2016 IEEE International Conference on Software Testing, Verification and Validation

Unit Test Generation During Software Development: EvoSuite Plugins for Maven, IntelliJ and Jenkins

Andrea Arcuri
Scienta, Norway, and
University of Luxembourg, Luxembourg

José Campos, Gordon Fraser
University of Sheffield
Sheffield, UK

***Abstract*—Different techniques to automatically generate unit tests for object oriented classes have been proposed, but how to integrate these tools into the daily activities of software development is a little investigated question. In this paper, we report on our experience in supporting industrial partners in introducing the EvoSuite automated JUnit test generation tool in their software development processes. The first step consisted of providing a plugin to the Apache Maven build infrastructure. The move from a research-oriented point-and-click tool to an automated step of the build process has implications on how developers interact with the tool and generated tests, and therefore, we produced a plugin for the popular IntelliJ Integrated Development Environment (IDE). As build automation is a core component of Continuous Integration (CI), we provide a further plugin to the Jenkins CI system, which allows developers to monitor the results of EvoSuite and integrate generated tests in their source tree. In this paper, we discuss the resulting architecture of the plugins, and the challenges arising when building such plugins. Although the plugins described are targeted for the EvoSuite tool, they can be adapted and their architecture can be reused for other test generation tools as well.**

I. Introduction

The EvoSuite tool automatically generates JUnit tests for Java software [1]–[4]. Given a class under test (CUT), EvoSuite creates sequences of calls that maximize testing criteria such as line and branch coverage, while at the same time generating JUnit assertions to capture the current behavior of the CUT. Experiments on open-source projects and industrial systems [5] have shown that EvoSuite can successfully achieve good code coverage — but how should it be integrated in the development process of the software engineers?

In order to answer this question, the interactions between a test generation tool and a software developer have been subjected to controlled empirical studies and observations [6]–[8]. However, the question of integrating test generation into the development process goes beyond the interactions of an individual developer with the tool: In an industrial setting, several developers work on the same, large code base, and a test generation tool should smoothly integrate into the current processes and tool chains of the software engineers.

There are different ways to access a test generation tool. The first option to run EvoSuite is from the command line. If the tool is compiled and assembled in a standalone executable jar (e.g., `evosuite.jar`), then it can be called on a CUT (e.g., `org.Foo`) as follows:

```
java -jar evosuite.jar org.Foo
```

However, in a typical Java project the full classpath needs to be specified (e.g., as a further command line input). This is necessary to tell the tool where to find the bytecode of the CUT and of all of its dependency classes. For example, if the target project is compiled in a folder called `build`, then in EvoSuite one can use:

```
java -jar evosuite.jar -class org.Foo -projectCP build
```

where the option `-class` is used to specify the CUT, and the option `-projectCP` is used for specifying the classpath.

This approach works fine if EvoSuite is used in a "static" context, e.g., when the classpath does not change, and a user tests the same specific set of classes several times. A typical example of such a scenario is the running of experiments on a set of benchmarks in an academic context [5] — which is quite different from an industrial use case. An industrial software system might have hundreds, if not thousands, of entries in the classpath, which might frequently change when developers push new changes to the source repository (e.g., Git, Mercurial or SVN). Manually specifying long classpaths for every single submodule is not a viable option.

Usability can be improved by integrating the test generation tool directly into an IDE. For example, EvoSuite has an Eclipse plugin [1] which includes a jar version of EvoSuite. Test generation can be activated by the developer by selecting individual classes, and the classpath is directly derived from the APIs of the IDE itself. However, this approach does not scale well to larger projects with many classes and frequent changes. Furthermore, EvoSuite requires changes to the build settings that have to be consistent for all developers of a software project, as EvoSuite's simulation of the *environment* of the CUT requires inclusion of a dependency jar file (containing mocking infrastructure for, e.g., the Java API of the file system [9] and networking [10]).

To overcome these problems, we have developed a set of plugins for common software development infrastructure in industrial Java projects. In particular, in this paper we present a plugin to control EvoSuite from Apache Maven[1] (Section II), as well as plugins for IntelliJ IDEA[2] (Section III) and Jenkins CI[3] (Section IV) to interact with the Apache Maven plugin.

[1]https://maven.apache.org, accessed January 2016.
[2]https://www.jetbrains.com/idea, accessed January 2016.
[3]https://jenkins-ci.org, accessed January 2016.

978-1-5090-1827-7/16 $31.00 © 2016 IEEE
DOI 10.1109/ICST.2016.44

II. Unit Test Generation in Build Automation

Nowadays, the common standard in industry to compile and assemble Java software is to use automated build tools. Maven is perhaps the currently most popular one (an older one is Ant, whereas the more recent Gradle is currently gaining momentum). Integrating a unit test generation tool into an automated build tool consists of supporting execution of generated tests, as well as generation of new tests.

