Software testing techniques comparison: Effectiveness and scalability

YU MI, HONGLU XU

1 INTRODUCTION

Software testing is acknowledged to be a critical part in software development processes both in industry and in academia. In most situations, a large amount of time and effort will be spent on the software testing process. According to NIST[15], inadequate software testing infrastructure is responsible for $59.5 billion in annual costs. In Fazlullah's work[10], the author stated that forty to fifty percent of time and budget is suggested for testing in the development process, since many features like the reliability, maintainability, security and others need to be guaranteed for a software system.

With the increasing demand for software testing, large number of software testing tools, such as Klee, Randoop[12], EvoSuite[7], continue to spring up, and many software testing techniques, like Black Box Testing, White Box Testing and Gray Box Testing have been widely used in these testing tools. In this case, the problems we are interested in are: what is a good method that we can use to select the best software testing tool for a particular project? Which can be restated as: If we have software that needs to be tested, how can we compare all the software testing tools we found and select out the best one for this project?

In this project, we intend to compare software testing tools with our focus on comparing their effectiveness for revealing faults and their scalability. Our aim is to compare some testing tools and help software developers to select one or multiple performant and compatible software testing tools for testing. There are many methods and techniques needed to achieve this goal, and we will discuss these features in our paper.

2 RELATED WORKS

The related works can be separated into two types: comparing framework focus and testing technique focus. They are discussed separately below:

2.1 Comparing Frameworks

There has been several research papers focused on comparing efficiency, effectiveness and applicability of software testing techniques.

Fraser and Staats[8] examined whether automated unit test generation techniques can really help software testers, which is a cornerstone of our study. In their work, the authors focused on examining an automated testing tool called EvoSuite and aimed at checking how the tool impact structural code coverage, ability to detect faults, ability to detect regression faults and the number of tests that mismatch the intended behavior of the tested class. As a result, the authors found that automated test generation can be effective at producing tests suites with high code coverage but cannot confirm whether such tools can reveal faults better than manual testers.

Sharma[17] did a study of several Object-oriented software testing tools and made several contributions about comparing testing techniques. In their work, the author selected three tools, EvoSuite, JUnit Test Case Builder and CodePro AnalytiX to perform the comparison. They mainly focus on the tools' feature, usability, debugging, requirement and performance metrics and give

a throughout analysis about these tools. Their work inspired our research and can be used as a reference in designing our study procedures.

Eldth and Hasson[4] have developed a framework of comparing testing techniques where they mainly deal with fault detection and did more than 70 distinct experiments. In their work, the authors prepared code samples with injected faults and did experiments with a test technique on the code samples. After collecting data of experiments, they analyze and evaluate results. After doing experiments on multiple testing techniques, the authors also compared the results with each other to figure out which technique is better in the aspect of efficiency, effectiveness and applicability. Another contribution worth mentioning is, the authors also examined the fault propagation in the sample code. Faults in code can propagate at different levels during testing process, and the authors also count the ability to discover code faults earlier as a part of efficiency.

A. Budd and A. Demillo[2] discussed about mutation test in testing functional correctness of programs. In their work, they build a framework for studying the program mutation testing method from both theoretical and empirical viewpoints. The authors stated that there is very real trade-off in the two types of research mentioned above. Although their results have dealt solely with program testing, they found that the the potential for developing other software methodologies that try to exploit some fact of programming process should be overlooked.

2.2 Testing Techniques

There are also some research papers on examining the testing techniques, which could provide more reference on our analysis of a specified testing tool.

Rojas and Vivanti[16] focused on the effectiveness of whole test suite generation and selected EvoSuite as their test generating tool. In their work, the authors found that in search-based testing, the testing problem is cast as a search problem, but have issues like: if a coverage goal is infeasible, all the search effort to try to cover codes would be wasted, and the search for coverage is independent, thus useful information could not be shared between individual researches. At this point, the authors investigated whether there are specific coverage goals for witch the traditional approach is better. Based on an empirical study performed on 100 Java classes, their study showed that indeed there are cases in which the traditional approach provides better results.

