# *White-box or Structural Testing*

# *Outline*

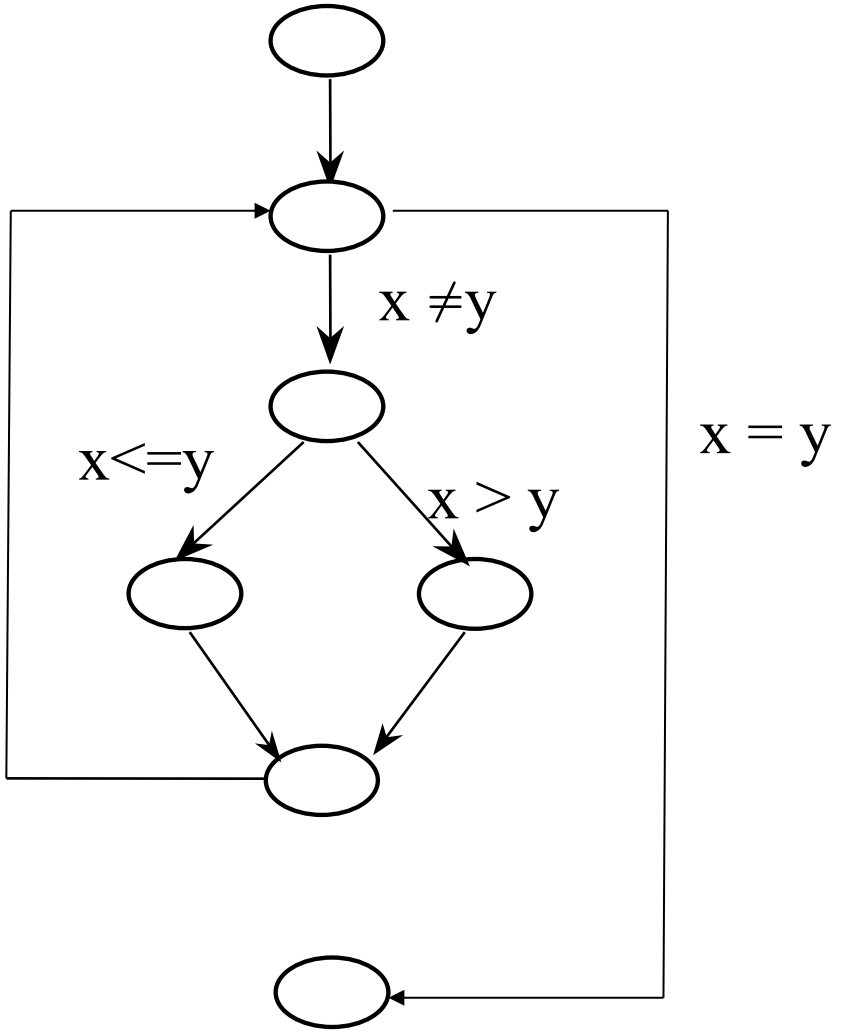
- Control flow coverage
  - Statement, Edge, Condition, Path coverage
- Data flow coverage
  - Definitions-Usages of data
- Analyzing coverage data
- Mutation Testing
- Integration testing
  - Strategies and criteria
- Conclusions
  - Generating test data, tools, Marick's Recommendations

# *Basic Definitions*

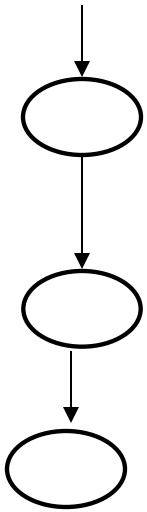
- Directed graph
- Nodes are blocks of sequential statements
- Edges are transfers of control
- Edges may be labeled with predicate representing the condition of control transfer
- Several conventions for flow graphs models with subtle differences

# *Control Flow Graph*

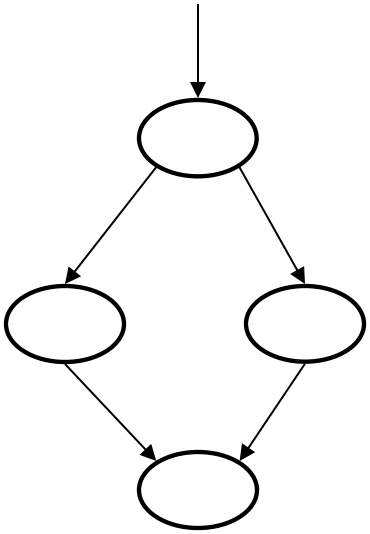
```
read(x);read(y)
while x ≠y loop
   if x>y then
        x := x - y;
   else
        y := y - x;
   end if;
end loop;
gcd := x;
```



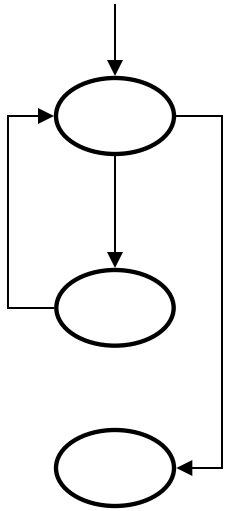
# *Basics of CFG*



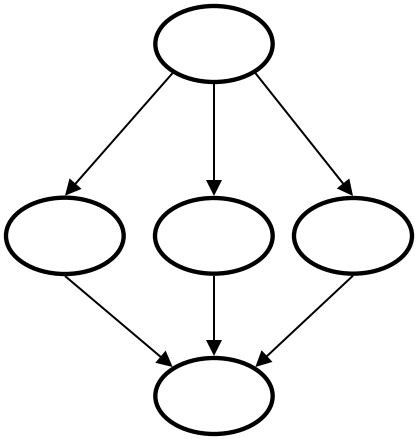
Sequence



If-Then-Else

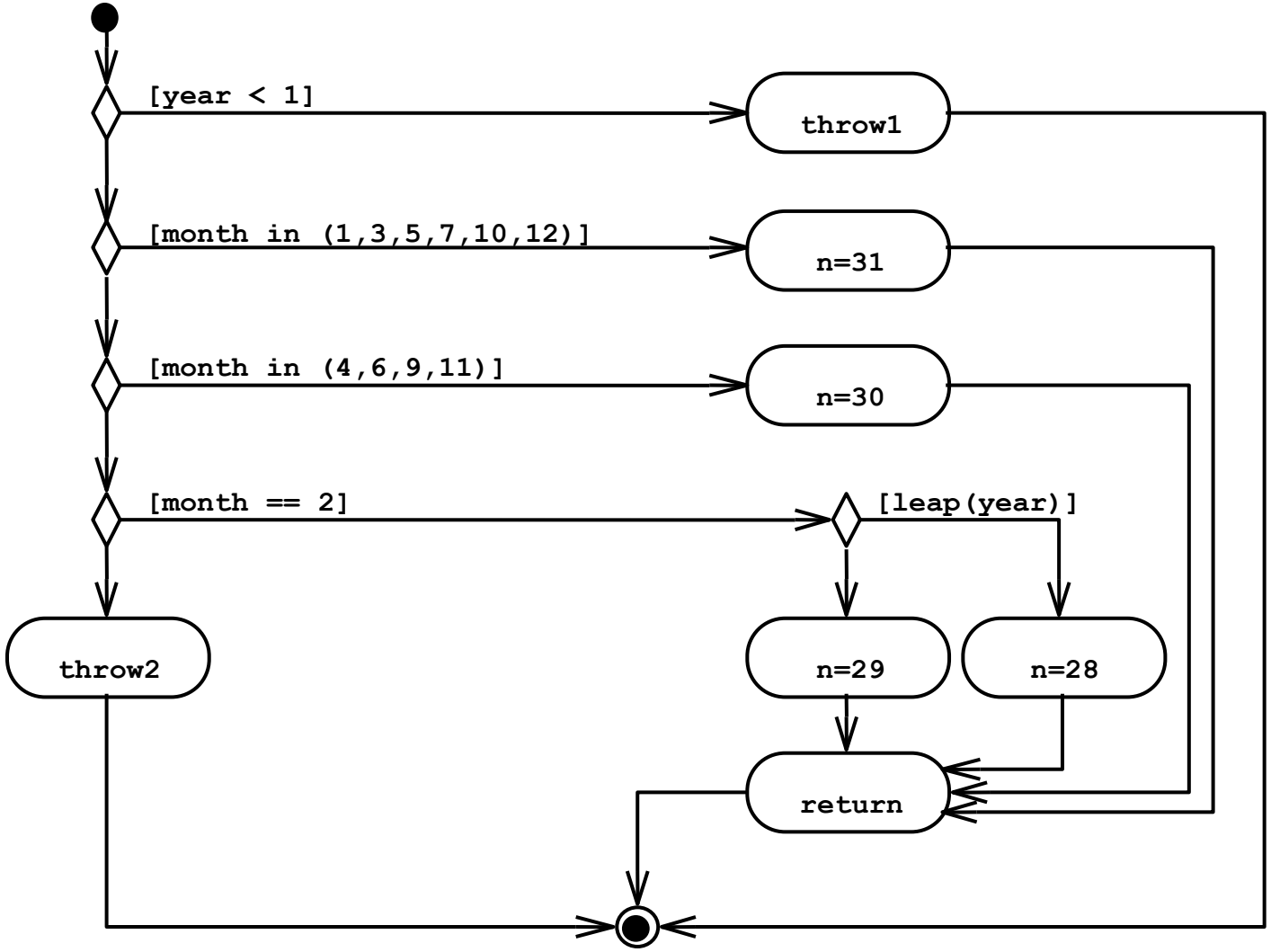


While loop



Switch

# *UML Activity Diagram*



# *Statement/Node Coverage*

- Statement coverage: faults cannot be discovered if the parts containing them are not executed
- Equivalent to covering all nodes in CFG
- Executing a statement is a weak guarantee of correctness, but easy to achieve
- In general, several inputs execute the same statements – important question on practice is how can we minimize test cases?

# *Incompleteness*

- Statement coverage may lead to incompleteness

```
if x < 0 then
        x := -x;
else
        null;
end if
z := x;
```

A negative x would result in the coverage of all statements. But not exercising x >= 0 would not cover all cases (implicit code in italic). However, doing nothing for the case x >= 0 may turn out to be wrong and need to be tested.

# *Edge Coverage*

- Use the program structure, the control flow graph (CFG)
- **Edge coverage criterion:** Select a test set T such that, by executing P for each d in T, each edge of P's control flow graph is traversed at least once
- Exercise all conditions that govern control flow of the program with true and false values

# *Code Example*

```
counter:= 0;
found := false;
if number_of_items != 0 then counter :=1;
   while (not found) and counter < number_of_items loop
        if table(counter) = desired_element then
                found := true;
        end if;
        counter := counter + 1;
   end loop;
end if;
if found then write ("the desired element exists in
           the table");
else write ("the desired element does not exists in
        the table");
end if;
```

≤

# *Test Set*

- We choose a test set with 0 items and a table with 3 items, the second being the desired one (|T| = 2)
- For the second test case, the loop body is executed twice, once executing the **then** branch.
- The edge coverage criterion is fulfilled and the error is not discovered by the test set
- Not all possible values of the *constituents* of the **while loop** condition have been exercised

# *Condition Coverage*

- Further strengthen edge coverage
- **Condition Coverage Criterion:** Select a test set T such that, by executing P for each element in T, each edge of P's control flow graph is traversed, and all possible values of the constituents of compound conditions are exercised at least once
- **Compound conditions:** C1 **and** C2 **or** C3 … where Ci's are relational expressions or boolean variables (atomic conditions)
- **Modified condition coverage**: Only combinations of values such that every Ci drives the overall condition truth value twice (true and false).

