Master's in Software Engineering

Software Testing, Verification and Validation

Implementing a Mutation Testing Tool using LARA

Utilization Manual

Beatriz Tavares (up201903011)

Diana Ramos (up201909457)

December19th, 2019

Index

Introduction

In the context of the *Teste, Verificação e Validação de Software* (TVVS) class, integrated in the Software Engineering Masters (MESW) course, it was proposed to implement a set of mutators in LARA language with support of a compilation tool called *Kadabra*.

Kadabra is a java-to-java compilation tool that supports LARA, an aspect-oriented programming language based on JavaScript, both developed by members of *the Faculdade de Engenharia da Universidade do Porto* (FEUP) and are still under development and refinement. Although LARA is being taken in consideration for the purpose of this project, it is also known to be used in other areas, for instance, for developing applications for high-performance embedded systems [1].

This report focuses on the mutators implemented, including specifications about the technologies used, and it is divided in four main parts: (1) a brief explanation of Mutation Testing and the needed software requirements to run the chosen *Kadabra* tool; (2) the mutations requested and implemented; (3) a user manual with essential operations to run the mutations tests developed and (4) a developer manual so that the code structure and implementation measures are understand, which will make it easier for future features implementation.

A little Bit about Mutation Testing

Mutation Testing is a type of software testing that checks tests themselves [2]: changes are performed to certain code statements to check that the implemented test cases are able to find errors. If those test cases find errors, then it may mean that they are suitable and accurate. If not, there should be performed improvements to those test cases if possible.

Mutation Testing is a type of White Box Testing, which is the testing of a software structure, design and coding and in which that same code is visible to the tester.

The changes made to the code statements should be carefully thought and analyzed before implemented, as it is wanted to avoid negative impact in the overall purpose of the program.

LARA language is already known for implementing test mutators, and for that, it has a good basis to the work developed in the scope of this project.

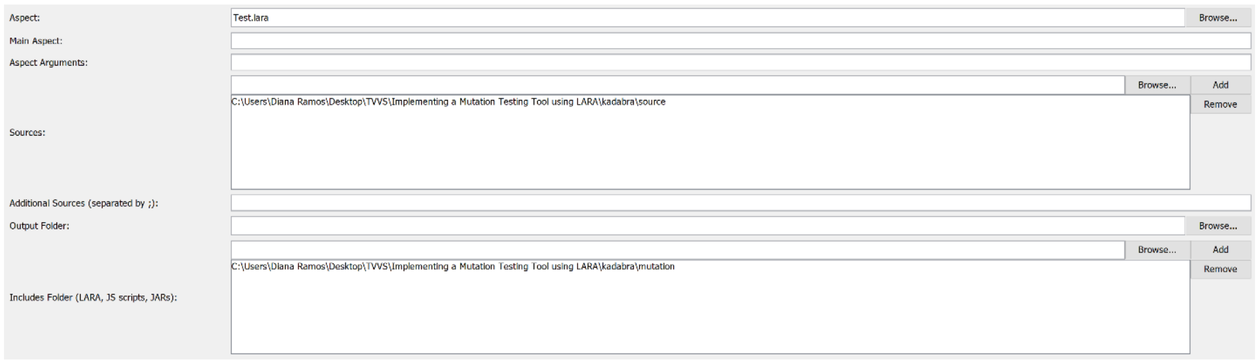
Prerequisites for Kadabra's Mutation Testing Tool

1) Downloading ***Kadabra*** compilation tool:
    a. Access the Kadabra's online page: http://specs.fe.up.pt/tools/kadabra/;
    b. Click on "Download Kadabra", under the "Resources" tab;
    c. Unzip the downloaded folder;
    d. Click on "kadabra.jar" executable to run it offline.

<u>Note:</u> after unzipping, one of the objects is a folder called "JavaWeaver_lib", which contains libraries for java and, since this is a compilation tool under development, it is important to download it frequently so that tool is always the most recent.

2) Configuring ***Kadabra*** compilation tool (see figure 1):
    a. With the tool opened, click on the "Options" tab;
    b. Insert the following information:
        i. Name of the main Aspect file with the following syntax: <*mutator_to_test*>.lara.
        ii. Source java code folder path with the help of the "Browse" and then "Add" button;
        iii. LARA scripts folder with the help of the "Browse" and then "Add" button.

<u>Note:</u> there are two versions of this tool: online and offline. The first already has support of some basic tutorials and it does not need any prerequisite; however, when implementing more extended applications, it is recommended to use the offline version to fully benefit from Kadabra.


Aspect: Test.lara
Main Aspect:
Aspect Arguments:
Sources: C:\Users\Diana Ramos\Desktop\TVVS\Implementing a Mutation Testing Tool using LARA\kadabra\source
Additional Sources (separated by ;):
Output Folder:
Includes Folder (LARA, JS scripts, JARs): C:\Users\Diana Ramos\Desktop\TVVS\Implementing a Mutation Testing Tool using LARA\kadabra\mutation

Screenshot 1: *example of an "options" tab configuration.*

3) Java version 11 or higher.

Implemented Mutation Testing Operators

These are the following operators requested to be implemented.

