Implementation of mutation testing operators using LARA

José Cunha, Nelson Almeida

***Index Terms*— Java, Kadabra, LARA, Source Code Mutation**

I. Introduction

This document approaches certain mutators developed for Kadabra, a Java source-to-source compiler tool based on the LARA language, within the context of TVVS course at FEUP. Two manuals are presented in this report. One developer manual for those who wish to develop new mutators for Kadabra, and one user manual for those who intend to generate mutated code through the available Kadabra programming interface.

A. Abbreviations and Acronyms

LHS	Left hand side of a binary expression
RHS	Right hand side of a binary expression
AST	Abstract Syntax Tree
joinpoint	A given node in the AST of the program that will be mutated
$jp	Code variable representing a joinpoint
BWO	BitWise Operator Mutator
COD	Conditional Operator Deletion Mutator
COI	Conditional Operator Insertion Mutator
CDL	Constant Deletion Mutator
CCL	Constructor Call Mutator
IPC	Explicit Super Call Mutator
NVC	Non Void Call Mutator
RMC	Remove Conditional Mutator
RTV	Return Value Mutator

II. Setting up Kadabra

Firstly it's necessary to download the tool and extract the .zip to a folder. The contents should be, at the time of writing, kadabra.jar, the tool itself, and a folder named JavaWeaver_lib.

Before continuing it is suggested that LARA scripts, java source files, and test classes are saved on their separate folders to keep files organized and make future configurations easier, as reconfigurations occur often.

After downloading Kadabra, the tool requires a small setup before its use. After running the .jar a window with 3 tabs will show up. On the second tab "Options", there are 3 fields that must be filled in, "Aspect:", "Sources:" and "Add Classpaths:".

"Aspect:" should point to a .lara file that the tool will take as the main file. If there are no .lara files yet, the 3rd tab "LARA Editor" can be used to create and save a new .lara file.

"Sources:" should point to the java source files that will be subject to the source mutations.

"Classpaths:" should point to the extracted folder JavaWeaver_lib.

After having these fields filled in the "Save" or "Save as..." buttons save the configuration in a file. Finally, "Options file:" in the 1st tab "Program" should point to the previously saved configuration file.

Whenever it is necessary to change the main LARA script or the target java source code, "Aspect:" and "Sources:" needs to be changed.

III. User Manual

This section addresses the results produced by the implemented mutators and their instantiation, including the more specific ones which deal with multiples constructor arguments. Also instructs new developers on the proper use of the Mutator interface to generate mutated code.

A. Implemented mutators

It's important to denote that the first argument of every mutator's constructor (**$jp**) is the joinpoint used as starting point for mutation targets search. Meaning, only the joinpoint and its descendant nodes will be searched for possible mutation targets.

BWO:

```
new BitWiseOperatorMutator($jp, $target_op,
↪  [$mutation_ops...]);
```

Replaces a given bitwise operator by another bitwise operator or by the RHS/LHS of the bitwise expression. If the operator is unary, it can be removed.

$target_op - bitwise operator to be replaced

[$mutation_ops...] - list of possibilities to replace the target operator (lhs, rhs, |, &, ^, "). " is the only unary operator allowed - removes ~ in unary expression

COD:

```
new ConditionalOperatorDeletionMutator($jp);
```

Removes the unary operator ! from conditions in ifs, loops and ternary operators.

COI:

```
new ConditionalOperatorInsertionMutator($jp);
```

Applies the unary operator ! to the whole condition in ifs, loops and ternary operators.

CDL:

```
new ConstantDeletionMutator($jp, $target_constant);
```

Removes a constant (final) variable from arithmetic expressions.

$target_constant - name of the variable declared as final, which will be eliminated from arithmetic expressions

CCL:

```
new ConstructorCallMutator($jp);
```

Replaces a class constructor call by null value.

IPC:

```
new InheritanceIPCMutator($jp);
```

Deletes super constructor calls.

NVC:

```
new NonVoidCallMutator($jp);
```

Replaces calls to non void methods by a default value:

Primitive or boxed BOOLEAN	false
INT, BYTE, SHORT or LONG	0
FLOAT or DOUBLE	0.0
CHAR or STRING	'\u0000'
OBJECT	null

RMC:

```
new RemoveConditionalMutator($jp);
```

Replaces conditions in ifs, loops and ternary operators by true.

