WHITE-BOX TESTING - TEST-SUITE ESTIMATION

CS3213 FSE

Prof. Abhik Roychoudhury

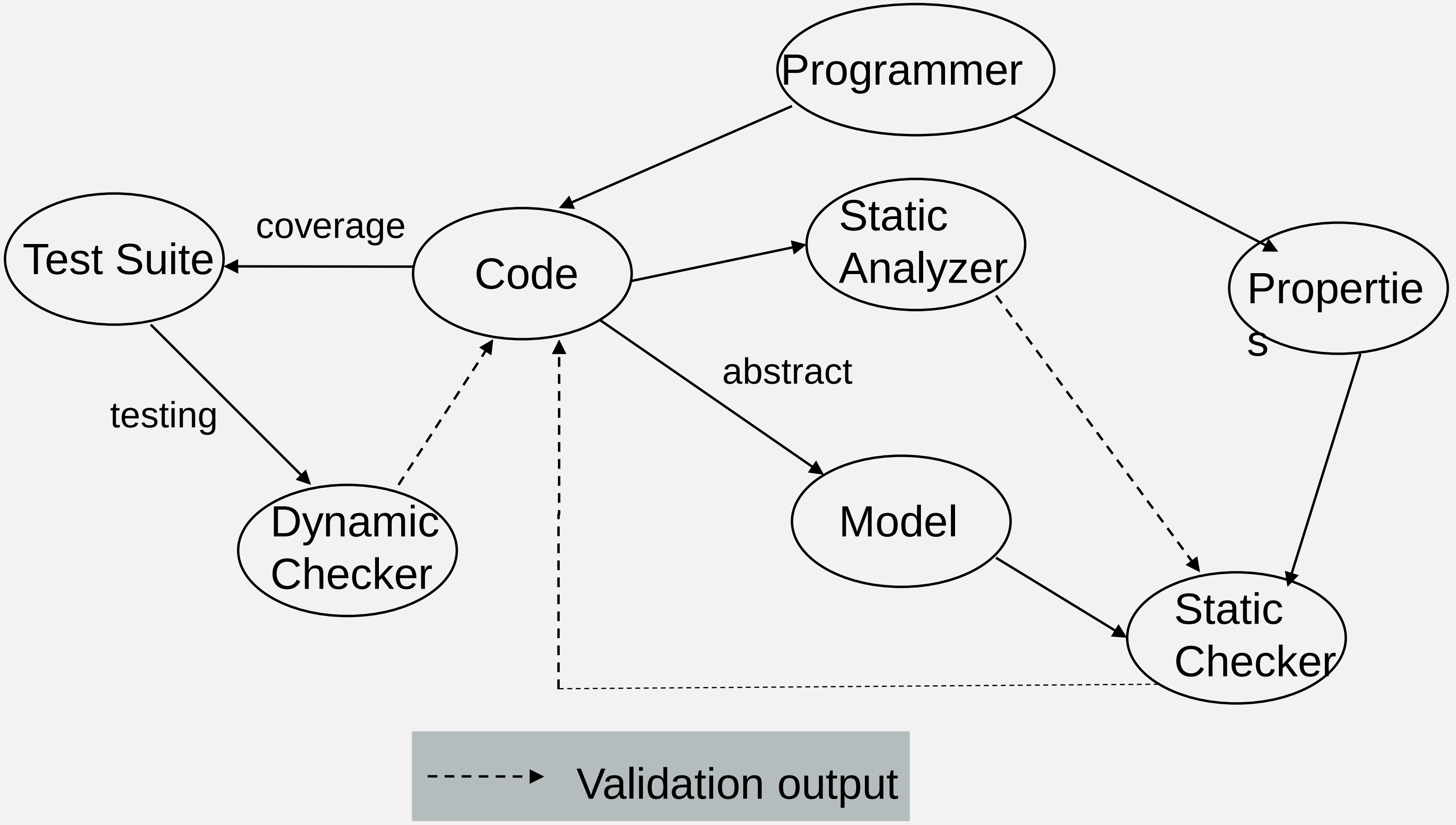
National University of Singapore

WHAT WE DID EARLIER

- System Requirements: Use-cases, Scenarios, Sequence Diagrams
- System structure: Class diagrams
- Discussion on semantics
- System behavior: State diagrams
- Discussion of the thinking behind your course project
- Static analysis and vulnerability detection: Secure SE
- Software Debugging

- Today
  - **White-box Testing**

NO MODEL MAY BE AVAILABLE.



