A Tutorial on Using and Extending the EvoSuite Search-Based Test Generator

Gordon Fraser

University of Passau, Germany
gordon.fraser@uni-passau.de

Abstract. EvoSuite is an automated unit test generation tool for Java. It takes as input a Java class under test, and produces JUnit tests optimised for code coverage, and enhanced with regression assertions, as output. This paper is a tutorial on how to use EvoSuite to generate tests, on how to build and extend EvoSuite, and how to use EvoSuite to run experiments on search-based testing.

1 Introduction

EvoSuite [6] is a tool that automatically generates JUnit test cases for Java classes. It applies search-based techniques, such as genetic algorithms, to generate these tests. Besides various optimisations at the algorithmic level proposed over time (e.g., [3,9,11]), EvoSuite also implements many different Java-specific optimisations (e.g., mocking of interactions with the filesystem [4] or network [5]) and has reached a good level of maturity (read: it does not crash *too* often). While the principle techniques underlying EvoSuite and their empirical evaluations have been published (e.g., [7]), the aim of this article is to provide an introduction to the tool from a user and researcher point of view.

The tutorial is structured in three parts: First, we describe how to generate tests with EvoSuite from the command line. Second, we show how to build and extend EvoSuite. Finally, in the third part we provide an example of how EvoSuite can be used to run experiments, for example to evaluate different configurations or extensions to EvoSuite. This tutorial covers a subset of the online tutorial available at `http://www.evosuite.org`.

2 Using EvoSuite

There are plugins [2] to use EvoSuite within different IDEs (e.g., IntelliJ and Eclipse), and there is a Maven plugin that simplifies the usage in larger projects. In this tutorial, however, we will focus on the basic use case as a standalone application, on the command line. For this, EvoSuite is available as an executable `jar` (Java Archive) file. The latest release of EvoSuite is always available at `http://www.evosuite.org/downloads/`, or in the release section on EvoSuite's GitHub page at `http://github.com/EvoSuite/evosuite/`. At the time of this writing, the latest release version was 1.0.6; the filenames stated in this article refer to this version number, but obviously new releases will lead to changed filenames. There are two different `jar` files:

- `evosuite-1.0.6.jar` is the main file used to generate tests, including all its dependencies.
- `evosuite-standalone-runtime-1.0.6.jar` is an archive containing only those parts of EvoSuite and its dependencies that are necessary in order to execute tests generated by EvoSuite.

In this tutorial, we will assume that you have these `jar`-files. Furthermore, for several parts of the tutorial you will need Apache Maven[1].

2.1 Invoking EvoSuite

As the name suggests, the executable jar file can be executed. To do so, call EvoSuite like this:

```
java -jar evosuite-1.0.6.jar
```

You should see the following output:

```
* EvoSuite 1.0.6
usage: EvoSuite
...
```

This output is EvoSuite listing all the possible command-line options, as we haven't told EvoSuite what to do yet. To make the rest of this tutorial easier to read, we will create an environment variable to point to EvoSuite, e.g.:

```
export EVOSUITE="java -jar $(pwd)/evosuite-1.0.6.jar"
```

Now we can simply invoke EvoSuite by typing:

```
$EVOSUITE
```

(If you are not using the Bash shell, the commands to create an alias `$EVOSUITE` might differ.)

2.2 Generating Tests

As a running example in this tutorial, we will use the `tutorial.Stack` class shown in Figure 1. We will assume that this file is part of a standard Java project structure, where the source code of the `Stack` class is kept in the file `src/main/java/tutorial/Stack.java`, and the compiled bytecode is placed in the directory `target/classes`. You can find a project set up like this as a Maven project in our online tutorial[2].

[1] https://maven.apache.org/
[2] http://evosuite.org/files/tutorial/Tutorial_Stack.zip

```java
package tutorial;

import java.util.EmptyStackException;

public class Stack<T> {
    private int capacity = 10;
    private int pointer = 0;
    private T[] objects = (T[]) new Object[capacity];

    public void push(T o) {
        if(pointer >= capacity)
            throw new RuntimeException("Stack exceeded capacity!");
        objects[pointer++] = o;
    }

    public T pop() {
        if(pointer <= 0)
            throw new EmptyStackException();
        return objects[--pointer];
    }

  public boolean isEmpty() {
        return pointer <= 0;
    }
}
```

Fig. 1. Example Java class `tutorial.Stack` used in the tutorial.

To generate tests with EvoSuite, there are two essential pieces of information that EvoSuite needs: (1) What is the class under test, and (2) what is the classpath where it can find the bytecode of the class under test and its dependencies. The class under test is specified using the `-class` argument (assuming we are targeting a single class). Note that we need to use the fully qualified class name; that is, we need to include the package name. Thus, in our example, we need to use `-class tutorial.Stack`.

The classpath is specified using the `-projectCP` argument. This takes a regular classpath entry, like you would specify when using `java -cp` or by setting `export CLASSPATH=...`. As we assumed that compiled bytecode is placed in `target/classes` (as is, for example, done by Maven), this is the classpath which we specify using `-projectCP target/classes`. Thus, we can now run EvoSuite as follows:

```
$EVOSUITE -class tutorial.Stack -projectCP target/classes
```

Note that this assumes that the `Stack` class has been compiled, and there exists a resulting file `target/classes/tutorial/Stack.class`. If you don't have this and don't know how to produce it, consider getting the example project set up[2]. If everything worked correctly, then EvoSuite has now produced two files:

```
evosuite-tests/tutorial/Stack_ESTest.java
evosuite-tests/tutorial/Stack_ESTest_scaffolding.java
```

Let's take a closer look at these two files. If we look into the scaffolding file, we'll see lots of things happening in methods annotated with `@Before` and `@After`. These are JUnit annotations which ensure that these methods are executed before/after execution of each individual test. The reason for all this is that EvoSuite avoids flaky tests by controlling everything that might be non-deterministic. The scaffolding ensures that tests are always executed in the same consistent state, so they should really only fail if they reveal a bug, not because they are flaky. The scaffolding may look a bit scary, but the good news is that you'll probably never need to look at it.

The tests are in the main `Stack_ESTest.java` file. The test class inherits from the scaffolding, such that all the setup/pulldown happens without showing all the overhead to ensure tests are not flaky:

```
@RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism
    = true, useVFS = true, useVNET = true, resetStaticState = true,
    separateClassLoader = true)
public class Stack_ESTest extends Stack_ESTest_scaffolding {
   // ...
```

Besides inheriting from the scaffolding, we also see some annotation that is specific to EvoSuite. The test class declares that it will be executed with the EvoRunner, rather than a default JUnit runner. The test runner takes a couple of parameters that tell it which parts of the execution environment are controlled. You can safely ignore these for now – the values for these parameters are set automatically by EvoSuite.