- **Literals and Byte, Short, Long, Float, Double:**

Afected types	Mutation Nº	Mutation
Any literal	1	0 to 1
	2	0 to -1
boolean	3	True to false
	4	False to true
String	5	One string to “ ”
Int, byte and short	6	1 to 0
	7	-1 to 1
	8	5 to -1
	9	If other value than increment by 1
long	10	1 to 0
	11	If other value than increment by 1
float	12	1.0 to 0.0
	13	2.0 to 0.0
	14	If other value than switch to 1.0
double	15	1.0 to 0
	16	If other value than switch to 1.0

Table 1: *Functional requirements for the Literal-related mutator.*

- **Operator for Constants:**

Afected types	Mutation
Any numeric global variable	Variable value to 1
	Variable value to 0
	Variable value to -1
	Variable value multiplied by -1
	Variable value added 1
	Variable value minus 1

Table 2: *Functional requirements for the global variables-related mutator.*

- **Arithmetic Operators:**

Mutation Name	Short Description	Resulted Mutant
Arithmetic Operator Deletion Mutator	This mutator detects any binary expression and behaves the following way.	s = a + b is split into s = a and s = b
		s = (a + b) + c is split into s = a + b; s = c; s = a + c and s = b + c

***Table 3:** Functional requirements for the arithmetic operator-related mutator*

- **Java Methods:**

Mutation Name	Short Description	Resulted Mutant
Fail on Null (FON)	Every time there is a code statement referencing an object, there is inserted code to check if that object is null (before the reference). It should be launched a message "fail on empty" if dealing with Strings and if the null condition is verified. For other types of objects, it should be launched a message "fail on null" if the null condition is verified.	s = a + b is split into s = a and s = b

***Table 4:** Functional requirements for the FON-related mutator.*

- **Nullify Input Variable:** every time a method receives an object reference, it is assigned to null in the beginning of the method.

- **Nullify Return Value:** every time a method returns an object, it is replaced by null.

- **Nullify Object Initialization:** every time there is a new statement, it is replaced with null.

- **Remove Null Check:**

Mutation Name	Short Description	Resulted Mutant
Remove Null Check Mutator	Every time there is a binary relational statement containing null at one side it is negated.	null != a mutates into null == a
		a == null mutates into a != null

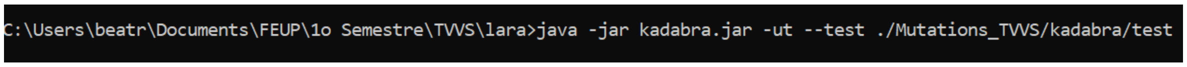
Table 5: *Functional requirement for the remove null check-related mutator.*

User Manual

This chapter is focused on the user side of this project and describes how he/she can run this tool and test Java code via two ways.

Running the Test Using the Unit Testing Interface

1) Go to the root folder of the project;
2) Open a command line and insert the following line:
    a. java -jar <path_to_kadabra.jar> -ut –-test <path_to_test_folder>:



```
C:\Users\beatr\Documents\FEUP\1o Semestre\TVVS\lara>java -jar kadabra.jar -ut --test ./Mutations_TVVS/kadabra/test
```

Screenshot 2: *Example of a command line to run the implemented unit tests.*

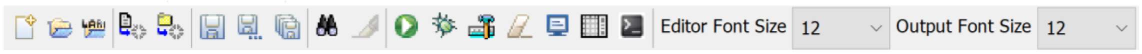
3) You should see the tests running.

Running the Test Using Kadabra's Interface

1) Define a main aspect named Test.lara;
2) Copy one of the already defined mutators, for instance the testArithmeticOperatorDeletionMutator function;
3) Create a Java code test, for instance the following code:

```
public int operation() {
        int sum = a + b + c;
        return sum;
}
```

4) Link both files;
5) Go to Kadabra's upper tab and click on the green run button:



Screenshot 3: *Kadabra's upper tab.*

6) You should see some output in the console.

Developer Manual

This chapter is focused on the developer side of this project and describes all implemented mutators interfaces with some code exemplification.

In order to perform the mutations implementation, there was used a common Mutator interface, which is something already existing in LARA. Before explaining the developed mutators operators, there will be an explanation about how the project is structured, how the developer can implement more mutators and link them with the current system and how this interface was used and its purposes.

Project Structure

This project is structured in four parts: the implemented mutations, the source file in java code, the mutator tester and the expected output. The following package diagram is an overview of the system file organization regarding a single mutator, the literal mutator interface as an example. All mutators follow the same structure.

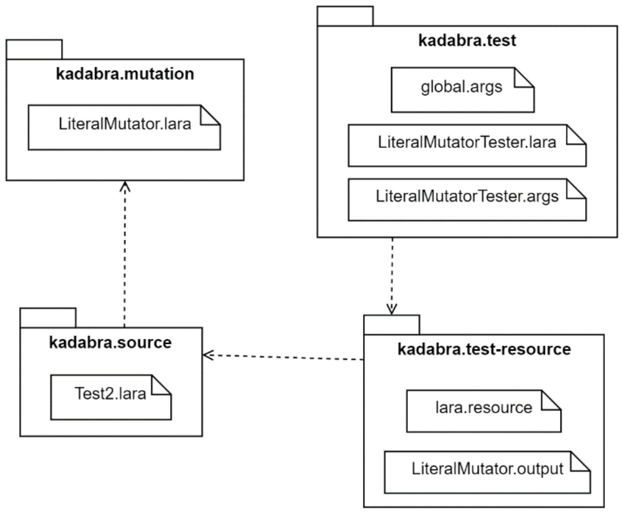


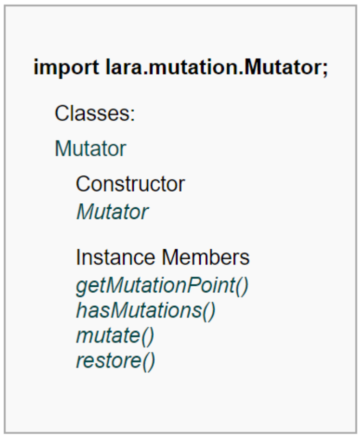
Diagram 1: *Package diagram representing the file organization of the Literal Mutator.*

The implemented unit testing interface will perform the mutation on the detected Java source code, which is located in the *kadabra.source* folder, by instantiating the desired mutator, located in *kadabra.mutation* folder on a mutator tester, located in *kadabra.test* folder and check if the resulted mutants are the expected ones, by checking the output files located in the *kadabra.test-resource* folder.

If there were to be implemented other mutator interfaces, the developer need to follow this structure and create the same files, only customizing the names and logic of the mutation operation.

