Automatic Test Case Generation: What If Test Code Quality Matters?

Fabio Palomba[1], Annibale Panichella[2], Andy Zaidman[2]
Rocco Oliveto[3], Andrea De Lucia[1]
[1]University of Salerno, Italy — [2]Delft University of Technology, The Netherlands
[3]University of Molise, Italy

ABSTRACT

Test case generation tools that optimize code coverage have been extensively investigated. Recently, researchers have suggested to add other non-coverage criteria, such as memory consumption or readability, to increase the practical usefulness of generated tests. In this paper, we observe that test code quality metrics, and test cohesion and coupling in particular, are valuable candidates as additional criteria. Indeed, tests with low cohesion and/or high coupling have been shown to have a negative impact on future maintenance activities. In an exploratory investigation we show that most generated tests are indeed affected by poor test code quality. For this reason, we incorporate cohesion and coupling metrics into the main loop of search-based algorithm for test case generation. Through an empirical study we show that our approach is not only able to generate tests that are more cohesive and less coupled, but can (i) increase branch coverage up to 10% when enough time is given to the search and (ii) result in statistically shorter tests.

CCS Concepts

•**Software and its engineering** → Software testing and debugging;

Keywords

Evolutionary testing, many-objective optimization, branch coverage, test code quality

1. INTRODUCTION

Automated test case generation tools have been widely studied in literature in order to reduce the cost of software testing. Generating unit tests via automated techniques helps developers to maximize the percentage of code elements (e.g., statements) being exercised according to well-established code coverage criteria [41]. Previous studies acknowledged additional important benefits, including but not limited to triggering undeclared exceptions and failures [21], or achieving debugging effectiveness comparable to manually written tests [11, 36].

However, conventional approaches to test case generation mainly focus on code coverage as a unique goal to achieve, without taking into account other factors that can be relevant for testers. For example, Afshan *et al.* [2] highlighted that one such factor is the effort needed to manually check test data input and test results (e.g., assertions) in order to assess whether the software behaves as intended. Therefore, they have incorporated language models into the data generation process with the aim of generating natural language like input strings to improve human readability.

Recently, Daka *et al.* [14] used a post-processing technique to optimize readability by mutating generated tests leveraging a domain-specific model of unit test readability based on human judgement. Other non coverage-based criteria exploited in literature for test case generation include execution time [17, 37], memory consumption [25], test size [19, 31, 35], and ability to reveal faults [37].

In this paper, we focus on *test code quality* metrics to consider in addition to code coverage. Poorly designed tests are known to have a negative impact on test maintenance, as they are more difficult to adjust when production code changes [7, 30, 47, 33]. Automated tests first need to be maintained when they are generated, since testers need to manually validate each test case to check the assertions (oracle cost) [2, 9]. In addition, tests also need to be maintained and eventually updated according to the changes performed in the production code during later development activities. Therefore, we argue that achieving easily maintainable tests is a desirable and important goal in test case generation. The related literature provides a plethora of metrics to detect poorly designed tests, such as rules for test smells detection [22, 43, 44]. In the context of this paper, we consider two simple, yet critical quality metrics for evaluating test code maintainability, namely *test cohesion* and *test coupling*. According to Meszaros [29], maintainable tests must be as simple (cohesive) as possible, i.e., each test should not verify too much functionality at the same time to avoid *test obfuscation*. Furthermore, *test overlap* (or test coupling) should be minimized so that only few tests are affected by any future change [29], improving test readability and simplifying future maintenance activities.

For measuring test cohesion and test coupling we rely on Information Retrieval (IR) methods, similarly to previous papers for assessing the quality of production code [26, 39, 42]. Specifically, we define two novel metrics, namely *Coupling Between Test Methods* (CBTM) and *Lack of Cohesion*

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany
© 2016 ACM. 978-1-4503-4390-9/16/07...$15.00
http://dx.doi.org/10.1145/2931037.2931057

of a Test Method (LCTM) inspired by *conceptual coupling* and *conceptual cohesion*, which are two well-known metrics to assess code quality [42]. We choose IR methods since previous studies [26, 39, 42] demonstrated how textual analysis often outperforms structural metrics in its ability to describe cohesion and coupling phenomena.

To evaluate to what extent automatically generated test cases present design problems, we conducted a large scale preliminary study on the SF110 dataset [20] and using Evosuite [19] as test case generation tool. This analysis revealed that most automatically generated tests suffer from high coupling with other tests in the same test suite. Moreover, up to 28% of test cases (test methods in JUnit) suffer from low cohesion. Given the results of this exploratory analysis, we propose to incorporate our quality metrics LCTM and CBTM into the main loop of Evosuite to guide the search toward more cohesive and less coupled tests. For this, we extended the MOSA algorithm, a novel many-objective genetic algorithm recently proposed by Panichella *et al.* [35], by incorporating our quality metrics within the selection mechanism.

70 To evaluate our quality-based variant of MOSA, we conducted a second empirical study on 43 randomly sampled classes from the SF110 dataset. The results indicate that, besides improving both LCTM and CBTM scores with respect to the original MOSA and Evosuite's default search strategy (Whole Suite), our quality-based MOSA approach leads to other valuable positive effects: (i) when giving more time to the search, incorporating test code quality metrics may reduce the probability of early convergence increasing the final branch coverage; and (ii) the generated tests tend to be shorter, suggesting the possibility to complement existing post-search minimization strategies.

