COMP5111 – Fundamentals of Software Testing and Analysis

Symbolic Execution

Shing-Chi Cheung

Computer Science & Engineering

HKUST

Automatic Software Testing

- Random testing
- Symbolic testing
- Concolic testing

Automatic Software Testing

- Random testing
- Symbolic testing
- Concolic testing

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
  }
}
```

- Key idea: execute programs using symbolic input values instead of concrete execution
- Concrete execution x=0, y=1
- Symbolic execution x=a, y=b

Adapted from http://www.zvonimir.info/teaching/sv-2012-fall/sv_lecture_11.pdf

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

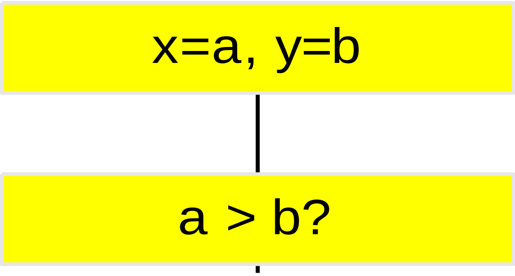
x=a, y=b

☞

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

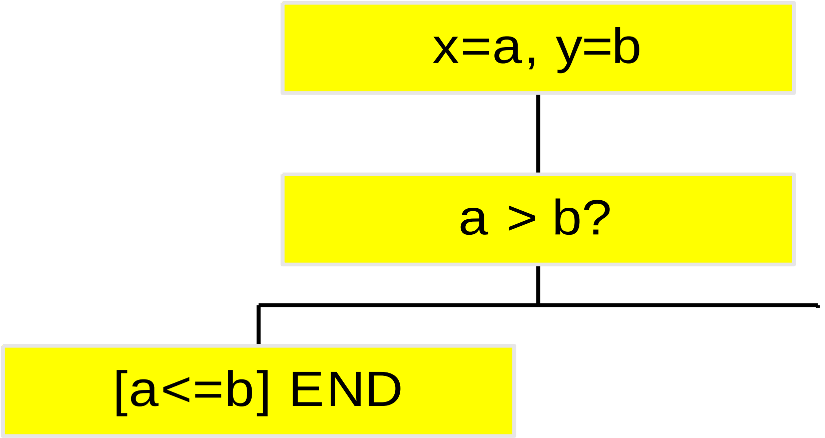
Symbolic Testing (a.k.a. Symbolic Execution)

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



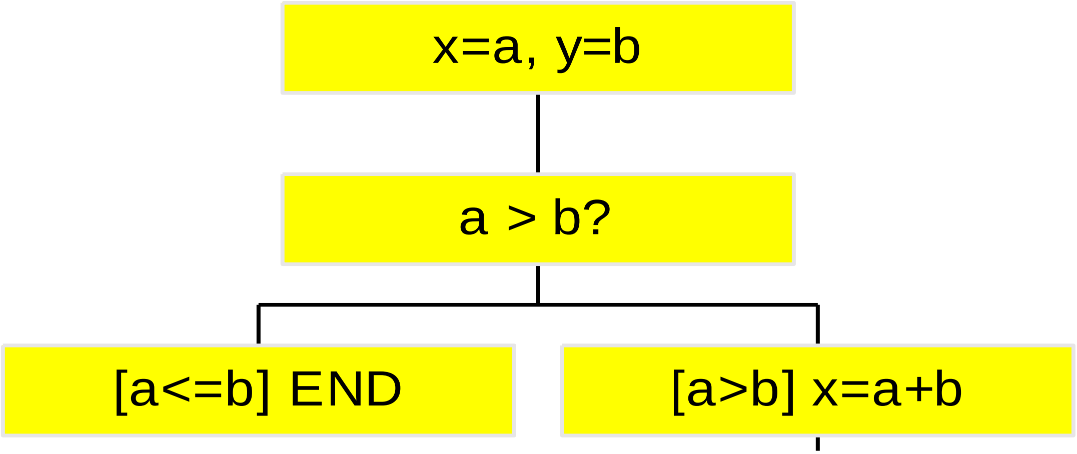
Symbolic Testing (a.k.a. Symbolic Execution)

```
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
  }
}
```



x=a, y=b

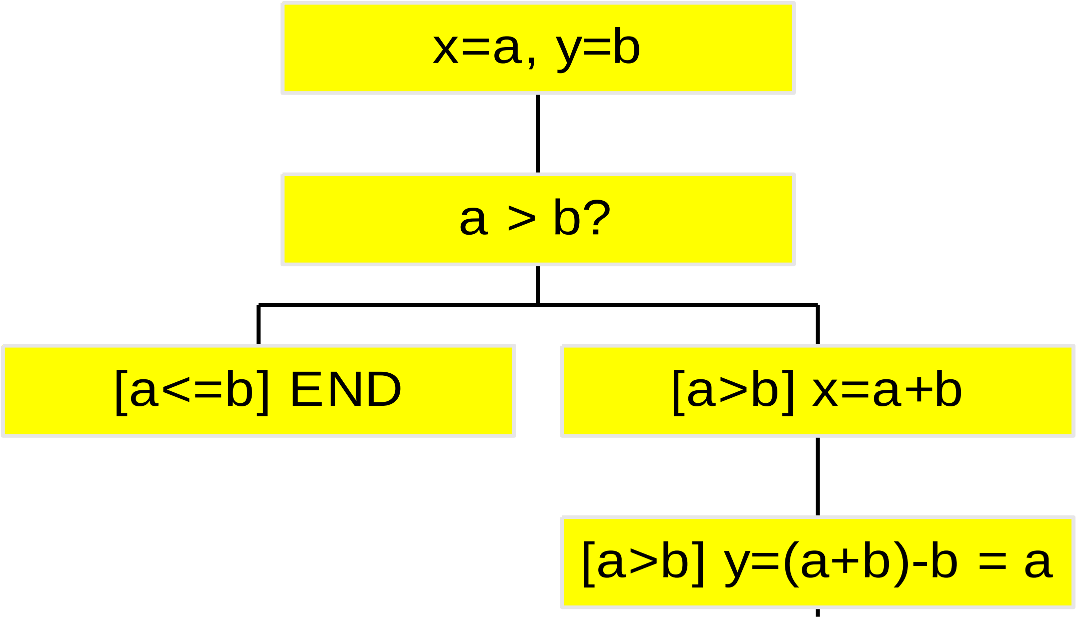
a > b?

[a<=b] END

[a>b] x=a+b

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
     }
   }
```



x=a, y=b

a > b?

