Construction and Verification of Software

2017 - 2018

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 10 - Dynamic validation (Testing)

João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

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

- **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.

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 specifications.

```
   // 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 specifications.

```
// 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

```java
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;

    …
```

```java
    …

    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() { // 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…

```
// 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);
}
```

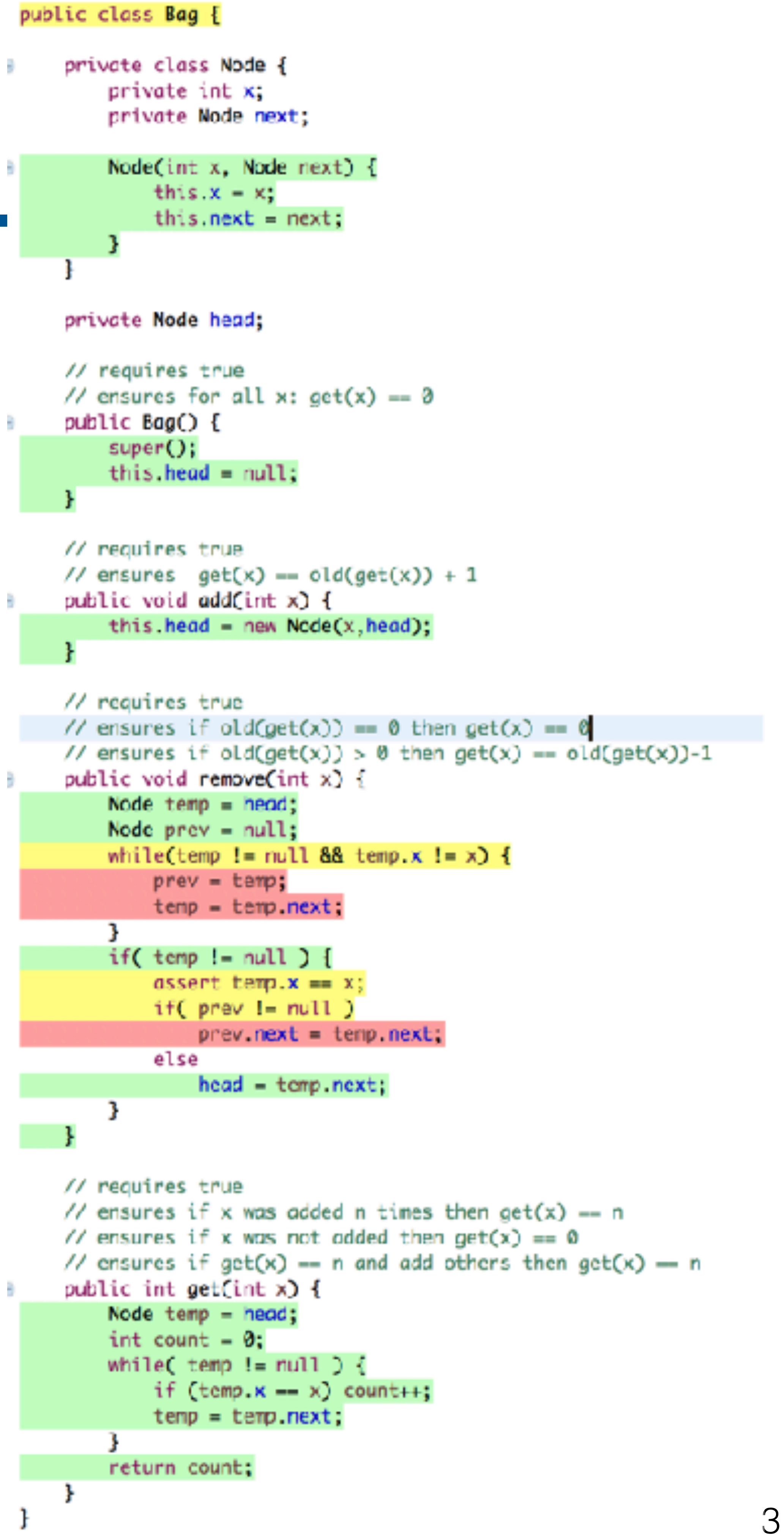
using junit

- 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;
    }
}
```

