COMP5111 – Fundamentals of Software Testing and Analysis

Reviews

Shing-Chi Cheung

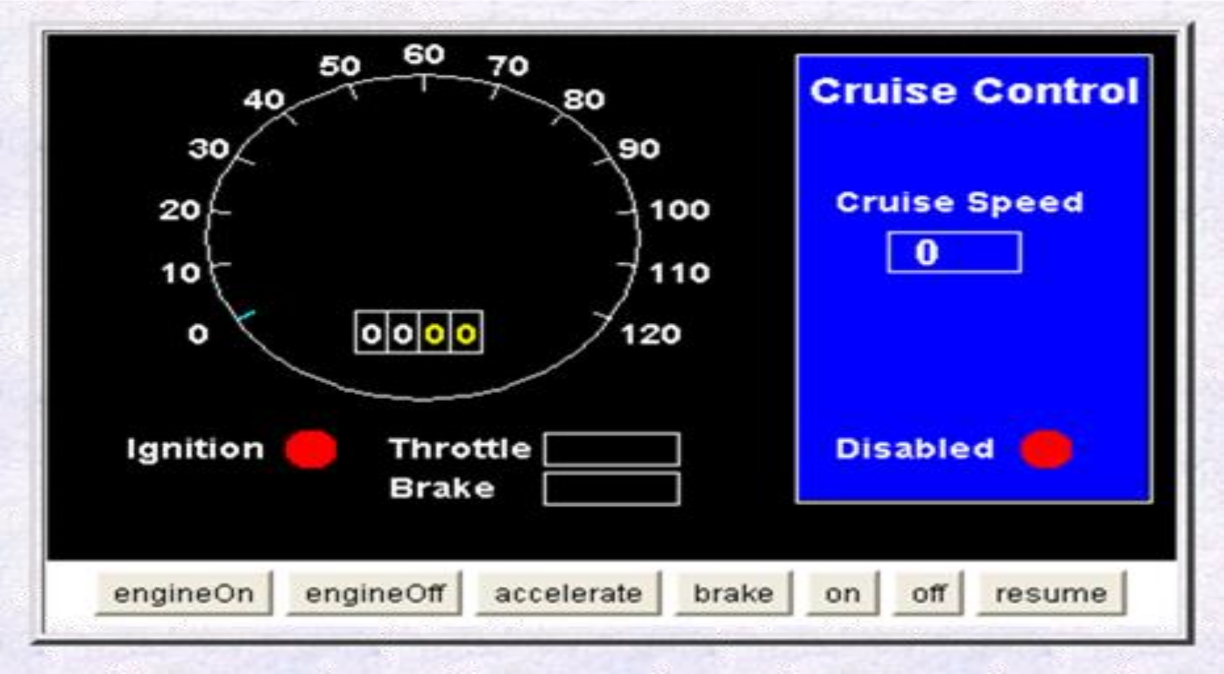
Department of Computer Science & Engineering

HKUST

Notes Review

Here! Try it out Yourself!

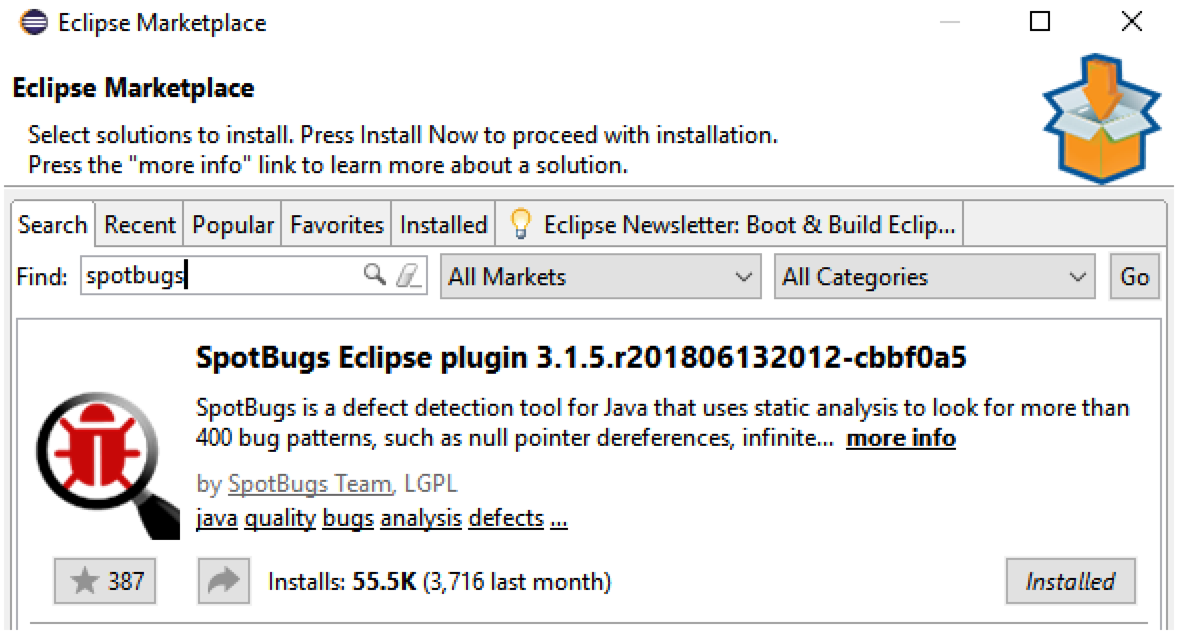
(http://www.cse.ust.hk/~scc/teaching/CruiseControl.html)



Recall: Static and Dynamic Analysis

- Static Analysis: Validate without program execution
  - This include code review and symbolic execution
- Dynamic Analysis: Validate by executing the program with real inputs
  - Testing
- Big Code Analysis: Validate based on big data mining from public repositories and forums

SpotBugs

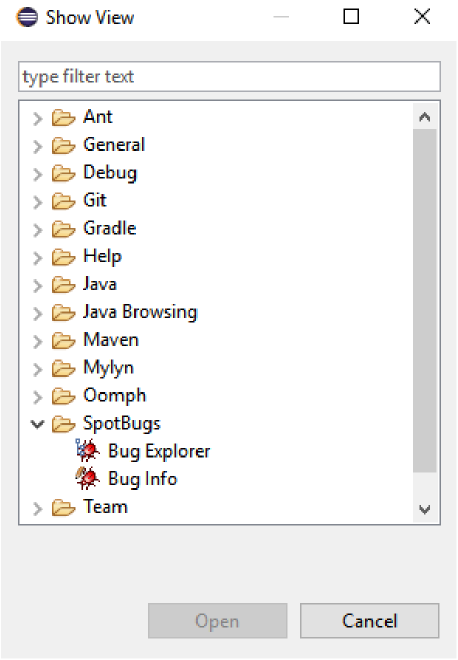


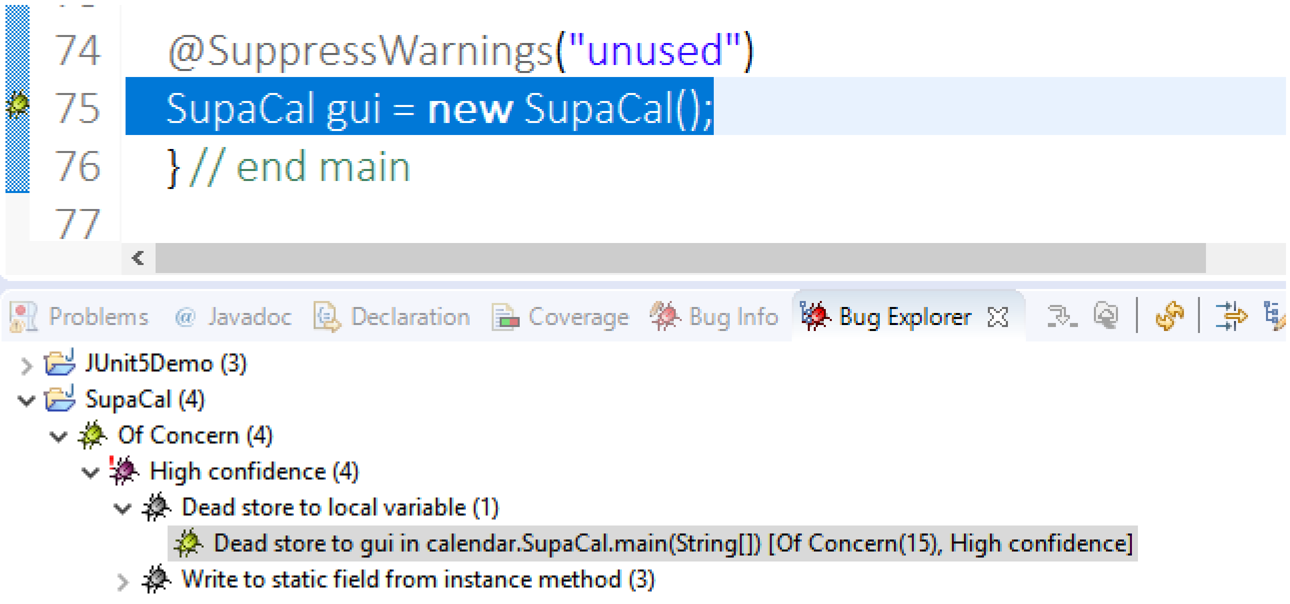
One of the most popular Eclipse Plugins!!

What SpotBugs can do?

- It comes with 200+ rules divided into different categories:
  - Correctness
    - e.g., infinite recursive loop, reads a field that is never written
  - Bad practice
    - e.g., code that drops exceptions or fails to close a file
  - Performance
  - Multithreaded correctness
  - Dodgy
    - e.g., unused local variables or unchecked casts

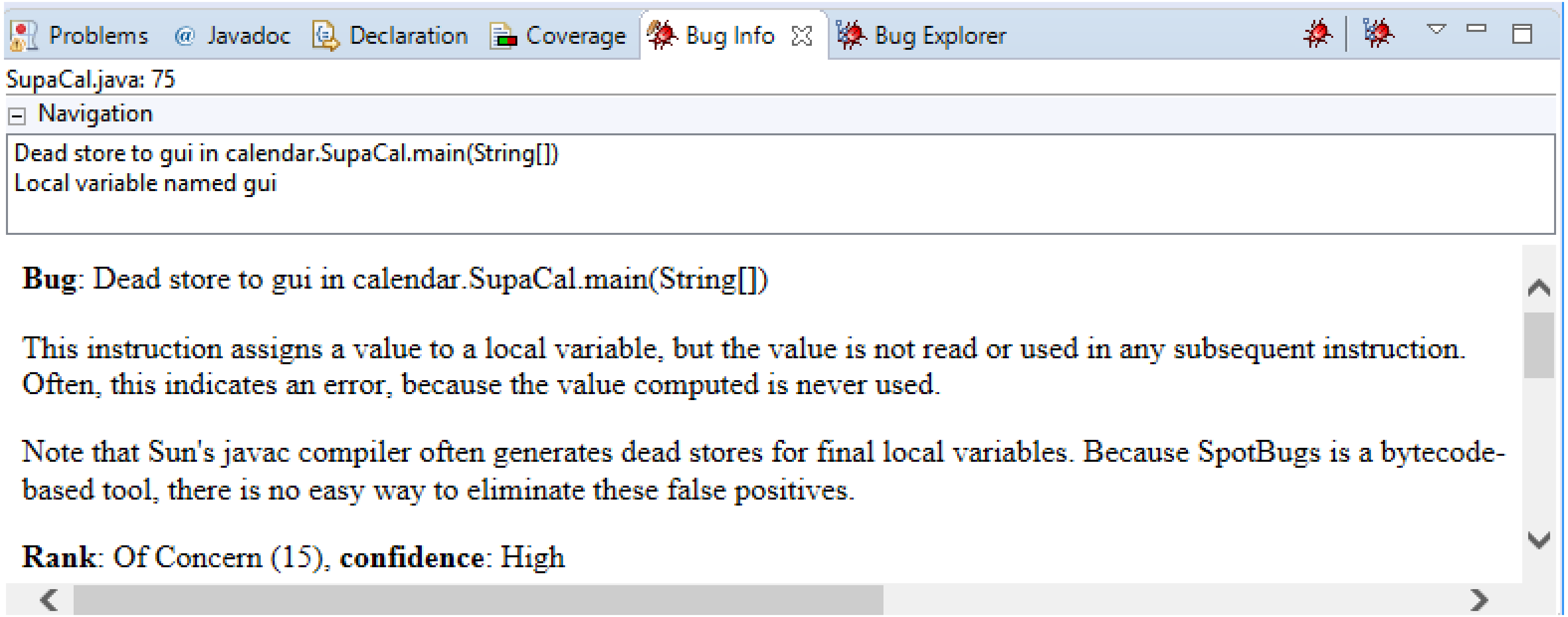
Using SpotBugs – Bug Explorer





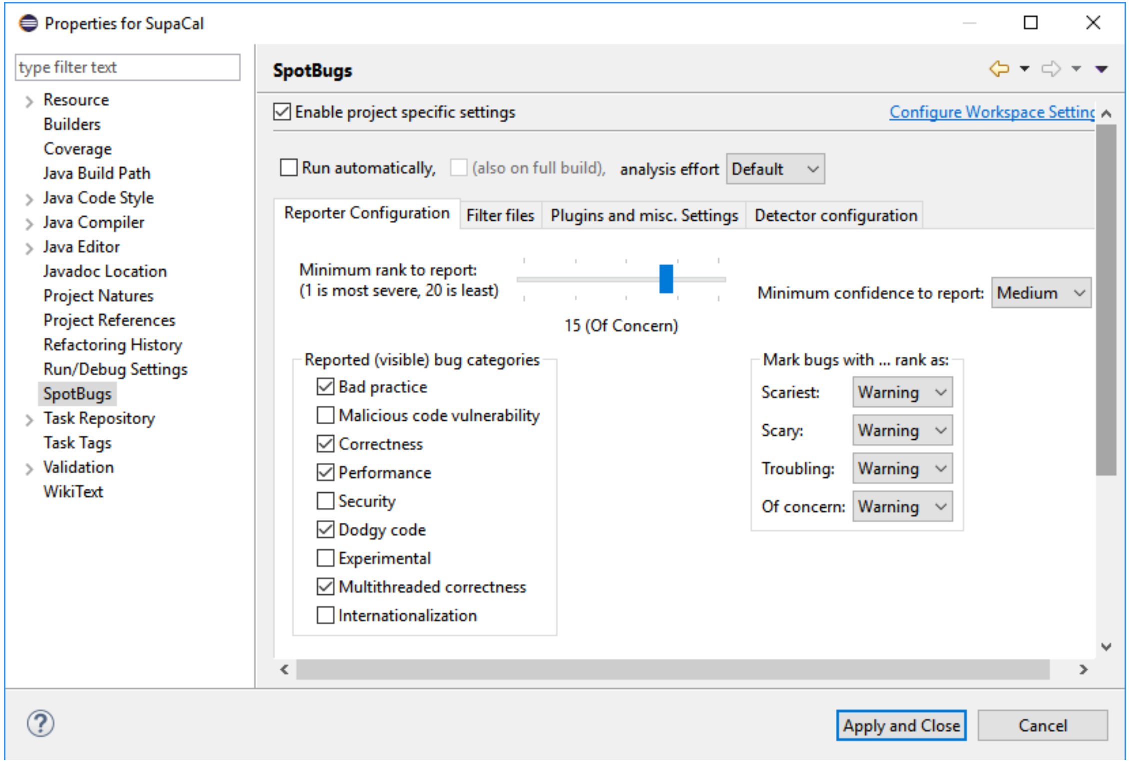
Windows → Show View → Other … → SpotBugs