Mutator Interface

The mutators classes implemented in the scope of this project are all prototypes of this Mutator LARA class, and for that, they overwrite its constructor and its methods, which can be checked in the following image [2]:


import lara.mutation.Mutator;

Classes:

Mutator

Constructor

Mutator

Instance Members

getMutationPoint()
hasMutations()
mutate()
restore()

Figure 1: *Mutator interface implemented by LARA [4].*

To initialize every single mutator class, the constructor will be used and it's there were we define the points in the code analyzed that are going to suffer the mutation. When a specific mutator class is initialized, there is going to be a search for specific expressions, declarations, variables (filtered by certain conditions) or other type of attributes, accordingly to what the mutator pretends.

When a mutation point is found in the code, it's going to be added to an array of mutations points.

After the mutator class is initialized, a ***hasMutations()*** function will be running until it doesn't have more mutation points to transform. While this function is working, we perform a call to the ***mutate()*** function, which is going to perform the mutation desired. So that we can see which point we are dealing with, we can call a ***getMutationPoint()*** function.

Finally, for each mutation point, after it is mutated, it should be done a ***restore()*** operation, so that the new mutation point doesn't overwrite the old mutation point and we keep the original code.

Implemented Mutators

This section explains how each mutator mentioned above were implemented and how they deal with exceptional cases.

Arithmetic Operator Deletion Mutator Interface

This mutator will, firstly, search for every binary expression, which means, every operation between two variables.

So, imagining one of the binary expressions found is this code line:

sum = a + b

This is the binary tree seen by LARA:

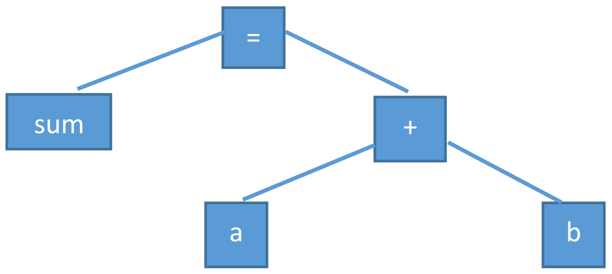


Figure 2: *Binary tree of sum=a+b.*

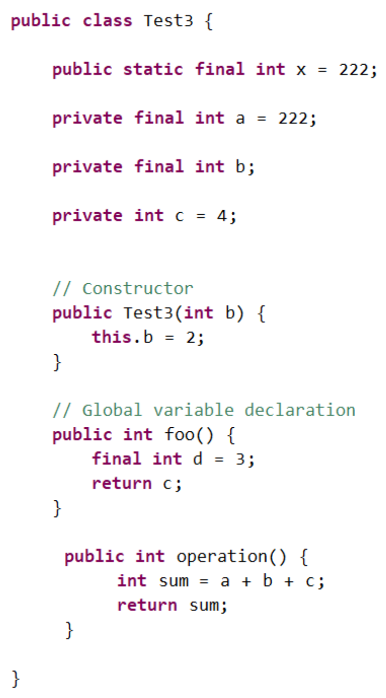
The mutator will store the mutation point, which is going to be the binary expression a + b and it will look at its left (a) and its right (b). As it must run two times, so that it can check the left and then the right, we will use a *Boolean* variable called *isFirst*: it is set to false to look at the binary expression left and, once is done, it is set to true to look at the binary expression right.

```
ArithmeticOperatorDeletionMutator.prototype._mutatePrivate = function() {

    var mutationPoint = this.mutationPoints[this.currentIndex];

    this.previousValue = mutationPoint;

    if (this.isFirst === false) {
        var leftOperand = mutationPoint.lhs.copy();
        this.newValue = mutationPoint.insertReplace(leftOperand);
        this.isFirst = true;
    }
    else {
        var rightOperand = mutationPoint.rhs.copy();
        this.newValue = mutationPoint.insertReplace(rightOperand);
        this.isFirst = false;
        this.currentIndex++;
    }
```

Excerpt 1: *Mutation private function regarding the arithmetic operator deletion mutator.*

The *leftOperand* will be a copy of the binary expression left, which also returns an expression. Then it is made a replace: the binary expression is replaced by its left Operand (a). The same thing is made to the *rightOperand* but now replacing the binary expression by its right expression (b).

To check if the mutator was performing correctly, we used the following java test class:



```
public class Test3 {

    public static final int x = 222;

    private final int a = 222;

    private final int b;

    private int c = 4;


    // Constructor
    public Test3(int b) {
        this.b = 2;
    }

    // Global variable declaration
    public int foo() {
        final int d = 3;
        return c;
    }

     public int operation() {
          int sum = a + b + c;
          return sum;
     }

}
```

Excerpt 2: *Java source code to test the arithmetic operator deletion mutator.*

As you can see, we have a more complex operation (int sum = a + b + c) which will be detected by the mutator. The binary expression tree seen by LARA is now the following:

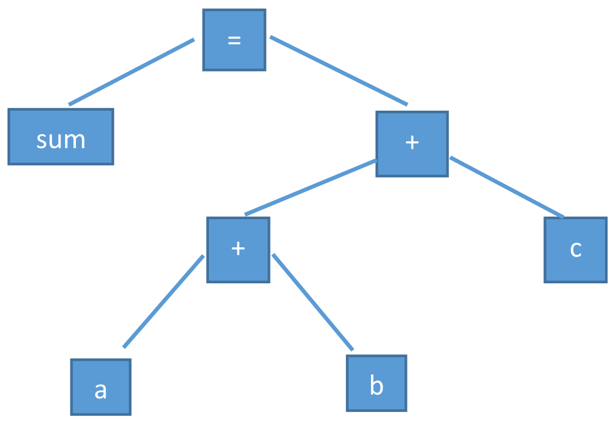


Figure 3: *Binary tree of sum=a+b+c.*

The binary expression detected is (a + b) + c. As we have 2 operands in the same operation, there is going to be 4 different possible results:

- sum = a + b
- sum = c
- sum = a + c
- sum = b + c

For the first result, the tree searches the first operand (+) and it looks at its left, replacing the operand by the expression a + b; for the second result, the tree is still searching in the first operand and it looks at its right, replacing the operand by the expression c; for the third result, the tree is now searching at the second operand (+) and is going to look at its left, replacing the second operand by a and joins it with what it has above (+ c). Finally, for the fourth result, the tree is searching at the second operand by now looking at its right, replacing the second operand by b and joins it with what is has above (+ c).