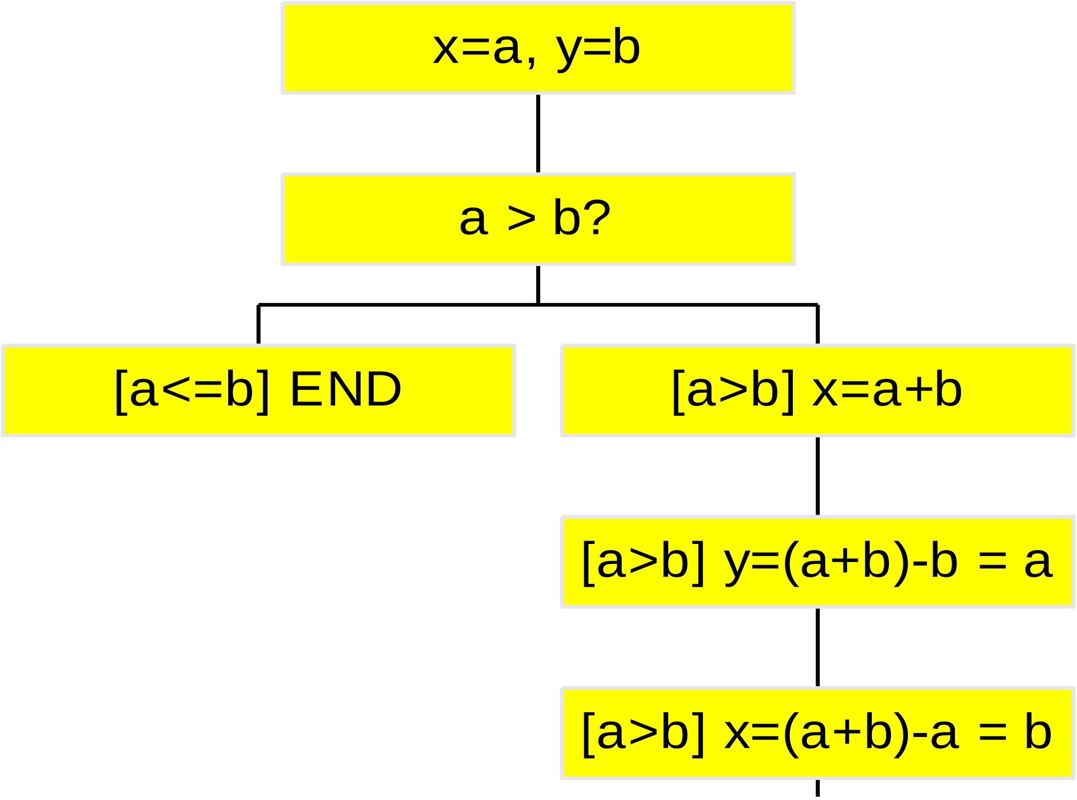
[a<=b] END

[a>b] x=a+b

[a>b] y=(a+b)-b = a

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
  }
}
```



x=a, y=b

a > b?

[a<=b] END

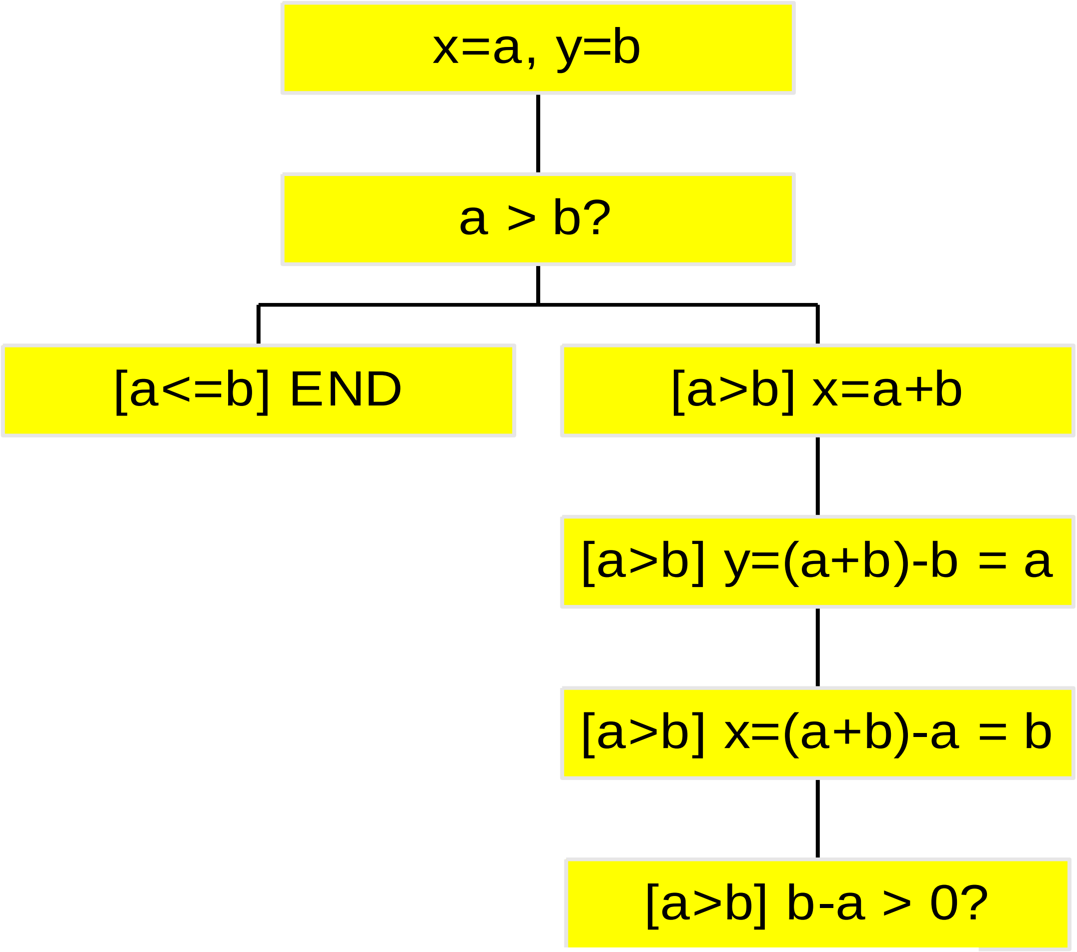
[a>b] x=a+b

[a>b] y=(a+b)-b = a

[a>b] x=(a+b)-a = b

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
  }
}
```



x=a, y=b

a > b?

[a<=b] END

[a>b] x=a+b

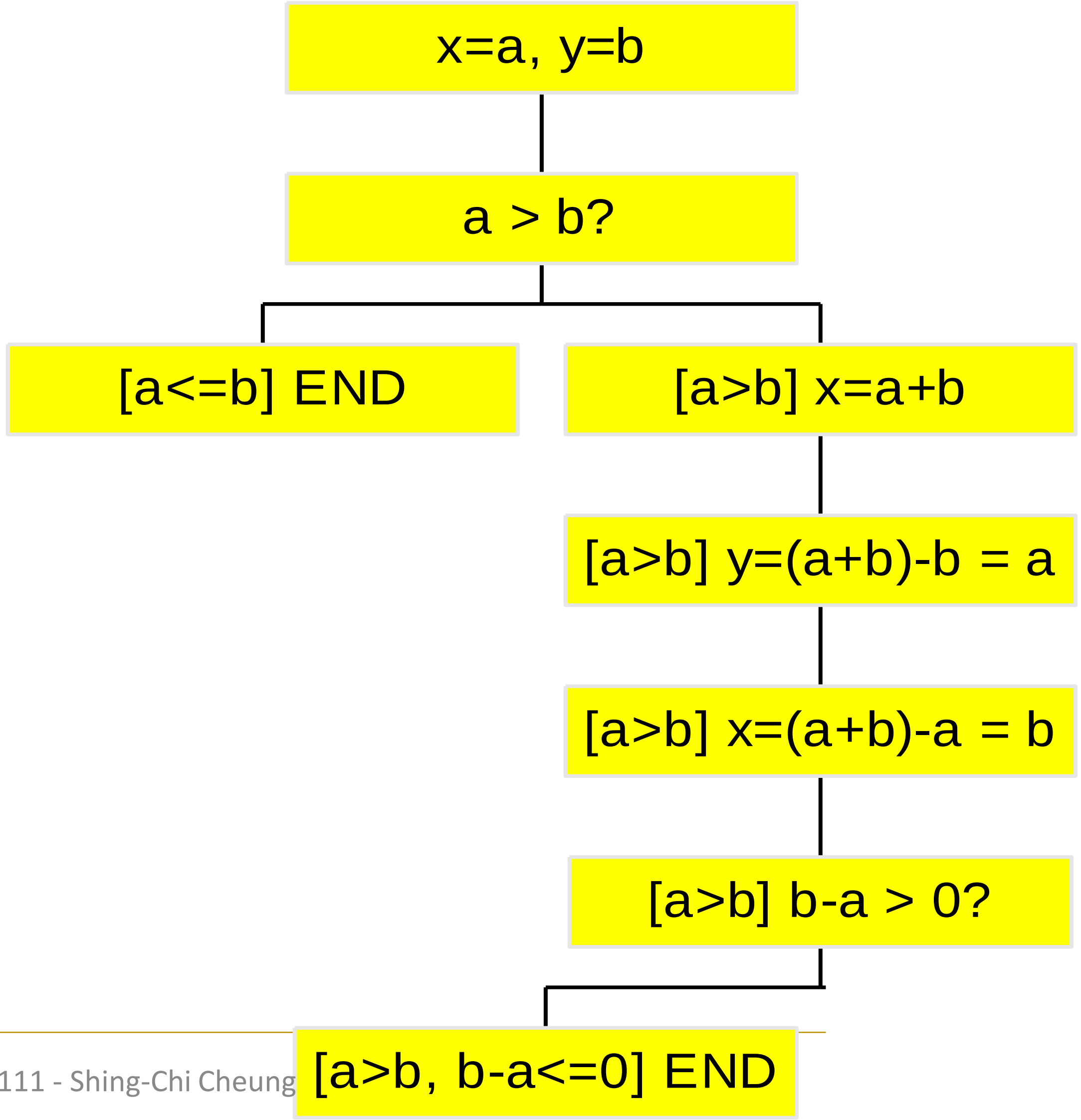
[a>b] y=(a+b)-b = a

[a>b] x=(a+b)-a = b

[a>b] b-a > 0?

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
 }
}
```



x=a, y=b

a > b?

[a<=b] END

[a>b] x=a+b

[a>b] y=(a+b)-b = a

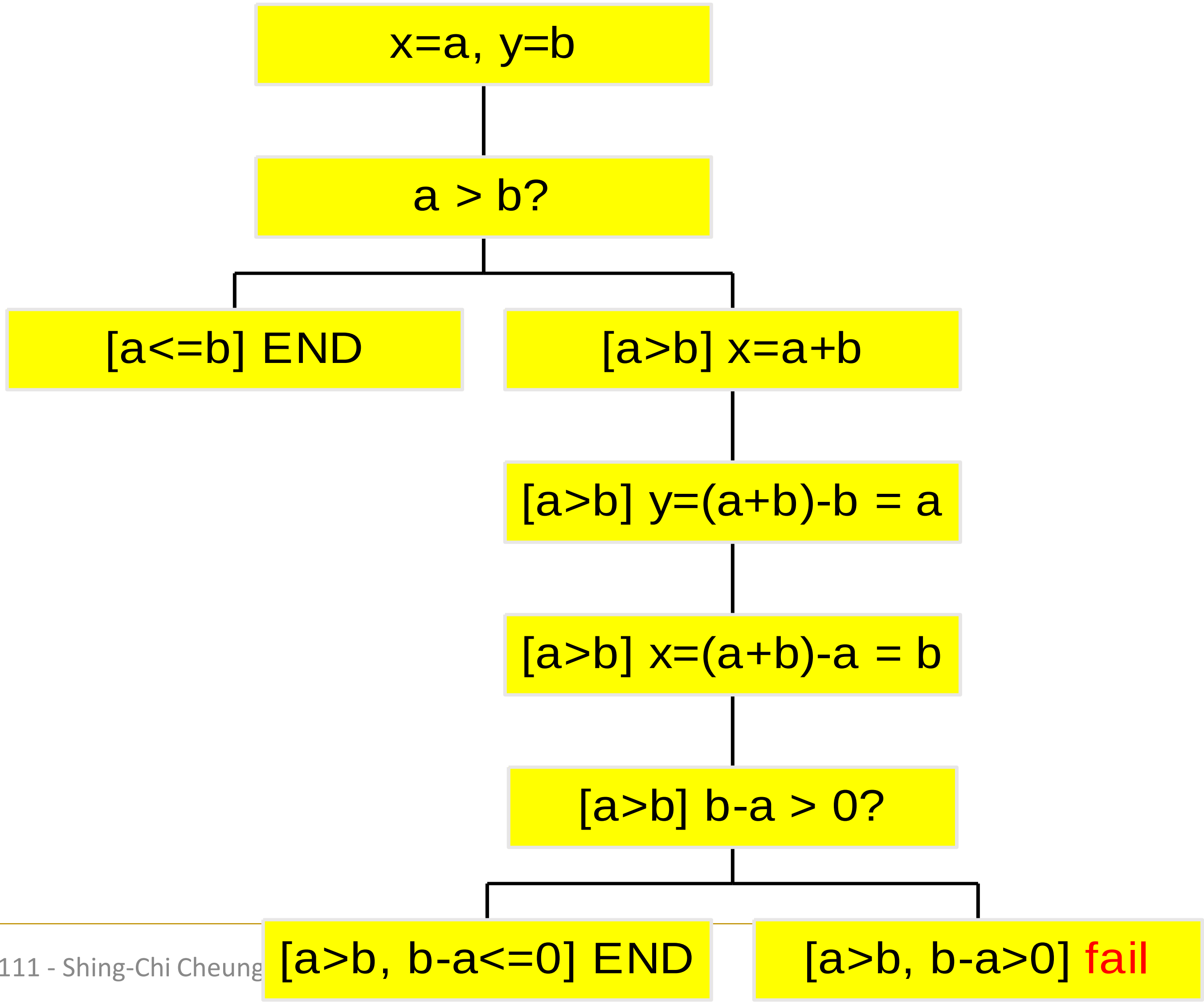
[a>b] x=(a+b)-a = b

[a>b] b-a > 0?

[a>b, b-a<=0] END

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 (Symbolic Execution Tree)

```
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

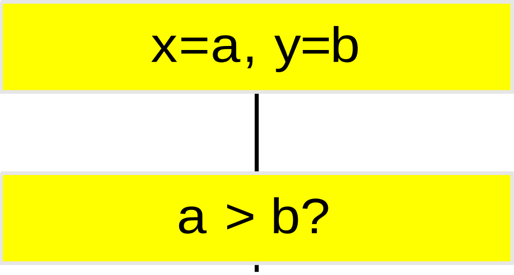
Symbolic Testing (a.k.a. Symbolic Execution)