Glass-box testing

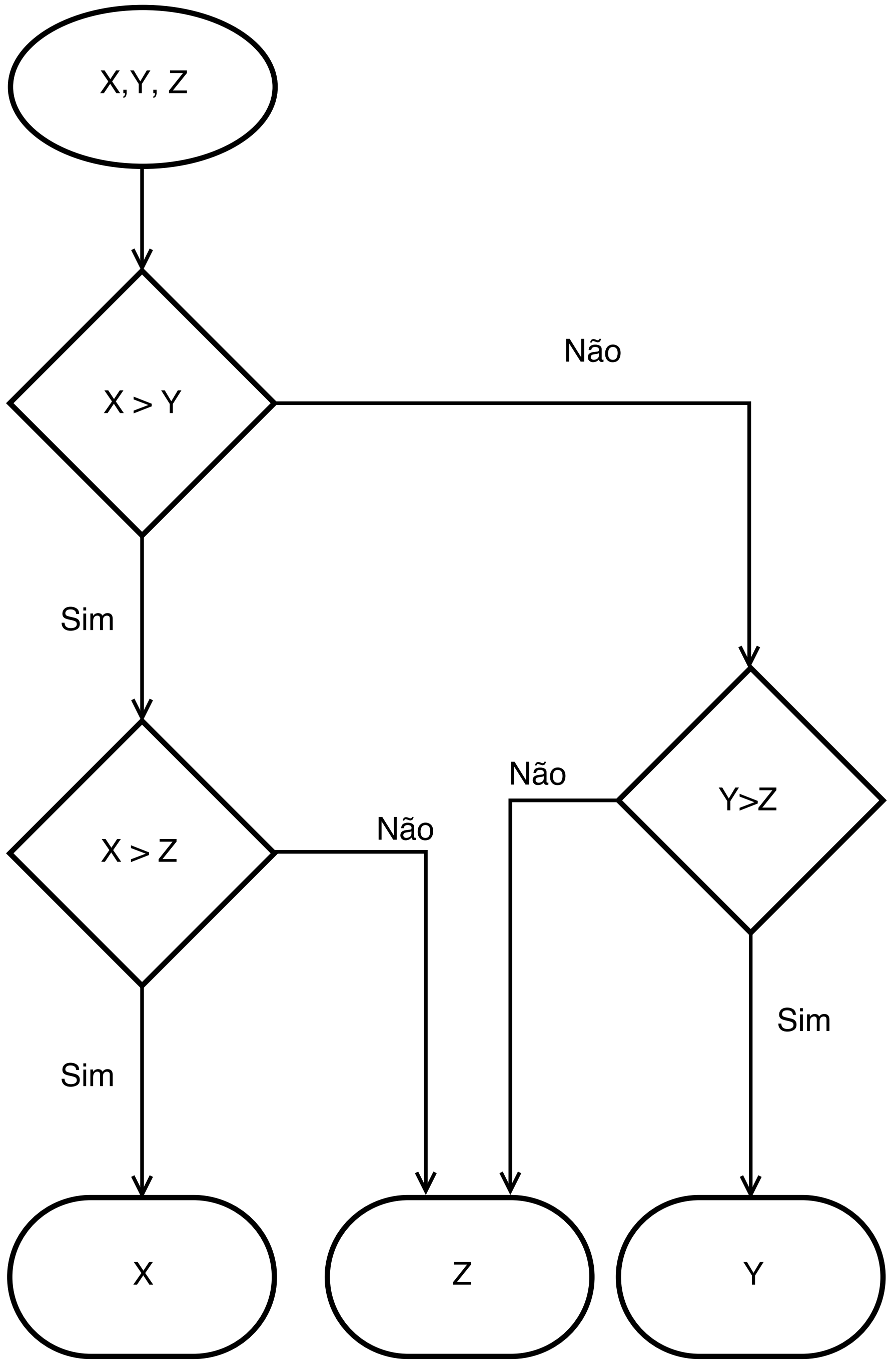
Glass-Box Testing

- 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

