



Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Automated Test Generation in Continuous Integration Systems

João Henrique Gonçalves Veras

Trabalho de Graduação

Recife
04 de julho de 2018

Universidade Federal de Pernambuco
Centro de Informática

João Henrique Gonçalves Veras

Automated Test Generation in Continuous Integration Systems

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Juliano Manabu Iyoda*

Recife
04 de julho de 2018

Acknowledgements

First of all, I would like to thank my family for having supported me the whole time and for being there in every hard moments during the graduation process.

An special thankful for my lifemate, Raissa, who have motivated me in hard times and also was always there for me.

I am very grateful to my advisor Prof. Juliano Iyoda, to whom I had the opportunity to work with not on academically but also in a software development project that I could learn a lot and have been patient for helping me to write down the monograph.

I am also grateful for every professor from the Center of Informatics that have taught me thing that I will take to my whole life, not only professionally mas also personally.

I would like to thank people who I have being working for years from the Motorola facility at the the Center of Informatics. Also thank to every friend who worked with me during one and a half year in the junior enterprise CITi. All of them were very meaningful for me and for my professional life.

Finally, I acknowledge the remarkable people I worked and studied with. I couldn't graduate alone.

*The first step is to establish that something is possible; then probability will
occur.*

—ELON MUSK

Resumo

Diversos desafios surgem durante o desenvolvimento de um software e um deles é certamente garantir a confiabilidade para se ter um produto final com maior segurança e menor número de falhas possível. O teste é a forma utilizada de identificar as falhas e fragilidades de um produto, e com software não é diferente. Testes automatizados podem ser criados e assim gerar uma certa cobertura de testes para módulos específicos de um software, garantindo assim, que a cada mudança esse módulo funciona sempre como esperado. Entretanto, em um cenário realista, os testes automatizados geralmente não são postos em prática por "desperdiçar" o tempo dos desenvolvedores e atrasar as entregas. Isso motivou a criação de ferramentas que automatizam a geração dos testes. Este trabalho pretende estudar o uso dessas ferramentas trabalhando em conjunto com ferramentas de integração contínua e propõe uma nova alternativa que possa ser o mais automático possível. Implementamos um plugin do sistema de integração contínua Jenkins com Randoop, uma ferramenta de geração de testes automatizados. Fizemos uma avaliação empírica comparando o uso de testes manuais, Jenkins integrado ao EvoSuite (desenvolvido por Acuri *et al.* [9]) e Jenkins integrado ao Randoop (desenvolvido neste trabalho). Executamos as ferramentas em três sujeitos: Spark, JSoup e Joda-time. Os resultados mostram que a integração contínua com geração automática de testes é benéfica em ambos os casos (EvoSuite e Randoop), pois milhares de testes são criados e executados de forma automática. Randoop se mostrou levemente mais consistente, pois, para o Spark, ele encontrou bugs enquanto EvoSuite, não. Para o JSoup e o Joda-time, ambos encontraram bugs.

Palavras-chave: integração contínua, teste contínuo, testes automatizados, engenharia de software

Abstract

Several challenges arise during software development and one of them is certainly to guarantee its reliability in order to have a final product with greater security and a few failures. Testing is the most popular technique to identify the weaknesses of a product and with software is no different. Automated tests can be created and thus generate a certain coverage of tests for specific modules of a software, and this way ensuring that, at each change, this module will always work as expected. However, in a realistic scenario, some developers teams do not make it part of the development process for regarding it a waste of time, and this has led to the creation of tools that automate the generation of test cases. This work intends to study the use of such tools working together with Continuous Integration tools and propose a new alternative that may be as automatic as possible. We implemented a plugin for the Jenkins continuous integration tool with Randoop, a tool for automated test case generation. We carried out an empirical evaluation comparing the use of manual test generation, Jenkins integrated to EvoSuite (developed by Acuri *et al.* [9]) and Jenkins integrated to Randoop (developed in this work). We applied these treatments to three subjects: Spark, JSoup, and Joda-time. The results show that the continuous integration with automated test case generation is beneficial in both cases (EvoSuite and Randoop) as thousands of test cases are generated automatically. Randoop has been slightly more consistent as, for Spark, it found bugs while EvoSuite did not. For JSoup and Joda-time both EvoSuite and Randoop found bugs.

Keywords: Continuous integration, continuous testing, automated test, software testing, software engineering.

Contents

1	Introduction	1
2	Background	3
2.1	Continuous Integration	3
2.1.1	Software build	3
2.1.2	Build automation	4
2.2	Apache Maven	4
2.2.1	Basic structure	5
2.2.2	Adding dependencies	5
2.3	Jenkins	5
2.3.1	Plugins	6
2.4	Automated Test Input Generation	8
2.4.1	EvoSuite	8
2.4.2	Randoop	9
2.5	Continuous test generation	10
3	Continuous Test Generation for Randoop	12
3.1	Proposed tool	12
3.2	Extending Jenkins	12
3.2.1	Project configuration	12
3.2.2	Test generation input	14
3.2.3	Test generation output	15
3.3	Installing and configuring the plugin	15
4	Experiments and Results	18
4.1	Subjects selection	18
4.2	Experiments procedures	18
4.3	Experiments results	19
4.3.1	Spark	20
4.3.2	JSoup	21
4.3.3	Joda-time	22
4.4	Concluding Remarks	22
5	Related Work	23
5.1	Concluding Remarks	24

6 Conclusion**25**

List of Figures

2.1	Continuous integration process	4
2.2	Pom file example	5
2.3	JUnit dependency added in the pom.xml file	6
2.4	Jenkins welcome page	6
2.5	Jenkins plugins available page	7
2.6	Jenkins project archetype generated	7
2.7	Java Person class	9
2.8	Unit test for class Person generated by EvoSuite	10
3.1	CTG diagram for Randoop	13
3.2	Application console during Randoop test generation	15
3.3	Jenkins upload plugin section	16
3.4	Randoop post-action configuration	17
4.1	Spark project jobs for EvoSuite experiments	19
4.2	Experiments procedure diagram	20

List of Tables

4.1	Experiments subjects information.	18
4.2	The results for spark.	21
4.3	The results for jsoup.	22
4.4	The results for joda-time.	22

Introduction

Software development teams have, in their daily basis activities, many challenges related to integration, testing, and building of their code to make it as reliable as possible. However, some steps are not properly executed and sometimes are ignored. Although testing is essential to ensure whether the software is reliable so that the development team can find and fix errors more straightforwardly and quickly, some developers teams do not make it part of the development process due to waste of time in creating tests [22].

The difficulty to create automated tests has motivated some research related to Automated Test Generation (ATG) [19]. These techniques generate input values for the developed code aiming to reach some coverage goal or to reach a failure. They typically use methods like Symbolic Execution, Search-based Testing, and Random Testing.

In addition to that, in order to speed up the development, software engineers have been using Continuous Integration (CI) systems to integrate changes in the main codebase. According to Martin Fowler, Continuous Integration is a software development practice where teams can integrate their code frequently [13]. Every time a person pushes a code to a repository, changes are validated automatically by an automated building process. In this building process, the code can be verified and tested to detect errors and integration issues. This allows people to develop more reliable software.

In order to better address the software testing in daily activities, a Continuous Integration system (e.g., Jenkins [4]) can be extended with Automated Test Input Generation and introduce the concept of Continuous Test Generation (CTG) [10]. The goal of CTG is to give immediate feedback while the build pipeline is running in the CI system. This means that, for every change in the codebase, test cases are produced and run, and the developers are able to identify errors rapidly.