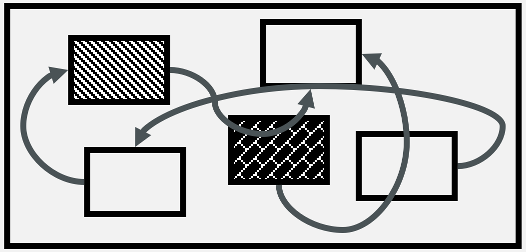
PROGRAMMING

Creativity + Precision

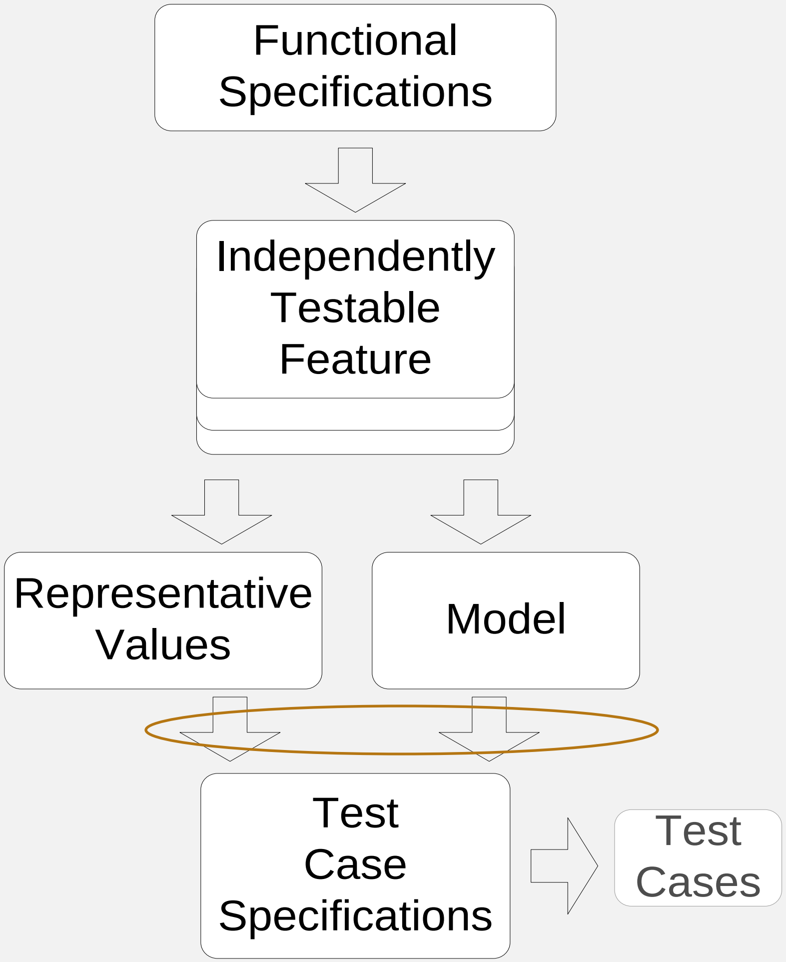
APPROACHES TO TESTING

- Black Box/Functional/Requirements based – treat requirements as rule

- White Box/Structural/Implementation based - *today*



FUNCTIONAL TESTNG



Need to consider combinations of values / models from different testable features.