Using SpotBugs – Bug Info



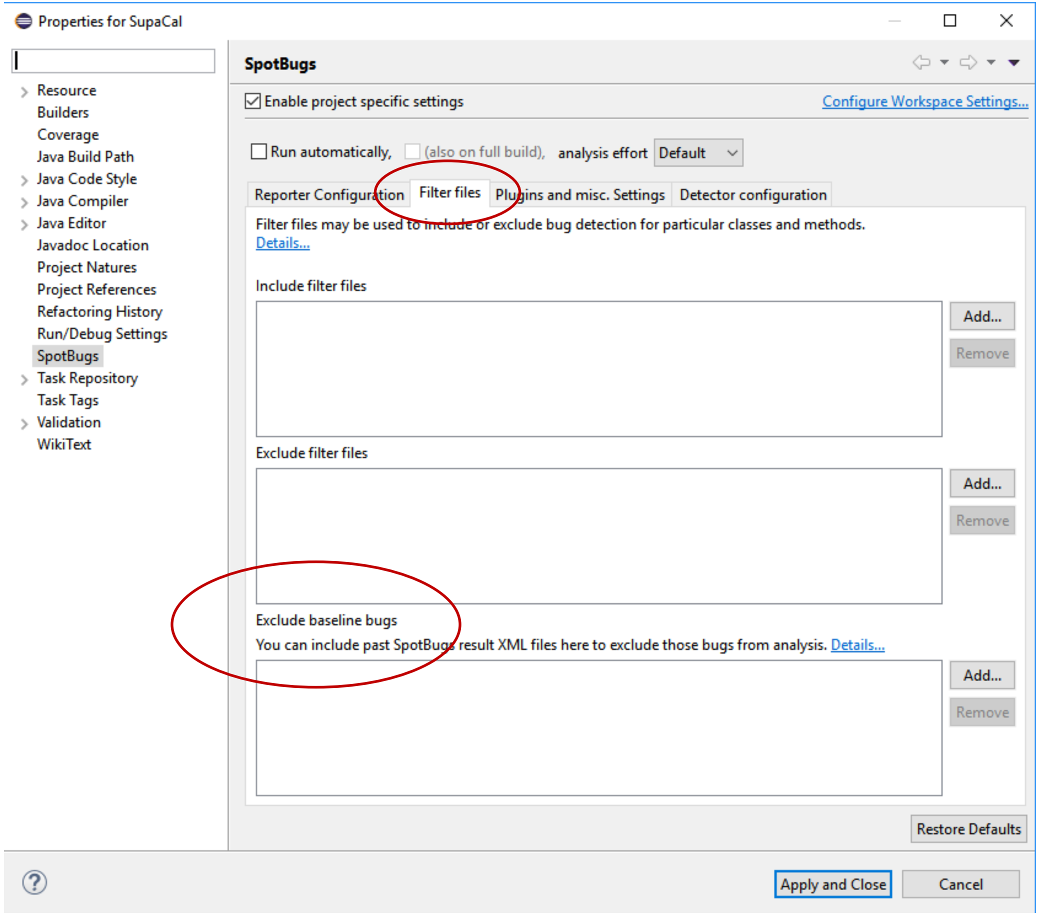
Using SpotBugs - Filtering Bugs

Select the Properties of a target project -> SpotBugs



Baseline bugs

- Show only new bug warnings
- Exclude all warnings arising from the previous release
- Commonly used by practitioners



Lessons Learnt with SpotBugs

- May not need to fix all warnings
  - Many warnings do not necessarily cause problems in program execution
  - They are often just inconsistencies
- Engineering effort is limited and zero sum
- Aim at getting the best return of our time in using SpotBugs

Software Faults, Errors & Failures

- Software Fault : A static defect in the software

Faults in software are design mistakes and will always exist

- Software Error : An incorrect internal state that is the manifestation of some fault
- Software Failure : External, incorrect behavior with respect to the requirements or other description of the expected behavior

Illustrative Example

```
public static int numZero (int[] x) {
// Effects: if x == null throw NullPointerException
// else return the number of occurrences of 0 in x
   int count = 0;
   for (int i = 1; i < x.length; i++) {
       if (x[i] == 0) {
            count++;
       }
   }
   return count;
}
```

Illustrative Example – Software Fault

```
public static int numZero (int[] x) {
// Effects: if x == null throw NullPointerException
// else return the number of occurrences of 0 in x
   int count = 0;
   for (int i = 1; i < x.length; i++) {
       if (x[i] == 0) {
            count++;
       }
   }
   return count;
}
```

Illustrative Example – Software Error

```
public static int numZero (int[] x) {
// Effects: if x == null throw NullPointerException
// else return the number of occurrences of 0 in x
   int count = 0;
   for (int i = 1; i < x.length; i++) {
      if (x[i] == 0) {
         count++;
      }
   }
   return count;
}
```

numZero([2, 7, 0])

Any failure occur?
Any error occur?

Illustrative Example – Software Error

```
public static int numZero (int[] x) {
// Effects: if x == null throw NullPointerException
// else return the number of occurrences of 0 in x
   int count = 0;
   for (int i = 1; i < x.length; i++) {
       if (x[i] == 0) {
             count++;
       }
   }
}
```

numZero([2, 7, 0])

Any failure occur?

Any error occur?

State error occurs at the first iteration where the state is (x=[2,7,0], count=0, i=1, PC=if). The correct state should be (x=[2,7,0], count=0, i=0, PC=if).

Illustrative Example – Software Error

```
public static int numZero (int[] x) {
// Effects: if x == null throw NullPointerException
// else return the number of occurrences of 0 in x
   int count = 0;
   for (int i = 1; i < x.length; i++) {
       if (x[i] == 0) {
            count++;
       }
   }
   return count;
}
```

numZero([0, 7, 2])

Any failure occur?
Any error occur?

Can you think of a program example and a test case that triggers a fault but causes no error and failure?

```
int square (int x) {
  int result = x + x;
  return result;
}
```

Test case: square(2)

Testing & Debugging

- Testing : Finding inputs that cause the software to fail
- Debugging : The process of finding a fault given a failure

Fault & Failure Model

Three conditions necessary for a failure to occur

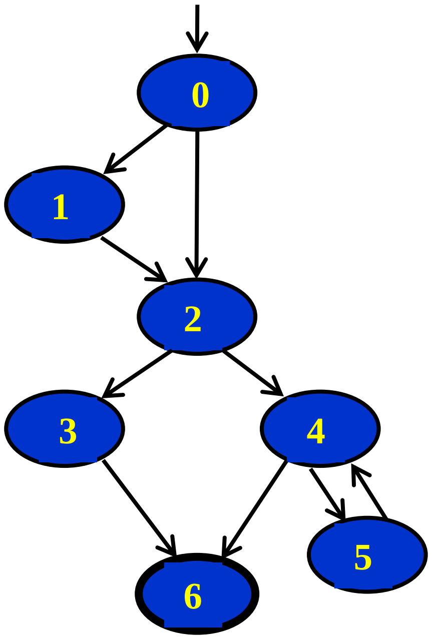
1. Reachability : The location or locations in the program that contain the fault must be reached
2. Infection : The state of the program must be incorrect
3. Propagation : The infected state must propagate to cause some output of the program to be incorrect

Fault & Failure Model

Three conditions necessary for a failure to occur

1. Reachability : The location or locations in the program that contain the fault must be reached
2. Infection : The state of the program must be incorrect
3. Propagation : The infected state must propagate to cause some output of the program to be incorrect

Structural Coverage Example



Node Coverage

TR = { 0, 1, 2, 3, 4, 5, 6 }

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 5, 4, 6]

Edge Coverage

TR = { (0,1), (0,2), (1,2), (2,3), (2,4), (3,6), (4,5), (4,6), (5,4) }

Test Paths: [0, 1, 2, 3, 6] [0, 2, 4, 5, 4, 6]

Edge-Pair Coverage

TR = { [0,1,2], [0,2,3], [0,2,4], [1,2,3], [1,2,4], [2,3,6], [2,4,5], [2,4,6], [4,5,4], [5,4,5], [5,4,6] }

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 6] [0, 2, 3, 6] [0, 2, 4, 5, 4, 5, 4, 6]

Complete Path Coverage

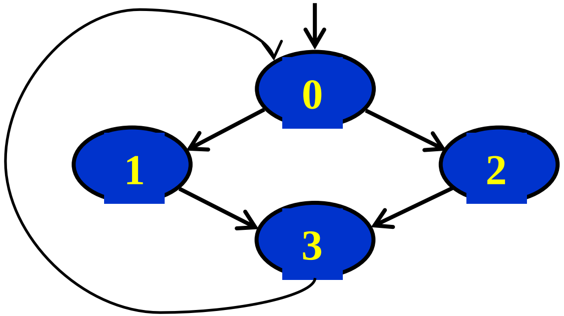
Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 6] [0, 1, 2, 4, 5, 4, 6] [0, 1, 2, 4, 5, 4, 5, 4, 6] [0, 1, 2, 4, 5, 4, 5, 4, 5, 4, 6] …

Simple Paths and Prime Paths

- Simple Path : *A path from node n_i to n_j is simple if no node appears more than once, except possibly the first and last nodes are the same*
  - No internal loops
  - May include other subpaths
  - A loop is a simple path

Any paths can be created by composing simple paths!

- Prime Path : *A simple path that does not appear as a proper subpath of any other simple path*

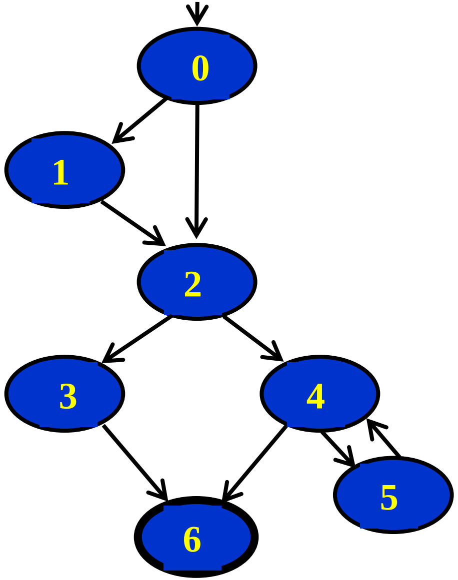


Simple Paths : [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1], [0, 1, 3], [0, 2, 3], [1, 3, 0], [2, 3, 0], [3, 0, 1], [3, 0, 2], [0, 1], [0, 2], [1, 3], [2, 3], [3, 0], [0], [1], [2], [3]

Prime Paths : [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1]

More Prime Path Example

- The following graph has 38 simple paths
- Only nine *prime paths*



<u>Prime Paths</u>

[0, 1, 2, 3, 6]
[0, 1, 2, 4, 5]
[0, 1, 2, 4, 6]
[0, 2, 3, 6]
[0, 2, 4, 5]
[0, 2, 4, 6]
[5, 4, 6]
[4, 5, 4]
[5, 4, 5]

Execute loop 0 times

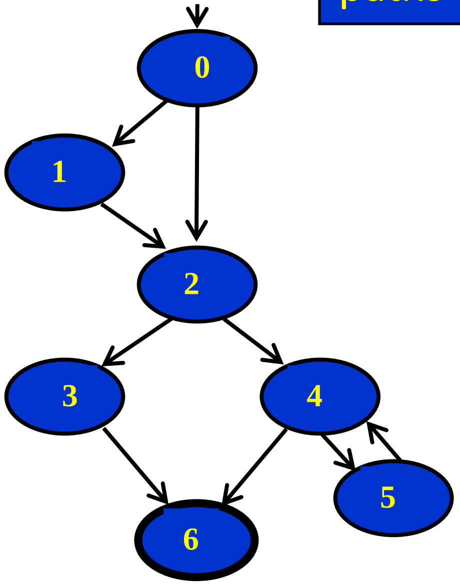
Execute loop at least once

Execute loop more than once

Finding Prime Paths

'!' means path terminates

'*' means path cycles



Simple paths

Len 0
[0]
[1]
[2]
[3]
[4]
[5]
[6] !