These 4 results are stored in the file *ArithmeticOperatorDeletionMutator.output* and compared with the result provided by the test. We can see the test involved operations in the following sequence diagram:

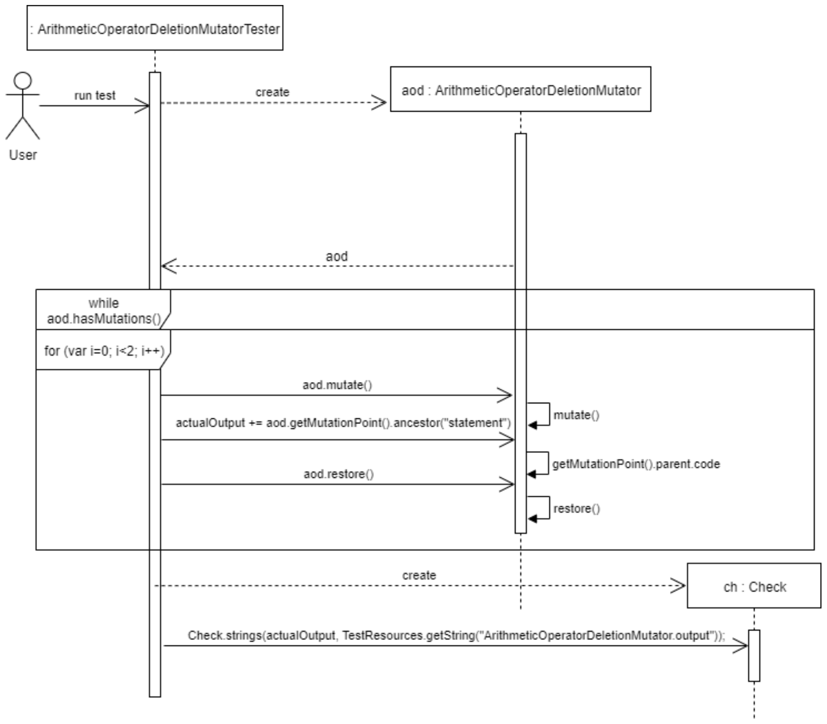

: ArithmeticOperatorDeletionMutatorTester
User
run test
create
aod : ArithmeticOperatorDeletionMutator
aod
while
aod.hasMutations()
for (var i=0; i<2; i++)
aod.mutate()
mutate()
actualOutput += aod.getMutationPoint().ancestor("statement")
getMutationPoint().parent.code
aod.restore()
restore()
create
ch : Check
Check.strings(actualOutput, TestResources.getString("ArithmeticOperatorDeletionMutator.output"));

Diagram 2: *Sequence diagram regarding running a single operator mutator.*

Fail on Null Mutator Interface

This mutator will, firstly, search for every code variable that is being referenced (it is only thrown a *NullPointerException* in java when trying to call something on an object) and that is not primitive, which means, that is an object.

When found the mutation points, if one is a String, it will be thrown a *NullPointerException* with the message “fail on empty”; if one is another type of Object, it will be thrown a *NullPointerException* with the message “fail on null”. We can see this in code bellow:

```
FailOnNull.prototype._mutatePrivate = function() {

    var mutationPoint = this.mutationPoints[this.currentIndex];

        if (isString(mutationPoint)) {
            this.newValue = mutationPoint.insertBefore(%{if ([[mutationPoint]] == null)
            { throw new NullPointerException("fail on empty") }}%);
        }
        else {
            this.newValue = mutationPoint.insertBefore(%{if ([[mutationPoint]] == null)
            { throw new NullPointerException("fail on null") }}%);
        }
```

Excerpt 3: *Mutation private function regarding the null pointer exception mutator.*

To check if the mutator was performing correctly, we the following java test class was used:

```
public class Test4 {

    private String b;
    private Test4 c;

    //Constructor
    public Test4(String b) {
        this.b = new String("");
        this.c = null;
    }

    //Global variable declaration
    public int test1() {
        this.c.test5();
        return 0;
    }

    public int test2(String object) {
        object = new String();
        if (object.length() == 2) {
            b = "hey";
        }
        return -1;
    }

    public double test4(long a, boolean b) {
        return 0;
    }

    public void test5() {
    }

}
```

Excerpt 4: *Java source code to test the arithmetic operator deletion mutator.*

The output from the test should be compare with the *FailOnNull.output* file, which is the following:

```
/**
 * Inserted by Kadabra
 */
if (this.c == null)
                           { throw new NullPointerException("fail on null") }
/**
 * Inserted by Kadabra
 */
if (object == null)
                           { throw new NullPointerException("fail on empty") }
```

Excerpt 5: *Generated Java code (mutant).*

We can see the overview test involved operations in the following sequence diagram:

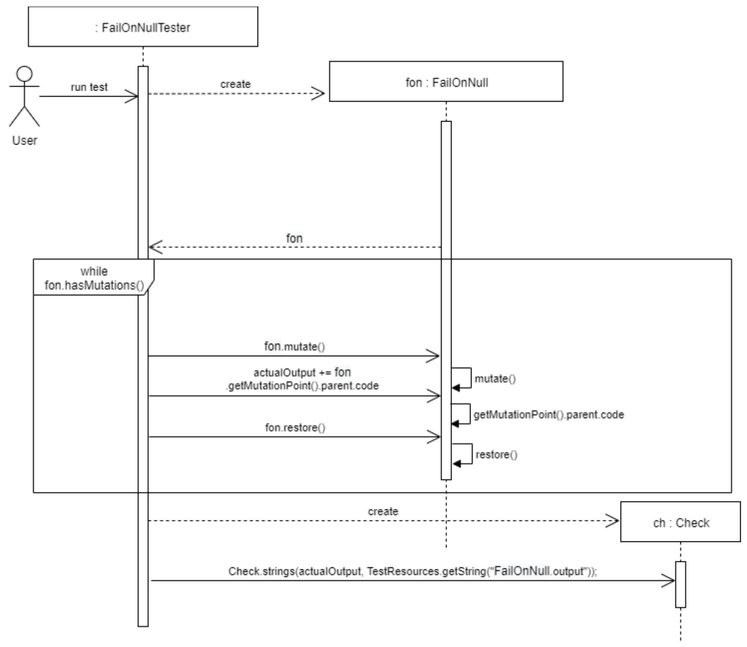


Diagram 3: *Sequence diagram regarding running a single fail on null mutator.*

Nullify Input Variable Interface

This mutator will, firstly, search for every method in the code that has at least one parameter not primitive, which means, at least one parameter of type Object. Thus, there will be a Boolean variable that is set to true when it is found an Object parameter in a method so that this method is added to the mutation points, just like the code bellow:

```
for(var $method of WeaverJps.search('method').get()) {
     for(var $param of $method.params) {
          if (!$param.isPrimitive) {
               hasPrimitiveParams = true;
          }
     }
     if (hasPrimitiveParams === true) {
          this.mutationPoints.push($method);
     }
     hasPrimitiveParams = false;
}
```

Excerpt 6: *Search mechanism.*

For each mutation point detected, or method, each parameter that is an object will be assigned null. This assignment is inserted before the method body code, as we can see below:

```
var mutationPoint = this.mutationPoints[this.currentIndex];
this.previousValue = mutationPoint.code;

for (var $param of mutationPoint.params) {
     if (!$param.isPrimitive) {
          mutationPoint.body.insertBegin(%{[[$param.name]] = null}%);
          this.newValue = mutationPoint;
     }
}

this.currentIndex++;
```

Excerpt 7: *Assigning null to reference.*

To check if the mutator was performing correctly, we used the following java test class:

```
public class Test2 {

    private String b;
    private Test2 c;
    //Constructor
    public Test2(String b) {
        this.b = new String("");
        this.c = null;
    }

    //Global variable declaration
    public int test1() {
        this.b = new String("");
        int a = c.test2("");
        a += a;
        return 0;
    }

    public int test2(String object) {
        if (null != object) {
            return 1;
        }
        if (object == null) {
            return 2;
        }
        b += b;
        return -1;
    }

    public String test3(double a) {
        return new String("");
    }
}
```

Excerpt 8*: Java source code to test the null input variable mutator.*

The output from the test should be compare with the *NullifyInputVariable.output* file, whose important snippet is the following:

```
public int test2(String object) {
    /**
     * Inserted by Kadabra
     */
    object = null;
    if (null != object) {
        return 1;
    }
    if (object == null) {
        return 2;
    }
    b += b;
    return -1;
}
```

Excerpt 9: *Generated Java code (mutant).*

<u>Note:</u> The sequence diagram with an overview of all involved test operations it the same as the one showed in Fail on Null Interface, except for the names used in the classes and files.

Nullify Return Value Interface

This mutator will, firstly, search for every return statement in the code that belongs to a method which return type is not primitive, which means, it will search for every method that returns an Object. When found, it will store on the mutation points array the corresponding return statement.

Each mutation point, return statement, will be replaced by "return null".

```
NullifyReturnValue.prototype._mutatePrivate = function() {

    var mutationPoint = this.mutationPoints[this.currentIndex];
    this.previousValue = mutationPoint.code;

    this.newValue = mutationPoint.insertReplace("return null");

    this.currentIndex++;
```

Excerpt 10: *Mutation private function regarding the null return value mutator.*

To check if the mutator was performing correctly, we used the following java test class:

```
public class Test2 {

    private String b;
    private Test2 c;
    //Constructor
    public Test2(String b) {
        this.b = new String("");
        this.c = null;
    }

    //Global variable declaration
    public int test1() {
        this.b = new String("");
        int a = c.test2("");
        a += a;
        return 0;
    }

    public int test2(String object) {
        if (null != object) {
            return 1;
        }
        if (object == null) {
            return 2;
        }
        b += b;
        return -1;
    }

    public String test3(double a) {
        return new String("");
    }
}
```

Excerpt 11*: Java source code to test the null return value mutator.*

The output from the test should be compare with the *NullifyReturnValue.output* file, which content is the following:

```
{
    /**
     * Inserted by Kadabra
     */
    return null;
}
```

Excerpt 12: *Generated Java code (mutant).*

<u>Note:</u> The sequence diagram with an overview of all involved test operations it the same as the one showed in Fail on Null Interface, except for the names used in the classes and files.

Nullify Object Initialization Interface

This mutator will, firstly, search for every “new” statement in the code (initialization of Objects) and store it into the mutation points array.

For each mutation point, or for each “new” statement, it will replace it by null, as you can see below:

```
NullifyObjectInitialization.prototype._mutatePrivate = function() {

    var mutationPoint = this.mutationPoints[this.currentIndex];
    this.previousValue = mutationPoint.code;

    this.newValue = mutationPoint.insertReplace("null");

    this.currentIndex++;
```

Excerpt 13: *Mutation private function regarding the nullify object initialization mutator.*

To check if the mutator was performing correctly, we used the following java test class:

```
public class Test2 {

     private String b;
     private Test2 c;
     //Constructor
     public Test2(String b) {
          this.b = new String("");
          this.c = null;
     }

     //Global variable declaration
     public int test1() {
          this.b = new String("");
          int a = c.test2("");
          a += a;
          return 0;
     }

     public int test2(String object) {
          if (null != object) {
               return 1;
          }
          if (object == null) {
               return 2;
          }
          b += b;
          return -1;
     }

     public String test3(double a) {
          return new String("");
     }
}
```

Excerpt 14: *Java source code.*

The output from the test should be compare with the *NullifyObjectInitialization.output* file, which is the following:

```
this.b = null
this.b = null
return null
```

Excerpt 15: *Generated Java code (mutant).*

<u>Note:</u> The sequence diagram with an overview of all involved test operations it the same as the one showed in Fail on Null Interface, except for the names used in the classes and files.