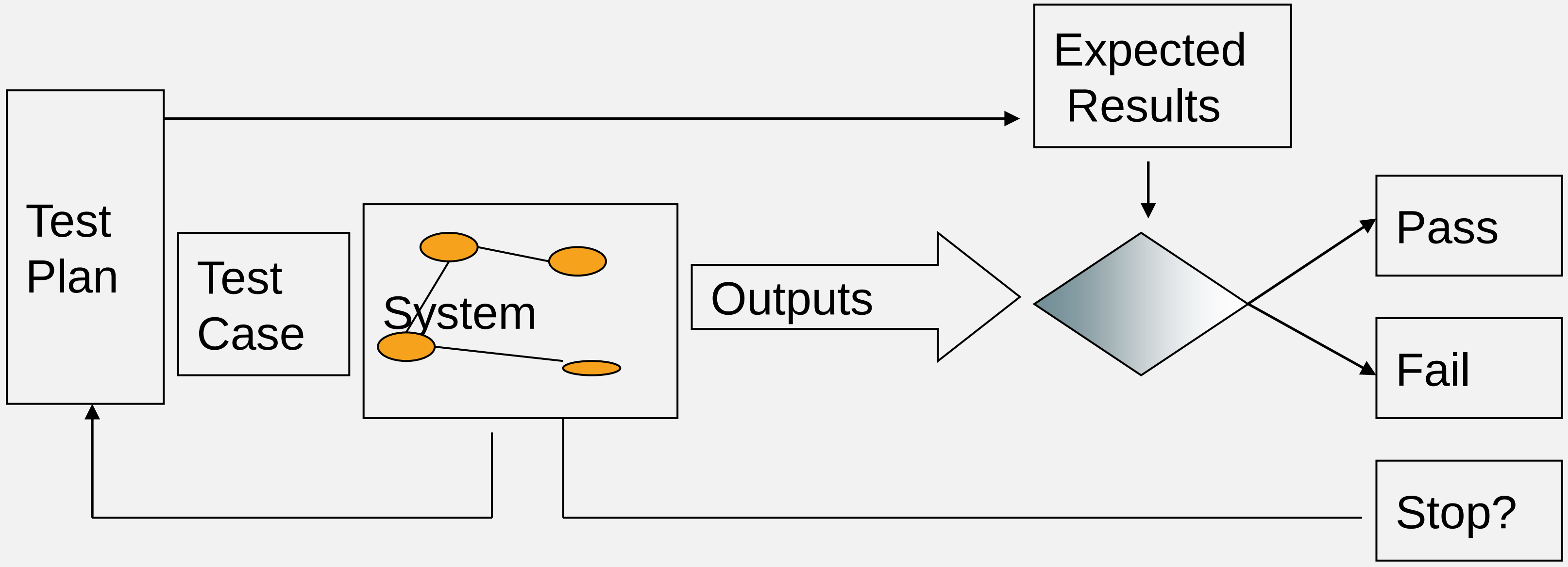
Deal with combinatorial explosion, Techniques exist for handling these.

WHITE-BOX TESTING

Testing that takes into account the internal mechanism of a system or component.

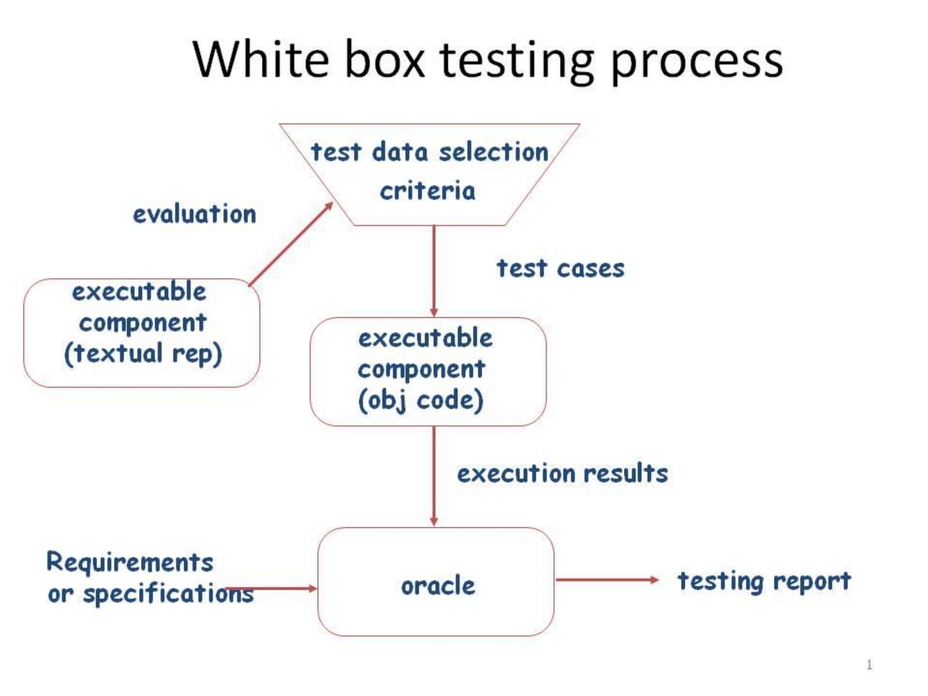
— IEEE

- aka Structural Testing, Glass Box Testing



WHITE BOX/STRUCTURAL TEST DATA SELECTION

- **Coverage based**
  - *Control- flow and data-flow criteria.*
- Fault-based
  - e.g., mutation testing
- Failure-based
  - domain and computation based
  - use representations created by *symbolic execution*



STRUCTURAL TESTING

Structural Coverage based on control-flow criteria

LEARNING OBJECTIVES

- Understand rationale for structural testing
  - How structural (code-based or glass-box) testing complements functional (black-box) testing
- Recognize and distinguish basic terms
  - Adequacy, coverage
- Recognize and distinguish characteristics of common structural criteria
- Understand practical uses and limitations of structural testing

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*

EXAMPLE- ERRORS GETTING MASKED

```
int x; /* Input variable */
int y;
int o; /* Output variable */

input(x);

if (x > 0) {
     y = 3; //change: y = 2;
     if (x − y > 0)
          o = y;
     else
          o = 0;
} else
     o = −1;

if (x > 20)
     o = 10;

output(o);
```

Questions for the class

When will the effects of the change be seen?

