Random Test Generator and Executor

for Java code

Project Report

Nicola Gilberti 198739

University of Trento
MSc Computer Science

Contents

Abstract

Nowadays, software has a big impact in all aspects of our society. More and more companies develop software and testing plays a crucial role for ensuring its quality.

Web applications present new testing challenges with respect to traditional software testing, due to their dynamic and heterogeneous nature. In this document, a tool (Web Random Generator) able to create a test suite for a given application following a random testing approach is presented. WRGen first instruments Page Objects developed for the given application to measure coverage. Then it generates random test cases as sequence of Page Object methods.

Finally, a comparison with a state-of-the-art random generator tool (Evosuite) on 5 web applications is carried out. In terms of coverage, WRGen outperforms Evosuite random generator over all subjects.

Nicola Gilberti
<nicola.gilberti *at* studenti.unitn.it>

1. Background

The project is developed in Java [1] and is based on basic Java programming technologies such as, a Java compiler (Javac [2]), Reflection [3], to dynamically analyze classes at runtime, and Maven [4], to manage dependencies and for the cross-platform compatibility.

In addition to those elements, a Java parser, called Spoon, is used. Next section introduces Spoon and its application in this project.

1.1 Spoon

Spoon [5] is a Java library that gives the ability to create and modify Java source code. Therefore, with this library, a programmer can transform or analyze Java code dynamically.

Spoon takes advantage of the Java Reflection [3] feature to build the abstract syntax tree (AST) of the code under analysis, but also to give a programming interface (intercession API) that lets the user modify and generate Java source code.

Regarding the code that a programmer can insert, Spoon gives different options to check its compliance with the Java grammar. The first option leverages generics to manipulate the AST that give to the programmer a feedback in case of bad code. Another option that Spoon implements is the template engine with static checks that let the programmer insert code automatically ensuring that it is well formed.

How it works

Spoon works primarily on the AST, giving the possibility to manipulate Java code elements. The *Processor* and the *Factory* classes are the key-components to use to analyze and transform the code. The Processor lets the programmer analyze the AST, while the Factory permits to modify the AST, adding and/or removing elements from the syntax tree. The concept is similar to the read and write operations, where the first operation (Processor) is read-only while the second (Factory) can also write.

In particular, the Processor class is utilised for querying Java code elements. That operation is possible thanks to the visit pattern applied to the Spoon model (AST). All the code elements have an *Accept* method, therefore each element can be visited by a visitor object. For instance, in Listing 1.1, a Processor is used to search for an empty catch block. As the code shows, the Processor works on a *CtCatch*,

which is the Compile Time Catch given by the Spoon Metamodel, and checks for statements inside the CtCatch body.

```
public class CatchProcessor extends AbstractProcessor<CtCatch> {
    public void process(CtCatch element) {
        if (element.getBody().getStatements().size() == 0) {
            getFactory().getEnvironment().report(this, Level.WARN, element
    , "empty catch clause");
        }
    }
}
```

Listing 1.1: Processor example taken from Spoon documentation

The Factory class gives the coder the ability to create new elements, and add them to the Syntax Tree under analysis. There is more than one Factory class, where each one is specialized to facilitate the creation of specific code elements.

```
Factory factory = this.getFactory();
String snippet = this.getLogName() + ".testOut(Thread.currentThread())";
CtCodeSnippetStatement snippetFinish = factory.Code().
    createCodeSnippetStatement(snippet);
CtBlock finalizerBlock = factory.Core().createBlock();
finalizerBlock.addStatement(snippetFinish);
ctTry.setFinalizer(finalizerBlock);
CtBlock methodBlock = factory.Core().createBlock();
methodBlock.addStatement(ctTry);
element.setBody(methodBlock);
```

Listing 1.2: Factory example taken from Spoon Projects

In Listing 1.2, a factory object is used to create a *try* block inside an existing Java method. The *Code* method of the factory object is used to create code elements (for example, *snippets*) while the *Core* helps in creating blocks of code, that will eventually contain *Code* generated elements.