Len 1
[0, 1]
[0, 2]
[1, 2]
[2, 3]
[2, 4]
[3, 6] !
[4, 6] !
[4, 5]
[5, 4]

Len 2
[0, 1, 2]
[0, 2, 3]
[0, 2, 4]
[1, 2, 3]
[1, 2, 4]
[2, 3, 6] !
[2, 4, 6] !
[2, 4, 5] !
[4, 5, 4] *
[5, 4, 6] !
[5, 4, 5] *

Len 3
[0, 1, 2, 3]
[0, 1, 2, 4]
[0, 2, 3, 6] !
[0, 2, 4, 6] !
[0, 2, 4, 5] !
[1, 2, 3, 6] !
[1, 2, 4, 5] !
[1, 2, 4, 6] !

Len 4
[0, 1, 2, 3, 6] !
[0, 1, 2, 4, 6] !
[0, 1, 2, 4, 5] !

Prime Paths

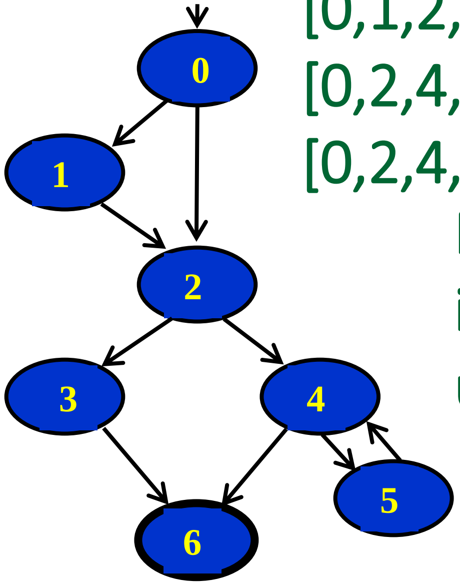
Prime Path Coverage

- A simple, elegant and finite criterion that requires loops to be executed as well as skipped

Prime Path Coverage (PPC) : TR contains each prime path in G.

- **Will tour all paths of length 0, 1, …**
- **That is, it subsumes node, edge, and edge-pair coverage**

Test Set Construction for Prime Path Coverage



T = {[0,1,2,3,6], [0,1,2,4,6], [0,1,2,4,5,4,6], [0,2,3,6], [0,2,4,6], [0,2,4,5,4,6], [0,2,4,5,4,5,4,6]}

Note that test set is not necessarily unique.

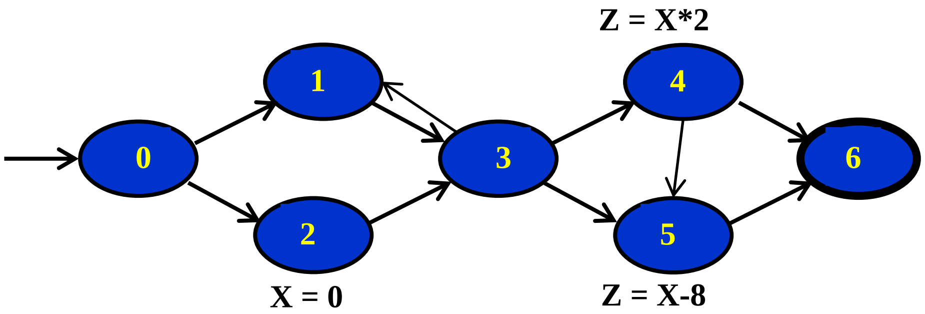
Len 2
[0, 1, 2]
[0, 2, 3]
[0, 2, 4]
[1, 2, 3]
[1, 2, 4]
[2, 3, 6] !
[2, 4, 6] !
[2, 4, 5] !
[4, 5, 4] *
[5, 4, 6] !
[5, 4, 5] *

Len 3
[0, 1, 2, 3]
[0, 1, 2, 4]
[0, 2, 3, 6] !
[0, 2, 4, 6] !
[0, 2, 4, 5] !
[1, 2, 3, 6] !
[1, 2, 4, 5] !
[1, 2, 4, 6] !

Len 4
[0, 1, 2, 3, 6] !
[0, 1, 2, 4, 6] !
[0, 1, 2, 4, 5] !

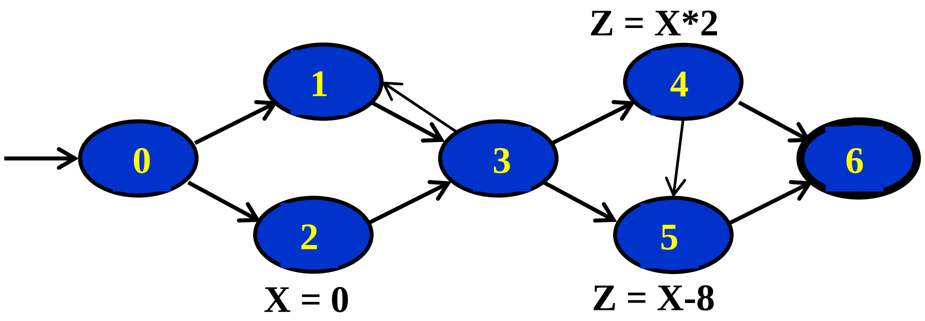
Prime Paths

Data Flow Testing Exercise



All-defs for *X*

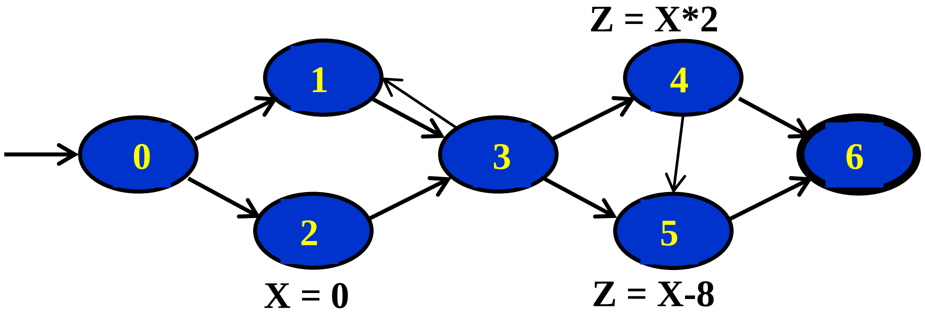
Data Flow Testing Exercise



All-defs for *X*
[2, 3, 4]
or
[2, 3, 5]
or
[2, 3, 4, 5]

All-uses for *X*

Data Flow Testing Exercise



All-defs for *X*

[2, 3, 4]

or

[2, 3, 5]

or

[2, 3, 4, 5]

All-uses for *X*

[2, 3, 4]

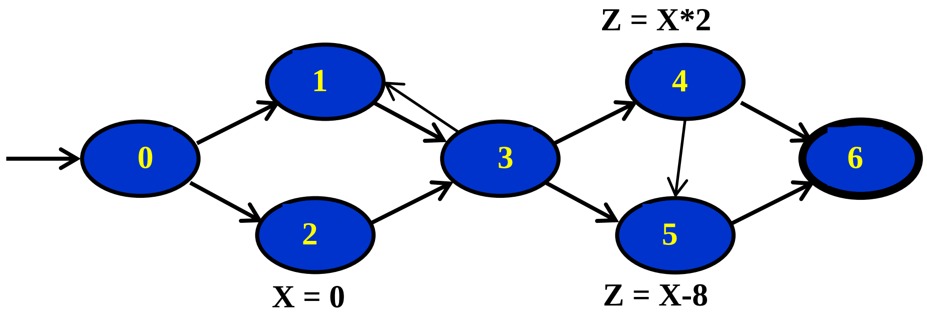
[2, 3, 5]

or

[2, 3, 4, 5]

All-du-paths for *X*

Data Flow Testing Exercise



All-defs for *X*

[2, 3, 4]

or

[2, 3, 5]

or

[2, 3, 4, 5]

All-uses for *X*

[2, 3, 4]

[2, 3, 5]

or

[2, 3, 4, 5]

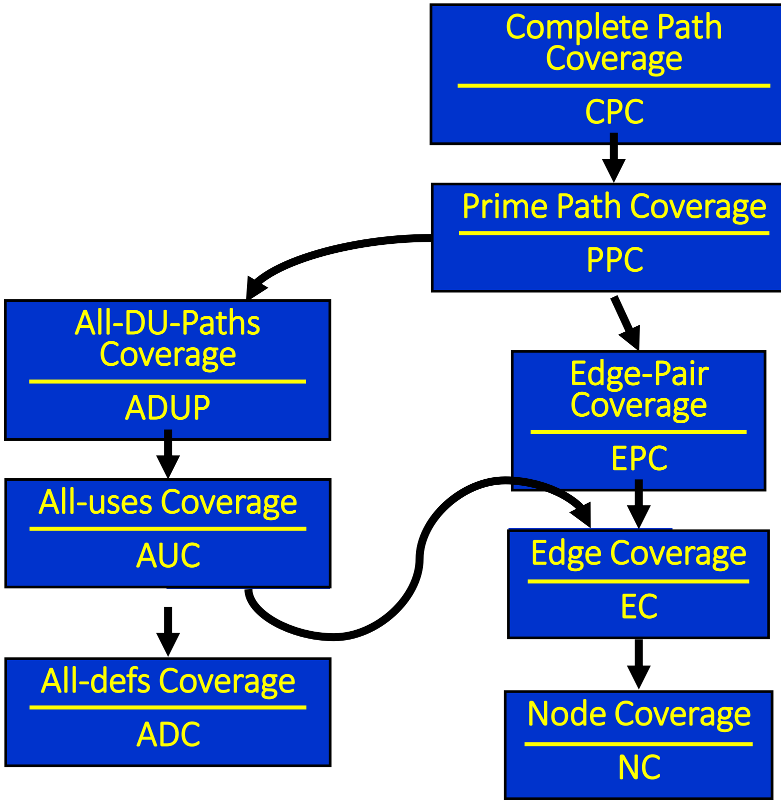
All-du-paths for *X*

~~[2, 3, 4]~~

[2, 3, 5]

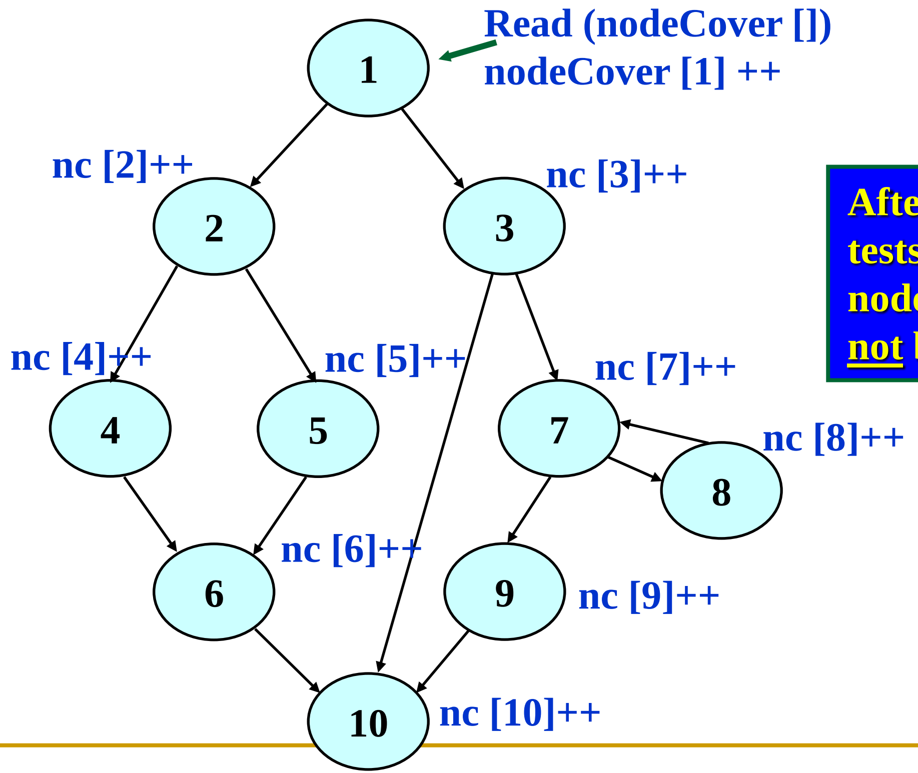
[2, 3, 4, 5]

Graph Coverage Criteria Subsumption



Statement Coverage Example

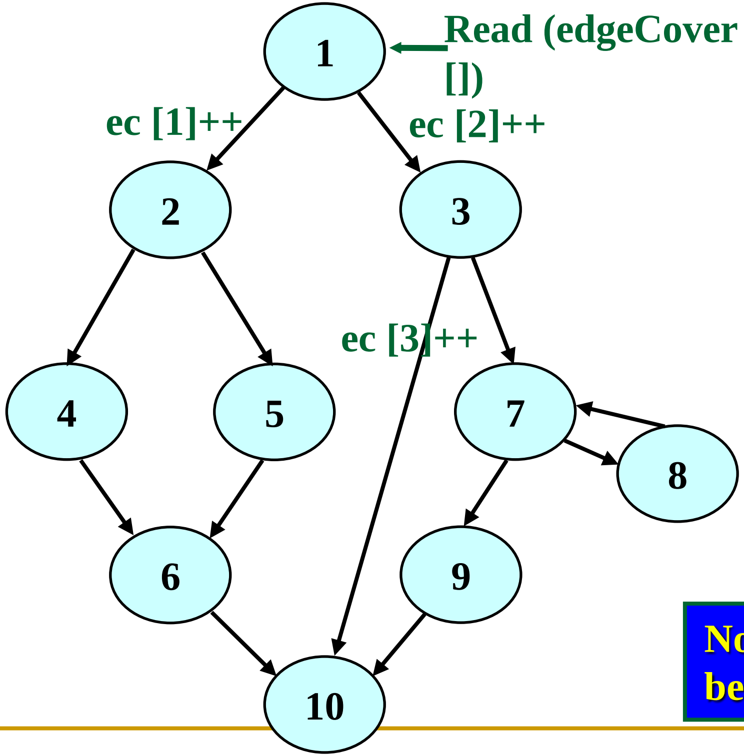
int nodeCover[] = {0,0,0,0,0,0,0,0,0,0}



After running a sequence of tests, any node for which nodeCover[node]==0 has <u>not</u> been covered.