2. COMPUTING TEST CODE QUALITY METRICS

Designing test code for maintainability is a key principle of *Test Driven Development* (TDD) [10]. Specifically, writing highly cohesive and weakly coupled test cases ensures that each test method provides a specific responsibility, making it easily understandable and maintainable [10]. Even if in the context of automatic unit test generation the problem of maintainability could be relaxed (test cases can be regenerated when something changes in production code), it is still important to remark that automated tests need to be maintained once they are generated, since testers must manually validate each test case to check the assertions [9]. Moreover, as indicated by a Rojas *et al.* [40], automatically generated tests are not immediately used but they are refined by testers in order to (i) modify/remove generated test code, (ii) add new tests when automated tools leave uncovered branches and (iii) combine generated with manual written tests [40]. It is worth noting that in this paper we look for coupling among test methods (representing single test cases in the JUnit framework) of the same JUnit class (i.e., a test suite in JUnit). Unlike other strategies for computing coupling (e.g., looking for coupling between test and production code [7]), our goal is to measure to what extent a test method has relationships with the other test methods of the same class, in order to evaluate the quality of the decomposition of the methods in the JUnit class. To compute the quality of test methods, in this paper we define two metrics exploiting textual information, able to measure the degree of cohesion and coupling of a test method. It is worth noting that, as pointed out in previous work (e.g., [39, 42]), textual analysis can be successfully used for measuring source code quality. Moreover, textual analysis often outperforms structural metrics in its ability to describe the phenomenon [26, 39, 42]. In the following subsections, we report the process we follow to compute these metrics.

2.1 Textual Information Extraction and Processing

Starting from the set of test artifacts composing the software project under analysis, during the first step we extract the textual content characterizing each test method by selecting only the textual elements actually needed for the textual analysis process, i.e., source code identifiers and comments. The textual elements are then normalized by using a typical Information Retrieval (IR) normalization process. Thus, the terms contained in the source code are transformed by applying the following steps [8]: (i) separating composite identifiers using camel case splitting, which splits words based on underscores, capital letters and numerical digits; (ii) putting extracted words in lower case; (iii) removing special characters, programming keywords and common English stop words; and (iv) stemming words to their original roots via Porter's stemmer [38]. Finally, the normalized words are weighted using the *term frequency - inverse document frequency* (*tf-idf*) schema [8], which reduces the relevance of too generic words that are contained in most source components. Therefore, the resulting textual content is individually analyzed in order to apply cohesion and coupling metrics. To compute them, we rely on Latent Semantic Indexing (LSI) [16], namely an extension of the Vector Space Model (VSM) [8], which models code components as vectors of terms occurring in a given software system. LSI uses Singular Value Decomposition (SVD) [13] to cluster code components according to the relationships among words and among code components (co-occurrences). Then, the original vectors (code components) are projected into a reduced k space of concepts to limit the effect of textual noise. For the choice of size of the reduced space (k) we used the heuristic proposed by Kuhn *et al.* [24] that provided good results in many software engineering applications, i.e., $k = (m \times n)^{0.2}$ where m denotes the vocabulary size and n denotes the number of documents (code components in our case). Finally, the textual similarity among software components is measured as the cosine of the angle between the corresponding vectors.

2.2 Computing Test Method Coupling

To measure the degree of coupling of a test method, our conjecture is that *methods having high coupling have high textual similarity with the other methods contained in the test suite*. Following this conjecture, we compute the coupling metric by applying the average of the textual similarity between the test method under analysis and the other test methods contained in the JUnit test class. Formally, let t_a be the test method under analysis and let $T = \{t_1, \ldots, t_n\}$ be the set of the test methods contained in the test suite, we compute the *Coupling Between Test Methods* (CBTM)

as follows:

$$CBTM(t_a) = \text{mean} \sum_{k=0}^{n} sim(t_a, t_k), t_a \neq t_k \quad (1)$$

where n is the number of test methods in T (excluding t_a) , and $sim(t_a, t_k)$ denotes the cosine similarity between t_a, the test method under analysis, and another test method t_k in T. The resulting value of $CBTM(t_a) \in [0, 1]$. The higher the $CBTM(t_a)$, the higher the coupling between t_a and the other test methods in the JUnit test class.

2.3 Computing Test Method Cohesion

When computing test method cohesion, our conjecture is that methods having low cohesion are characterized by low *textual similarity among the tested methods*. It is important to note that test methods generally have few lines of code, and therefore have a limited number of terms for setting up a similarity technique. This is the reason why we firstly expand the test method calls with the actual source code called by the test method, and then we start the textual-based computation. More formally, let $t = \{c_1, \ldots c_n\}$ be the JUnit test method under analysis where c_i is the *i-th* method call in t. We first modify the test method by replacing all its method calls with the corresponding body of the called production methods. Note that in this operation we only replace the test method calls with the source code of the corresponding production method, without replacing the calls done by the production method, i.e., we do not recursively replace all the method calls. Formally, the modified test method is $t' = \{c'_1, \ldots c'_n\}$ where the generic method call c_i is replaced by the corresponding production code body c'_i. Starting from t' we compute the *Lack of Cohesion of a Test Method* (LCTM) as the average similarity between its constituent methods c'_i as follows:

$$LCTM(t) = 1 - \underset{i \neq j}{\text{mean}}\ sim(c'_i, c'_j) \quad (2)$$

where n is the number of method calls in t, and $sim(c'_i, c'_j)$ denotes the cosine similarity between two modified method calls c'_i and c'_j in t'. Based on the above definition, LCTM(t) $\in [0, 1]$. If a test method is cohesive, then the LCTM(t) is close to zero.

