Construction and Verification of Software

2021 - 2022

MIEI - Integrated Master in Computer Science and Informatics

Consolidation block

Lecture 11 - Dynamic Verification (Testing)

Bernardo Toninho (btoninho@fct.unl.pt)

based on previous editions by **João Seco** and **Luís Caires**

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Part I

Introduction

Testes de Software

"*Testing shows the presence, not the absence of bugs*"

Edsger W. Dijkstra, 1969

Test-based validation

- It consists in executing a program in a **set of pre-determined scenarios** (inputs) and in the observation of its results and effects.
- Tests do not identify the source of bugs in a precise way, they just **signal their existence**. To get to the actual spot in a program we need **debugging techniques**.
- To identify the maximum number of bugs, we need to pick **the right set of inputs**.
- Exhaustive enumeration is not a solution. There are simply too many scenarios to list, execute and compare results.

Testing approaches

How to write tests from the specification of a program

- **Black-Box tests**

  Generating test scenarios based on the specs of the system, without looking at the actual implementation.

  Independent of the implicit development options. The tests and corresponding results can be examined by people not involved in the programming.

- **Glass-Box tests**

  Complement black-box tests with information about the structure of the code and all the options and paths taken.

Part I

Black-Box Testing

Black-Box testing

- Test generation explores the alternatives in the existing specifications of an ADT.

```
// requires x >= 0 && 0.00001 < epsilon && epsilon < 0.001
// ensures x-epsilon <= \result * \result <= x+epsilon
public static double sqrt(double x, double epsilon) {
    …
}
```

- Preconditions are:

```
x > 0  ||  x == 0
0.00001 < epsilon && epsilon < 0.001
```

- All different possibilities are:

```
x > 0  && 0.00001 < epsilon && epsilon < 0.001
x == 0 && 0.00001 < epsilon && epsilon < 0.001
```

- For each possibility, you pick a pair of values that satisfy the conditions and check for the post condition.

Black-Box testing

- Test generation explores the alternatives in the existing specifications of an ADT.

```
// requires n >= 1
// ensures if n is Prime \result == true else \result == false
public static boolean isPrime(int n) {
    …
}
```

- Preconditions are:

```
n = 1 || n > 1
```

- All different possibilities are:

```
any n where n is prime
any n where n is not prime
```

- For each possibility, you pick a pair of values that satisfy the conditions and check for the post condition.

Black-Box testing

- Test generation explores the alternatives in the existing specifications of an ADT.

```
// requires true
// ensures if a is null throws NullPointerException else
//         if x is in a, a[\result] == x, else throws NotFoundException
public static int indexOf(int[] a, int x)
    throws NullPointerException, NotFoundException { … }
```

- Preconditions are:

```
true
```

- All different possibilities are:

```
a == null and any x
any a, x where x is in a
any a, x where x is not in a
```

- For each possibility, you pick a pair of values that satisfy the conditions and check for the post condition.

Black-Box testing

1. Identify disjunctions in preconditions. List all scenarios with values that cover all combinations of valid preconditions

2. Identify disjunctions in postconditions. List all scenarios that produce results for all combinations of valid post-combinations (including exceptions)

3. Test special border conditions (declared limits, 0, 1, 2, many for collections). Test overflow conditions in arithmetic operations.

4. Test aliasing cases because usually developers assume that arguments are independent (recall Separation Logic).

Exercise

- Consider a multi-set implemented with a linked-list (with repetitions).
- Write the specifications for all the methods
- Write black-box tests for those specifications

```
public class Bag {

    private class Node {
        private int x;
        private Node next;

        Node(int x, Node next) {
            this.x = x;
            this.next = next;
        }
    }

    private Node head;

    …
```

```
    …

    public Bag() {
        super();
        this.head = null;
    }

    public void add(int x) { … }

    public void remove(int x) { … }

    public int get(int x) { … }

}
```

using junit…

```
// requires true
// ensures for all x: get(x) == 0
public Bag() {
    super();
    this.head = null;
}

@Test
public void testCtor() {
    Bag b = new Bag();
    Assertions.assertTrue(b.get(0) == 0);
    Assertions.assertTrue(b.get(1) == 0);
}
```

- Impossible to cover all possible cases in black box tests, but it seems reasonable to admit that two different elements are enough.

using junit…

```
// requires true
// ensures  get(x) == old(get(x)) + 1
// ensures  if x /= y and add(x) then get(y) == old(get(y))
public void add(int x) {
    this.head = new Node(x,head);
}
```

using junit...

```
// requires true
// ensures  get(x) == old(get(x)) + 1
// ensures  if x /= y and add(x) then get(y) == old(get(y))
public void add(int x) {
    this.head = new Node(x,head);
}
@Test
public void testAdd1() {
    Bag b = new Bag(); b.add(1);
    Assertions.assertTrue(b.get(1) == 1);
}
```

using junit...

```
// requires true
// ensures  get(x) == old(get(x)) + 1
// ensures  if x /= y and add(x) then get(y) == old(get(y))
public void add(int x) {
    this.head = new Node(x,head);
}
@Test
public void testAdd1() {
    Bag b = new Bag(); b.add(1);
    Assertions.assertTrue(b.get(1) == 1);
}
@Test
public void testAdd2() {
    Bag b = new Bag(); b.add(1); b.add(1);
    Assertions.assertTrue(b.get(1) == 2);
}
```