The rest of the file consists of the actual tests. The tests use JUnit 4 and are annotated with `@Test`. Because automatically generated tests sometimes do silly things causing infinite loops, all tests have a specified timeout, with a default value of 4 seconds.

2.3 Running Tests

Let's compile the tests. The compiler will need several things on the classpath:

– `target/classes`: This is the classpath directory containing the compiled bytecode, which we need for the `tutorial.Stack` class.
– `evosuite-standalone-runtime-1.0.6.jar`: This is the EvoSuite runtime library (you can also use the full EvoSuite jar file instead of this, although that will lead to more output since it uses EvoSuite's logger configuration).
– `evosuite-tests`: This is the root directory where EvoSuite put the test class files.
– `junit-4.12.jar` and `hamcrest-core-1.3.jar`: We need JUnit to execute JUnit tests.

To automatically resolve the JUnit and Hamcrest dependencies, an easy way is to use the Maven-version of our example project[2] and use Maven to retrieve the dependencies:

```
mvn dependency:copy-dependencies
```

This will download the two jar files and put them into `target/dependency`.

Now we need to tell the Java compiler where to find all these things, for which we set the CLASSPATH environment variable:[3]

```
export CLASSPATH=target/classes:evosuite-runtime-1.0.6.jar:\
       evosuite-tests:target/dependency/junit-4.12.jar:\
       target/dependency/hamcrest-core-1.3.jar
```

For now, we will simply compile the tests in place. Check the online tutorial[4] if you want to see how to integrate EvoSuite into the Maven project properly, such that Maven takes care of compiling the tests. Type the following command:

```
javac evosuite-tests/tutorial/*.java
```

Check that there are the two .class files in `evosuite-tests/tutorial`. If they are not there, then check what error messages the Java compiler gave you – most likely some part of the classpath is not set correctly. If they were compiled correctly, we can now run the tests on the commandline:

```
java org.junit.runner.JUnitCore tutorial.Stack_ESTest
```

If you followed all the steps so far correctly, you should see the following output:

```
JUnit version 4.12
.....
Time: 2.021

OK (5 tests)
```

Congratulations! You just generated and executed an EvoSuite test suite!

2.4 Configuring EvoSuite

Now let's take a closer look at how we can influence what EvoSuite does. First, we had to wait quite a while until test generation completed – even though this is such a simple class. A simple way to tell EvoSuite that we've waited long enough for test generation is to simply hit Ctrl+C while it is generating tests. EvoSuite will stop the search, and write the test cases generated up to that point. If you

[3] Note that, as is common, wrapped lines at the commandline are indicated with a backslash "\" in this paper. These lines are only wrapped to fit the text in the paper, you can also type these commands on a single line.

[4] `http://www.evosuite.org/documentation/tutorial-part-2/`

hit Ctrl+C a second time, this will kill EvoSuite completely. To try this out, generate some more tests:

```
$EVOSUITE -class tutorial.Stack -projectCP target/classes
```

After a couple of seconds, when you think coverage is sufficient, hit Ctrl+C and wait for the tests to be written. If you wait 10-20 seconds, you will notice that the tests we got still cover all the lines in the `Stack` class. So why does EvoSuite take so long? The reason is that EvoSuite by default targets not only lines of code, but attempts to satisfy a range of different testing criteria, including things like mutation testing. Some of the testing goals described by these criteria are infeasible, which means that there exist no tests that satisfy; some other goals are just so difficult to cover that EvoSuite cannot easily produce the tests. This is a well-known aspect of test generation, and to deal with it, EvoSuite uses a fixed amount of time for test generation, and stops generating tests once this time has been used up. By default, this is 60 seconds. If we want to change this, then besides manually stopping EvoSuite, we have two options: Either we change the testing criteria to avoid the stronger criteria that may not be satisfiable, or we set the timeout explicitly.

Let's start by generating tests for a weaker criterion. We'll use branch coverage, which requires that all if-conditions evaluate to true and false, and all lines of code are covered. We can set the criterion using the `-criterion` argument. To generate branch coverage tests, type:

```
$EVOSUITE -class tutorial.Stack -projectCP target/classes \
          -criterion branch
```

EvoSuite will work for a couple of seconds, but once it has reached 100% branch coverage it will terminate and give us a branch coverage test suite.

Alternatively, we can tell EvoSuite how much time to spend on test generation. EvoSuite uses search-based techniques, so the time it spends on test generation is called the search budget. Unlike the target criterion, the search budget is not a command line argument, but one of many properties that configure how EvoSuite behaves. To set properties, we can use the `-Dproperty=value` command line argument. For example, to specify the search budget to 20 seconds, we would use the following command:

```
$EVOSUITE -class tutorial.Stack -projectCP target/classes \
          -Dsearch_budget=20
```

EvoSuite has many properties that can all be set using the `-Dproperty=value` syntax. To get an overview of the available properties, type the following command:

```
$EVOSUITE -listParameters
```

For example, by default EvoSuite will apply minimization to test cases, which means that it removes all statements that are not strictly needed to satisfy

```
package tutorial;

import org.junit.Test;
import org.junit.Assert;

public class StackTest {
  @Test
  public void test() {
    Stack<Object> stack = new Stack<Object>();
    stack.push(new Object());
    Assert.assertFalse(stack.isEmpty());
  }
}
```

Fig. 2. Manually written test class for the `Stack` class.

the coverage goals; this can be deactivated using `-Dminimize=false`. EvoSuite also minimizes the assertions it adds, and this can be changed by switching the assertion generation strategy, e.g. to `-Dassertion_strategy=all`. Thus, to generate long tests with loads of assertions we could use the following command:

```
$EVOSUITE -class tutorial.Stack -projectCP target/classes \
          -Dsearch_budget=20 -Dminimize=false -Dassertion_strategy=all
```

2.5 Working with existing tests

Let's assume we have previously written some tests for our `Stack` class manually. For example, suppose the file `src/test/java/tutorial/StackTest.java` contains a test suite consisting of a single test shown in Figure 2. This is not a very exciting test, and also one that EvoSuite could easily generate. However, in practice you might have already written some tests at the point you invoke EvoSuite, and so maybe you don't want to see generated tests for code you have already covered.

We can tell EvoSuite to only output tests that are not already covered using the `junit` property. For example, to tell EvoSuite to only give us tests that are not already covered by `tutorial.StackTest`, we would set the property using `-Djunit=tutorial.StackTest`. If we have multiple test classes, we can use a colon-separated list for the property.