# *Uncovering Hidden Edges*

```
if c1 and c2 then
   st;
else
   sf;
end if;
```

```
if c1 then
   if c2 then
        st;
   else
        sf;
   end if;
else
   sf;
end if;
```

- Two equivalent programs
  - though you would write the left one
- Edge coverage
  - would not compulsorily cover the “hidden” edges,
  - Ex.: C2 = false might not be covered
- Condition coverage would

# *Example*

- International Standard DO-178B for airborne systems certification (1992)
- Example : A && (B || C)

| | ABC | Res. | Corr. False Case |
|---|---|---|---|
| 1 | TTT | T | A (5) |
| 2 | TTF | T | A (6), B (4) |
| 3 | TFT | T | A (7), C (4) |
| 4 | TFF | F | B (2), C (3) |
| 5 | FTT | F | A (1) |
| 6 | FTF | F | A (2) |
| 7 | FFT | F | A (3) |
| 8 | FFF | F | - |

Take a pair for each constituent :

- A : (1,5), or (2,6), or (3,7)
- B : (2,4)
- C : (3,4)

Two minimal sets to cover the modified condition criterion :

- (2,3,4,6) or (2,3,4,7)

That is 4 test cases (instead of 8 for all possible combinations)

# *Path Coverage*

- **Path Coverage Criterion:** Select a test T such that, by executing P for each d in T, all paths leading from the initial to the final node of P' s control flow graph are traversed
- In practice, however, the number of paths is too large, if not infinite
- Some paths are infeasible
- It is key to determine "critical paths"

# *Example*

```
if x ≠0 then
        y := 5;
else
        z := z – x;
end if;
if z > 1 then
        z := z / x;
else
        z := 0;
end if;
```

T1 = {<x=0, z =1>, <x =1, z=3>}

Executes all edges but do not show risk of division by 0

T2 = {<x=0, z =3>, <x =1, z=1>}

Would find the problem by exercising the remaining possible flows of control through the program fragment

T1 U T2 -> all paths covered

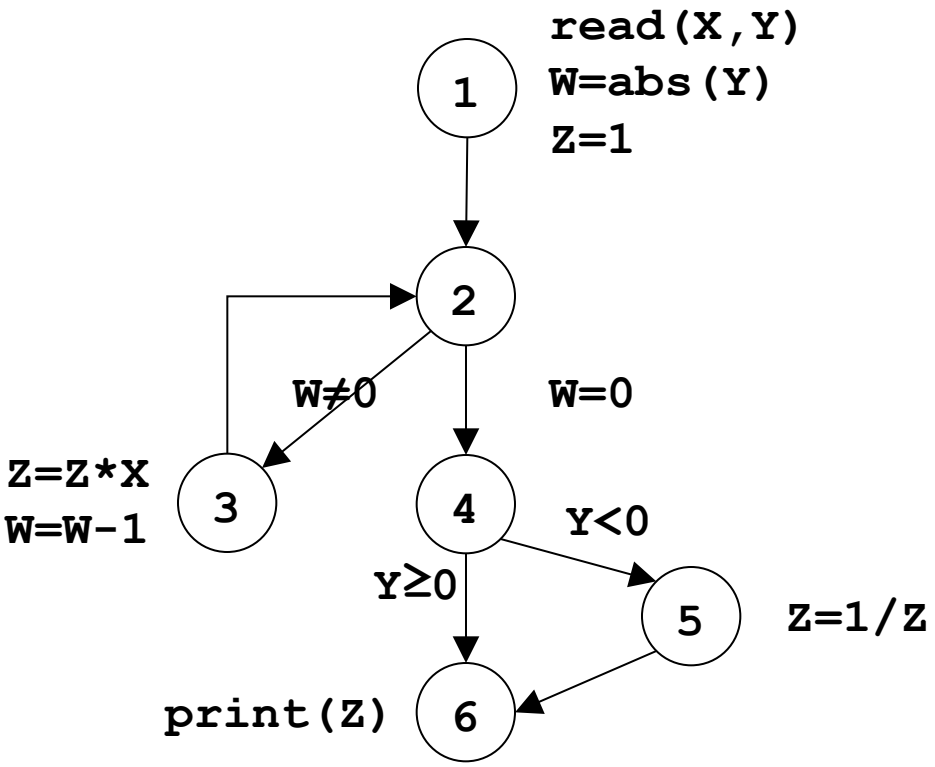
# *Dealing with Loops*

- Look for conditions that execute loops
  - Zero times
  - A maximum number of times
  - A average number of times (statistical criterion)
- For example, in the search algorithm
  - Skipping the loop (the table is empty)
  - Executing the loop once or twice and then finding the element
  - Searching the entire table without finding the desired element

# *Another Example: Power Function*

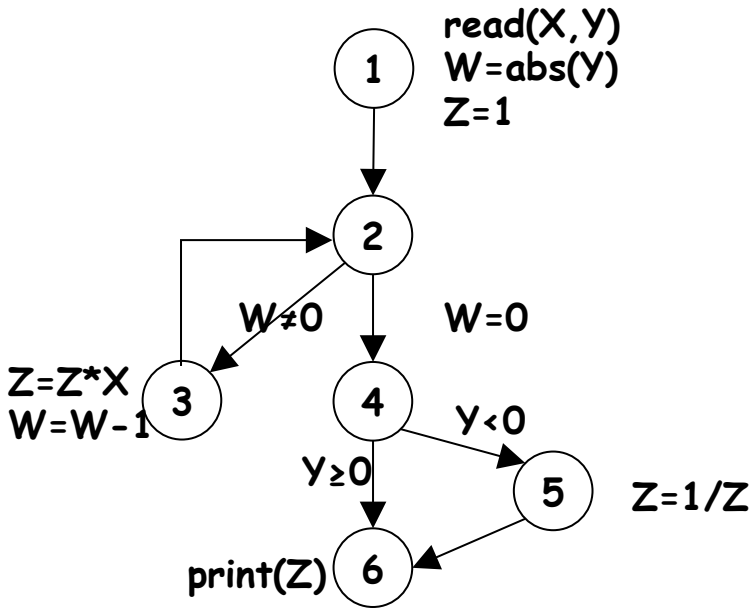
**Program computing Z=X^Y**

```
BEGIN
  read (X, Y) ;
  W = abs(Y) ;
  Z = 1 ;
  WHILE (W <> 0) DO
    Z = Z * X ;
    W = W - 1 ;
  END
  IF (Y < 0) THEN
    Z = 1 / Z ;
  END
  print (Z) ;
END
```



# *Example: Control-Flow Testing*

- All paths
  - Infeasible path
    - ✓ 1 ->2 -> 4 -> 5-> 6
  - Infinite number of paths :
    - ✓ As many ways to iterate
      2 -> (3 -> 2)* as values of Abs(Y)
- All branches
  - Two test cases are enough
    - ✓ Y<0 : 1 -> 2 -> (3 ->2)+ -> 4 -> 5 -> 6
    - ✓ Y≥0 : 1 -> 2 -> (3 -> 2)* -> 4 -> 6
- All statements
  - One test case is enough
    - ✓ Y<0 : 1 -> 2 -> (3 ->2)+ ->4 -> 5 -> 6



# *Deriving Input Values*

- Not all statements are usually reachable in real world programs
- It is not always possible to decide automatically if a statement is reachable and the percentage of reachable statements
- When one does not reach a 100% coverage, it is therefore difficult to determine the reason
- Tools are needed to support this activity – and there exist good tools
- Research focuses on search algorithms to help automate coverage
- Control flow testing is, in general, more applicable to testing in the small

# *Outline*

- Control flow coverage
  - Statement, Edge, Condition, Path coverage
- Data flow coverage
  - Definitions-Usages of data
- Analyzing coverage data
- Mutation Testing
- Integration testing
  - Strategies and criteria
- Conclusions
  - Generating test data, tools, Marick's Recommendations

# *Data Flow Analysis*

- CFG paths that are significant for the data flow in the program
- Focuses on the assignment of values to variables and their uses
- Analyze occurrences of variables
- Definition occurrence: value is bound to variable
- Use occurrence: value of variable is referred
- Predicate use: variable used to decide whether predicate is true
- Computational use: compute a value for defining other variables or output value

# *FACTORIAL Example*

```
1. public int factorial(int n){
2.     int i, result = 1;
3.     for (i=2; i<=n; i++) {
4.         result = result * i;
5.         }
6.     return result;
7. }
```

| Variable | Definition line | Use line |
|---|---|---|
| n | 1 | 3 |
| result | 2 | 4 |
| result | 2 | 6 |
| result | 4 | 4 |
| result | 4 | 6 |

# *Basic Definitions*

- Node n in CFG(P) is a **defining node** of the variable v in V, written as DEF(v,n), iff the value of the variable v is defined in the statement corresponding to node n

- Node n in CFG(P) is a **usage node** of the variable v in V, written as USE(v,n), iff the value of the variable v is used in the statement corresponding to node n

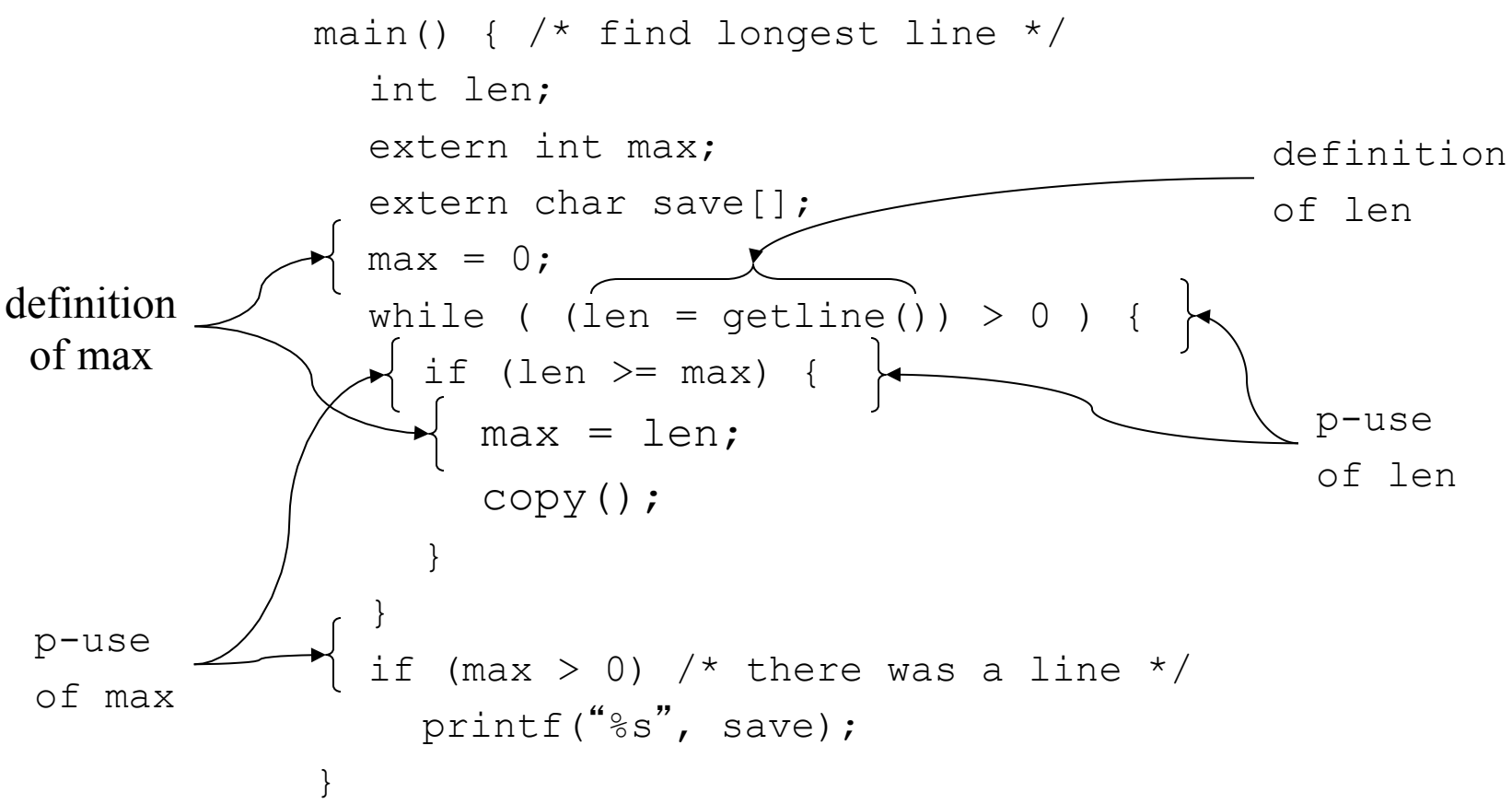
- A usage node USE(v,n) is a **predicate use** (denoted as P-Use) iff the statement n is a predicate statement, otherwise USE(v,n) is a **computation use** (denoted as C-use)