3. INVESTIGATING THE QUALITY OF AUTOMATICALLY GENERATED TESTS

The *goal* of this study is to apply the previously defined coupling and cohesion test code metrics in order to investigate the quality of JUnit classes automatically generated using *EvoSuite* [18]. Thus, we formulate the following RQ:

> **RQ_0:** *To what extent do automatically generated test cases present design problems?*

Specifically, we aim at analyzing to what extent design problems affect automatically generated test cases. The *purpose* is to investigate possible gains (if any) from the application of quality metrics in the context of automatic test case generation.

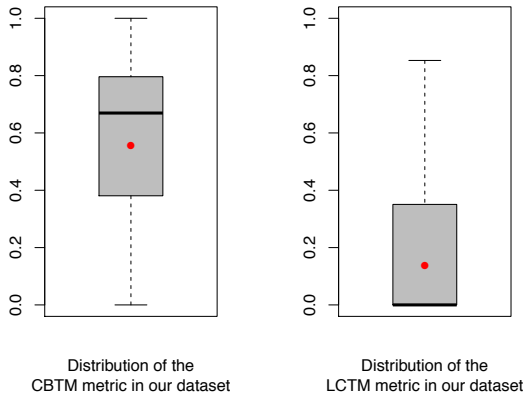
3.1 Experimental Procedure

The *context* of this study is *SF110*, the open dataset[1] produced by Fraser and Arcuri [20], which contains 110 open source projects from SourceForge[2], for a total of 23,886 testable classes. We choose to analyze this dataset, and therefore the behavior of the *EvoSuite* tool, because of the need to have a dataset able to generalize the results of our study. Table 1 provides a summary of the statistics of the SF110 dataset.

Table 1: RQ_p: Characteristics of the SF110 dataset

Characteristic	Value
Number of Projects	110
Number of Testable Classes	23,886
Lines of Code	6,628,619
Number of Java Files	27,997

Figure 1: Distribution of LCTM and CBTM metrics over the SF110 dataset.



To address **RQ_0**, we developed a tool implementing the quality metrics described in Section 2. More precisely, our tool mines the source code of each project in order to extract EvoSuite's automatically generated JUnit classes. It is worth noting that the SF110 dataset already contains, for each project, a set of automatically generated test suites. Thus, we did not need to run EvoSuite, but we rely on the test classes present in the dataset, taking them as representative of the quality of test code produced by EvoSuite. Then, we analyzed all the test methods contained in the extracted JUnit classes in order to assign to each of them a value of cohesion and coupling. The output of the tool consists of two distributions: the first one concerning a coupling value for each test method, the second one having a cohesion value for each test method. The analysis of the results has been done by discussing the descriptive statistics of such distributions.

Listing 1: Example of test method with low cohesion.

```
1 public void test12() throws Throwable {
2       JSTerm jSTerm0 = new JSTerm();
3       jSTerm0.makeVariable();
4       jSTerm0.add((Object) "");
5       jSTerm0.matches(jSTerm0);
6       assertEquals(false, jSTerm0.isGround());
7       assertEquals(true, jSTerm0.isVariable());
8 }
```

[1] http://www.evosuite.org/experimental-data/sf110/

[2] http://sourceforge.net

3.2 Analysis of the Results

Figure 1 shows the distributions of the two quality metrics defined in Section 2 over the the test methods in our dataset. Moreover, the red dots represent the mean values of such distributions. Analyzing the CBTM distribution, we observed that, over the total of 83,408 test methods, the median value is 0.67, i.e., half of the test methods (i.e., 41,704) have a high value for CBTM and thus likely have high coupling. This result highlights how most of the test methods suffer of high coupling with the other methods of a test suite. On average, a test method has a value of coupling of 0.55. On the other hand, it is interesting to notice how the results for the cohesion distribution are completely different compared to the coupling one. In this case, we found that most of the test methods are in fact cohesive. This is not unexpected, as EvoSuite applies a minimization process to the generated test methods, i.e. it minimizes the number of statements of the test method by removing the statements that do not contribute to the coverage of the method under test [23]. However, even with this minimization process in place, there are 23,188 out of the total 83,408 (i.e., 28%) test methods having a lack of cohesion higher than 0.35. As an example, Listing 1 presents a JUnit test method from the `JSTermEvoSuiteTest` class originating from the `Shop` project. We can observe that the test method checks the return value of two different methods of the production code, i.e., `isGround` and `isVariable`. This method has a value of LCTM of 0.83, clearly indicating the lack of cohesion of the test case.

In Summary. From the results achieved after the analysis of the quality metrics on the SF110 dataset, we can conclude that even though measures have been taken to keep test code quality under control during the process of automatic test case generation (e.g., the minimization process), the design of the automatically generated test cases can still be improved in terms of coupling and cohesion.