Edge Coverage Instrumentation

int edgeCover[] = {0,0,0,0,0,0,0,0,0,0,0,0,0}

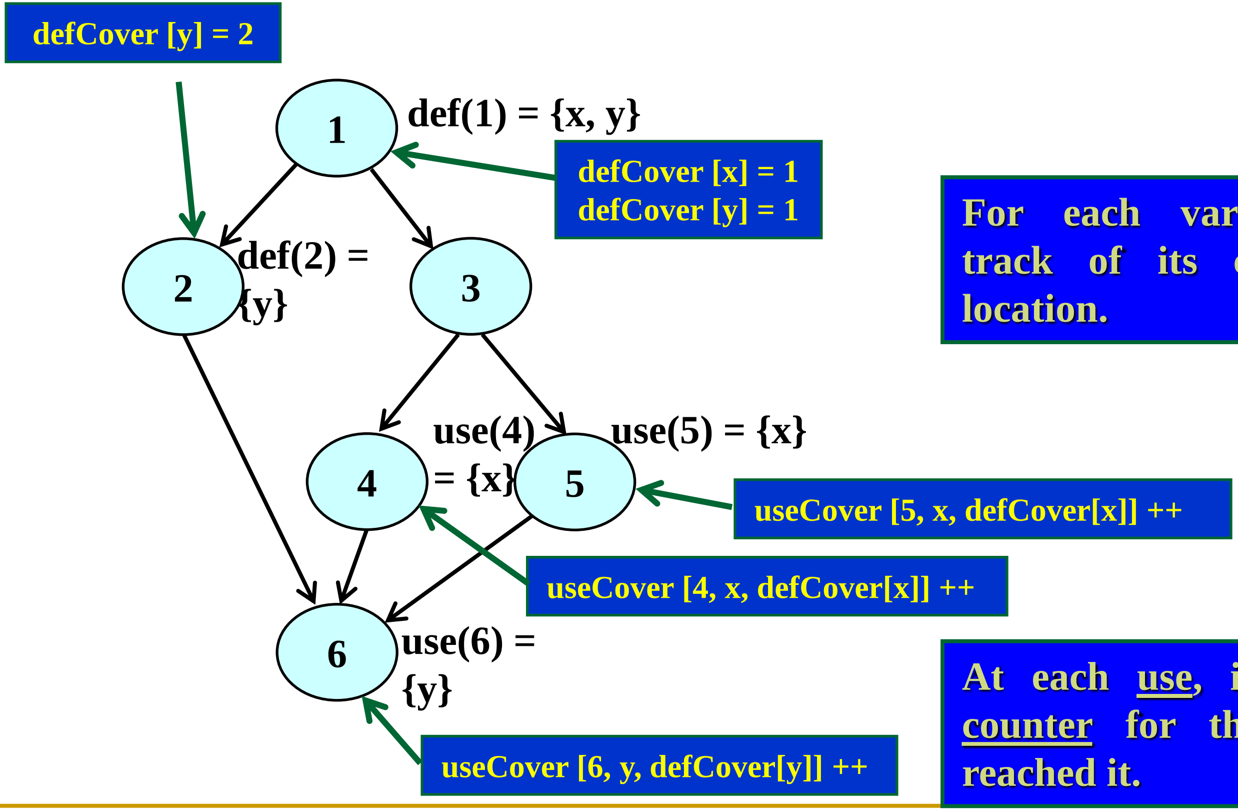


For each edge e, put edgeCover[e]++ on the edge.

If edgeCover[e] == 0, e has <u>not</u> been covered.

Note that the arrays could be boolean

All-Uses Coverage Instrumentation

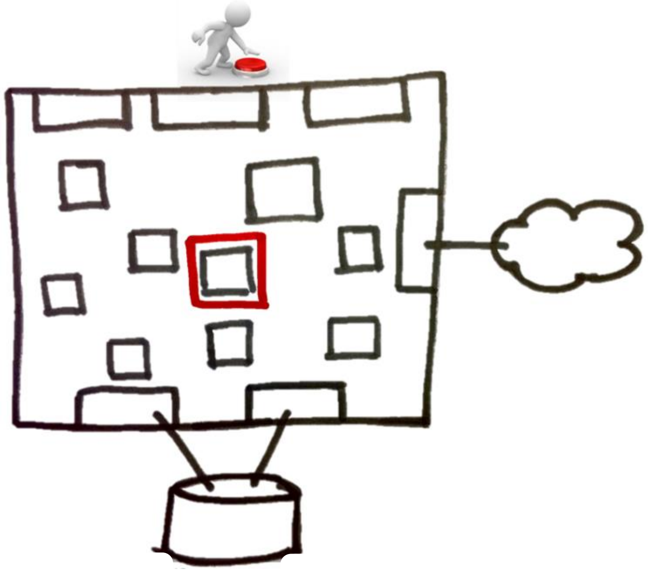


For each variable, keep track of its current <u>def</u> location.

At each <u>use</u>, increment a <u>counter</u> for the def that reached it.

What is Unit Testing?

- To isolate each part of program and show that individual parts are correct.
- JUnit is a de facto framework used by Java practitioners to test their code.
- It is a built-in Eclipse plug-in.
- Its latest version is 5.



Putting it in Eclipse

```
public class RectangleTest {

 Rectangle r = new Rectangle(2, 4);
 final int height = 2, width = 4;
 final double tolerance = 0.000001;

@Test
public void testGetArea() {
  double expectedArea = width * height;
  double actualArea = r.getArea();
  assertEquals(expectedArea, actualArea,
      tolerance, "getArea fails");
}

@Test
public void testResize() {
  r.resize(0.25, 2);
  assertEquals(0.25*height, r.getHeight(), tolerance,
      "resize - height fails");
  assertEquals(2*width, r.getWidth(), tolerance,
      "resize - width fails");
}
… // other tests
}
```

BeforeEach and AfterEach test case

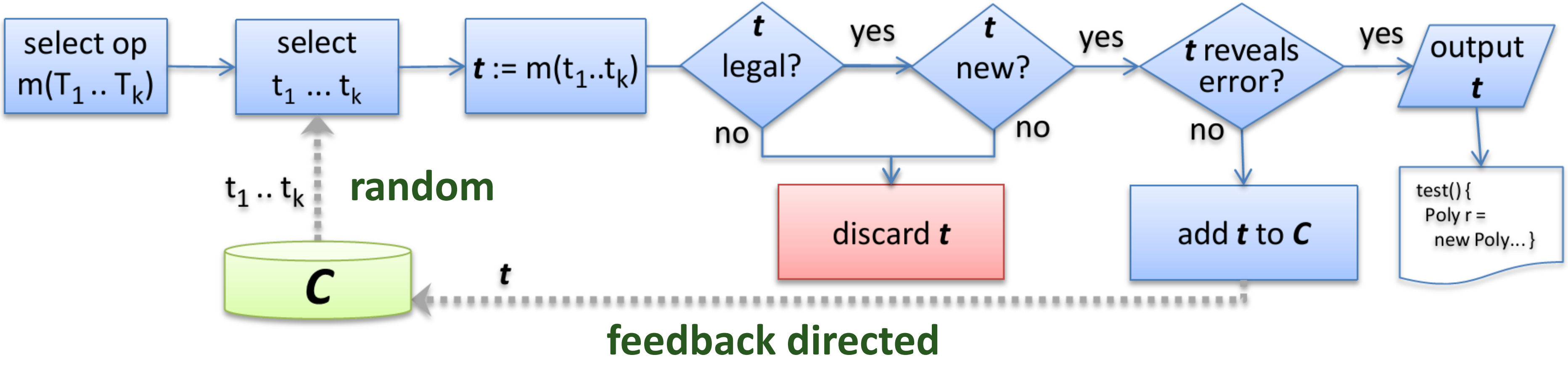
```
@BeforeEach
public void setUp() throws Exception {
        // create objects and common variables for your tests here
        // setUp() is called before EACH test case is run
}

@AfterEach
public void tearDown() throws Exception {
        // put code here to reset or release objects, e.g., p1=null;
        // to garbage collect unused objects or resources
        // tearDown() is called after EACH test case is run
}
```

Grand Challenge

- No need to write tests
- No need to locate faults
- No need to fix faults

Directed Random Test Generator



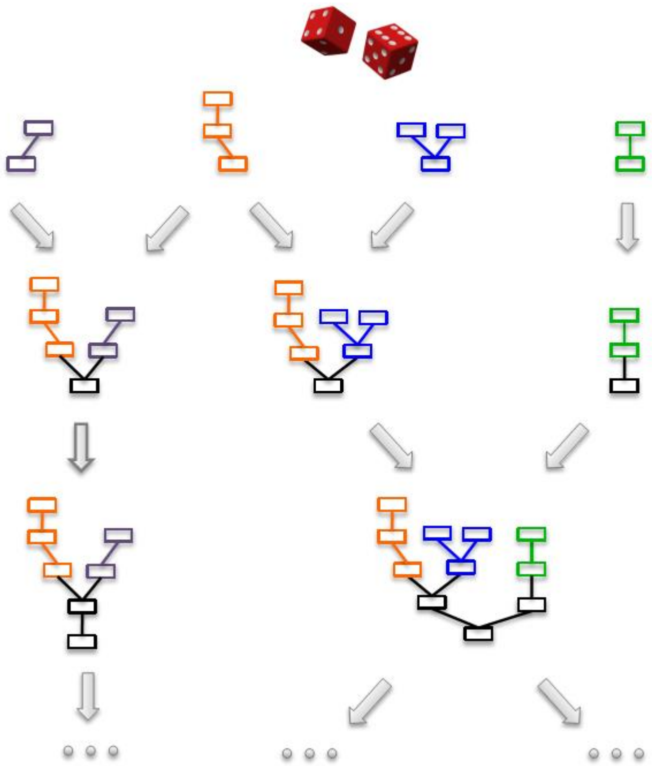
- *C* is a repository containing possible terms used by the Test Generator.

Directed Random Test Generator

- Generate simple inputs randomly from repository *C*.

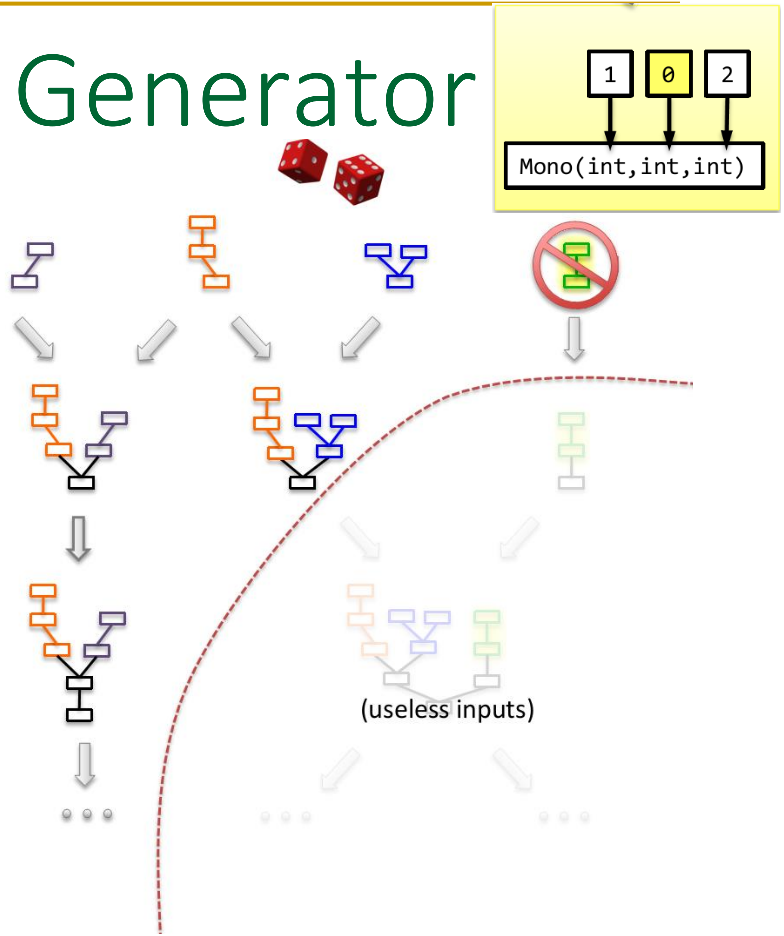
Directed Random Test Generator

- Generate simple inputs randomly from repository *C*.
- Build new inputs incrementally from previous ones.