A. Integrating Generated Tests in Maven

In order to make tests deterministic and isolate them from the environment, EvoSuite requires the inclusion of a runtime library [9]. When using a build tool like Maven, it is easy to add third-party libraries. For example, the runtime dependency for the generated tests of EvoSuite can be easily added (and automatically downloaded) for example by copy&pasting the following entry into the `pom.xml` file defining the build:

```
<dependency>
  <groupId>org.evosuite</groupId>
  <artifactId>evosuite-standalone-runtime</artifactId>
  <version>1.0.2</version>
  <scope>test</scope>
</dependency>
```

Once this is set, the generated tests can use this library, which is now part of the classpath. This is important because, when a software project is compiled and packaged (e.g., with the command `mvn package`), all the test cases are executed as well to validate the build.

However, when we generated test cases for one of our industrial partners for the first time, building the target project turned into mayhem: some generated tests failed, as well as some of the existing manual tests (i.e, the JUnit tests manually written by the software engineers), breaking the build. The reason is due to how classes are instrumented: The tests generated by EvoSuite activate a Java Agent to perform runtime bytecode instrumentation, which is needed to replace some of the Java API classes with our mocks [9]. The instrumentation is done when the tests are run, and can only be done when a class is loaded for the first time. On one hand, if the manual existing tests are run first before the EvoSuite ones, the bytecode of the CUTs would be already loaded, and instrumentation cannot take place, breaking (i.e., make them fail) all the generated tests depending on it. On the other hand, if manual tests are run last, they will use the instrumented versions, and possibly fail because they do not have the simulated environment configured for them.

There might be different ways to handle this issue, as for example forcing those different sets of tests to run on independently spawned JVMs. However, this might incur some burden on the software engineers' side, who would need to perform the configuration, and adapt (if even possible) all other tools used to report and visualise the test results (as we experienced). Our solution is twofold: (1) each of our mocks has a rollback functionality [10], which is automatically activated after a test is finished, so running manual tests after the generated ones is not a problem; (2) we created a *listener* for the Maven test executor, which forces the loading and instrumentation of all CUTs before *any* test is run, manual tests included. Given this solution, engineers can run all the tests in any order, and in the same classloader/JVM. This is achieved simply by integrating the following entry into the `pom.xml` where the Maven test runner is defined (i.e., in `maven-surefire-plugin`):

```
<property>
 <name>listener</name>
 <value>org.evosuite.runtime.InitializingListener</value>
</property>
```

B. Generating Tests with Maven

The configuration options discussed so far handle the case of running generated tests, but there remains the task of generating these tests in the first place. Although invoking EvoSuite on the local machine of a software engineer from an IDE may be a viable scenario during software development, it is likely not the best for legacy systems. When using EvoSuite for the first time on a large industrial software system with thousands of classes, it is more reasonable to run EvoSuite on a remote dedicated server, as it would be a very time consuming task. To simplify the configuration of this (e.g., to avoid manually configuring classpaths on systems with dozens of `pom.xml` files in a hierarchy of submodules) and to avoid the need to prepare scripts to invoke EvoSuite accordingly, we implemented a Maven plugin with an embedded version of EvoSuite. For example, to generate tests for all classes in a system using 64 cores, a software engineer can simply type:

```
mvn -Dcores=64 evosuite:generate
```

To get an overview of all execution goals, the EvoSuite Maven plugin can be called as follows:

```
mvn evosuite:help
```

or as follows:

```
mvn evosuite:help -Ddetail=true -Dgoal=generate
```

to get the list of parameters of, e.g., the `generate` goal. In particular, it is possible to configure aspects such as number of cores used (`cores`), memory allocated (`memoryInMB`), or time spent per class (`timeInMinutesPerClass`).

It is further possible to influence how the time is allocated to individual classes using the `strategy` parameter: The `simple` strategy allocates the time specified in the `timeInMinutesPerClass` per class. By default, EvoSuite will use the `budget` strategy, which allocates a time-budget proportional to the complexity of a class. As a proxy to complexity, EvoSuite uses the number of branches to determine whether class A is more complex than class B. That is, classes with more branches will have more time available to be tested. First, EvoSuite determines the maximum and the minimum time budget available. The minimum time budget is the minimum time per class (by default 1 minute) multiplied by the total number of classes. The maximum time budget is `timeInMinutesPerClass` multiplied by the total number of classes. The difference between maximum and minimum time budget is called `extraTime` and it is used to give more time to

more complex classes. Assuming there is an `extraTime` of e, the time budget per branch will be $\frac{e}{|branches|}$. Then, each CUT C, will have a time budget of $minTimeBudgetPerClass + (timePerBranch \times |branches_C|)$.

Further implemented strategies are the experimental `seeding` strategy [11], where EvoSuite tries to test classes in the order of dependencies to allow re-use of Java objects, and `history`, which is explained in Section IV.

To get an overview of tests generated so far, one can use:

```
mvn evosuite:info
```

By default, EvoSuite stores tests in the `.evosuite/evosuite-tests` hidden folder. Once the developer has inspected the tests and decided to integrate them into the source folder, this can be done using the following command:

```
mvn evosuite:export
```

The export command copies the generated tests to another folder, which can be set with the `targetFolder` option (default value is `src/test/java`). Tests will only be executed by the `mvn test` command once they are in `src/test` (unless Maven is configured otherwise).