# *Basic Definitions II*

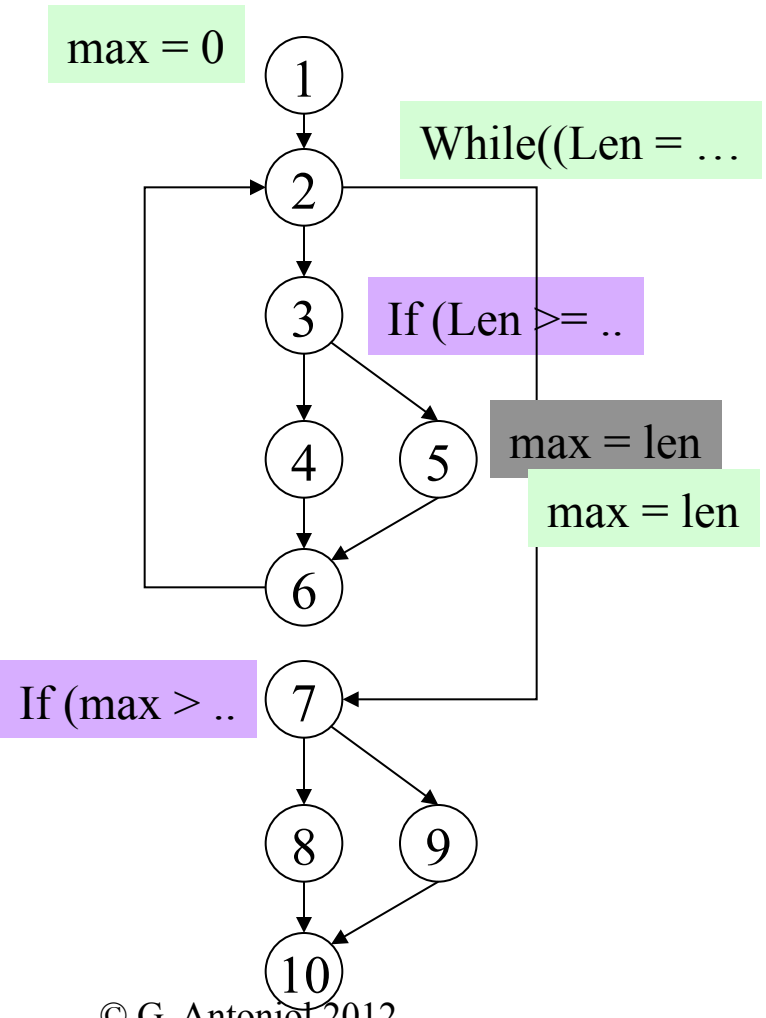
- A **definition-use (sub)path** with respect to a variable v (denoted du-path) is a (sub)path in PATHS(P) such that, for some v in V, there are define and usage nodes DEF(v, m) and USE(v, n) such that m and n are initial and final nodes of the (sub)path.

- A **definition-clear (sub)path** with respect to a variable v (denoted dc-path) is a definition-use path in PATH(P) with initial and final nodes DEF(v, m) and USE(v, n) such that no other node in the path is a defining node of v.

# *Simple Example*



# *Simple Example (CFG)*




| | |
|---|---|
| v = … | Definition DEF(v, n) |
| v >= … | P-use USE(v,n) |
| … = …v … | C-use USE(v,n) |

- DEF(max, 1)
- DEF(len, 2)
- DEF(max, 5)
- C-USE(len, 5)
- P-USE(len, 2)
- P-USE(len, 3)
- P-USE(max, 3)
- P-USE(max, 7)

# *Criteria Formal Definitions I*

- The set T satisfies the **all-Definitions** criterion for the program P iff for every variable v in V, T contains definition clear paths from every defining node of v to a use of v.

- The set T satisfies the **all-Uses** criterion for the program P iff for every variable v in V, T contains at least one definition clear path from every defining node of v to every reachable use of v.

- The set T satisfies the **all-P-Uses/Some C-Uses** criterion for the program P iff for every variable v in V, T contains at least one definition clear path from every defining node of v to every predicate use of v, and if a definition of v has no P-Uses, there is a definition-clear path to at least one computation use.

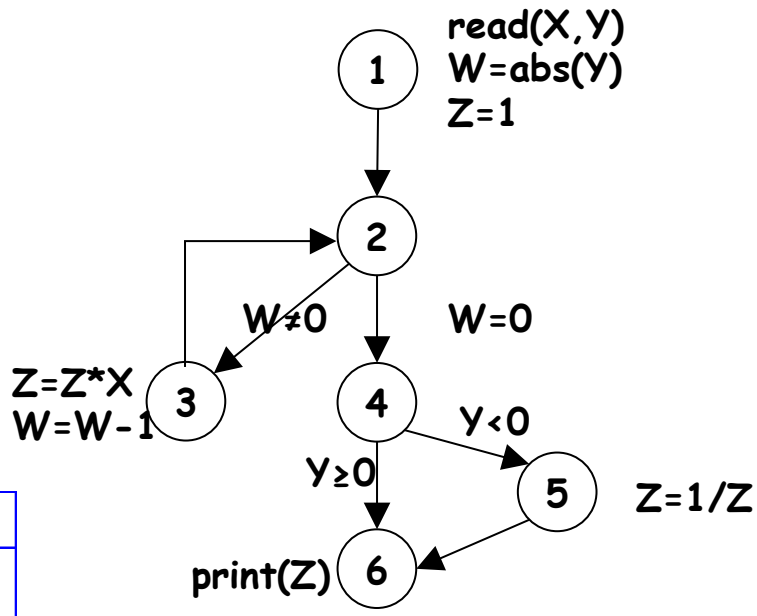
# *Criteria Formal Definitions II*

- The set T satisfies the **all-C-Uses/Some P-Uses** criterion for the program P iff for every variable v in V, T contains at least one definition-clear path from every defining node of v to every computation use of v, and if a definition of v has no C-Uses, there is a definition-clear path to at least one predicate use.

- The set T satisfies the **all-DU-Paths** criterion for the program P iff for every variable v in V, T contains all definition-clear paths from every defining node of v to every reachable use of v, and that these paths are either single loops traversals, or they are cycle free.

# POWER Example

| node i | def(i) | c-use(i) | edge(i,j) | p-use(i,j) |
|---|---|---|---|---|
| 1 | X, Y, W, Z | | (1,2) | |
| 2 | | | (2,3)<br>(2,4) | W<br>W |
| 3 | W, Z | X, W, Z | (3,2) | |
| 4 | | | (4,5)<br>(4,6) | Y<br>Y |
| 5 | Z | Z | (5,6) | |
| 6 | | Z | | |

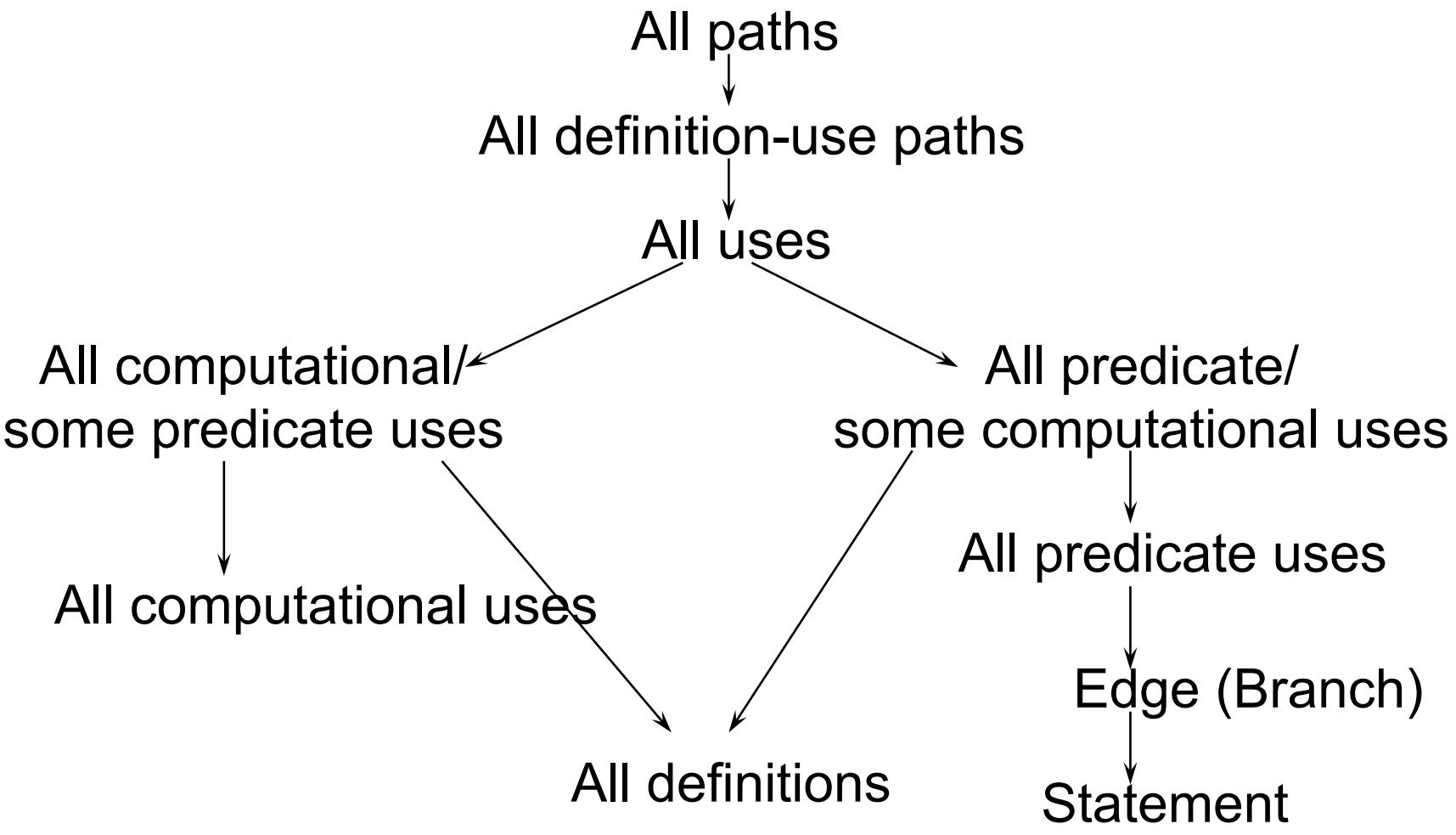



| node i | dcu(v,i) | dpu(v,i) |
|---|---|---|
| 1 | dcu(X,1) = {3}<br>dcu(Z,1) = {3,6}<br>dcu(W,1) = {3} | dpu(Y,1) = {(4,5),(4,6)}<br>dpu(W,1) = {(2,3),(2,4)} |
| 3 | dcu(W,3) = {3}<br>dcu(Z,3) = {3,5,6} | dpu(W,3) = {(2,3),(2,4)} |
| 5 | dcu(Z,5) = {6} | |

# *Discussion*

- Generates test data according to the way data is manipulated in the program
- Help define intermediary criteria between all-edges testing (possibly too weak) and all-paths testing (often impossible)
- But needs effective tool support (see last slide)

# *Subsunction rules*



# *Measuring Code Coverage*

- One advantage of structural criteria is that their coverage can be measured *automatically*
- To control testing progress
- To assess testing completeness in terms of remaining faults and reliability
- To help fix targets for testers
- High coverage is not a guarantee of fault-free software, just an element of information to increase our confidence -> statistical models

# *Outline*

- Control flow coverage
  - Statement, Edge, Condition, Path coverage
- Data flow coverage
  - Definitions-Usages of data
- Analyzing coverage data
- Mutation Testing
- Integration testing
  - Strategies and criteria
- Conclusions
  - Generating test data, tools, Marick's Recommendations