using junit...

```
// requires true
// ensures  get(x) == old(get(x)) + 1
// ensures  if x /= y and add(x) then get(y) == old(get(y))
public void add(int x) {
    this.head = new Node(x,head);
}
@Test
public void testAdd1() {
    Bag b = new Bag(); b.add(1);
    Assertions.assertTrue(b.get(1) == 1);
}
@Test
public void testAdd2() {
    Bag b = new Bag(); b.add(1); b.add(1);
    Assertions.assertTrue(b.get(1) == 2);
}
@Test
public void testAdd3() {
    Bag b = new Bag(); b.add(1); b.add(1); b.add(2);
    Assertions.assertTrue(b.get(1) == 2);
}
```

using junit...

```
// requires true
// ensures if old(get(x)) == 0 then get(x) == 0
// ensures if old(get(x)) > 0 then get(x) == old(get(x))-1
public void remove(int x) {
    Node temp = head;
    Node prev = null;
    while(temp != null && temp.x != x) {
        prev = temp;
        temp = temp.next;
    }
    if( temp != null ) {
        assert temp.x == x;
        if( prev != null )
            prev.next = temp.next;
        else
            head = temp.next;
    }
}
```

using junit…

```
// requires true
// ensures if old(get(x)) == 0 then get(x) == 0
// ensures if old(get(x)) > 0 then get(x) == old(get(x))-1
public void remove(int x) { … }
@Test
public void testRemove1() {
    Bag b = new Bag(); b.remove(1);
    Assertions.assertTrue(b.get(1) == 0);
}
```

using junit…

```
// requires true
// ensures if old(get(x)) == 0 then get(x) == 0
// ensures if old(get(x)) > 0 then get(x) == old(get(x))-1
public void remove(int x) { … }
@Test
public void testRemove1() {
    Bag b = new Bag(); b.remove(1);
    Assertions.assertTrue(b.get(1) == 0);
}
@Test
public void testRemove2() {
    Bag b = new Bag(); b.add(1); b.remove(1);
    Assertions.assertTrue(b.get(1) == 0);
}
```

using junit…

```
// requires true
// ensures if old(get(x)) == 0 then get(x) == 0
// ensures if old(get(x)) > 0 then get(x) == old(get(x))-1
public void remove(int x) { … }
@Test
public void testRemove1() {
    Bag b = new Bag(); b.remove(1);
    Assertions.assertTrue(b.get(1) == 0);
}
@Test
public void testRemove2() {
    Bag b = new Bag(); b.add(1); b.remove(1);
    Assertions.assertTrue(b.get(1) == 0);
}
@Test
public void testRemove3() {
    Bag b = new Bag(); b.add(1); b.add(1); b.remove(1);
    Assertions.assertTrue(b.get(1) == 1);
}
```

using junit…

```
// requires true
// ensures if old(get(x)) == 0 then get(x) == 0
// ensures if old(get(x)) > 0 then get(x) == old(get(x))-1
public void remove(int x) { … }
@Test
public void testRemove1() {
    Bag b = new Bag(); b.remove(1);
    Assertions.assertTrue(b.get(1) == 0);
}
@Test
public void testRemove2() {
    Bag b = new Bag(); b.add(1); b.remove(1);
    Assertions.assertTrue(b.get(1) == 0);
}
@Test
public void testRemove3() {
    Bag b = new Bag(); b.add(1); b.add(1); b.remove(1);
    Assertions.assertTrue(b.get(1) == 1);
}
@Test
public void testRemove4() {
    Bag b = new Bag(); b.add(1); b.add(1); b.add(2); b.remove(2);
    Assertions.assertTrue(b.get(1) == 2 && b.get(2) == 0);
}
```

using junit…

```
// requires true
// ensures if old(get(x)) == 0 then get(x) == 0
// ensures if old(get(x)) > 0 then get(x) == old(get(x))-1
public void remove(int x) { … }
@Test
public void testRemove1() {
    Bag b = new Bag(); b.remove(1);
    Assertions.assertTrue(b.get(1) == 0);
}
@Test
public void testRemove2() {
    Bag b = new Bag(); b.add(1); b.remove(1);
    Assertions.assertTrue(b.get(1) == 0);
}
@Test
public void testRemove3() {
    Bag b = new Bag(); b.add(1); b.add(1); b.remove(1);
    Assertions.assertTrue(b.get(1) == 1);
}
@Test
public void testRemove4() { // if x /= y then remove(x) ==> old(get(y)) == get(y)
    Bag b = new Bag(); b.add(1); b.add(1); b.add(2); b.remove(2);
    Assertions.assertTrue(b.get(1) == 2 && b.get(2) == 0);
}
```

using junit

- All specifications are based on other available operations, it's not possible to use abstract states that are not really implemented. Even get must be tested…

```
// requires true
// ensures if x was added n times then get(x) == n
// ensures if x was not added then get(x) == 0
// ensures if get(x) == n and add others then get(x) == n
public int get(int x) {
    Node temp = head;
    int count = 0;
    while( temp != null ) {
        if (temp.x == x) count++;
        temp = temp.next;
    }
    return count;
}
```

