CS409
Software Testing

TAN, Shin Hwei
陈馨慧
Southern University of Science and Technology

Administrative Info

- The score for project proposal was uploaded in Sakai. Some members in the group didn't submit
- Lab session today will be final presentation

Administrative Info: Schedule

- Dec 18: MP3 was due
- Dec 21 Lecture: Final Exam Review
- Dec 21 Lab: Final Presentation (All members need to attend and present)
- Dec 25: Final Report due
- Dec 25: All lab assignments due
- Jan 4: Final exam at 16:30-18:30 荔园2栋101（Lychee Hill, Building 2, Room 101）

Administrative Info

All lab assignments due on 25 December 2020, 11.59pm:

Remember to write the answers for all question in README.md and include your name and student id for all assignments:

- Android-graph Lab: https://classroom.github.com/a/-wVDOh_I
- Fuzzing Lab: https://classroom.github.com/a/WOPbCjnZ
- Graph Lab: https://classroom.github.com/a/rkg8YIET
- ISP-Lab(group assignment): https://classroom.github.com/g/tCTOdiKH
- Junit-Lab1: https://classroom.github.com/a/TnI4NoVY
- Junit2: https://classroom.github.com/a/8TQabGyd
- Logic coverage lab: https://classroom.github.com/a/6i6xSkX7
- Logic source code lab: https://classroom.github.com/a/WbKNTVOr
- Monkey delta lab: https://classroom.github.com/a/yq3B85aj
- TDD lab(group assignment): https://classroom.github.com/g/Rf2Mkwo7

Exam Review

Writing a Junit 4 Test case

```
@Test
    public void test1() {

       //A public void method annotated with @Test will be executed as a test case.
    }
```

- A test should be annotated with @Test
- A test do not return anything (return type=void)
- A test should be public

Example test for testing static methods

```
public class Calculation {
    public static int add(int a, int b) {
        return a + b;
    }
    public static int sub(int a, int b) {
        return a - b;
    }
}
```

```
@Test
public void testFailedAdd() {
int total = 9;
int sum = Calculation.add(value1, value2);
assertNotSame(sum, total);
}

@Test
public void testSub() {
int total = 0;
int sub = Calculation.sub(4, 4);
assertEquals(sub, total);
}
```

- Static methods do not depend on the need to create object of a class.
  - You can refer them by the **class name itself (e.g., *Calculation.sub(4.4)*)**

Which tests is correct?

Example 2 is correct!

- Correct method signature should be assertEquals(expected,actual)

Example 1

```
@Test
public void sizeTest() {
 MyStack s = new MyStack ();
 assertEquals (s.size (),0);
}
```

Example 2

```
@Test
public void emptyTest() {
  MyStack s = new MyStack ();
  assertEquals (0, s.size ());
}
```

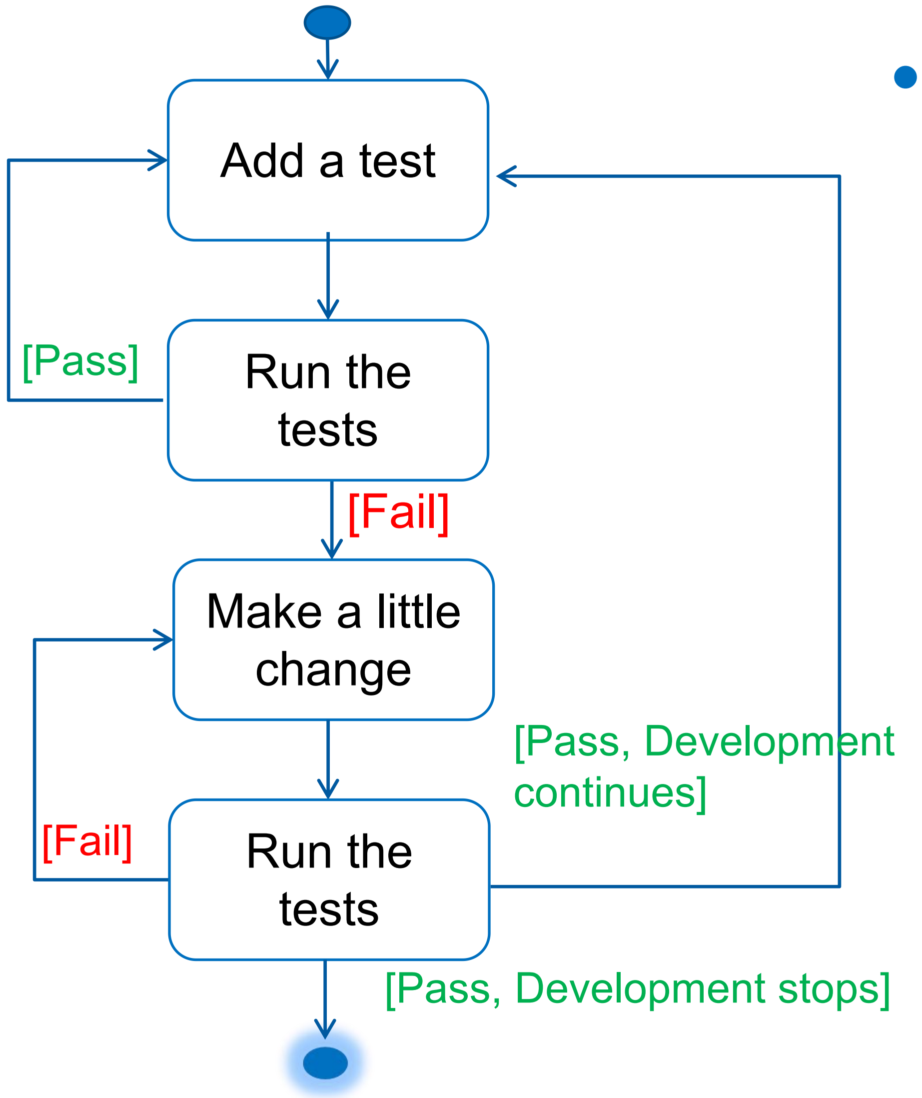
How to test exceptional behavior?