Directed Random Test Generator

- Executes inputs
- Discards the ones useless for extension
  - illegal, redundant
- Prune input space



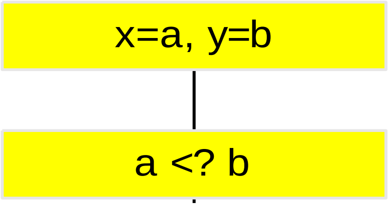
Symbolic Testing (a.k.a. Symbolic Execution)

x=a, y=b

```
☞ foo (int x, int y) {
    if (x>y) {
      x = x + y;
      y = x - y;
      x = x - y;
      if (x - y > 0) {
           assert (false); // bug
      }
    }
  }
```

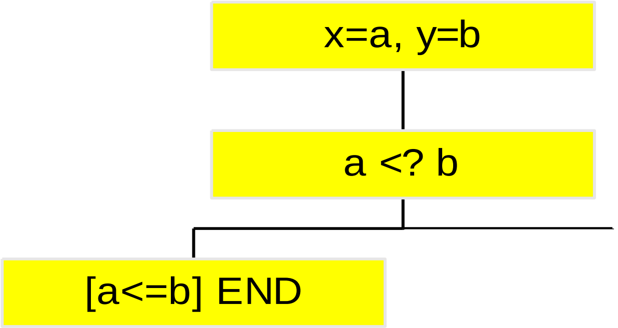
Symbolic Testing (a.k.a. Symbolic Execution)

```
foo (int x, int y) {
  if (x>y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0) {
        assert (false); // bug
    }
  }
}
```



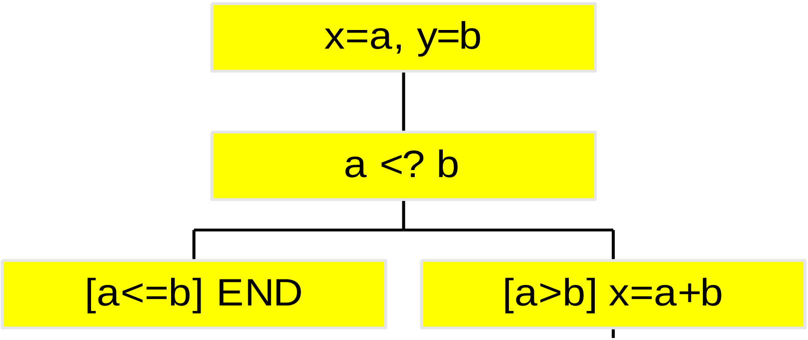
Symbolic Testing (a.k.a. Symbolic Execution)

```
foo (int x, int y) {
  if (x>y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0) {
        assert (false); // bug
    }
  }
}
```



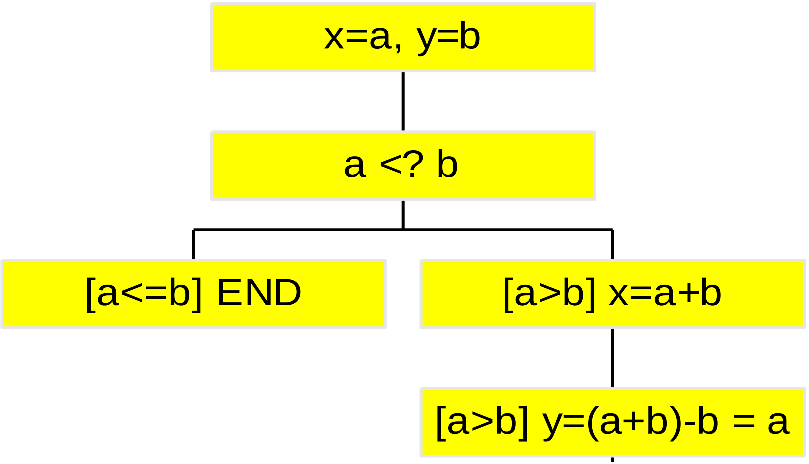
Symbolic Testing (a.k.a. Symbolic Execution)

```
foo (int x, int y) {
  if (x>y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0) {
        assert (false); // bug
    }
  }
}
```



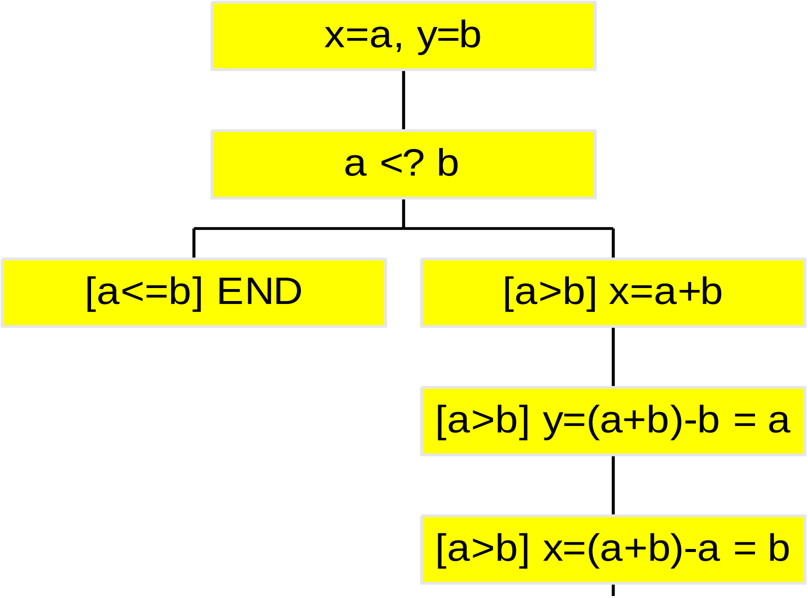
Symbolic Testing (a.k.a. Symbolic Execution)

```
foo (int x, int y) {
  if (x>y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0) {
        assert (false); // bug
    }
  }
}
```



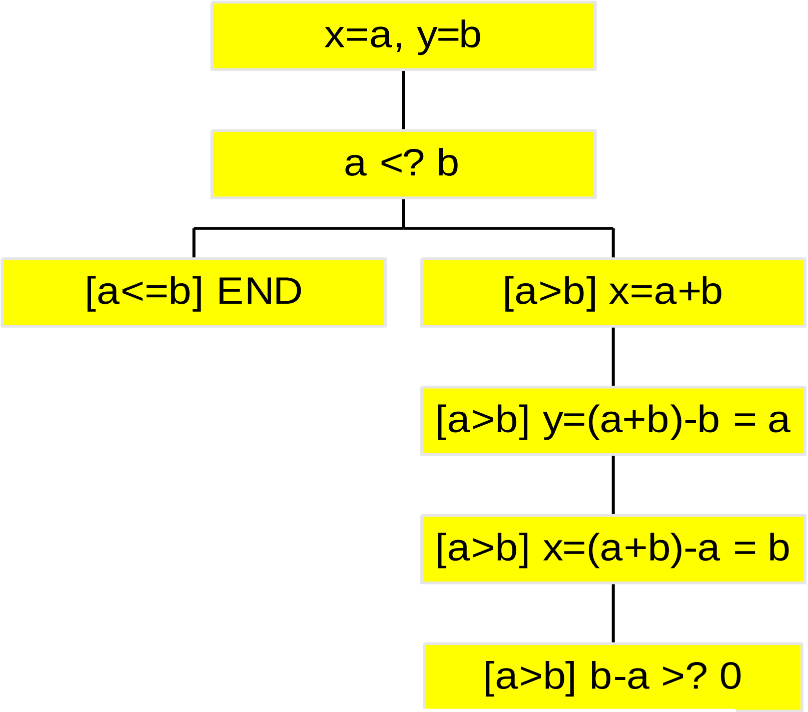
Symbolic Testing (a.k.a. Symbolic Execution)

```
foo (int x, int y) {
  if (x>y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0) {
        assert (false); // bug
    }
  }
}
```



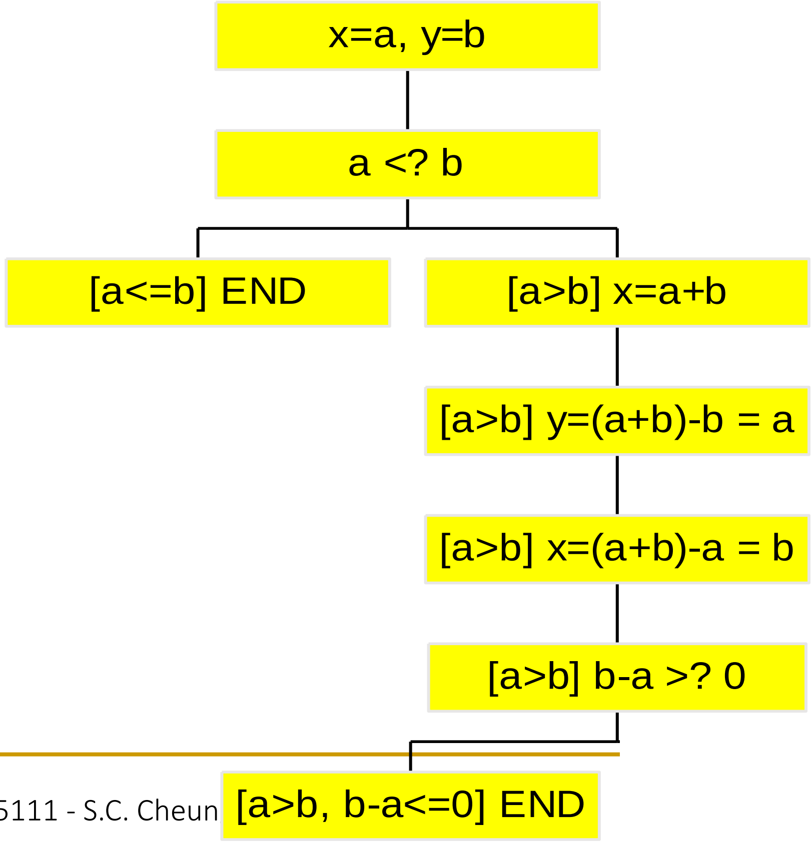
Symbolic Testing (a.k.a. Symbolic Execution)

```
foo (int x, int y) {
  if (x>y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0) {
        assert (false); // bug
    }
  }
}
```



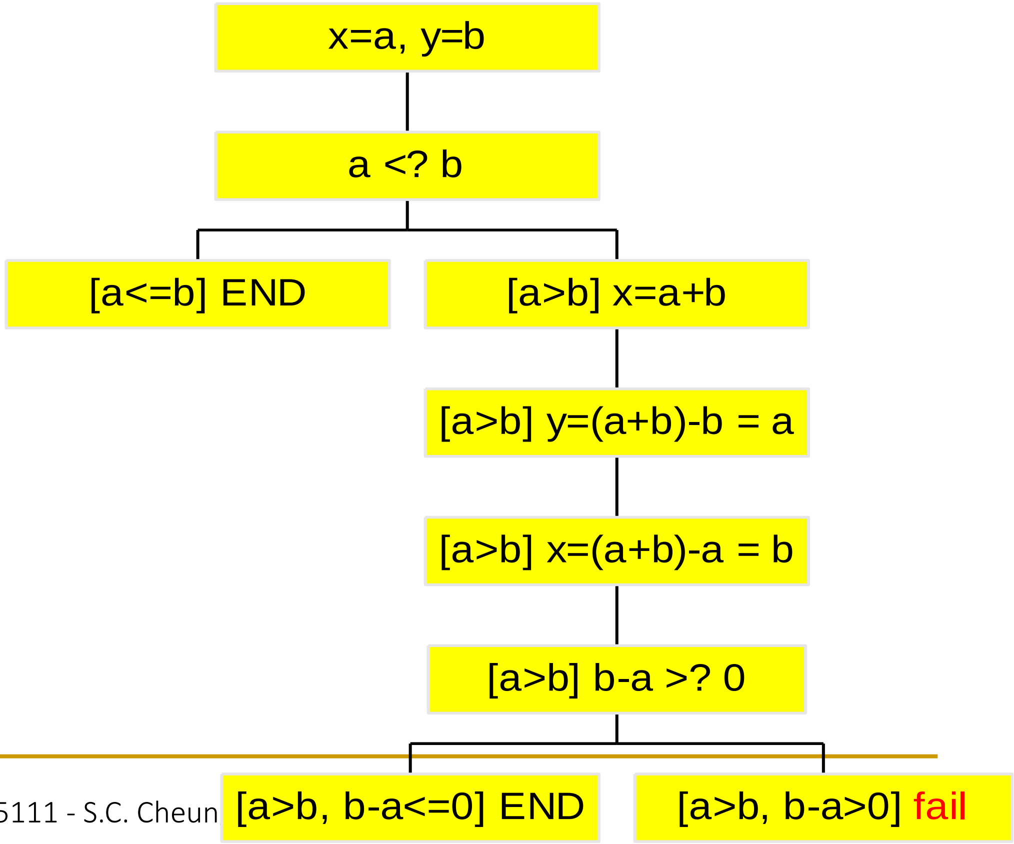
Symbolic Testing (a.k.a. Symbolic Execution)

```
foo (int x, int y) {
  if (x>y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0) {
        assert (false); // bug
    }
  }
}
```



Symbolic Testing (a.k.a. Symbolic Execution)

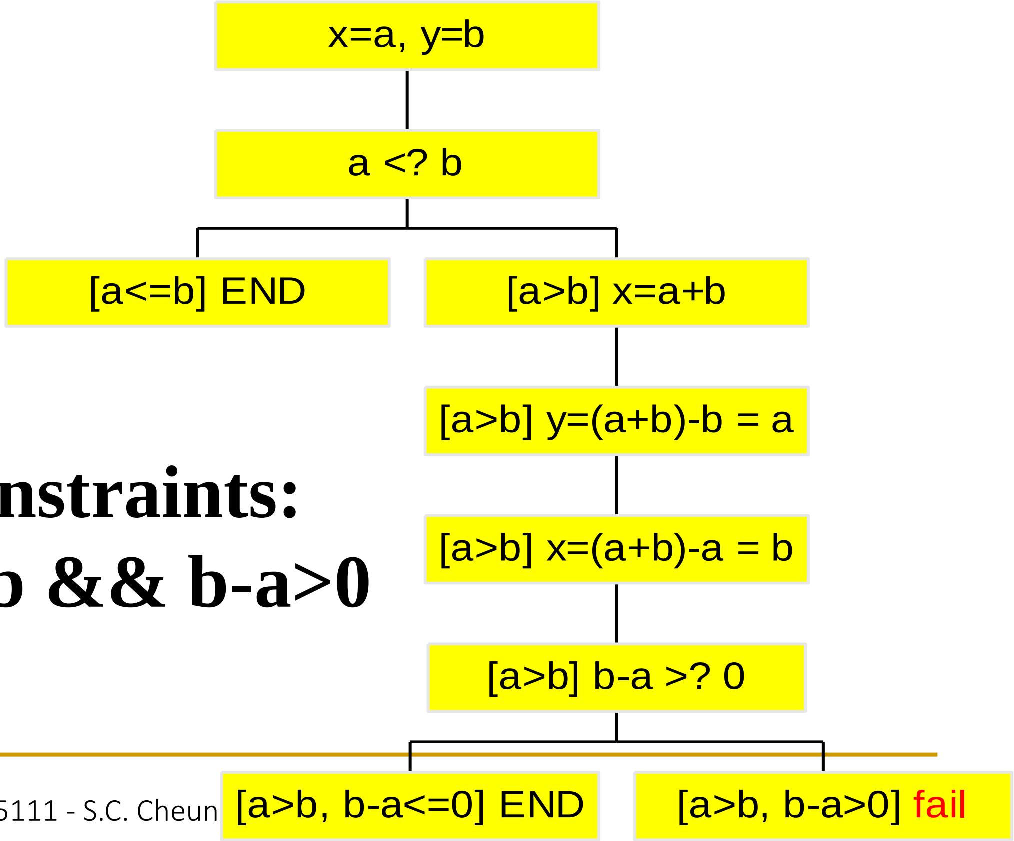
```
foo (int x, int y) {
  if (x>y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0) {
        assert (false); // bug
    }
  }
}
```