Black-box testing

- Black-box testing is not guaranteed to not cover all cases, all possible executions, not even all lines of code.
- Coverage tools are already shipped with IDEs (e.g. Eclipse and IntelliJ)

```
public class Bag {

    private class Node {
        private int x;
        private Node next;

        Node(int x, Node next) {
            this.x = x;
            this.next = next;
        }
    }

    private Node head;

    // requires true
    // ensures for all x: get(x) == 0
    public Bag() {
        super();
        this.head = null;
    }

    // requires true
    // ensures  get(x) == old(get(x)) + 1
    public void add(int x) {
        this.head = new Node(x,head);
    }

    // requires true
    // ensures if old(get(x)) == 0 then get(x) == 0
    // ensures if old(get(x)) > 0 then get(x) == old(get(x))-1
    public void remove(int x) {
        Node temp = head;
        Node prev = null;
        while(temp != null && temp.x != x) {
            prev = temp;
            temp = temp.next;
        }
        if( temp != null ) {
            assert temp.x == x;
            if( prev != null )
                prev.next = temp.next;
            else
                head = temp.next;
        }
    }

    // requires true
    // ensures if x was added n times then get(x) == n
    // ensures if x was not added then get(x) == 0
    // ensures if get(x) == n and add others then get(x) == n
    public int get(int x) {
        Node temp = head;
        int count = 0;
        while( temp != null ) {
            if (temp.x == x) count++;
            temp = temp.next;
        }
        return count;
    }
}
```

Construction and Verification of Software

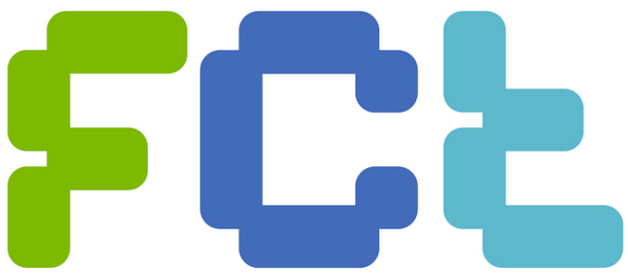
~~2019 – 2020~~ 2020 - 2021

MIEI - Integrated Master in Computer Science and Informatics

Consolidation block

Lecture 12 - Part III - Glass-box testing

João Costa Seco (joao.seco@fct.unl.pt)