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
  }
}
```



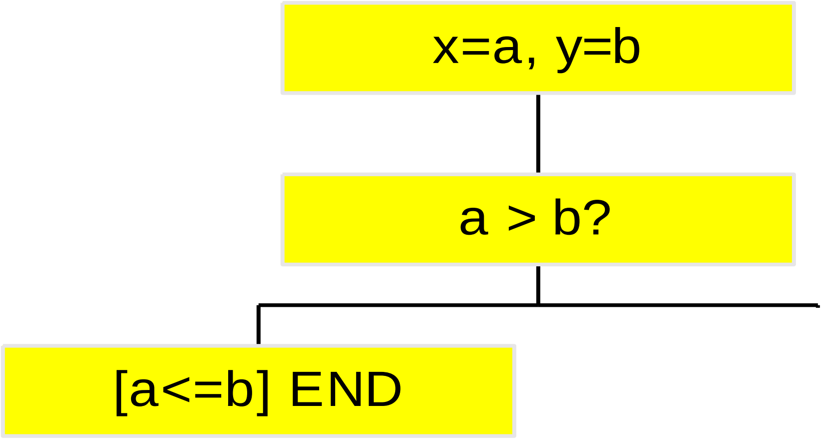
x=a, y=b

a > b?

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

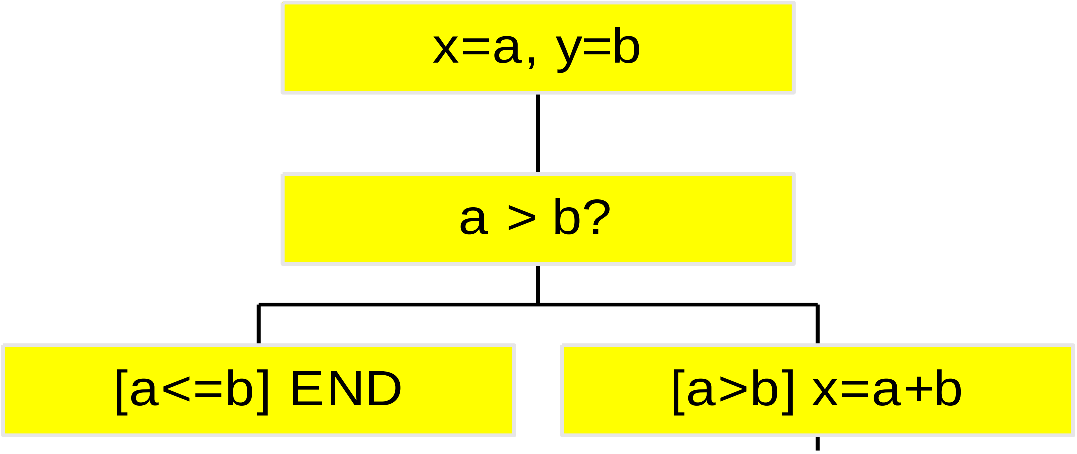
```
foo (int& x, int& y) {
  if (x>y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0)
        assert (false); // bug
  }
}
```

☞



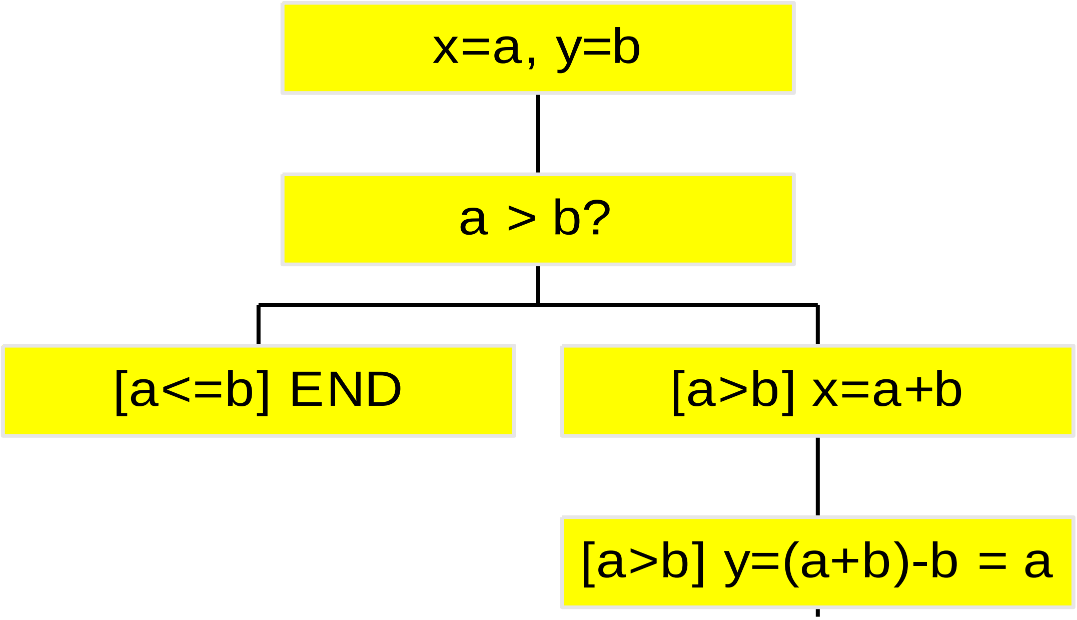
Symbolic Testing (a.k.a. Symbolic Execution)

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



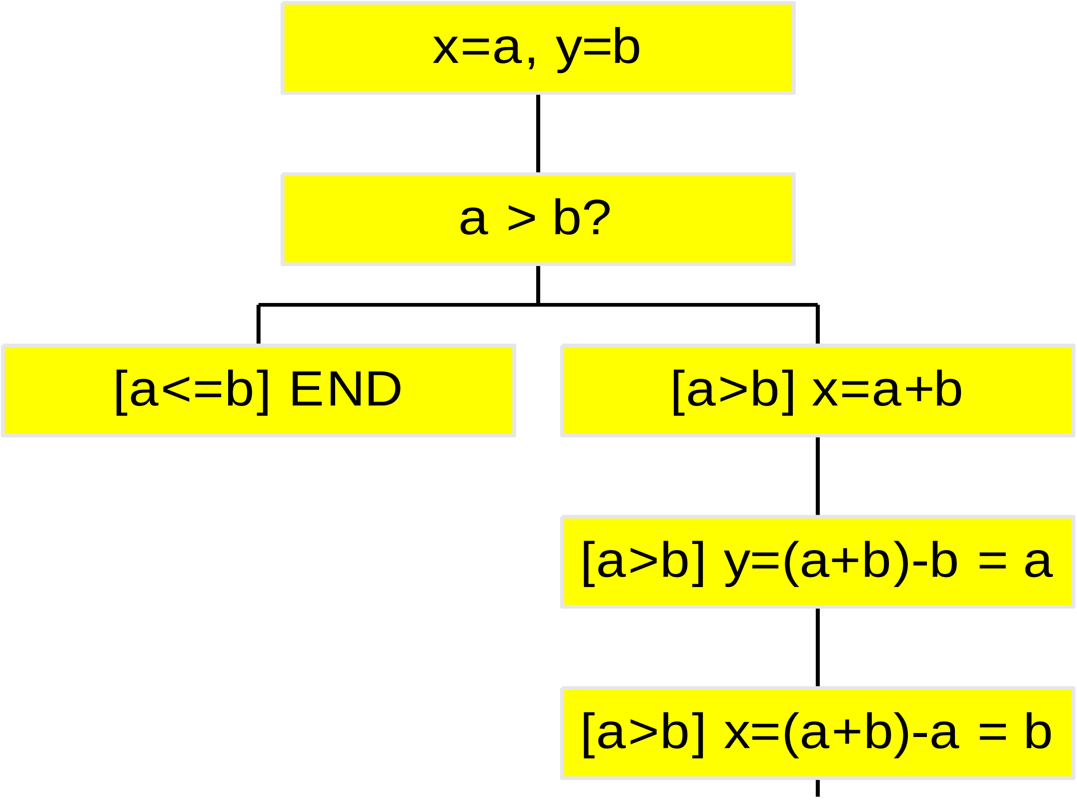
Symbolic Testing (a.k.a. Symbolic Execution)