Remove Null Check Interface

This mutator will, firstly, search for every binary expression which operand is != or == and that at one side of the operation is placed null, adding it to the mutation points array.

It will check if the left or the right of the binary expression is of type null, as you can check here:

```
for(var $binaryExpression of WeaverJps.searchFromInclusive($startingPoint, 'binaryExpression')
    if ($binaryExpression.operator === '!=' || $binaryExpression.operator === '==') {
        if ($binaryExpression.rhs.type === "<nulltype>"
        || $binaryExpression.lhs.type === "<nulltype>") {
            this.mutationPoints.push($binaryExpression);
        }
    }
}
```

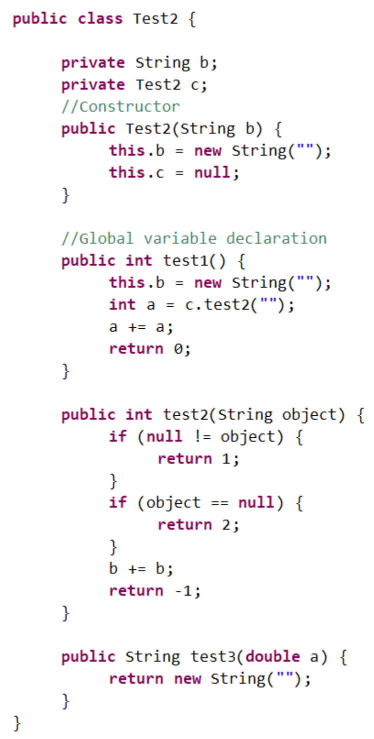
Excerpt 16: *Search mechanism.*

For each mutation point, which means, for each binary expression with null at one side, if it finds out that the operator is != it converts it to ==, and the vice versa. We can verify it with the code below:

```
RemoveNullCheck.prototype._mutatePrivate = function() {

    var mutationPoint = this.mutationPoints[this.currentIndex];

    this.previousValue = mutationPoint.copy();

    if (mutationPoint.operator === '!=') {
        mutationPoint.setOperator('==');
        this.newValue = mutationPoint;
    }

    else if (mutationPoint.operator === '==') {
        mutationPoint.setOperator('!=');
        this.newValue = mutationPoint;
    }
```

Excerpt 17: *Mutation private function regarding the remove null check mutator.*

To check if the mutator was performing correctly, we used the following java test class:



```java
public class Test2 {

     private String b;
     private Test2 c;
     //Constructor
     public Test2(String b) {
          this.b = new String("");
          this.c = null;
     }

     //Global variable declaration
     public int test1() {
          this.b = new String("");
          int a = c.test2("");
          a += a;
          return 0;
     }

     public int test2(String object) {
          if (null != object) {
               return 1;
          }
          if (object == null) {
               return 2;
          }
          b += b;
          return -1;
     }

     public String test3(double a) {
          return new String("");
     }
}
```

Excerpt 18: *Java source code.*

The output from the test should be compare with the *RemoveNullCheck.output* file, which content is the following:

```java
if (null == object) {
     return 1;
}
if (object != null) {
     return 2;
}
```

Excerpt 19: *Generated Java code (mutant).*

<u>Note:</u> The sequence diagram with an overview of all involved test operations it the same as the one showed in Fail on Null Interface, except for the names used in the classes and files.

Literal Mutator Interface

One of the mutators implemented was the literal mutator interface which mainly identified all value-based variables that are explicitly defined above, and switched their value for others. For instance, if there is the following line on a java code:

int x = 0;

The output should be:

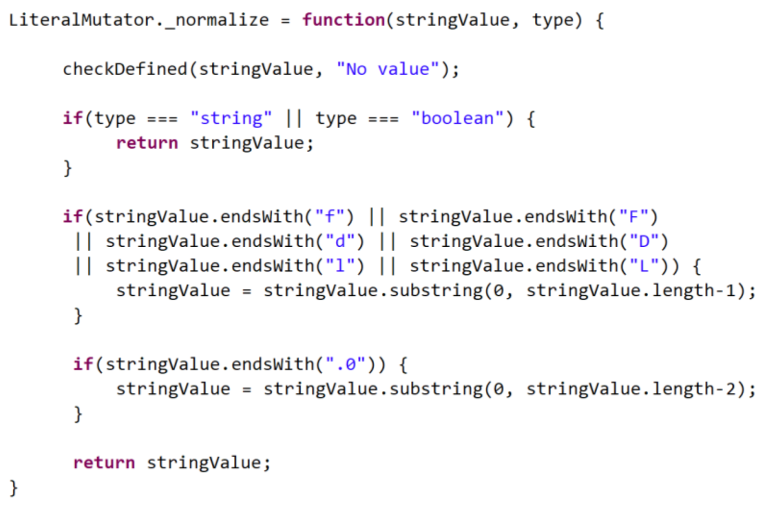
int x = 1;

int x = 1;

Note: This specific case should result in two different mutations with the same value due to the requirements above.

For this purpose, first we need to identify the type of node we are dealing with, namely a "literal" so it can be taken into account only these nodes. The values of any variable are the targets here so the mutator could only deal with a string (that is the value) rather than a node manipulated by LARA. Although this was a simpler way of implementing it, it becomes limited functionality wise and having a node into consideration, not only its value, is a more appropriate solution.