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Part III

Glass-Box Testing

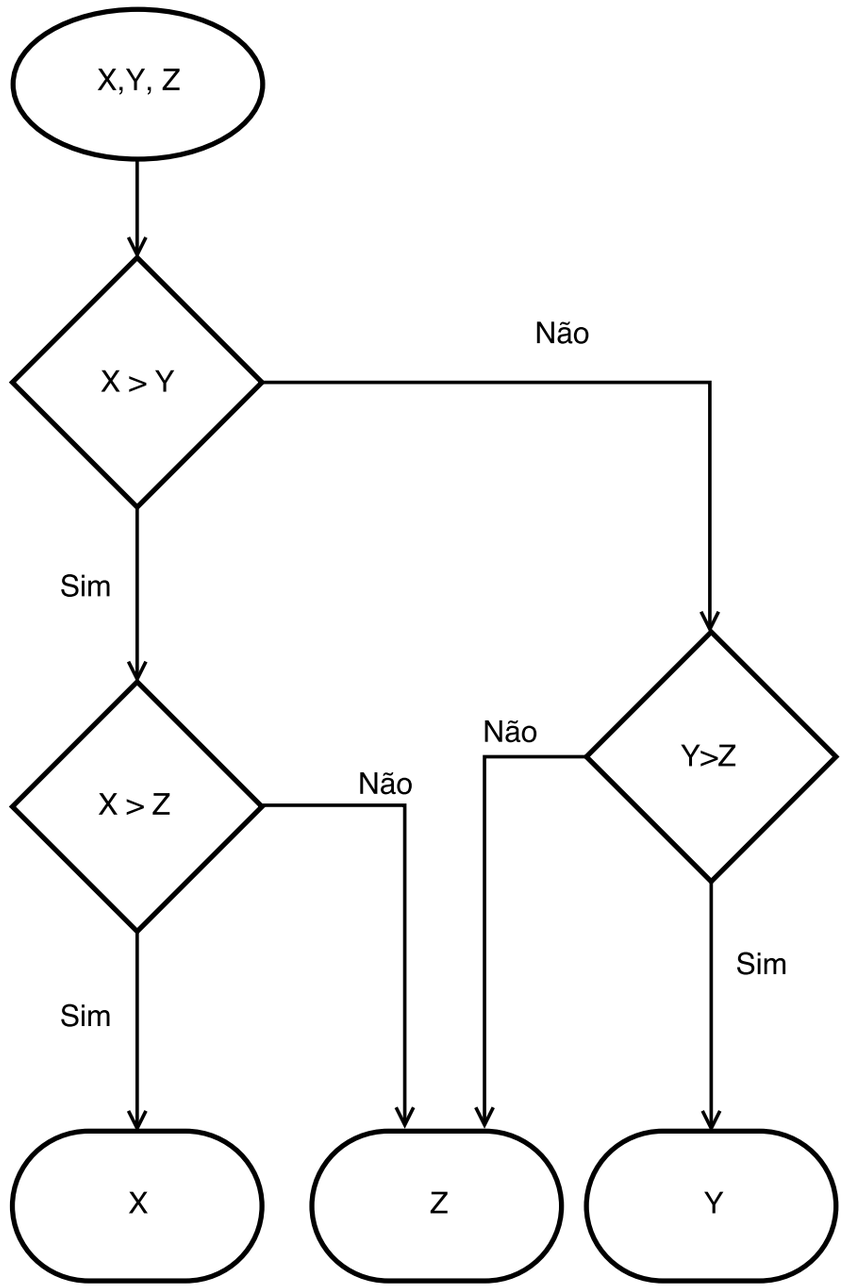
Glass-Box Testing

- Glass-box testing complements the testing process based on the specifications, and looking at the actual code.
- The goal is to test each branch of the control graph at least once.
- **Myth**: Glass-box testing explores all possible paths.
- **(Sad) Truth**: developers tend to ignore specs which introduces a dangerous bias.
- To explore all paths does not ensure correctness. There are data dependencies of actual values being used.
- To ensure that each part of the CFG is to exercise all decision nodes of the program.

Glass-Box Testing

- There are N^3 possible scenarios for this function, and only 4 possible paths in the graph. There is not much information for black-box testing.

```
// requires true
// ensures \result >= x
// ensures \result >= y
// ensures \result >= z
public int maxOfThree(int x,
                      int y,
                      int z) {
    if( x > y )
        if( x > z )
            return x;
        else
            return z;
    else
        if( y > z )
            return y;
        else
            return z;
}
```



Glass-Box Testing

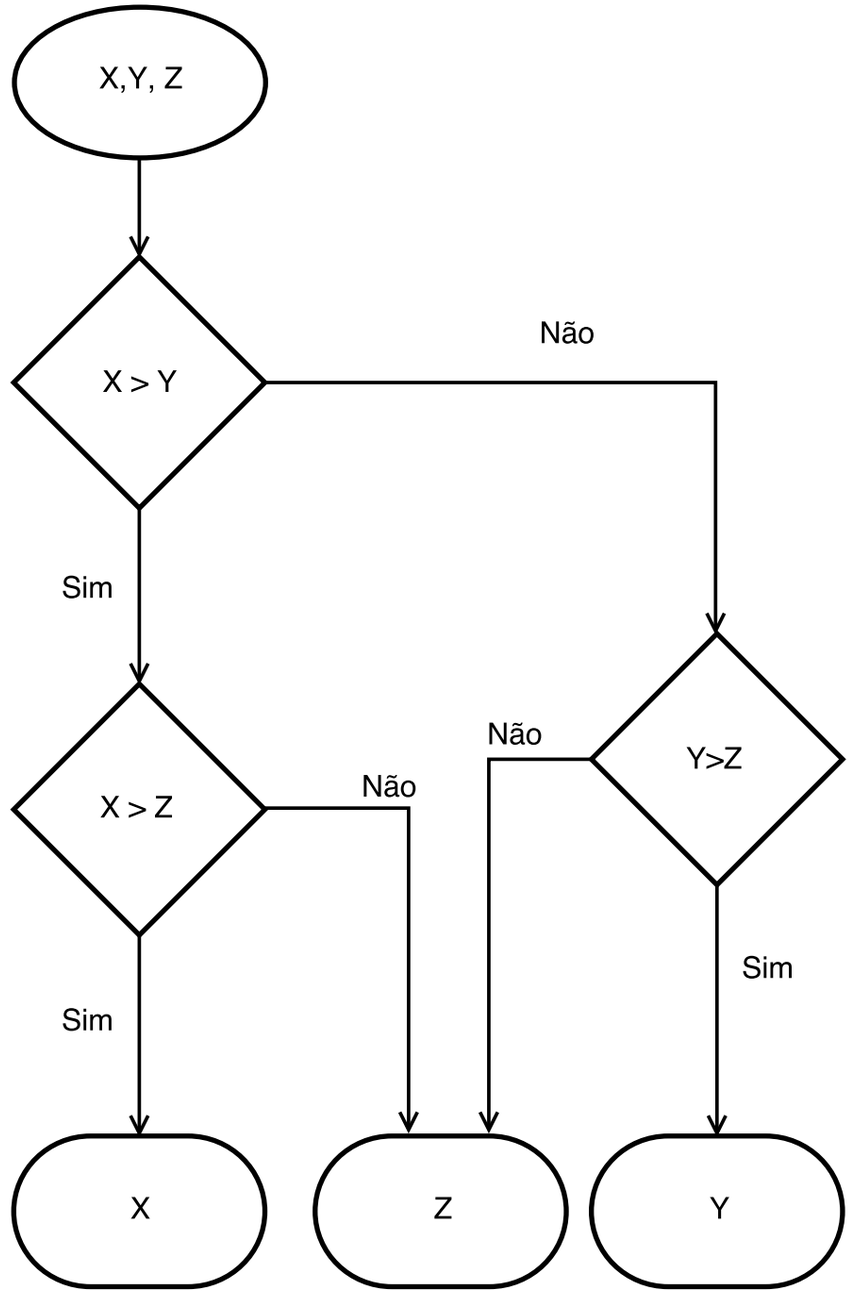
- N^3 possible scenarios, only 4 possible paths.
- These tests are “path-complete”:

```
maxOfThree(10,1,2) = 10
maxOfThree(10,1,15) = 15
maxOfThree(2,10,15) = 15
maxOfThree(2,10,3) = 10
```