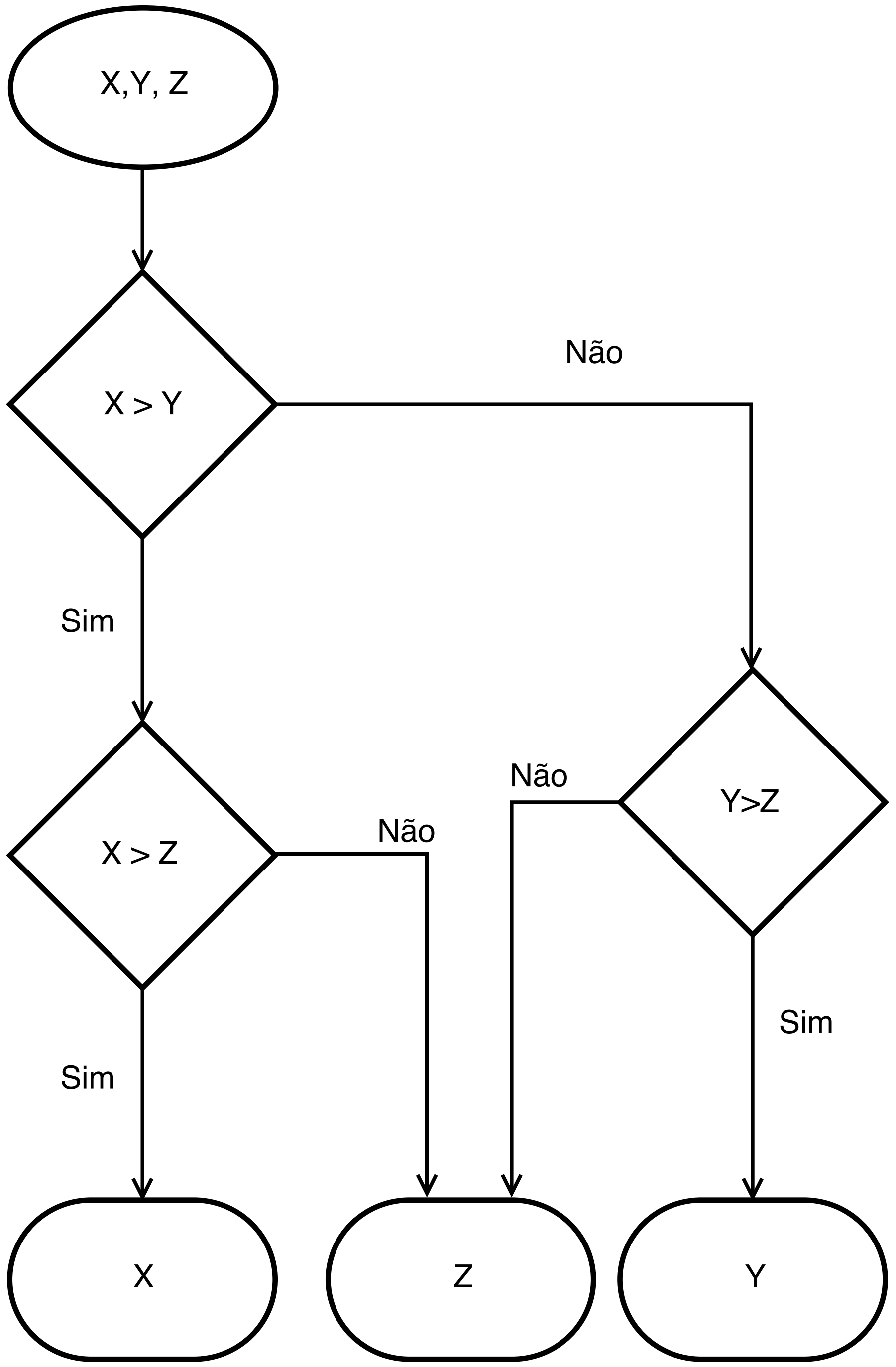
- N^3 possible scenarios, only 4 possible paths.
- These tests are “path-complete”:

```