- 2 ways of testing
  - Using try-catch and fail()
  - Using @Test(expected=Exception.class)

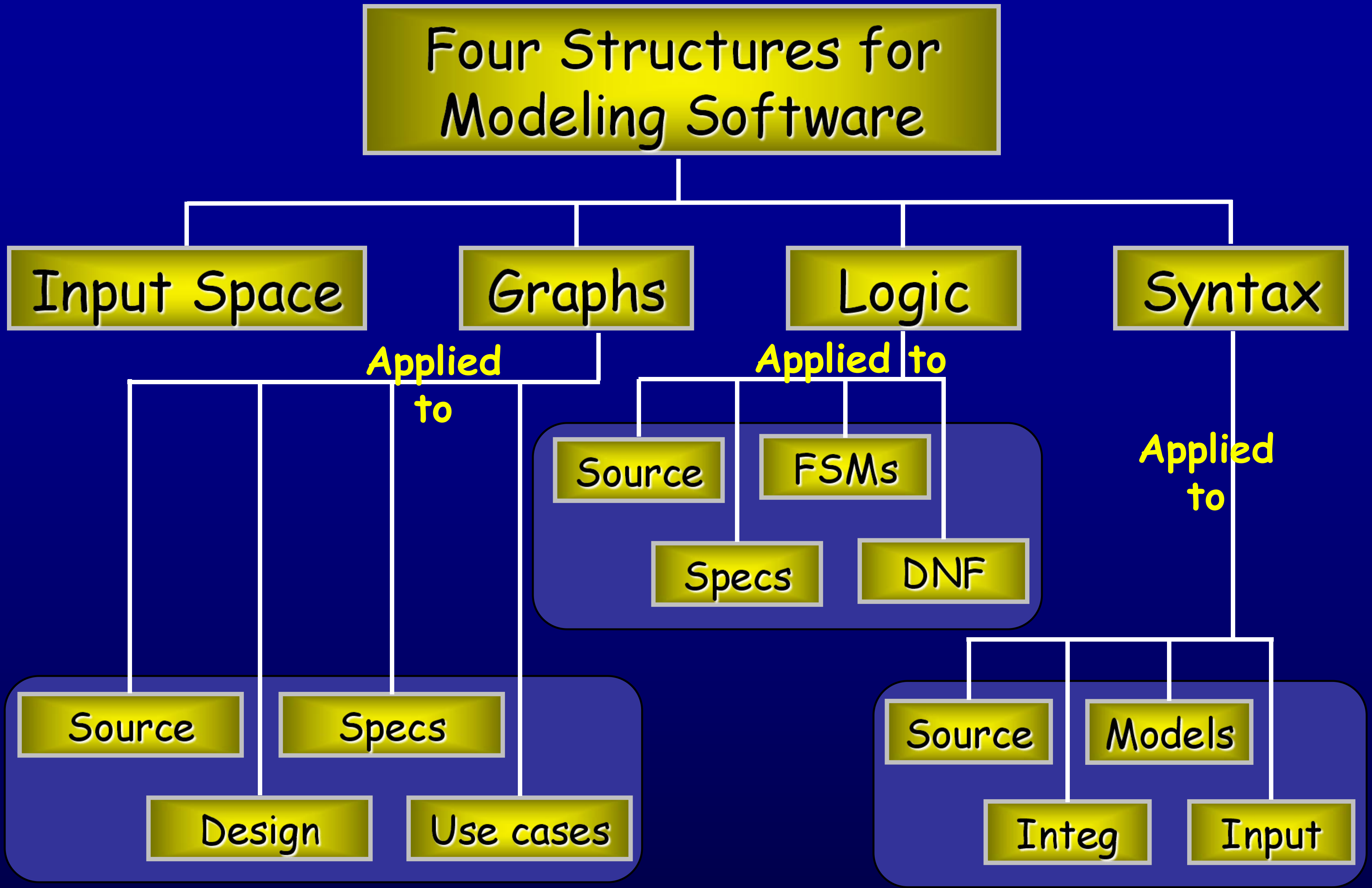
```
@Test
    public void testReadFile2() {
        try {
            FileReader reader = new FileReader("test.txt");
            reader.read();
            reader.close();
            fail("Expected an IOException to be thrown");
        } catch (IOException e) {
            assertThat(e.getMessage(), is("test.txt (No such file or directory)"));
        }
    }
```

```
@Test(expected = FileNotFoundException.class)
    public void testReadFile() throws IOException {
        FileReader reader = new FileReader("test.txt");
        reader.read();
        reader.close();
    }
```

Steps in Test Driven Development (TDD)



- The iterative process
  - Quickly add a test.
  - Run all tests and see the new one fail.
  - Make a little change to code.
  - Run all tests and see them all succeed.
  - Refactor to remove duplication.