We also need to tell EvoSuite where to find this test, as it needs to execute the test. So let's first make sure that the test is compiled and passes. If we have set up our project as a Maven project, we can simply run the following command:

```
mvn test
```

(If you are not using Maven or the example project provided online, you can also invoke `JUnitCore` as described above, but with the corresponding classname). This should give you the following output (among some other messages):

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
 Running tutorial.StackTest
 Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
     0.091 sec

 Results :

 Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

If the test doesn't pass then most likely you have edited (and broken?) the `Stack` class and should fix it.

If you are using Maven to run tests, then for EvoSuite the interesting part is that Maven placed the bytecode of this test into the directory `target/test-classes`. If we want to know how great this test suite is, we can ask EvoSuite to measure the coverage for us. EvoSuite supports the command `-measureCoverage`, and we need to specify the class under test (`-class tutorial.Stack`), the tests we are interested in (`-Djunit=tutorial.StackTest`), the classpath containing the class under test and the tests (`-projectCP target/classes:target/test-classes`), and optionally, which criteria we are interested (e.g., `-criterion branch`):

```
$EVOSUITE -measureCoverage -class tutorial.Stack
    -Djunit=tutorial.StackTest \
          -criterion branch -projectCP
              target/classes:target/test-classes
```

This should give you the following output (among other messages):

```
* Total number of covered goals: 3 / 7
* Total coverage: 43%
```

If we now only want to have tests that cover the remaining 4 branch coverage goals, we would invoke EvoSuite as follows:

```
$EVOSUITE -class tutorial.Stack -Djunit=tutorial.StackTest \
          -projectCP target/classes:target/test-classes \
          -criterion branch
```

Take a look at the file `evosuite-tests/tutorial/Stack_ESTest.java` to check that it worked.

2.6 Running EvoSuite on multiple classes

Our example project only has a single class, so all calls to EvoSuite so far used the argument `-class`. However, sometimes we might want to target more than just a single class, for example when generating a regression test suite. In this case, we can replace the `-class` argument with either `-prefix` or `-target`.

The `-target` argument specifies a classpath entry (e.g., directory or jar file). EvoSuite will then be invoked sequentially on every testable class it can find in that classpath entry. If you want to know which classes EvoSuite thinks are testable (e.g., public), then type the following command:

```
$EVOSUITE -listClasses -target target/classes
```

Since our example project only contains one class, the output should be just our example class:

```
tutorial.Stack
```

To invoke EvoSuite on all the classes in a classpath entry, type the following:

```
$EVOSUITE -target target/classes
```

EvoSuite will now go and test each class it finds, one at a time. Alternatively, we might want to test all classes in a certain package. To test all classes in the `tutorial` package, type the following command:

```
$EVOSUITE -prefix tutorial
```

As our project has only one class this will again just test the Stack.

The arguments `-target` and `-prefix` will run EvoSuite sequentially on each class they find. If your project is large, this might not be the ideal strategy. In fact, if your project is large and you want to use EvoSuite repeatedly, you will probably not want to run things manually on the command line, but instead use Maven to automate and parallelise things. This is not covered in this paper, but you can find a tutorial for this online[4].

3 Extending EvoSuite

EvoSuite is not only intended to serve as a test generator for developers, but also as a platform to support experimentation in search-based software testing. Often, this involves modifying or extending EvoSuite. In this section, we take a look at how one can build EvoSuite from sources, and how one can extend it.

3.1 Obtaining the EvoSuite source code

The source code of EvoSuite is available on GitHub in a public Git repository. The first step of this part of the tutorial thus consists of checking out the source code. How to do this will differ depending on which IDE you prefer to use. On the command line, we would check out the repository with Git directly:

```
git clone https://github.com/EvoSuite/evosuite.git
```

The source code is organised into several Maven sub-modules. That is, there is one parent `pom.xml` in the main directory of the source code you just checked out, and then there are several separate sub-projects in subdirectories. Let's have a closer look at the main sub-modules:

– `master`: EvoSuite uses a master-client architecture because things can go wrong when executing randomly generated tests (e.g., we could run out of memory). The client sends the current search result to the master process every now and then, so that even if things go wrong, we still get some tests in the end. The `master` module handles the user input on the command line (e.g., parsing of command line options), and then spawns client processes to do the actual test generation.
– `client`: The client contains all the heavy lifting. The genetic algorithm is in here, the internal representation of test cases and test suites used by the algorithm, the search operators, mechanisms to execute the test cases, all the bytecode instrumentation that is needed to produce trace information from which to calculate fitness values.
– `runtime`: This is the runtime library, i.e., all the instrumentation that is needed to make test execution deterministic, the mocked Java API, etc.
– `plugins`: There are several sub-projects in here that are plugins for various third-party tools, such as Maven, IntelliJ, Eclipse, or Jenkins.

Besides these, there are several other modules or sub-directories. You will not usually need to access any of these, but in case you are curious what they are:

– `standalone_runtime`: There is no source code in this library, this is simply a Maven sub-module that produces a standalone jar file, i.e., one that includes all the dependencies of the runtime library.
– `shaded`: There is no source code in here either; this is a Maven module that produces a version of EvoSuite where the package name is renamed from `org.evosuite` to something else. This is to allow EvoSuite to be applied to itself (which otherwise wouldn't work, as EvoSuite refuses to instrument its own code).
– `generated`: This is a sub-module in which we are putting tests generated by EvoSuite to test EvoSuite. This is still work in progress.
– `release_results`: This is not a Maven sub-module, it is just a collection of data that represents the results of the experiment on the SF110 dataset we conduct every time we perform a release.
– `src`: No Java source code in here, only some Maven-related meta-data.
– `removed`: Some source code files that are not used in the main source tree but have been useful to keep as a reference.

3.2 Building EvoSuite

If you know Maven, then it will probably not come as a surprise to you that, using Maven, Evosuite can be compiled using:

```
mvn compile
```

Most likely, your IDE will do this for you automatically. However, it is important that your IDE supports Maven, and that you have configured the project as a

Maven project. If you haven't done this, what you will get are error message complaining that the compiler cannot find classes in the package `org.evosuite.xsd`. These classes are generated automatically by jaxb based on an XML schema – and this is only done if you properly compile the project with Maven.

Recall that the EvoSuite distribution consists of two jar files – one with the standalone runtime dependencies, and one for test generation. You can generate these by invoking:

```
mvn package
```

The main EvoSuite jar file is generated in the master sub-module: `master/target`. You can validate that this is the case by invoking the executable with Java:

```
java -jar master/target/evosuite-master-1.0.7-SNAPSHOT.jar
```

You should now see the help text with the usage instructions. The standalone runtime library is in directory `standalone_runtime/target/`.