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.

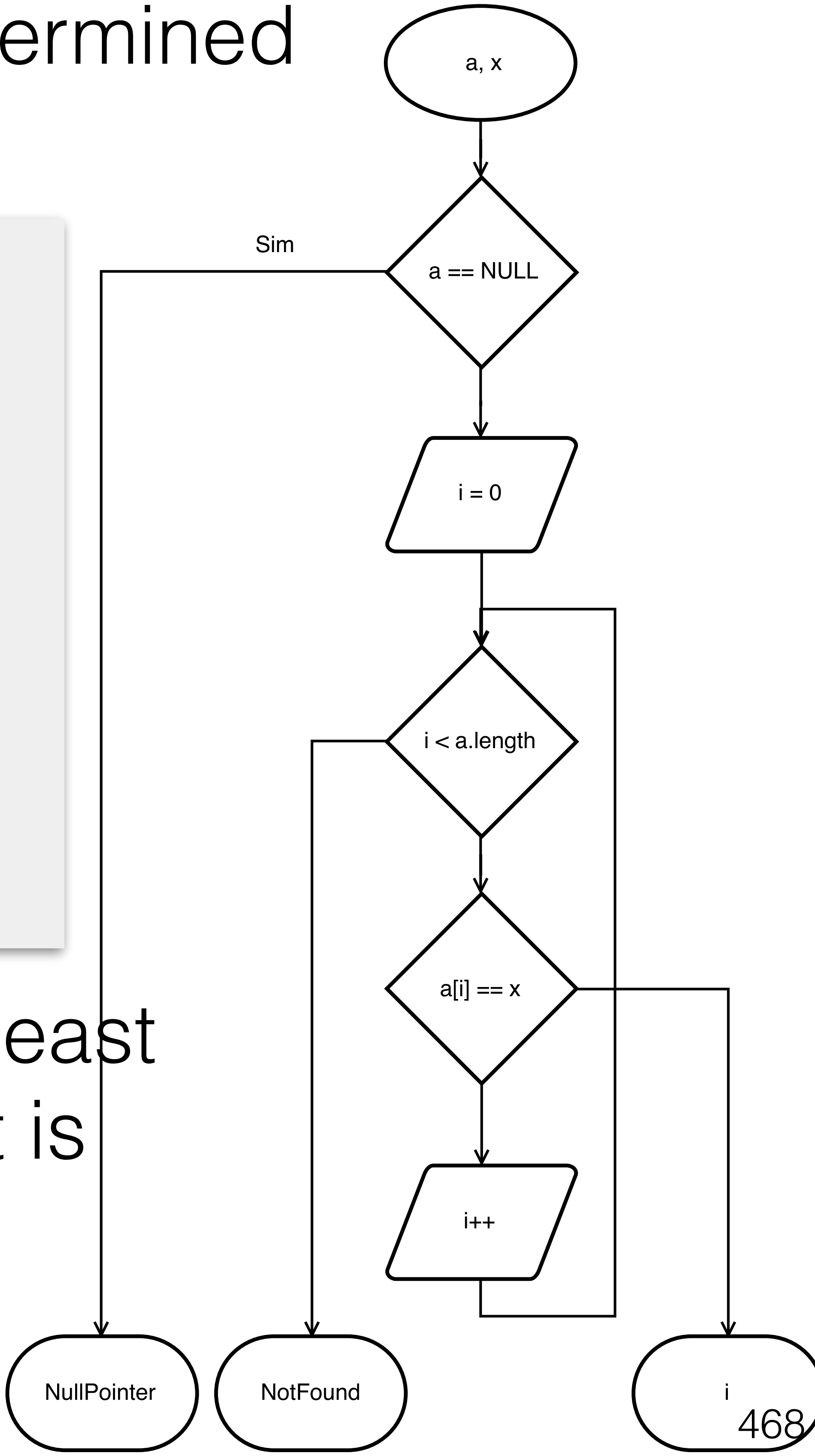


Glass-Box Testing

- 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).