Four Structures for Modeling Software
Input Space
Graphs
Logic
Syntax
Applied to
Applied to
Applied to
Source
Specs
Design
Use cases
Source
FSMs
Specs
DNF
Source
Models
Integ
Input

INPUT DOMAIN MODELING

IDM

Interface-based

- Develops characteristics directly from individual input parameters
- Consider syntax
- Example：relationship with zero(>0, =0,<0)

Functionality-based

- Develops characteristics from a behavioral view of the program under test
- Consider domain and semantic knowledge
- Example：Types of Triangle

Interface & Functionality-Based Exercises

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Interface-Based Approach

Two parameters : list, element

Characteristics :

- list is null (block1 = true, block2 = false)
- list is empty (block1 = true, block2 = false)

Functionality-Based Approach

Two parameters : list, element

Characteristics :

- number of occurrences of element in list
  - (0, 1, >1)
- element occurs first in list
  - (true, false)
- element occurs last in list
  - (true, false)

Perform IDM for:

- Interface-Based
- Functionality-Based

Coverage Criteria for IDM

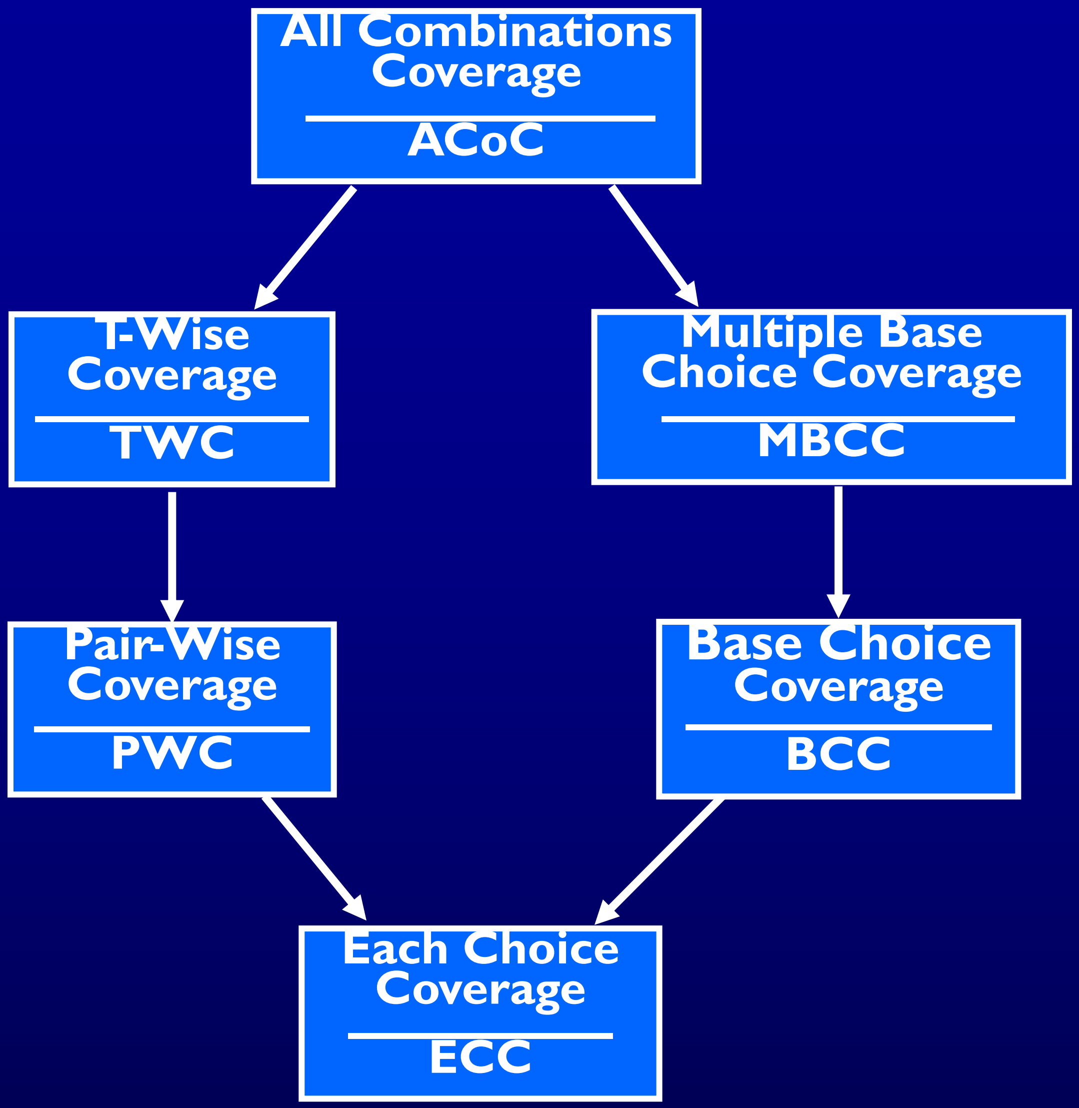
All Combinations (ACoC) : All combinations of blocks from all characteristics must be used.