```
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
  }
}
```



x=a, y=b

a > b?

[a<=b] END

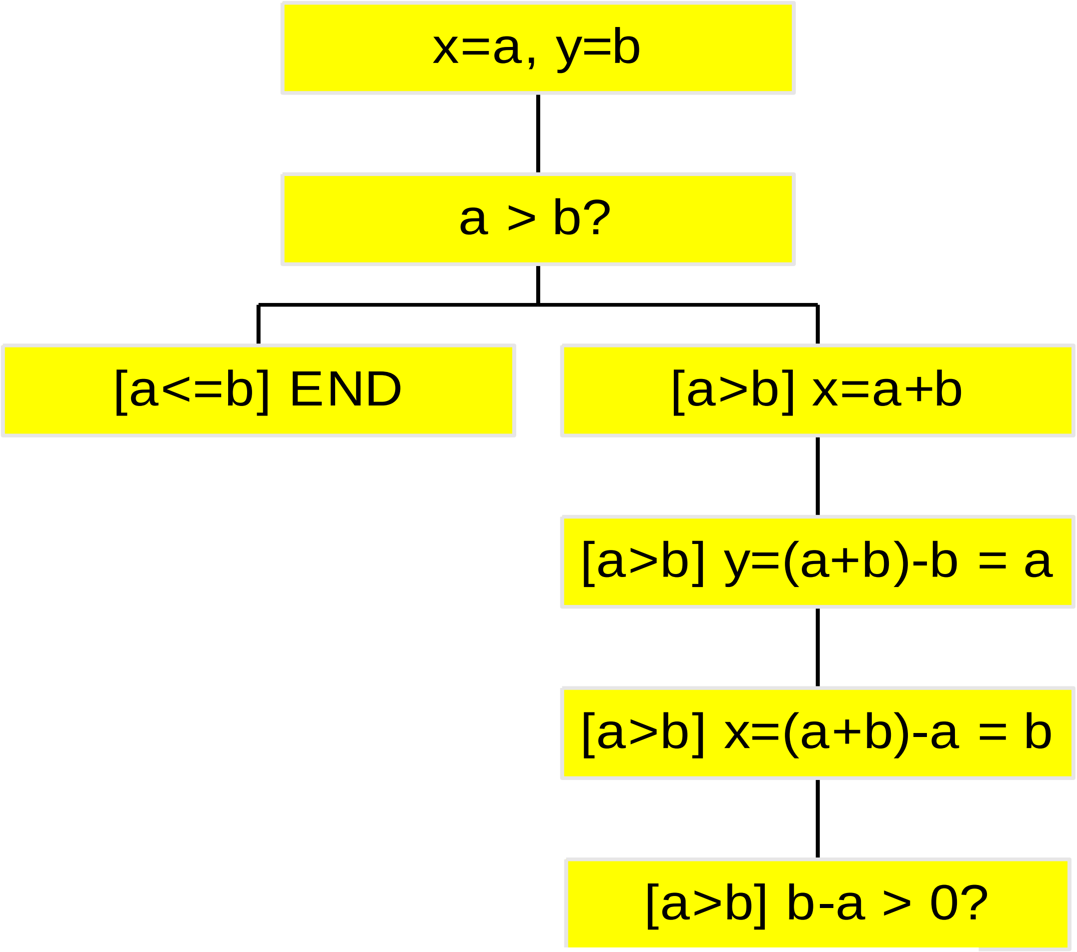
[a>b] x=a+b

[a>b] y=(a+b)-b = a

[a>b] x=(a+b)-a = b

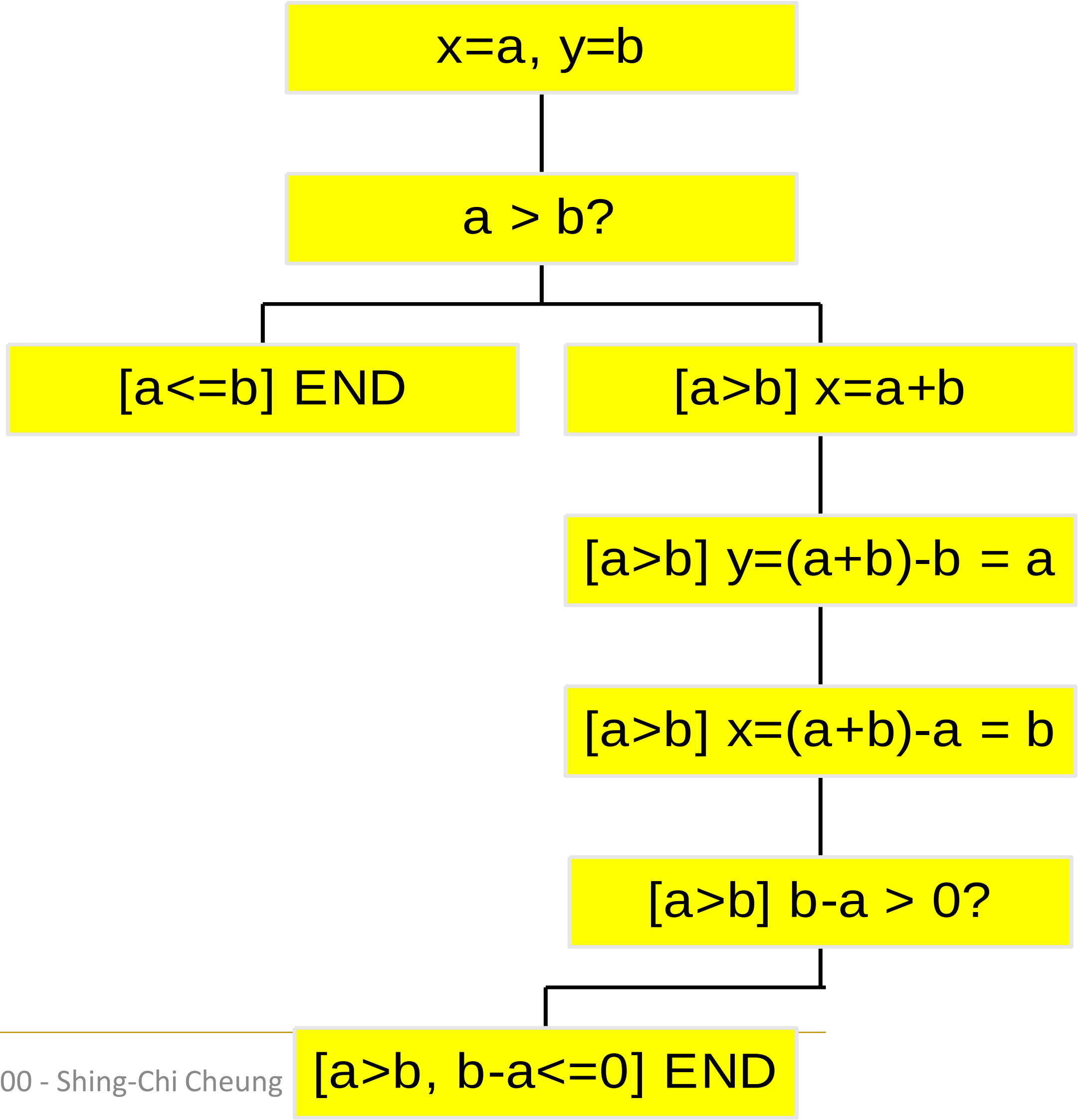
Symbolic Testing (a.k.a. Symbolic Execution)

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



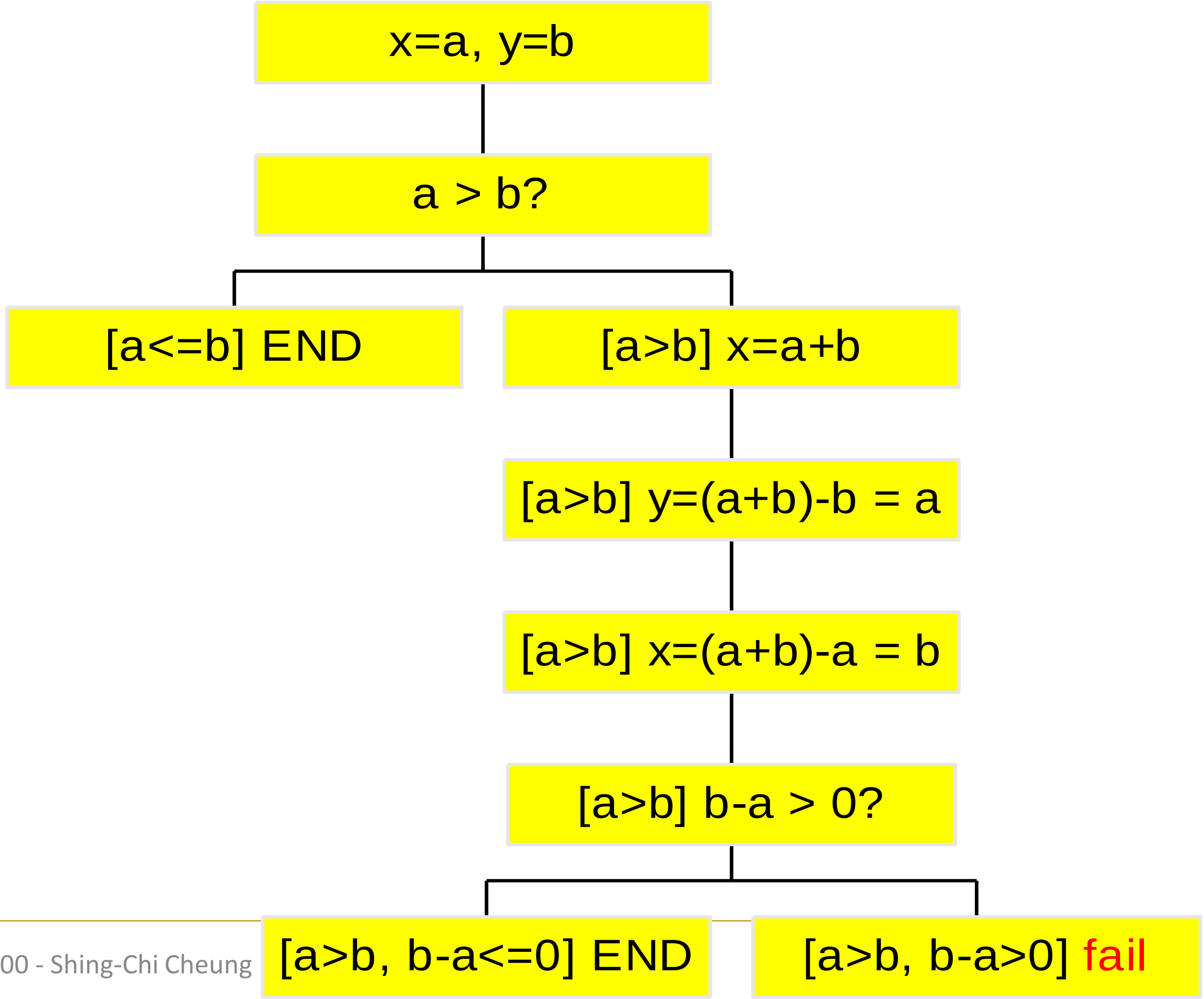
Symbolic Testing (a.k.a. Symbolic Execution)

```
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
  }
}
```



x=a, y=b

a > b?

[a<=b] END

[a>b] x=a+b

[a>b] y=(a+b)-b = a

[a>b] x=(a+b)-a = b

[a>b] b-a > 0?

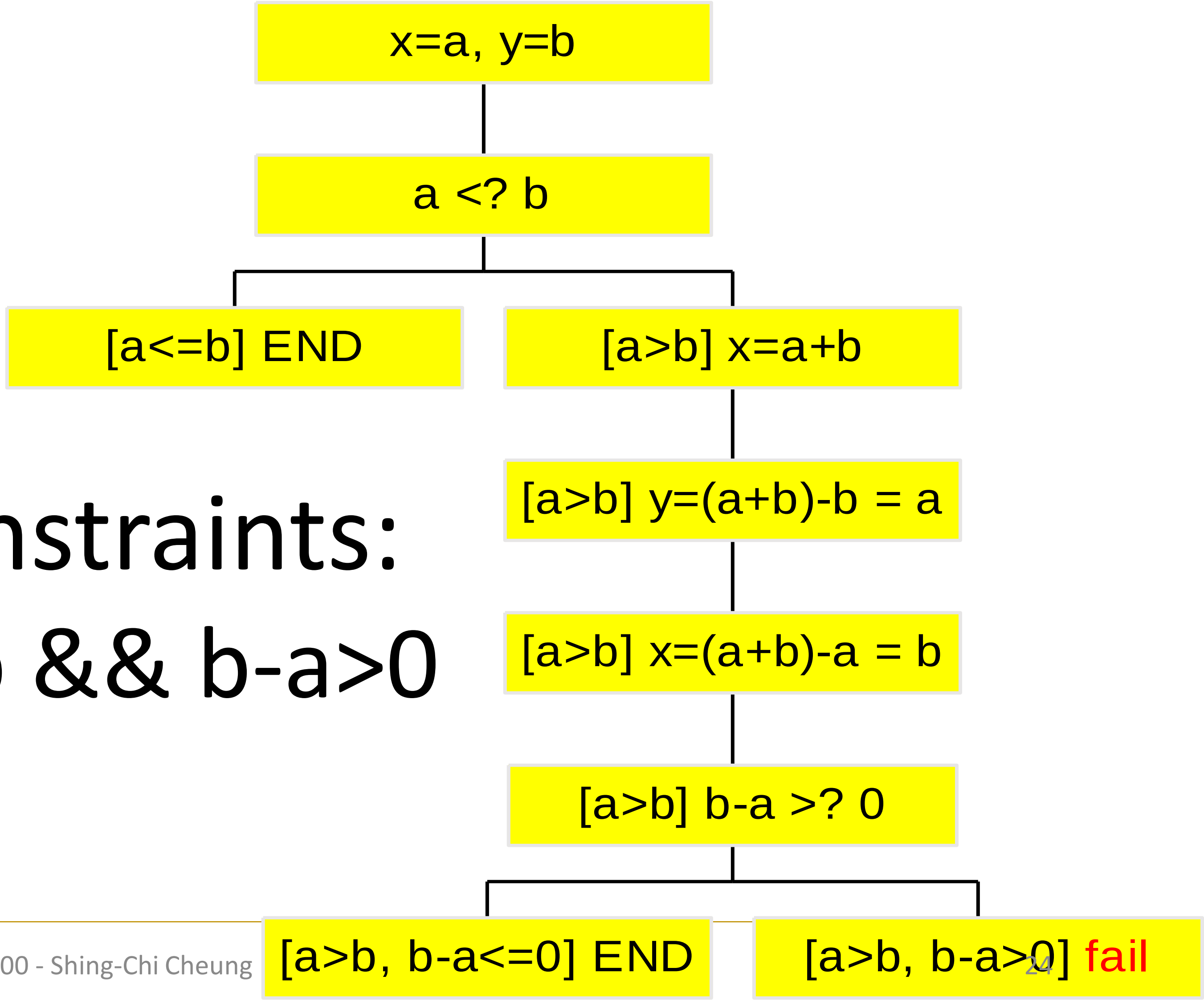
[a>b, b-a<=0] END

[a>b, b-a>0] fail

Symbolic Testing (Symbolic Execution Tree)