When will the effects of the change be masked?

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 TEST DATA

- **Create functional test suite first,** then measure structural coverage to identify see what is missing

- Question to be discussed later:
  - ***Can structural test generation be automated?***

- Questions discussed now:
  - ***Various coverage criteria***

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?

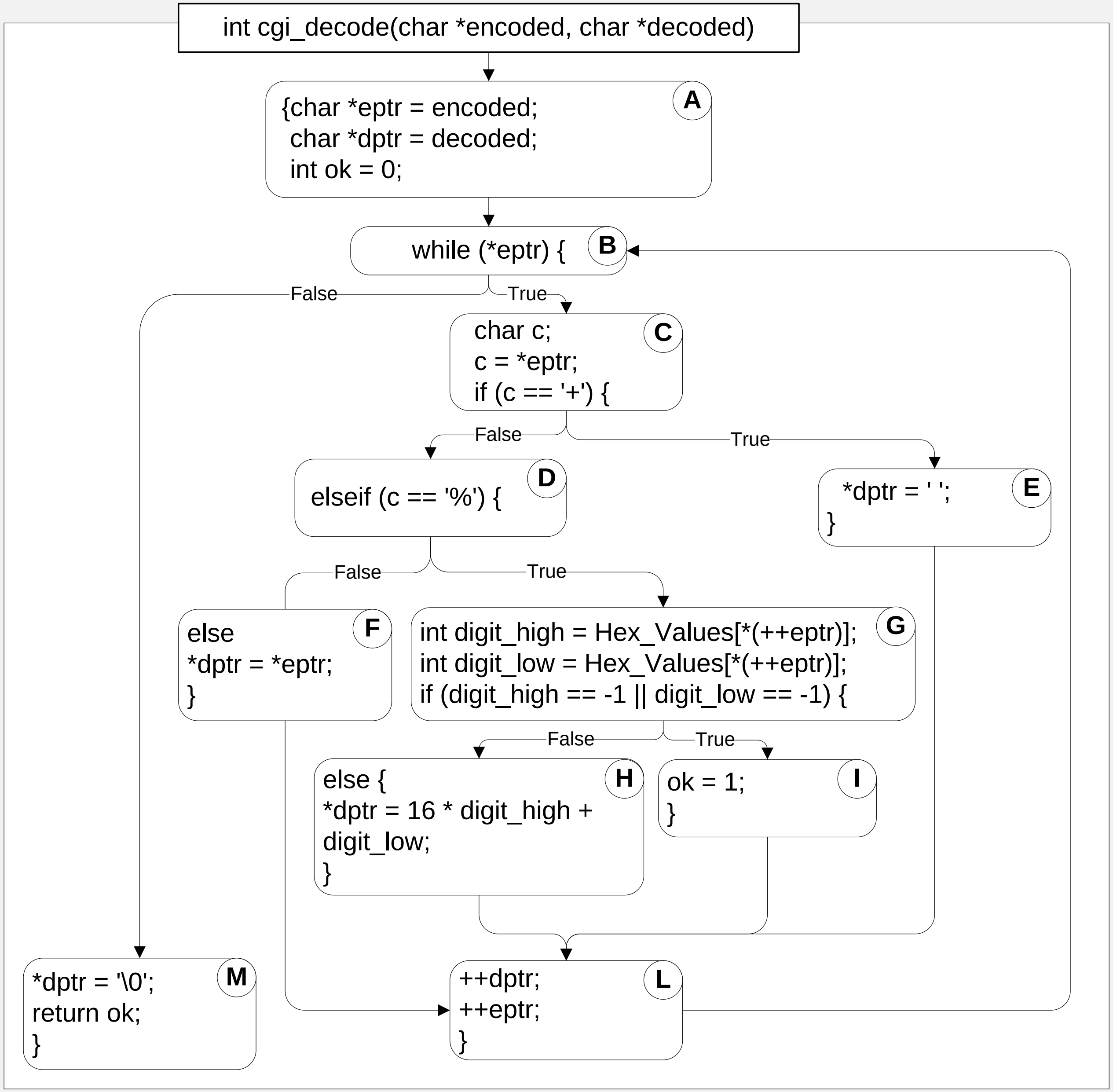
- Nodes in a control flow graph often represent basic blocks of multiple statements
  - Some standards refer to *basic block* coverage or *node coverage*
  - Difference in granularity, not in concept

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.