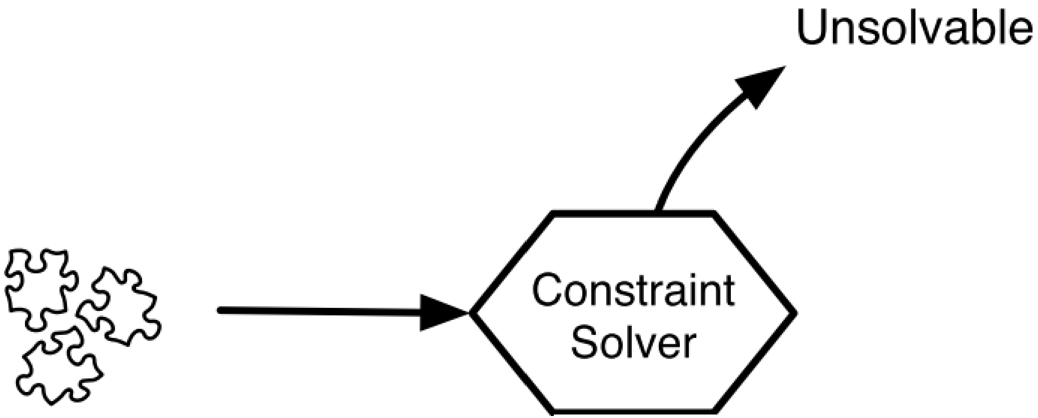
Symbolic Testing (a.k.a. Symbolic Execution)

```
foo (int x, int y) {
  if (x>y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0) {
        assert (false); // bug
    }
  }
}
```

Constraints:
a>b && b-a>0



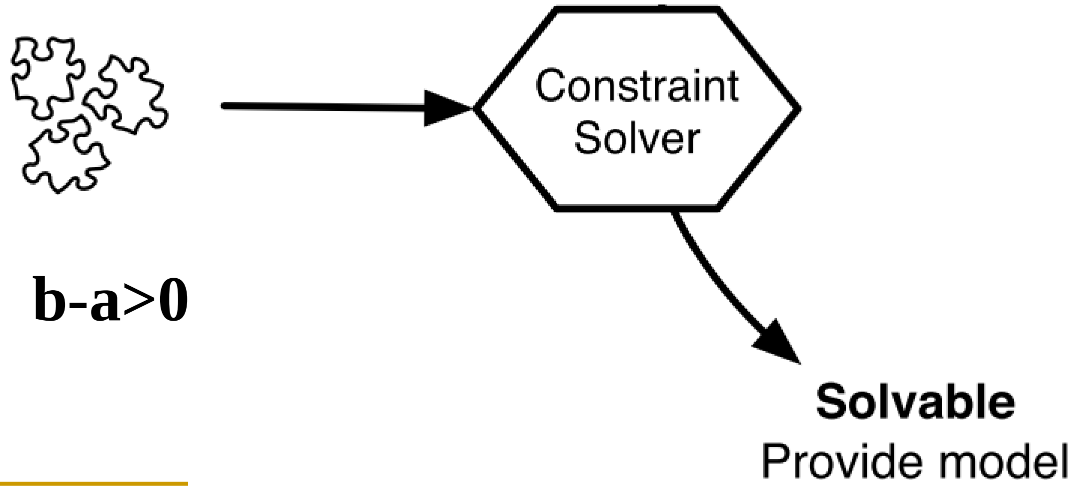
Using a Linear Constraint Solver



a>b && b-a>0

Efficient linear constraint solvers are available

Constraint Solving with What-if Analysis



b-a>0

a = 0, b = 1

Concolic = Concrete + Symbolic

```
int foo(int x, int y) {
  int z = square(x);
  if (z > 100 && y > 20)
    assert(false);
  return y*z;
}
```

x = 15, y = 10

z = 225

225 > 100 && 10 > 20

return 2250

Execute program concretely

Test: foo(15, 10)

Concolic = Concrete + Symbolic

```
int foo(int x, int y) {
  int z = square(x);
  if (z > 100 && y > 20)
    assert(false);
  return y*z;
}
```

x = 15, y = 10

z = 225

225 > 100 && 10 > 20

return 2250

x = X, y = Y

z = square(X)

?(square(X) > 100 && Y > 20)

[!(square(X) > 100 && Y > 20)]
return Y*square(X)

Execute program concretely
Collect symbolic path condition

Test: foo(15, 10)

Concolic Testing

```
int foo(int x, int y) {
  int z = square(x);
  if (z > 100 && y > 20)
    assert(false);
  return y*z;
}
```

Test: foo(15, 10)

x = 15, y = 10

z = 225

225 > 100 && 10 > 20

return 2250

x = X, y = Y

z = square(X)

?(square(X) > 100 && Y > 20)

[square(X) > 100, Y > 20]

[!(square(X) > 100 && Y > 20)]
return Y*square(X)

Execute program concretely

Collect symbolic path condition

Negate a constraint on the path condition and solve it

Concolic Testing

```
int foo(int x, int y) {
  int z = square(x);
  if (z > 100 && y > 20)
    assert(false);
  return y*z;
}
```

Test: foo(15, 10)

x = 15, y = 10

z = 225

225 > 100 && 10 > 20

return 2250

x = X, y = Y

z = square(X)

?(square(X) > 100 && Y > 20)

[225 > 100, Y > 20]

assert(false)

[!(square(X) > 100 && Y > 20)]
return Y*square(X)

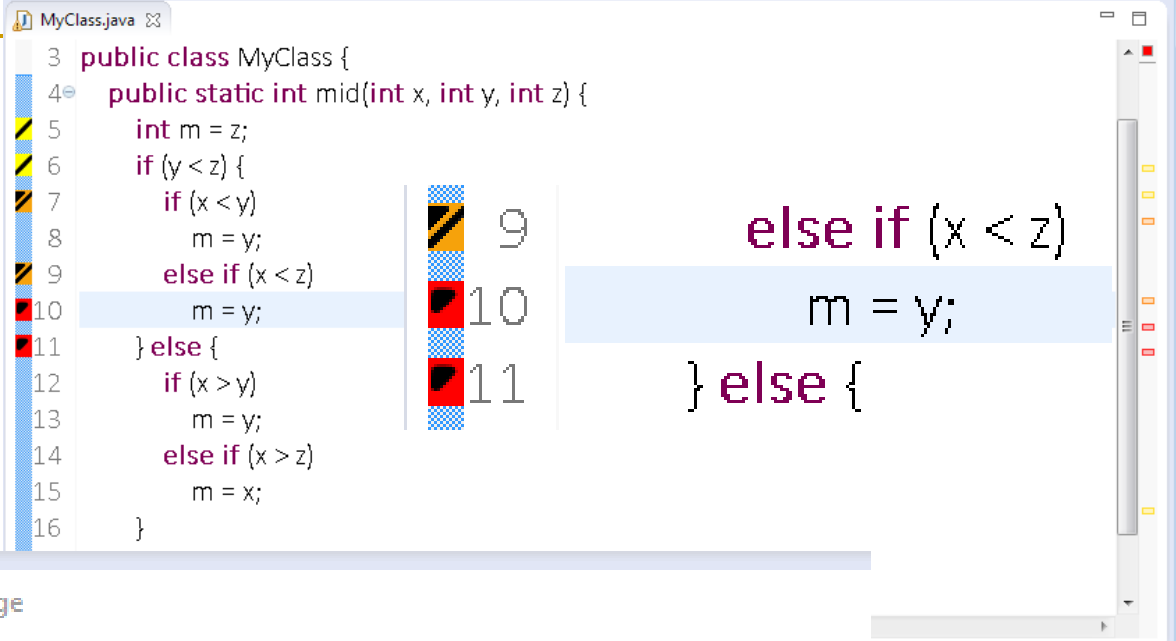
Fault Localization

mid( ) {	Runs					
int x, y, z, m;	1	2	3	4	5	6
read("Enter 3 numbers:", x, y, z);						
m = z;						
if (y < z) {						
if (x < y)						
m = y;						
else if (x < z)						
m = y;						
} else {						
if (x > y)						
m = y;						
else if (x > z)						
m = x;						
}						
print("Middle number is:", m);						
}	✓	✓	✓	✓	✗	✓

- Runs
  - 1: (1,1,2)
  - 2: (0,1,2)
  - 3: (2,1,0)
  - 4: (0,2,1)
  - 5: (1,0,2)
  - 6: (2,0,1)

GZoltar

- Likely faults are colored in red.
- Less likely faults are colored in orange.



Problems | Javadoc | Declaration | Coverage

0 errors, 7 warnings, 0 others

Description	Resource	Path	Location	Type
Warnings (7 items)				
Fault likelihood: 0.40824828	MyClass.java	/FaultLocalization/...	line 5	GZoltar Warni...
Fault likelihood: 0.40824828	MyClass.java	/FaultLocalization/...	line 6	GZoltar Warni...
Fault likelihood: 0.40824828	MyClass.java	/FaultLocalization/...	line 17	GZoltar Warni...
Fault likelihood: 0.5	MyClass.java	/FaultLocalization/...	line 7	GZoltar Orange
Fault likelihood: 0.57735026	MyClass.java	/FaultLocalization/...	line 9	GZoltar Orange
Fault likelihood: 0.70710677	MyClass.java	/FaultLocalization/...	line 10	GZoltar Error
Fault likelihood: 0.70710677	MyClass.java	/FaultLocalization/...	line 11	GZoltar Error

Fault Localization

mid( ) {	Runs					
int x, y, z, m;	1	2	3	4	5	6
read("Enter 3 numbers:", x, y, z);	●	●	●	●	●	●
m = z;	●	●	●	●	●	●
if (y < z) {	●	●	●	●	●	●
if (x < y)	●	●			●	●
m = y;		●				
else if (x < z)	●				●	●
m = y;	●				●	
} else {			●	●		
if (x > y)			●	●		
m = y;			●			
else if (x > z)				●		
m = x;						
}						
print("Middle number is:", m);	●	●	●	●	●	●
}	✓	✓	✓	✓	✗	✓

- Runs
  - 1: (1,1,2)
  - 2: (0,1,2)
  - 3: (2,1,0)
  - 4: (0,2,1)
  - 5: (1,0,2)
  - 6: (2,0,1)

Fault Localization

mid( ) {	Runs					
int x, y, z, m;	1	2	3	4	5	6
read(“Enter 3 numbers:”, x, y, z);	●	●	●	●	●	●
m = z;	●	●	●	●	●	●
if (y < z) {	●	●	●	●	●	●
if (x < y)	●	●			●	●
m = y;		●				
else if (x < z)	●				●	●
m = y; // *** BUG ***	●				●	
} else {			●	●		
if (x > y)			●	●		
m = y;			●			
else if (x > z)				●		
m = x;						
}						
print(“Middle number is:”, m);	●	●	●	●	●	●
}	✓	✓	✓	✓	✗	✓

- Runs
  - 1: (1,1,2)
  - 2: (0,1,2)
  - 3: (2,1,0)
  - 4: (0,2,1)
  - 5: (1,0,2)
  - 6: (2,0,1)

Fault Localization

mid( ) {			**Runs**			
int x, y, z, m;	1	2	3	4	5	6
read(“Enter 3 numbers:”, x, y, z);	●	●	●	●	●	●
m = z;	●	●	●	●	●	●
if (y < z) {	●	●	●	●	●	●
if (x < y)	●	●			●	●
m = y;		●				
else if (x < z)	●				●	●
m = y; // * BUG *****	●				●	
} else {			●	●		
if (x > y)			●	●		
m = y;			●			
else if (x > z)				●		
m = x;						
}						
print(“Middle number is:”, m);	●	●	●	●	●	●
}	✓	✓	✓	✓	✗	✓

- Premise
  - Bug participates more often in failing runs than successful runs
  - RIP model
    - Reachability
    - Infection
    - Propagation

Ranking function - Tarantula

- J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proc. of the 20th IEEE/ACM Conference on Automated Software Engineering*, pp. 273-282, Long Beach, California, USA, December, 2005

$$X/X+Y,\ X=(N_{EF}/N_F)\ \&\ Y=(N_{ES}/N_S)$$

N_{EF} : Number of failing runs executing the statement

N_{ES} : Number of successful runs executing the statement

N_F : Number of failing runs

N_S : Number of successful runs

Fault Localization

$X/X{+}Y$, $X{=}(N_{EF}/N_F)$ & $Y{=}(N_{ES}/N_S)$

$N_F : 1$, $N_S : 5$

mid( ) {	Runs						Tarantula
int x, y, z, m;	1	2	3	4	5	6	
read("Enter 3 numbers:", x, y, z);	●	●	●	●	●	●	0.5
m = z;	●	●	●	●	●	●	0.5
if (y < z) {	●	●	●	●	●	●	0.5
if (x < y)	●	●			●	●	0.625
m = y;		●					0.0
else if (x < z)	●				●	●	0.714
m = y; // *** BUG ***	●				●		**0.833**
} else {			●	●			0.0
if (x > y)			●	●			0.0
m = y;			●				0.0
else if (x > z)				●			0.0
m = x;							0.0
}							0.0
print("Middle number is:", m);	●	●	●	●	●	●	0.5
}	✓	✓	✓	✓	✗	✓	