```
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



x=a, y=b

a <? b

[a<=b] END

[a>b] x=a+b

[a>b] y=(a+b)-b = a

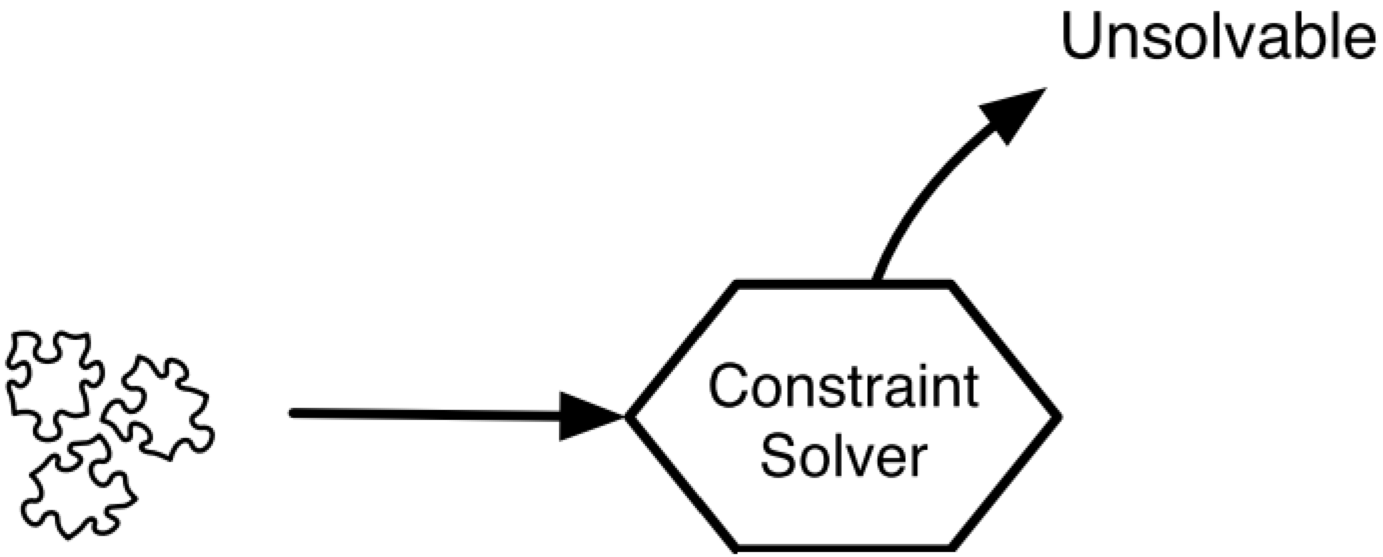
[a>b] x=(a+b)-a = b

[a>b] b-a >? 0

[a>b, b-a<=0] END

[a>b, b-a>0] fail

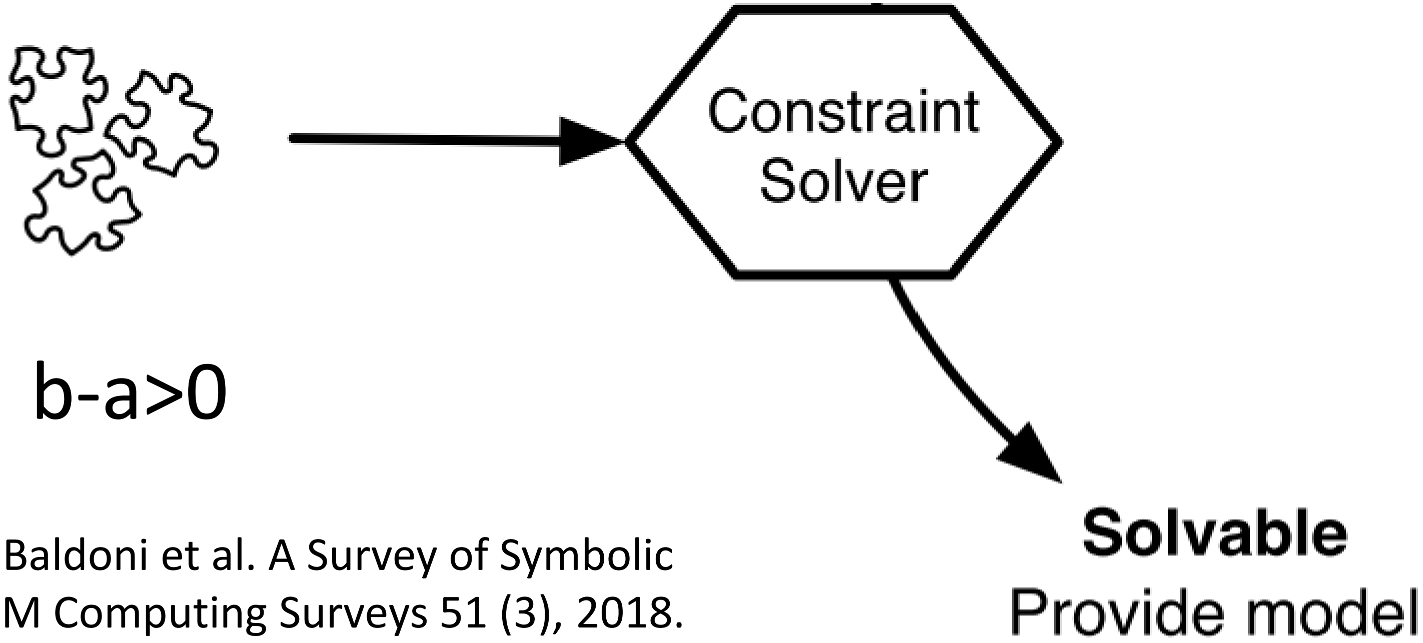
Using a Linear Constraint Solver



a>b && b-a>0

Efficient linear constraint solvers are available

Constraint Solving with What-if Analysis



b-a>0

a = 0, b = 1

Further reading: Roberto Baldoni et al. A Survey of Symbolic Execution Techniques, ACM Computing Surveys 51 (3), 2018.

Automatic Software Testing

- Random testing
- Symbolic testing
- Concolic testing

Koushik Sen

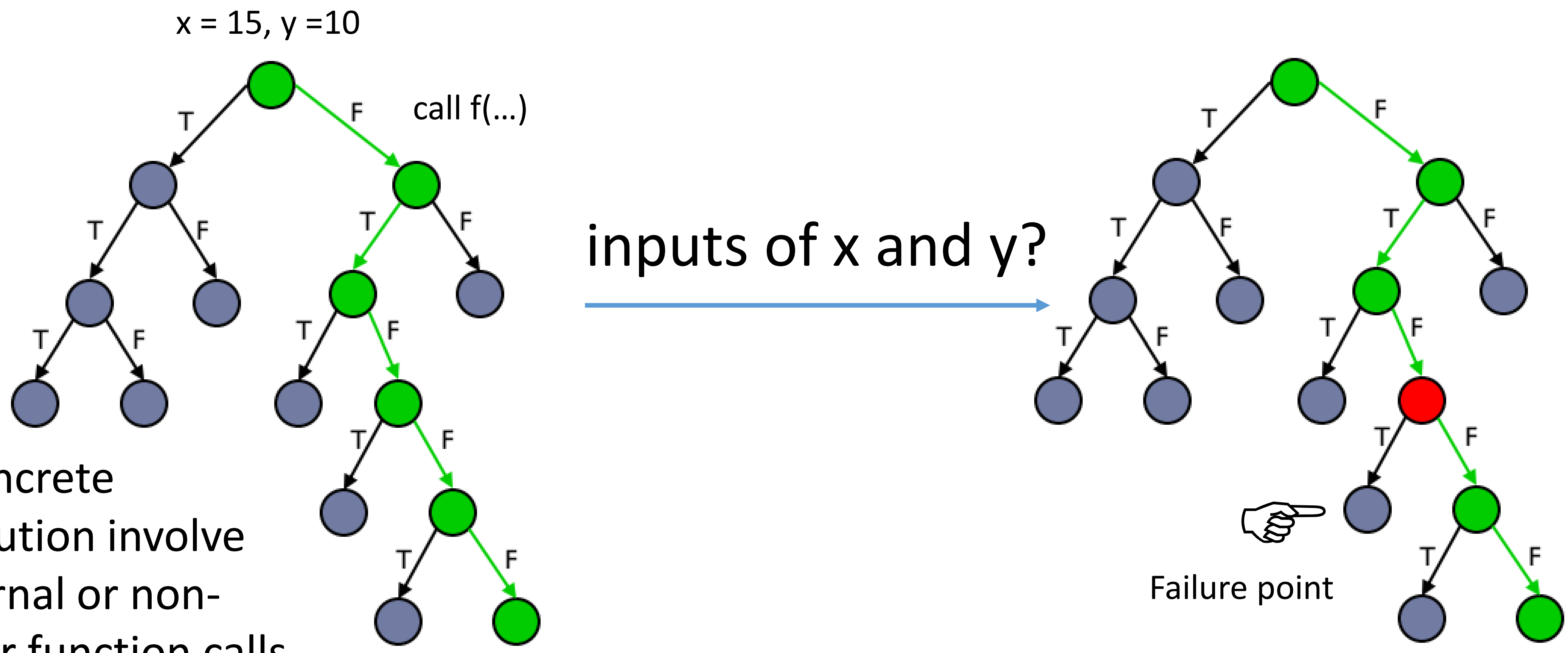
Professor, UC Berkeley

Research Areas

Programming Systems, Software Engineering, Programming Languages, and Formal Methods: Software Testing, Verification, Model Checking, Runtime Monitoring, Performance Evaluation, and Computational Logic