2. Test generation Algorithm

This chapter explains all the implementation details and the key-points of the test generation algorithm. WRGen generates Selenium-based end-to-end (E2E) test cases for web applications. Researchers [6] have used *Evosuite* [7] to generate E2E test cases for web applications. Evosuite is a unit test case generator for Java classes. Although Evosuite is a stable tool that is shown to be quite effective in generating test cases for Java classes, it is very complex and its extension to other test generation scenarios (as web applications) is not straightforward. The goal of this project is to implement a test case generator that performs only those operations that are needed for generating Selenium-based (E2E) test cases for web applications.

2.1 Master-slave architecture

The program performs different steps to achieve the final result and each step is performed by a different *worker*. Specifically, the program uses two distinct threads. This architecture, that other test case generators (as Evosuite [7]) implement, can be used to balance the work-load on different processors to increase the efficiency.

The algorithm use threads for specific purposes, defining distinct operations and scopes for each of them. Firstly the main thread start, managing all the imports from resources (see section 3.2) to prepare the environment for the other thread. In fact, the main thread updates dynamically the Java *classpath* in order to recover all the classes needed for the runtime execution.

The second thread (slave) is activated by the main thread (master) once the environment is set up. The second thread generates and executes test cases and sends messages to the main thread with the results of its computations.

The second thread instruments the input class, a Java class containing Page Object methods for a given web application, and generates tests as sequences of those methods (see section 2.4). Those tests are sent back to the primary thread through a *LinkedBlockingQueue* [8], that checks their validity and creates a well-formed test suite. The choice of a LinkedBlockingQueue is a default option to reduce the impact on the memory load during inactivity and to avoid errors due to concurrency at run-time. Even if there can't be simultaneous access from both threads, because the second one generates the message and it restarts creating a new one, the implementation follows the best-practices hints for correct data exchange between threads.

2.2 Class Instrumentor

This section explains how the input class is instrumented. The main dependency for this package is Spoon, explained in section 1.1.

The *ClassModifier* class prepares the Spoon environment. Specifically, it generates all the variables that are necessary to create/modify elements in the Java input class. The program needs to insert extra code in the input class, therefore a *factory* object must be instantiated. In addition, Spoon requires a data model of the class under analysis, as seen in section 1.1. This is generated by a Spoon object that returns, given the source-path of the input class under analysis, the AST of the input class.

```
protected void buildModel(File sourcePath) throws IOException {
    builder = new JDTBasedSpoonCompiler(factory);
    try {
        builder.addInputSource(sourcePath);
        builder.build();
    }catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Listing 2.1: The function to create the AST from the ClassModifier class

Once the AST is created than the class under test can be instrumented. WRGen can instrument the class under test to measure both branch and line coverage. In both cases, when the target (line or branch) is reached during the execution of a test, WRGen uses a list that collects the identifier of the line/branch covered.

In case of branch coverage, the instrumentor searches for *If* structures, and adds a statement (Listing 2.2) in the first line of each *then* and *else* block. A counter is used to uniquely identify a branch.

```
factory.Code().createCodeSnippetStatement("checker.add("+ counter++ +")");
```

Listing 2.2: Snippet creation

In case of line coverage the statement added is the same, while the analysis of the class under test is different. In fact the instrumentor takes as input the lines of the class under test to cover and performs a bottom-up scan searching for the lines. When there is a match it appends the instrumentation snippet shifting one line down the rest of the code.