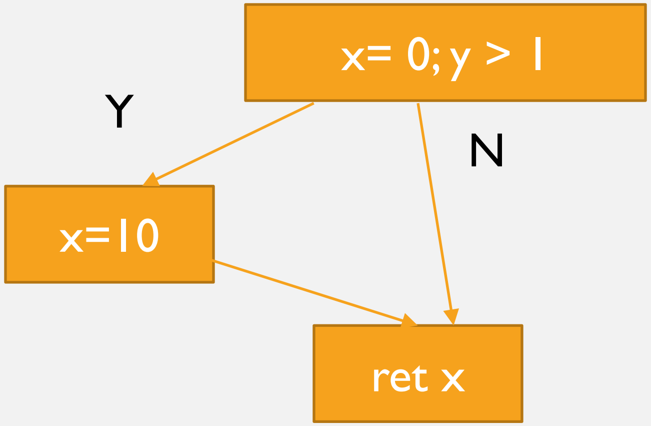
COVERAGE IS NOT SIZE

- Coverage does not depend on the number of test cases
  - $T_0, T_1 : T_1 >_{coverage} T_0$ $T_1 <_{cardinality} T_0$
  - $T_1, T_2 : T_2 =_{coverage} T_1$ $T_2 >_{cardinality} T_1$
- 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

IS IT ENOUGH?

- Why statement coverage may not be adequate?
- Complete statement coverage may not imply executing all branches in a program.

- **Construct an example program now in class to show it.**



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
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)

"ALL BRANCHES" CAN STILL MISS CONDITIONS

- Sample fault: missing operator

  digit_high == 1 || digit_low == -1

- Branch adequacy criterion can be satisfied by varying only digit_low
  - 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

EXAMPLE

- Condition h == 1 || l == -1

- Suppose it is buggy
  - Should be h == -1 || l == -1
  - Achieve branch coverage
    - < h == 0, l == 0>
    - < h == 0, l == -1>

  - Do not vary the faulty condition at all, and the variables involved!!

BASIC CONDITION TESTING

- Adequacy criterion:
  - each basic condition must be executed at least once to true, and …
  - at least once to false.
- 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

Construct an example program now in class to show this claim.

Branch and basic condition are not comparable

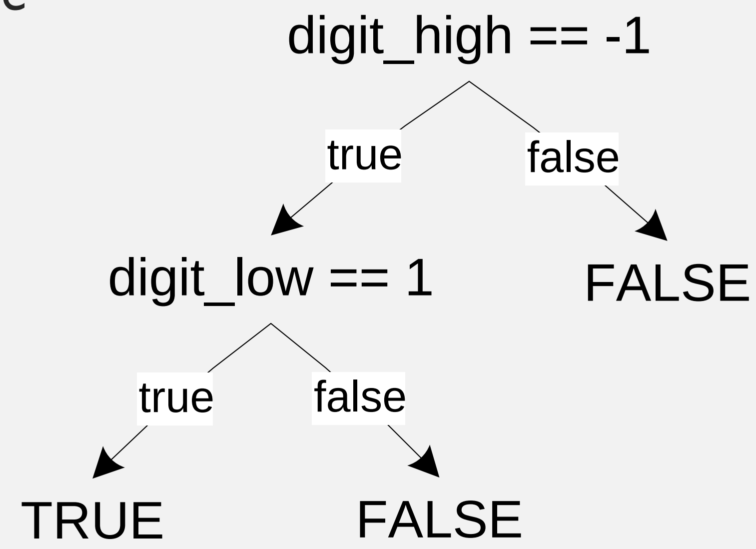
(neither implies the other)

a || b

(a == 0, b ==1)
(a == 1, b == 0)

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	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	—

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	a	b	c	d	e	outcome
(1)	<u>true</u>	--	<u>true</u>	--	<u>true</u>	true
(2)	false	<u>true</u>	true	--	true	true
(3)	true	--	false	<u>true</u>	true	true
(6)	true	--	true	--	<u>false</u>	false
(11)	true	--	<u>false</u>	<u>false</u>	--	false
(13)	<u>false</u>	<u>false</u>	--	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)

MC/DC – INDUSTRY STANDARD

- *"Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect the decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible outcomes."*