Kifetew and Tonella[14] stated that test case generation is intrinsically a multi-object problem, because the goal of test case generation is covering multiple test targets. Existing approaches either consider one target at a time or aggregate al targets into a single fitness function. However, Multi and many-objective optimisation algorithms (MOAs) have never been applied to this problem. In their work, the author presented DynaMOSA(Dynamic Many-Objective Sorting Algorithm) to address the test case generation problem in the context of coverage testing. They also carried out an empirical study on 346 Java classes using three coverage criteria and showed that DynaMOSA out performs whole suit approach in all these criteria.

Almasi and Hemmati[1] made an industrial evaluation of unit test generation in finding real faults in a financial application. The author stated that automated unit test generation has been extensively studied in recent years but people still have no idea how effective and applicable are the test generating tools in an industrial application. In their work, the authors investigated this problem using a life insurance and pension product calculator engine owned by SEB Life & Pension Hoding AB Riga Branch. TO study fault-finding effectiveness, the author extracted 25 real faults from the version history of the software project and applied EvoSuite and Randoop to generate test cases. As a result, they found EvoSuite can detect up to 56.4% of these faults and Randoop for 38.0%.

3 METHODOLOGY

The tools we are going to examine in our research is EvoSuite, Randoop and CodePro AnalytiX[9], we will discuss these tools as below and introduce our method of running test cases as below:

3.1 EvoSuite

EvoSuite is an automatic test suite generation tool for Object-Oriented software. It was proposed by Fraser and Arcuri. In their paper[7], the authors pointed out that faults, in many cases, do not lead to program crashes and deadlocks. Therefore, there is the need to provide small test suites to the user that can be manually verified. Thus, they developed EvoSuite which could perform whole test suite generation and mutation-based assertion generation. The authors stated that EvoSuite implements whole test suite generation as a search based approach, which could improve average branch coverage significantly. On the other hand, EvoSuite now includes a tool called μTest which deals with mutation testing. In mutation testing, artificial defects are seeded into a program and test cases are evaluated with respect to how many of these seeded defects they can distinguish from the original program. In the end, the authors also introduces the implementation of EvoSuite. The EvoSuite requires only the Java bytecode of the class under test and its dependencies. The bytecode is analyzed and instrumented, and EvoSuite produces a JUnit test suite for a given class.

EvoSuite is a Search based testing tool(SBST) which uses genetic algorithm[3] to generate test suites[6]. As is shown in Fig. 1, Evosuite will develop entire test suites targeting all coverage goals at the same time. After generating all the test suites, Evosuite uses genetic algorithm to mutate the test suites and search for the minimized test suite with maximized coverage.

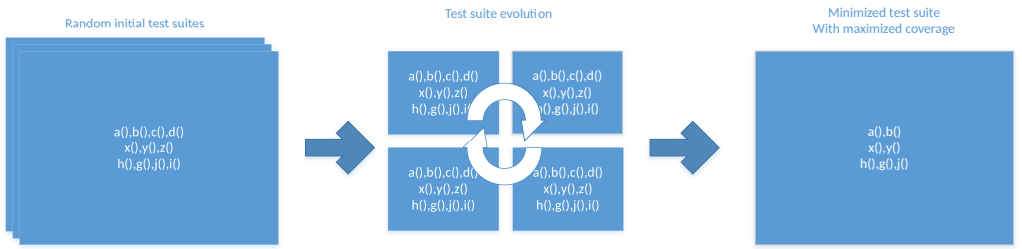

Fig. 1. Evosuite process

To generate test cases, we need to execute the following command:

```
java -jar evosuite-1.0.5.jar -class <classname> -projectCP <classpath>
```

Here the 'evosuite-1.0.5.jar' is the executable file obtained from EvoSuite website, '<classname>' indicates the name of class being tested, and '<classpath>' means the directory of the classes the project being tested is in.

3.2 Randoop

Randoop is a feedback-directed random testing tool for Java. It was proposed by Pacheco and Ernst. In their paper[12], the authors introduced that Randoop could output two test suites. One contains contract-violating tests that exhibit scenarios where the code under test leads to the violation of an API contract. The second suit that Randoop outputs contains regression tests, which do not violate contracts but instead capture an aspect of current implementation. The authors stated that

Randoop generates unit tests using feedback-directed random testing, which is inspired by random testing that uses execution feedback gathered from executing test inputs as they are created.