Each Choice Coverage (ECC) : One value from each block for each characteristic must be used in at least one test case.

Pair-Wise Coverage (PWC) : A value from each block for each characteristic must be combined with a value from every block for each other characteristic.

Base Choice Coverage (BCC) : A base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.

ISP Coverage Criteria Subsumption



GRAPH

Overview

- A common application of graph criteria is to program source
- Graph : Usually the control flow graph (CFG)
- Node coverage : Execute every statement
- Edge coverage : Execute every branch
- Loops : Looping structures such as for loops, while loops, etc.
- Data flow coverage : Augment the CFG
  - defs are statements that assign values to variables
  - uses are statements that use variables

Draw CFG Graph

- Draw one node for each basic block
- Connects basic block with edge
- Label each edge with branch predicate

Edge

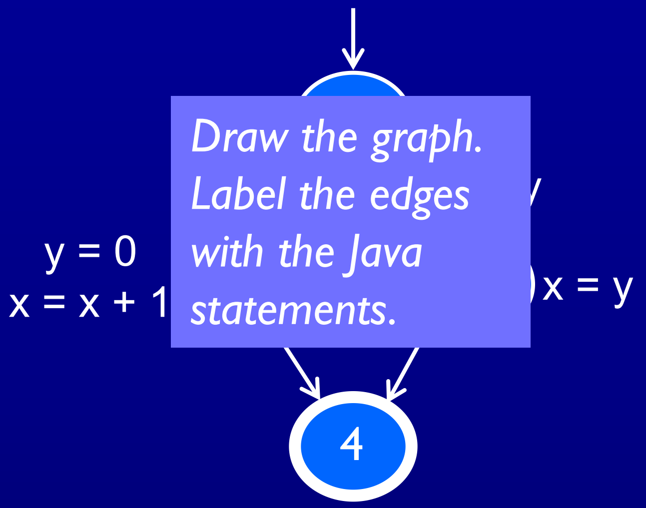
```
if (x ==0)
{
   y = 0;
   x = x + 1;
}else{
   y++;
}
```

Basic Blocks (One Node)

Basic Block (One Node)

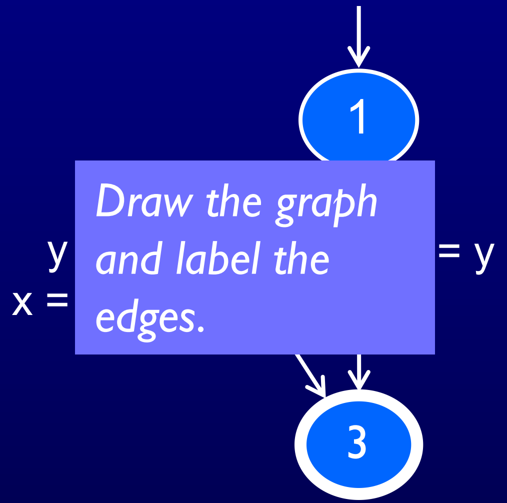
CFG : The if Statement

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```



Draw the graph. Label the edges with the Java statements.

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```



Draw the graph and label the edges.

Data Flow Definitions

- A usage node is a predicate use (**P-Use**) if variable v appears in a predicate expression (e.g., x>y)
- A usage node is a computation use (**C-Use**) if variable v appears in a computation (e.g., x+y)
- A definition-use path (**du-path**) with respect to a variable v is a path whose first node is a defining node for v, and its last node is a usage node for v
- A du-path with no other defining node for v is a definition-clear path (**dc-path**)

An Example

- What is the P-use of i?
- What is the C-use of i?

```
int max = 0;
int i = s.nextInt();
while (i > 0)
 if (i > max) {
   max = i;
 }
 i = s.nextInt();
}
System.out.println(max);
```

Definitions of max

A definition of i

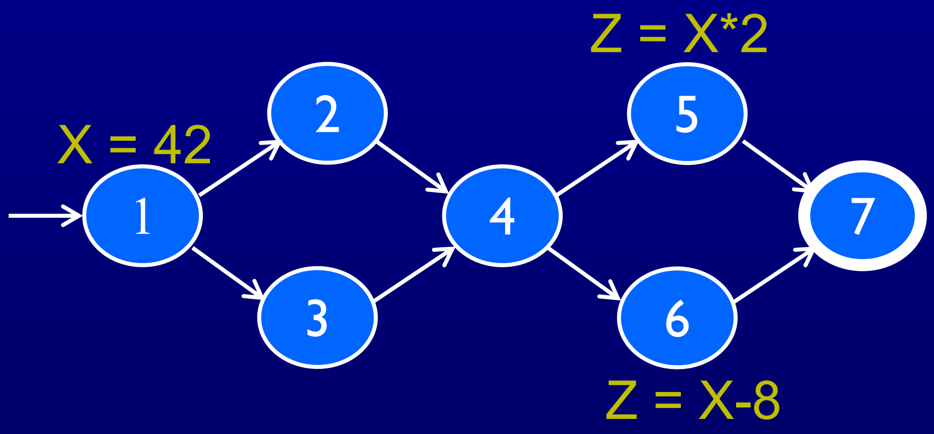
P-uses of i

A C-use of i

Data Flow Criteria

Goal : Ensure that values are computed and used correctly

- Definition (def) : A location where a value for a variable is stored into memory
- Use : A location where a variable's value is accessed