Building EvoSuite can take a while, but a lot of that time is spent executing unit tests. Although we don't recommend doing that, if you do need to build a jar file quickly and can't wait for the unit tests to complete, you can add `-DskipTests` to the Maven command line.

3.3 Testing EvoSuite

As with any Maven project, you will find the source code in `src/main/java` for every sub-module, and the tests in `src/test/java`.

EvoSuite has a fair number of unit tests, but it has a lot more system and integration tests (executing all system tests takes somewhere between 1-2 hours, depending on your machine). You can distinguish between the two types of tests based on the classname: all system tests have the suffix `SystemTest` in their name. Most of these system tests consist of a class under test that captures a specific testing challenge, and then invoke EvoSuite to check that it is able to cover the class fully, using a specific configuration.

In the test directories of the various sub-packages, you will find two main packages of classes: Everything with a package name starting with `org.evosuite` are the actual tests; the package `com.examples.with.different.packagename` package contains example classes under test used in the tests.

Let's take a closer look at one of the system tests. For example, open the class `org.evosuite.basic.NullStringSystemTest`, which you can find in the file `master/src/test/java/org/evosuite/basic/NullStringSystemTest.java` (Fig. 3).

The first thing worth noting is that this system test extends `SystemTestBase`. This is important for system tests, as it resets the state of EvoSuite (e.g., properties) and prepares everything for test execution (e.g., classpath). It also sets a couple of important properties for tests - if you are interested to see which ones they are, check out method `setDefaultPropertiesForTestCases` in the `SystemTestBase` class. In particular, it sets this property:

```
Properties.CLIENT_ON_THREAD = true;
```

```java
1  public class NullStringSystemTest extends SystemTestBase {
2
3    @Test
4    public void testNullString() {
5      EvoSuite evosuite = new EvoSuite();
6
7      String targetClass = NullString.class.getCanonicalName();
8
9      Properties.TARGET_CLASS = targetClass;
10
11     String[] command = new String[] { "-generateSuite", "-class",
           targetClass };
12
13     Object result = evosuite.parseCommandLine(command);
14     GeneticAlgorithm<?> ga = getGAFromResult(result);
15     TestSuiteChromosome best =
16         (TestSuiteChromosome) ga.getBestIndividual();
17     System.out.println("EvolvedTestSuite:\n" + best);
18
19     int goals = TestGenerationStrategy.getFitnessFactories().get(0)
20          .getCoverageGoals().size(); // assuming single fitness
              function
21     Assert.assertEquals("Wrong number of goals: ", 3, goals);
22     Assert.assertEquals("Non-optimal coverage: ", 1d,
23                         best.getCoverage(), 0.001);
24   }
25 }
```

Fig. 3. Example system test checking that EvoSuite can assign `null` values to parameters of type `String`.

This tells EvoSuite not to spawn a new process for the client (i.e., the part that runs the search and executes the tests). The reason for this is that a standard Java debugger will only allow you to work in the process it is attached to, not in any child processes spawned. So, if you want to, for example, set some breakpoints, it is essential that `Properties.CLIENT_ON_THREAD` is set to true, otherwise the debugger will not be involved when the breakpoint is passed.

The `testNullString` test starts by creating a new instance of EvoSuite (Line 5); then, it tells EvoSuite what the class under test is, by setting the property `Properties.TARGET_CLASS` to the fully qualified name of the class under test. As you can see, if you want to set any specific properties of EvoSuite for your test, you can simply set them in the test. The SystemTestBase will ensure that these properties are reset to their defaults after test execution. In our example, the class under test is `NullString`, which the class shown in Figure 4. On this class, we can only achieve 100% branch coverage if EvoSuite is able to provide a null and a non-null value for String parameters. Thus, this class serves to test whether EvoSuite properly supplies null values for strings.

```java
package com.examples.with.different.packagename;

public class NullString {

  public boolean isNull(String s){
    if(s==null){
      return true;
    } else {
      return false;
    }
  }
}
```

Fig. 4. `NullString` example class that is used as a target to check if EvoSuite can produce `null` values as parameters for methods that expect `String`s.

The test next invokes EvoSuite for the target class in Line 13. This essentially is the same as calling EvoSuite on the command line and passing in some arguments, which are captured in the `command` array here. EvoSuite will then generate some tests, and return an object that summarizes the test generation. `SystemTestBase` provides a helper function `getGAFromResult` to extract the genetic algorithm instance from this result object, called in Line 14. This GA object can be queried about various things, and most importantly, we can ask it for the best individual, i.e., the result of the test generation; this is done in Line 16. Given this test suite, we can do what we want with it – for example print it to stdout, like done in Line 17. Or, more importantly, we can write some assertions to check that the result is as expected. In this particular test, there are two assertions. The first assertion (Line 21) checks if the number of coverage goals for the class under test is 3. The second assertion (Line 23) checks that we have achieved 100% coverage. Checking the number of coverage goals has proven quite useful over time, as a change in the number of coverage goals (for whatever reason) will usually have implications on the coverage that can be achieved. Debugging this case is much easier if we know explicitly that this has happened, rather than when trying to guess why the coverage percentage is not as expected.

Try to execute the test and see if it passes. Then, insert the following line before the call to `evosuite.parseCommandLine`:

```java
Properties.NULL_PROBABILITY = 1.0;
```

Re-run the test again – EvoSuite is now configured to only generate null objects (i.e., with a probability of 1.0), so it should only achieve 67% branch coverage (it covers the default constructor and the true branch in the target method ‘isNull‘).

Now let’s remove that line again from the test to make sure we don’t have a broken test! (Re-run the test after removing the line to make sure it passes again.)

3.4 Extending the search algorithm

Now let's make some changes to EvoSuite. As you might know, EvoSuite uses a Genetic Algorithm to drive the test generation. In a nutshell, this means that there is a population of candidate solutions (chromosomes, which are test suites in this case), and these test suites are evolved using search operators that are intended to simulate natural evolution. A fitness function estimates how good each candidate solution is. The fittest individuals have the highest likelihood of reproducing, and if they are selected for reproduction, then two parent individuals are combined to produce two new offspring individuals using a crossover operator, and then mutation makes smaller changes to these offspring.

All this is implemented in the `client` module, in the `org.evosuite.ga` package. For the abstract superclass `org.evosuite.ga.metaheuristics.GeneticAlgorithm` there are several concrete implementations, such as `StandardGA` (a default textbook genetic algorithm), a `SteadyStateGA`, or EvoSuite's default, the `MonotonicGA`. If you look at the `GeneticAlgorithm` class you will see that the search algorithm has plenty of members, such as a selection operator `selectionFunction`, the crossover operator `crossoverFunction`, and a population (`population`). The population is a list because individuals are ranked by their fitness value; this value is calculated by the `fitnessFunctions`. This, in turn, is a list because EvoSuite typically is used with several fitness functions at the same time, and there is a fitness value for every fitness function.