RTV:

```
new ReturnValueMutator($jp);
```

Replaces method return value by a default value according to the method type:

INT, SHORT, LONG, CHAR, FLOAT or DOUBLE	0
Primitive or boxed BOOLEAN	true

B. Mutator Interface usage

The following code indicates how to iterate through all mutations generated by a single mutator class, apply the mutation and restore the program back to its original version:

```
1  var $jp = WeaverJps.root();
2
3  var mutator = new ReturnValueMutator($jp);
4
5  while(mutator.hasMutations()) {
6    // Mutate
7    mutator.mutate();
8
9    // Print mutation
10   print("mutated");
11   println($jp.srcCode);
12
13   // Restore
14   mutator.restore();
15 }
```

Line 1 is responsible for obtaining the root node of the original program's AST. On line 3 we initialize the mutator with the root node obtained previously. It's important to notice that upon creation, the mutator will obtain all nodes where the given mutation, in this case RTV, can be applied.

The while loop on line 5 ensures that the script only exits when all nodes available for mutation are mutated, while iterating to the next mutation available. Line 7 mutates the next node available for mutation. Line 11 displays the program with the created mutation. And, finally, the mutated node is restored to its original state, so that it does not affect affect further mutations.

IV. Developer Manual

This section presents information aimed at developers to allow both understanding of the work already done and how to extend it in the future.

A. Creating a new Mutator

The 1st step is creating a new .lara file. If the creation of several mutators is planned, it is advised the creation of a template file to speed up the initial process of creating a new file. All mutators must start with

```
import lara.mutation.Mutator;
```

Secondly, the constructor is defined. The first parameter is always a joinpoint from where the search for possible mutations is done. The constructor can accept more arguments than those defined explicitly in the constructor parameters. These are accessed with the function arrayFromArgs. The second parameter of this function is the index from which to get extra arguments (inclusive). The constructor always starts with a call to the parent constructor, followed by definition of instance variables, checks on the received values and search for joinpoints to apply mutations.

```
var NewMutator = function($joinPoint) {
  // Parent constructor
    Mutator.call(this);

  // Instance variables
  this.$joinPoint = $joinPoint;
  this.extraArgs = arrayFromArgs(arguments, 1);

  this.toMutate = [];
  this.currentIndex = 0;

  this.$currentJoinpoint = undefined;
  this.$originalJoinpoint = undefined;

  // Check received values

  // Search for target code to mutate and store it,
  ↪  example for if statements
  for($if of WeaverJps.searchFrom(this.$joinpoint,
  ↪  'if').get()) {
    this.toMutate.push($if);
  }

};
```

The created mutator must inherit from Mutator and implement its methods using

```
NewMutator.prototype =
↪  Object.create(Mutator.prototype);

NewMutator.prototype.hasMutations = function()
{ ... }

NewMutator.prototype._mutatePrivate = function()
{
    this.$currentJoinpoint =
    ↪  this.toMutate[this.currentIndex++];

  this.$originalJoinpoint =
  ↪  this.$currentJoinpoint.copy();

  // Modify $currentJoinpoint or get a joinpoint to
  ↪  replace it

  // this.$currentJoinpoint.insertReplace(
  ↪  $newJoinpoint );
  //or
  // this.$currentJoinpoint.attributeX = ...;
}

NewMutator.prototype._restorePrivate = function()
{
    this.$currentJoinpoint =
    ↪  this.$currentJoinpoint.insertReplace(
    ↪  this.$originalJoinpoint );
```

```
  this.$originalJoinpoint = undefined;
  this.$currentJoinpoint = undefined;
}
```

hasMutations is used to check if all mutations have been done. A boolean value is expected as the return.

_mutatePrivate, called as mutate(), is the function where the current joinpoint to mutate is fetched to the respective instance variable and the desired mutation is applied. Saving a copy of the joinpoint before the mutation in a separate instance variable is mandatory as to not lose the original reference. The instance variable with the current joinpoint should also be updated when the mutation occurs to be able to get the new joinpoint to later restore the original.

_restorePrivate, called as restore(), is the function where the code is restored to its original form. To do this, the same replacement process used in the mutations is used, this time, replacing the current joinpoint with the original reference. The two variables should be set as undefined again at the end.