To enable the EvoSuite plugin, the software engineer would just need to copy&paste the following plugin declaration to the root `pom.xml` file:

```
<plugin>
  <groupId>org.evosuite.plugins</groupId>
  <artifactId>evosuite-maven-plugin</artifactId>
  <version>1.0.2</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare</goal>
      </goals>
      <phase>process-test-classes</phase>
    </execution>
  </executions>
</plugin>
```

By doing this, there is no further need to do any installation or manual configuration: Maven will automatically take care of it. Note: if a plugin is not in the Maven Central Repository[4], one needs to add the URL of the server where the plugin is stored, but that needs to be done just once (e.g., in a corporate cache using a repository manager like Nexus[5]).

Once the EvoSuite Maven plugin is configured by editing the `pom.xml` file (which needs to be done only once), if an engineer wants to generate tests on a new server, then it is just a matter of uploading the target system there (e.g., `git clone` if Git is used as source repository manager), and then typing `mvn evosuite:generate`. That is all that is needed to generate tests with EvoSuite's default configuration (some parameters can be added to specify the number of cores to use, for how long to run EvoSuite, if only a subset of classes should be tested, etc.).

[4]http://search.maven.org, accessed January 2016.

[5]http://www.sonatype.com/nexus/solution-overview, accessed January 2016.

III. IDE Integration of Unit Test Generation

Once the generated unit tests require a runtime dependency to run, embedding EvoSuite within an IDE plugin (as in the past we did for Eclipse) becomes more difficult because of potential EvoSuite version mismatches: the IDE plugin could use version X, whereas the project could have dependency on Y. Trying to keep those versions aligned is not trivial: a software engineer might work on different projects at the same time, each one using a different version; a software engineer pushing a new version update in the build (e.g., by changing the dependency version in the `pom.xml` file and then committing the change with Git) would break the IDE plugin of all his/her colleagues, who would be forced to update their IDE plugin manually; etc.

Our solution is to keep the IDE plugin as lightweight as possible, and rely on the build itself to generate the tests. For example, the IDE plugin would just be used to select which are the CUTs, and what parameters to use (e.g., how long the search should last). Then, when tests need to be generated, the IDE plugin just spawns a process that calls `mvn evosuite:generate`. By doing this, it does not matter what version of EvoSuite the target project is configured with, and updating it will be transparent to the user. Furthermore, every time a new version of EvoSuite is released, there is no need to update the IDE plugin, just the `pom.xml` file (which needs to be done only once and just by one engineer).

However, to achieve this, the interfaces between the IDE plugin and the Maven plugin need to be *stable*. This is not really a problem in automated test data generation, where in general there are only few parameters a user is really interested into: for what CUTs should tests be generated for, what resources to use (memory, CPU cores, and time).

This approach worked well for some of our industrial partners, but not for all of them: For example, some use Gradle to build their software rather than Maven. Furthermore, relying on a build tool does not work when no build tool is used, e.g. when a new project is created directly in the IDE. To cope with the issue of handling build tools for which we have no plugin (yet), or handling cases of no build tool at all, we also found it necessary to have the option of using an external command line EvoSuite executable, which the IDE plugin calls on a separate spawned process. As the corresponding jar file does not need to be part of the build, it can be simply added directly to the source repository (e.g., Git) without needing to change anything regarding how the system is built. In this way, all developers in the same project will use the same version, and do not need to download and configure it manually.

Regarding the runtime dependency for the generated tests, this is not a problem for build tools like Ant/Ivy and Gradle, as they can make use of Maven repositories. However, when no build tool is employed, the runtime dependency needs to be added and configured manually (as for any other third-party dependency). Note: the EvoSuite executable could be used as runtime dependency as well (it is a superset of it), but it would bring many new third-party libraries in the build. This

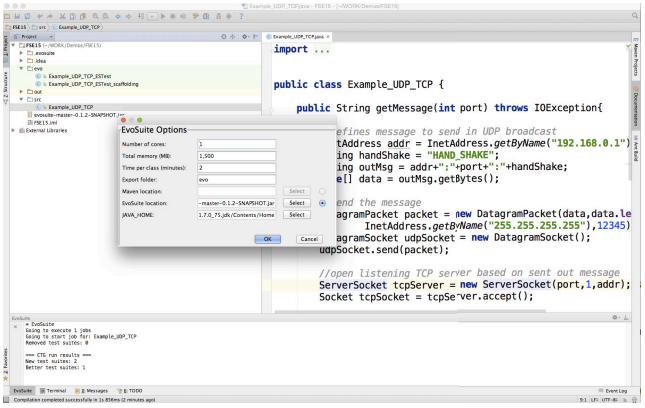

Fig. 1. Screenshot of the EVOSUITE plugin for IntelliJ IDEA, when applied on the code example from [10].

might lead to version mismatch problems if some of these libraries are already used in the project.