- They test the relevant cases, identify the border cases, etc.

```
public int maxOfThree(int x, int y, int z) {
  return x;
}
```

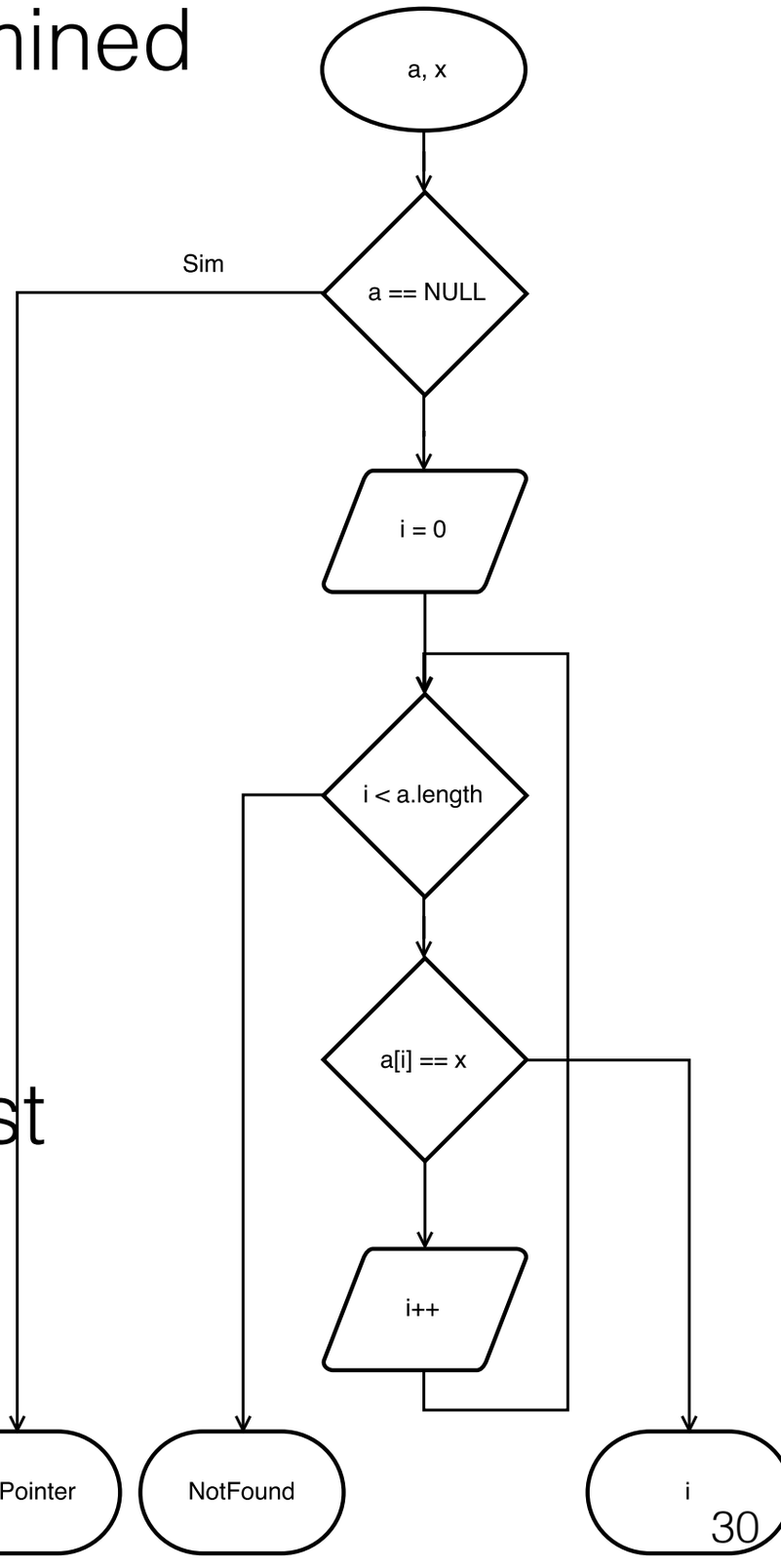
- This implementation has only one path and the test (2,1,1) passes.



Glass-Box Testing

- When using iteration there is an undetermined number of possible paths.

```
// requires true
// ensures if a is null throws NullPointerException else
//         if x is in a, a[\result] == x,
//         else throws NotFoundException
public static int indexOf(int[] a, int x)
  throws NullPointerException, NotFoundException {
   if( a == null )
      throw new NullPointerException();
   for(int i = 0; i < a.length; i++)
      if( a[i] == x ) return i;
   throw new NotFoundException();
}
```