4. PUTTING QUALITY METRICS INTO THE LOOP

Results from our preliminary study revealed that automatically generated tests can be affected by high coupling as well as by low cohesion. Therefore, we would like to investigate whether incorporating quality metrics in the main loop of an evolutionary unit test generation tool will guide the search toward better tests, i.e, test with lower coupling and higher cohesion. Given the nature of evolutionary search algorithms used in test data generation tools, one theoretically simple strategy would be to consider both *cohesion*, and *coupling* as further objectives to consider in addition to code coverage which is the traditional main objective, within a multi-objective paradigm. This apparently simple strategy lead to the usage of Pareto efficient algorithms, such as NSGA-II, which by definition generate optimal trade-offs between cohesion, coupling and code coverage. However, working with such trade-offs is not particular useful for practitioners: imagine tests that have high cohesion and low coupling, but that do not exercise/test any additional code element (e.g., branch) compared to previously executed tests. Moreover, previous work that applies multi-objective approaches to combine code coverage (traditional objective) with non-coverage based objectives have reported a detrimental effect on the final code coverage [17, 25, 32, 37]. For example, Ferrer *et al.* [17] combined two conflicting goals, i.e., the coverage (to maximize) and the oracle cost (to minimize), using several multi-objective algorithms, including NSGA-II [15] and SPEA2 [48]. Results of their empirical study reported lower coverage scores for multi-objective algorithms if compared to algorithms focused on coverage only [17]. In our case, test code quality metrics can be considered only as secondary objectives if compared to code coverage. Indeed, two test cases can be compared in terms of cohesion and coupling if and only if they cover the same code elements, such as statements or branches. Vice versa, if two tests cover and exercise two different portions of code, they should be incomparable in terms of test code quality (i.e., cohesion and coupling) because preferring one test over the other one will lead to a decrease in code coverage. Another important aspect to deal with is the proper encoding schema to use. The default situation in EvoSuite is that a chromosome (individual) is a test suite composed of a number of test cases, where each test is a random sequence of statements (e.g., method calls). Thus, individuals have a variable number of test cases with a variable number of statements each. Suites are, then, evolved using genetic algorithms (GAs) targeted at maximizing a code coverage metric, e.g., branch coverage. However, in such encoding schema GAs select, evaluate and evolve test suites while our quality metrics (CBTM, and LCTM) are designed to detect cohesion and coupling issues at the test case level. Evaluating cohesion and coupling at the test suite level is not so trivial: it implies aggregating CBTM and LCTM scores of all test cases in each test suite. Using such an aggregation, test cases with good levels of cohesion (low LCTM) but contained in a test suite with an average to bad level of cohesion (high LCTM) may be not selected. Finally, coupling (i.e., CBTM) may vary when tests are inserted into different test suites: a test case may be highly coupled with test cases within a given test suite, while the same test case may have a low coupling if inserted in a different suite. For the reasons reported above, we incorporate our test code quality metrics into MOSA (Many-Objective Sorting Algorithm) [35], a novel many-objective algorithm proposed by Panichella *et al.* [35] for branch coverage[3]. We selected such an algorithm because (i) it evolves test cases and not suites, making it easier to incorporate CBTM and LCTM; (ii) it targets branch coverage as main goal to maximize plus a secondary non-coverage based metric, i.e., test case size to minimize; (iii) it leads to higher or competitive branch coverage compared to existing search algorithms for test case generation [1, 35]. MOSA targets all branches in the class under test at once, which are considered as distinct and different objectives to be optimized [35]. Each objective measures the *closeness* of a test case t to cover the corresponding uncovered branch b_i according to the normalized *branch distance*, and the *approach level* [28], i.e., the minimum number of control dependencies between the executed traces and the branch. In a nutshell, MOSA starts the search by generating random tests (chromosomes) with variable length and statements. Such tests are then evolved by applying a crossover operator and a mutation operator to generate new tests, as usual in GAs. Selection of tests for either reproduction and survival is based on a novel many-objective sorting algorithm, which extends the notion of dominance and gives

[3]The quality-based MOSA algorithm is publicly available in our online appendix [34].

higher probability of being selected to those test cases that are closest to at least one of the uncovered branches [35]. In the case of more tests having the same *branch distance + approach level* scores, MOSA uses a secondary *non-coverage* based criterion which is the test case size (number of statements) such that the test case with the lowest number of statements is preferred. Finally, the final test suite is built using an *archive* that keeps track of test cases as soon as they cover yet uncovered branches of the program under test. The archive is updated at the end of each generation by considering both main (branch closeness) and secondary (test size) objectives [35]. In order to incorporate CBTM and LCTM into MOSA, we suggest to replace the secondary objective proposed by Panichella *et al.*, i.e., the test case size, with a new secondary objective which takes into account both cohesion (LCTM) and coupling (CBTM) among test cases within a specific generation. More specifically, let us consider a generic population (set of test cases) $T = \{t_1, \ldots, t_n\}$ generated by MOSA at the end of a given generation. For each test $t_i \in T$ we can compute the corresponding *lack of cohesion* LCTM(t_i) as the average textual dissimilarity between its constituent method calls (to minimize) as proposed in Section 2. We can also compute the coupling CBTM(t_i) for each test $t_i \in T$ as the average textual similarity between t_i and all other test cases $t_j \neq t_i,\ t_j \in T$ in the current population (to miminize). Once we computed LCTM and CBTM for all tests in the current population, we use a sum-aggregation approach to have only one final quality score for each test t_i as follows:

$$quality\_score(t_i) = \alpha \times \text{LCTM}(t_i) \ + \ \beta \times \text{CBTM}(t_i) \quad (3)$$

where α and β are linear combination coefficients taking values within the interval $[0; 1]$. An optimal *quality_score* value for t_i would be closer to zero, since it implies that t_i is highly cohesive (low *lack of cohesion* LCTM) and poorly coupled (low coupling CBTM) to other tests in the same population. Different α and β coefficients assign different weights to the constituent metrics. However, in this paper we set $\alpha = 0.5$ and $\beta = 0.5$ in order to give the same importance to LCTM (cohesion) and CBTM (coupling). Our *quality_score* is incorporated into MOSA by replacing the original secondary criterion (i.e., test case size). Therefore, at the end of each generation MOSA still uses the *preference criterion* to increase the selection pressure by giving higher priority to those tests that are closer to cover at least one of the uncovered branches (closeness based on the approach level and branch distance). However, when there are multiple test cases equally close to cover an uncovered branch b_i, among them we select the one with the lowest *quality_score*. Finally, we also modify the procedure to update the archive using the algorithm reported for completeness in Algorithm 1. The archive is the data structure used to store test cases as soon as they cover yet uncovered branches. In [35] the archive is updated at the end of each generation to keep track of new tests (covering previously uncovered branches) and again using the test case size as secondary criterion. In our case, when updating the archive, we re-compute the *quality_score* but this time computing the CBTM metric (coupling) with respect to all tests stored in the archive (line 10 in Algorithm 1). Indeed, when updating the archive new additional tests may worsen then overall coupling among tests in the archive, which is the final test suite. Therefore, Algorithm 1 selects among the set of all test covering a specific branch b_i the one presenting the best internal cohesion (LCTM) and with the lowest coupling (CBTM) with the whole archive.