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

SUMMARY

- We defined a number of adequacy criteria
  - Test-suite estimation, NOT test-suite construction
- Full coverage is usually unattainable
  - Remember that attainability is an undecidable problem!
- …and when attainable, "test generation" 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
- Rather than requiring full adequacy, the "degree of adequacy" of a test suite is estimated by coverage measures
  - May drive test improvement

DATA FLOW TESTING

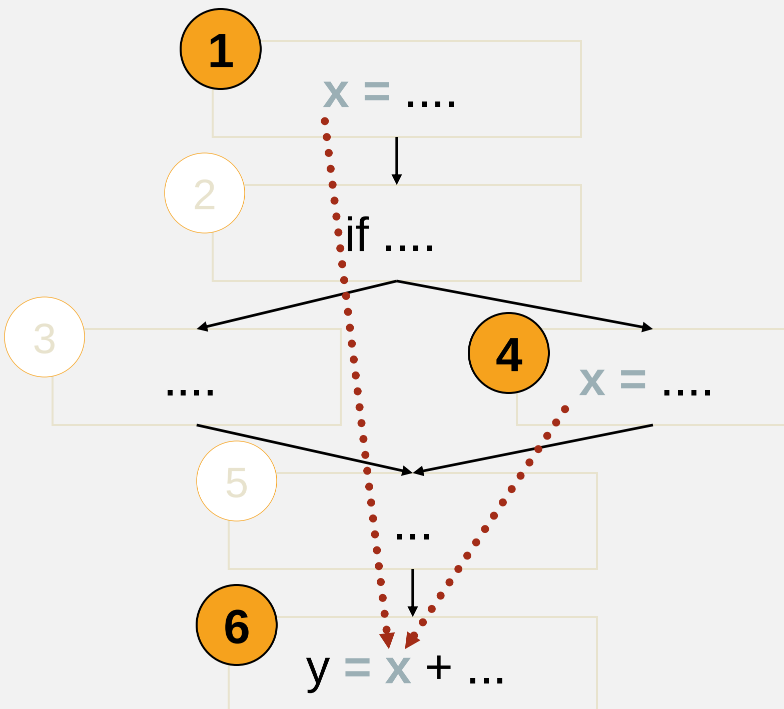
White-box testing

Coverage based on data-flow criteria

MOTIVATION

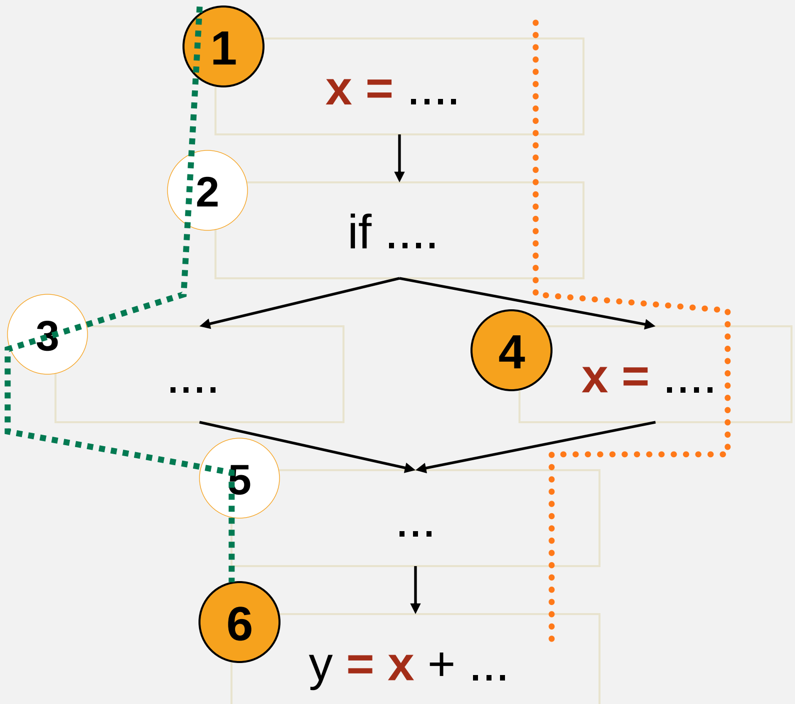
- Middle ground in structural testing
  - Node and edge coverage don't test interactions
  - Path-based criteria require impractical number of test cases
    - And only a few paths uncover additional faults, anyway
  - Need to distinguish "important" paths
- Intuition: Statements interact through *data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

RECAP: REACHING DEF.