B. Creating a new Mutator Test

The 1st step is creating a new .lara file. If the creation of several mutator tests is planned, it is advised the creation of a template file to speed up the initial process of creating a new file. All mutator tests must start with

```
import weaver.WeaverJps;
import kadabra.mutation.NewMutator;

aspectdef NewMutatorTest
```

Afterward, it is necessary to define the joinpoint(s) that will be used as the root to search for possible mutations. The LARA syntax can be used to search for specific joinpoints or to simply get the root of source code as shown below.

```
var $jp = WeaverJps.root();
```

Finally, the mutator is instantiated and the mutations are applied. The example code displays a loop where each iteration a mutation is applied, displayed on the console and reverted until it covers all available mutations.

```
var mutator = new NewMutator($jp, "|", ["lhs", "^",
↪  "rhs", "&"]);

while(mutator.hasMutations()) {
  // Mutate
  mutator.mutate();

  // Print
  println($jp.srcCode);

  // Restore operator
  mutator.restore();
}
```

C. Mutators Implementation

BWO: Searches for expressions that have the target operator.

```
for($exp of WeaverJps.searchFrom(this.$joinPoint,
↪  'expression').get()) {
  if($exp.operator === bitwiseOperator) {
    this.toMutate.push($exp);
  }
}
```

For unary expressions, remove the unary operator and for binary expressions replace the operator or replace the entire expressions by one of its operands. For each expression goes through all received new operators.

```
BitWiseOperatorMutator.prototype._mutatePrivate =
↪  function() {
  // Obtain new operator, increment index
  var newOp =
  ↪  this.newOperators[this.newOperatorIndex++];

  // Set new operator
  if(newOp === "")
    this.unaryMutate();
  else
    this.binaryMutate(newOp);

}

BitWiseOperatorMutator.prototype.unaryMutate =
↪  function() {
  var toReplace =
  ↪  this.$bitwiseExpression.operand.copy();
  this.$bitwiseExpression =
  ↪  this.$bitwiseExpression.insertReplace(toReplace);
}

BitWiseOperatorMutator.prototype.binaryMutate =
↪  function(newOp) {
  if(newOp === "lhs") {
    var lhs = this.$bitwiseExpression.lhs.copy();
    this.$bitwiseExpression =
    ↪  this.$bitwiseExpression.insertReplace(lhs);
  }
  else if(newOp === "rhs") {
    var rhs = this.$bitwiseExpression.rhs.copy();
    this.$bitwiseExpression =
    ↪  this.$bitwiseExpression.insertReplace(rhs);
  }
  else
    this.$bitwiseExpression.operator = newOp;
}
```

COD: Searches for joinpoints which are of the type if, ternary or loop and that don't have their conditions negated.

```
for($if of WeaverJps.searchFrom(this.$joinpoint,
↪  'if').get()) {
  if($if.cond.instanceOf('unaryExpression') &&
  ↪  $if.cond.operator === '!')
    this.toMutate.push($if);
}

for($ternary of
↪  WeaverJps.searchFrom(this.$joinpoint,
↪  'ternary').get()) {
  if($ternary.cond.instanceOf('unaryExpression') &&
  ↪  $ternary.cond.operator === '!')
    this.toMutate.push($ternary);
}

for($loop of WeaverJps.searchFrom(this.$joinpoint,
↪  'loop').get()) {
  if($loop.cond.instanceOf('unaryExpression') &&
  ↪  $loop.cond.operator === '!')
    this.toMutate.push($loop);
}
```

Replaces the joinpoint of the unaryExpression with its operand thus removing the ! operator.

```
this.$conditional.cond.insertReplace(
↪  this.$conditional.cond.operand.copy() );
```

COI: Searches for joinpoints which are of the type if, ternary or loop and that have their conditions negated.

```
for($if of WeaverJps.searchFrom(this.$joinpoint,
↪  'if').get()) {
  if(!($if.cond.instanceOf('unaryExpression') &&
  ↪  $if.cond.operator === '!'))
    this.toMutate.push($if);
}
```

```
for($ternary of
↪  WeaverJps.searchFrom(this.$joinpoint,
↪  'ternary').get()) {
  if(!($ternary.cond.instanceOf('unaryExpression')
  ↪  && $ternary.cond.operator === '!'))
    this.toMutate.push($ternary);
}

for($loop of WeaverJps.searchFrom(this.$joinpoint,
↪  'loop').get()) {
  if(!($loop.cond.instanceOf('unaryExpression') &&
  ↪  $loop.cond.operator === '!'))
    this.toMutate.push($loop);
}
```