Algorithm 1: Quality-based Update Archive

```
   Input:
   A population T
   An archive A
   Result: The updated archive A
1  begin
2      for each branch b_i do
3          t_best ⟵ ∅
4          best_quality ⟵ ∞
5          if b_i already covered then
6              t_best ⟵ test case in A covering b_i
7              best_quality ⟵
                 0.5 × LCTM(t_j) + 0.5 × CBTM(t_j, A)
8          for t_j ∈ T do
9              score ⟵ objective score of t_j for branch b_i
10             quality ⟵ 0.5 × LCTM(t_j) + 0.5 × CBTM(t_j, A)
11             if score == 0 and quality ≤ best_quality then
12                 t_best ⟵ {t_j}
13                 best_quality ⟵ quality
14         if t_best ≠ ∅ then
15             A ⟵ t_best
```

5. STUDY II: EMPIRICAL EVALUATION

In this section we describe an empirical study aimed at evaluating the effect of cohesion and coupling on the performance of search algorithms for test case generation. In particular, our second study is steered by the following research questions:

> **RQ$_1$:** *Does quality optimization produce more cohesive and less coupled tests?*
>
> **RQ$_2$:** *Does quality optimization affect the achieved branch coverage?*
>
> **RQ$_3$:** *Does quality optimization affect the size of produced tests?*

The usage of coupling and cohesion metrics as part of the selection procedure in search algorithms is expected to improve cohesion and coupling of the produced tests. However, with **RQ$_1$** we want to verify whether this is true and whether such (eventual) improvements are statistically significant. One potential drawback of combining coverage criteria with non-coverage criteria is that such combination may have detrimental effects on the final code coverage since part of the search effort is not devoted to maximize coverage. Hence, **RQ$_2$** is aimed at verifying whether branch coverage increases or decreases when incorporating cohesion and coupling metrics using our proposed strategy. More cohesive and less coupled tests should imply that they do not contain unnecessary statements, i.e., statements that cover branches already covered by other tests in the final test suite. For this reason, **RQ$_3$** addresses whether the usage of quality optimization results in shorter tests if compared to a standard strategy targeting branch coverage only.

5.1 Experimental Procedure

For our second study we randomly selected 43 classes from the *SF110* corpus, instead of considering all 23,886 classes used in our first study. This selection is due to the high

Table 2: Classes in Our Dataset

No	Subject	Class	Branches
1	newzgrabber	Newzgrabber.Base64Decoder	59
2	falselight	Services	15
3	htf-bomberman	client.network.ClientMsgReceiver	15
4	openhre	com.browsersoft.openhre.hl7.impl.regular.ExpressionMatrixImpl	58
5	rif	com.densebrain.rif.client.RIFClassLoader	3
6	sweethome3d	com.eteks.sweethome3d.model.HomeEnvironment	80
7	tullibee	com.ib.client.EWrapperMsgGenerator	67
8	caloriecount	com.lts.application.international.MessageFormatter	52
9	caloriecount	com.lts.io.ArchiveScanner	45
10	jmca	com.soops.CEN4010.JMCA.JParser.JavaParser	7938
11	jmca	com.soops.CEN4010.JMCA.JParser.JavaParserTokenManager	1707
12	jmca	com.soops.CEN4010.JMCA.JParser.SimpleNode	67
13	freeminf	freemind.controller.Controller	410
14	lhamacaw	macaw.businessLayer.SupportingDocument	38
15	lhamacaw	macaw.businessLayer.Variable	153
16	xbus	net.sf.xbus.protocol.records.RecordTypeMessage	52
17	schemaspy	net.sourceforge.schemaspy.model.ForeignKeyConstraint	72
18	squirrelSQL	net.sourceforge.squirrelsql.client.session.Session	204
19	squirrelSQL	net.sourceforge.squirrelsql.plugins.dbcopy.util.DBUtil	463
20	lagoon	nu.staldal.lagoon.LagoonCLI	65
21	dom4j	org.dom4j.io.OutputFormat	67
22	dom4j	org.dom4j.io.SAXReader	101
23	openjms	org.exolab.jms.config.ConnectionFactories	114
24	firebird	org.firebirdsql.jdbc.FBCachedFetcher	118
25	jcvi-javacommon	org.jcvi.jillion.assembly.consed.phd.PhdBuilder	27
26	jcvi-javacommon	org.jcvi.jillion.core.Range	426
27	jcvi-javacommon	org.jcvi.jillion.core.residue.nt.DefaultNucleotideCodec	74
28	jcvi-javacommon	org.jcvi.jillion.core.util.FileIterator	67
29	jsecurity	org.jsecurity.authc.UsernamePasswordToken	29
30	pdfsam	org.pdfsam.guiclient.commons.models.VisualListModel	143
31	quickserver	org.quickserver.net.server.TheClient	43
32	quickserver	org.quickserver.util.xmlreader.AdvancedSettings	52
33	gangup	state.Player	44
34	weka	weka.classifiers.Evaluation	809
35	weka	weka.classifiers.bayes.NaiveBayesMultinomialText	194
36	weka	weka.classifiers.rules.JRip	416
37	weka	weka.core.Optimization	448
38	weka	weka.core.stemmers.LovinsStemmer	428
39	weka	weka.experiment.ResultMatrix	441
40	weka	weka.filters.unsupervised.attribute.Discretize	242
41	wheelwebtool	wheel.asm.Frame	687
42	wheelwebtool	wheel.components.Component	480
43	wheelwebtool	wheel.json.JSONObject	294