- 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

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

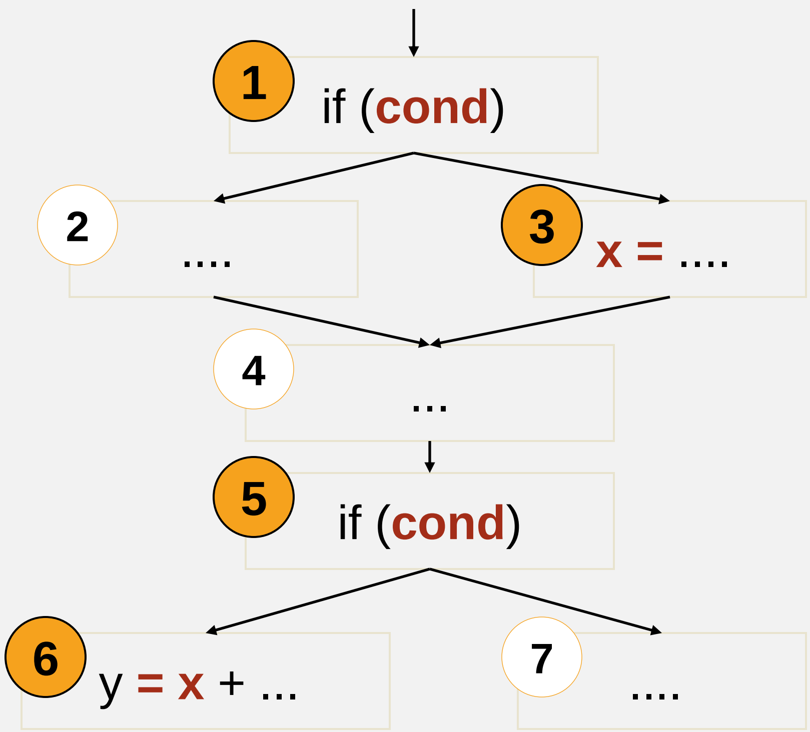
- **All DU pairs**: Each DU pair is exercised by at least one test case

Corresponding coverage fractions can also be defined

ALIASING

- 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?

INFEASIBILITY



- Suppose *cond* has not changed between 1 and 5
  - Or the conditions could be different, but the first implies the second
- Then (3,6) 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

SUMMARY

- Data flow testing attempts to distinguish "important" paths: Interactions between statements
  - Intermediate between simple statement and branch coverage and more expensive path-based structural testing
- Cover Def-Use (DU) pairs: From computation of value to its use
  - Intuition: Bad computed value is revealed only when it is used
  - Levels: All DU pairs, all DU paths, all defs (some use)
- Limits: Aliases, infeasible paths
  - Worst case is bad (undecidable properties, exponential blowup of paths), so pragmatic compromises are required

MUTATION TESTING

Abhik Roychoudhury

National University of Singapore

TEST-SUITE ESTIMATION

- Change the program slightly
  - One line change to introduce an error.
  - Called a Mutant program.
- Check if your test suite can “detect” the error
  - At least one test fails.
- Decide if your test suite is “adequate”

INADEQUATE TEST-SUITES

- Suppose, no test can kill a given mutant.
- Why could this happen?
  - Test suite does not check all behaviors?
  - The mutant is semantically equivalent to the original program?
    - Program equivalence checking – undecidable.

EXAMPLE - MUTANTS

```
Input: a, index
1.   base = a;
2.   sentinel = base;
3.   offset = index;
4.   address = base + offset;
5.   output address, sentinel
```

```
Input: a, index
1.   base = a - 1;
2.   sentinel = base;
3.   offset = index;
4.   address = base + offset;
5.   output address, sentinel
```

```
Input: a, index
1.   base = a;
2.   sentinel = base;
3.   offset = index - 1;
4.   address = base + offset;
5.   output address, sentinel
```

WHY MUTATE?

- Develop program P
- Come up with test suite T based on use-cases and your own intuition
- Test P against T, fix all failing tests.
- P now passes against T
  - Take it for code review in your company.
  - A comment from a colleague
    - In line 75 in file xyz, shouldn't we have

      sentinel = base+1

HOW TO COUNTER SUCH COMMENTS?