This architecture is different from what we originally had for our Eclipse plugin. To experiment with it, we started a new plugin for a different IDE, namely IntelliJ IDEA. This was also driven by the fact that most of our industrial partners use IntelliJ and not Eclipse. Figure 1 shows a screenshot of applying EVOSUITE to generate tests for the motivating example used in [10]. A user can select one or more classes or packages in the project view, right click on them, and start the action `Run EvoSuite`. This will show a popup dialog, in which some settings (e.g., for how long to run EVOSUITE) can be chosen before starting the test data generation. Progress is shown in a tool window view.

One of the first things we found out by working on the IntelliJ plugin is that, in general, embedding and executing a test data generation tool on the same JVM of the plugin (as we did with Eclipse) is not a viable option. If you are compiling a project with Java 8, for example, that does *not* mean that the IDE itself is running on Java 8 (recall that IDEs like IntelliJ, Eclipse and NetBeans are written in Java and execute in a JVM). For example, up to version 14, IntelliJ for Mac used Java 6, although IntelliJ can be used to develop software for Java 8. The reason is due to some major performance GUI issues in the JVM for Mac in both Java 7 and Java 8. An IDE plugin will run in the same JVM of the IDE, and so needs to be compiled with a non-higher version of Java. In our case, as EVOSUITE is currently developed/compiled for Java 8, calling it directly for the IDE plugin would crash it due to bytecode version mismatch. The test data generation tool has to be called on a spawned process using its own JVM.

IV. CONTINOUS TEST GENERATION

Although generating tests on demand (e.g., by directly invoking the Maven/IntelliJ plugins) on a developer machine is feasible, there can be many reasons for running test generation on a remote server. In particular, running EVOSUITE on many classes repeatedly after source code changes might be cumbersome. To address this problem, we introduced the concept of Continous Test Generation [11], [12] (CTG), where Continous Integration (CI) (e.g., Jenkins and Bamboo) is extended with automated test generation. In a nutshell, a remote server will run EVOSUITE at each new code commit using the EVOSUITE Maven plugin. Only the new tests that improve upon the existing regression suites will be sent to the users using a plugin to the Jenkins CI system.

A. Invoking EvoSuite in the Context of CTG

During CTG, EVOSUITE will be invoked on the same software project repeatedly using the Maven plugin, by setting the strategy to `history`. This strategy changes the budget allocation such that more time is spent on new or modified classes than old classes, under the assumption that new or modified code is more likely to be faulty [13]. Furthermore, instead of starting each test generation from scratch, the `history` strategy re-uses previously generated test suites as a seed when generating the initial population of the Genetic Algorithm, to start test generation with some code coverage, instead of trying to cover goals already covered by a previous execution of CTG.

The Maven plugin creates a directory called `.evosuite` under the project directory where all the files generated and/or used during test generation are kept. To be independent of any Source Control Management (SCM), we have implemented a very simple system to check which classes (i.e., Java files) have changed from one commit to another one. Under `.evosuite`, CTG creates two files: `hash_file` and `history_file`. Both files are based on a two column format, and are automatically created by the EVOSUITE Maven plugin. The first one contains as many rows as there are Java files in the Maven project, and each row is composed of the full path of each Java file and its hash. The hash value allows EVOSUITE to determine whether a Java file has been changed. Although this precisely identifies which Java files (i.e., classes) have been changed, it does not take into account whether the change was in fact a source change or just, for example, a JavaDoc change. Future work should try to improve this feature using, for example, a diff parser, or comparing the AST of the previous Java file and the current one. The second file (`history_file`) just keeps the list of new/modified classes. A class is considered *new* if there is no record of that class on the `hash_file`. A class is considered as *modified*, if its current hash value is different from the value on `hash_file`. Similar to Git output, the first column of `history_file` is the status of the Java file: "A" means added, and "M" means modified. The second column is the full path of the Java file.

Each CTG call also creates a temporary directory (under the `.evosuite` folder) named with the format `tmp_year_month_day_hour_minutes_seconds`. All files (such as .log, .csv, .java files, etc) generated by EVOSUITE during each test generation will be saved in this temporary directory.

At the end of each test generation, the best test suites will be copied to a folder called `best-tests`. This folder will only contain test suites that improve over already existing tests.

For this, it is necessary to measure the coverage achieved by the existing tests, which can be done using the following command:

```
mvn evosuite:coverage
```

This command instruments all classes under `src/main/java` (for typical Maven projects) and runs all test cases from `src/test/java`. EVOSUITE executes all test cases using the JUnit API on all classes, and determines the coverage achieved on all of EVOSUITE's target code coverage criteria. Future improvements of this option will try to re-use `maven-surefire`[6] plugin to run the test cases instead of directly using the JUnit API.

In order to be copied to `best-tests`, a test suite for a CUT needs to either be (a) generated for a class that has been added or modified, (b) achieve a higher code coverage than the existing tests, or (c) cover at least one additional coverage goal that is not covered by the existing tests.

B. Accessing Generated Tests from Jenkins

Once CTG is part of the build process (e.g., through the Maven plugin), then integrating it in a CI system becomes easier. We have developed a plugin for the Jenkins CI system which allows developers to:

- Visualize code coverage and time spent on test generation;
- Get statistic values like coverage per criterion, number of testable classes, number of generated test cases, total time spent on test generation per project, module, build, or class;
- View the source-code of the generated test suites per class;
- Commit and push the new generated test suites to a specific branch[7].