Security

Concolic = Concrete + Symbolic



x = 15, y =10

call f(…)

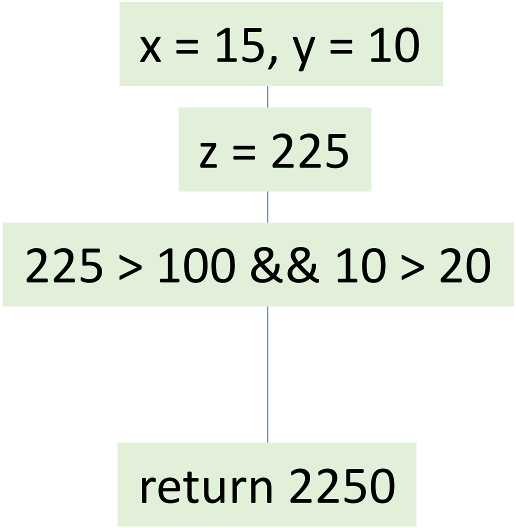
inputs of x and y?

Failure point

A concrete execution involve external or non-linear function calls

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

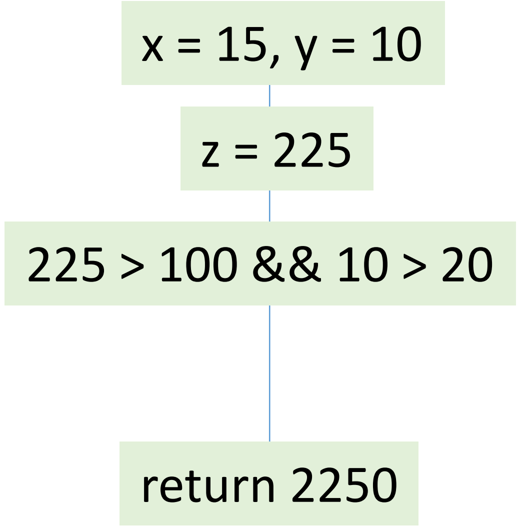
return 2250

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;
}
```

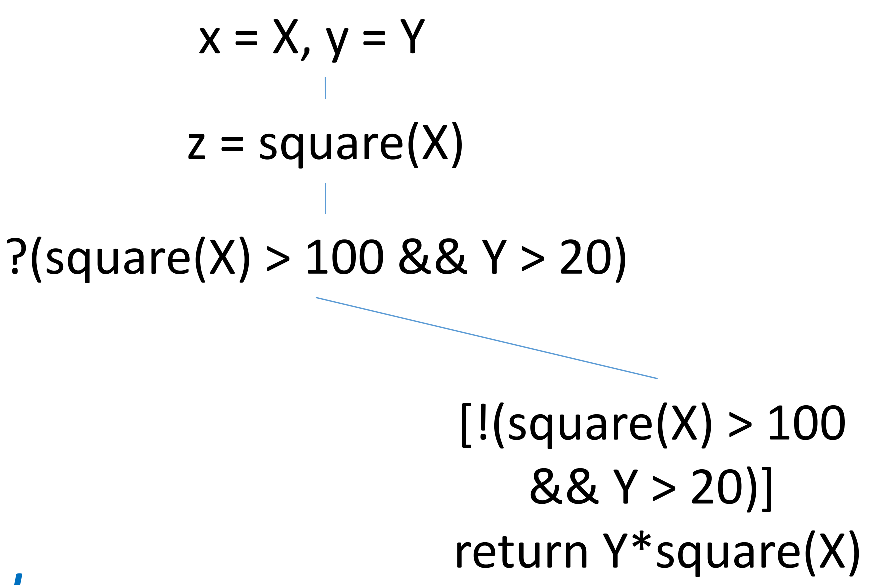


x = 15, y = 10

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)

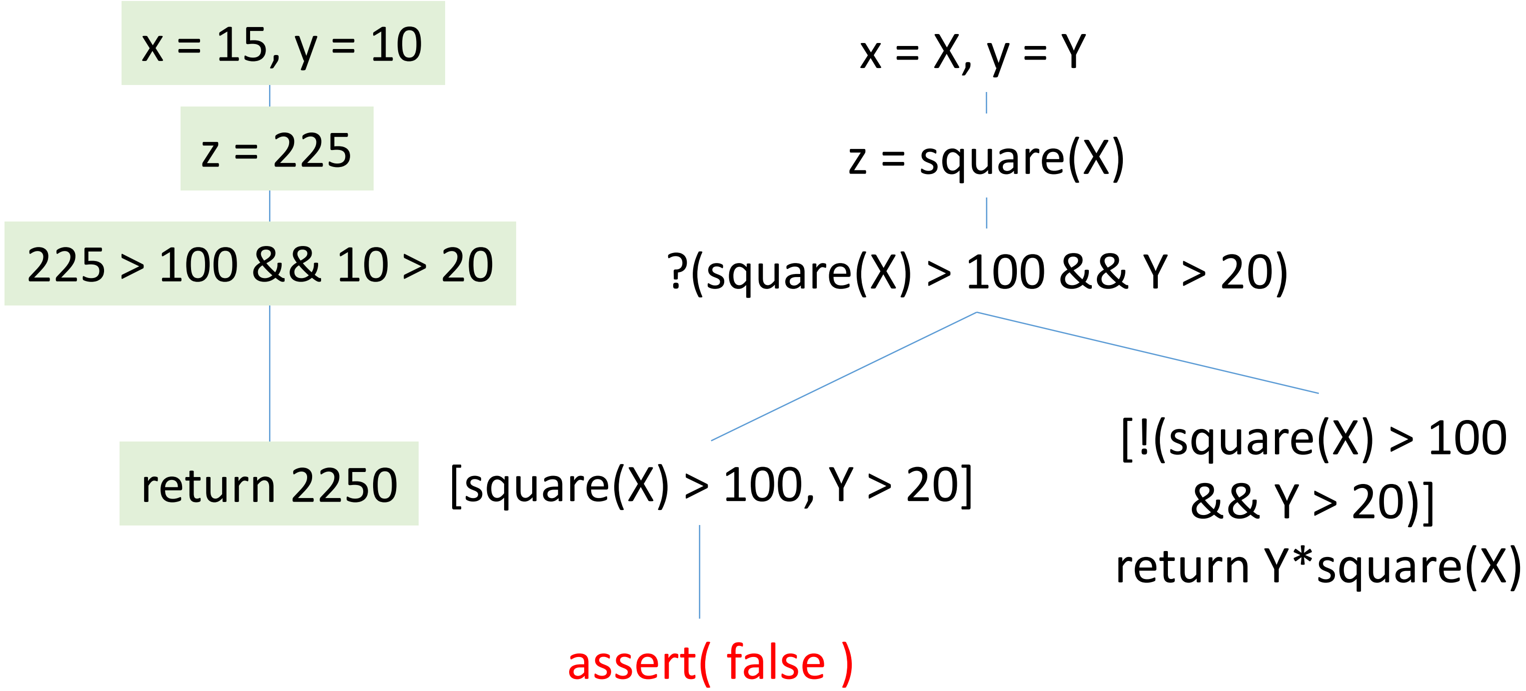
Execute program concretely

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;
}
```



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]

assert(false)

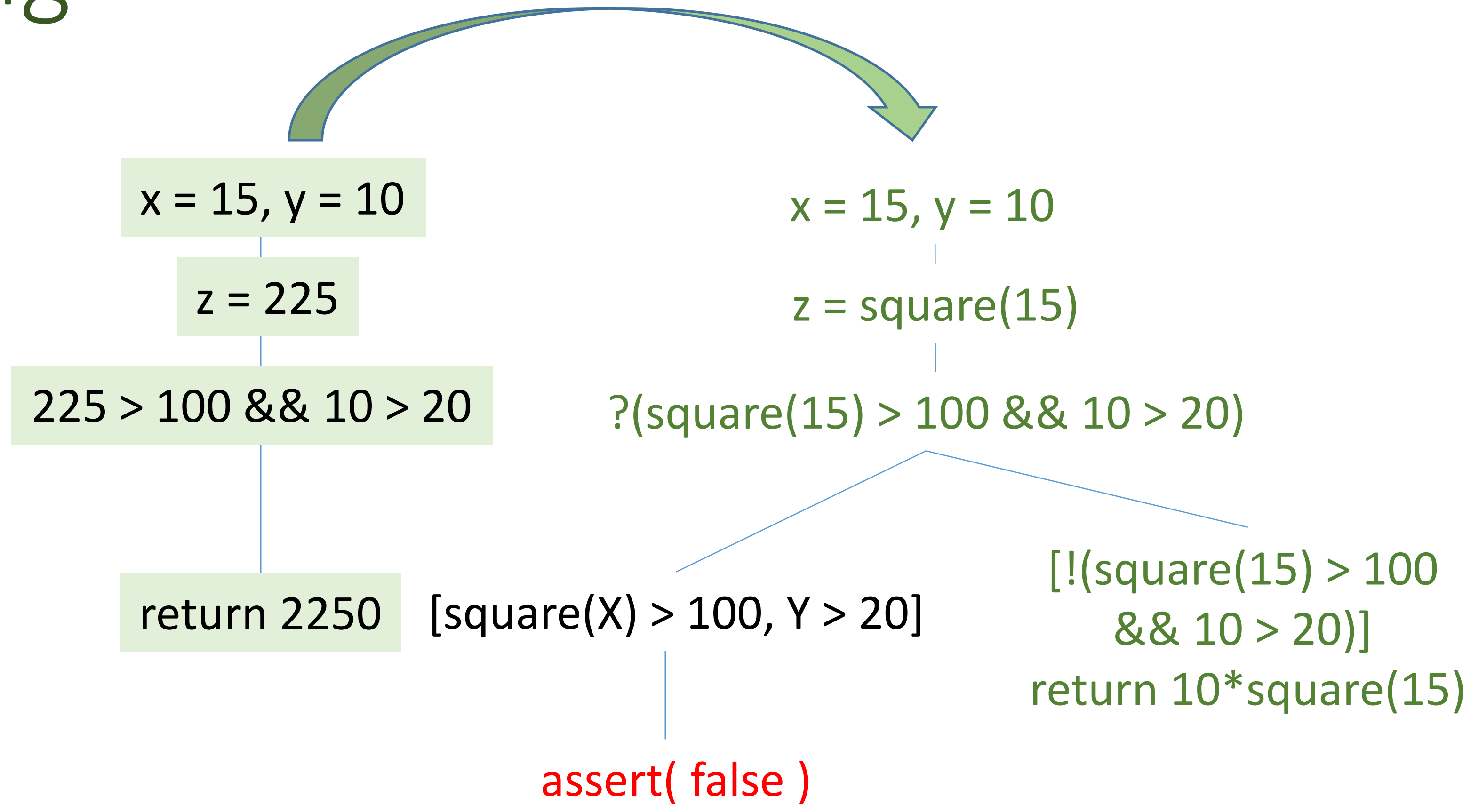
[!(square(X) > 100 && Y > 20)]
return Y*square(X)