Program-based Mutation Testing

Original Method

```
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;
Δ 1  minVal = B;
        if (B < A)
Δ 2  if (B > A)
Δ 3  if (B < minVal)
        {
             minVal = B;
Δ 4          Bomb ();
Δ 5          minVal = A;
Δ 6          minVal = failOnZ
        }
        return (minVal);
} // end Min
```

Replace one variable with another

Changes operator

Immediate runtime failure … if reached

Immediate runtime failure if B==0 else does nothing

6 mutants

Each represents a separate program

Syntax-Based Coverage Criteria

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

- **The RIP 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**
- **The RIP model leads to two variants of mutation coverage …**

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 <u>strongly kill</u> *m* if and only if the <u>output</u> 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 <u>weakly kill</u> *m* if and only if the <u>state</u> of the execution of *P* on *t* is different from the state of the execution of *m* immediately on *t* after *l*

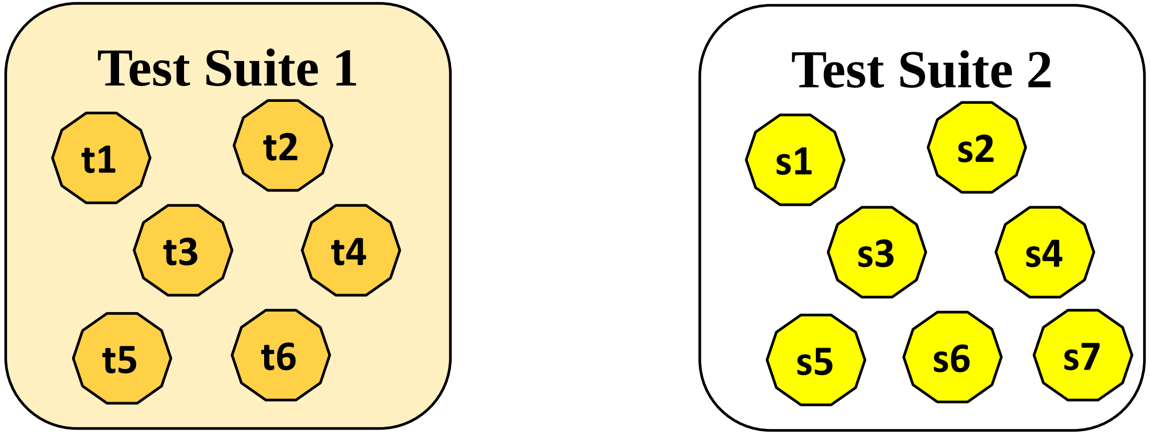
  - **<u>Weakly killing</u> satisfies reachability and infection, but not propagation**

Search-based Test Generation

- Given class under test (CUT), EvoSuite automatically generates a test suite using a genetic algorithm
  1. Create 2 initial test suites by adding method calls randomly and insert the test suites into current generation
  2. Select two test suites from current generation
  3. Create 2 new test suites by crossover (exchange test cases of the suites)
  4. Modify two test suites (test drivers) from step 3 with mutation operators (remove, insert, change operators)
  5. Insert the new two test suites from step 4 to next generation if coverage of the new two test suites are higher than that of parents
  6. Repeat 2~5 until next generation has sufficient number of test suites
  7. Repeat 2~6 until time limit is reached or all branches are covered
  8. Select a test suit with the highest branch coverage and insert assertions by observing differences in execution traces between the original CUT and a mutated CUT

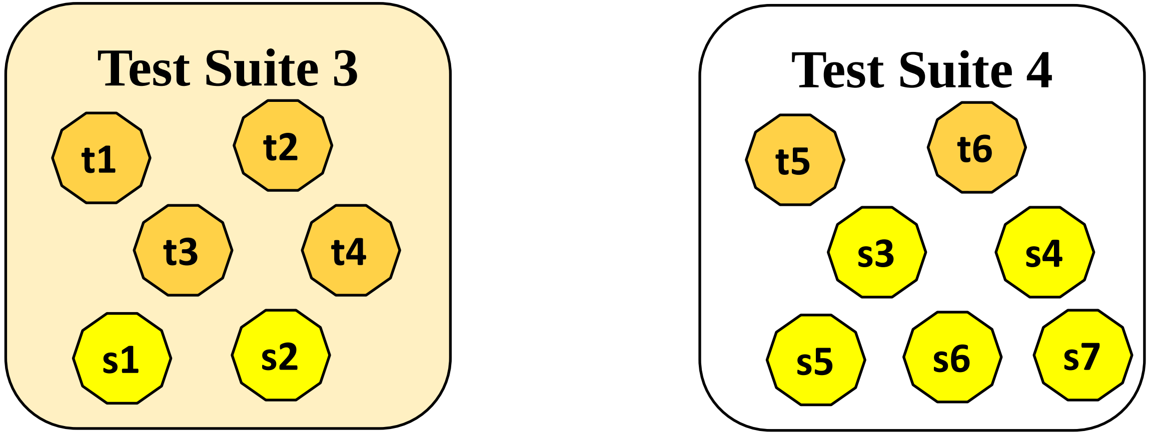
EvoSuite

1. Create 2 initial test suites by adding method calls randomly and insert the test suites into current generation

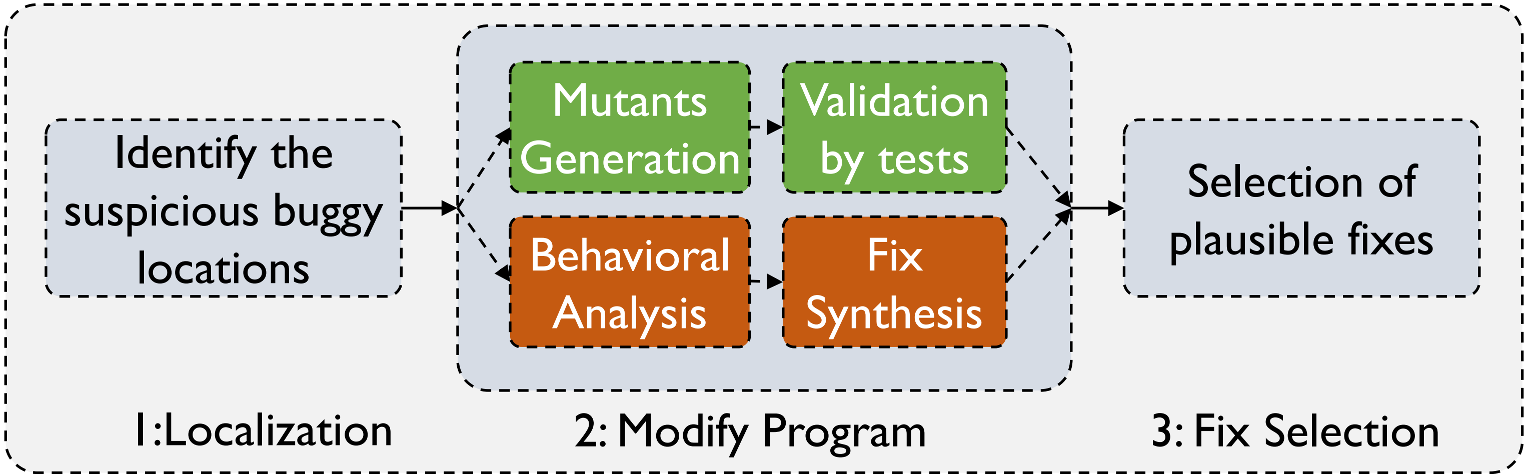


EvoSuite

2. Select two test suites from current generation
3. Create 2 new test suites by crossover (exchange test cases of the suites)



General Process

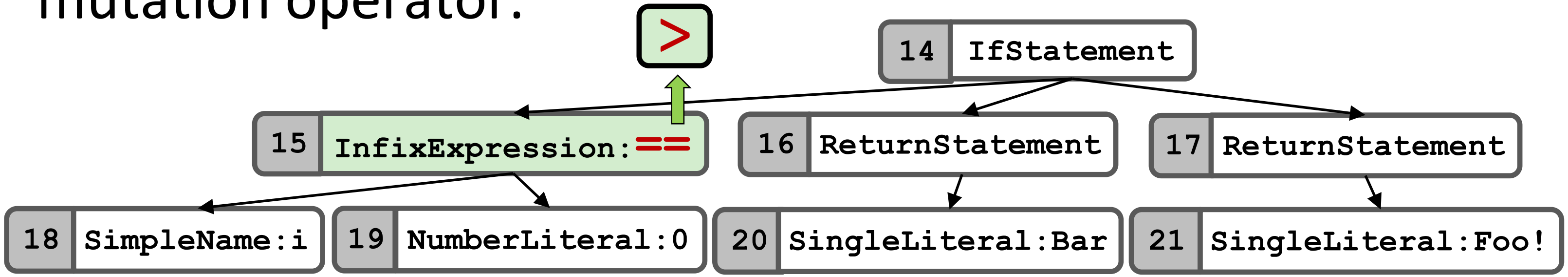


Mutation-driven (also known as generate-and-validate)

Semantics-driven

Mutation Operators

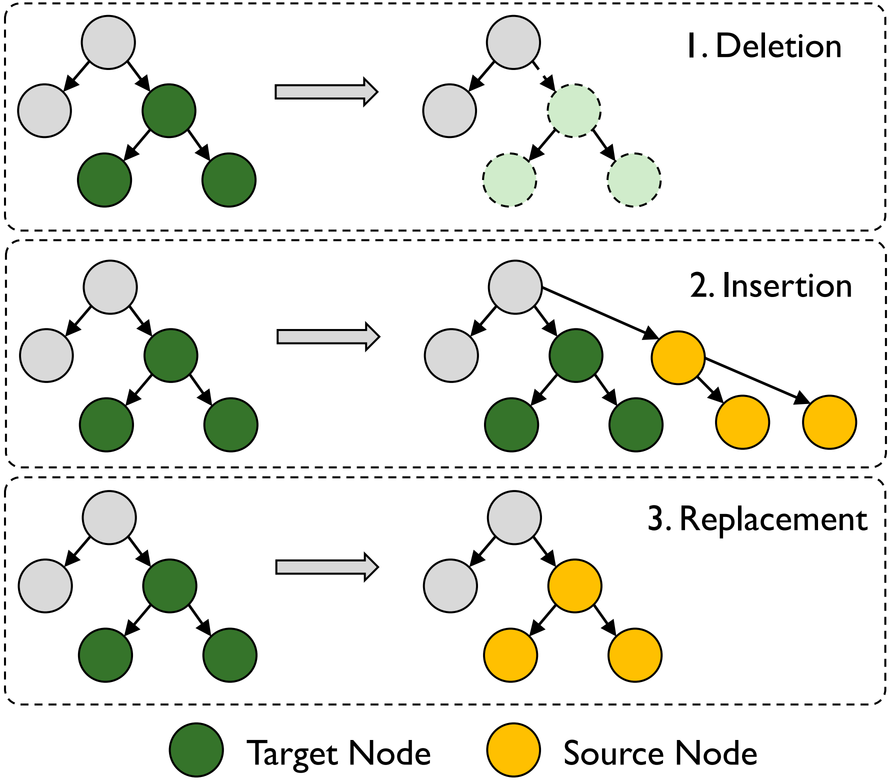
- A mutant is a new version of a program that is created by making a small syntactic change to the original program.
- An operation which creates a mutant of a program is called a mutation operator.




```java
public class Test {
  public String foo(int i) {
    if (i == 0) return "Bar";
    else return "Foo!";
  }
}
```

```java
public class Test {
  public String foo(int i) {
    if (i > 0) return "Bar";
    else return "Foo!";
  }
}
```

Mutation Operators



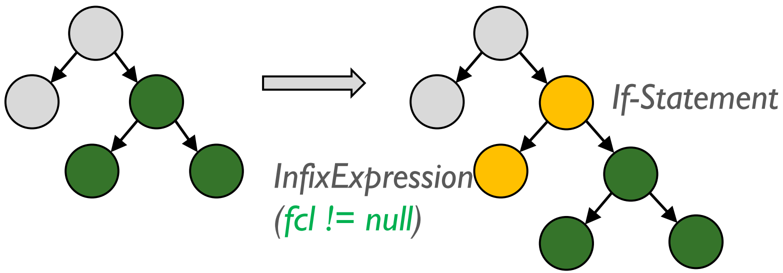
Redundancy Assumption [Qi et al., ICSE14, Tao et al., FSE14]

The fix ingredients already exist elsewhere in the code.