As is shown in Fig. 2, Randoop takes as input a set of classes under test, a time limit and optionally, a set of "contract checkers" which extend those used by Randoop as default. Randoop outputs two suites, which is discussed above. The feedback-directed random testing is a technique inspired by random testing that uses execution feedback gathered from executing text inputs as they are created, to avoid generating redundant and illegal inputs[13]. Randoop creates method sequences incrementally, by randomly selecting a method call to apply and selecting arguments from previously constructed sequences. When the method sequence is created, a new sequence is executed and checked against a set of contracts. Sequences that lead to contract violations are output to the user as contract-violating tests. Sequences that exhibit normal behavior are output as regression tests, only normally-behaving sequences are used to generate new sequences.

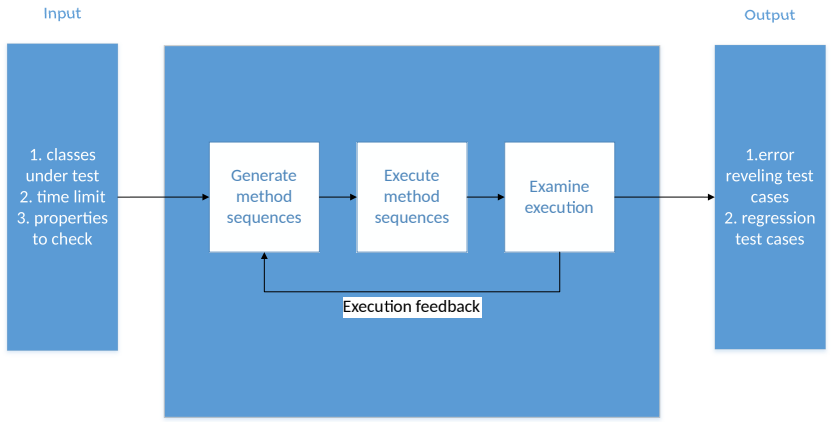

Fig. 2. Randoop process

To generate test cases, we need to execute the following command:

```
java -ea -classpath <myclasspath>:$(RANDOOP_JAR) randoop.main.Main gentests --testclass=<
    classname> --timelimit=<timelimit>
```

Here the 'myclasspath' is the path of class being tested and '$(RANDOOP_JAR)' is a environment variable which contains the path of Randoop executable files. '<classname>' means the name of class being tested and '<timelimit>' is the time limit for the generation process, using the unit of seconds.

3.3 CodePro AnalytiX

CodePro AnalytiX[9] is a tool developed by Instantiations which lacks of public documentations. Here we view it as a commercial test generating tool and compare it with other tools. According to the work of Sharma[17], CodePro AnalytiX does JUnit test case generation through a blend of both static code analysis and by dynamically implementing the code to be tested in order to detect behavior of code. Test cases will be generated as a separate project.

Typically, CodePro AnalytiX works as a Eclipse plugin. To generate test case using CodePro AnalytiX, we need to right-click on the class file in Eclipse IDE, select 'CodePro Tools' and click on 'Generate Test Cases' command as is shown in Fig. 3.