The Jenkins plugins relies on information produced by the underlying Maven plugin, which generates a file `project_info.xml` with detailed information. Consequently, reproducing the functionality of the Jenkins plugin for other CI platforms should be straightforward.

Currently, the EVOSUITE Jenkins plugin is available for download on our webpage at `www.evosuite.org/downloads`. To install it, Jenkins provides an "Upload Plugin" option in the "Manage Jenkins" menu, where the `evosuite.hpi` file can be uploaded and installed. Once installed, the EVOSUITE Jenkins plugin runs as a "post-build" step, in which the output of the EVOSUITE Maven plugin is displayed on the CI web interface. This is exactly the same type of architecture used by Emma[8] (a widely used Java tool for code coverage): the Emma Maven plugin needs to be added to the `pom.xml` project descriptor, and then it needs to be called as part of the CI build. To enable the Jenkins plugin, users just have to access the "configure" page of their project and add EVOSUITE as one of the "post-build" actions. As shown in Figure 2, there are three options to configure EVOSUITE:

[6]The Maven Surefire plugin is used during the test phase of a maven project to execute all unit tests.

[7]Currently, EVOSUITE just supports Git repositories.

[8]https://wiki.jenkins-ci.org/display/JENKINS/Emma+Plugin, accessed January 2016.

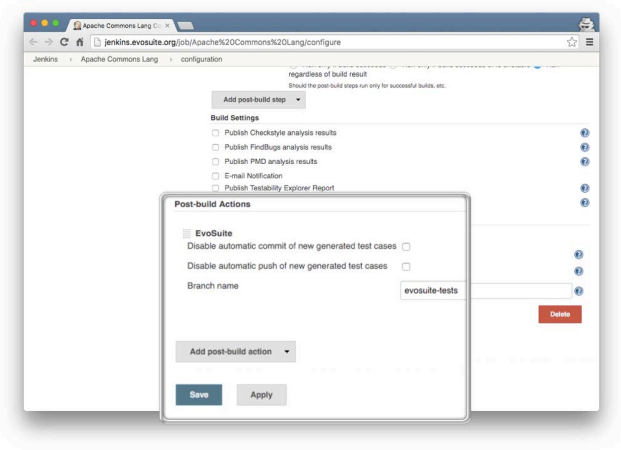

Fig. 2. Configuring the EVOSUITE Jenkins plugin.

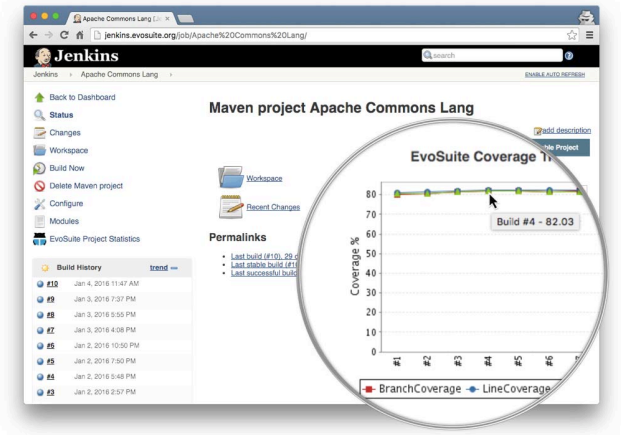

Fig. 3. Jenkins dashboard with EVOSUITE plugin applied on Apache Commons Lang project.

- **Automatic commits**: The plugin can be configured to automatically commit newly generated test suites to the Git repository. If this option is deactivated, then the generated test suites will remain on the CI system and users can still use the CI web interface to access the generated test suites of each class.
- **Automatic push**: The plugin can be configured to automatically push commits of generated tests to a remote repository.
- **Branch name**: To minimize interference with mainstream development, it is possible to let the plugin push to a specific branch of the repository.

Consequently, when the development team of a project is already running a CI server like Jenkins and is using a build tool like Maven, then adding and configuring the EVOSUITE Jenkins plugin is a matter of a few minutes. Once configured, and after the first execution of CTG on the project under test, a coverage plot will be shown on the main page of the project (as shown in Figure 3).

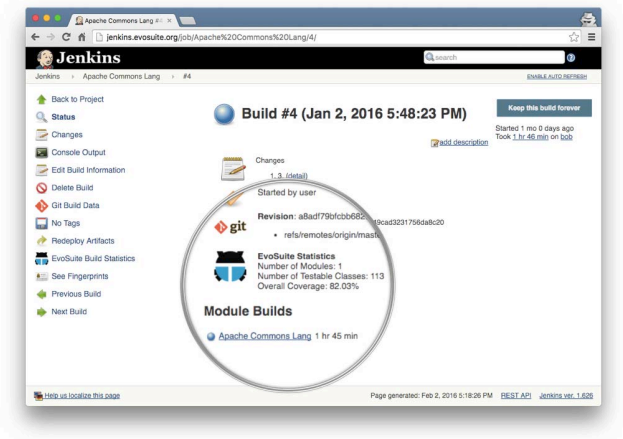

Fig. 4. Jenkins build dashboard with EvoSuite statistics like, for example, *number of testable classes*, or *overall coverage*.