# *Test Productivity*

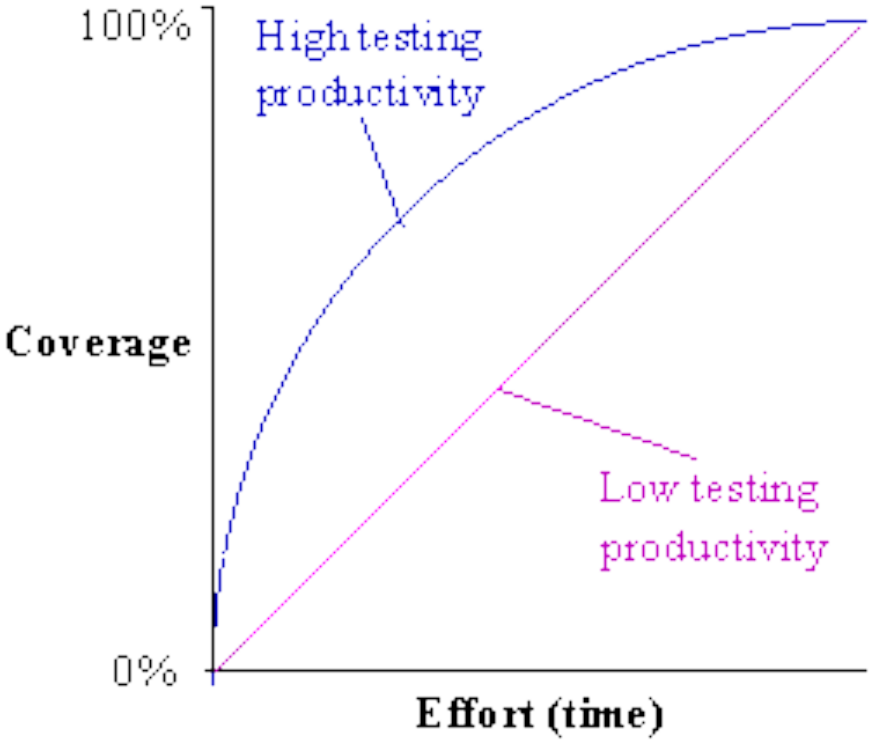



Figure 1: Coverage rate

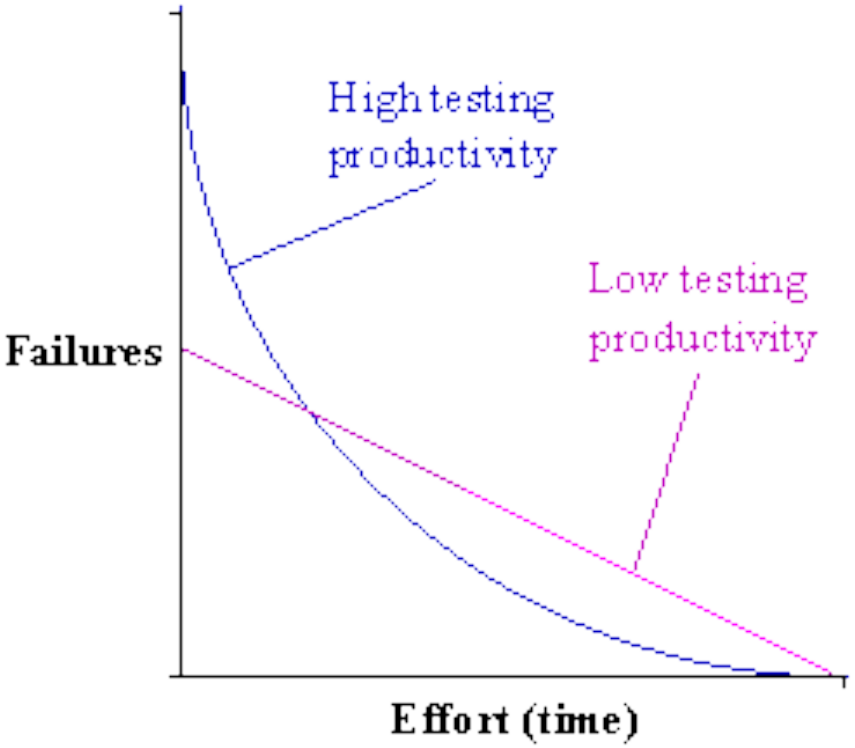
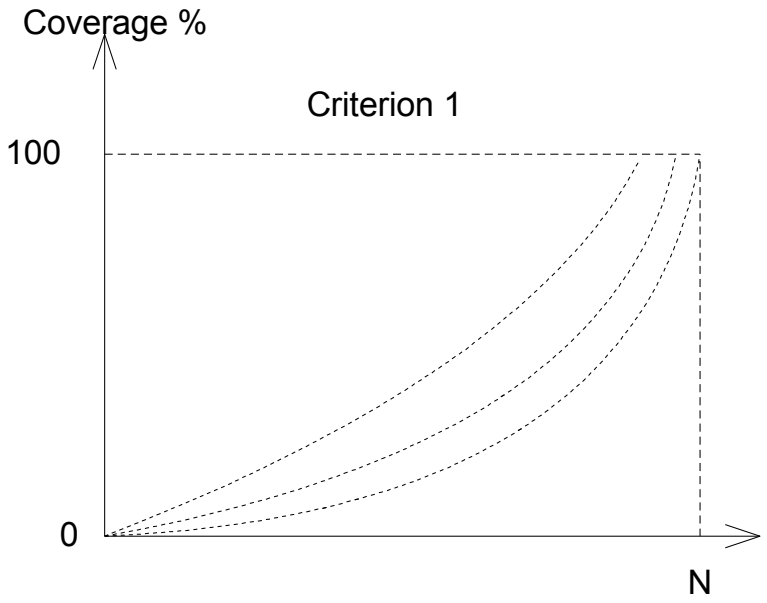
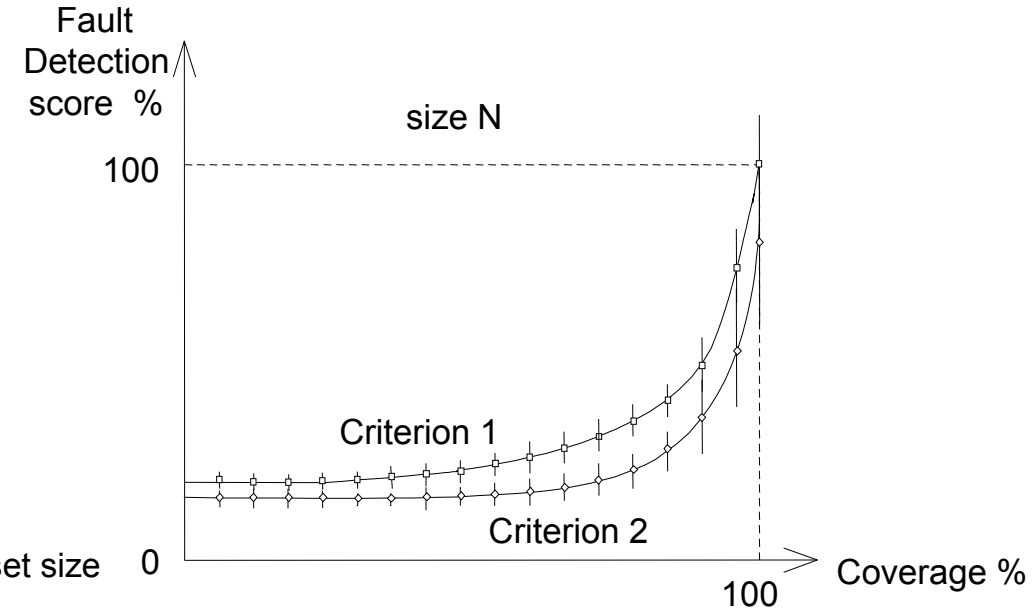



Figure 2: Failure discovery rate

# *Typical Analyses*



# *Hutchins et al Experiment*

- The All-Edges and All-DU coverage criteria were applied to 130 faulty program versions derived from 7 C programs (141-512 LOC) by seeding realistic faults
- The 130 faults were created by 10 different people, mostly without knowledge of each other's work; their goal was to be as realistic as possible.
- The test generation procedure was designed to produce a wide range both of test size and test coverage percentages, i.e., at least 30 test sets for each 2% coverage interval for each program
- They examined the relationship between fault detection and test set coverage / size

# One Program Example

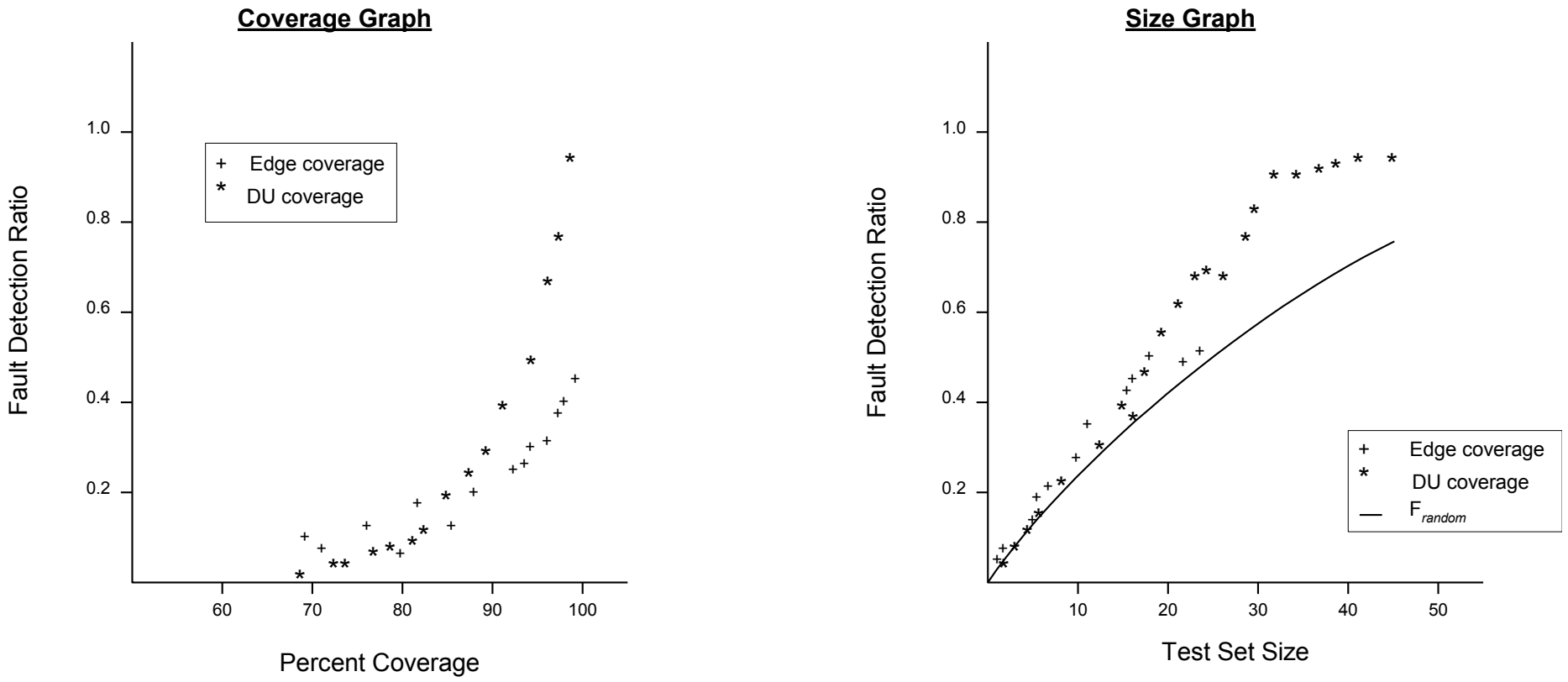



Figure: Fault Detection Ratios for One Faulty Program

# *Results*

- Both coverage criteria performed better than random test selection – especially DU-coverage
- Significant improvements occurred as coverage increased from 90% to 100%
- 100% coverage alone is not a reliable indicator of the effectiveness of a test set – especially edge coverage
- Wide variation in test effectiveness for a given coverage
- As expected, on average, achieving all-DU coverage required significantly larger test sets all-Edge coverage

# *Outline*

- Control flow coverage
  - Statement, Edge, Condition, Path coverage
- Data flow coverage
  - Definitions-Usages of data
- Analyzing coverage data
- Mutation Testing
- Integration testing
  - Strategies and criteria
- Conclusions
  - Generating test data, tools, Marick's Recommendations

# *Mutation Testing: Definitions*

- *Fault-based Testing*: directed towards "typical" faults that could occur in a program
- Syntactic variations are applied to in a systematic way to the program to create faulty versions that exhibit different behavior (Mutant)
- E.g., x = a + b into x = a - b
- Mutation testing helps a user create effective test data in an interactive manner
- The goal is to cause each faulty version (mutant) to exhibit a different behavior

# *Different Mutants*

- Stillborn mutants: Syntactically incorrect, killed by compiler
- Trivial mutants: Killed by almost any test case
- Equivalent mutant: Always produces the same output as the original program
- None of the above are interesting from a mutation testing perspective