```
1  private void modConstructors(CtClass cc) {
2      List<?> tmp;
3      CtConstructor<?> constructor;
4      Set<?> constructors = cc.getConstructors();
5      Iterator<?> itc = constructors.iterator();
6      while(itc.hasNext()) {
7          constructor = (CtConstructor<?>) itc.next();
8          CtBlock<?> ctb = constructor.getBody();
9          tmp = ctb.getStatements();
10         List<CtStatement> newStatements = new ArrayList<CtStatement>();
11         CtCodeSnippetStatement newStatement = factory.Code().
       createCodeSnippetStatement("checker = new ArrayList()");
12         newStatements.add(newStatement);
13         for(int j = 0; j < tmp.size(); j++){
14             newStatements.add((CtStatement) tmp.get(j));
15         }
16         ctb.setStatements(newStatements);
17     }
18 }
```

Listing 2.3: The function to add the ArrayList instantiation snippet

The instrumentor (see Listing 2.3) creates extra methods in the class under test that are needed during test case execution, like getting data from the list of targets and resetting it after every test execution.

Finally the instrumentor loops through all the methods of the class under test to store the parameter types each method takes as input. This information is useful during test generation, to generate input values of the right type.

2.3 Input generator

The goal of this chapter is to explain how the primitive input types are generated by WRGen.

Java reflection APIs only offer the possibility to instantiate objects at runtime. Therefore, Java primitive types have to be instantiated using their object representations (*Integer* for *int*, *Character* for *char*...). The type *String* is already an object in the Java language, there is no primitive counterpart.

When the test generator requires a new variable of a primitive type, the primitive input generator creates an object of that type using the default constructor, assigning a fixed value (what is the fixed value?).

Then, it checks if a value of that type was previously generated. If yes, with an equal probability (which is?) the program generates a new value or picks a previously generated one (if no, what happens?). There is a third option for object types that is the *null* value, but it is chosen with a smaller probability (which is and why is it smaller?).

When a previously generated value is chosen, the input generator links the new variable with the one the previously generated value of the same type was assigned to. Otherwise, a random value is generated using the *Random* Java class. For some primitive types the Random class can generate values directly, as in the case of *int*, *float*, *long*, *double*, *boolean*. Listing 2.4 shows how these primitive types are handled by the input generator.

```
case "java.lang.Integer":
    obj.setValue(random.nextInt());
case "java.lang.Float":
    obj.setValue(random.nextFloat());
case "java.lang.Long":
    obj.setValue(random.nextLong());
case "java.lang.Double":
    obj.setValue(random.nextDouble());
case "java.lang.Boolean":
    obj.setValue((random.nextBoolean());
```

Listing 2.4: Random default generator

However, in case of *short*, the input generator generates a random *int* value that is then transformed in a *short* value using the Java type definition for *short*, as shown in Listing 2.5.

```
case "java.lang.Short":
    obj.setValue(random.nextInt(65536) - 32768);
```

Listing 2.5: Random short generator

Char and String require more operations, thus the input generator uses an extra function (see Listing 2.6).

```
public static String generateRandomChars(String candidateChars, int length
    , Random random) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < length; i++) {
        sb.append(candidateChars.charAt(random.nextInt(candidateChars.
    length())));
    }
    return sb.toString();
}
```

Listing 2.6: Random String/char generator

The difference between String and char is the length, which is 1 for char, and it is random generated for String. The variable *candidateChars* in Listing 2.6 is a string of desired characters. For example, it could be the concatenation of upper/lower

case letters and numbers if those are the desired characters a user of WRGen wants in his/her random string values (is it a parameter of your tool?).

In the Java language, there are also other types of objects and one of them is the *Enumerator*. The input generator considers *Enum* as a primitive type and when one is required, it randomly chooses a value from those specified by the particular Enum type. With Enums, the equal operation work differently and so, the links for the usage of old values are not necessary.

If the input parameter to be generated is not of a primitive type, a recursive approach is adopted (see 1). The non primitive type class is analyzed, collecting all its constructors. Then one is randomly picked and, if the constructor requires input parameters, the random generation process is called recursively on each parameter (line 14). If the constructor does not require any input parameter, Java reflection can instantiate it directly. If the same constructor, with at least one input parameter, was previously instantiated with a probability of XX is picked. As in the case of primitive values the reference to the previous variable assignment is saved accordingly.

Algorithm 1 Instantiator