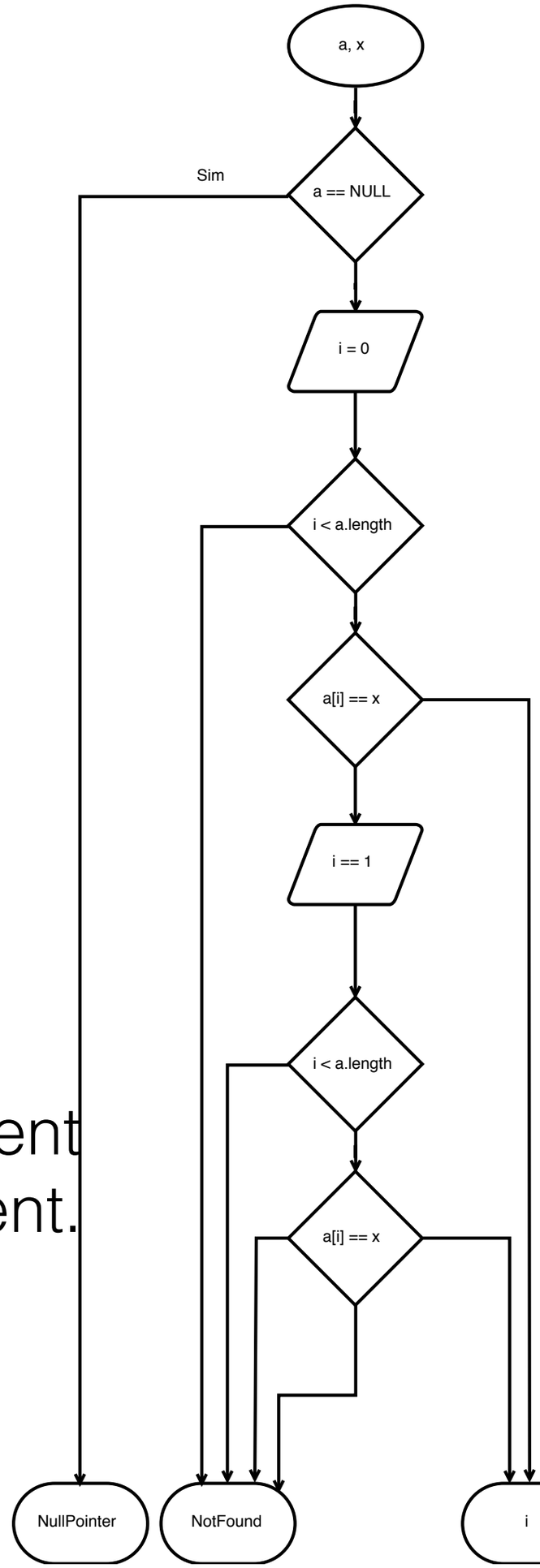


- Tests should run the body of loops at least twice (to ensure that the loop invariant is maintained by the loop body).

Glass-Box Testing

- Besides the spec based tests, the CFG indicates 7 more tests.

```
// requires true
// ensures if a is null throws NullPointerException else
//         if x is in a, a[\result] == x,
//         else throws NotFoundException
public static int indexOf(int[] a, int x)
  throws NullPointerException, NotFoundException {
   if( a == null )
      throw new NullPointerException();
   for(int i = 0; i < a.length; i++)
      if( a[i] == x ) return i;
   throw new NotFoundException();
}
```



- An array with 2 positions, the searched element in the first position, in the last, and non-existent.
- Test with size 0 and size 1.
- Test the exceptions.

Glass-Box Testing

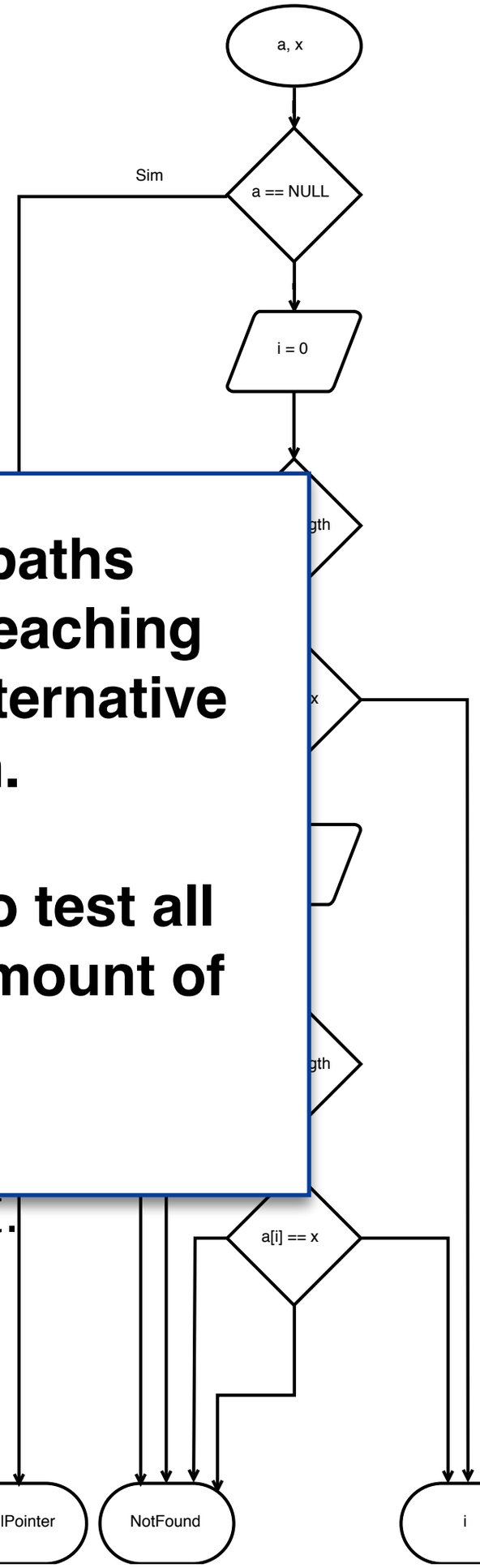
- Besides the spec based tests, the CFG indicates 7 more tests.

```
// requires true
// ensures i
//         i
//         e
public stati
  throws Nul
   if( a ==
      throw
   for(int i
      if( a[
   throw new
}
```

Not always the number of possible paths corresponds to the number of edges reaching terminator nodes. At times, there are alternative paths in the middle of the graph.