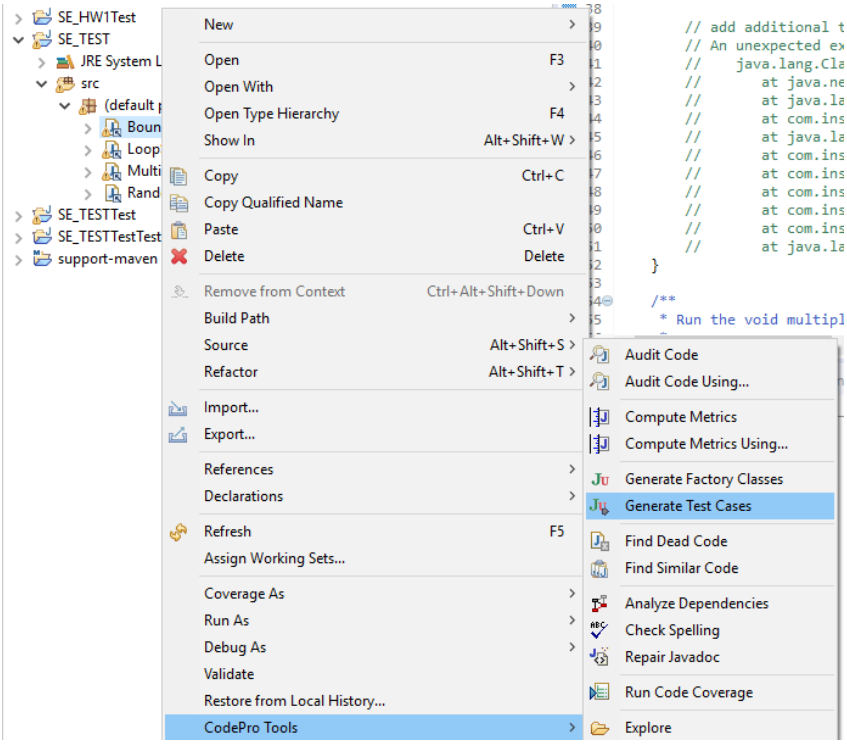

Fig. 3. CodePro Generate Test Cases

3.4 To run Test cases

To run test cases, we need to select the test cases file generated by the test generation tools mentioned above. Right click the test file and select 'Run As', click the 'JUnit Test' in the menu. This process is shown as Fig.4.

4 EXPERIMENTS

Our main goal is to compare these three testing tools to figure out which tool has a better efficiency and scalability in generating test codes. There are many typical properties to evaluate a testing tool, such as coverage, amount of test cases generated, test case generation speed and time, fault revealing and failure detection effectiveness. In our work, we will focus on the two most important properties for the experiment: coverages and fault revealing effectiveness.

4.1 Coverage

4.1.1 Definition of coverage. Coverage percentage is a representative measurement to evaluate a testing tool. Beside the line coverage, there are actually a lot of other types of coverage that will provide some useful information to the developers. Measurement like branch coverage, exception coverage and method coverage are very important information for the developers. By inspecting on these coverage, developer will acquire the concept of reliabilities of these test cases that generated by the testing tools. For example, if the method coverage is low for a particular test generator, the developer probably need to implement their own test code with high method coverages so that