computational cost required for the comparison of different randomized algorithms. Details of the selected classes can be found in Table 2. The size of the selected classes varies from 3 branches for the class `RIFClassLoader` up to 7,938 branches for the class `JavaParser`. For each class in our sample we ran EvoSuite using (i) MOSA enriched with our test code quality metrics; (ii) MOSA without test code quality metrics; and (ii) the *whole suite* strategy, which is the default strategy in EvoSuite. We decided to consider a second baseline in order to have higher confidence of the generalizability of our results with respect to other well-established search algorithms for test case generation. All three algorithms were executed by targeting branch coverage as criterion to maximize. For each execution we collected (i) cohesion (LCTM) and coupling (CBTM) scores, branch coverage scores, and the final test suite length. We use a search budget of two minutes, thus, the search terminated when two minutes are reached or when maximum (100%) branch coverage was achieved. To take into account the randomness of the algorithms used in the study, we run each algorithm on each selected classes 30 times, for a total of 3 (search algorithms) $\times 43$ (classes) $\times 30$ (repetitions) $= 3,870$ different executions. All the test suites generated by the three experimented algorithms over the 30 runs are available in our online appendix [34].

To address $\mathbf{RQ}_1$ we compare cohesion and coupling scores for the tests produced by the three search algorithms at the end of the search. More precisely, we combine LCTM (cohesion) and CBTM (coupling) into a unique scalar value, defined as *Quality Metric*$= LCTM + CBTM$, in order to have only one scalar value to simplify the comparison. The two metrics LCTM and CBTM are computed using the equations reported in Section 2. Lower *quality metric* scores indicate that the obtained tests are highly cohesive (low LCTM) and poorly coupled (low CBTM). To verify whether the differences (if any) are statistical significant, we use the non-parametric Wilcoxon Rank Sum test [12] with p-value $= 0.05$ as threshold for significance. Besides testing the statistical significance, we measure the effect size of the differences using the Varga-Delaney measure ($\hat{A}_{12}$) [45] following the guidelines in [4] for assessing randomized algorithms. To address $\mathbf{RQ}_2$, we compare branch coverage scores achieved by the three search algorithms. Also for this research question we use both the Wilcoxon test and the Varga-Delaney measure to provide statistical support to our findings. For $\mathbf{RQ}_3$, we compare the *size* of the resulting test suites, where the size of a test suite is measured as the sum of the lengths (number of statements) of each composing test case. Note that we analyze the test suite size only for classes where we do not observe a significant improvement in terms of coverage. This is because test suites achieving higher branch coverage are expected to have more test cases (and more statements) than other suites with lower branch coverage for the same class under test. In other words, the test size metric comes into play for comparison if and only if two test suites under analysis achieve the same coverage score. It is important to notice that EvoSuite already applies post-processing steps in order to reduce the size of the final test suites improving readability and tests conciseness. Indeed, at the end of the search process a generated test suite and its constituent test cases are post-processed for mimimization. During this step, statements that do not contribute to satisfying each individual covered branch are removed from the tests. In the context of our study, such post-processing steps are applied to all three search algorithms under study. Therefore, any difference in test suite size obtained upon post-process minimization can be interpreted as the effect of incorporating cohesion and coupling metrics into the main loop of the search algorithm ($\mathbf{RQ}_3$).

5.2 Parameters Setting

We apply encoding schema and genetic operators for test cases available in EvoSuite. Specifically, we use a dynamic encoding schema, where each test case is a random sequence of statements (e.g., method calls) with variable length. Test cases are combined using the *single-point* crossover which randomly exchanges statements between two selected tests. Finally, each test is randomly modified by a mutation operator, which can add, delete or change statements with uniform probability. Therefore, the length of test cases can vary during the search. As parameter values we use the default parameters setting in EvoSuite since a previous study [5] demonstrated that parameter tuning does have impact on the performance of test case generation tools, which provide comparable results with default settings widely used in literature. For completeness, the main parameter values are [20]:

- **Population size**: we used the default population size of 50 chromosomes.
- **Search budget**: we restrict the search budget to two minutes for each independent run.
- **Crossover**: two selected test cases are combined using the single-point crossover with probability $P_c = 0.75$.
- **Mutation**: we use a uniform mutation function with probability $P_m = 1/k$ where k is the number of statements in the test to mutate.
- **Selection function**: tournament selection is used with

tournament *size* = 10, which is the default setting in EvoSuite.