```
1:  procedure INSTANTIATE(params)
2:      for all params do
3:          param ← params.next
4:          if isPrimitive(param) then
5:              param ← INSTANTIATEWITHREFLECTIONOROLDVALUE
6:          else if isEnum(param) then
7:              param ← PICKRANDOMENUMCONSTANT(param)
8:          else if isInterface(param) then
9:              param ← INSTANTIATE(param.getSubclass.getParameter())
10:         else
11:             if directlyInstantiable(param) then
12:                 param ← INSTANTIATEWITHREFLECTIONOROLDVALUE
13:             else
14:                 requireInstances ← INSTANTIATE(param.getParameter())
15:                 param ← INSTANTIATEWITHREFLECTIONOROLDVALUE(requireInstances)
        return params                                  ▷ All the instantiated objects
```

Let us suppose that there is a class A that has to be instantiated. It is neither a primitive type nor an enum, so one of its constructor is picked randomly (line 10 in 1). Suppose that the one picked requires two parameters, like an int type and another class B. Therefore, the generator is called recursively on the int variable and on the class B (line 14). Regarding the int type, the program could generate a new value or pick an old one if it was previously generated (line 5) remind

the probability value to the reader. Suppose that the class *B* has only the default empty constructor, thus Java reflection can instantiate it directly (line 12). Once all the parameters of class *A* are created class *A* can be instantiated with those values (line 15). With this recursive procedure the input generator is able to instantiate parameters of any types.

Regarding interfaces, that cannot be instantiated directly, the input generator has to search in the classpath for all those classes that implement the required interface. Listing 2.7 shows how the interfaces are managed by the input generator.

```
1 ClassPathScanningCandidateComponentProvider provider = new
      ClassPathScanningCandidateComponentProvider(false);
2 provider.addIncludeFilter(new AssignableTypeFilter(target));
3 Set<BeanDefinition> components = provider.findCandidateComponents("./");
4 for (BeanDefinition component : components){
5     Class cls = Class.forName(component.getBeanClassName());
6     // use class cls found
7 }
```

Listing 2.7: how to find a sub-class of an interface

The *ClassPathScanningCandidateComponentProvider* and *BeanDefinition* are taken from the Spring Framework [9]. How is the choice performed when there is more than one class that implements the required interface? Randomly? Once a class that implements the required interface is found what happens?

2.4 Test generator

The test generator takes as input a graph representing the class under test. The graph is a directed graph in which nodes model the states of the web application under test and edges are methods of the class under test that represent transitions, namely actions that lead the web application from one state to another.

explain with figure how the graph looks like (lo aggiungo io)

2.4.1 Test Case Creation

The test generator generates random paths from the given graph representing the sequence of actions (methods of the class under test) to be performed on the web application upon execution. In order to do that, it performs a *random walk* in the graph given a starting node, representing the home page of the web application under test, the maximum length of the path and the *selected* edge that determines the stopping criterion for the random walk. The selected edge is the method of the class under test that needs to be contained in the generated sequence. If the random walk does not traverse the selected edge but it terminates because the maximum path length is reached, the test generator computes the shortest path

from the last node of the path produced by the random walk to the target node of the selected edge.

The selected edge is determined by looping through the list all methods of the class under test. In fact, each method has a unique target that needs to be covered upon execution. Therefore, the test generator, at each step of the test generation, tries to generate a sequence in order that the selected method can be executed properly. If the selected method is then covered upon execution, it is removed from the list.

The method sequence generated with the random walk is *abstract*, meaning that it cannot be executed directly since no input values are specified. The input generator (section 2.3) takes care of generating the proper input values for each method in the sequence. Once the inputs are generated, the method sequence is an actual test case that can be executed.

The test generator instantiates the instrumented class and the methods in the sequence are executed in order. All the targets that are covered upon the test case execution are saved.

The test case generation phase ends when all the targets are covered or a time-out is reached.

2.4.2 Test Suite Creation

The final test suite is then created dynamically based on the coverage of each executed test cases.

The problem of finding a minimal set of test cases that covers all the targets is NP-hard (https://en.wikipedia.org/wiki/Set_cover_problem). WRGen uses a simple greedy heuristic (Listing 2.8). The heuristic starts by adding to the final test suite the test case that covered more targets and then repeatedly adds the test case that covered the targets not covered by the test suite. If a test case covers the same target of another in the final test suite but it has less statements (smaller size), it is selected to be in the final test suite.