Test: foo(15, 10)

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)

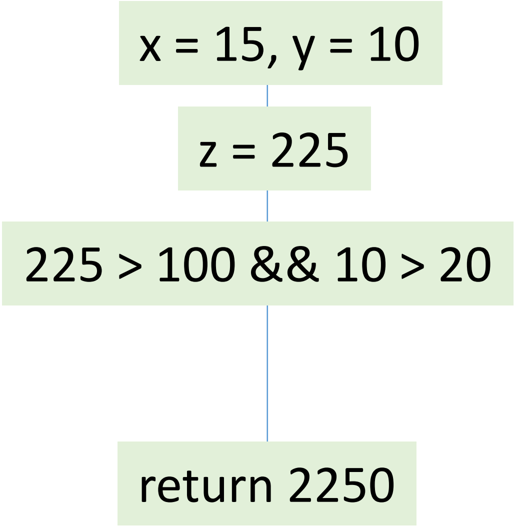
The concrete test and our target share a long prefix in execution

→ The concrete test inputs should partially solve the negated path condition

→ Only need to solve remaining unsolved constraints, which are likely linear

Concolic Testing

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

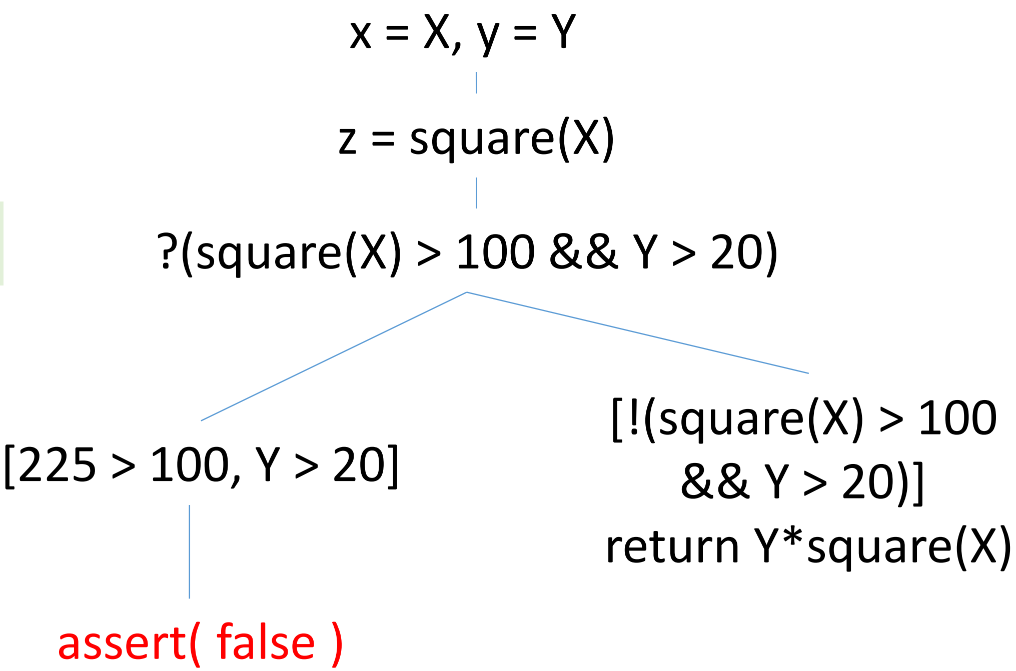


x = 15, y = 10

z = 225

225 > 100 && 10 > 20

return 2250



x = X, y = Y

z = square(X)

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

[225 > 100, Y > 20]

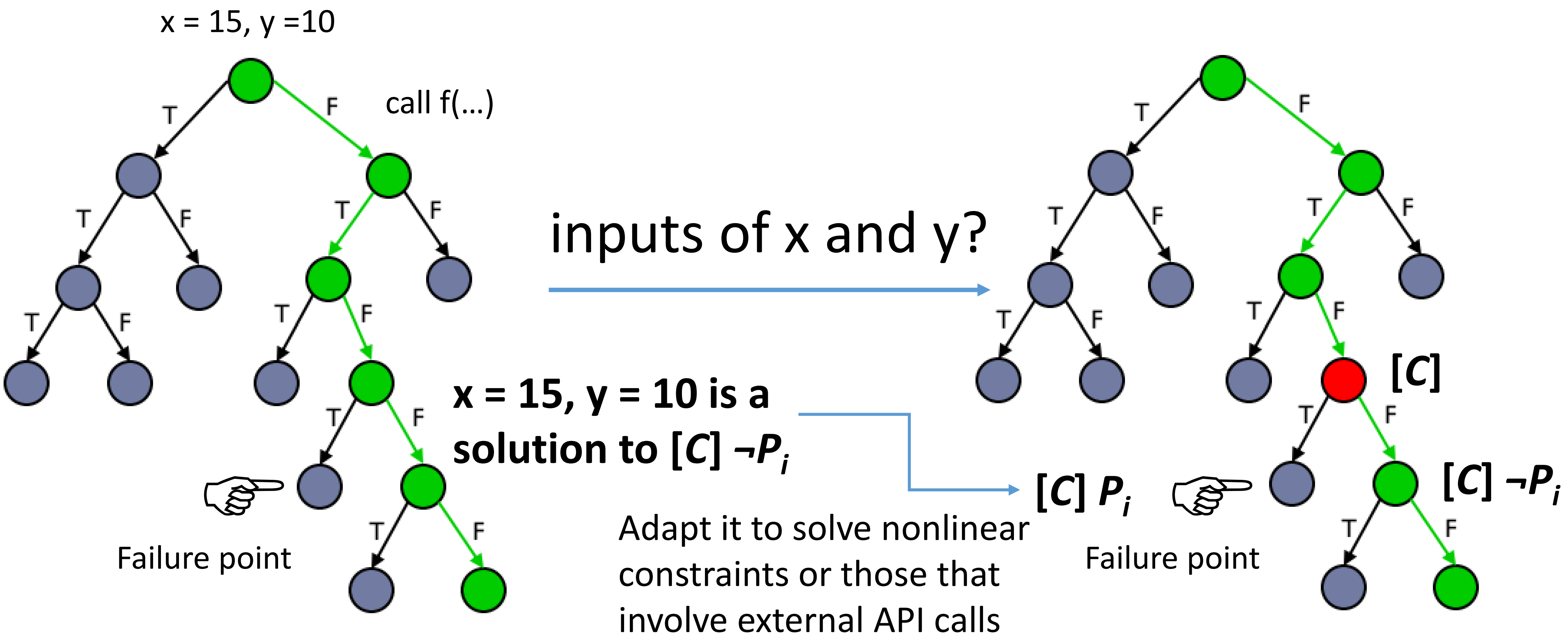
assert(false)

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

Test: foo(15, 10)

Test: foo(15, 21)

Concolic = Concrete + Symbolic (Summary)



x = 15, y =10

call f(…)

inputs of x and y?

x = 15, y = 10 is a solution to [*C*] ¬*P_i*

Adapt it to solve nonlinear constraints or those that involve external API calls

[*C*]

[*C*] *P_i*

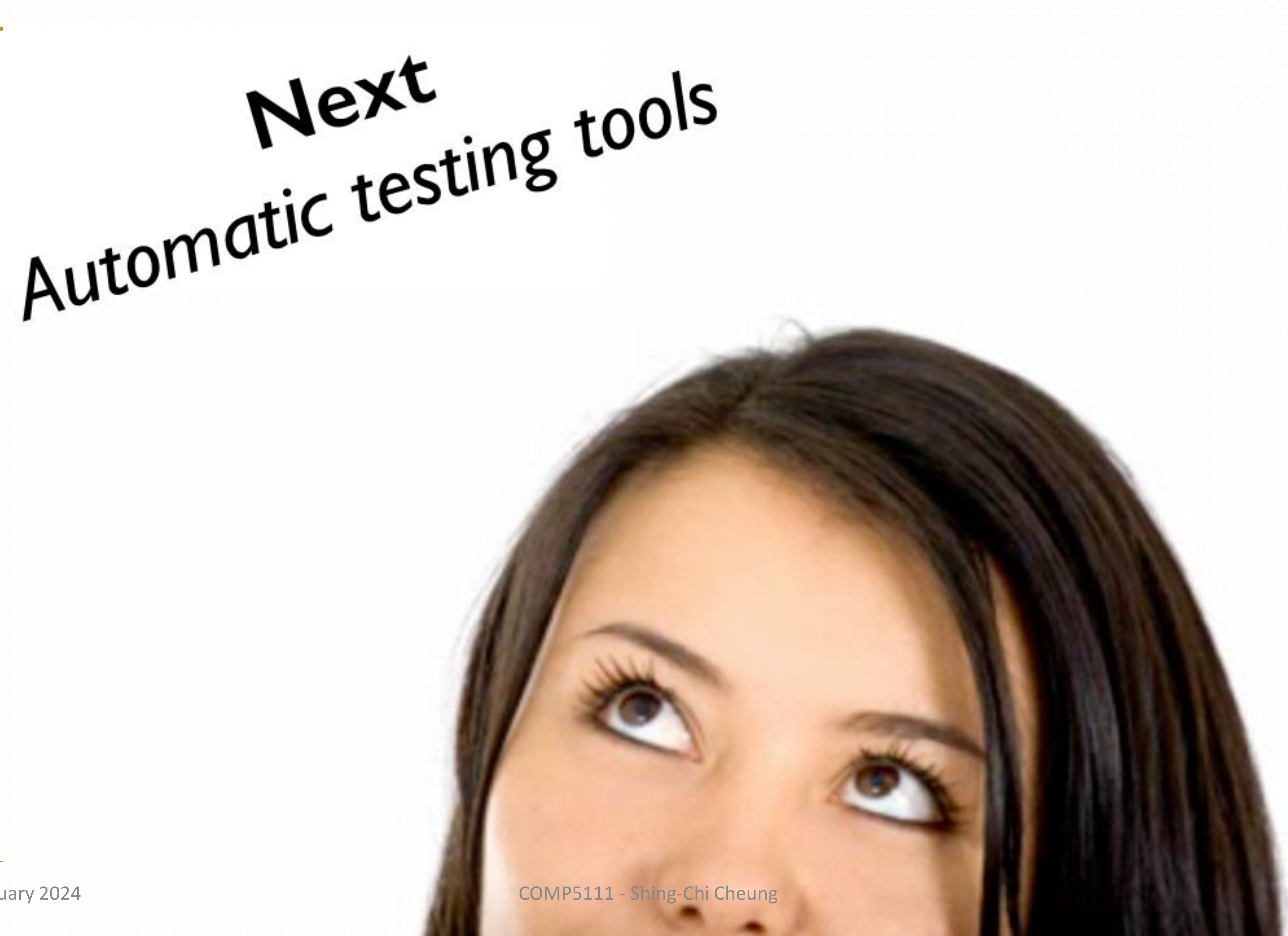
[*C*] ¬*P_i*

Failure point

Failure point

Next

Automatic testing tools



Evosuite

With Dynamic Symbolic Execution Support

Transfer Test Inputs to JUnit Tests

```
public static boolean compare(int a, int b) {
 if (a >= b) {
  return true;
 }
 else {
  return false;
 }
}
```