5.3 Analysis of the Results

Table 3 reports the results for $\mathbf{RQ}_1$. Specifically, the table reports the average *quality metric* (LCTM + CBTM) obtained over 30 independents runs as well as the results of the Wilcoxon test and the Vargha-Delaney measure. Values $\hat{A}_{12} < 0.5$ indicate that the quality-based MOSA leads to higher cohesion and lower coupling than other algorithms considered in the comparison. In general, we can notice that tests generated by the quality-based MOSA have better *quality metric* scores as we would have expected. Indeed, for 33 out of 43 classes the quality-based MOSA produces tests with a statistically significant better (lower) *quality* metric when compared to MOSA. The $\hat{A}_{12}$ measure shows that for classes with statistical significant differences, the effect size is mostly *large* (76%) or *medium* (18%). Surprisingly, for three classes the quality-based MOSA obtains (statistically significant) worse scores than MOSA alone. One of these three classes is `AdvancedSettings` for which the quality-based MOSA produces slightly higher (+0.03) LCTM (lack of cohesion) and CBTM (coupling) when compared to MOSA (see line number 32 in Table 3 — cross-compare with Table 2 to find index). However, we notice that all algorithms quickly reach 100% of branch coverage after a few generations for such a class, which contains only 52 trivial branches (see Table 4 for coverage results). Hence, the quality-based MOSA ended the search after few generations without giving enough time (generations) for optimizing cohesion and coupling. The comparison with the Whole Suite strategy also reveals positive results for our quality-based algorithm. Indeed, for 32 classes (out of 43) the quality-based MOSA generates tests that are statistically more cohesive and less coupled with respect to those generated by Whole Suite, showing a *large* effect size in most of cases (78%). For only four classes, Whole Suite produces statistically better results (see lines 11, 18, 32, and 41 in Table 3). After manual investigation, we found that this happens only for those classes where all search algorithms perform only a few generations, thus, giving limited search time to optimize test quality (as for example for the class `AdvancedSettings`). In order to investigate the *quality metrics* scores of tests generated during the search, Figure 2 depicts the average quality metric scores (i.e., cohesion + coupling) achieved by the three experimented algorithms over time (5 minutes) for the class `LovinsStemmer` from the *weka* library. As we can see, the average quality score for our quality-based MOSA is always lower than those achieved by MOSA and Whole Suite. In particular, the average quality for Whole Suite does not vary over the time, while for MOSA either with or without quality optimization, we observe a general decreasing trend. However, for the quality-based MOSA the overall cohesion+coupling score always remains lower or equal to 1.30 while for MOSA it ranges between 1.55 and 1.30. We obtained consistent results also for all other non-trivial classes in our second empirical study.

For branch coverage ($\mathbf{RQ}_2$), the results are reported in Table 4. Specifically, the table reports the average branch coverage ([0,1]) obtained over 30 independents runs as well as the results of the Wilcoxon test and the Vargha-Delaney measure. Values $\hat{A}_{12} > 0.5$ indicate that the quality-based MOSA leads to higher branch coverage than the alternative

Table 3: Quality score achieved by Quality-based MOSA, MOSA and Whole Suite, together with p-values resulting from the Wilcoxon test and Vargha-Delaney $\hat{A}_{12}$ effect size. We use S, M, and L to indicate small, medium and large effect sizes respectively. Significantly p-values are reported in boldface.

N.	LCTM+CBTM			Quality-based MOSA vs. MOSA			Quality-based MOSA vs. Whole Suite		
	Q-based MOSA	MOSA	Whole Suite	p-value	A_{12}		p-value	A_{12}	
1	1.15	1.26	1.21	**0.01**	0.30	M	0.09	0.37	S
2	0.68	0.79	0.76	**<0.01**	0.20	L	**<0.01**	0.27	M
3	1.28	-	1.31	**<0.01**	0.20	L	**<0.01**	0.26	L
4	1.31	1.32	1.32	0.21	0.41	S	0.20	0.40	S
5	0.85	1.01	1.01	**<0.01**	0.19	L	**<0.01**	L	
6	1.13	1.17	1.15	**<0.01**	0.16	L	**0.04**	S	
7	0.76	0.88	0.86	**<0.01**	0.00	L	**<0.01**	0.01	L
8	0.94	1.12	1.12	**<0.01**	0.00	L	**<0.01**	0.00	L
9	1.01	1.23	1.25	**<0.01**	0.08	L	**<0.01**	0.05	L
10	1.05	1.01	1.02	**<0.01**	0.86	L	0.32	0.58	S
11	1.35	1.37	1.28	**0.01**	0.32	M	**<0.01**	0.88	L
12	1.10	1.22	1.18	**<0.01**	0.05	L	**<0.01**	0.13	L
13	0.52	0.53	0.54	**0.05**	0.35	S	**<0.01**	0.25	L
14	1.28	1.30	1.30	0.55	0.45	S	0.47	0.44	-
15	1.20	1.27	1.27	**<0.01**	0.04	L	**<0.01**	0.04	L
16	0.95	1.17	1.18	**<0.01**	0.00	L	**<0.01**	0.00	L
17	0.90	1.12	1.12	**<0.01**	0.00	L	**<0.01**	0.00	L
18	0.20	0.00	0.00	**<0.01**	0.70	M	**<0.01**	0.70	M
19	0.71	0.76	0.74	**<0.01**	0.16	L	**<0.01**	0.23	L
20	1.04	1.11	1.09	**<0.01**	0.25	L	**0.03**	0.34	S
21	1.12	1.22	1.21	**<0.01**	0.02	L	**<0.01**	0.04	L
22	0.96	1.05	1.05	**<0.01**	0.02	L	**<0.01**	0.01	L
23	1.20	1.20	1.20	0.23	0.59	S	0.92	0.49	-
24	1.28	1.30	1.28	**<0.01**	0.20	L	**0.01**	0.32	M
25	0.96	1.04	1.04	**<0.01**	0.22	L	**<0.01**	0.24	L
26	1.12	1.16	1.14	**<0.01**	0.11	L	0.87	0.49	-
27	0.73	0.77	0.76	**<0.01**	0.00	L	**<0.01**	0.00	L
28	1.22	1.31	1.30	**<0.01**	0.02	L	**<0.01**	0.07	L
29	0.92	1.08	1.04	**<0.01**	0.00	L	**<0.01**	0.03	L
30	1.18	1.21	1.21	**<0.01**	0.12	L	**<0.01**	0.11	L
31	1.18	1.20	1.20	**0.01**	0.30	M	**0.02**	0.32	M
32	1.30	1.27	1.27	**<0.01**	0.84	L	**0.01**	0.85	L
33	1.10	1.12	1.11	0.12	0.38	S	0.37	0.43	-
34	1.14	1.18	1.21	**0.02**	0.32	M	**<0.01**	0.18	L
35	1.11	1.26	1.25	**<0.01**	0.00	L	**<0.01**	0.00	L
36	0.99	1.03	1.04	**<0.01**	0.23	L	**<0.01**	0.21	L
37	1.30	1.33	1.35	**<0.01**	0.27	M	**<0.01**	0.22	L
38	1.14	1.21	1.21	**<0.01**	0.10	L	**<0.01**	0.10	L
39	1.10	0.99	1.00	**<0.01**	1.00	L	**<0.01**	1.00	L
40	0.99	1.23	1.23	**<0.01**	0.00	L	**<0.01**	0.00	L
41	1.27	1.31	1.25	**<0.01**	0.28	M	**0.02**	0.67	M
42	1.10	1.10	1.18	0.84	0.48	S	**<0.01**	0.02	L
43	1.01	1.04	1.06	0.03	0.34	S	**<0.01**	0.25	L