```
1  private static void addToFinalTestCase(TestCase newTest) {
2    HashSet<Integer> tmp1 = newTest.getBranchCov();
3    boolean flag=true;
4    if(finalTests.isEmpty()) {
5      finalTests.add(newTest);
6    }else {
7      for(int i=0;i<finalTests.size();i++) {
8        TestCase oldTest = finalTests.get(i);
9        if(sameValues(oldTest.getBranchCov(),newTest.getBranchCov())) {
10         flag=false;
11         if(oldTest.getMethList().size()>newTest.getMethList().size()) {
12           finalTests.set(i, newTest);
13         }
```

```
14          break;
15        }else {
16          if(newTest.getBranchCov().containsAll(oldTest.getBranchCov())) {
17            finalTests.set(i, newTest);
18            flag=false;
19            break;
20          }else if(oldTest.getBranchCov().containsAll(newTest.getBranchCov()
    )){
21            flag=false;
22            break;
23          }else {
24            tmp1.removeAll(oldTest.getBranchCov());
25          }
26        }
27      }
28      if(!tmp1.isEmpty() && flag) {
29        finalTests.add(newTest);
30      }
31    }
32  }
```

Listing 2.8: Check-test method

Once the final test suite is created it has to be printed out in order to be executed. The final test suite is a JUnit test suite. The imports are automatically handled by Spoon and are created when Spoon is used to create the instrumented class. Listing 2.9 shows how to set Spoon correctly for this purpose.

```
1 factory.getEnvironment().setAutoImports(true);
```

Listing 2.9: How to let Spoon automatically manage imports during class generation

Then the test suite class is created. Each test case starts with a common pattern that is as following. First, the JUnit annotation used to identify tests, namely *@Test*. As second element, a comment that gives information about the coverage of the test. More specifically the total coverage of the test, as percentage, and a list of identifiers that represent the lines (or branches) of the class under test covered. The third element is the method declaration with the name of the test, followed by the instantiation of the class under test.

```
1  public class Tester{
2
3      @Test
4      //test case 2 coverage: 0.07906976744186046
5      //branch covered: [96, 161, 98, 163, 133, 201, 170, 171, 109, 206,
       112, 49, 178, 88, 121, 122, 156]
6      public void test0(){
7          main.ClassUnderTest obj = new main.ClassUnderTest();
```

```
8         ...
```

Listing 2.10: Example test case, first part

Here, tests could differ. If the class under test has multiple constructors, the instantiation is preceded by the parameter assignments required by the constructor.

```
1         int var3920 = 1599533343;
2     Id var3910 = new Id(var3920);
3     WidgetFeedTitle var3911 = WidgetFeedTitle.PAGEKITNEWS;
4     WidgetFeedUrl var3912 = WidgetFeedUrl.PAGEKIT;
5         int var3920 = var120;
6     WidgetFeedNumberPosts var3913 = new WidgetFeedNumberPosts(var3920);
7     WidgetFeedPostContent var3914 = WidgetFeedPostContent.SHOWALLPOSTS;
8 try{
9     obj.editFeedWidgetDashboardContainerPage(var3910,var3911,var3912,
      var3913,var3914);
10 }catch(po_utils.NotInTheRightPageObjectException e){}
```

Listing 2.11: example of some possible printing cases

Each test also differs for the method sequence and the execution outputs. The test suite printer is able to manage all the input variable assignments and it also creates try-catch blocks in methods which thrown exceptions upon execution.

Listing 2.11 shows how the method *editFeedWidgetDashboardContainerPage* of the class under test for the application *Pagekit* is called. Tabulation is used to define the depth level of the variables. For example at line 1 there is a *2-Tab* variable meaning that it this is required for a *1-Tab* variable assignment, which is in line 2. All the 1-Tab depth variables are the input for the method of the class under test in line 9. Upon execution, this method throws a *NotInTheRightPageObjectException*, therefore, it is inside a try-catch block. Moreover, the int variable at line 5 is assigned the value of another variable of the same type previously generated (*var120*).

Each test case ends with an assertion, that checks that all the targets covered during the test case generation by that test, are also covered when the same test is executed within the final test suite. This can be done by instantiating the instrumented class under test in the final test suite.

3. Usage

This chapter explain all the features that are necessary for the project to run.
Specifically it will show some *not* well-known tags in the POM, and the properties file necessary for the Environment.
The project on Github contains also a .sh script with an example of how the program can be runned.

3.1 Project Object Model

The POM contains information about the project and various configuration detail used by Maven to build the project.
To compile classes at runtime, a specific *JDK* library has to be imported as a dependency inside the project.
Because it depends on *JDK* and so on the machine architecture, a <*profile*> tag is necessary to identify all the possible options, to maintain the code independent from the machine.