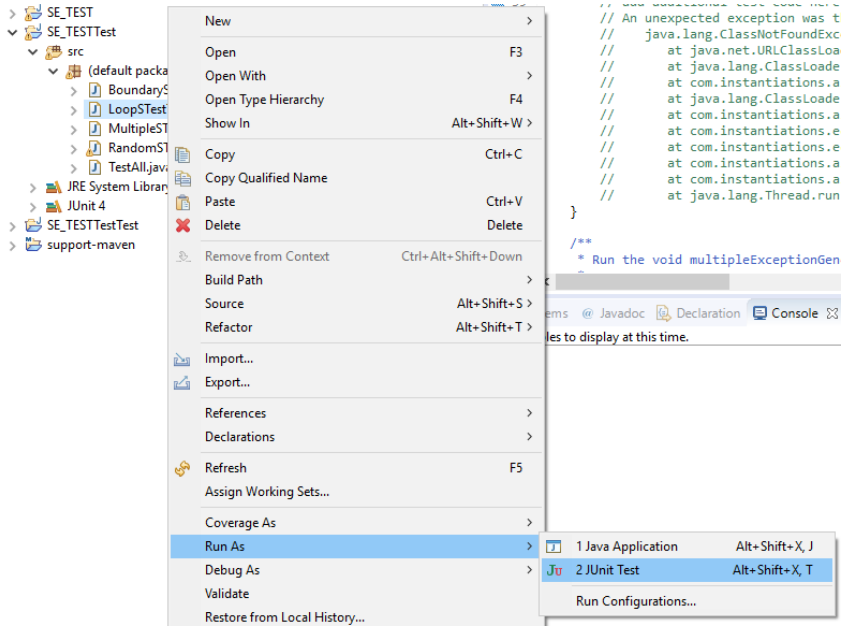

Fig. 4. To run Test Cases

the chance that some critical failures have been ignored in these test cases will be lowered. In this condition, Evosuite has the advantage of displaying multiple types of coverage for the developer.

```
* Going to analyze the coverage criteria
* Coverage analysis for criterion LINE
* Coverage of criterion LINE: 100% (no goals)
* Coverage analysis for criterion BRANCH
* Coverage of criterion BRANCH: 100% (no goals)
* Coverage analysis for criterion EXCEPTION
* Coverage of criterion EXCEPTION: 100% (no goals)
* Coverage analysis for criterion WEAKMUTATION
* Coverage of criterion WEAKMUTATION: 100% (no goals)
* Coverage analysis for criterion OUTPUT
* Coverage of criterion OUTPUT: 100% (no goals)
* Coverage analysis for criterion METHOD
* Coverage of criterion METHOD: 0%
* Total number of goals: 20
* Number of covered goals: 0
* Coverage analysis for criterion METHODNOEXCEPTION
* Coverage of criterion METHODNOEXCEPTION: 0%
* Total number of goals: 20
* Number of covered goals: 0
* Coverage analysis for criterion CBRANCH
* Coverage of criterion CBRANCH: 100% (no goals)
* Generated 0 tests with total length 0
* Resulting test suite's coverage: 0% (average coverage for all fitness functions)
* Generating assertions
* Resulting test suite's mutation score: 100%
* Compiling and checking tests
* Writing JUnit test case 'CounterLocalService_ESTest' to evosuite-tests
* Done!
```

Fig. 5. Sample output of Evosuite

The Fig.5 describes a sample output of the Evosuite test generation results in the command line. We can conclude that there are lots of other coverages measured beside the line coverage provided

in the result text. Whats more, there are 'number of goals' and 'number of covered goal' shown in the result, which provides a straightforward way to inspect and re-check the reliabilities of these test cases. However, Randoop and CodePro will not provide any coverage information in their result. Instead, they will leave this task to the compiler and the IDE tools like Eclipse and Intellij IDEA. Thus, Evosuite has a better way to provide the coverages information to the developer.

4.1.2 Line Coverage. Line coverage is the basis and the most important properties to examine a JUnit test case. Despite measured by the testing tools itself (like Evosuite), or measured by the development tools. There will always be line coverages presented for the test cases that generated by the test generator. Thus, it is considered as the main property for us to measure the effectiveness of test generation among these three testing tools in our experiment. In order to make a fair environment for each testing tool, we will use the line coverages that displayed on the IDEs as their coverage, and more specifically, we will use the Intellij IDEA and Eclipse in our experiment.

Fig. 6. Covered lines and uncovered lines in IDE

Fig.6 shows the covered lines and uncovered lines from a sample test case coverage result run by Eclipse. While running a JUnit test case in an IDE. The IDE will try to run all the test case instructions, where it will call the methods from the source code, and try to see whether all the source code instructions have been called. If an instruction is called by the test cases, the IDE will mark this instruction as 'covered'. On the other hand, if an instruction never been called by the test cases, this line will be marked as 'uncovered'. Thus, the line coverage will be defined as Equation 1:

$$LineCoverage = \frac{Numberof CoveredLine}{TotalNumberof Instructions} \times 100\% \tag{1}$$

This value can easily describe what is the percentage of instructions or lines that have been run through during the tests. Thus, it is obvious that a JUnit test case with greater percentage of line coverages will be better compared to a test case that shows less coverages in the result.

4.1.3 Results. We applied the measurement method described above to all the three test generation tools. All of them will generate multiple test files, which contains many JUnit test cases. After running these test cases with the coverage instruction, these test cases will display line coverage as long as the marks that which line is covered. By comparing these information, we can try to find the advantages and disadvantages of these testing techniques that used by these test generation tools.

For getting a decent amount of results, we need to have a lot of sample source code for experiment. In that case, we chose a project that listed in the Evosuite test results lists[5]: 'liferay-6.0.6-master' [11]. This project contains nearly 7000 classes, so it should be a decent test sample for our experiment.

TARGET_CLASS	Evosuite Line Coverage	Codepro Line Coverage	Randoop Line Coverage
com.liferay.counter.service.CounterLocalServiceWrapper	1	0.82	0.82
com.liferay.counter.service.CounterLocalServiceUtil	0.84	0.84	0.78
com.liferay.counter.model.CounterWrapper	1	0.96	0.12
com.liferay.counter.service.persistence.CounterFinderUtil	0.75	0.81	0.75
com.liferay.portal.kernel.annotation.Isolation	1	None	1
com.liferay.portal.kernel.annotation.Propagation	1	None	1
com.liferay.portal.kernel.audit.AuditMessage	0.24	0.65	0.2
com.liferay.portal.kernel.audit.AuditMessageFactoryUtil	1	1	1
com.liferay.portal.kernel.audit.AuditRequestThreadLocal	1	1	1
com.liferay.portal.kernel.audit.AuditRouterUtil	0.83	0.66	0.83
com.liferay.portal.kernel.bean.ClassLoaderBeanHandler	0.86	1	0.73
com.liferay.portal.kernel.bean.PortletBeanLocatorUtil	0.47	0.431	0.73
com.liferay.portal.kernel.bean.ReadOnlyBeanHandler	0.55	0.88	0.55
com.liferay.portal.kernel.bi.reporting.ByteArrayReportResultContainer	1	1	0.95
com.liferay.portal.kernel.bi.reporting.ContextClassloaderReportDesignRetriever	1	1	1