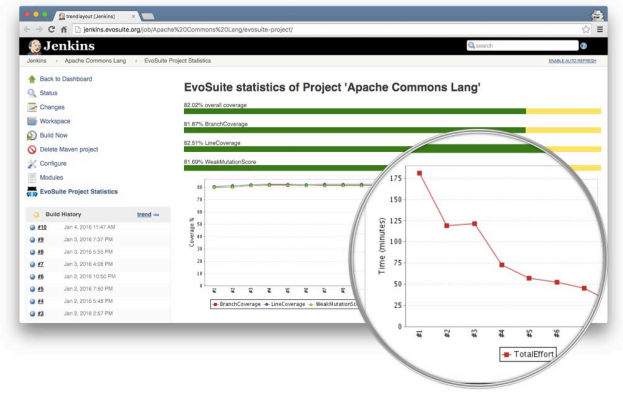

Fig. 5. EvoSuite statistics such as *overall and coverage achieved by each criterion*, and *time spend on generation* of a project.

In the plot shown in Figure 3, the x-axis represents the commits, and y-axis represents the coverage achieved by each criterion. The plot is clickable and redirects users to the selected build (see Figure 4). The commit and push steps are executed after the end of the CTG phase, and all test cases under the directory `.evosuite/best-tests` (which just keeps the best generated test suites so far, as explained in the previous section) will be committed and pushed.

On the project dashboard, users also have access to a button called "EvoSuite Project Statistics" where the overall coverage, the coverage per criterion, and the time spent on test generations is reported (see Figure 5). Similar, on the build and module pages (and in addition to coverage values) is also reported the number of test cases generated. On the class page (see Figure 6) users could also view the source-code of the generated test suite.

V. Lessons Learnt

While developing the plugins for Maven, IntelliJ IDEA and Jenkins, we learned several important lessons, which we discuss in this section.

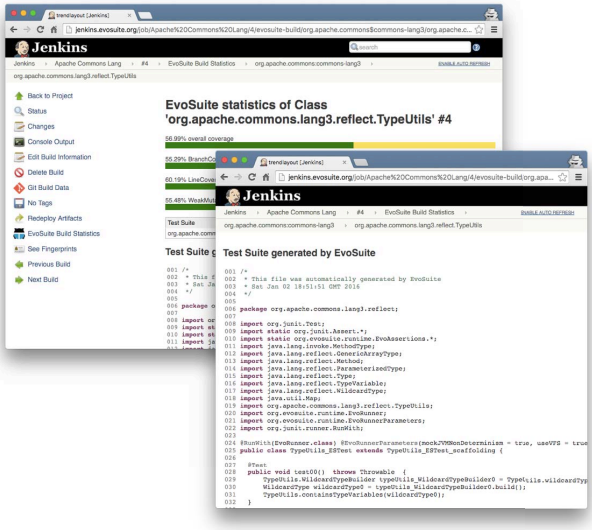

Fig. 6. EvoSuite statistics of a class and the source code of the generated test suite.

A. Industry Collaboration

Applying test data generation techniques in industry showed us[9] new problems and contexts we did not think about before. An example is the mixed execution of already existing and newly created tests, as discussed in Section II. We also came to know new tools which we did not hear about before, like for example IntelliJ IDEA and Gradle. In industry there is a large set of commonly used tools, which shape the software development processes. Technology transfer from academic research to industry practice has to take those tools into account, and how a new research technique could be integrated with them. This can only be achieved with close collaboration with industry, and by applying techniques resulting from research on real systems.

B. Lightweight Plugins

Developing a plugin is usually a very time consuming and tedious task — not necessarily because of specific technical challenges, but rather due to a systematic lack of documentation. Most tools we analysed provide some tutorials on how to write plugins, but these are very basic. API documentation in form of JavaDocs is usually very scarce, if it exists at all. For example, at the time of writing this paper, IntelliJ IDEA does not even have browsable JavaDoc documentation. The "recommended" way to learn how to develop plugins for IntelliJ IDEA is to check out its source code, and also to study other already existing open-source plugins for it. The same happened during the development of the Jenkins plugin: Although there are more than 1300 Jenkins plugins (at the time of writing this paper)

[9]One of the authors of this paper is a software consultant working in industry. Many of the architectural choices were based on feedback throughout the years on applying EvoSuite in gas & oil exploration companies like WesternGeco, and telecoms like Telenor. Disclaimer: This paper only expresses the personal opinions of the authors, and is not an official statement made by these companies.

and the documentation to setup the IDE (Eclipse or IntelliJ IDEA) to develop and build a Jenkins plugin is very complete, the documentation of, for example, how to keep data from one build to another is very poor. To our surprise, Jenkins does not read several files to build all the web interface every time a page is loaded. Instead, it serializes all data of a build after finishing it. This is of course a feature that speed up Jenkins, but it took us a while to understand it and properly use it, due to the lack of documentation.