Defs: def (1) = { X }

def (5) = { Z

def (6) = { Z

Uses: use (5) = { X }

use (6) = { X }

Fill in these sets

The values given in defs should reach at least one, some, or all possible use

Graph Coverage Criteria

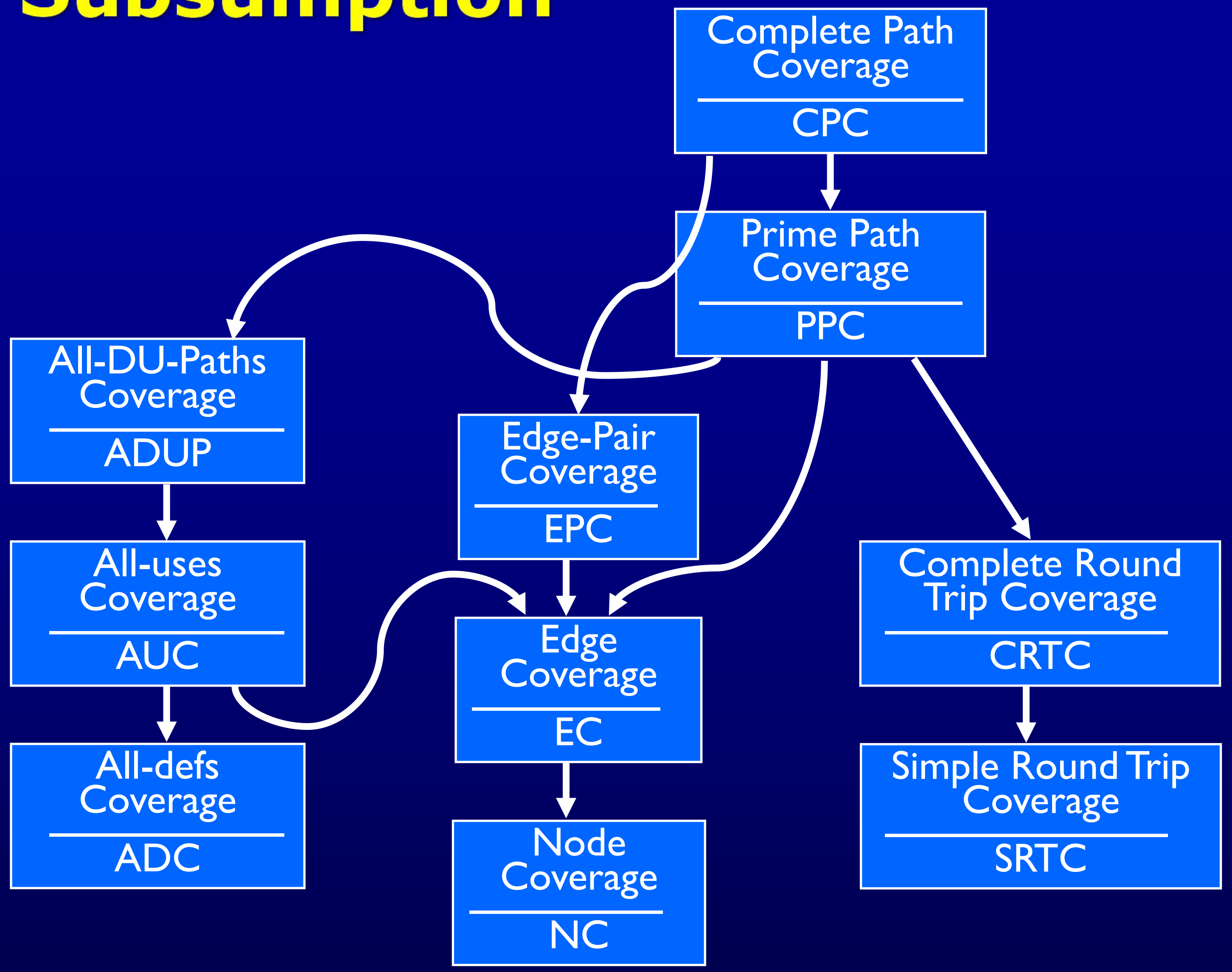
Node Coverage (NC) : TR contains each reachable node in G

Edge Coverage (EC) : TR contains each reachable path of length up to 1, inclusive, in G.

Edge-Pair Coverage (EPC) : TR contains each reachable path of length up to 2, inclusive, in G.