Surrounds the conditional with parenthesis and prepends the ! operator. Replaces the conditional joinpoint with the modified source code.

```
var cond = this.$conditional.cond;
var newSrc = "!(" + cond.srcCode + ")";
cond.insertReplace(newSrc);
```

CDL: Searches for binary expressions in which at least one of the operands is the target constant to delete.

```
for($binaryExpression of
↪  WeaverJps.searchFrom($joinpoint,
↪  'binaryExpression').get()) {
  var $lhs = $binaryExpression.lhs;
  var $rhs = $binaryExpression.rhs;
  if(((this.targetConstant.contains( $lhs.srcCode )
  ↪  && $lhs.isFinal)
    || (this.targetConstant.contains( $rhs.srcCode )
    ↪  && $rhs.isFinal))
    && $binaryExpression.type !== 'boolean') {
    this.toMutate.push($binaryExpression);
  }
}
```

Replaces the found binary expressions with the side that is not the constant. If both sides are the target constant it eliminates one of them.

```
if(this.targetConstant.contains(
↪  this.$node.lhs.srcCode )) {
  this.$node =
  ↪  this.$node.insertReplace(this.$node.rhs);
} else if(this.targetConstant.contains(
↪  this.$node.rhs.srcCode )) {
  this.$node =
  ↪  this.$node.insertReplace(this.$node.lhs);
}
```

CCL: Searches for explicit constructor calls that are not super().

```
for($ref of WeaverJps.searchFrom(this.$joinpoint,
↪  'reference').get().reverse()) {
  // Check it is a constructor call reference
  if($ref.name === "<init>" && $ref.type ===
  ↪  "Executable" && $ref.parent.srcCode !==
  ↪  "super()")
    this.toMutate.push($ref);
}
```

Replaces them with ”null”.

```
this.$referenceParent =
↪  this.$referenceParent.insertReplace("null");
```

IPC: Searches for super calls inside constructors.

```
for($constructor of WeaverJps.searchFrom($joinpoint,
↪  'constructor').get()) {
  // Check if constructor contains a super call in
  ↪  its source code
  if(!$constructor.srcCode.includes('super('))
    continue;
  // Search for super calls inside constructors
  ↪  (since they can only appear inside
  ↪  constructors, we avoid searching the whole
  ↪  program)
  for($descendant of $constructor.descendants) {
    if(  $descendant.instanceOf('call') &&
      $descendant.srcCode.startsWith( 'super(') &&
      $descendant.srcCode.endsWith(')')
    ) {
      this.toMutate.push($descendant);
      break;
    }
  }
}
```

Replaces super() constructor call by a comment indicating the removal.

```
this.$superCall = this.$superCall.insertReplace("//
↪  Super constructor call has been removed");
```

NVC: Searches for non-void method calls.

```
var validAncestors = ['assignment', 'localVariable',
↪  'if', 'loop'];
for(validAncestor of validAncestors) {
  for($ancestor of WeaverJps.searchFrom($joinpoint,
  ↪  validAncestor).get()) {
    /* When the ancestor is an if or a loop, only
    ↪  the ancestors on the condition will be
    ↪  analyzed.
     Because the ones present in the body will be
↪  analyzed by assignment expressions. */
    var descendants = (validAncestor === 'if' ||
    ↪  validAncestor === 'loop') ?
    ↪  $ancestor.cond.descendants :
    ↪  $ancestor.descendants;

    for($descendant of descendants) {
      // Search for non-void calls
      if($descendant.instanceOf('call') &&
      ↪  $descendant.returnType !== 'void') {
        // Store call for later modification
        var mutationValue =
        ↪  typeList.includes($descendant.returnType)
        ↪  ? typeToValue[$descendant.returnType] :
        ↪  'null';
        this.toMutate.push([$descendant,
        ↪  mutationValue]);
      }
    }
  }
}
```