Although there are several automatic test generation tools, just a few are being used in combination with CI systems. For example, for the programming language Java, there is a way to integrate a CI system to a test case generator: Jenkins (an open source CI system) [4] can be integrated to EvoSuite [14], and Parasoft Jtest [7]. EvoSuite is a tool that uses the search-based approach to automate the production of test suites that achieve high code coverage. Parasoft Jtest is a tool that automates static analysis and unit testing. To the best of our knowledge there are no other integration as such. In this work, we implemented a new integration between Jenkins and Randoop [8], a tool for random generation of automated test unit for Java. We carried out experiments comparing test cases generated manually, the Jenkins-EvoSuite integration, and our Jenkins-Randoop integration. We applied these treatments to three subjects: Spark, JSoup, and Joda-time. Our results show that Continuous Testing Generation with both EvoSuite and Randoop is beneficial: thousands of test cases are generated automatically. Randoop

was slightly more consistent than EvoSuite: it found bugs in Spark while EvoSuite did not. For JSoup and Joda-time, both EvoSuite and Randoop found bugs.

This monograph is organized as follows. Initially, we introduce basic concepts to better understand our work (Chapter 2). We propose a Jenkins plugin that integrates with Randoop and that allows users to choose a tool that better suites their needs to generate automated tests (Chapter 3). Then we describe how we organized our experiments for these chosen projects (Chapter 4). Chapter 5 presents a brief review on related works, and Chapter 6 concludes.

CHAPTER 2

Background

In this section we will cover the subjects necessary for a better understanding of the studied area.

2.1 Continuous Integration

According to Martin Fowler, Continuous Integration (CI) is a software development practice where teams can integrate their code frequently [13]. The process of integrating code can be a real nightmare for software development teams if not done in the correct way. Although it may not be necessary for small teams working on a system with few dependencies, when working in a large team and on a system with many dependencies, there is a great need to ensure that every component works properly together [11].

Moreover, Continuous Integration, in its simplest form, can be put into practice as a tool that watches the source code. Whenever a change happens in the source code, usually a version control system such as git [3], the CI system retrieves the code, generates a build and tests it. Provided that no issues were found, compiled code can be pushed to production. If anything goes wrong the developers will be notified so they can fix it immediately [24].

As shown in Figure 2.1, the CI process starts with the developer pushing the code to the version control system repository. First, the CI server identifies the changes and fetches the code from the version control repository. Second it needs to know what steps to take to integrate the software, run the tests, and do other tasks. The input for this step is usually a build script that contains instructions to generate the build. Finally the CI system sends feedback to specific project members informing if the build passed or failed and if it is ready for deploy or if it was already deployed.

Not only Continuous Integration systems can integrate and test code but also can help developers to keep track of important information about the source code such as code coverage, health check, and code metrics. In addition to that, CI systems can simplify the deployment by automating it or providing an interface (typically, a button) that does the deployment process in one-click.

2.1.1 Software build

A software build is much more than just the compiled code. A build is the final version of a software. It is the compiled, tested, inspected, among other things, version of a software [11]. If a build is generated, it means that the software works as expected with all components

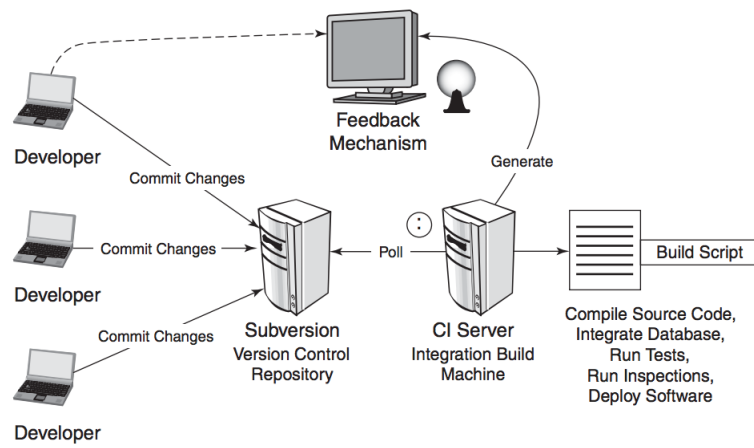


Figure 2.1 1. The developers push the code to the System Control Version 2. The Continuous Integration server identifies that there are changes in the source code and fetches the code. The automation build process starts and the build is tested. Image credit: Duvall and Glover [11]

integrated.

2.1.2 Build automation

Every step needed to generate a software build can be done one by one manually. Although it seems simple in some cases, it can get really confusing in a project with many dependencies and integration. The more steps the build needs to be completed the more mistakes the developers are prone to commit. Another key point to remember is that a build process may take a long time to finish and waiting for every step to finish can be a waste of time.

The most recommended solution for generating a software build is to automate the whole build process. By automating the build process, the developers can focus more on building a more reliable software and worry less about how to install the software in a computer. To give an illustration about the importance of automated builds, imagine that only a small change was made in the code and then the development team has to run the whole build process, step by step, again.

Basically, by automating a build we are able to run all steps in a single command. There are already plenty of tools that provide functionality to help to automate builds across many platforms and for diverse programming languages. The Java community, for instance, has Ant, Gradle and Maven as the most used according to the survey by David Kiss [2]. For this monograph, we focus on the Apache Maven [17].

2.2 Apache Maven

Apache Maven [17] is a tool for building and managing any Java-based project. Apache Maven has as its main objective to make the build process easy, provide a uniform build system, pro-

vide quality project information, provide guidelines for best practices in development, and allow transparent migration to new features [17].

2.2.1 Basic structure

Every Maven's project contains the Project Object Model represented by the file called *pom.xml*. This file contains information related to the project including project configuration, project version, dependencies, description, and so on. The code snippet in Figure 2.2 is an example of the most basic *pom.xml* file for a project called *my-project*.

There are three required fields in a pom file **groupId**, **artifactId** and **version**. The field **groupId** represents a unique identifier of a directory structure containing the project. Although the groupId is important and unique, it will never be used to identify a maven project, the field **artifactId** is, in most cases, the name that represents the project. The **version** field is generally used for software version control.

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4       http://maven.apache.org/xsd/maven-4.0.0.xsd">
5   <modelVersion>1.0.0</modelVersion>
6
7   <groupId>br.ufpe.cin.my.project</groupId>
8   <artifactId>my-project</artifactId>
9   <version>1.0</version>
10 </project>
```

Figure 2.2 Basic pom.xml file. Credits: the author

2.2.2 Adding dependencies

The project dependencies are listed in the *pom.xml* file as well as the project information. Maven manages the dependencies list in order to keep it as simple as possible and easy to identify the other projects integrated with the one being developed. Every dependency goes inside the `dependencies` tag. In Figure 2.3 the JUnit project is listed as a dependency.

The command `mvn install` downloads all dependencies for the Maven project and make them available for the project.

2.3 Jenkins

Jenkins [24] is an open source Continuous Integration tool written in Java. Jenkins has become very popular among development teams for the ease of use, appealing interface and very low learning curve. Another key point that makes Jenkins to be chosen by many development teams is that it can be adapted to any team's purpose through the many available plugins.

Although Jenkins needs to run in a dedicated host, it is not a difficult task to set up the