Fig. 7. Sample result for coverages

Fig.7 shows a sample result for our experiment. From left to right (except for the first column), each column shows the line coverages that got from the compiler for each target class and for each testing tool. From this sample, lots of useful information can be extracted. We can conclude that although most of the line coverages for a particular class is different to each other among all three testing tools, it still appears to achieve the same coverage among the methods in some classes, such as 'com.liferay.portal.kernel.audit.AuditMessageFactoryUtil' and 'com.liferay.portal.kernel.bi.reporting.ContextClassliaderReportDesignRetriever'. All the test cases that generated by these three test generation tools seems getting a similar coverage to each other for these two sample classes. It is a common sense that a good test generation tool should generate the test cases that could get a nearly 100 percent score for a simple source class, which contains few conditions and branches, and doesnt have many method recalls. That is the reason that for many classes in our sample, all the three test generation tools behave good.

However, we also see a lot of divergence in the Fig.7. There are a great number of reasons that why the line coverage is different, and it is our goal to analyze and conclude the situation that using one or two test generation tools are better than using others.

The first method we used is analyze the JUnit test cases that generated by the test generators. We can see from the result table that there are two sample classes which have no coverage from the CodePro test cases. It is because there are some errors in the test code generated by CodePro, and the test file failed during running. Fig.8 shows the bug that caused the failure in CodePro test cases and Fig.9 shows the way that Evosuite implemented for this class. After analyzing the source code and other test case samples, we conclude that CodePro is not behave very well in the situation of enum and 'serialized' classes. In most of these situations, it will generate false code or generate the test cases with low coverage. In the meantime, Randoop and Evosuite behave normal in these situations.

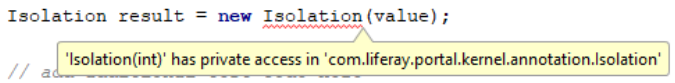

Fig. 8. CodePro generated test case

```
Isolation isolation0 = Isolation.valueOf("SERIALIZABLE");
assertEquals(Isolation.SERIALIZABLE, isolation0);
```

Fig. 9. EvoSuite generated test case

Another thing we noticed that is Randoops test cases has the low coverage for class 'com.liferay.counter.model.CounterWrapper', while other two tools behave normally. Further investigation shows that Randoop didnt generate the normal test cases for this class, instead, an error test file is generated only for the purpose of detecting a particular fault. It is because during the Randoop generation progress, it encountered a java exception 'java.lang.NullPointerException', and instead of ignoring the fault and keep going, it aborted and focused on this fault. This technique is good for fault detection, which we will discuss in the following section, but not very effective to get the line coverage information. Thus, in the situation of exceptions, Randoop behave worse on the coverage property.

Another method we used is to analyze the source code with the coverage marks, so we could have the idea what kind of code will be covered, what will not. In the result of class 'com.liferay.portal.kernel.bean.ClassLoaderBeanHandler', the CodePros test cases give a great score, which indicates that the source class should be a simple class. However, both Randoop and Evosuites behaviors are not really good for a simple source class. After doing some investigations with the cover marks and the source code, we found a particular place that cause the problem.

```
        }
        catch (InvocationTargetException ite) {
            throw ite.getTargetException();
        }
```

Fig. 10. Try and catch operation

Fig.10 shows the place in the source code that Randoop and Evosuites test cases did not cover. It is a typical try and catch operation. Randoop and Evosuite didnt generate the situation that this particular exception occurs, but CodePro implemented this situation well. We also analyzed many other similar situations, most of them appear CodePro could handle the try and catch situation better.

4.2 Fault detection

Another main purpose that the develop will use the JUnit test cases is to detect fault, whether caused by type error, logic false or other reasons. A good test generation tool should generate the test case that could reveal most of the common exceptions from the source code, and, multiple exceptions should be revealed in a single cast if there exist that many in one source file. Since we need to orient the specific exceptions for testing, the sample code from other 'liferay' will not be good experiment data for us. Thus, we tried to implement our own sample code for testing.