```
<profile>
    <id>windows_profile</id>
    <activation>
        <os>
            <family>Windows</family>
        </os>
    </activation>
    <properties>
        <toolsjar>${java.home}\..\lib\tools.jar</toolsjar>
    </properties>
</profile>
```

Listing 3.1: Windows default profile for jdk lib

There are specific path for each OS type, and Maven accept only specific *OS family*.
<*toolsjar*> is an ad-hoc tag created to define a new property, in this case the path to the tools.jar lib.
This variable will be used during a plugin execution.
In the next listing 3.2, there are few element to consider. The phase in which this operation work is decidable by the programmer, in this case the *install* phase is selected.
The configuration of the library is based on the one installed on your machine, but in this case the *JDK* library has no different option if the OS change.

The version is the value that the runner of the program *MUST* verify.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-install-plugin</artifactId>
    <version>2.5.2</version>
    <executions>
        <execution>
            <id>install-external</id>
            <phase>install</phase>
            <configuration>
                <file>${toolsjar}</file>
                <repositoryLayout>default</repositoryLayout>
                <groupId>com.sun</groupId>
                <artifactId>tools</artifactId>
                <version>1.8.0</version>
                <packaging>jar</packaging>
                <generatePom>true</generatePom>
            </configuration>
            <goals>
                <goal>install-file</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Listing 3.2: How to 'mvn install' the tools.jar

3.2 Properties

A .properties is a file extension for files mainly used in Java related technologies to store the configurable parameters of an application.
This project require information about the project that has to analyse and about the specific class to instrument.

- **FileName**:
  It is required the name of the Class to test, with the complete package route.
  E.g.:FileName = com.main.ClassUnderTest
- **ProjectName**:
  It is required the name of the Project that contains the class.
- **PathToProjectDir**:
  The program has to know where the project you are referring to is placed.
- **LineToCover**:
  The line-coverage has to know which line to instrument.
  E.g.:LineToCover = 33:47:60:72:85:99:115:132:144:157

- **Separator**:
  Lines in the previous tag are a list of line numbers divided by a separator. The default accepted is ':' but it can be used a different one defining it here.
- **ExecutionTestTimer**:
  This is the time the algorithm has to find tests. It is in millisecond, so 60000 is equal to 1 minute.
- **MaxNumberOfMethodXTest**:
  Each test can contain a random number of method calls. This attribute is an upper bound to it.
- **GraphName**:
  As seen in 2.4, this project work generating a sequence of method following a graph.
- **GraphDirPath**:
  The program has to know where the graph you are referring to is placed.
- **StartNodeGraph**:
  It is the starting point of the graph path analysis.
- **RequiredPath**:
  The program use reflection to execute the external project written in the first item, but it is not able to know all the dependencies it need.

4. Empirical evaluation

This chapter focus on the evaluation of the proposed algorithm, making a comparison between this and the Evosuite method. The comparison is done executing 10 time both programs on the applications and analyzing the output in terms of coverage and number of test executed to define the final testsuite. The evaluation is performed by statistical tests.