Tests are approximations. In general, to test all paths is not possible in a reasonable amount of time.

- An arra
  in the first position, in the last, and non-existent.
- Test with size 0 and size 1.
- Test the exceptions.



Summary: dealing with Loops

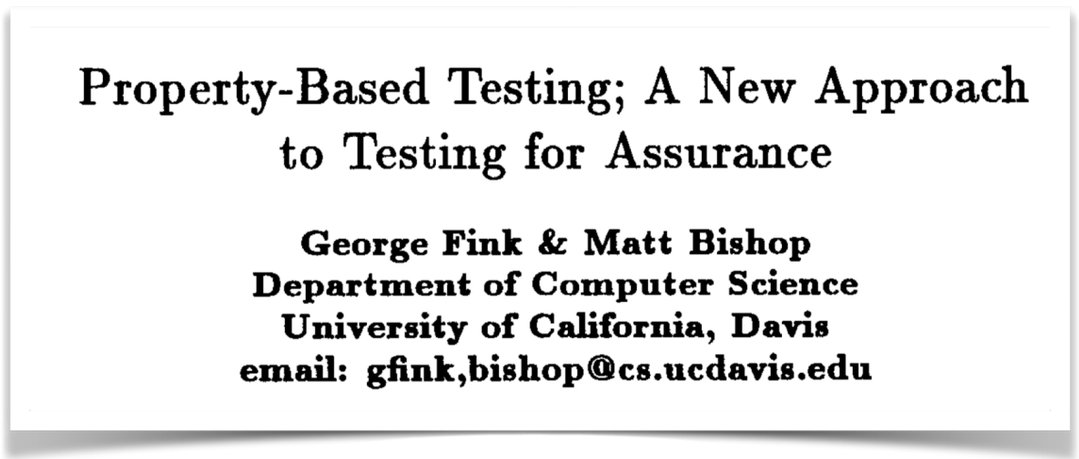
- Loops with well determined number of iterations (for loops) are ran at least twice so that all conditions are tested and the invariant is tested in the end of the loop body (as input for the second operation).
- For loops with undetermined number of iterations, loops must be tested in zero, one and two iterations, and all exiting conditions.
- For recursive functions, we should test the base case and the inductive case. Tests with none or one recursive call, and all return possibilities.

Part IV

Property Based Testing

Property Based Testing

- What is PBT? “The thing that QuickCheck does”
- “Property-based testing uses property specifications and a data-flow analysis of the program to guide evaluation of test executions for correctness and completeness.”
- Instead of producing individual tests by hand, with the spec in mind, one uses value generators, and constraints on values to feed test scenarios to the program.


Property-Based Testing; A New Approach to Testing for Assurance

George Fink & Matt Bishop
Department of Computer Science
University of California, Davis
email: gfink,bishop@cs.ucdavis.edu

https://hypothesis.works/articles/what-is-property-based-testing/

https://medium.com/criteo-labs/introduction-to-property-based-testing-f5236229d237

JUnit

```
public class FindATriangle {

    public static Triangle isThisATriangle(int a, int b, int c)
        throws NotATriangle
    {
        if( a <= 0 || b <= 0 || c <= 0 ||
            !( a + b > c && a + c > b && b + c > a) ) {

            throw new NotATriangle();
        }

        if ( a == b && a == c)
            return Triangle.EQUILATERAL;
        else if( a == b || a == c || b == c)
            return Triangle.ISOSCELES;
        else
            return Triangle.SCALENE;
    }

}
```

remember this?

JUnit

```
public class FindATriangle {

    public static Triangle isThisATriangle(int a, int b, int c)
        throws NotATriangle
{
}
```

```
@Test
public void EqualSidesShouldBeEquilateral() {
    try {
        assertTrue(FindATriangle.isThisATriangle(3,3,3) == Triangle.EQUILATERAL );
    } catch (NotATriangle notATriangle) {
        fail();
    }

@Test
public void FailureZero() {
    try {
        FindATriangle.isThisATriangle(0, 10, 10);
    } catch (NotATriangle notATriangle) {
        assertTrue(true);
    }
}
```

You know how hard it was to produce a list of convincing tests in the first lecture.