Complete Path Coverage (CPC) : TR contains all paths in G

Graph Coverage Criteria Subsumption



LOGIC

Predicate & Clause

$$\underbrace{\overbrace{(a < b)}^{\text{clause 1 /condition 1}} \vee \overbrace{f(z)}^{\text{clause 2 /condition 2}}}_{\text{predicate/ decision}}$$

- A *predicate* is an expression that evaluates to a boolean value
- A *clause* is a predicate with no logical operators

Predicate and Clause Coverage

- The first two criteria require that each predicate and each clause be evaluated to both true and false

<u>Predicate Coverage (PC)</u> : For each *p* in *P*, *TR* contains two requirements: *p* evaluates to true, and *p* evaluates to false.

- PC does not evaluate all the clauses, so …

<u>Clause Coverage (CC)</u> : For each *c* in *C*, *TR* contains two requirements: *c* evaluates to true, and *c* evaluates to false.

ACC, RACC, CACC

$p = a \lor (b \land c)$

Major clause : a

a = true, b = false, c = true

a = false, b = false, c = false

c = false

Is this allowed ?

- 3 separate criteria :
  - General Active Clause Coverage (GACC)
    - Minor clauses do not need to be the same
  - Restricted Active Clause Coverage (RACC)
    - Minor clauses do need to be the same
  - Correlated Active Clause Coverage (CACC)
    - Minor clauses force the predicate to become both true and false

Making Clauses Determine a Predicate

(8.1.5)

- Finding values for minor clauses c_j is easy for simple predicates
- But how to find values for more complicated predicates ?
- Definitional approach:
  - $p_{c=true}$ is predicate p with every occurrence of c replaced by *true*
  - $p_{c=false}$ is predicate p with every occurrence of c replaced by *false*
- To find values for the minor clauses, connect $p_{c=true}$ and $p_{c=false}$ with exclusive *OR*

$$p_c = p_{c=true} \oplus p_{c=false}$$

- After solving, p_c describes exactly the values needed for c to determine p

Exercises: the values needed for a to determine *p*

$$p = a \vee b$$

$$
\begin{aligned}
p_a &= p_{a=true} \oplus p_{a=false} \\
&= (true \vee b) \text{ XOR } (false \vee b) \\
&= true \text{ XOR } b \\
&= \neg b
\end{aligned}
$$

$$p = a \wedge b$$

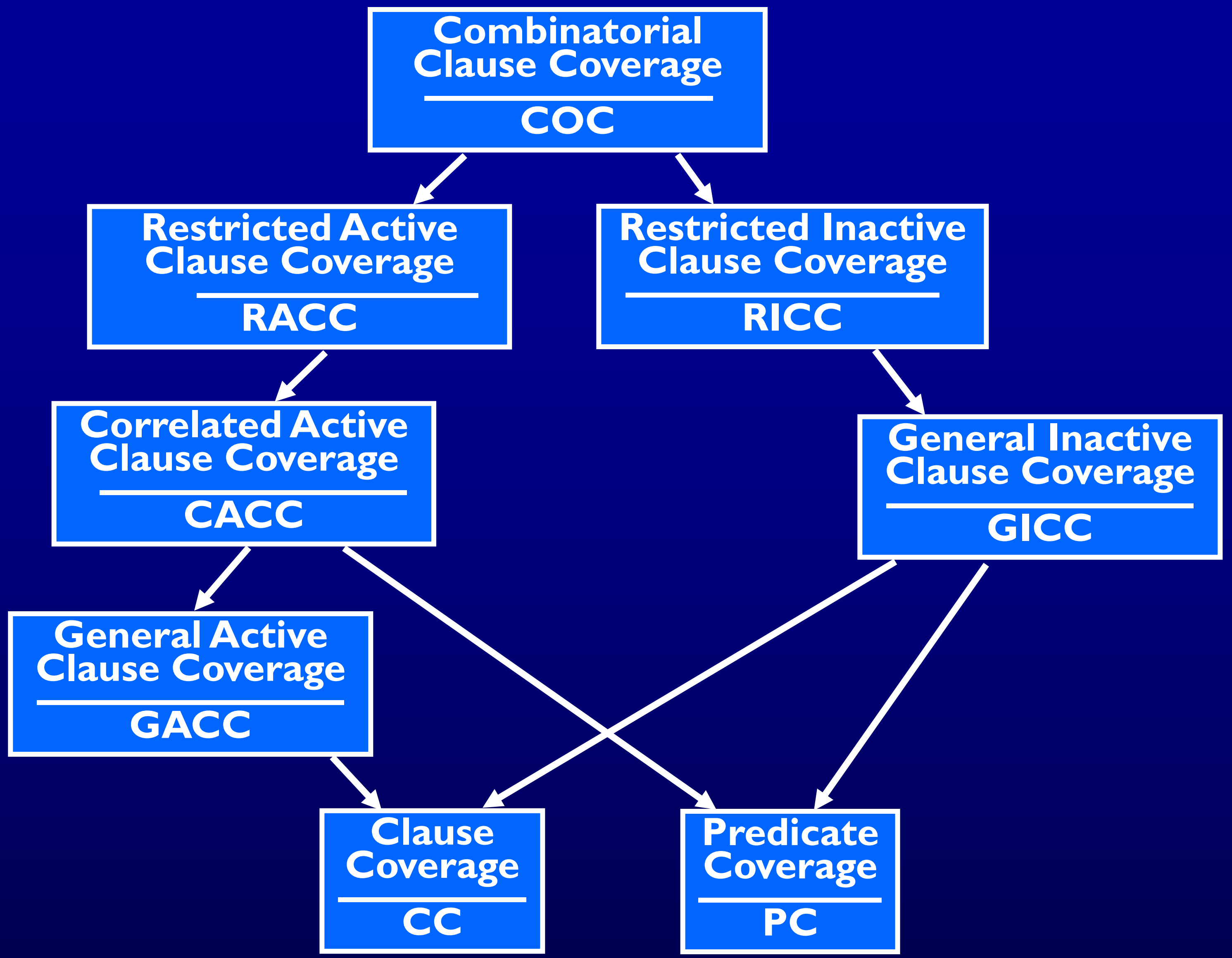
$$
\begin{aligned}
p_a &= p_{a=true} \oplus p_{a=false} \\
&= (true \wedge b) \oplus (false \wedge b) \\
&= b \oplus false \\
&= b
\end{aligned}
$$