```

11      ...
12      <dependencies>
13        <dependency>
14          <groupId>junit</groupId>
15          <artifactId>junit</artifactId>
16          <version>4.0</version>
17          <type>jar</type>
18          <scope>test</scope>
19          <optional>true</optional>
20        </dependency>
21      ...
22    </dependencies>
23    ...

```

Figure 2.3 JUnit added as a dependency in the pom.xml file. Credits: the author

environment and get started. In this monograph we have used Jenkins running in a Docker [18] container that makes the installation process even simpler and requiring fewer steps and with no need to install dependencies. The Jenkins server up and running is shown in Figure 2.4.

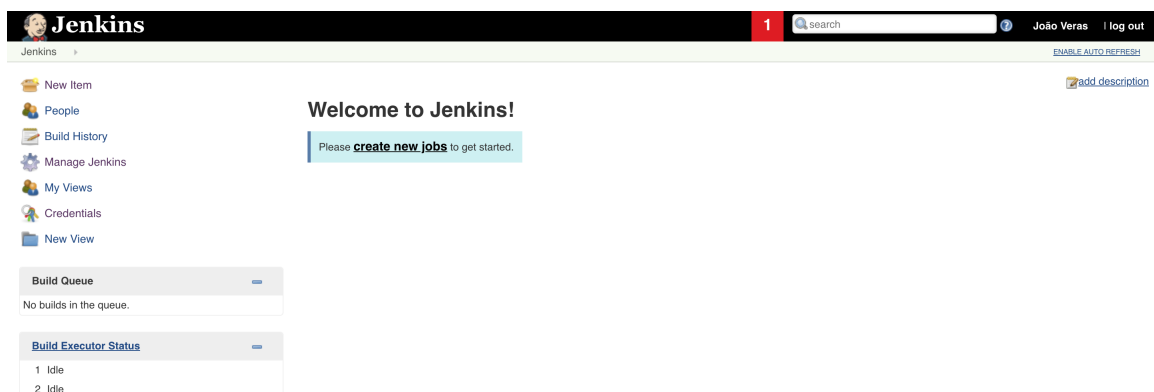


Figure 2.4 Jenkins welcome page.

2.3.1 Plugins

To give a better idea of Jenkins' versatility, it has more than a thousand plugins, at the time of writing this monograph [5]. In addition to that, a new plugin can be created and uploaded directly to Jenkins so it can better suite the user needs. Moreover, Jenkins provides a documentation to support developers to create custom plugins.

Plugins can be used for performance benchmarking [25], regression tests [12], and reporting, among other features. As can be seen in Figure 2.5, the available plugins can be found in the *Manage Plugins* section.

The base programming language to extend Jenkins via plugins is Java. Since it is over-

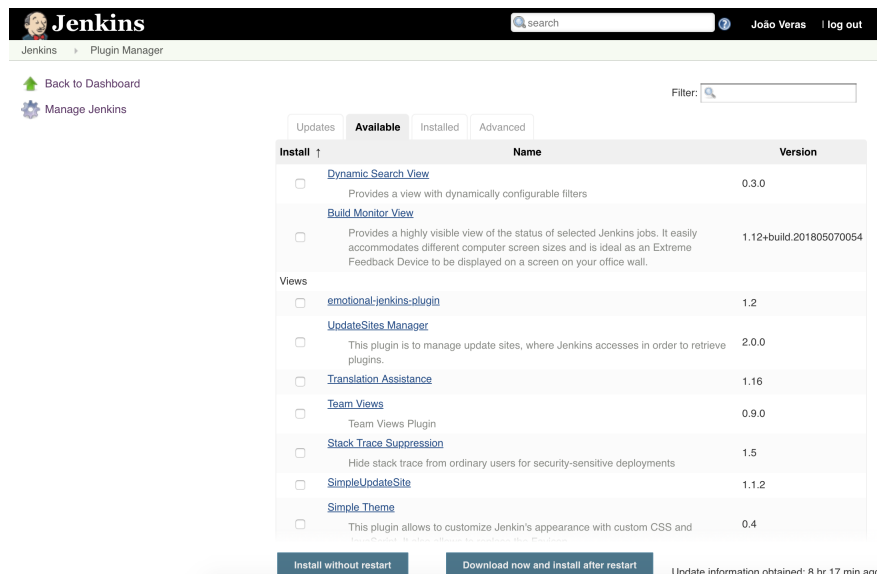


Figure 2.5 Jenkins plugins available page.

whelmingly used by other plugins, Maven is recommended to build the code. The plugin's development can get even easier by running a command that let developers to design an archetype related to Jenkins. For instance, the Hello World Project archetype below can be generated through the command:

```
mvn -U archetype:generate -Dfilter=io.jenkins.archetypes:
```

After choosing the proper options, the archetype will be generated with an output similar to what can be seen in Figure 2.6.

```
[INFO] -----
[INFO] Using following parameters for creating project from Archetype: hello-world-plugin:1.4
[INFO] -----
[INFO] Parameter: groupId, Value: unused
[INFO] Parameter: artifactId, Value: tcc
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: io.jenkins.plugins.sample
[INFO] Parameter: packageInPathFormat, Value: io/jenkins/plugins/sample
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: io.jenkins.plugins.sample
[INFO] Parameter: groupId, Value: unused
[INFO] Parameter: artifactId, Value: tcc
[INFO] Project created from Archetype in dir: /Users/jveras/tcc
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 21.706 s
[INFO] Finished at: 2018-05-26T19:44:05-03:00
[INFO] Final Memory: 15M/164M
[INFO] -----
```

Figure 2.6 [Jenkins project archetype generated.

2.4 Automated Test Input Generation

Alessandro Orso and Gregg Rothermel in their research travelogue paper [19] have investigated what are the most significant contributions to testing since 2000 and what are the largest open challenges, in this area, for the future. Among the contributions the researchers have made, the automated test input generation was the most mentioned one.

Automated test input generation is by no mean a new topic in the testing area. Notwithstanding with the limitations of the technology before 2000, all research were important and had its relevance. However, it was in the last decade that the area gained relevance and the most important results and contributions have appeared [19]. Furthermore, supporting technologies and advances were made in the areas such as symbolic execution, search-based testing, random and fuzz testing, and combinations of these techniques.

2.4.1 EvoSuite

EVOSUITE [14] is an automated test input generation tool that achieves high code coverage and provides assertions. Moreover, EVOSUITE combines not only techniques in the state-of-art, for instance hybrid search, dynamic symbolic execution and testability transformation, but also novel techniques such as whole test suite generation and mutation-based assertion generation.

In order to better address a development team's needs, EVOSUITE comes with few alternatives to generate tests. The tests can be generated through command line, IDE plugins and Maven plugin. There is also an integration, which is not finished, with Jenkins [9]. Although the integration with Jenkins is still under development, we were able to do some experiments with open source projects and decided to use it in this work. In further chapters the experiments with EVOSUITE Jenkin's plugin will be better detailed.

The first method to generate test cases is by running a command line. Once we obtain EvoSuite's compiled and assembled jar file, we can call it to generate tests. The prerequisites in order to run the command line is to use Java version 8. To generate tests for a single class, it takes the the fully qualified class name which includes the package name. As an example, for a class called Person (see Figure 2.7) in the package com.example and the class path target/class, the full command line would be:

```
java -jar evosuite.jar -class tutorial.Stack -projectCP target/class.
```

The results, if no problems were found, would be two Java files:

```
evosuite-tests/example/Person_ESTest.java  
evosuite-tests/example/Person_ESTest_scaffolding.java
```

The scaffolding file contains some methods using the JUnit [6] annotations such as @After and @Before, representing actions before and after each test execution, to ensure that the tests should only fail if they really find a bug. However, this file is not important for this work as we are focusing on the generated unit tests that is located in other file. As it can be seen in Figure 2.8, the Person_ESTest class contains a certain number of JUnit tests and it can get bigger depending on the command parameters.