Transfer Test Inputs to JUnit Tests

```
public static boolean compare(int a, int b) {
 if (a >= b) {
  return true;
 }
 else {
  return false;
 }
}
```

```
@Test(timeout = 4000)
public void test0()  throws Throwable  {
    boolean boolean0 = SimpleProgram.compare(1, 0);
    assertTrue(boolean0);
}
@Test(timeout = 4000)
public void test1()  throws Throwable  {
    boolean boolean0 = SimpleProgram.compare(0, 0);
    assertTrue(boolean0);
}
@Test(timeout = 4000)
public void test2()  throws Throwable  {
    boolean boolean0 = SimpleProgram.compare((-1106), 0);
    assertFalse(boolean0);
}
```

Evosuite

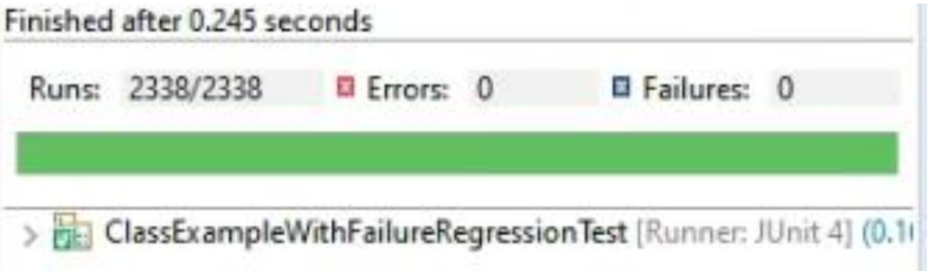
```
public class ClassExampleWithFailure {
        public static int foo(int x, int y) {
                int z = sq(x);
                if (y > 20 && z == 144)
                        assert(false);
                return y*z;
        }
        …
}
```

Evosuite

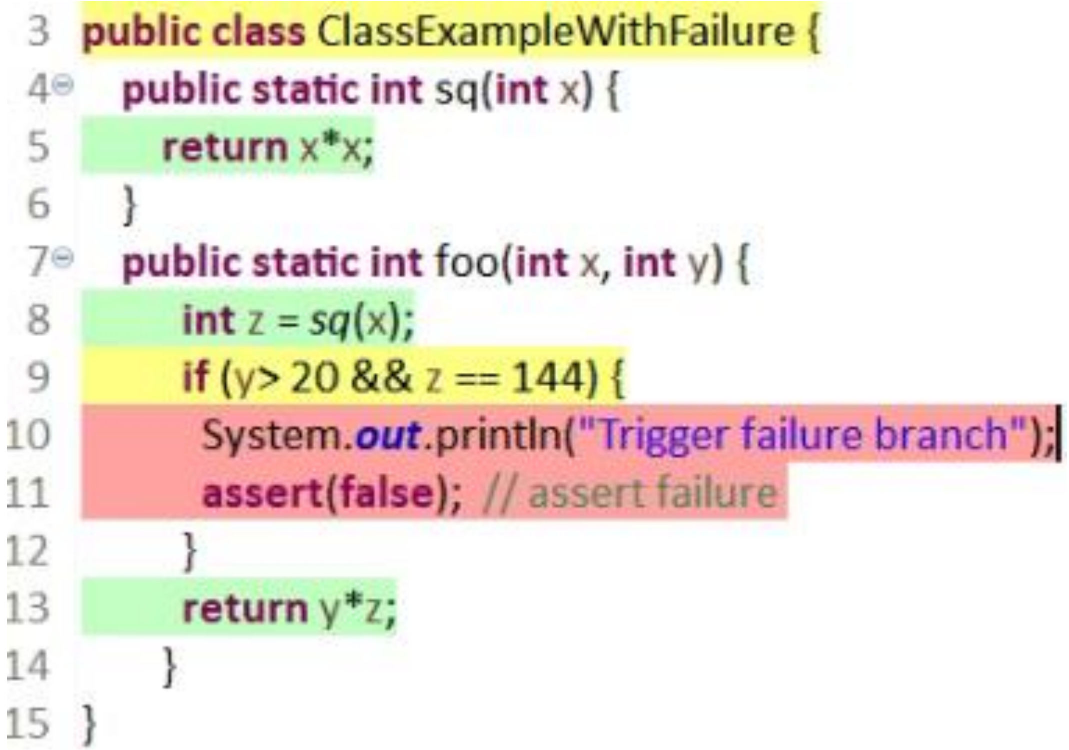
```
public class ClassExampleWithFailure {
        public static int foo(int x, int y) {
                int z = sq(x);
                if (y > 20 && z == 144)
                        assert(false);
                return y*z;
        }
        …
}
```

```
@Test(timeout = 4000)
 public void test6()  throws Throwable  {
   try {
     ClassExampleWithFailure.foo(12, 51);
   } catch(AssertionError e) {
      fail("Expecting exception: AssertionError");
   } // …
 }

 @Test(timeout = 4000)
 public void test7()  throws Throwable  {
   int int0 = ClassExampleWithFailure.foo((-1158), 0);
   assertEquals(0, int0);
 }
```

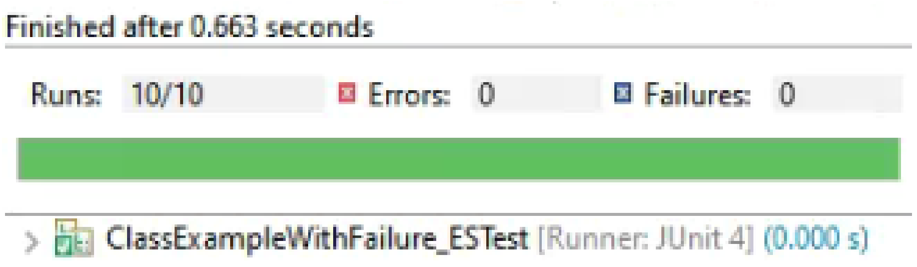


```
Finished after 0.245 seconds
Runs: 2338/2338    Errors: 0    Failures: 0
> ClassExampleWithFailureRegressionTest [Runner: JUnit 4] (0.1
```

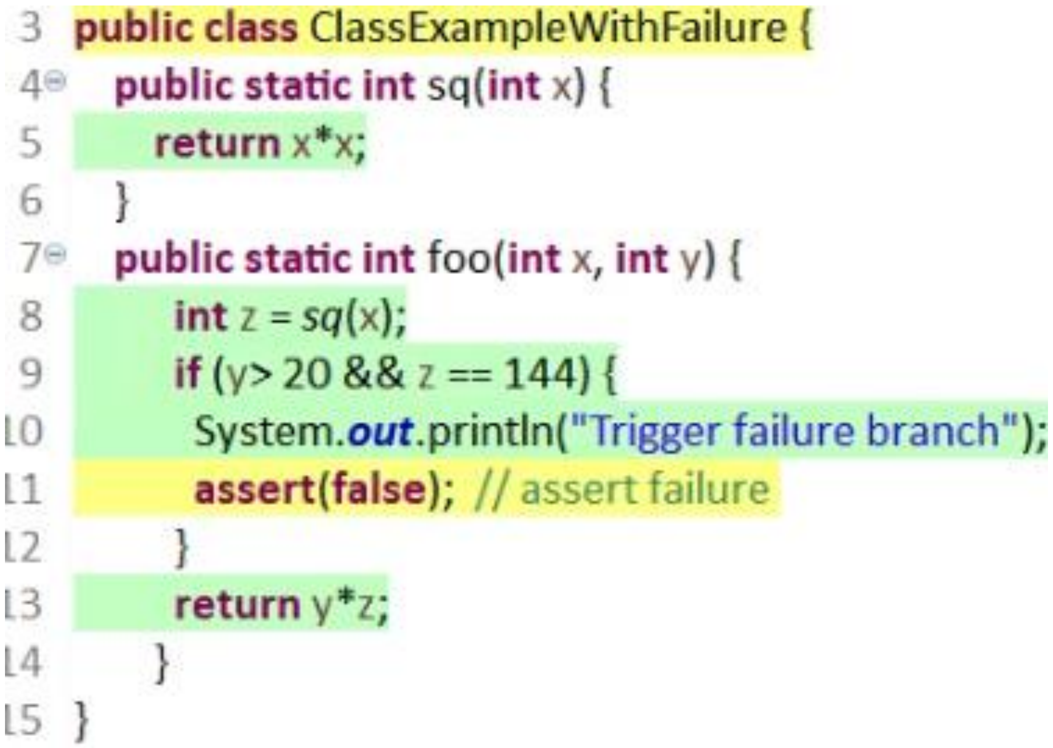


```
public class ClassExampleWithFailure {
  public static int sq(int x) {
    return x*x;
  }
  public static int foo(int x, int y) {
     int z = sq(x);
     if (y> 20 && z == 144) {
      System.out.println("Trigger failure branch");
      assert(false);  // assert failure
     }
     return y*z;
    }
}
```

Coverage by Randoop Generated Tests



```
Finished after 0.663 seconds
Runs: 10/10    Errors: 0    Failures: 0
> ClassExampleWithFailure_ESTest [Runner: JUnit 4] (0.000 s)
```



```
public class ClassExampleWithFailure {
  public static int sq(int x) {
    return x*x;
  }
  public static int foo(int x, int y) {
     int z = sq(x);
     if (y> 20 && z == 144) {
      System.out.println("Trigger failure branch");
      assert(false);  // assert failure
     }
     return y*z;
    }
}
```

Coverage by Evosuite Generated Tests