The distribution of the result cannot be assessed, caused by the random approach followed, so a t-test can't be performed. Furthermore the number of tests make useless the use of a t-test or similar parametric analysis.
However, the *Wilcoxon rank sum test*, also known as Mann–Whitney U test, avoid those limitations. It quantifies significant differences between the results, considering a significance level of 0.05.
To ensure the correctness of results, the analysis use also the *Vargha and Delaney* statistic. This test emphasize the magnitude of the differences, given by the U test, between the two algorithms.

Table 4.1: Evaluation of the application on web services

Test details			Statistic		Wilcoxon rank sum test		Vargha and Delaney
WebSite	Analysis	Application	Mean	Standard Deviation	W	p-value	A
Dimeshift	Coverage	WRGen	35	6.236096	82	0.01684	0.82 (large)
		Evosuite	28.9	3.813718			
	# Test	WRGen	24	1.414214	79.5	0.02631	0.795 (large)
		Evosuite	21.8	2.347576			
Retroboard	Coverage	WRGen	68.9	4.72464	100	0.0001433	1 (large)
		Evosuite	53.9	2.330951			
	# Test	WRGen	26.1	2.233582	43.5	0.6436	0.435 (negligible)
		Evosuite	26.3	2.110819			
Pagekit	Coverage	WRGen	19.7	3.093003	93	0.001218	0.93 (large)
		Evosuite	14.1	2.378141			
	# Test	WRGen	4	0.8164966	17.5	0.01108	0.175 (large)
		Evosuite	5.2	0.9189366			
Phoenix	Coverage	WRGen	59.2	8.283853	86.5	0.00599	0.865 (large)
		Evosuite	47.9	8.238797			
	# Test	WRGen	9.2	1.032796	23.5	0.04131	0.235 (large)
		Evosuite	10.3	1.567021			
Splittypie	Coverage	WRGen	40.4	4.526465	96	0.0005501	0.96 (large)
		Evosuite	26.2	7.568942			
	# Test	WRGen	17.2	3.224903	6.5	0.0008922	0.065 (large)
		Evosuite	21.9	2.078995			

The evaluation is also executed limiting the number of methods callable for each unit test, *@Test*, but letting the test suite to grow in number of unit test infinitely. The execution time is also restricted, so the program has not to run endlessly and eventually halt when it is reached the total coverage.
Respectively, the maximum number of method is *40* while the time is *60 s*, and they are given as input with the properties file 3.2.
The table 4.1 show the results for each application under analysis.
The results on the web applications, give us some interesting results about the algorithm. In all the coverage analysis, the algorithm presented has a greater performance compared to the counterpart. In fact, the p-value is below the agreed risk of 5 percent (0.05), so one significant difference can be assumed.
The Vargha and Delaney value confirm each result, pointing the magnitude of the differences to *large*, in favour of the Web Random Generator. The result, in case of number of tests, are slightly different than before.

Figure 4.1: Dimeshift Analysis

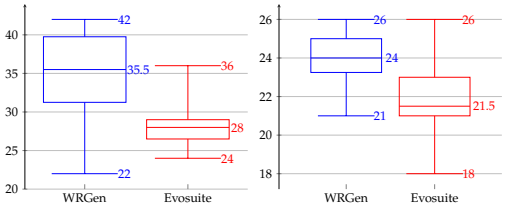
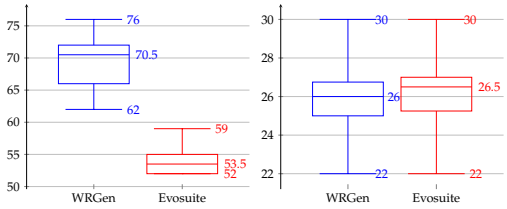


Figure 4.2: Retroboard Analysis



The first service, *Dimeshift*, point out that the WRGen has an higher value, however the number of test in each test-suite is preferable to be as thin as possible. Thus, even if the WRGen has a capacity to cover better the Web-app, in this case it require also a greater test-suite to pursue the scope.
We can check this results also with the box-plot representation 4.1. For the coverage is simple to find how the WRGen has a greater variance, possible synonym of a correct random generation. The number of tests, also, show that Evosuite on average require fewer tests to pursue the goal.
Nevertheless, the *Retroboard* application results on the test analysis are quite different. The p-value is higher than the risk, so the Wilcoxom test tells us nothing. Moreover the A value see no differences on the result of the two under test programs. With the box-plot 4.2, in fact, we can see that the behaviour of the two algorithm is the same.