$$p = a \vee (b \wedge c)$$

$$
\begin{aligned}
p_a &= p_{a=true} \oplus p_{a=false} \\
&= (true \vee (b \wedge c)) \oplus (false \vee (b \wedge c)) \\
&= true \oplus (b \wedge c) \\
&= \neg (b \wedge c) \\
&= \neg b \vee \neg c
\end{aligned}
$$

- **"*NOT b* $\vee$ *NOT c*" means either *b* or *c* can be false**
- **RACC requires the same choice for both values of *a*, CACC does not**

Logic Criteria Subsumption



SYNTAX

Examples

```
DebitCard>>= anotherDebitCard
  ^(type = anotherDebitCard type)
   and: [ number = anotherDebitCard number ]
```

Operator: Change #and: by #or:

```
CreditCard>>= anotherDebitCard
  ^(type = anotherDebitCard type)
    or: [ number = anotherDebitCard number ]
```

How does it work?
2nd Step: Try to Kill the Mutant

The "Mutant"

A Killer
tries to kill the Mutant!

The Test Suite

All tests run → The Mutant Survives!!!

A test fails or errors → The Mutant Dies

Meaning…

The Mutant Survives → The case generated by the mutant is not tested

The Mutant Dies → The case generated by the mutant is tested

Syntax-Based Coverage Criteria

<u>Mutation Coverage (MC)</u> : For each *m* $\in$ *M*, TR contains exactly one requirement, to kill *m*.

- The RIPR model:
  - *Reachability* : The test causes the faulty statement to be reached (in mutation – the mutated statement)
  - *Infection* : The test causes the faulty statement to result in an incorrect state
  - *Propagation* : The incorrect state propagates to incorrect output
  - *Revealability* : The tester must observe part of the incorrect output

Syntax-Based Coverage Criteria

1) Strongly Killing Mutants:

Given a mutant $m \in M$ for a program P and a test t, t is said to *strongly kill* m if and only if the output of t on P is different from the output of t on m

2) Weakly Killing Mutants:

Given a mutant $m \in M$ that modifies a location l in a program P, and a test t, t is said to *weakly kill* m if and only if the state of the execution of P on t is different from the state of the execution of m on t immediately after l

- Weakly killing satisfies reachability and infection, but not propagation

Weak Mutation

Weak Mutation Coverage (WMC) : For each $m \in M$, TR contains exactly one requirement, to weakly kill m.

- "Weak mutation" is so named because it is easier to kill mutants under this assumption
- Weak mutation also requires less analysis
- A few mutants can be killed under weak mutation but not under strong mutation (no propagation)
- Studies have found that test sets that weakly kill all mutants also strongly kill most mutants

Weak Mutation Example

Mutant 1 in the Min( ) example is:

```
int Min (int A, int B)
{
      int minVal;
      minVal = A;
Δ 1  minVal = B;
      if (B < A)
      {
          minVal = B;
      }
      return (minVal);
} // end Min
```

With one or two partners :

1. Find a test that weakly kills the mutant, but not strongly
2. Generalize : What must be true to weakly kill the mutant, but not strongly?
3. Try to write down the conditions needed to (i) reach the mutated statement, (ii) infect the program state, and (iii) propagate to output

Weak Mutation Example

```
    minVal = A;
Δ 1 minVal = B;
    if (B < A)
        minVal = B;
```

1. Find a test that weakly kills the mutant, but not strongly

   A = 5, B = 3

2. Generalize : What must be true to weakly kill the mutant, but not strongly?

   B < A // minVal is set to B on for both

3. RIP conditions

   Reachability : *true* // we always reach

   Infection : *A ≠ B // minVal has a different value*

   Propagation : *(B < A) = false // Take a different branch*

Equivalent Mutation Example

Mutant 3 in the Min() example is equivalent:

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

With one or two partners

1. Convince yourselves that this mutant is equivalent
2. Briefly explain why
3. Try to prove the equivalence

   Hint : Think about what must be true to kill the mutant

Equivalent Mutation Example

```
      minVal = A;
      if (B < A)
Δ 3   if (B < minVal)
```