# *Basic Idea*

- Take a program and test data generated for that program (using another test technique)
- Create a number of *similar* programs (mutants), each differing from the original in one small way, i.e., each possessing a fault
- E.g., replace addition operator by multiplication operator
- The original test data are then run through the *mutants*
- If test data detect differences in mutants, then the mutants are said to be *dead,* and the test set is *adequate*

# *Basic Idea II*

- A mutant remains *live* either because it is equivalent to the original program (functionally identical though syntactically different – *equivalent mutant*) or the test set is inadequate to kill the mutant
- In the latter case, the test data need to be re-examined, possibly augmented to kill the *live* mutant
- For the automated generation of mutants, we use *mutation operators* , that is predefined program modification rules (I.e., corresponding to a fault model)

# *Example of Mutation Operators*

- Constant replacement
- Scalar variable replacement
- Scalar variable for constant replacement
- Constant for scalar variable replacement
- Array reference for constant replacement
- Array reference for scalar variable replacement
- Constant for array reference replacement
- Scalar variable for array reference replacement
- Array reference for array reference replacement
- Source constant replacement
- Data statement alteration
- Comparable array name replacement
- Arithmetic operator replacement
- Relational operator replacement
- Logical connector replacement
- Absolute value insertion
- Unary operator insertion
- Statement analysis
- Statement deletion
- Return statement replacement

# *Example of Mutation Operators II*

Specific to object-oriented programming languages:

- Replacing a type with a compatible subtype (inheritance)
- Changing the access modifier of an attribute, a method
- Changing the instance creation expression (inheritance)
- Changing the order of parameters in the definition of a method
- Changing the order of parameters in a call
- Removing an overloading method
- Reducing the number of parameters
- Removing an overriding method
- Removing a hiding Field
- Adding a hiding field

# *Specifying Mutations Operators*

- Ideally, we would like the mutation operators to be representative of (and generate) all realistic types of faults that could occur in practice.
- Mutation operators change with programming languages, design and specification paradigms, though there is much overlap.
- In general, the number of mutation operators is large as they are supposed to capture all possible syntactic variations in a program.
- Some recent studies seem to suggest that mutants are good indicators of test effectiveness (Andrews et al 2004).

# *Mutation Coverage*

- Complete coverage equate to killing all non-equivalent mutants
- The amount of coverage is called “mutation score”
- We can see each mutant as a test requirement
- The number of mutants depends on the definition of mutation operators and the syntax/structure of the software
- Numbers of mutants tend to be large, even for small programs (random sampling?)

# *Simple Example*

**Original Function**

```
int Min (int A, int B)
   int minVal;
{
   minVal = A;
   if (B < A)
   {
       minVal = B;
   }
return (minVal);
} // end Min
```

**With Embedded Mutants**

```
int Min (int A, int B)
   int minVal;
{
   minVal = A;
Δ1 minVal = B;
   if (B < A)
Δ2 if (B > A)
Δ3 if (B < minVal)
   {
       minVal = B;
Δ4     Bomb();
Δ5     minVal = A;
Δ6     minVal = failOnZero (B);
   }
return (minVal);
} // end Min
```

# *Discussion of Example*

- Mutant 3 is equivalent as, at this point, `minVal` and `A` have the same value

- Mutant 1: In order to find an appropriate test case, we must
  - Reach the fault seeded during execution (Reachability)
    - ✓ Always true
  - Cause the program state to be incorrect (Infection)
    - ✓ A <> B
  - Cause the program output to be incorrect (Propagation)
    - ✓ (B<A) = false

# *Assumptions*

- What about more complex errors, involving several statements?
- *Competent programmer assumption*: They write programs that are nearly correct
- *Coupling effect assumption*: Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors
- There is some empirical evidence of these hypotheses

# *Simple Example*

**Specification:**

The program prompts the user for a positive integer in the range 1 to 20 and then for a string of that length. The program then prompts for a character and returns the position in the string at which the character was first found or a message indicating that the character was not present in the string. The user has the option to search for more characters.

# *Code Chunk*

```
…
found := FALSE;
i := 1;
while(not(found)) and (i <= x) do
begin
   if a[i] = c then
       found := TRUE
   else
       i := i + 1
end
…
```

# *Test Cases 1*

| Input | | | | Expected Output |
|---|---|---|---|---|
| x | a | c | Response | |
| 25 | | | | Input Integer between 1 and 20 |
| 1 | x | x | | Character x appears at position 1 |
| | | | y | |
| | | a | | Character does not occur in string |
| | | | n | |

# *Mutant 1*

- Replace `Found := FALSE;` with `Found := TRUE;`
- Re-run original test data set
- Only one small change at a time to avoid the danger of introduced faults with interfering effects
- Failure: “character a appears at position 1” instead of saying it does not occur in the string
- Mutant 1 is killed

# *Mutant 2*

- Replace `i:=1;` with `x:=1;`
- Running our original test data set fails to reveal the fault
- As a result of the bug, only position 1 in string will be searched for
- We need to increase our string length and search for a character further along it
- We modify the test data so as
  - To preserve the effect of earlier tests
  - To make sure the live mutant is killed

# *Test Cases 2*

| Input | | | | Expected Output |
|---|---|---|---|---|
| x | a | c | Response | |
| 25 | | | | Input Integer between 1 and 20 |
| 3 | xCv | x | | Character x appears at position 1 |
| | | | y | |
| | | a | | Character does not occur in string |
| | | | y | |
| | | v | | Character v appears at position 3 |
| | | | n | |

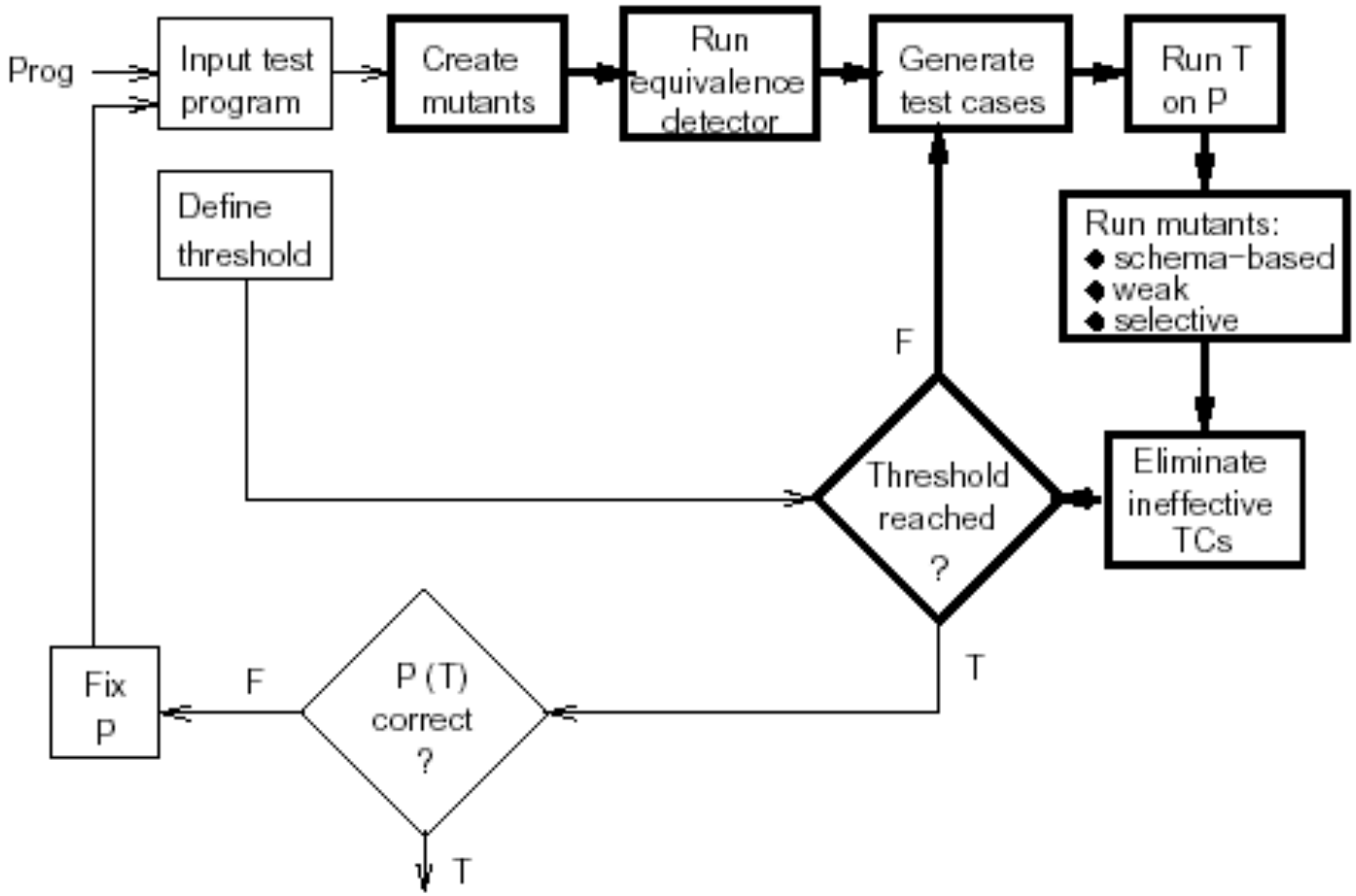
# *Mutant 3*

- `i := i + 1;` is replaced with `i:= i +2;`
- Again, our test data fails to kill the mutant
- We must augment the test data to search for a character in the middle of the string
- With the new test data, mutant 3 is dead
- Many other changes could be made on this short piece of code, e.g., changing array reference, changing relational operator

# *Test Case 3*

| Input | | | | Expected Output |
|---|---|---|---|---|
| x | a | c | Response | |
| 25 | | | | Input Integer between 1 and 20 |
| 1 | xCv | x | | Character x appears at position 1 |
| | | | y | |
| | | a | | Character does not occur in string |
| | | | y | |
| | | v | | Character v appears at position 3 |
| | | | y | |
| | | C | | Character C appears at position 2 |
| | | | n | |

# *Mutation Testing Process*



# *Discussion*

- It measures the quality of test cases
- It provides the tester with a clear target (mutants to kill)
- Mutation testing shows certain kinds of faults (specified by the fault model) are unlikely
- It does force the programmer to scrutinize the code and think of the test data that will expose certain kinds of faults
- Computationally intensive, a possibly very large number of mutants is generated: random sampling, selective mutation operators (Offut)
- Equivalent mutants are a practical problem: It is in general an undecidable problem
- Probably most useful at unit testing level
- Some systems have been designed to help but still time consuming

# *Other Applications*

- Mutation operators and systems are also very useful for assessing the effectiveness of test strategies – they have been used in a number of case studies
  - Define a set of realistic mutation operators
  - Generate mutants (automatically)
  - Generate test cases according to alternative strategies
  - Assess the *mutation score* (percentage of mutants killed)

# *Other Applications II*

- We have seen mutation operators on code
- But there is work on
  - Mutation operators for module interfaces (aimed at integration testing)
  - Mutation operators on specifications: Petri nets, state machines, … (aimed at system testing)

# *Outline*

- Control flow coverage
  - Statement, Edge, Condition, Path coverage
- Data flow coverage
  - Definitions-Usages of data
- Analyzing coverage data
- Mutation Testing
- Integration testing
  - Strategies
  - Criteria
- Conclusions
  - Generating test data, tools, Marick's Recommendations