4.2.1 Exception / Fault Definition. For these three testing techniques, different method will be used to display revealed faults/exceptions. Both Evosuite and Randoop will include the revealed exceptions in the test cases they generated.

```
@Test(timeout = 4000)
public void test0()  throws Throwable  {
    Random.setNextRandom((-433));
    BoundarySTest boundarySTest0 = new BoundarySTest();
    // Undeclared exception!
    try {
      boundarySTest0.outBoundaryGenertor();
      fail("Expecting exception: ClassCastException");

    } catch(ClassCastException e) {
       //
       // java.lang.Integer cannot be cast to java.lang.String
       //
       verifyException("com.selftest.BoundarySTest", e);
    }
}
```

Fig. 11. Evosuite revealed exception

```
@Test
public void test1() throws Throwable {
    if (debug)
        System.out.format("%n%s%n", "RegressionTest0.test1");
    BoundarySTest boundarySTest0 = new BoundarySTest();
    try {
        boundarySTest0.outBoundaryGenertor();
        org.junit.Assert.fail("Expected exception of type java.lang.ArrayIndexOutOfBoundsException");
    } catch (ArrayIndexOutOfBoundsException e) {
    }
}
```

Fig. 12. Randoop revealed exception

Fig.11 and Fig.12 shows the test code that generated by Evosuite and Randoop. We could see that certain exceptions have been detected by the testing technique, and in order to test it, both of them used some verification methods to make sure this exception does occur while operating the test cases. This is actually an easy and clear way to tell the developer what exception occurred and in which method it occurred. On the other hand, CodePro used a different way to display the exceptions.

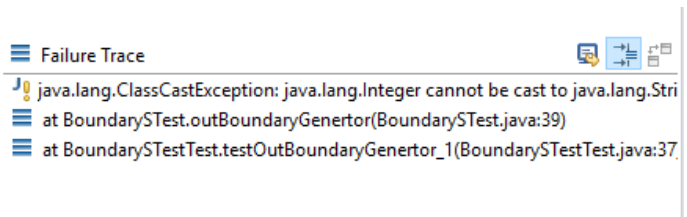

Fig. 13. CodePro revealed exception

Fig.13 shows the particular display method of CodePro for revealing faults/exceptions. Since CodePro is a plugin of Eclipse, it works well with Eclipse, and with the function 'failure trace', it can show the developers exceptions occurred during the test. Whats more, when cooperating with the compiler, it can show the particular lines that caused the exception, which gives a clearer way to show the detailed information to the developer.

Thus, for presenting the information, CodePro has the advantage of showing the related source code lines. However, since Evosuite and Randoop provided the frame to handle the exceptions, it is easier for the developers to add their own test code to the place where the exceptions occur.

4.2.2 Fault Detection Effectiveness. We classified the exceptions and faults that will occur in normal situations into several sub-problems.

Common Exceptions. 'Out of Boundary Exception' is one of the most common exceptions that will occur during the development. It may cause by logic and type errors, and in many dynamic coding situations, this exception may occur randomly during the runtime. To simulate this situation, we also used the random number to be as a condition for the exception occur. For this sample source code, both Randoop and Evosuite will use a lot of test iterations to see whether exceptions occurred at a least once in these test iterations, if so, they will generate this kind of exception in the test cases and assert it while running test cases. On the other hand, CodePro will generate multiple test cases and try to use the test cases to reveal the exceptions in the runtime. Since the test cases generation is limited (max 30 test cases by default for CodePro), the chance to reveal the exception for a random situation for CodePro is much lower than Evosuite and Randoop. Thus, for common exceptions revealing, Randoop and Evosuite can do better.

Logical Errors. 'Infinite loop' is a very typical logical error. It also can occur randomly for a dynamic situation. However, no exceptions and errors will be thrown by the compiler if an infinite loop happens, so this makes it harder to detect this situation. Randoop will not generate any test cases for an infinite loop situation. It probably because Randoop didnt finish its own detection and aborted. CodePro also falls into the infinity loop situation. It will generate normal test cases, but these test cases will still call the infinity loop method and wont finish. On the other side, Evosuite act very wisely. It will generate three test cases for three conditions that fits this situation.