Still, the WRGen is more performer, with a higher coverage compared to the same number of test executed.

Figure 4.3: Phoenix Analysis

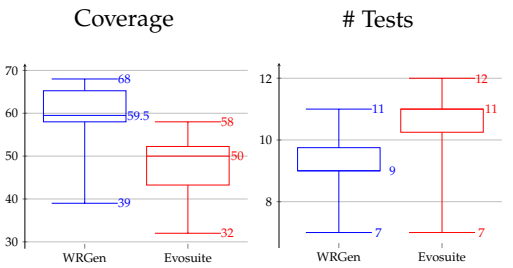


Figure 4.4: Splittypie Analysis

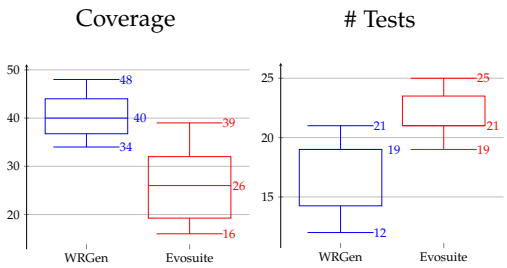
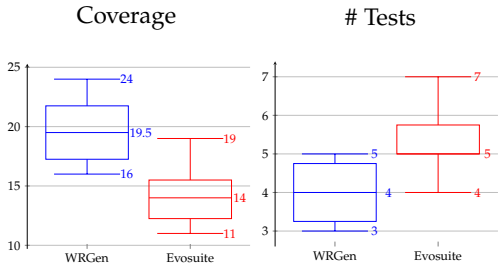


Figure 4.5: Pagekit Analysis



With these last three services, the results are similar.
As defined before, The U test accordingly with the A value tell that the test-suite created by the WRGen has a higher coverage in respect of the counterpart.
Additionally, the Vargha and Delaney method on the number of tests define that the Evosuite approach leads to generate a higher number of test. However, the higher number of possible final tests do not help the algorithm to find out a better test-suite. As before, the results are easily checkable with the graphical representation.

5. Conclusion

Starting from the evaluation, the method proposed has a better behaviour in terms of coverage and also number of test necessary. This can be caused by the great number of operation the Evosuite approach has to overcome, while this technique is ad-hoc for the purpose.
The evaluation is performed only on five application, and the constraints limit both programs, hence an higher number of Web-services or a different set of restriction can be defined to test new behaviours.
Despite this, the results are very positive and give solid feedback on the project. Since now the algorithm proposed can afford the problem of the automatic test generation, providing an adequate test-suite as output.
The technology used have a great impact on the performances, so a different approach on the instrumentation part could lead to different results.
In case of line coverage the technology used can be overwhelming, while an ad-hoc parser could perform the same operation with cheaper time/computation effort. The library used is a kind of parser, indeed, and a major number of external dependencies can be expensive too.
About the general instrumentation phase other techniques, like byte-code manipulation or other source-code handler, could make differences.
Moreover, the printing phase require extra workload during the instantiation steps, and other solution can be adopted. The use of a complex structure can reduce some computational/memory costs for a giant number of instantiations per methods. Nonetheless, the complex structure has to be chosen wisely, or the benefit are no longer available.
In conclusion, despite the limited tests, results are still impressive and the proposed algorithm seems to work correctly. As always, future work can be done to improve the performances and the efficiency on test generation phase.

So, in case you have questions, comments, suggestions or have found a bug, please do not hesitate to contact me. You can find my contact details below.

Nicola Gilberti
UniTN MSc student
nicola.gilberti *at* studenti.unitn.it

Bibliography

[1] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2000.

[2] Wikimedia Foundation Inc., "*javac*." Available on `https://en.wikipedia.org/wiki/Javac`.

[3] Oracle, "*Reflection in Java*." Available on java.lang.reflect package description.

[4] R. Bharathan, *Apache Maven Cookbook*. Packt Publishing Ltd, 2015.

[5] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015.

[6] M. Biagiola, F. Ricca, and P. Tonella, "Search based path and input data generation for web application testing," in *International Symposium on Search Based Software Engineering*, pp. 18–32, Springer, 2017.

[7] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), pp. 416–419, ACM, 2011.

[8] Oracle, "*LinkedBlockingQueue in Java*." Available on LinkedBlockingQueue<E> Docs.

[9] Pivotal.io, "*Spring Framework*." Available on Spring Framework Docs.