One step ahead: Parametrized tests in JUnit 5

- Being able to automatised many test cases in a single definition.

```
public class Numbers {
    public static boolean isOdd(int number) {
        return number % 2 != 0;
    }
}
```

```
@ParameterizedTest
@ValueSource(ints = {1, 3, 5, -3, 15, Integer.MAX_VALUE}) // six numbers
void isOdd_ShouldReturnTrueForOddNumbers(int number) {
    assertTrue(Numbers.isOdd(number));
}
```

```
public class Strings {
    public static boolean isBlank(String input) {
        return input == null || input.trim().isEmpty();
    }
}
```

```
@ParameterizedTest
@ValueSource(strings = {"", "  "})
void isBlank_ShouldReturnTrueForNullOrBlankStrings(String input) {
    assertTrue(Strings.isBlank(input));
}
```


source: https://www.baeldung.com/parameterized-tests-junit-5

Two steps ahead: Quickcheck

QuickCheck:
A Lightweight Tool for Random Testing of Haskell Programs

Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

John Hughes
Chalmers University of Technology
rjmh@cs.chalmers.se

Quickcheck

Qui
A Lightweight To
of Hask

Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

ogy

Language	Library
C	theft
C++	CppQuickCheck
Clojure	test.check
Coq	QuickChick
F#	FsCheck
Go	gopter
Haskell	Hedgehog
Java	QuickTheories
JavaScript	jsverify
PHP	Eris
Python	Hypothesis
Ruby	Rantly
Rust	Quickcheck
Scala	ScalaCheck
Swift	Swiftcheck

Quickcheck

QuickCheck:
A Lightweight Tool for Random Testing of Haskell Programs

Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

John Hughes
Chalmers University of Technology
rjmh@cs.chalmers.se

```
@Property
public void TwoEqualSidesShouldBeISOSCELES(@InRange(minInt = 1) int a, @InRange(minInt = 1) int b) {

    assumeThat(a,not(equalTo(b)));
    assumeThat(2*a,not(lessThan(b)));

    try {
        assertThat(FindATriangle.isThisATriangle(a,b,a), equalTo(Triangle.ISOSCELES) );
    } catch (NotATriangle notATriangle) {
        fail();
    }
}
```

Quickcheck

QuickCheck:
A Lightweight Tool for Random Testing of Haskell Programs

Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

John Hughes
Chalmers University of Technology
rjmh@cs.chalmers.se

```
@Property
public void DifferentSidesShouldBeSCALENE(
    @InRange(minInt = 1) int a,
    @InRange(minInt = 1) int b,
    @InRange(minInt = 1) int c) {

    assumeThat(a,not(equalTo(b)));
    assumeThat(a,not(equalTo(c)));
    assumeThat(b,not(equalTo(c)));

   try {
        assertThat(FindATriangle.isThisATriangle(a,b,c), equalTo(Triangle.SCALENE) );
    } catch (NotATriangle notATriangle) {
        fail();
    }
}
```

Is this OK??

Quickcheck

- We need to feed (in)valid triangles to the tests, not only three numbers…

```
public class TriGen extends Generator<Triple> {

    public TriGen() {
        super(Triple.class);
    }

    @Override
    public Triple generate(SourceOfRandomness sourceOfRandomness,
                           GenerationStatus generationStatus) {
        int a = sourceOfRandomness.nextInt(1,Integer.MAX_VALUE/3);
        int b = sourceOfRandomness.nextInt(1,Integer.MAX_VALUE/3);
        int c = a + b - 10;

        return Triple.createRandom(sourceOfRandomness, a,b,c);
    }
}
```

Quickcheck

- We need to feed (in)valid triangles to the tests, not only three numbers…

```
public class NonTriGen extends Generator<Triple> {

    public NonTriGen() {
        super(Triple.class);
    }

    @Override
    public Triple generate(SourceOfRandomness sourceOfRandomness,
                           GenerationStatus generationStatus) {
        int a = sourceOfRandomness.nextInt(1,Integer.MAX_VALUE/3);
        int b = sourceOfRandomness.nextInt(1,Integer.MAX_VALUE/3);
        int c = a + b + sourceOfRandomness.nextInt(0, Integer.MAX_VALUE/3);

        return Triple.createRandom(sourceOfRandomness, a, b, c);
    }
}
```

Quickcheck

- We need to feed (in)valid triangles to the tests, not only three numbers…

```
@Property
public void SidesMustSumUp1(@From(TriGen.class) Triple t) {

    Exception e = null;
    try {
        FindATriangle.isThisATriangle(t.a,t.b,t.c);
    } catch (NotATriangle notATriangle) {
        e = notATriangle;
    }
    assertThat(e,nullValue());
}

@Property
public void SidesMustSumUp2(@From(NonTriGen.class) Triple t) {

    Exception e = null;
    try {
        FindATriangle.isThisATriangle(t.a,t.b,t.c);
    } catch (NotATriangle notATriangle) {
        e = notATriangle;
    }
    assertThat(e,notNullValue());
}
```

Fuzzing - Not really PBT

- “Fuzzing is feeding a piece of code (function, program, etc.) data from a large corpus, possibly dynamically generated, possibly dependent on the results of execution on previous data, in order to see whether it fails.” David R. MacIver.

https://hypothesis.works/articles/what-is-property-based-testing/

- It is based on gradually mutating random inputs to stress test an application with different inputs.

- AFL (American Fuzzy Loop) approaches are association fuzzing with inspection of program traces to increase the coverage of tests (with genetic algorithms).

Summary

- Property based testing advocates the generation of a large number of tests from a declarative specification.
- Specifications (as the ones used in Dafny/Verifast) can be used as guides to write the properties that must be preserved by operations being tested.
- Quickcheck randomises and uses generators to build many and good inputs for tests, based on properties.
- Fuzzing extends that with a high volume of mutation and randomisation of inputs.
- Automatization of tests is needed, but high-level specification of tests and inputs is essential.