The `GeneticAlgorithm` class is configured with a `SinglePointCrossOver` by default. Let's have a closer look at how this class looks like – open up the class `org.evosuite.ga.operators.crossover.SinglePointCrossover` in an editor. The class extends the abstract class `CrossOverFunction`, and implements the method `crossOver`. The method receives two individuals as parents and chooses two crossover points `point1` and `point2` randomly, one for each of the two individuals. Then, it clones the parents, and on the resulting individuals it invokes the `crossover` method to do the actual work. This is the beauty of meta-heuristic search algorithms: The algorithm is independent of what the chromosomes represent.

Let's assume that we would like to implement an alternative crossover operator, which always cuts chromosomes in the middle, unlike the existing crossover operators which all choose random crossover points. Let's create a new Java class `org.evosuite.ga.operators.crossover.MiddleCrossOver` in the client module (in the directory `client/src/main/java/org/evosuite/ga/operators/crossover`). The class should extend the abstract class `CrossOverFunction`, which means it has to implement the method `crossOver`. The skeleton thus looks like this:

```java
package org.evosuite.ga.operators.crossover;

import org.evosuite.ga.Chromosome;
import org.evosuite.ga.ConstructionFailedException;

public class MiddleCrossOver extends CrossOverFunction {
    @Override
    public void crossOver(Chromosome parent1, Chromosome parent2)
        throws ConstructionFailedException {
         // TODO
    }
}
```

In order to implement this crossover function, we need to understand one important aspect: Textbook examples on genetic algorithms will usually assume a fixed number of genes in a chromosome. However, unlike many other standard applications of genetic algorithms, the size of individuals in EvoSuite can vary, as we cannot know the right number of test cases before we even start the search. Consequently, what is the "middle" is different for every individual.

Thus, the first thing we need to check is whether our individuals even have more than one test case. If they don't there's no way we can do any crossover:

```
    if (parent1.size() < 2 || parent2.size() < 2) {
        return;
    }
```

After this, we can assume that both parent chromosomes have at least 2 tests, and so we can calculate the middle of each of them:

```
    int middle1 = (int) Math.round(parent1.size() / 2.0);
    int middle2 = (int) Math.round(parent2.size() / 2.0);
```

The crossover operator in EvoSuite changes a chromosome in place. That means we first need to create the offspring as direct copies of the parents:

```
    Chromosome t1 = parent1.clone();
    Chromosome t2 = parent2.clone();
```

Now we can change the offspring using the `crossOver` method, which takes as parameters (1) the other chromosome with which to cross over, (2) the crossover point in the chromosome the method is invoked on, and (3) the crossover point in the other chromosome:

```
    parent1.crossOver(t2, middle1, middle2);
    parent2.crossOver(t1, middle2, middle1);
```

That's it! Let's write a test case `MiddleCrossOverTest.java` to find out if it works. Add the new file in the appropriate directory in the `client` module (`client/src/test/java/org/evosuite/ga/operators/crossover/`).

The tests in the client module have a `DummyChromosome` implementation that we use for the test. A `DummyChromosome` takes a list of integers, and does mutation and crossover. For example, we could create to parents with different sizes (e.g., 4 and 2), and then check if the resulting individuals have the right genes. For example, the test could look like this:

```
    @Test
    public void testSinglePointCrossOver() throws
        ConstructionFailedException {

       DummyChromosome parent1 = new DummyChromosome(1, 2, 3, 4);
       DummyChromosome parent2 = new DummyChromosome(5, 6);

       MiddleCrossOver xover = new MiddleCrossOver();

       DummyChromosome offspring1 = new DummyChromosome(parent1);
       DummyChromosome offspring2 = new DummyChromosome(parent2);

       xover.crossOver(offspring1, offspring2);

       assertEquals(Arrays.asList(1, 2, 6), offspring1.getGenes());
       assertEquals(Arrays.asList(5, 3, 4), offspring2.getGenes());
    }
```

If you did everything correctly, then this test should pass. Does it?

Now that we've got this wonderful new crossover operator, the next big question is: How do we make EvoSuite use it? EvoSuite is highly configurable, and the configuration is controlled by the class `org.evosuite.Properties` in the client module. In this class, you'll find all the different properties that EvoSuite supports – there are a lot of them. Each property consists of a public static field in all caps, which is how the properties are accessed from within code. In addition, each property has `@Parameter` annotation, in which we define a key – this is the key we use on the command line, if we set properties using the `-Dkey=value` syntax. If we look for crossover, we will find the following relevant code:

```
  public enum CrossoverFunction {
    SINGLEPOINTRELATIVE, SINGLEPOINTFIXED, SINGLEPOINT, COVERAGE
  }

  @Parameter(key = "crossover_function", group = "Search Algorithm",
      description = "Crossover function during search")
  public static CrossoverFunction CROSSOVER_FUNCTION =
      CrossoverFunction.SINGLEPOINTRELATIVE;
```

Thus, there is a property `Properties.CROSSOVER_FUNCTION`, and it is of type of the enum class `CrossoverFunction`, which contains all the possible crossover functions. In the future maybe EvoSuite will see some way to make it extensible at runtime, but for now we need to add our new crossover operator to the enum:

```
  public enum CrossoverFunction {
    SINGLEPOINTRELATIVE, SINGLEPOINTFIXED, SINGLEPOINT,
    COVERAGE, MIDDLE
```

```
}
```

The final thing we need to change is the place where this property is read and the crossover function is instantiated. If we look up where in the source code the property `Properties.CROSSOVER_FUNCTION` field is used, we see that it is used in `org.evosuite.strategy.PropertiesSuiteGAFactory` and `PropertiesTestGAFactory`. These are two factory classes that create and configure a genetic algorithm object based on the values in the Properties class. As we are doing whole test suite generation (it's EvoSuite's default), let's edit `PropertiesSuiteGAFactory`. Find the method `getCrossoverFunction()`. It contains a switch over the value of our property, and calls the corresponding constructor. Thus, we need to add a new case:

```
case MIDDLE:
    return new MiddleCrossOver();
```

That's it! Now we're ready to generate a jar file and use EvoSuite with our new crossover function. Recall that you can generate the jar file (which will be located in `master/target`) using:

```
mvn package
```

When we now run EvoSuite with this jar file, we can specify to use our new crossover function using `-Dcrossover_function=Middle`. Likely this operator will not make a difference – it's just an example for illustration purposes. However, in the next section we will look at how to run experiments with EvoSuite in general, and you could investigate this crossover operator with some similar experiments.