```
package com.example;
/**
 * @author João Veras
 */
public class Person {
    private final String firstName;
    private final String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
}
```

Figure 2.7 Java file example for a Person class. Credits: the author

Finally, after all tests are generated, EvoSuite gives as output a Comma-separated Value (CSV) file that contains statistic information about the tests generated, which is very useful for experiments. There are plenty of output variables that might help us to understand better the generated tests. Some examples of output variables include coverage, criterion used to generate tests, target class and mutation scores, among others.

2.4.2 Randoop

Carlos Pacheco and Michael D. Ernst defined in their article "Randoop: Feedback-Directed Random Testing for Java" [20], Randoop as a tool for random, but meaningful, generation of automated test unit for Java. Randoop takes as input the classes under test, the limit time for generating tests and optionally a set of properties to be checked. The output for the test generation is two test suites, one for error-revealing and another for regression (non error-revealing).

Randoop works by using *feedback-directed random testing* using execution feedback while generating tests in order to avoid redundant and illegal inputs. Randoop creates sequences of methods calls incrementally that generate and change objects. These method calls are randomly selected and the arguments are selected according to the previous sequences.

Although Randoop has a Maven plugin [27], it is not an official one and consequently the way for generating tests is via command line. Similarly to EVOSUITE, the jar file needs to be

```

/*
 * This file was automatically generated by EvoSuite
 * Sat Jun 09 20:16:10 GMT 2018
 */

import org.junit.Test;
import static org.junit.Assert.*;
import org.evosuite.runtime.EvoRunner;
import org.evosuite.runtime.EvoRunnerParameters;
import org.junit.runner.RunWith;
import example.Person;

@RunWith(EvoRunner.class)
public class PersonTest {

    @Test(timeout = 4000)
    public void test1() throws Throwable {
        Person person0 = new Person("uq2c", "uq2c");
        String string0 = person0.getLastName();
        assertEquals("uq2c", string0);
    }

    @Test(timeout = 4000)
    public void test2() throws Throwable {
        Person person0 = new Person((String) null, (String) null);
        String string0 = person0.getFirstName();
        assertNull(string0);
    }
}

```

Figure 2.8 Unit test for class Person generated by EvoSuite. Credits: the author

downloaded and, since Randoop runs on a Java 7 or Java 8 JVM, these dependencies must be satisfied. Three command are available to run, **gentests** to generate unit tests, **minimize** that makes it easier to diagnose errors in the Java files and **help** that provides a method's details.

Considering that \$RANDOOP_JAR is the path to Randoop *jar* and that we want to generate tests for the java.util.Collection during 60 seconds, the command would be:

```

java -classpath $RANDOOP_JAR randoop.main.Main gentests --testclass
java.util.Collections --time-limit=60

```

Two test suites will be generated if everything works as expected: ErrorTest and RegressionTest. The ErrorTest suite is error-revealing tests, while the RegressionTest suite does not reveal errors, but can be used to detect future regression bugs.

2.5 Continuous test generation

Continuous Test Generation (CTG) was introduced by José Campos, Andrea Arcuri, Gordon Fraser and Rui Abreu in the article "Continuous Test Generation: Enhancing Continuous Integration with Automated Test Generation" [10]. In summary, CTG happens when a Continuous Integration (CI) system is extended with an Automated Test Generation (ATG) tool.

Although tests can be generated with ATG tools such as EVOSUITE and Randoop directly in the developer's machine, it may not be very feasible. The developer's machine must allocate plenty of resources for the ATG tool and it may slow down the machine's performance and consequently impact other activities. The solution would be to generate tests in a CI system. CI systems run usually in a remote powerful machine and does not impact at all on others

activities in the development process.

Continuous Test Generation for Randoop

This chapter describes how Jenkins has been integrated to Randoop.

3.1 Proposed tool

In this chapter we propose a CTG system by extending Jenkins with Randoop in order to make it possible to generate tests during development. The main idea is to allow software teams to generate and execute tests in a remote machine and make the software development process more efficient. The solution provided is a Jenkins plugin that can be easily installed and used with few configurations.

An overview of the whole CTG process can be seen in Figure 3.1. First, the development team submits a new code to the git repository. Second, the CI starts with Jenkins cloning the code from the repository and generating the software build. Then, the Randoop's plugin scans the build in order to collect the classes to be under testing, and discarding abstract classes since they are not used for unit tests. After the tests are generated they are moved to the default test directory (e.g /src/test) inside the "randoop" folder. Finally, the tests are run and a feedback is sent to the development team warning them whether the CTG succeeded or failed. The dashed line is a further implementation for pushing the generated tests into the git repository in case everything works as expected.

3.2 Extending Jenkins

This section describes how Jenkins has been extended in terms of project configuration, input generation and output generation.

3.2.1 Project configuration

As mentioned before, the proposed tool is a Jenkins extension that can be easily installed and configured. The plugin is a Maven project. In the `pom.xml` file we had to explicitly download and unzip the Randoop plugin due the fact it is not officially available in the Maven central repository yet.

In order to download the Randoop plugin, we used *maven-download-plugin* which is basically a plugin meant to help a Maven user to download different files on different protocol in part of Maven build. The code snippet bellow represents the build part that downloads and unpacks maven.

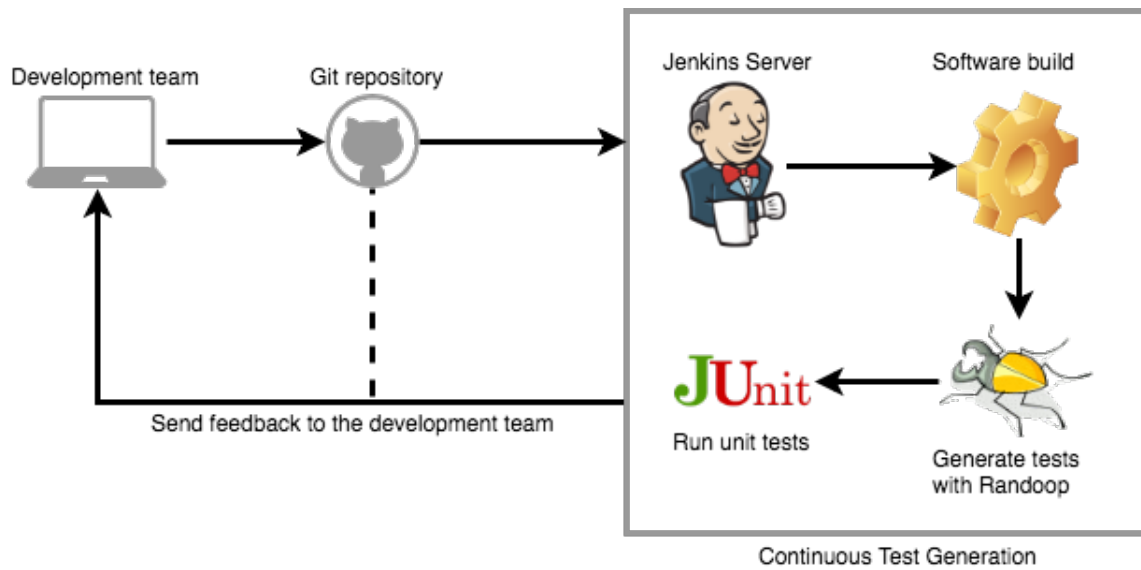


Figure 3.1 CTG diagram for Randoop. Credits: the author