Often, adding even some very basic functionalities requires hours if not days of first studying the source code of those tools, or asking questions on their developers' forums (in this regard, IntelliJ's forum was very useful). To complicate matters even more, the APIs of these tools are not really meant for maintainability (e.g., backward compatibility to previous versions, as usually done for widely used libraries), and can drastically change from release to release.

The lesson here is that plugins should be as lightweight as possible, where most of the functionalities should rather be in the test data generation tools. A plugin should be just used to start the test data generation with some parameters, and provide feedback on when the generation is finished, or issue warnings in case of errors.

Another lesson learnt is that, at least in our cases, it pays off to run the test data generation tools in a separated JVM. This is not only for Java version mismatch issues (recall Section III), but also for other technical details. The first is related to the handling of classloaders: EVOSUITE heavily relies on classloaders, for example to load and instrument CUTs, and also to infer the classpath of the JVM that started EVOSUITE automatically (this is needed when EVOSUITE spawns client processes). When run from command line, the classloader used to load EVOSUITE's entry point would be the system classloader, which usually is an instance of `URLClassLoader`. A `URLClassLoader` can be queried to obtain the classpath of the JVM (e.g., to find out which version of Java was used, and its URL on the local file system). However, this is practically never the case in plugins, where classes are usually loaded with custom classloaders. If a tool relies on the system classloader, then running it inside a plugin will simply fail (as it was in our case with EVOSUITE).

Furthermore, there are more subtle corner cases we encountered: During a demonstration of EVOSUITE with the Eclipse plugin, we decided to switch off the wifi connection just a minute before the demo started, in order to avoid possible annoying popups, like for example an incoming Skype call. Unfortunately, to the amusement of the audience, this had the side effect of making the EVOSUITE Eclipse plugin not working any more, although running EVOSUITE from command line was perfectly fine. Following debugging investigations led to us to the culprit: the *localhost* host name resolution. EVOSUITE uses RMI to control its spawn client processes. This implies opening a registry TCP port on the local host, which resulted in the IP address of the wifi network card. This mapping was cached in the JVM when Eclipse started. Switching off the wifi did not update the cache, and then EVOSUITE, which was running in the same JVM of Eclipse, was using this no longer valid IP address. This problem would not have happened if EVOSUITE was started in its own JVM. (Note, however, that a simple fix to this issue was to hardcode the address `127.0.0.1` instead of leaving the default resolution of the *localhost* variable).

Another benefit of running a test data generation tool on a separate process is revealed when there are problems, like a crash or hanging due to an infinite loop or deadlock. If such problems happen in a spawned process, then that will not have any major side effects on the IDE, and the software engineers will not need to restart it to continue coding. As generating tests is a time consuming activity (minutes or even hours, depending on the number of CUTs), a couple of seconds of overhead due to a new JVM launch should be negligible.

C. Compile Once, Test Everywhere

Java is a very portable language. Thanks to Java, we have been able to apply EVOSUITE and its plugins on all major operating systems, including Mac OS X, Linux, Solaris and Windows. However, this was not straightforward.

Among academics, Mac and Linux systems are very common. This latter is particularly the case because clusters of Linux computers are often used for research experiments. However, in industry Windows systems are not uncommon, and when applying EVOSUITE it turned out that initially our plugins did not work for that operating system.

A common issue is the handling of file paths, where Mac/Linux uses "/" as path delimiter, whereas Windows uses "\". However, this issue is simple to fix in Java by simply using the constant `File.separator` when creating path variables. Another minor issues is the visualisation of the GUI: for example, we noticed some small differences between Mac and Windows in the IntelliJ plugin pop-up dialogs. To resolve this problem one needs to open the plugin on both operating systems, and perform layout modifications until the pop-up dialogs are satisfactory in both systems.

However, there were also some more complex problems. In particular, Windows has limitations when it comes to start new processes: Process cannot take large inputs as parameter (e.g., typically max 8191 characters). In test data generation, large inputs are common, for example to specify the full classpath of the CUT, and the lists of CUTs to test. A workaround is to write such data on disk, and use the name and path of this file as input to the process; the process will then read from this file and apply its configurations. However, this approach does not work for the classpath, as that is an input to the JVM process, and not the Java program the JVM is running. Fortunately, this is a problem faced by all Java developers working on Windows, and there are many forums/blogs discussing workarounds. The solution we chose in EVOSUITE is that, when we need to spawn a process using a classpath C, we rather create a "pathing jar" on the fly. A pathing jar is a jar file with no data but a manifest configuration file, where the property `Class-Path` is set to C (after properly escaping it). Then, instead of using C as classpath when spawning a new process, the classpath will just point to the generated pathing jar.