As mentioned above, there are multiple ways of declaring a literal, for instance a variable of type integer, byte or short have the same declaration; however, when it comes to mutate a variable which has string attached to it, i.e. a long (long x = 1l), this rises some problems. This type of concerns is handled by a normalization static function, inside the mutator itself, as above:



```
LiteralMutator._normalize = function(stringValue, type) {

     checkDefined(stringValue, "No value");

     if(type === "string" || type === "boolean") {
          return stringValue;
     }

     if(stringValue.endsWith("f") || stringValue.endsWith("F")
      || stringValue.endsWith("d") || stringValue.endsWith("D")
      || stringValue.endsWith("l") || stringValue.endsWith("L")) {
          stringValue = stringValue.substring(0, stringValue.length-1);
      }

      if(stringValue.endsWith(".0")) {
          stringValue = stringValue.substring(0, stringValue.length-2);
      }

      return stringValue;
}
```

***Excerpt** 20: Normalization function located on the LiteralMutator.*

Since the affected nodes (see table 1) are both literals but from different types, that is, different types of variables, in some cases, will have different mutations, it was important to create a support for these in a more efficient way rather than duplicating code with some minor modification, that is why the following static function was implemented:

```
LiteralMutator.newTypeFilteredMutator = function($newValue, validTypes, oldValue) {
     checkDefined($newValue);
     checkDefined(validTypes);

     return new LiteralMutator($newValue, function(mutationPoint){
          if(!isUndefined(oldValue) &&
               LiteralMutator._normalize(mutationPoint.code, mutationPoint.type)
               !== LiteralMutator._normalize(oldValue, mutationPoint.type)) {
          ;
               return false;
          }

          if(validTypes === undefined || validTypes.length === 0) {
               return true;
          }

          return validTypes.includes(mutationPoint.type);
     });
}
```

Excerpt 21: *Filtering static function located on the LiteralMutator.*

This new function will be receiving a required a final value of a literal, types of variables and an optional target value. Let's consider an int, byte or short type of variable to mutate the literal from 0 to 1. In this case, this function will be called as followed:

```
LiteralMutator.newTypeFilteredMutator(KadabraNodes.literal("0", "int"), ['int', 'byte', 'short'], "1")
```

Excerpt 22: *Calling the newTypeFilteredMutator static function.*

After the node identification, it then will occur the mutation itself, which will require two main parts, namely the mutation and restore action of the literal. The first one will be the mutation itself, where the mutator will switch the two values and the second one will be responsible for restoring the variable value with the previous value, otherwise if there is more than one mutation for a certain literal, the value to change is not the correct one but the one resulted in the previous mutation.

The following sequence diagram represents the workflow when the mutator tester interface for literals is used, i.e. when the LiteralMutatorTester function is called.

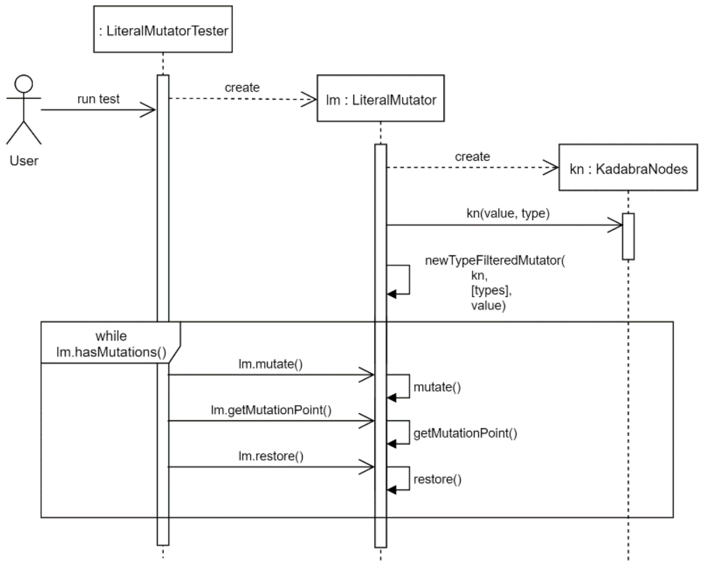


Diagram 4: *Sequence diagram regarding running a single literal mutator.*

Exceptional mutations that require arithmetic operations, for mutation nº 9 (for int, byte and short, if other value than increment by 1), the mutation is handled differently since the value of the variable is unknown to the common user, that is, the user will only run the tests and confirm the results and the search for literals is hidden in the mutator business logic, one method of solving this problem is to create a function, in this case a sum function, and send it to the mutator.

```
mutators.push(LiteralMutator.newTypeFilteredMutator(
     function($expr) {
          var newValue = Number($expr.code) + 1;
          return KadabraNodes.literal(newValue.toString(),$expr.type);
     }
     , ['int', 'byte', 'short'], "1"));
```

Excerpt 23: *Calling the newFilteredMutator with a function parameter.*

When the mutator is ready to perform the mutate action, it will verify if the passed parameter is indeed a function and will calculate the final value accordingly with how this function proceeds.

```
var valueToReplace = this.newValue;
println(valueToReplace);
if(isFunction(valueToReplace)) {
     valueToReplace = valueToReplace(mutationPoint);
}
```

Excerpt 24: *Arithmetic operations via an isFunction condition.*

Constant Mutator Interface

Another mutator implementation was only meant for constants, i.e. a global variables, and any variable with this tag would suffer from the mutation and switch values. For instance, if there is the following line on a java code:

private final int x = 2;

The output should be:

private final int x = 1;

private final int x = 0;

private final int x = -1;

private final int x = -2;

private final int x = 3;

private final int x = 1;

In this case, we want to verify if a certain variable is constant. In Java programming language that means that the reference must have the "final" tag before the value and there are still three cases of constants to be taken into account:

1) When the global variable is instantiated outside any method;
2) When the global variable is created outside any method but instantiated inside the constructor;
3) When the global variable is locally created and instantiated.