Replaces the method call by default values as mentioned in III. User Manual.

```
NonVoidCallMutator.prototype._mutatePrivate =
↪  function() {
  var mutationInfo =
  ↪  this.toMutate[this.currentIndex++];

  this.$callNode = mutationInfo[0];
  var mutationValue = mutationInfo[1];

  this.originalCallNode = this.$callNode.copy();

  /* Char and String mutation value is a null
  ↪  character, which generates a null node when
  ↪  using it directly with insertReplace.
     A KadabraNodes.literal is required to solve the
↪  problem */
  if(this.$callNode.returnType === 'char' ||
  ↪  this.$callNode.returnType === 'String') {
    var mutatedNode =
    ↪  KadabraNodes.literal(mutationValue,
    ↪  this.$callNode.returnType);
    this.$callNode =
    ↪  this.$callNode.insertReplace(mutatedNode);
  } else {
```

```
    this.$callNode =
    ↪ this.$callNode.insertReplace(mutationValue);
  }
}
```

RMC: Searches for ifs, loops and ternary operators.

```
var conditionalElements = ['if', 'ternary', 'loop'];
for(conditionalElement of conditionalElements) {
  for($element of WeaverJps.searchFrom($joinpoint,
  ↪ conditionalElement).get()) {
    println($element.cond);
    this.toMutate.push($element.cond);
  }
}
```

Replaces their conditions with ”true”.

```
var mutatedCondition = 'true';
this.$conditionalClause =
↪ this.$conditionalClause.insertReplace(mutatedCondition);
```

RTV: Searches for return joinpoints.

```
for($method of WeaverJps.searchFrom($joinpoint,
↪ 'method').get()) {
  // Check it is a method capable of being mutated
  var mutationValue;
  if(methodZeroTypes.contains($method.returnType)) {
    mutationValue = '0';
  } else
  ↪ if(methodTrueTypes.contains($method.returnType))
  ↪ {
    mutationValue = 'true';
  } else {
    continue;
  }

  // Store return statement for later modification
  var methodReturn = WeaverJps.searchFrom($method,
  ↪ 'return').first();
  this.toMutate.push([methodReturn, mutationValue]);
}
```

Replaces them with default values as mentioned in III. User Manual.

```
ReturnValueMutator.prototype._mutatePrivate =
↪ function() {
  var mutationInfo =
  ↪ this.toMutate[this.currentIndex++];

  this.$returnExpression = mutationInfo[0];
  var mutationValue = mutationInfo[1];

  this.originalReturnExpression =
  ↪ this.$returnExpression.copy();

  var mutatedReturn = 'return ' + mutationValue +
  ↪ ';';
  this.$returnExpression =
  ↪ this.$returnExpression.insertReplace(mutatedReturn);
}
```

REFERENCES

[1] H. Coles and T. Laurent and C. Henard and M. Papadakis and A. Ventresque. (2016, July). Demo: PIT a Practical Mutation Testing Tool for Java. presented at ISSTA. Available: https://researchrepository.ucd.ie/bitstream/10197/7748/4/ISSTA_2016_Demo_Camera_ready.pdf

[2] Yu-Seung Ma and J. Offutt and C. Henard and M. Papadakis and A. Ventresque. (2016, July). Description of muJava's Method-level Mutation Operators. Available: https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf

[3] Yu-Seung Ma and J. Offutt and C. Henard and M. Papadakis and A. Ventresque. (2005, November). Description of Class Mutation Mutation Operators for Java. Available: https://courses.cs.ut.ee/LTAT.05.006/2019_spring/uploads/Main/mutopsClass.pdf

[4] P. Ammann and J. Offutt, “Introduction to software testing”, USA: Cambridge University Press, 2016. Available: https://books.google.pt/books?id=bQtQDQAAQBAJ&pg=PA263&lpg=PA263&dq=ior+rename+overriding+method&source=bl&ots=fyaR4041gQ&sig=ACfU3U0x3qLuwbU-L1xYB4kCN6Ic2f702g&hl=pt-PT&sa=X&ved=2ahUKEwit_vCe7JnmAhUJtRoKHegSCAEQ6AEwAHoECAYQAQ#v=onepage&q=ior%20rename%20overriding%20method&f=false