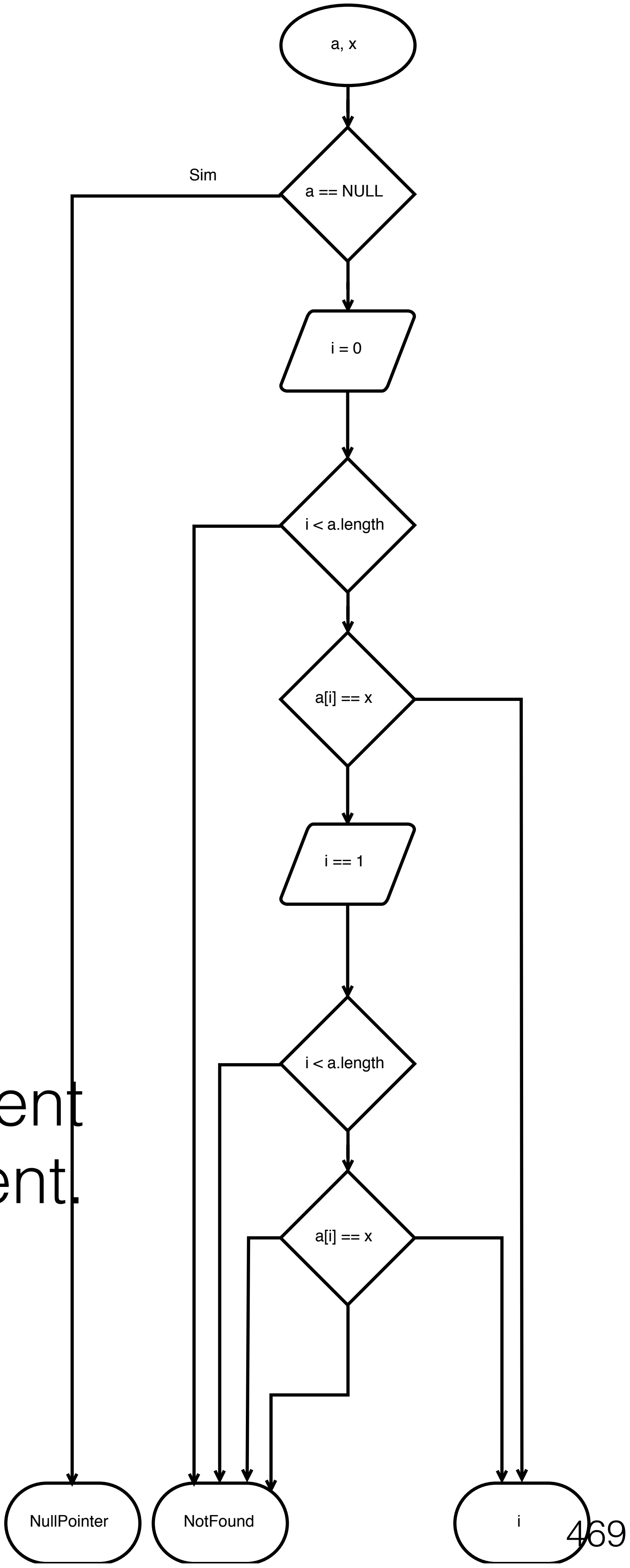


Glass-Box Testing

- 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
}
```