4 Running Experiments with EvoSuite

4.1 Preparing the experiment

For the third part of the tutorial, we will be looking at how one can collect data about the test generation. We will use a simple example scenario: EvoSuite by default uses a combination of different coverage criteria [10]. What are the effects of this combination over using just branch coverage as target criterion? A reasonable hypothesis would be that the combination leads to more tests, and better test suites. But is that actually true? Let's run an experiment to find out!

The experiment will involve running EvoSuite on a number of classes with its default configuration and configured to only use branch coverage, and then to take different measurements of the resulting test suites. When doing experiments of this kind, the selection of classes has implications on how much our results generalize: If we use a very specific and small selection of classes, then whatever our findings, they may only be relevant to that particular type of classes. Therefore, we generally would want to select as many as possible, as diverse as possible, and as representative as possible classes in order to get results that generalize. However, this is not the aim of this tutorial, so let's just use a selection of classes we've prepared for this tutorial. The tutorial assumes that you download and extract the archive containing the selection of example classes (but note you can, in principle, use any collection of Java classes instead):

```
wget http://evosuite.org/files/tutorial/Tutorial_Experiments.zip
unzip Tutorial_Experiments.zip
```

Change into the main directory again, and compile the example project with Maven:

```
cd Tutorial_Experiments
mvn compile
```

We will be invoking EvoSuite directly in this part of the tutorial. To avoid having to set the classpath repeatedly, let's set up EvoSuite. First, we need to download all dependency jar files of the example project. To make things slightly more challenging, the class `tutorial.Bank` has a (quite artificial) dependency on the Apache Commons Collections library. When running EvoSuite from Maven, then Maven downloads all dependencies and sets up the classpath for us automatically – but when we run EvoSuite directly it is our responsibility to set up a correct classpath. Fortunately, this is easy enough: To download all dependencies, type the following Maven command:

```
mvn dependency:copy-dependencies -DincludeScope=runtime
```

This command downloads all dependency jar files, and puts them into the `target/dependency` directory. The reason for specifying the scope to be runtime using `-DincludeScope=runtime` is that the project has test dependencies on JUnit and EvoSuite – but neither of these dependencies are necessary in order to generate some tests for the classe under test, we really just need the compile and runtime dependencies. Thus, the full project classpath consists of the classes in `target/classes` and the jar file `target/dependency/commons-collections-3.2.2.jar`. We can store this information by creating an `evosuite.properties` file that saves this classpath, by use the following command:

```
$EVOSUITE -setup target/classes
    target/dependency/commons-collections-3.2.2.jar
```

Check that the resulting evosuite-files/evosuite.properties at the top has the correct classpath set:

```
CP=target/classes:target/dependency/commons-collections-3.2.2.jar
```

4.2 Collecting data with EvoSuite

Let's start by invoking EvoSuite on the `Stack` class in our project, targeting only branch coverage:

```
$EVOSUITE -class tutorial.Person -criterion branch
```

We have already had a closer look at the test suites that EvoSuite produces. However, EvoSuite also produces data to document what happened. This is stored in the following file:

```
evosuite-report/statistics.csv
```

Use your favourite editor to have a closer look at this file. You should see something like this:

```
TARGET_CLASS,criterion,Coverage,Total_Goals,Covered_Goals
tutorial.Person,BRANCH,1.0,3,3
```

This file is in comma-separated value format. The first row contains headers showing what the individual columns contain, and then the rows contain the actual data. The first column contains the name of the class we tested (`tutorial.Person`). The second column shows us the coverage criteria that we used – in this case we see the full list of criteria that EvoSuite uses by default, separated by semincolons. The third column tells us the achieved coverage – 1.0 in this case, which means we have 100% coverage (yay!). This is calculated based on the ratio of coverage goals covered to total goals (last two columns).

Let's test the same class again, but this time using line and branch coverage:

```
$EVOSUITE -class tutorial.Person -criterion line:branch
```

If we look at `evosuite-report/statistics.csv` again we'll see a new row:

```
TARGET_CLASS,criterion,Coverage,Total_Goals,Covered_Goals
tutorial.Person,BRANCH,1.0,3,3
tutorial.Person,LINE;BRANCH,1.0,9,9
```

As you can see, we now have a new entry for our second call to EvoSuite, where we specified branch and line coverage as target criteria.

Let's try another class and criterion:

```
$EVOSUITE -class tutorial.Company -criterion line
```

Again, the `evosuite-report/statistics.csv` file will now contain a new line:

```
TARGET_CLASS,criterion,Coverage,Total_Goals,Covered_Goals
tutorial.Person,BRANCH,1.0,3,3
tutorial.Person,LINE;BRANCH,1.0,9,9
tutorial.Company,LINE,1.0,4,4
```

The `tutorial.Company` class has four lines of code, and the generated tests cover all of them. Great!

4.3 Setting output variables

We now know where to find data about the test generation. However, the data we have seen does not help us to answer the questions we would like to investigate. Recall that our scenario was that we wanted to know if the default combination of criteria leads to more tests, and better test suites. We cannot answer this with the data in the statistics.csv files currently — the coverage values cannot be compared (they refer to different criteria), and neither can the numbers of goals.

Fortunately, we can generate more data than just the columns our data file has shown us so far. EvoSuite has a property `output_variables` which determines which values should be written to the `statistics.csv` file. Before we do that, let's remove the old statistics.csv file:

```
rm evosuite-report/statistics.csv
```

This is important if we decide to change the columns of the data files – our data file currently has a header row and three data rows that assume there are five columns; if we change the columns, and additional rows will not match the existing data.

Now, let's include some new values. There are two main types of output variables: *runtime variables*, which are the result of computation (e.g., the coverage), whereas *properties* are the input properties we can set. For example, `TARGET_CLASS` and `criterion` are properties, whereas `Total_Goals` and `Covered_Goals` are runtime variables. There are some inconsistencies in terms of which variables are capitalised – this is for historic reasons, as changing the runtime variable names may break existing experimental infrastructure. However, in a future major release we may decide to change the variable names to a consistent format.

Let's think about what values we would like to include. Our first question is whether the combination of criteria leads to more tests. The corresponding output variable is `Size`, which reports the number of tests. However, let's not forget that these are unit tests, where a single test can consist of several statements. Thus, we can also use the `Length` variable to count the total number of statements, which is maybe a better representation of the size of a test suite.