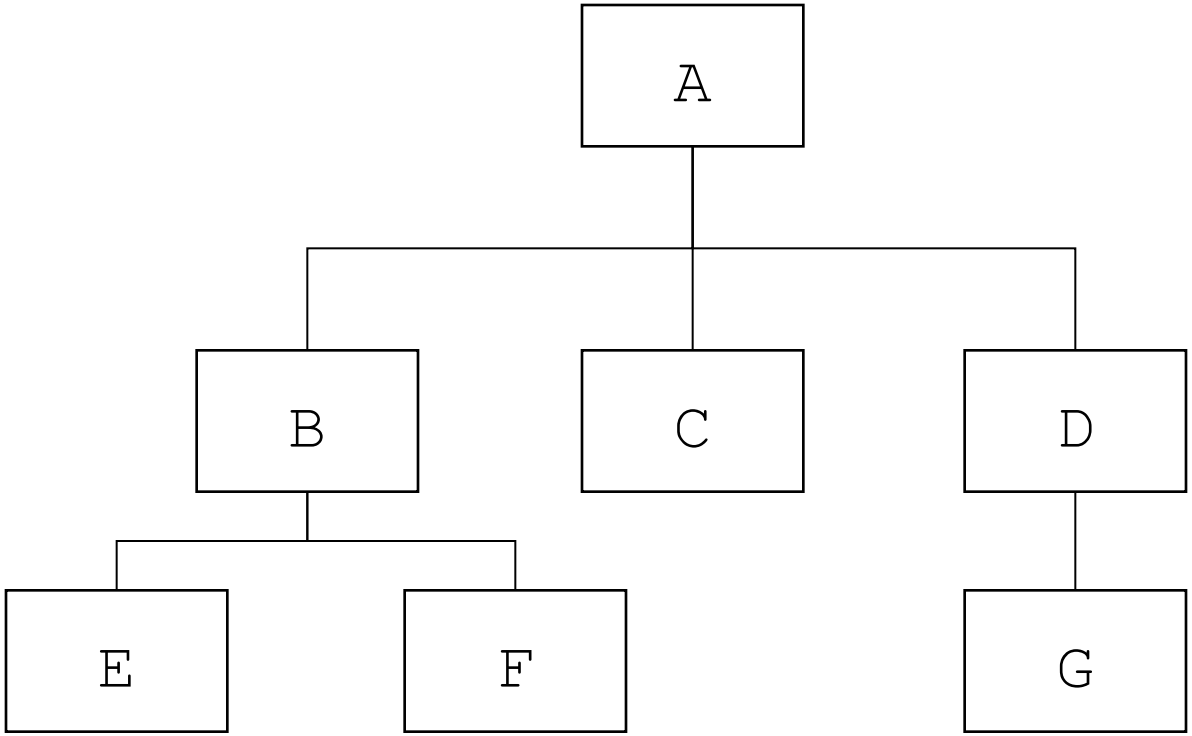
# *Preliminary Remarks I*

- When components have been tested to a satisfactory level, we combine them into working systems
- Components are still likely to be faulty as test stubs and drivers used during unit testing are only approximations of the components they simulate
- In addition, possible interface faults
  - e.g., assumptions about parameter semantics
- Order of integration? Drivers and stubs are expensive and the order affects the cost of testing

# *Preliminary Remarks II*

- Integration is planned so that when a failure occurs, we have some idea of what caused it
- Development is not sequential but activities are overlapping: some components may be in coding, unit testing, and integration testing
- Integration strategy affects the order of coding, unit testing, and the cost and thoroughness of testing

# *Generic Example*




Example of a hierarchy of modules, defined by usage dependencies between modules

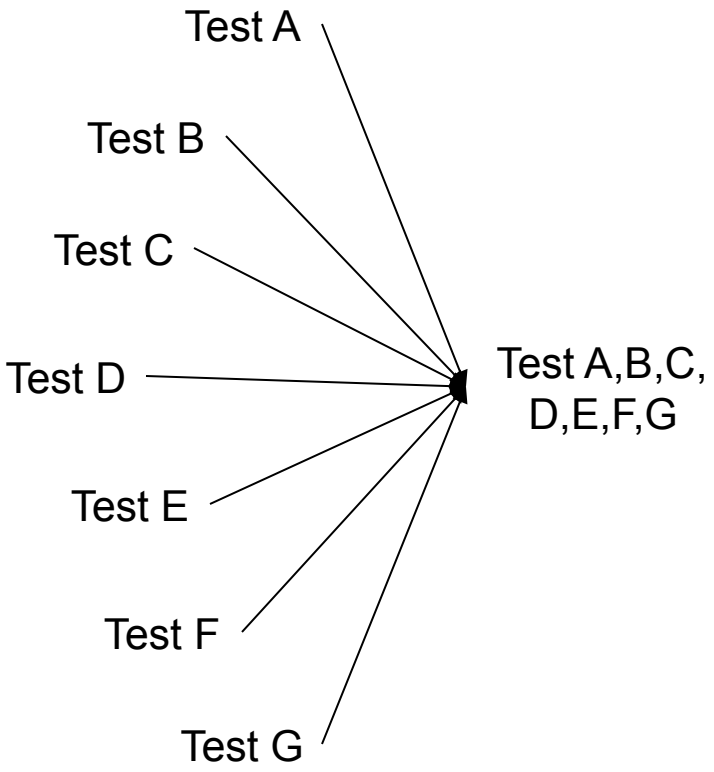
# *Big Bang Integration*

- All components are brought together at once into a system and tested as entire system

☹ Difficult to find the root cause of any failure

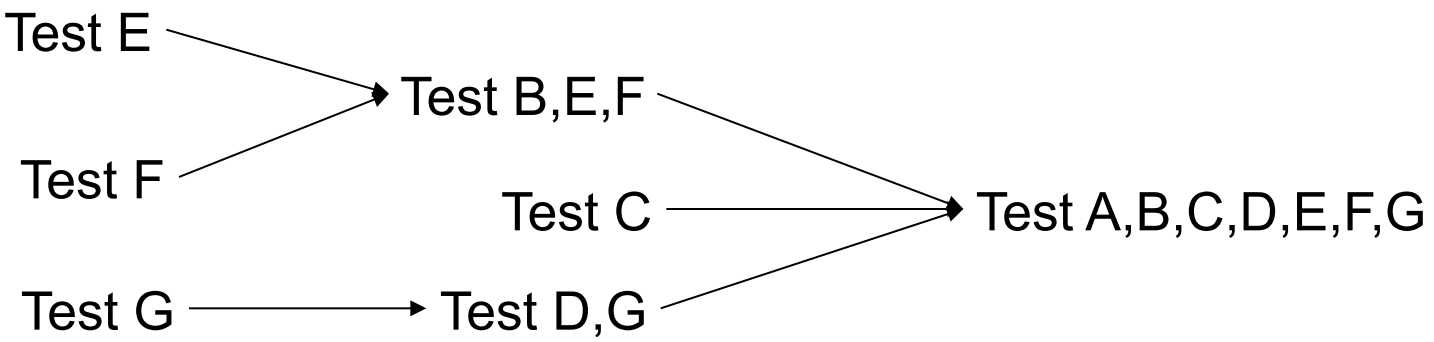
☹ Wait for all components to be ready

=> Not recommended



# *Bottom-Up Integration I*

- Context: Design is based on functional decomposition
- Each component at the lowest of the hierarchy are tested first (e.g., E, F, and G)
- If a problem happens when testing B (with E and F), the heuristic says that its cause is either in B, or the interface between between B and E or F.

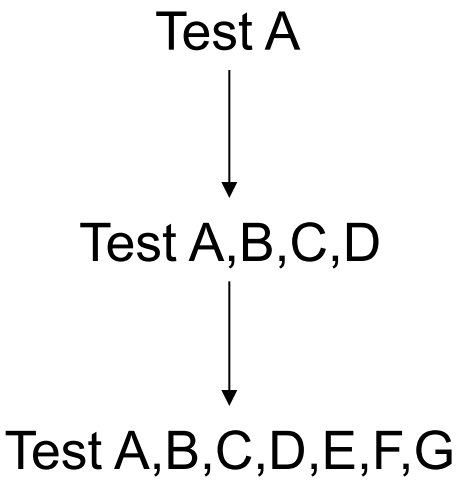


# *Bottom-Up Integration II*

- ☺ Need to develop component *drivers* but no *stubs*
  - In our example, we need 3 drivers for E&F, G, and B&C&D.
- ☺ Strategy helps us isolate faults more easily, in particular interface faults
- ☺ Convenient when many general purpose utility routines, invoked often by others, at the lowest levels.
- ☹ Problem: top level components may be more important (major system activities) but last to be tested, e.g., user interface components
- ☹ Faults in top level sometimes reflect faults in design, e.g., subsystem decomposition – have to be corrected early

# *Top-Down Integration I*

- Reverse of Bottom-up
- When components that are not yet tested are needed
  - we use “stubs” which simulate the activity of missing components

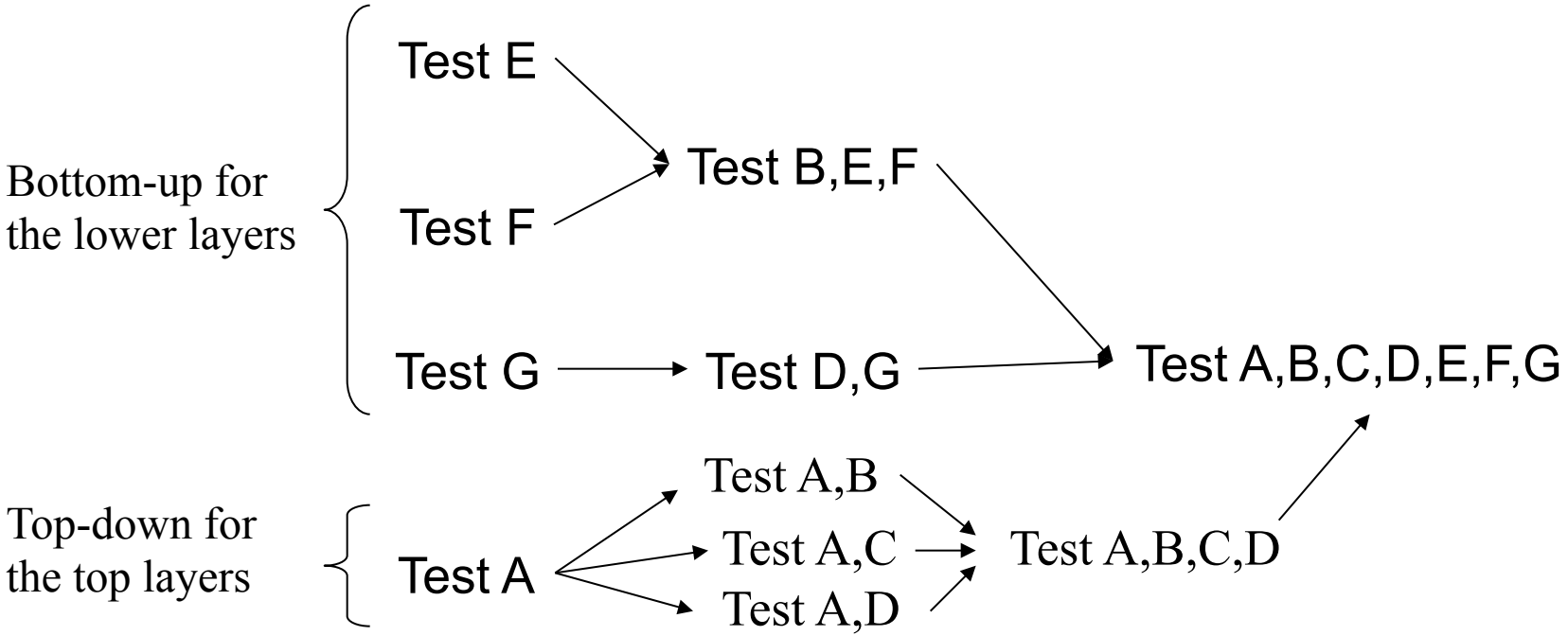


# *Top-Down Integration II*

- ☺ "Controlling" components are tested first, e.g., user interface
- ☺ Early working system
- ☺ Development of lower level components deferred
- ☺ Reusable test cases as integration goes on
- ☹ Driver programs not needed here, but stubs are time consuming and error prone
- ☹ Writing stubs may turn out to be complex and their correctness may affect the validity of the test
- ☹ Inadequate if low-level components specifications are unstable

# *Sandwich Integration I*

1. Choose a target layer (the middle one in our example)
2. Top down approach used in top layers
3. Bottom-up approach in lower layers



# *Sandwich Integration II*

- ☺ Allows to test general-purpose utilities' correctness at the beginning
- ☺ No need of stubs for utilities
- ☺ Allows integration testing of top components to begin early: Tests "control" component early
- ☺ No need for test drivers of top layer
- ☹ But may not test thoroughly the individual components of target layer (e.g., C in the example is not unit tested)