Another major issues we faced when running EVOSUITE on Windows is the termination of spawned processes, although this might simply be a limitation of the JVM: Commands like `Process.destroy` (to kill a spawned process) and `System.exit` (to terminate the execution of the current process) do not work reliably on Windows, resulting in processes that are kept on running indefinitely. This is challenging to debug, but fortunately, as it affects all Java programmers working on Windows, there are plenty of forums/blogs discussing it. In particular, in Windows one has to make sure that all streams (*in*, *out* and *err*) between a parent and a spawned process are closed before attempting a `destroy` or a `exit` call.

To be on the safe side and to avoid the possibility of EVOSUITE leaving orphan processes, the entry point of EVOSUITE (e.g., IntelliJ or Maven plugins) starts a TCP server, and gives its port number as input to all the spawned processes. Each spawned process will connect to the entry point, and check if the connection is on every few seconds. If the connection goes down for any reason, then the spawned process will terminate itself. This approach ensures that, when a user stops EVOSUITE, no spawned process can be left hanging, as the TCP server in the entry point will not exist any more. The benefit of this approach is that it is operating system agnostic, as it does not rely on any adhoc operating-sytem specific method to make sure that no spawned process is left hanging.

VI. CONCLUSIONS

In this paper, we presented three plugins we developed for EVOSUITE to make it usable from Maven, IntelliJ IDEA and Jenkins. This was done in order to improve the integration of EVOSUITE into the development process for large industrial software projects. We discussed the motivations for our architectural choices, based on our experience in starting to apply EVOSUITE among our industrial partners, and presented technical details and lessons learnt.

The architecture of our plugins is not specific to EVOSUITE, and could in principle be reused for other test data generation tools as well, like for example Randoop [14], jTExpert [15], GRT [16] and T3i [17]. However, to this end we would need to formalize the names of the input parameters (e.g., how to specify the classes to test and how many cores could be used at most) that are passed to those tools, and they would then need to be updated to use this information.

Further future work is required to support also other IDEs and build tools. For example, we will update our current Eclipse plugin to have the same architecture as our IntelliJ plugin. Furthermore, as Gradle is gaining momentum in industry, we are planning to support it as well. In this paper we presented the first prototype version of the EVOSUITE Jenkins plugin, and although it is usable, there is much potential for additional functionalities: For example, although coverage of existing tests is measured, this is not yet used in coverage visualizations. In particular, it would be helpful to see in detail which parts of all CUTs are covered by existing tests, which parts are covered by newly generated tests, and which parts are not yet covered at all. Furthermore, support to other SCMs would be beneficial.

EVOSUITE and its plugins are freely available for download. Their source code is released under the LGPL open-source license, and it is hosted on GitHub. For more information, visit our webpage at: *www.evosuite.org*.

Acknowledgments. Supported by the National Research Fund, Luxembourg (FNR/P10/03) and the EPSRC project "EXOGEN" (EP/K030353/1).

REFERENCES

[1] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-oriented Software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, 2011, pp. 416–419.

[2] ——, "EvoSuite: On the Challenges of Test Case Generation in the Real World," in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ser. ICST '13, 2013, pp. 362–369.

[3] ——, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[4] ——, "1600 Faults in 100 Projects: Automatically Finding Faults While Achieving High Coverage with EvoSuite," *Empirical Software Engineering*, pp. 1–29, 2013.

[5] ——, "A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, pp. 8:1–8:42, Dec. 2014.

[6] J. M. Rojas, G. Fraser, and A. Arcuri, "Automated Unit Test Generation During Software Development: A Controlled Experiment and Think-aloud Observations," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, 2015, pp. 338–349.

[7] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 4, pp. 23:1–23:49, Sep. 2015.

[8] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella, "Do Automatically Generated Test Cases Make Debugging Easier? An Experimental Assessment of Debugging Effectiveness and Efficiency," *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 1, pp. 5:1–5:38, Dec. 2015.

[9] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated Unit Test Generation for Classes with Environment Dependencies," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, 2014, pp. 79–90.

[10] ——, "Generating TCP/UDP Network Data for Automated Unit Test Generation," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 155–165.

[11] J. Campos, A. Arcuri, G. Fraser, and R. Abreu, "Continuous Test Generation: Enhancing Continuous Integration with Automated Test Generation," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, pp. 55–66.

[12] J. Campos, G. Fraser, A. Arcuri, and R. Abreu, "Continuous Test Generation on Guava," in *Search-Based Software Engineering*, ser. Lecture Notes in Computer Science, M. Barros and Y. Labiche, Eds., 2015, vol. 9275, pp. 228–234.

[13] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, Jul. 2000.

[14] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed Random Testing for Java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07, 2007, pp. 815–816.

[15] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc, "Instance Generator and Problem Representation to Improve Object Oriented Code Coverage," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, 2015.

[16] L. Ma, C. Artho, C. Zhang, H. Sato, M. Hagiya, Y. Tanabe, and M. Yamamoto, "GRT at the SBST 2015 Tool Competition," in *Proceedings of the Eighth International Workshop on Search-Based Software Testing*, ser. SBST '15, 2015, pp. 48–51.

[17] I. S. W. B. Prasetya, "T3i: A Tool for Generating and Querying Test Suites for Java," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015.