```
<plugin>
  <groupId>com.googlecode.maven-download-plugin</groupId>
  <artifactId>download-maven-plugin</artifactId>
  <version>1.3.0</version>
  <executions>
    <execution>
      <id>download-randoop-all</id>
      <phase>validate</phase>
      <goals>
        <goal>wget</goal>
      </goals>
      <configuration>
        <url>${randoop-url}</url>
        <unpack>false</unpack>
        <outputDirectory>
          ${project.build.directory}/randoop-lib
        </outputDirectory>
      </configuration>
    </execution>
    <execution>
      <id>unpack-randoop-all</id>
      <phase>validate</phase>
      <goals>
        <goal>wget</goal>
      </goals>
      <configuration>
        <url>${randoop-url}</url>
        <unpack>true</unpack>
        <outputDirectory>
```

```

        ${project.build.outputDirectory}
    </outputDirectory>
</configuration>
</execution>
</executions>
</plugin>

```

The property *randoop-url* points to the Randoop releases url in the Randoop's Github website. After the plugin is downloaded it needs to be installed. So we added the following code as part of the project build as well:

```

<plugin>
  <artifactId>maven-install-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>install-file</goal>
      </goals>
      <phase>initialize</phase>
      <configuration>
        <artifactId>randoop-all</artifactId>
        <groupId>randoop</groupId>
        <version>${randoop-version}</version>
        <packaging>jar</packaging>
        <generatePom>true</generatePom>
        <file>
          ${project.build.directory}/randoop-lib
          /randoop-all-${randoop-version}.jar
        </file>
      </configuration>
    </execution>
  </executions>
</plugin>

```

Finally, the Randoop plugin can be set as a project dependency and it is ready to be used.

```

<dependency>
  <groupId>randoop</groupId>
  <artifactId>randoop-all</artifactId>
  <version>${randoop-version}</version>
  <scope>provided</scope>
</dependency>

```

3.2.2 Test generation input

We have used four main input parameters: *-classpath*, *-time-limit*, *-junit-package-name*, *-junit-output-dir* and *-testclass*. The parameter *-classpath* is a Java specific parameter to tell where Java will search for classes in order to run the program. The time (in seconds) to generate tests is captured by the parameter *-time-limit*. The parameter *-junit-package-name* is the

name of the package for the generated JUnit files and `-junit-output-dir` is the name of the directory to which JUnit files should be written. The classes to be tested are represented by the parameter `-testclass`, which should contain the fully-qualified raw name of a class to test, such as *java.util.List*.

In order to cover every project class, we implemented a scanner method that searches in the `target/classes` folder for Java classes. Once `target/classes` is in the `-classpath` parameter, tests can be generated for them through the parameter `-testclass`.

3.2.3 Test generation output

The tests starts to be generated during the Maven test phase. In the application console, it is possible to follow the test generation progress (Figure 3.2). As mentioned before, in the Randoop introduction section, there are two types of files generated: regression tests and error tests. Both of them are generated in the directory */target/generated-test-sources/* related to the project base directory. Due to the fact that there is no integration with the source code versioning system, the generated test must be downloaded from Jenkins workspace in order to integrate them into the codebase by committing the changes to the code repository.

```
Progress update: steps=1, test inputs generated=0, failing inputs=0 (7
Progress update: steps=1000, test inputs generated=720, failing inputs=8
Progress update: steps=2000, test inputs generated=1530, failing inputs=31
Progress update: steps=3000, test inputs generated=2399, failing inputs=41
Progress update: steps=4000, test inputs generated=3241, failing inputs=44
Progress update: steps=5000, test inputs generated=4105, failing inputs=54
Progress update: steps=5885, test inputs generated=4852, failing inputs=62
Progress update: steps=5885, test inputs generated=4853, failing inputs=62
Normal method executions: 1061666
Exceptional method executions: 800

Average method execution time (normal termination):    0.000630
Average method execution time (exceptional termination): 0.0886