1. Convince yourselves that this mutant is equivalent
2. Briefly explain why

   A and minVal have the same value at the mutated statement

3. Try to prove the equivalence

   Hint : Think about what must be true to kill the mutant

   Infection : (B < A) != (B < minVal)
   Previous statement : minVal = A
   Substitute : (B < A) != (B < A)
   Contradiction … therefore, equivalent

```
public static int cal (int month1, int day1, int month2, int day2, int year){
    int numDays;
  if (month2 == month1)
 Δ2  numDays = day2 + day1;
      numDays  = day2 - day1;
    else{
      // Skip month 0.
      int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
      // Are we in a leap year?
      int m4 = year % 4;
      int m100 = year % 100;
      int m400 = year % 400;
if ((m4 != 0) || ((m100 == 0) && (m400 != 0)))
        daysIn[2] = 28;
      else
        daysIn[2] = 29;

      // start with days in the two months
      numDays = day2 + (daysIn[month1] - day1);

      // add the days in the intervening months
      for (int i = month1 + 1; i <= month2-1; i++)
        numDays = daysIn[i] + numDays;
    }
    return (numDays);}
```

1. Find a test that weakly kills the mutant, but not strongly
2. Generalize : What must be true to weakly kill the mutant, but not strongly?
3. Try to write down the conditions needed to (i) reach the mutated statement, (ii) infect the program state, and (iii) propagate to output

OTHER TECHNIQUES

Automated Debugging

Debugging techniques that use program executions in different ways:

- Statistical Fault Localization
  - Assign scores to program statements based on their occurrence in passing / failing tests.
- Dynamic Slicing
  - A dynamic slice contains all statements that affect the value of a variable at a program point for a particular execution of the program rather than all statements that may have affected the value of a variable at a program point for any arbitrary execution of the program.
- Delta Debugging
  - A minimization algorithm
  - Finds 1-minimal instead of local minimum test case due to performance.

What is symbolic execution?

Testing/ Concrete Execution

– Each test only explores one possible execution

- `assert(f(3) == 5)`

Symbolic execution

- *Generalizes* test cases
  - Allows *unknown* symbolic variables in evaluation
    - `y = α;`
    - `assert(f(y) == 2*y-1);`
  - If execution path depends on *unknown*, conceptually *fork* symbolic executor
    - ```
      int f(int x) {
        if (x > 0) then return 2*x - 1;
        else return 10;
      }
      ```
      ```

Path condition computation

in == 5

```
1 input in;
2 z = 0; x = 0;
3 if  (in > 0){
4     z = in *2;
5    x = in +2;
6    x = x + 2;
7  }
8 else  …
9 if ( z  > x){
   return error;
}
```

Line#	Assignment store	Path condition
1	{}	*true*
2	{(z,0),(x,0)}	*true*
3	{(z,0),(x,0)}	*in > 0*
4	{(z,2*in), (x,0)}	*in > 0*
5	{(z,2*in), (x,in+2)}	*in > 0*
6	{(z,2*in), (x, in+4)}	*in > 0*
7	{(z, 2*in), (x, in+4)}	*in > 0*
9	{(z, 2*in), (x, in+4)}	$in>0 \wedge (2*in > in +4)$

What is Z3?

- State-of-the-art SMT solver from Microsoft Research.
- Used to check the satisfiability of logical formulas over one or more theories.
- It is a low level tool that is often used as a component in the context of other tools that require solving logical formulas

Script for Z3?

- Z3 input format is an extension of the SMT-LIB 2.0 standard.
- A Z3 script is a sequence of commands. Z3 maintains a stack of user provided formulas and declarations. These are the **assertions** provided by the user.
  - **declare-const** declares a constant of a given type.
  - **declare-fun** declares a function.
  - **assert** adds a formula into the Z3 internal stack.
  - **check-sat** returns sat if the set of formulas in the Z3 stack is satisfiable.
  - When the command **check-sat** returns sat, the command **get-model** can be used to retrieve an interpretation that makes all formulas on the Z3 internal stack true.

Fuzzing

- Automatically generate test cases
- Many slightly anomalous test cases are input into a target
- Application is monitored for errors
- Inputs are generally either file based (.pdf, .png, .wav, etc.) or network based (http, SNMP, etc.)
- Example fuzzer
  - AFL
  - libfuzzer

Mutation-based vs. Generation-based

- Mutation-based fuzzer
  - Pros: Easy to set up and automate, little to no knowledge of input format required
  - Cons: Limited by initial corpus, may fall for protocols with checksums and other hard checks
- Generation-based fuzzers
  - Pros: Completeness, can deal with complex dependncies (e.g, checksum)
  - Cons: writing generators is hard, performance depends on the quality of the spec

What is program synthesis?

Goal: Synthesize a computational concept in some underlying language from user intent using some search technique.

- Example Synthesizer:
  - FlashFill
  - Sketch
  - SQLizer

Synthesis

=

an unusually concise / intuitive programming language

+

a compiler based on search

Synthesis Techniques

- Programming by Example
- Syntax-guided Synthesis
- Counter-example guided inductive synthesis
- Programming by Sketching