Fig.14 shows a sample test code generated by Evosuite to reveal the infinity loop situation. In the source code, a random number will be generated as a condition, if that number is greater than 1000000, an infinity loop will occur, otherwise, program will exit normally. In Fig.14, we can conclude that Evosuite used its own implemented 'Random' class to generate three cases, the case that an exact number is generated that the program will not cause the infinity loop, the case that a number generated that will cause the infinity loop, and the normal case test. There are no exception verifications, but still, it provided three cases for the developer for examining and adding their own code. Thus, in this situation, Evosuite seems to be a better tool for some logical errors.

Multiple Exceptions. In many situations, there will be several faults/exceptions even in a single class, and it is desirable to have the test cases to detect all the exceptions. Randoop and CodePro seem do not have this function implemented. We have them run through our own sample code, which will use a random number to decide which one or multiple exceptions will trigger, but Randoop seems only got the first exceptions it encountered, so as CodePro. Evosuite, not surprised, take advantage of its Random technique, and generated all the test cases for these exceptions. Thus, Evosuite is better at detect multiple exceptions, too.

```java
@Test(timeout = 4000)
public void test0()  throws Throwable  {
    LoopSTest loopSTest0 = new LoopSTest();
    Random.setNextRandom(1000000);
    loopSTest0.loopGenerator();
}

@Test(timeout = 4000)
public void test1()  throws Throwable  {
    LoopSTest loopSTest0 = new LoopSTest();
    Random.setNextRandom(1000038);
    // Undeclared exception!
    loopSTest0.loopGenerator();
}

@Test(timeout = 4000)
public void test2()  throws Throwable  {
    LoopSTest loopSTest0 = new LoopSTest();
    loopSTest0.loopGenerator();
}
}
```

Fig. 14. Evosuite revealed infinite loop

5 DISCUSSION

One main flaw of our comparison lies in the lack of comparing the ability of detecting failure in the sample code. Failure in software testing means that the code we test could not meet all the requirements which may not satisfy the contract of customers. This flaw has two reasons:

First, it is hard for us to figure out the requirement of the sample code. Since we are using Liferay repository[11] as our sample code, their requirement is not specified enough for us to test the test case generating tools.

On the other hand, CodePro and Evosuite does not support the function of examining code failure. However, their test cases does have the entry of testers to modify so that testers can write codes to detect code failure manually. Such approach does not reflect the failure detecting ability of these test generating technique so that we cannot compare this criteria.

REFERENCES

[1] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Jānis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press, 263–272.

[2] Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1980. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 220–233.

[3] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[4] Sigrid Eldh, Hans Hansson, Sasikumar Punnekkat, Anders Pettersson, and Daniel Sundmark. 2006. A framework for comparing efficiency, effectiveness and applicability of software testing techniques. In *Testing: Academic and Industrial Conference-Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*. IEEE, 159–170.

[5] Evosuite.org. 2017. Result for version 1.0.5. http://www.evosuite.org/files/1_0_5/results.html/. (2017).

[6] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary generation of whole test suites. In *Quality Software (QSIC), 2011 11th International Conference on*. IEEE, 31–40.

[7] Gordon Fraser and Andrea Arcuri. 2011. 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*. ACM, 416–419.

[8] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 4 (2015), 23.

[9] Instantiations. 2008. CodePro AnalytiX Datasheet. https://wiki.eclipse.org/images/7/75/CodeProDatasheet.pdf/. (2008).

[10] Syed Roohullah Jan, Syed TauhidUllah Shah, Zia Ullah Johar, Yasin Shah, and Fazlullah Khan. 2016. An Innovative Approach to Investigate Various Software Testing Techniques and Strategies. *International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET), Print ISSN* (2016), 2395–1990.

[11] liferay. 2017. Liferay repository. https://github.com/liferay/. (2017).

[12] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 815–816.

[13] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 75–84.

[14] Annibale Panichella, Fitsum Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* (2017).

[15] Strategic Planning. 2002. The economic impacts of inadequate infrastructure for software testing. (2002).

[16] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering* 22, 2 (2017), 852–893.

[17] Rashmi Sharma, Anju Saha, and AK Mahapatra. 2016. A Comparative Study of Object-Oriented Software Testing Tools. (2016).