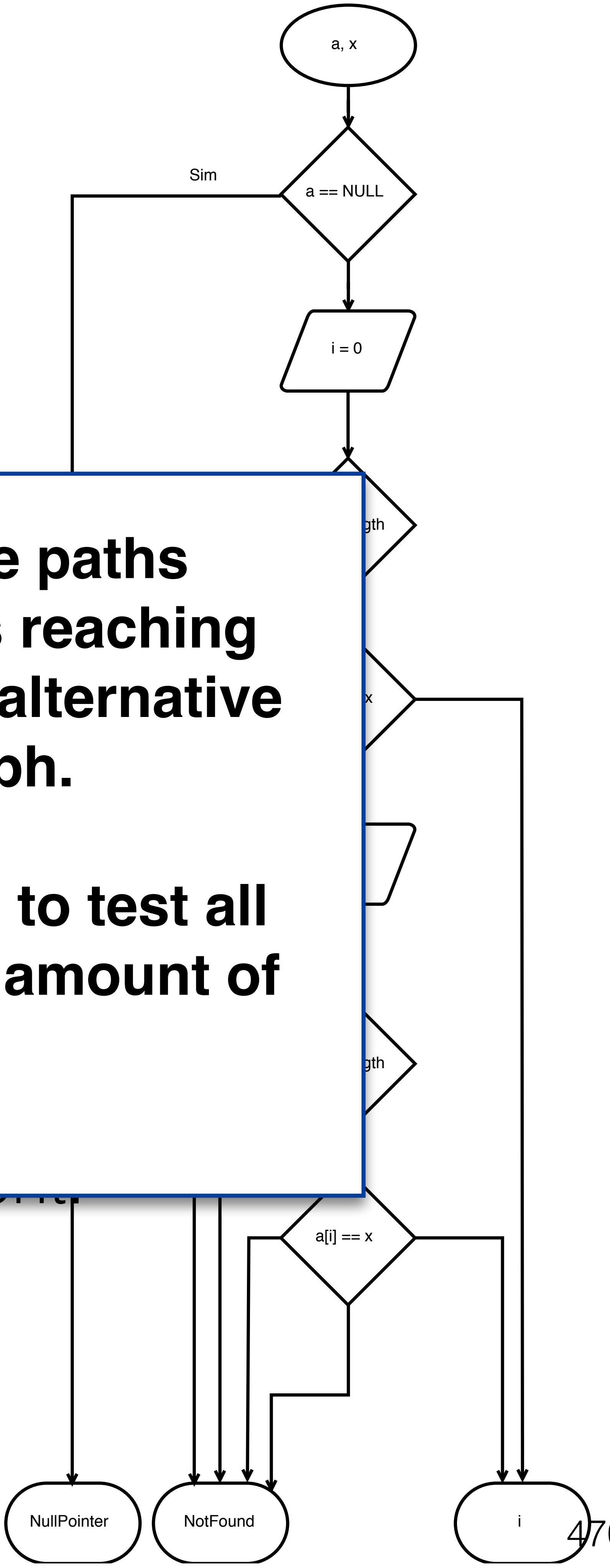
- An arra
  in the fi

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.

- 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.

Debugging

Debugging

1. Formulate hypothesis from the input data.
2. Create a test that recreates the error systematically.
3. Execute the test with dynamic contract checking for pre- and post-conditions.
4. Repeat 1, 2, 3 with refined knowledge.
5. Identify the source of the error, correct the code and specs.
6. Reenforce the badly specified contracts.
7. Add the produced tests the database of regression tests.
8. Check if all other tests still succeed.

Testing and Debugging

- Unit testing - Module by module
- Integration testing - More than a module.
- Regression testing - In case of evolution or modification
- Human Testing (Code reviews, Code Walkthrough)
- Test Driven Development
  - The development of new functionalities and the correction of bugs start with a test and ends with its success.
- Continuous Integration - test -> commit
- Continuous Delivery - test -> commit -> deploy
- Testing with a framework (mock services)

Quickcheck

Construction and Verification of Software

2017 - 2018

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Handout 4
João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

CVS 17/18 - Handout 4

Consider a domotic system that provides a hub for a set of devices, sensors or actuators, and control rules.

Sensors A sensor is updated (internally) by an internal timer that updates its state with readings that can be read concurrently by threads in the domotic control system. In this case, sensor readings are simulated by with a random value, or by some function over time. A sensor can be read concurrently. Sensors have a scale, and if a reading is out of the scale, the registered value is either the maximum or the minimum of the scale.

Actuators Actuators have an internal state with a property named "value" (setter and getter). The value can be read and written concurrently.

Rules Rules are objects that have references to sensors and actuators. Their behavior is triggered by an internal timer. There are several examples of rules: Indoor lighting (if someone is present and external light is below a certain value, turn lights on); Office hours (lock doors outside the working hours); AutoWindows (Close windows automatically after 5PM); Rain/Wind Protection (Close windows if rain sensor or wind sensor is above a threshold).

CVS 17/18 - Handout 4

- Implement a class to represent an **actuator** as a simple concurrent memory cell (get & set) for an integer value.
- Implement a class to represent a **rule** (Indoor lights), that has an internal timer, reads a set of sensors and changes a set of actuators (Lamps).
- Implement a class to represent a **rule** (Rain/Wind Protection), that has an internal timer, reads a set of sensors and changes a set of actuators (Windows).
- Implement a class to **log** the triggering of rules. Every time a rule is triggered (and produces results) the name of the rule, the input and output values are logged. (Use a NR1W monitor.)
- Implement **the domotic system** that hosts both rules running concurrently and sharing the same log system, and a dashboard that lists the actions triggered in the system.
- You need to establish contracts and verify your code with verifast so that it contains no execution errors.