path from definition to each c-use
- **All p-uses:** at least one path from definition to each p-use (considering T and F branches)
- **All c-uses/some p-uses**: at least one path from definition to each c-use. If there is no c-use then to some p-use
- **All p-uses/some c-uses**: at least one path from definition to each p-use (T and F). If there is no p-use then to some c-use
- **All-uses**: at least one path from definition to each c-use and p-use (T and F).

Subsumes relation between data flow criteria



Data-flow criteria for object oriented code

- Structural criteria focus on testing the logic of single procedures.
- In object oriented code it is also important to test the interaction among different methods
- *Intuition*: The behavior of a method may depend on the execution of other methods.



Example

```
class Coinbox{
  int totalQtrs;
  int curQtrs;
  boolean allowVend;

  public CoinBox(){
    totalQtrs=0;
    curQtrs=0;
    allowVend=false;
  }

  public void returnQtrs(){
    curQtrs = 0
  }

  public void addQtr() {
    curQtrs++;
    allowVend = true;
  }

  public vend() {
    if(allowedVend){
      totalQtrs = totalQtrs + curQtrs;
      curQtrs = 0;
      allowVend = false;
    }
  }
}
```

Example

```
class Coinbox{
  int totalQtrs;
  int curQtrs;
  boolean allowVend;

  public CoinBox(){
    totalQtrs=0;
    curQtrs=0;
    allowVend=false;
  }

  public void returnQtrs(){
    curQtrs = 0
  }

  public void addQtr() {
    curQtrs++;
    allowVend = true;
  }

  public vend() {
    if(allowedVend){
      totalQtrs = totalQtrs + curQtrs;
      curQtrs = 0;
      allowVend = false;
    }
  }
}
```

You can drink for free!

Dataflow testing can help

- **Intra-method pairs**: definition and use of the same variable are within the same method, and the def-use pair is exercised during a single invocation of that method.(e.g. allowVend in vend())
- **Inter-method pairs:** definition and use of the same variable are in different methods, and the path from the definition to the use can be found by following the method invocations in the control flow graph.
- **Intra-class pairs:** definition and use of instance variables are in two different methods and the path from the definition to the use can be exercised in the two methods are invocked one after the other (allowVend in addQtr() and vend())

Example

```
class Coinbox{
  int totalQtrs;
  int curQtrs;
  boolean allowVend;

  public CoinBox(){
    totalQtrs=0;
    curQtrs=0;
    allowVend=false;
  }

  public void returnQtrs(){
    curQtrs = 0
  }

  public void addQtr() {
    curQtrs++;
    allowVend = true;
  }

  public vend() {
    if(allowedVend){
      totalQtrs = totalQtrs + curQtrs;
      curQtrs = 0;
      allowVend = false;
    }
  }
}
```

du pair

du pair

requires testing of returnQtrs(); vend()

Limitation of classic structural and dataflow criteria

- Classic structural and dataflow criteria solely consider the structure or the dataflow of the program under test (statements, branches, du pairs...).

Method under test

```
int abs(int n) {
  if (n < 0)
    return n * -1;
  else
    return n;

}
```

TC1

```
public void testAbsWithoutCheck() {
  int x = abs(-4);
}
```

TC2

```
public void testAbsWithCheck() {
  int x = abs(-4);
  assertEquals(4, x);
}
```

TC1 and TC2 test exactly the same functionality, but TC2 is better than TC1because it checks the result.

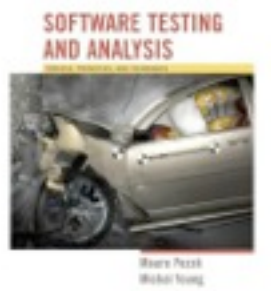
Oracle adequacy

- Defining the adequacy of the oracle can help differentiating test cases with “good” oracle and “bad, incomplete” oracle.
- It gives another guidance to improve the test suites as well.
- Oracle adequacy criteria complement the classic criteria.