Our second question is whether the combination of criteria leads to better tests. A standard way to evaluate test suites is by measuring coverage – but which criterion would we use to measure this? A better way might be to compare the test suites in terms of their mutation scores. The mutation score is a metric based on the idea of Mutation Analysis, and quantifies how many artificial faults a test suite can find. There are several mutation analysis frameworks for Java available, but EvoSuite also has a basic mutation functionality integrated [8], as it can aim to generate tests that kill mutants directly. The output variable for this is `MutationScore`.

To summarize, for our experiment we would like to have the following data:

- Class under test (`TARGET_CLASS`)
- Criteria (`criterion`)
- Size (`Size`)
- Length (`Length`)
- Mutation score (`MutationScore`)

The list of variables is passed as a comma separated list to the `output_variables` property. Let's try this out:

```
$EVOSUITE -class tutorial.Company -criterion branch
    -Doutput_variables=TARGET_CLASS,criterion,Size,Length,MutationScore
```

If you look at the resulting `evosuite-report/statistics.csv` file, you should see something like this:

```
TARGET_CLASS,criterion,Size,Length,MutationScore
tutorial.Company,BRANCH,1,2,1.0
```

Thus, we have just generated one test consisting of two statements, and this test killed all the mutants EvoSuite generated for the class.

If we look at the test suite in `evosuite-tests/tutorial/Company_ESTest.java` you should see something like this:

```
    @Test(timeout = 4000)
    public void test0() throws Throwable {
        Company company0 = new Company("");
        String string0 = company0.getName();
        assertEquals("", string0);
    }
```

Note that the assertion is not included in EvoSuite's statement count. This is because assertions are not generated as part of the search-based test generation, but are added in a post-processing step.

4.4 Running an experiment

Now let's run an actual experiment and gather some data. We would like to get information on all classes in our project, so we need to run EvoSuite on all of them. Furthermore, let's not forget that EvoSuite is randomized: If you run it twice in sequence, you will get different results. That also means that if you get a very large test suite in one run, you may get a test suite with a different size in the next run. In general, when we have randomized algorithms, we need to run repetitions, and statistically analyze our data. Therefore, we'll generate tests on all our classes, and repeat this 10 times. Furthermore, we need to do all this twice, once with only branch coverage, and once with the default criteria. Before we start the experiment, let's remove the old statistics.csv file again:

```
rm evosuite-report/statistics.csv
```

Now, let's run the experiment. We will tell EvoSuite to test all classes in the `tutorial` package using the `-prefix` argument, and pass in the target criterion (branch) as well as our output variables.

```
$EVOSUITE -criterion branch -prefix tutorial -Dshow_progress=false \
   -Doutput_variables==TARGET_CLASS,criterion,Size,Length,MutationScore
```

We added `-Dshow_progress=false`; this isn't essential, but the progress bar does tend to clutter up log files if we perform larger numbers of runs, so we deactivated it here. If you look at the data file, you should see something like this:

```
TARGET_CLASS,criterion,Size,Length,MutationScore
```

```
tutorial.ATM,BRANCH,10,75,0.3888888888888889
tutorial.ATMCard,BRANCH,8,40,1.0
tutorial.Bank,BRANCH,4,15,0.8
tutorial.BankAccount,BRANCH,2,6,0.8
tutorial.Owner,BRANCH,1,1,1.0
tutorial.CurrentAccount,BRANCH,2,7,0.6521739130434783
tutorial.SavingsAccount,BRANCH,2,7,0.8529411764705882
tutorial.Company,BRANCH,1,2,1.0
tutorial.Person,BRANCH,2,4,0.0
```

We now have data for all classes, for the first configuration we are interested in (branch coverage). If we re-run this command without the `-criterion branch` argument, we'll get some more data for all classes for the other configuration (default coverage criteria). When analysing this data, we need to distinguish between the two configurations; we can either use the `criterion` column we have already added, or we can also label our configurations, using the `-Dconfiguration_id=name` syntax, and then including this property in the output variables. Thus, to run our experiment, we will need the following two commands, one for branch coverage, one for the default combination:

```
$EVOSUITE -Dconfiguration_id=Default \
    -prefix tutorial -Doutput_variables=configuration_id,\
    TARGET_CLASS,criterion,Size,Length,MutationScore
$EVOSUITE -Dconfiguration_id=Branch -criterion branch \
    -prefix tutorial -Doutput_variables=configuration_id,\
    TARGET_CLASS,criterion,Size,Length,MutationScore
```

This will result in something like the following in `statistics.csv`:

```
configuration_id,TARGET_CLASS,criterion,Size,Length,MutationScore
Default,tutorial.ATM,[...],14,109,0.3611111111111111
Default,tutorial.ATMCard,[...],13,65,1.0
Default,tutorial.Bank,[...],6,22,0.8
Default,tutorial.BankAccount,[...],8,24,1.0
Default,tutorial.Owner,[...],1,1,1.0
Default,tutorial.CurrentAccount,[...],4,12,0.7608695652173914
Default,tutorial.SavingsAccount,[...],4,12,0.8823529411764706
Default,tutorial.Company,[...],3,6,1.0
Default,tutorial.Person,[...],6,12,1.0
Branch,tutorial.ATM,BRANCH,10,77,0.4166666666666667
Branch,tutorial.ATMCard,BRANCH,8,40,1.0
Branch,tutorial.Bank,BRANCH,4,15,0.8
Branch,tutorial.BankAccount,BRANCH,2,6,0.8
Branch,tutorial.Owner,BRANCH,1,1,1.0
Branch,tutorial.CurrentAccount,BRANCH,2,7,0.6739130434782609
Branch,tutorial.SavingsAccount,BRANCH,3,8,0.6470588235294118
Branch,tutorial.Company,BRANCH,1,2,1.0
Branch,tutorial.Person,BRANCH,2,4,0.0
```

(In this example, we replaced the list of criteria (`LINE;BRANCH;`...) with `[...]` to make it fit into this article.)

Just by eyeballing the results, we can see that the default configuration leads to more tests in all classes except `tutorial.Owner`. Your specific data will look different – in the data down above, the mutation score is higher for `tutorial.Person`, `tutorial.SavingsAccount`, `tutorial.CurrentAccount`, but surprisingly, lower for `tutorial.ATM`. How can that be the case? Recall that EvoSuite is randomized — sometimes test generation will be lucky to hit a specific value that is good at killing some mutants, sometimes it isn't. What we need to establish, then, is not whether one configuration is better than the other in one particular run, but on average. Thus, we need to repeat our experiment several times, and do some more rigorous analysis.