# *Modified Sandwich Testing I*

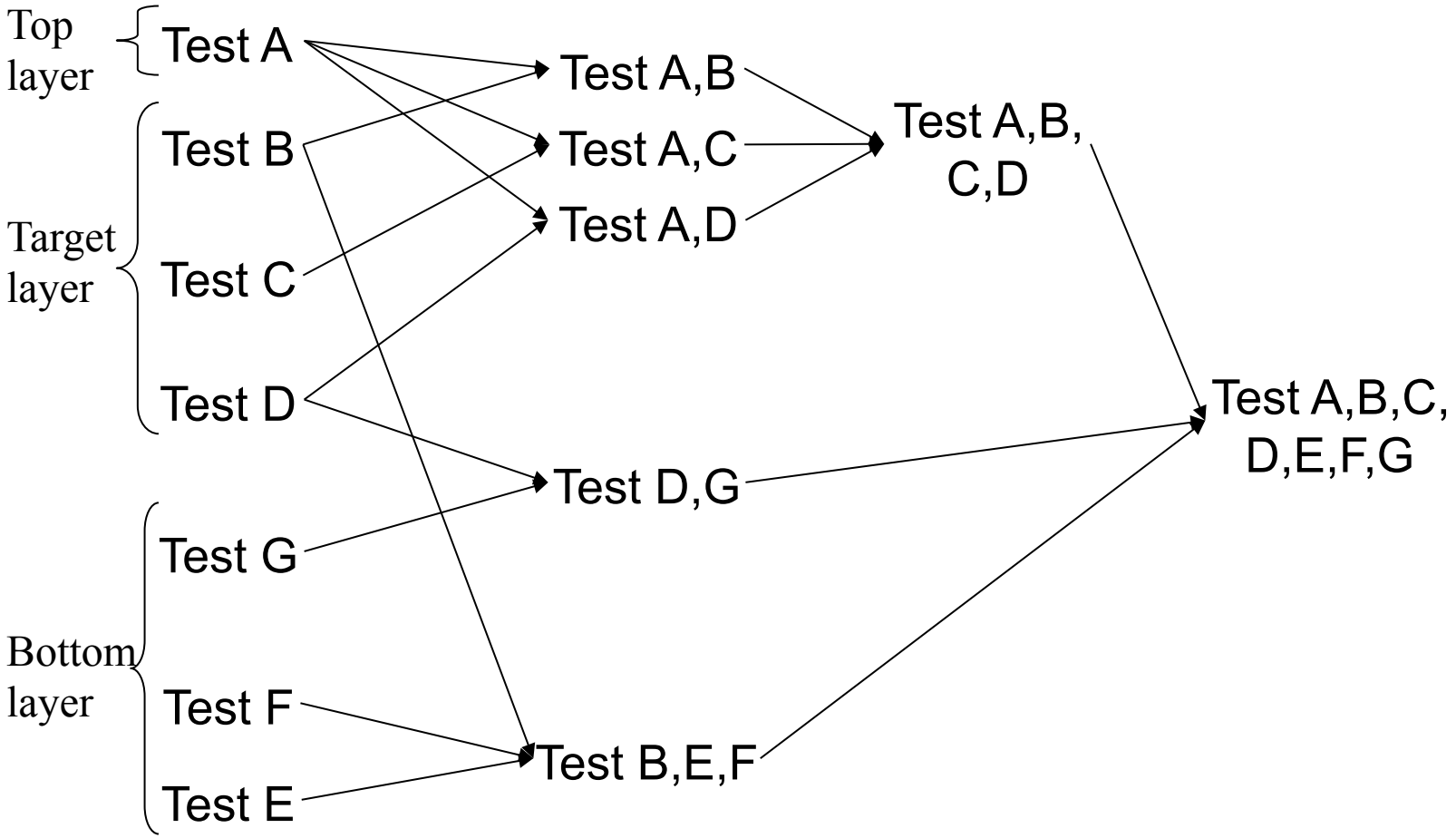
1. Individual layer tests
   - A top layer test with stubs for the target layer
   - A target layer test with drivers and stubs replacing the top and bottom layers
   - A bottom layer test with a driver for the target layer

2. Combined layer tests
   - The top layer accesses the target layer
   - The bottom layer is accessed by the target layer

- Many testing activities can be performed in parallel – shorter testing time

- But additional test stubs and drivers

# *Modified Sandwich Testing II*



# *Criteria for Comparison*

- Stubs, drivers
- Time to "working" system:
  - System you can demonstrate (e.g., to future users)
- Integration start: we want it as early as possible
- Ability to test paths:
  - Ensure path coverage of components.
  - Top-down is hard because it is difficult to control the inputs of lower level components through higher level ones. Sandwich strategy alleviates a bit the problem as the top-down hierarchy is more shallow.
- Ability to plan and control:
  - Top-down is hard because of the inherent instability of lower level components, which may invalidate the integration testing of higher components if their spec changes. Recall they may not exist yet when integration testing of higher level components starts. Sandwich inherits the "hard" score for the same reason.

# *Integration Strategies - Overview*

| | Bottom-up | Top-down | Big-Bang | Sandwich |
|---|---|---|---|---|
| Integration Start | Early | Early | Late | Early |
| Time to working system | Late | Early | Late | Early |
| Drivers | Yes | No | No | Yes |
| Stubs | No | Yes | No | Yes |
| Ability to test paths | Easy | Hard | Easy | Medium |
| Ability to plan and control | Easy | Hard | Easy | Hard |

# *Outline*

- Control flow coverage
  - Statement, Edge, Condition, Path coverage
- Data flow coverage
  - Definitions-Usages of data
- Analyzing coverage data
- Mutation Testing
- Integration testing
  - Strategies
  - Criteria
- Conclusions
  - Generating test data, tools, Marick's Recommendations

# *Coupling-based Criteria*

- Refer to the testing of interfaces between units / modules to assure they have consistent assumptions and communicate correctly.
  - Coupling between two units measures the dependency relations between two units by reflecting the interconnections between units; faults in one unit may affect the coupled unit (Yourdon and Constantine, 1979)
  - Coupling-based Coverage Criteria are proposed by Jin and Offutt (98), specifically aimed at integration testing in a non-OO context, based on data flow analysis

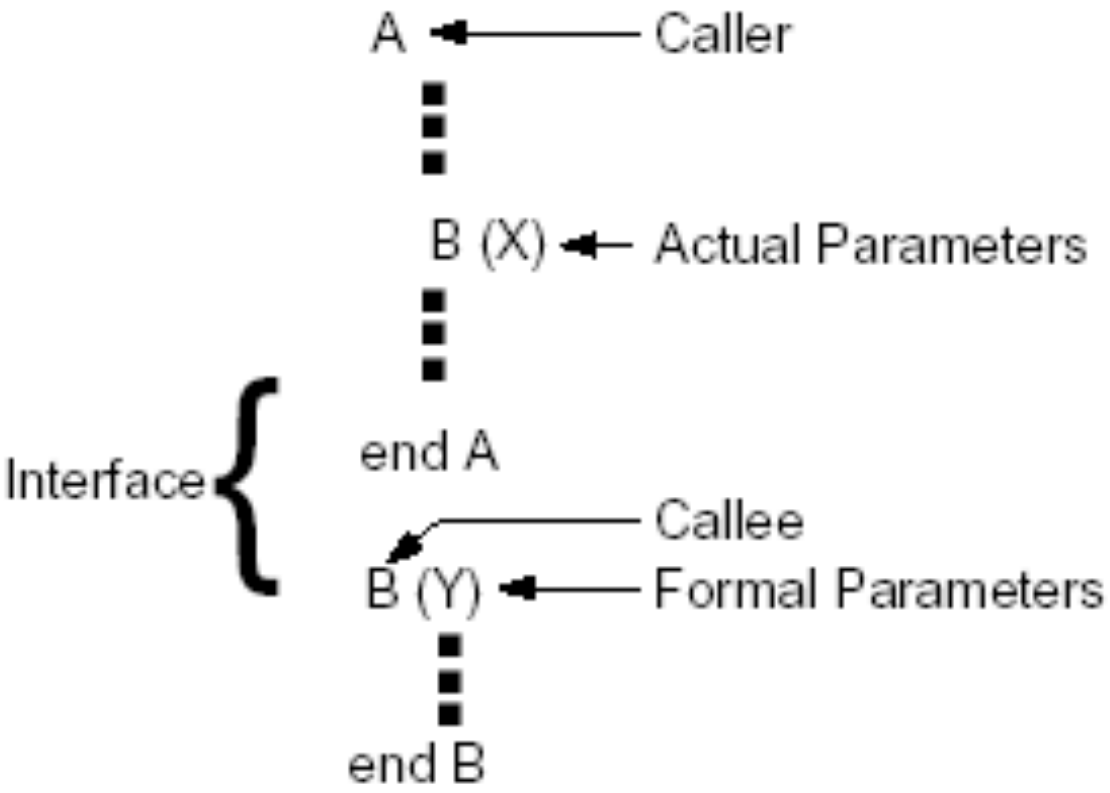
# *General Principles*

- Each module to be integrated should have passed an isolated (unit) test
- Integration testing must be performed at a higher level of abstraction – looking at program as atomic building blocks and focusing on their interconnections
- Goal: *Guide* testers during integration testing, help define a criterion to determine when to *stop* integration testing

# *Basic Definitions*

- The *interface* between two units is the mapping of *actual* to *formal* parameters
- Data flow connections between units are often complex: rich source of faults
- During integration testing, we want to look at *definitions* and *uses* across different units
- To increase our confidence in interfaces, we want to ensure that variables defined in *caller* units are appropriately used in *callee* units
- Look at variables definitions *before* calls and returns to other units, and uses of variables just *after* calls and returns from the called unit
- We refer to *coupling variables* for variables that are defined in one unit and used in another.

# *Short Example*

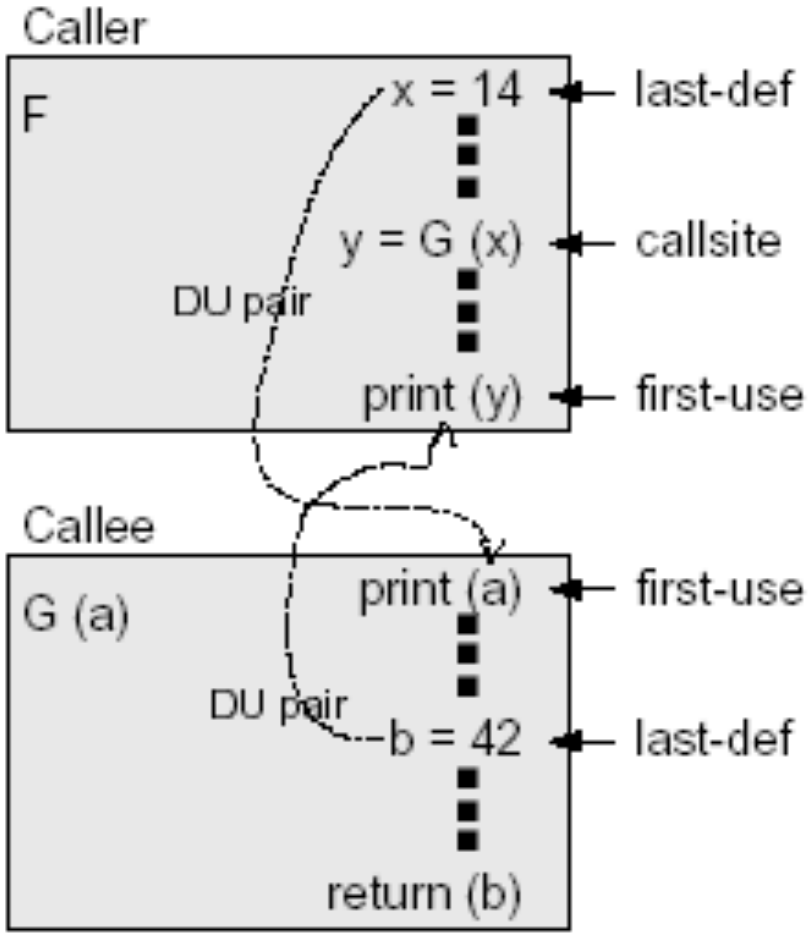



Ammann & Offutt

# *Basic Definitions*

- We differentiate three types of coupling between units:
  - Parameter coupling
  - Shared data coupling (i.e., non local variables shared by several modules)
  - External device coupling (i.e., references of several modules to the same external device)
- Concepts here apply equally to all coupling types, though discussions will be in terms of parameters
- *Call sites*: statements in caller (A) where callee (B) is invoked
- *Last-Defs*: The set of nodes (CFG) that define x for which there is a def-clear path from the node through the callsite / return to a use in the other unit.
- *First-Uses*: The set of nodes (CFG) that have uses of y and for which there exists a def-clear and use-clear path between the callsite (if the use is in the caller) or the entry point (if the use is in the callee) and the nodes.