For the first case, the mutator needs to search any node with the "field", "assignment" and "local variable" type and validate if the node has the tag final.

```
if(!$field.isFinal) {
     continue;
}
```

Excerpt 25: *Validating if a certain field is a final variable.*

A "field" and "local variable" are nodes with a value associated; however, when we try to manipulate an "assignment" type of node, as it suggests, the node is composed by two parts, one on the left and the other on the right of the equals operator, therefore, when trying to access the value of an assignment, the "rhs" which is the right part of an assignment is needed.

Similar to the literal mutator interface, there are exceptional cases where the mutation is a result of an arithmetic operation so both the problem and solution are extended to this mutator, where the constructor is initialized as the following:

```
mutators.push(new ConstantMutator(
     function($expr) {
               var newValue = Number($expr);
               newValue = newValue * (-1);
               return KadabraNodes.literal(newValue.toString(),"int");
     }
));
```

Excerpt 26: *Inserting a calculation function in the ConstantMutator.*

When the mutator is ready to perform the mutate operation, it will verify if the first parameter is a function and then calculate the final value with the established function on the constructor.

```
if(isFunction(this.$expr)){
          var tem= this.$expr(this.previousValue);
          this.newValue = this.previousValue.insertReplace(tem);
}else{
          this.newValue = this.previousValue.insertReplace(this.$expr);
}
```

Excerpt 27: *Arithmetic operations via an isFunction condition.*

The following sequence diagram represents the workflow when the mutator tester interface for global variables is used, i.e. when the ConstantMutatorTester function is called.

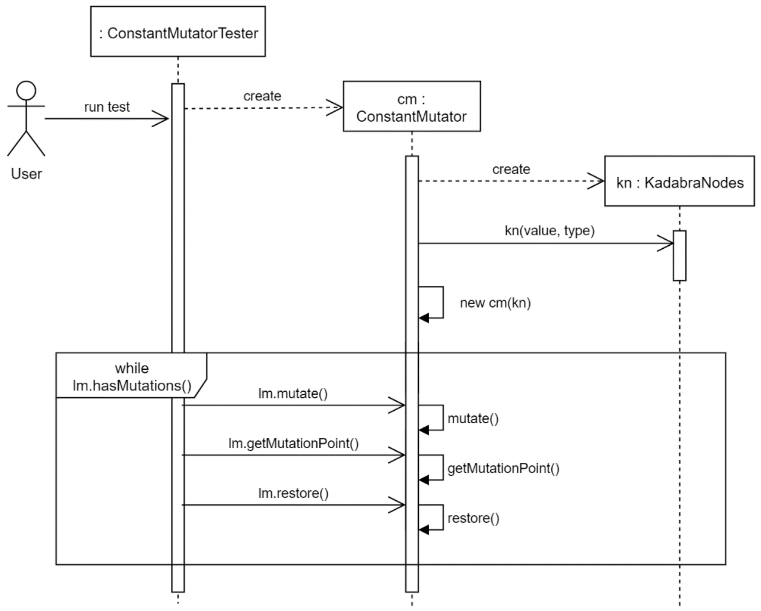


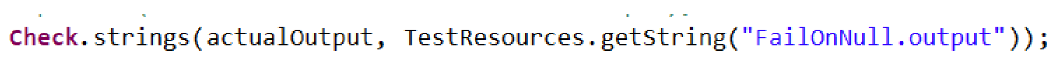
Diagram 5: *Sequence diagram regarding running a single constant mutator.*

Future Work

To add more Mutation interfaces, the developer should check how existing classes are structured and defined in the folder **mutation** and create its own class with the extension ***.lara*** for the specific purpose of the mutation.

When written the mutation class, it should perform a unit test to check if it's working properly. For this, the developer needs a java test class, which can be one of the already existing classes in the **source** package or can create a new one with the extension ***.java***.

Then, it should create a tester class as the ones inside the package **test** with the extension ***.lara***, following the same structure and performing the changes needed. To check if the result is equal to the expected, in the tester class, it should be made a call to this method:

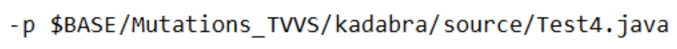


```
Check.strings(actualOutput, TestResources.getString("FailOnNull.output"));
```

Excerpt 28: *Check if mutator generates the correct mutants.*

Thus, it should be created a file in the package **test-resources** with the specific name and the extension ***.output***.

For each new tester class created, it should be created a file with the same name and the extension ***.args.*** In this file it should be defined the path to java test class, as you can see below:



```
-p $BASE/Mutations_TVVS/kadabra/source/Test4.java
```

Excerpt 29: *Argument to create a test file.*

Notice that the file ***global.args*** in the package **test** defines the included folder so that the tester classes import their referenced mutator *lara* class. On the other hand, the file ***lara.resource*** defines the path one should import when comparing the tester class result with the expected inside the corresponding ***.output*** file.

With these instructions, the developer should be prepared to implement new mutator tests, starting from the already existing project basis.

Conclusion

Although it was rough at the beginning to understand the LARA language structure and its specifications, with the help of João Bispo, one of the head researchers and developers of this language, this project implementation became simpler.

With this work, we acquired deeper knowledge in the field of Mutation Testing and had the opportunity to learn about a new language (although it has similarities with JavaScript) and how to work on a different developing tool.

The process turned out to be a rich experience, where we had to overcome barriers and become more professionally independent.

References

[1] http://specs.fe.up.pt/tools/lara/doku.php?id=start (17.12.2019 at 23:27h).

[2] Mutation Testing: https://www.guru99.com/mutation-testing.html (18.12.2019 at 14:45h)

[3] LARA speficifications: http://specs.fe.up.pt/tools/jackdaw/doc/# (19.12.2019 at 09:55h)

[4] LARA framework's documentation: http://specs.fe.up.pt/tools/kadabra/doc/#