- The same class *[Debroy et al., ICSTVV10, Martinez et al., ICSE14]*
- The same application *[LeGoues et al., ICSE12, Weimer et al., ASE13, DeMarco et al., ISSTA14 , Long et al., ESEC/FSE15, Long et al., POPL16]*
- Other applications *[Sidiroglou-Douskos et al., PLDI15]*

Mutation Operators

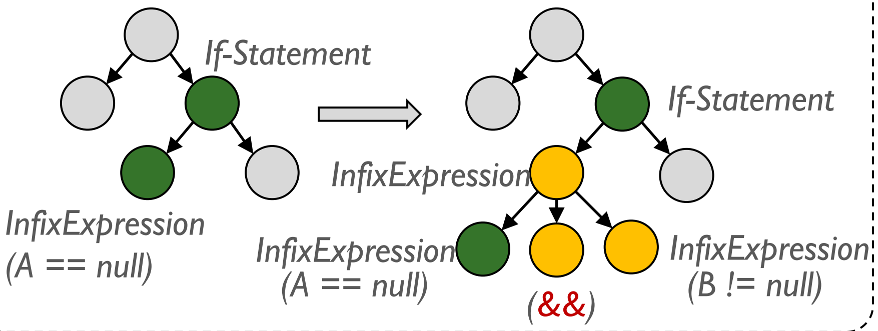
4. Adding *If-Guard* Condition




```
      project.getProject();
      ....
 +     if (fcl != null) {
         removeFileListener(fcl);
 +     }

```

5. Tightening Condition



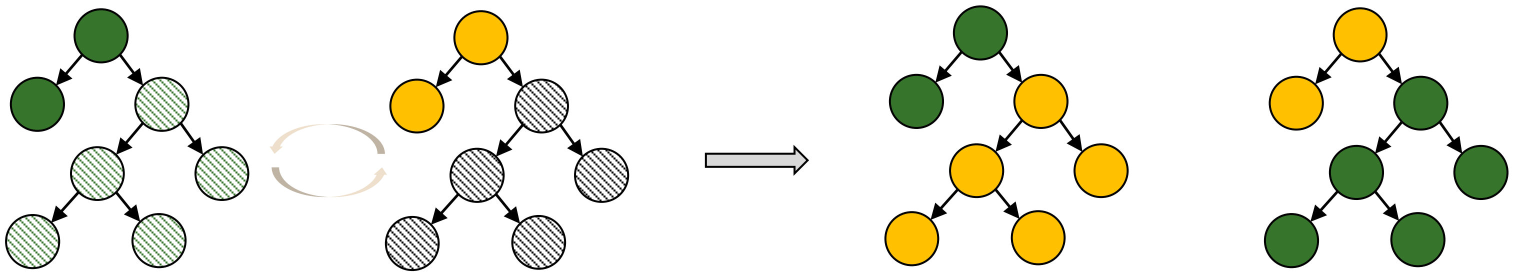
Many real bug fixes are related to buggy if conditions [EmSE 2009].

```
 +  if (A == null && B != null){
      result = annotated;
    } else {
      result = getResult();
    }
```

Loosening condition can be achieved in the same way by replacing the operator && with ||.

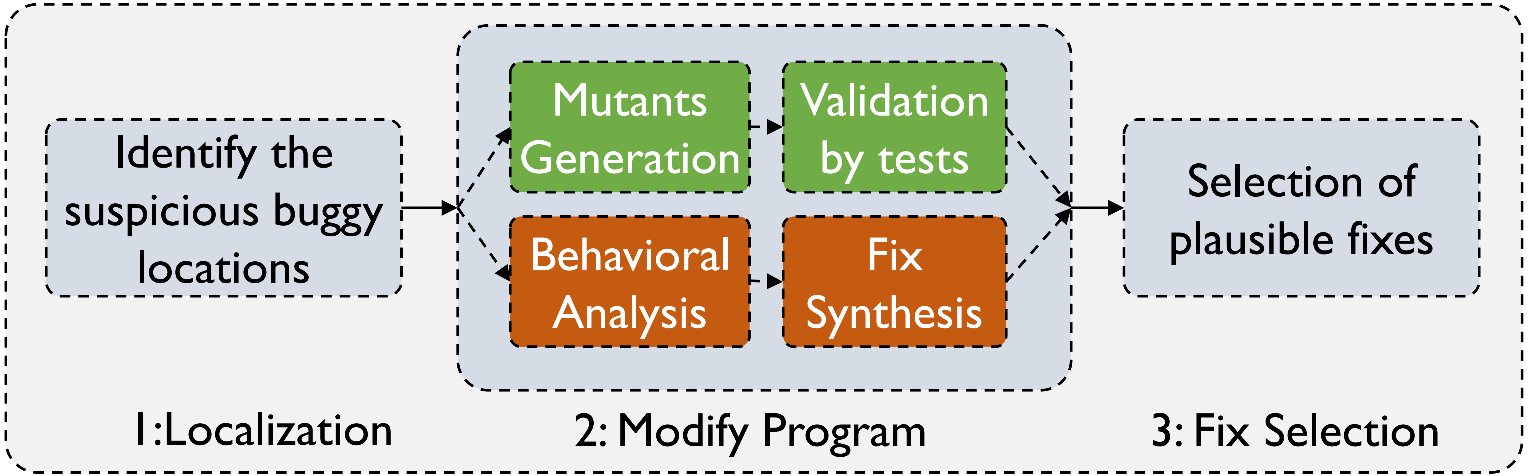
Patch Generation - GenProg

- Mutation Operator: *Insertion*, *Replacement*, and *Deletion*.
- Crossover Operator: Cross over between two candidates.



Candidate P_1 *Candidate* P_2 *Candidate* C_1 *Candidate* C_2

General Process



Mutation-driven (also known as generate-and-validate)

Semantics-driven

Condition Fix Synthesis – SPR [ESEC/FSE15]

```
void foo(...) {
    ...
    char *buf = malloc(nsize);


    memcpy(buf, a, nsize);
    ...
}
```

Add a potentially guarded control statement

- Create a template fix with an *abstract condition*
- Try different values of the *abstract condition* during the symbolic execution (initially set to 0) Flip the value when the execution failed.
- Record the *variable values* at each condition invocation(*buf*, *nsize*)
- Synthesize a concrete fix using the recorded variables

Condition Fix Synthesis – SPR [ESEC/FSE15]

```
void foo(...) {
    ...
    char *buf = malloc(nsize);
    if (_abst_cond())
        return;
    memcpy(buf, a, nsize);
    ...
}
```

	_abst_cond()	buf	nsize	
1st	0	0x30aa...	100	
2nd	1	0	100000	F→P
3rd	0	0x30ab...	200	
4th	1	0	200000	F→P

_abst_cond() ⟹ buf == 0

SPR fixes 38 bugs out of 69, while **GenProg** fixes only 16, **AE** fixes 25.

RHS of Assignment – SemFix [ICSE13]

- An *expression* is treated as a *function*
- The repair candidates are in one of the two forms:
  - $x = F_{buggy}(\dots) \rightarrow x = F_{fixed}(\dots)$
  - $if\ (F_{buggy}(\dots)) \rightarrow if(F_{fixed}(\dots))$
- The function takes all the *accessible variables* as parameters

RHS of Assignment – SemFix [ICSE13]

```
int G(int a, int b, int c) {
    int bias;
    if (a)
        bias=c;
    else
        bias = b;
    if (bias > c)
        return 1;
    else return 0;
}
```

$bias = F(a, b, c)$

Test	Inputs			Expected output	Observed output	Status
	a	b	c			
1	1	0	100	0	0	Pass
2	1	11	110	1	0	Fail
3	0	100	50	1	1	Pass
4	1	-20	60	1	0	Fail
5	0	0	10	0	0	Pass

Test 2 $< 1,11,110 >$

Test 1 $< 1,0,100 >$

Test 4 $< 1,-20,60 >$

$F(a, b, c) = b + 100 \Leftarrow bias = F(1,11,110) > 110 \wedge F(1,0,100) \leq 100 \wedge F(1,-20,60) > 60$

Pointer Operations are Common

	Referencing (Create location)	Dereferencing (Access location)	Aliasing (Copy pointer)
C:	my_t *p = &var; p = malloc(8);	int x = *ptr; x = ptr2->field;	my_t *pa; pa = pb;
Java:	A a = new A();	int x = a.f;	A a = b;

Pointer related bugs are also common

- Null pointer dereference
- Memory leaks
- Array index out of bounds
- Uninitialized pointers
- Mismatched malloc / free
- Buffer overflows
- Breaking the type system (casts & unions)
- …

```
void foobar(int i) {
  char* p = new char[10];
  if(i) {
    p = 0;   // memory leak
  }
  if ( p->value == 0 ) … // null pointe
  delete[] p;
}
```

1,340,561 (82.6%) out of the 1,622,375 code revisions of IF-clauses filed at GitHub as at Sept 2015 involve null-pointer checks.

Difference between Alias and Points-to

Example:

p = &a; q = &b;
if (…)
  p = &c;
else
  q = &c;
*p = *q + d;

- Alias emphasizes the simultaneity.
  - (p, q) is an alias pair if p and q refer to the same memory location simultaneously after executing a set of program instructions.
- Points-to emphasizes individuality.
  - p→c and q→c are two independent events.
  - $pts(p) \cap pts(q) \neq \Phi$ does not mean (p, q) is a true alias pair. For example, in the snippet on the left, *p never alias to *q.

$pts(p) = \{a, c\}, pts(q) = \{b, c\}$

Basics of Points-to Analysis

Points-to analysis has two parts:

- Abstract the given program (build the abstract domains of pointers and memories)
- Process the program constructs such as assignment “p = q;”

Program Abstraction

Able to distinguish one function call from another

```
public Object foo () {
    Object p1 = new Integer () ;   // o1
    Object q1 = new Integer () ;   // o2
    Object p2 = bar ( p1 ) ;       // c1
    Object q2 = bar ( q1 ) ;       // c2
}

public Object bar ( Object r ) {
    return r ;
}
```

Context Sensitivity:

- Function bar has two invocations, which creates two instances of r;
- If we distinguish the two invocations of bar with the callsite labels c1 and c2, we can distinguish the two instances of r by r^{c1} and r^{c2}.

Context Sensitive

- Whether different calling contexts are distinguished

```
void yellow()
{
1. red(1);
2. red(2);
3. green();
}
```

```
void red(int x)
{
..
}
```

```
void green()
{
  green();
  yellow();
}
```

Context sensitive distinguishes 2 different calls to red( )

Basics of Points-to Analysis

Points-to analysis has two parts:

- Abstract the given program (build the abstract domains of pointers and memories)
- Process the program constructs such as assignment “p = q;”

Handling Program Constructs

Program Point:

- Every statement s in the program has two program points:
  - the point before executing s
  - the point after executing s
- Unless otherwise specified, our discussion refers to the point after executing a statement.

Flow Sensitive

- A *flow* sensitive analysis considers the order (flow) of statements
  - Flow insensitive = usually linear-type algorithm
  - Flow sensitive = usually at least quadratic (dataflow)
- Examples:
  - Type checking is flow insensitive since a variable has a single type regardless of the order of statements
  - Detecting uninitialized variables requires flow sensitivity

```
   x = 4;
6. ....
   x = 5;
```

Flow sensitive analysis distinguishes values of x before and after line 6, flow insensitive analysis cannot.

Handling Program Constructs

Path Sensitivity:

- A path sensitive analysis maintains branch conditions along each ***execution path***
  - Requires extreme care to make the analysis scalable
  - Subsumes flow sensitivity

Handling Program Constructs

Field Sensitivity

- A field sensitive points-to analysis distinguishes the fields defined in the same structure (or class in Java/C++) while field insensitive analysis doesn't.
- The field insensitive analysis is especially important for C language. In theory, the field sensitivity is unsound for C and requires exponential time to complete.

```
struct T {
    int *p, *q;
};
```

Andersen's Analysis

- Evaluation rules for different constraints (statements):

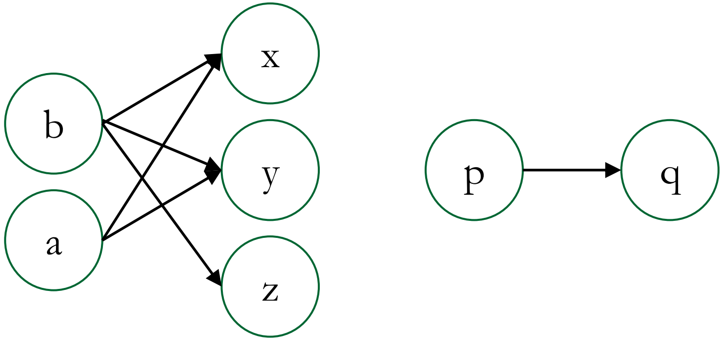
Constraint Type	Symbolic Form	Evaluation Rule
Base	p = &x	$pts(p) = pts(p) \cup \{x\}$
Simple	p = q	$pts(p) = pts(p) \cup pts(q)$
Store	*(p+c) = q	$\forall x \subseteq pts(p), pts(x) = pts(x) \cup pts(q)$
Load	p = *(q+c)	$\forall x \subseteq pts(q), pts(p) = pts(p) \cup pts(x)$

1. c is a constant
2. The store and load constraints are also called complex constraints.

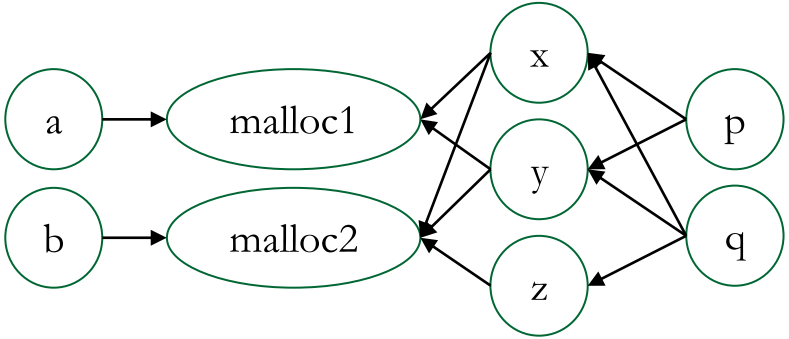
Andersen's Analysis

- The complexity is $O(n^3)$, where n is the number of pointers and we have $O(n)$ statements. This is because we examine in each iteration $O(n)$ statements, and in the worst case we have $O(n^2)$ iterations.
- Recent work observes: Close to $O(n^2)$ if:
  - Few statements dereference each variable
  - Control flow graphs not too complex
  - Both conditions are common in practice

Final Pointer Assignment Graph:



Final Points-to Graph:



Important Notes

- May 18 (Tuesday), 16:30-19:30am, Online over Canvas and Zoom. Open notes (hard copies only). 3 hours.
- All topics covered in lectures will be examined.
- Exam will mostly focus on concepts and principles.
- Your screen on desktop/laptop must be wholly occupied by Canvas
- Must enable your Zoom video camera on mobile phone during exam
- Video camera must show the side-view of you, your keyboard, mouse and the Canvas screen
- Bring along your student ID card
- Multiple choices: Each question can have multiple answers; no point received if any of the choices selected is incorrect
- About 10% will be on coding – basic Java knowledge would be good enough
- Prepare white papers for sketch work for short/long questions

End of Review

Examination will be open notes!