# *Short Example*



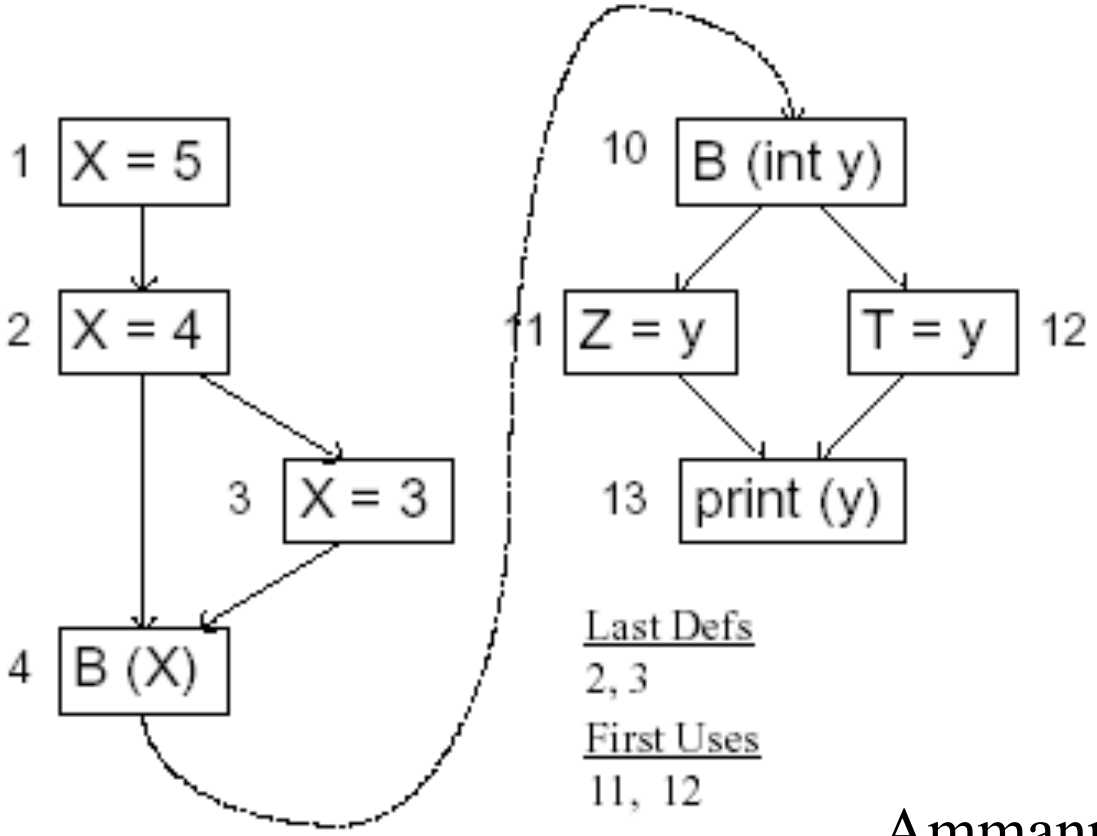

Ammann & Offutt

# *Coupling Paths & Criteria*

- A coupling du-path (or coupling path) is a path from a last-def to a first-use

- List of criteria (apply to 3 types of coupling ):
  - call-coupling:
    - ✓ requires execution of all call sites in the caller.
  - all-coupling-defs:
    - ✓ requires that for each coupling definition at least one coupling path to **at least one** reachable coupling use is executed.
  - all-coupling-uses:
    - ✓ requires that for each coupling definition at least one coupling path to **each** reachable coupling use is executed.
  - all-coupling-paths:
    - ✓ requires that **all** loop-free coupling paths be executed.

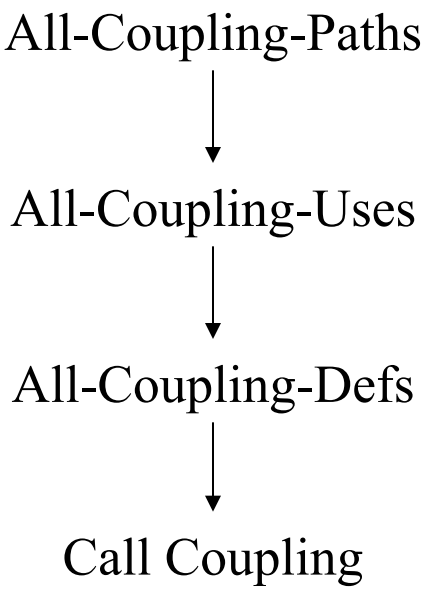
# *Example with two CFG's*




Ammann & Offutt

# *Parameter Coupling*

- Subsumption hierarchy:



# *Larger Example*

```
1.   procedure QUADRATIC is
2.   …
3.   begin
4.     GET (Control_Flag);
5.     if (Control_Flag = 1) then
6.       GET(X); --- last-def-before-call (X)
7.       GET(Y); --- last-def-before-call (Y)
8.       GET(Z); --- last-def-before-call (Z)
9.     else
10.      X := 0; --- last-def-before-call (X)
11.      Y := 0; --- last-def-before-call (Y)
12.      Z := 0; --- last-def-before-call (Z)
13.    end if;
14.    OK := TRUE; --- last-def-before-call (OK)
15.    ROOT(X,Y,Z,R1,R2,OK); --- call-site
16.    if OK then --- first-use-after-call (OK)
17.      PUT(R1); --- first-use-after-call (R1)
18.      PUT(R2); --- first-use-after-call (R2)
19.    else
20.      PUT("No solution");
21.    end if;
22.  end QUADRATIC;
```

```
1.   procedure ROOT(A,B,C: in FLOAT;
2.                   ROOT1,ROOT2: out FLOAT;
3.                   Result: in out BOOLEAN) is
4.   …
5.   D: FLOAT
6.   …
7.   begin
8.     D := B**2-4.0*A*C;
9.          --- first-use-in-callee (A,B,C)
10.    if (Result and D < 0.0) then
11.         --- first-use-in-callee (Result)
12.      Result := FALSE
13.         --- last-def-before-return (Result)
14.      return;
15.    end if;
16.    ROOT1 := (-B+sqrt(D))/(2.0*A);
17.         --- last-def-before-return (ROOT1)
18.    ROOT2 := (-B-sqrt(D))/(2.0*A);
19.         --- last-def-before-return (ROOT2)
20.    Result := TRUE;
21.         --- last-def-before-return (Result)
22.  end ROOT;
```

# *Example II*

- Last-def-before -call(QUADRATIC,15,x) = {6, 10}
- Last-def-before -return(ROOT,result) = {12,20}
- First-use-after-call(QUADRATIC,15,ROOT1) = {17}
- First-use-in-callee(ROOT,A) = {8}

# *Example III*

- T1 = (1, 1, 1, 1) I.e., (Control_Flag,X,Y,Z)
- T2 = (1, 1, 2, 1)
- T3 = (0, 1, 1, 1)
- {T1} satisfies call-coupling
- {T2,T3} satisfies all-coupling-defs , all-coupling-uses, all-coupling-paths

# Java Version

```
1  // Program to compute the quadratic root for two numbers
2  import java.lang.Math;
3
4  class Quadratic
5  {
6  private static float Root1, Root2;
7
8  public static void main (String[] argv)
9  {
10    int X, Y, Z;
11    boolean ok;
12    int controlFlag = Integer.parseInt (argv[0]);
13    if (controlFlag == 1)
14    {
15       X = Integer.parseInt (argv[1]);
16       Y = Integer.parseInt (argv[2]);
17       Z = Integer.parseInt (argv[3]);
18    }
19    else
20    {
21       X = 10;
22       Y = 9;
23       Z = 12;
24    }
25    ok = Root (X, Y, Z);
26    if (ok)
27       System.out.println
28          ("Quadratic: " + Root1 + Root2);
29    else
30       System.out.println ("No solution.");
31 }
32
33 // Three positive integers, finds the quadratic root
34 private static boolean Root (int  A, int B, int C)
```

# Java Version (continued)

```
35  {
36     float D;
37
38     D = (float) Math.pow((double)B, (double)2-4.0*A*C);
39     if (D < 0.0)
40     {
41        Result = false;
42        return (Result);
43     }
44     Root1 = (float) ((-B + Math.sqrt(D)) / (2.0*A));
45     Root2 = (float) ((-B - Math.sqrt(D)) / (2.0*A));
46     Result = true;
47     return (Result);
48  } // End method Root
49
50  } // End class Quadratic
```

# *Case Study*

- All-coupling-uses criterion
- Comparison with Category-partition testing
- Mistix program, C, 31 function units, 65 function calls, 533 LOCs, 21 faults seeded (but 12 could be detected), test cases devised manually
- The faults, category-partition tests, and all-coupling-uses tests were created by different people

# *Results*

- The coupling-based technique performed better (11/12 vs 7/12) than category-partition with half as many test cases (37 vs 72).
- Threats to validity: fault sample, small program, comparisons with other integration test techniques is missing
  - Selecting and Using Data for Integration Testing, Harrold and Soffa, IEEE Software, March 1991
  - A Study of Integration Testing and Software Regression at the Integration Level, Leung and White, IEEE Int. Conf. On Soft. Maintenance, 1990

# *Remarks*

- Empirical studies are rare but they are crucial as theory is of little help to evaluate testing strategies
- But general issues are:
  - How representative are the faults seeded (type, size)?
  - How representative is the program (size, complexity)?
  - Were test cases derived independently of faults? (automation preferred)
  - Results' uncertainty boundaries (faults seeded are samples from a distribution) – it helps determine if observed differences can be obtained by chance

# *Outline*

- Control flow coverage
  - Statement, Edge, Condition, Path coverage
- Data flow coverage
  - Definitions-Usages of data
- Analyzing coverage data
- Mutation Testing
- Integration testing
  - Strategies
  - Criteria
- Conclusions
  - Generating test data, tools, Marick's Recommendations

# *Generating Code-based Tests*

- To find test inputs that will execute an arbitrary statement Q within a program source, the tester must work backward from Q through the program's flow of control to an input statement
- For simple programs, this amounts to solving a set of simultaneous inequalities in the input variables of the program, each inequality describing the proper path through one conditional
- Conditionals may be expressed in local variable values derived from the inputs and those must figure in the inequalities as well

# *Example*

```
int z;
scanf("%d%d", &x, &y);
if (x > 3) {
  z = x+y;
  y+= x;
  if (2*z == y) {
    /* statement to be covered */
…
```

Inequalities:

- `x> 3`
- `2(x+y)=x+y`
  ⇔ `x = -y`

1 Solution:

`X = 4`

`Y= -4`

# *Problems*

- The presence of loops and recursion in the code makes it impossible to write and solve the inequalities in general
- Each pass through a loop may alter the values of variables that figure in a following conditional and the number of passes cannot be determined by static analysis
- Coverage may be 100% and the tester may yet miss some functionalities (omission faults)

# *Marick's Recommendations*

Brian Marick recommends the following approach (for large scale testing):

1. Generate functional tests from requirements and design to try every function.
2. Check the structural coverage after the functional tests are all verified to be successful.
3. Where the structural coverage is imperfect, generate functional tests (not structural) that induce the additional coverage.

This works because form (structure) should follow function!

- Uncovered code must have some purpose, and that purpose has not been invoked, so some function is untested

# *Tools*

- Program Instrumentors
  - Instrument programs (probes)
  - Perform analysis, e.g., coverage, debugging
- Mutation systems
  - User communicates the types of mutants required
  - Mutants are created, and run
  - Perform automatic comparisons and measure test effectiveness (ratio of dead mutants)
- Automatic test drivers (test harness)
  - Simulate the environment for running tests
  - Facilities to specify test data, results

# *Tools*

- Test generation
  - Telcordia: http://xsuds.agreenhouse.com/papers/ieee.html
  - Parasoft Jtest, and C++Test: http://www.parasoft.com/

- Code coverage
  - gcov gcc/g++ coverage tools
  - Rational Purify/Coverage: http://www.rational.com
  - Rational Test RT
  - Telcordia: http://xsuds.argreenhouse.com/
  - McCabe Test and Coverage Server: http://www.mccabe.com
  - IPL Cantata and Cantata++: http://www.qcsltd.com/products.htm