Checked coverage

- **Idea**: Use dynamic slicing to determine the checked coverage (statements that were not only executed, but that actually contribute to the results checked by oracles)
- Insufficient oracles result in a low checked coverage.
- Statements that are executed, but whose outcomes are never checked would be considered *uncovered.* One could improve the oracles to actually check these outcomes.

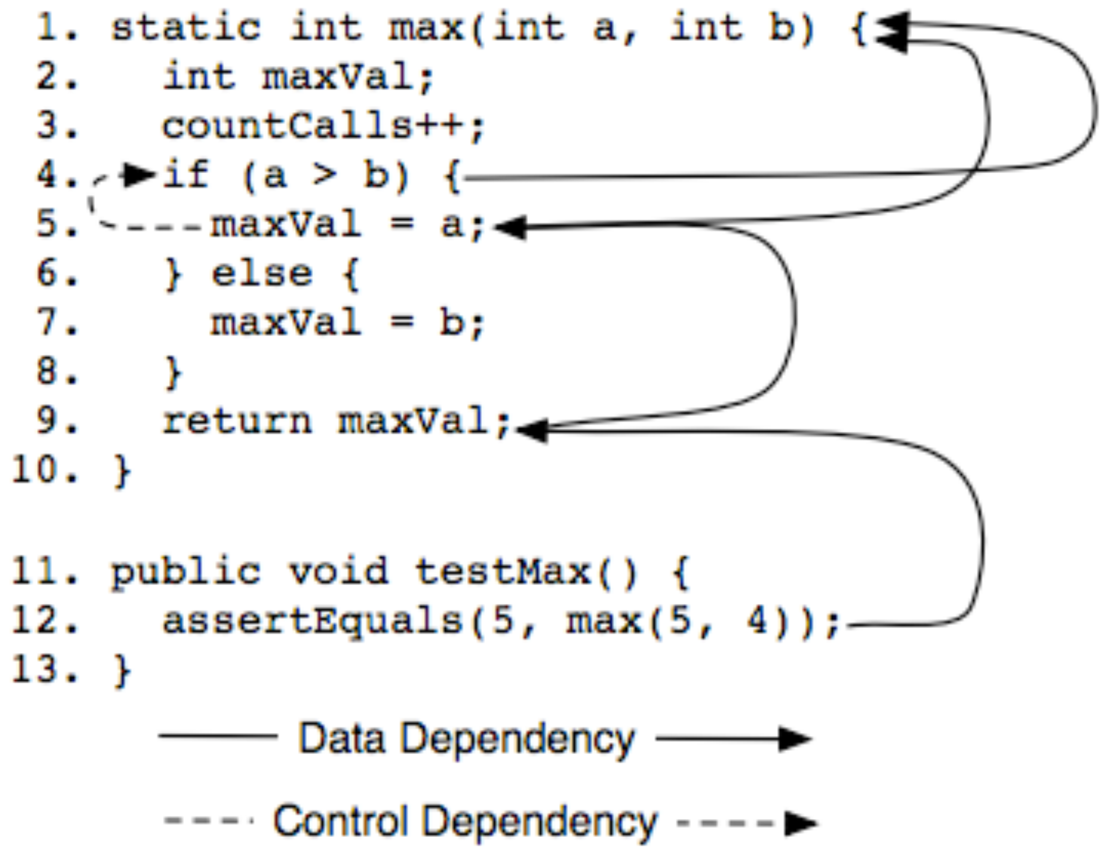
Schuler, Zeller ICST 2011



Checked coverage

- Focus on those features that actually contribute to the results checked by the oracles.
- Compute the dynamic backward slice of test oracles (i.e. all statements that contribute to the checked result). This slice constitutes the checked coverage.
- checked coverage/sliceable statements (statements with defs, uses predicates).

Checked coverage



```
static int max(int a, int b) {
  int maxVal;
  countCalls++;
  if (a > b) {
    maxVal = a;
  } else {
    maxVal = b;
  }
  return maxVal;
}

public void testMax() {
  assertEquals(5, max(5, 4));
}
```

Checked coverage subsumes statement coverage