- Depend on your reputation
  - I have been coding for 25 years – I know what I did, program passed all tests !
- Connect it back to requirements –
  - may be hard to do, as all program variables do not correspond to quantities mentioned in requirements.
- **Submit the results from *Mutation Testing***

MUTATION TESTING

- Develop program P and test-suite T.
- Generate all mutants of P automatically
  - As per the given mutation operators of P, decided by the programming language.
- How many of the mutants are killed by T
  - Mutation score = (# of killed mutants) / (Total # of mutants)

MUTATION SCORE

Mutation score = (# of killed mutants) / (Total # of mutants)

Can modify it to

$$\text{Mutation score} = \frac{\#\ \text{of killed mutants}}{\text{Total}\ \#\ \text{of mutants} - \#\ \text{of equivalent mutants}}$$

# of equivalent mutants cannot be found exactly – undecidable.

Can replace it with # of equivalent mutants found (using some heuristics, which must be incomplete).

```java
public class Add {
        public static int sum (int a, int b){
          return a+b;
        }
        public static double sum (double a, double b){
           return a+b;
        }
        public static long sum (long a, long b){
            return a+b;
        }
}
```

TC1:

Add o = new Add();

print(o.sum(1,2));

print(o.sum(1.0,2.0));

MutationScore(*TC1) = ?*

```java
public class Add {
        public static int sum(int a, int b) {
        return ++a + b; }
        public static double sum(double a, double b) {
        return a + b; }
        public static long sum(long a, long b) {
        return a + b; }
}
```

```java
public class Add {
        public static int sum(int a, int b) {
        return  a + b; }
        public static double sum(double a, double b) {
        return a + b; }
        public static long sum(long a, long b) {
        return --a + b;}
}
```

```java
public class Add {
        public static int sum(int a, int b) {
        return a + b; }
        public static double sum(double a, double b) {
        return a - b; }
        public static long sum(long a, long b) {
        return a + b;}
}
```

LARGE NUMBER OF MUTANTS!

```
int triangle(int a, int b, int c) {
  if (a <= 0 || b <= 0 || c <= 0) {
      return 4; // invalid
  }
  if (! (a + b > c && a + c > b && b + c > a)) {
      return 4; // invalid
  }
  if (a == b && b == c) {
      return 1; // equilateral
  }
  if (a == b || b == c || a == c) {
      return 2; // isosceles
  }
  return 3;  // scalene
}
```

a + b > c   42 mutants

a - b > c	a * b > c	a / b > c
a % b > c	a > c	b > c
abs(a) - b > c	a - abs(b) > c	a - b > abs(c)
abs(a - b) > c	0 - b > c	a - 0 > c
a - b >= c	a - b < c	a - b <= c
a - b = c	a - b != c	b - b > c
a - a > c	c - b > c	a - c > c
a - b > a	a - b > b	a - b > c
++a - b > c	a - ++b > c	a - b > ++c
--a - b > c	a - --b > c	a - b > --c
++(a - b) > c	--(a - b) > c	-a - b > c
a - -b > c	a - b > -c	(a - b) > c
a - b > 0	-abs(a) - b > c	a - -abs(b) > c
a - b > -abs(c)	-abs(a - b) > c	0 > c

WEAK MUTATION

- Problem: There are lots of mutants. Running each test case to completion on every mutant is expensive
  - Number of mutants grows with the square of program size
- Approach:
  - Execute meta-mutant (with many seeded faults) together with original program
  - Mark a seeded fault as "killed" as soon as a difference in intermediate state is found
    - Without waiting for program completion
    - Re-start with new mutant selection after each "kill"

USING COVERAGE INFORMATION

- Select only test cases which cover the changed code.

- For a test to kill a mutant
  - It should execute the changed code (E)
  - Infect the program state (I, typically achieved)
  - Propagate the infection to program output (P)

- Without execution of changed code, no difference in behavior can be observed!

USING COVERAGE INFORMATION

```
int triangle(int a, int b, int c){

    if (a <= 0 || b <=0 || c <= 0){
        return 4;   // not  a triangle
    }
    if (!(a+b >c && a +c > b && b + c >a)){
        return 4;   // not a triangle
    }
    if  (a == b && b == c){
        return 1;  // equilateral
    }
    if (a == b || b == c || a == c){
        return 2;  // isosceles
    }
    return 3;   // scalene
}
```

(0,0,0)
(1,1,3)
(2,2,2)
(2, 2,3)
(2,3, 4)
(0,1,1)
(4,3,2)
(1,1,1)
(2,3,2)

Only these tests execute mutants in this line

MUTATION TESTING ASSUMPTIONS

- **Competent programmer hypothesis:**
  - Programs are nearly correct
    - Real faults are small variations from the correct program
    - => Mutants are reasonable models of real buggy programs
- **Coupling effect hypothesis:**
  - Tests that find simple faults also find more complex faults
    - Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is good at finding real faults too