A simple way to do the repetitions would be to simply wrap the call in a bash-loop to run it, for example, 5 times:

```
for I in {1..5}; do $EVOSUITE -Dconfiguration_id=Default [...] ; done
for I in {1..5}; do $EVOSUITE -Dconfiguration_id=Branch [...] ; done
```

This is going to take quite a while. In fact, 5 repetitions is not even a suitably large number for serious experiments, ideally you'd want 30 repetitions or more to get representative results.

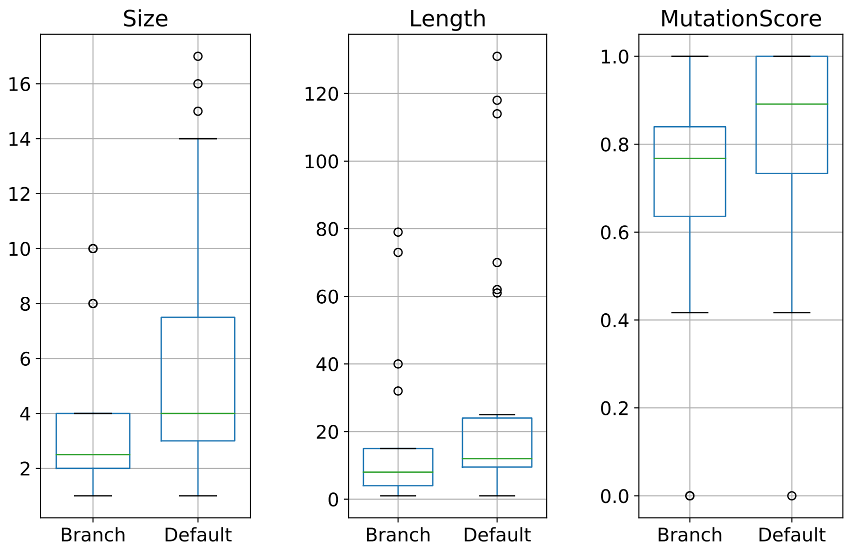
4.5 Analyzing results

Now we have some data – from at least one run, and if you were patient enough, maybe from 5 or more additional runs. What are we going to do with that data? The best thing to do now is to use statistical analysis package to process and analyze the data. For example, using Python's Matplotlib[5] we can produce the boxplots shown in Figure 5. Besides visualizing the data, you will also need to statistically analyze it [1]. If we consider the data of our experiment, you will find that, with statistical significance, we can say that test suites generated for branch coverage have different sizes, numbers of statements, and mutation scores than those generated for the default criteria. The effect size tells us that for all three of these properties there is a medium increase when using the default configuration over the branch configuration. So all in all, it sounds like a good idea to use the default configuration! (After all, that is why it is the default configuration...)

4.6 Other useful variables

To get a full overview of the available output variables, the best place is currently the source code, in particular the file `RuntimeVariable.java` in the `client` module (package `org.evosuite.statistics`). For example, if you want to know how certain values evolved over time, there are timeline variables that capture this data for you. Assume we would like to see how branch coverage evolves over the first 30 seconds of the search, and we want to sample once every second.

[5] https://matplotlib.org/



(a) Number of Tests (b) Number of Statements (c) Mutation Score

Fig. 5. Analysis of the results on the branch coverage vs. default criteria comparison.

To do this, we would add an output variable `CoverageTimeline`, and specify the sampling interval using `-Dtimeline_interval=1000`:

```
$EVOSUITE -class tutorial.ATM -criterion branch \
-Doutput_variables=TARGET_CLASS,BranchCoverage,CoverageTimeline \
-Dtimeline_interval=1000 -Dsearch_budget=30
```

As we specified a time budget of 30 seconds in total (`-Dsearch_budget=30`), the `statistics.csv` file will now have 30 columns labeled `CoverageTimeline_T1` up to `CoverageTimeline_T30`, with the individual values for each second of the search.

As another interesting example, the `BranchCoverageBitString` variable will produce a string of `0` and `1` digits, where each digit represents one branch in the program, and 1 indicates that the branch was covered. This bitstring allows us to compare whether specific branches were covered by specific configurations.

5 Conclusions

In this tutorial, we covered basic usage of EvoSuite on the command-line, some simple changes to EvoSuite's source code, and some basic experiments. If you want to learn more about EvoSuite, here are some pointers:

– `http://www.evosuite.org`: The main EvoSuite website contains many papers related to EvoSuite, experimental data to reproduce past experiments, and documentation. The documentation includes a more elaborate version

of this tutorial, and instructions on how to use the different plugins (e.g., Maven).
– `https://github.com/EvoSuite/evosuite`: EvoSuite is open source, licensed with the GNU Lesser General Public License version 3. The source code repository is on GitHub, as is an issue tracker. Since EvoSuite is an open source project, its continued maintenance depends on contributions. If you produce work or improvements to EvoSuite, please do consider to feed them back to the project!

Acknowledgements

This work is supported by EPSRC project EP/N023978/2.

References

1. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability (STVR) **24**(3) (2012)
2. Arcuri, A., Campos, J., Fraser, G.: Unit test generation during software development: EvoSuite plugins for Maven, IntelliJ and Jenkins. In: IEEE International Conference on Software Testing, Verification, and Validation (ICST) (2016)
3. Arcuri, A., Fraser, G.: Parameter tuning or default values? An empirical investigation in search-based software engineering. Empirical Software Engineering (EMSE) **18**(3), 594–623 (2013)
4. Arcuri, A., Fraser, G., Galeotti, J.P.: Automated unit test generation for classes with environment dependencies. In: ACM/IEEE International Conference on Automated Software Engineering (ASE). pp. 79–90. ACM (2014)
5. Arcuri, A., Fraser, G., Galeotti, J.P.: Generating TCP/UDP Network Data for Automated Unit Test Generation. In: ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). pp. 155–165 (2015)
6. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering (TSE) **39**(2) (2013)
7. Fraser, G., Arcuri, A.: A large-scale evaluation of automated unit test generation using EvoSuite. ACM Transactions on Software Engineering and Methodology (TOSEM) **24**(2) (2014)
8. Fraser, G., Arcuri, A.: Achieving scalable mutation-based generation of whole test suites. Empirical Software Engineering **20**(3), 783–812 (2015)
9. Panichella, A., Kifetew, F.M., Tonella, P.: Reformulating branch coverage as a many-objective optimization problem. In: IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 1–10. IEEE (2015)
10. Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A.: Combining multiple coverage criteria in search-based unit test generation. In: International Symposium on Search Based Software Engineering. pp. 93–108. Springer (2015)
11. Rojas, J.M., Fraser, G., Arcuri, A.: Seeding strategies in search-based unit test generation. Software Testing, Verification and Reliability **26**(5), 366–401 (2016)