Error-revealing test output:
Error-revealing test count: 62
Writing JUnit tests...
```

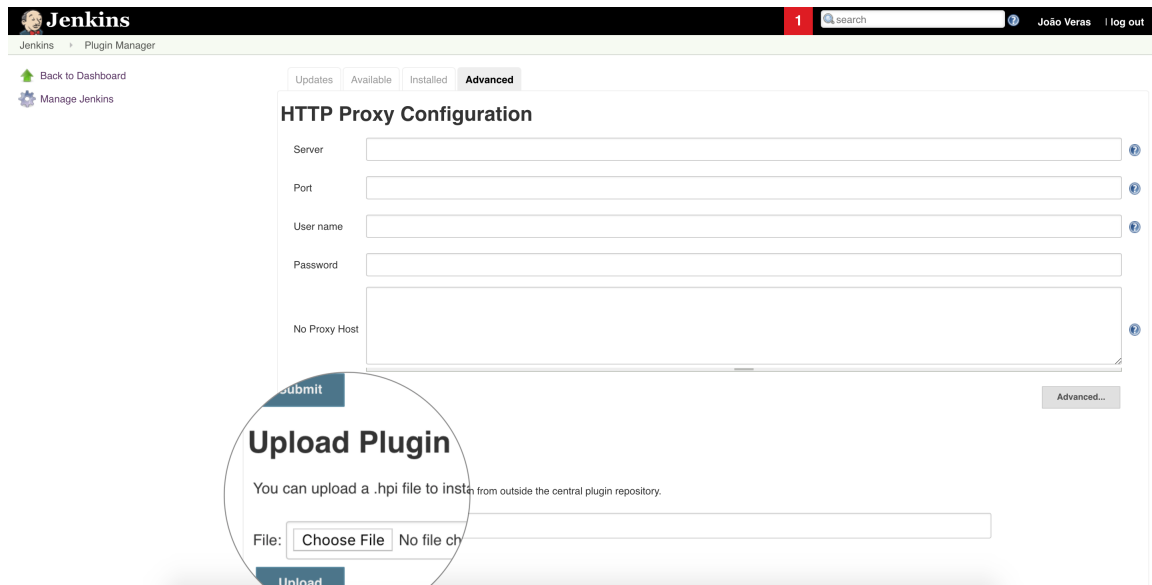
Figure 3.2 Application console during Randoop test generation. Credits: the author

3.3 Installing and configuring the plugin

Jenkins provides many ways of installing a plugin and, for the proposed Randoop plugin, since it is not an official plugin, it needs to be packaged as an `.hpi` file and uploaded directly into the Jenkins running system. Jenkins plugins need to have an `.hpi` extension in order to be uploaded into the Jenkins system. In order to generate it, some changes need to be made into the `pom.xml` file. Maven provides the field `packaging` to define how the project will be packaged when

built. In order to generate the hpi file, the packaging values must be set to hpi as the default is jar. After packaging, just run the command `mvn package`.

Once the hpi file is generated, it can be uploaded in the "Manage plugins" section under the "Advanced" tab (see Figure 3.3). After the installation, the next step is to configure the build process to include the Randoop test generation as a post-build action. As can be seen in Figure 3.4, the step name is currently "Randoop".



The screenshot shows the Jenkins web interface. At the top, the Jenkins logo and navigation links are visible. The main content area is titled 'Plugin Manager' and has tabs for 'Updates', 'Available', 'Installed', and 'Advanced'. The 'Advanced' tab is selected. Below the tabs, there is a section for 'HTTP Proxy Configuration' with input fields for 'Server', 'Port', 'User name', 'Password', and a checkbox for 'No Proxy Host'. Below this, there is a section titled 'Upload Plugin' which includes a text box for the file name and a 'Choose File' button. A circular callout highlights the 'Upload Plugin' section.

Figure 3.3 Jenkins upload plugin section. Credits: the author

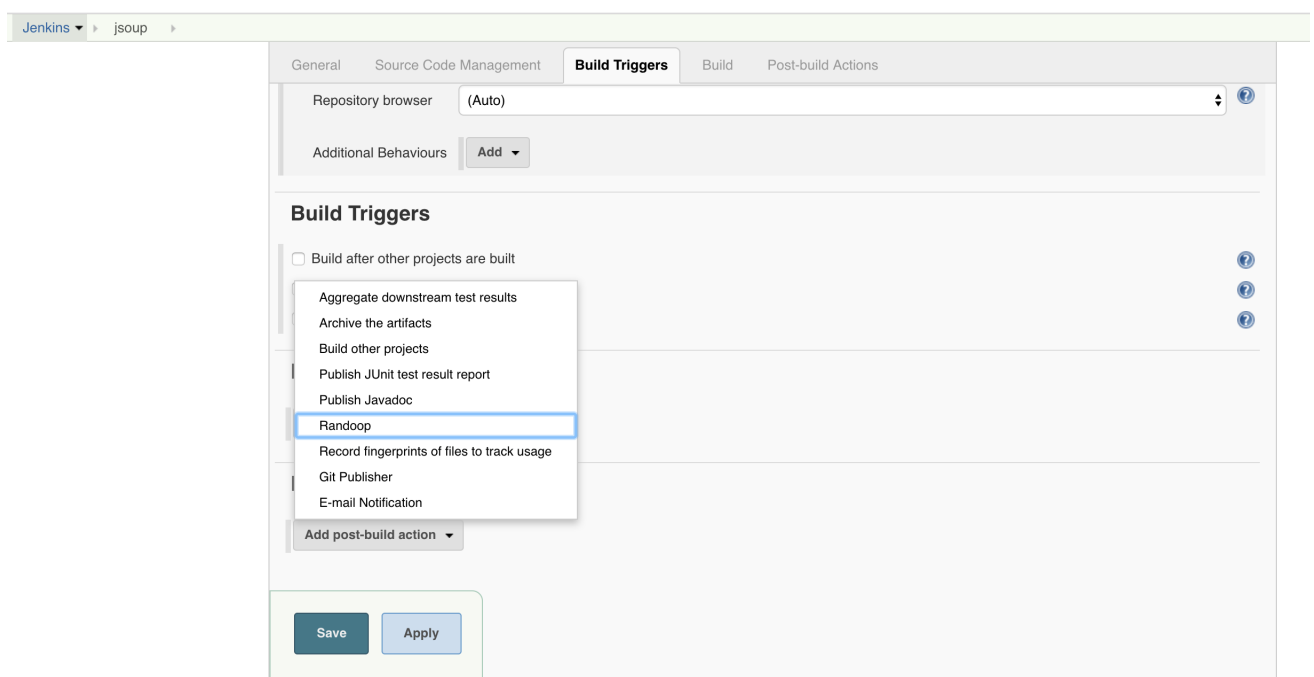


Figure 3.4 Randoop post-action configuration. Credits: the author

Experiments and Results

To compare the test generation between our proposed Randoop plugin and the EvoSuite plugin, we performed an experiment using open source projects from Github¹.

4.1 Subjects selection

We used three different Github open source repositories for our experiments. The main requirements were that the project should have commits since 2014 (in order to organize the commits in a time-line)², and that it had to be a Maven project. The need to be a Maven project was due to the limitation of the EvoSuite Jenkins plugin which currently accepts only Maven projects. We also preferred to select projects with a considerable amount of classes and good popularity on Github. A summarized information about the subjects selected can be seen in Table 4.1.

Project name	Commits	Github stars	Classes
spark	940	7,527	133
jsoup	1,199	6,056	79
joda-time	2,073	755	179

Table 4.1 Experiments subjects information.

The project **spark** is a micro framework for creating web applications; **jsoup** is a Java library for working with real-world HTML; **joda-time** is a Java library that provides a quality replacement for the Java date and time classes.

4.2 Experiments procedures

For every project, we separated about five commits with an interval from six months to one year between them. First, we forked the main repository and then created different branches for each commit. The master branch was always used representing the current state of the project. The experiments can be summarized in the following steps:

¹<https://github.com/>

²Commits before 2014 usually had deprecated dependencies or very old Java compiler version set in the pom.xml file

Fork the repository For each project, we have forked it to a repository in which we have writing access. This way, we could add configurations into the project and push changes to the git repository.

Choose past commits and create branches In the Github website, we can search for past commits, and we used this facility to choose commits in the range of six months to one year. Branches for the forked repository were created for each commit allowing us to modify the past commits and push changes as well.

Create and configure Jenkins jobs For each branch (representing the git commits) we created a Jenkins job and configured them differently. The configurations in common was to build the build using the Maven command `MVN CLEAN INSTALL`. For the non-master branches, we configured a pre-step in order to checkout them before building the project.

Run the configured jobs First, we ran the job without the ATG tool and saved the unit test results. Secondly, we added the ATG tool into the project build process. Finally we built the project one more time with the generated tests. Figure 4.1 shows the JENKINS builds at the end of the experiment for the project spark.

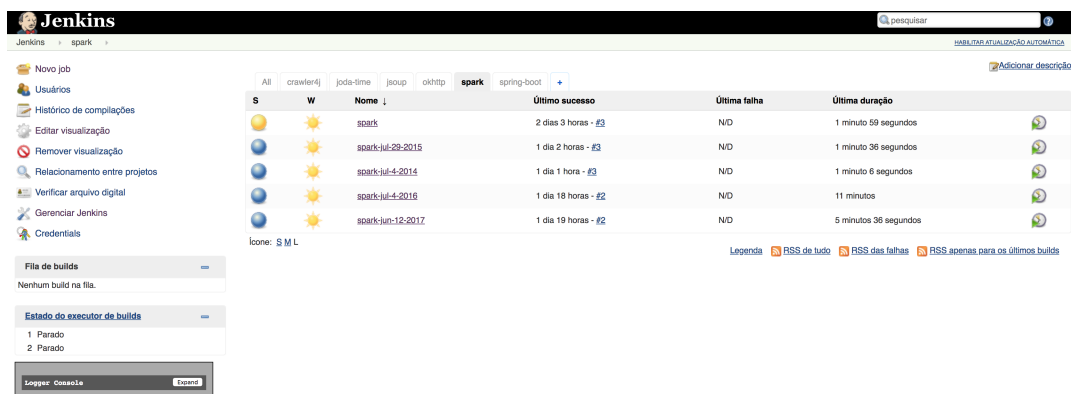


Figure 4.1 Spark project jobs for EvoSuite experiments. Credits: the author

As can be seen in Figure 4.2, after the commits were separated, JENKINS could build every one of them separately.

4.3 Experiments results

In this section we show the results of the experiments applied to the subjects selected. The tests were generated with the default parameters with the Maven command:

```
mvn -DmemoryInMB=8000 -Dcores=8 evosuite:generate evosuite:export
```

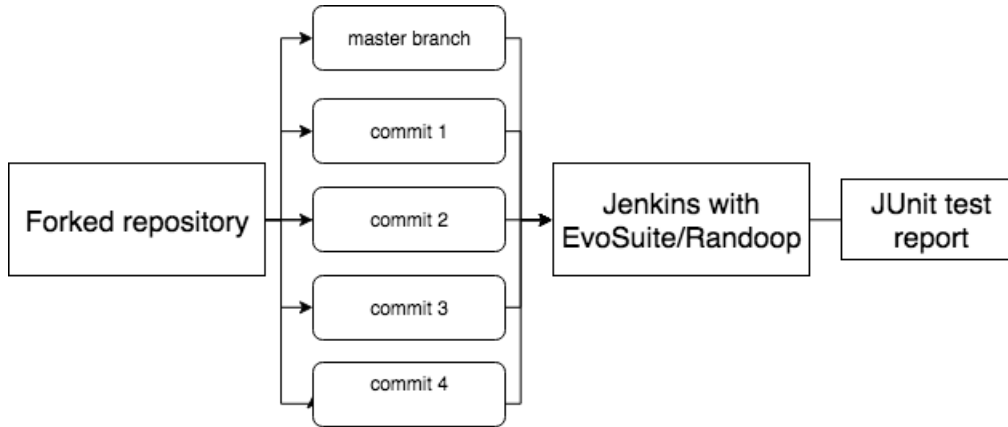


Figure 4.2 Experiments procedure diagram. Credits: the author

The parameters `memoryInMB` and `cores` represent the machine's resource allocation for EvoSuite to generate the tests. The prefix *evosuite* is followed by other two goals: *generate* and *export*. The goal *generate* is the main command to start test generation and *export* moves files from the *.evosuite* folder to the target folder defined as `${targetFolder}` in `pom.xml` as shown in the following example:

```

<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <version>1.8</version>
  <executions>
    <execution>
      <id>add-test-source</id>
      <phase>generate-test-sources</phase>
      <goals>
        <goal>add-test-source</goal>
      </goals>
      <configuration>
        <sources>
          <source>${targetFolder}</source>
        </sources>
      </configuration>
    </execution>
  </executions>
</plugin>

```

In what follows we present the results of our experiments grouped by subject.

4.3.1 Spark

The first experiment was carried out for the Spark project (see Table 4.2). This project did not have many tests and probably reached a small code coverage, and yet EvoSuite could generate

166 tests and no failures were found in the 2014 commit. Randoop could generate much more tests and found few failures (53 failures for 2,089 tests generated). One year later, in the commit from 2015, the project increased in size and the consequence was that more tests were generated by EvoSuite. The same experiment was done with the commit from 2016. However, due to the number of errors in running the plugin, no tests were generated neither for EvoSuite nor Randoop. A deeper investigation is needed to investigate the reason both tools could not generate any test for this specific commit. The red row in Table 4.2 represents the results obtained for this experiment. The commit from 2017 and 2018 (represented by the master branch at the time this work is written) have similar results with a few generated test and no failures found for EvoSuite but bugs were found, in a similar ratio from past commits, for Randoop.

Table 4.2 The results for spark.

Commit Date	spark					
	Manual		EvoSuite		Randoop	
	Tests	Failures	Tests Generated	Failures	Tests Generated	Failures
Jun 11, 2018	302	0	92	0	1,831	3
Jun 12, 2017	292	0	35	0	1,357	44
Jul 4, 2016	283	0	0	0	0	0
Jul 29, 2015	72	0	361	0	1,795	114
Jul 4, 2014	68	0	166	0	2,089	53

The conclusion for the experiment with the subject Spark was that EvoSuite might not have been better for bug finding then manual test generation. However, the time saving in test generation and execution could be a determinant factor for choosing EvoSuite since 2014. For Randoop, the failures found could be investigated to determine if they were fatal or expected (especially in the cases of null inputs generated by Randoop as this would never be a case with the application running for real).

4.3.2 JSoup

The JSoup project is larger than Spark as we can see in the results in Table 4.3. Thus, more tests could be generated and more bugs could be found for Randoop and EvoSuite. Even though the JSoup project have plenty of tests created by its developers, much more tests could be generated with both tools and many bugs could be found. From 2014 to 2018, 255 tests were created manually and more than 2,000 could have been generated by invoking EvoSuite during development since 2014. Randoop could generate more tests than EvoSuite in 2016 and 2017. However, fewer failures were found by Randoop. Randoop found more failures than EvoSuite only in 2018.

A result which is worth to highlight, for EvoSuite, is the first row in Table 4.3. Until 2017, a pattern could be found with an approximated number of test generated and test failures, but in 2018 much more tests were generated and only one failure was found. A future investigation on the tests generated from 2017 to 2018 is worth it in order to compare with the previous

Table 4.3 The results for jsoup.

jsoup						
Commit Date	Manual		EvoSuite		Randoop	
	Tests	Failures	Tests Generated	Failures	Tests Generated	Failures
May 12, 2018	669	0	2,027	1	1,439	39
Jun 10, 2017	540	0	921	96	1,236	12
Jun 17, 2016	486	0	913	96	1,093	23
Jun 17, 2015	457	0	1,019	117	938	9
Jul 23, 2014	414	0	1,146	94	809	11

bugs found from 2014 to 2017 to verify if they were fixed and covered. For Randoop, the test generation history seems to have been more consistent with no outliers.

4.3.3 Joda-time

Joda-time is the largest project in the experiment (see Table 4.1) and this reflects in the generated tests numbers. Table 4.4 shows that manual tests did not increased much from 2014 to 2018. However, EvoSuite generated 1,542 ($3,329 - 1,787 = 1,542$) additional tests from 2014 to 2016.

Table 4.4 The results for joda-time.

joda-time						
Commit Date	Manual		EvoSuite		Randoop	
	Tests	Failures	Tests Generated	Failures	Tests Generated	Failures
Jun 11, 2018	4,222	0	3,199	180	835	92
Jun 12, 2017	4,207	0	3,365	196	1,448	204
Jul 4, 2016	4,191	0	3,329	342	578	69
Jul 29, 2015	4,157	0	2,788	560	1,305	168
Jul 4, 2014	4,093	0	1,787	113	1,254	153

The results for Randoop has more variations. For example, from 2015 to 2016 the number of generated tests decreased followed by the number of failures. On the other hand, the ratio between tests generated and bugs found was higher than the results for EvoSuite.

4.4 Concluding Remarks

In general, our experiments have shown that Continuous Test Generation with both EvoSuite and Randoop are beneficial: for some commits, thousands of test cases are generated automatically. Although Randoop has been slightly more consistent in bug founding (EvoSuite has not found any bugs for Spark), as they are complementary test generation tools (they use different approaches), we cannot dismiss one in favor of the other.

CHAPTER 5

Related Work

This chapter reviews related work on software testing, continuous integration, and automated test generation, which are the main topics of this work.

Campos *et al.* [10] have done a research on how to enhance CI systems with automated test generation. In this research, a significant improvement on test data and reduction in test generation time by up to +83% was achieved. On the other hand, the authors could identify that, even though the experiments shows as result improvement in test quality, it can cause some problems. Testability problems can be found when there are regular runs of CTG and some code improvements might be needed to be applied.

Arcuri *et al.* [9] proposed a solution that invokes EvoSuite during software development and one of the proposed solution is the integration of EvoSuite and Jenkins. The proposed Jenkins plugin uses an approach to select only tests for classes that have been added or modified, or tests that have achieved higher coverage, or tests that have aimed at a goal not covered in previous tests. The plugin also provides a visualization that shows the code coverage and time spent on test generation. Some statistic information are also available for testable classes, number of generated test cases, among others. Also, it is possible to commit and push the generated test into the repository configured for the build. In order to generate the tests, EvoSuite must be part of the software build process. This can be done through some Maven configuration steps. The current version of the plugin does not generate the tests without maven configuration.

Robinson *et al.* [21] present a suite of techniques that enhances Randoop. The problem found was that tests automatically generated were designed to increase coverage and find bugs rather than generate maintainable regression test suites. This led the research to propose a maintainable regression test automatically generated by Randoop on real software programs rather than library code. The main contribution was an extension for Randoop that creates more maintainable and easier to read codes with high coverage and mutation kill score.

Faser *et. al* [15] have tried to answer in their work whether ATG tools really helps software testers. The main purpose was to identify how ATG tools impact on the test process compared to manual testing. ATG tools are mostly evaluated by automatically generated metrics without involving humans. With that in mind, the experiments involved 49 humans and addressed the following research metrics: *the structural code coverage achieved during testing, the ability of testers to detect faults in the class under test, the number of tests mismatching the intended behaviour of the class and the ability of produced test suites to detect regression faults?*.

5.1 Concluding Remarks

In this work, we have proposed an extension for Jenkins to generate new test suites using Randoop. Due to the lack of documentation, the whole development was based on existing open source plugins. The plugin development was mainly based on the extension proposed by Arcuri *et al.* [9]. The proposed plugin was carefully studied in order to better understand the best way to generate tests in a CI system.

Although the proposed plugin was based on the EvoSuite plugin for Jenkins, different approaches were used to generate tests. For the EvoSuite plugin, the test generation must be part of the build process, i.e. changes in the Maven configuration file must be done in order to generate the tests. Our proposed plugin generates the tests without any changes in the main codebase. This way, the continuous test generation is completely handled by Jenkins.

The studies in Campos *et al.* [10] have motivated even more to enhance a continuous integration system with an ATG tool and to understand properly the benefits for generating test continuously and how to obtain results from experiments. In the same way, the presented techniques by Robinson *et al.* [21] have helped to better understand Randoop and the importance to make automated test generation even closer to the software development basis.

CHAPTER 6

Conclusion

In this work we have introduced a solution that allows development teams to generate automated random tests during software development. A Jenkins plugin for Randoop was created in order to make it as easy and automatic as possible. The plugin can generate and run regression tests while using previous tests as feedback for new tests. Every test case is saved in the Jenkins workspace and can be pushed into the git repository with extra Jenkins configuration.

Due to the lack of documentation on how to create Jenkins plugin, we relied on the Jenkins plugin proposed by Acuri *et al.* [9]. The source code is available in the EvoSuite github repository and we studied it in order to make it possible to replicate it for another ATG tool.

Another challenge was to create a history based experiments with the subjects selected. At first, we selected more than five projects to generate tests with EvoSuite and analyzed how CTG works with an existing tool. A lot of changes had to be made for large projects such as the Spring-Boot framework and even though we could not make the test generation as simple as it should be. Thus, we decided to keep three projects with simpler architecture but with a significant amount of classes.

Our results show how Continuous Test Generation is beneficial: thousands of test cases are produced and many failures were found.

As limitations, we are aware that our proposed tool can be improved. In future work, our central idea is to have a framework for ATG tools that allows development teams to choose any ATG tool among others in a CI system. Also, a visualization can be implemented for a better interpretation of the CTG.

Bibliography

- [1] Agitar One. <http://www.agitar.com>. Accessed: 2018-06-21.
- [2] Build tools – java survey results. <http://kaviddiss.com/2016/01/23/build-tools-java-survey-results/>. Accessed: 2018-06-27.
- [3] Git. <https://git-scm.com/>. Accessed: 2018-06-27.
- [4] Jenkins. <https://jenkins.io/>. Accessed: 2018-06-27.
- [5] Jenkins plugins. <https://plugins.jenkins.io/>. Accessed: 2018-05-26.
- [6] Junit. <https://junit.org/>. Accessed: 2018-07-11.
- [7] Parasoft Jtest. <https://www.parasoft.com/products/jtest>. Accessed: 2018-05-26.
- [8] Randoop - automatic unit test generation for Java. <https://randoop.github.io/randoop/>. Accessed: 2018-05-26.
- [9] ARCURI, A., CAMPOS, J., AND FRASER, G. Unit test generation during software development: EvoSuite plugins for Maven, IntelliJ and Jenkins. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (April 2016), pp. 401–408.
- [10] CAMPOS, J., ARCURI, A., FRASER, G., AND ABREU, R. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA, 2014), ASE '14, ACM, pp. 55–66.
- [11] DUVAL, P. M., M. S., AND GLOVER, A. *Continuous integration: improving software quality and reducing risk*. Addison-Wesley Signature Series (Fowler). Addison-Wesley Professional, 2007.
- [12] ELBAUM, S., ROTHERMEL, G., AND PENIX, J. Techniques for improving regression testing in Continuous Integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, ACM, pp. 235–245.
- [13] FOWLER, M. Continuous integration, May 2006.

- [14] FRASER, G., AND ARCURI, A. 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* (New York, NY, USA, 2011), ESEC/FSE '11, ACM, pp. 416–419.
- [15] FRASER, G., STAATS, M., MCMINN, P., ARCURI, A., AND PADBERG, F. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2013), ISSTA '13, ACM.
- [16] GOOSSENS, M., MITTELBACH, F., AND SAMARIN, A. *The \LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.
- [17] LOUKIDES, M., AND SONATYPE INC. (MOUNTAIN VIEW, C. *Maven: the definitive guide : [everything you need to know from ideation to deployment]*. O'Reilly, Beijing [u.a.], 2008.
- [18] MERKEL, D. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J.* 2014, 239 (Mar. 2014).
- [19] ORSO, A., AND ROTHERMEL, G. Software testing: A research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering* (New York, NY, USA, 2014), FOSE 2014, ACM, pp. 117–132.
- [20] PACHECO, C., AND ERNST, M. D. Randoop: Feedback-directed random testing for Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion* (New York, NY, USA, 2007), OOPSLA '07, ACM, pp. 815–816.
- [21] ROBINSON, B., ERNST, M. D., PERKINS, J. H., AUGUSTINE, V., AND LI, N. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2011), ASE '11, IEEE Computer Society, pp. 23–32.
- [22] SAFF, D., AND ERNST, M. D. Reducing wasted development time via continuous testing. In *Proceedings of the 14th International Symposium on Software Reliability Engineering* (Washington, DC, USA, 2003), ISSRE '03, IEEE Computer Society, pp. 281–.
- [23] SHAMSHIRI, S., JUST, R., ROJAS, J. M., FRASER, G., MCMINN, P., AND ARCURI, A. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Nov 2015), pp. 201–211.
- [24] SMART, J. F. *Jenkins: The Definitive Guide*. O'Reilly Media, 2011.

- [25] WALLER, J., EHMKE, N. C., AND HASSELBRING, W. Including performance benchmarks into Continuous Integration to enable DevOps. *SIGSOFT Softw. Eng. Notes* 40, 2 (Apr. 2015), 1–4.
- [26] WIKIPEDIA, THE FREE ENCYCLOPEDIA. Atomic force microscopy, 2013. [Online; accessed April 27, 2013].
- [27] ZAPLATYNSKI, M. randoop-maven-plugin. <https://github.com/zaplatynski/randoop-maven-plugin>, 2017.