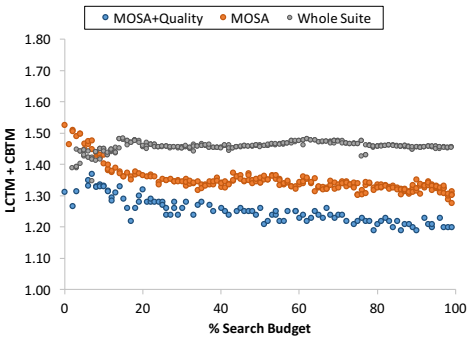


Figure 2: Cohesion and coupling over time for the class *LovinsStemmer* with a search budget of fivse minutes.

algorithm considered in the comparison. The obtained results show that in the majority of the cases (30 out of 43 classes) there is no difference in terms of branch coverage achieved by MOSA with and without quality optimization. This is an important result when compared to previous attempts reported in literature to combine coverage and non-coverage based criteria in test case generation, for which negative effects are reported for the final coverage [17]. From

Table 4: Branch coverage achieved by Quality-based MOSA, MOSA and Whole Suite, together with p-values resulting from the Wilcoxon test and Vargha-Delaney $\hat{A}_{12}$ effect size. We use S, M, and L to indicate small, medium and large effect sizes respectively. Significantly p-values are reported in boldface.

N.	Branch Coverage			Quality-based MOSA vs. MOSA			Quality-based MOSA vs. Whole Suite		
	Q-based MOSA	MOSA	Whole Suite	p-value	A_{12}		p-value	A_{12}	
1	0.64	0.72	0.62	0.69	0.47	-	0.48	0.55	-
2	0.68	0.72	0.73	0.01	0.37	S	**<0.01**	0.33	M
3	0.47	0.44	0.46	0.33	0.53	-	0.97	0.50	-
4	0.94	0.94	0.93	0.96	0.50	-	0.18	0.60	S
5	1.00	1.00	1.00	-	0.50	-	-	0.50	-
6	0.94	0.94	0.94	1.00	0.50	-	1.00	0.50	-
7	0.82	0.80	0.79	**0.01**	0.69	M	**<0.01**	0.73	M
8	0.84	0.82	0.83	**<0.01**	0.72	M	**0.02**	0.67	M
9	0.60	0.63	0.67	0.43	0.44	-	**<0.01**	0.26	L
10	0.21	0.22	0.14	0.28	0.42	S	**<0.01**	0.93	L
11	0.47	0.47	0.36	0.34	0.43	-	**<0.01**	0.82	L
12	0.81	0.85	0.81	**0.01**	0.31	M	0.83	0.52	-
13	0.04	0.04	0.04	**<0.01**	0.73	M	**<0.01**	0.71	M
14	1.00	1.00	1.00	-	0.50	-	-	0.50	-
15	0.94	0.96	0.97	**<0.01**	0.20	L	**<0.01**	0.16	L
16	0.52	0.51	0.53	0.34	0.56	-	0.25	0.43	-
17	0.81	0.81	0.79	0.33	0.48	-	**0.01**	0.62	S
18	0.00	0.00	0.00	-	0.50	-	-	0.50	-
19	0.21	0.22	0.19	**<0.01**	0.29	M	**<0.01**	0.73	M
20	0.31	0.32	0.33	0.12	0.39	S	**<0.01**	0.14	L
21	1.00	1.00	1.00	-	0.50	-	-	0.50	-
22	0.90	0.92	0.91	**<0.01**	0.27	M	0.09	0.37	S
23	0.89	0.87	0.33	0.34	0.57	-	**<0.01**	1.00	L
24	0.43	0.43	0.44	1.00	0.50	-	0.17	0.45	-
25	0.96	0.91	0.97	0.49	0.54	-	0.29	0.57	-
26	0.69	0.73	0.57	**<0.01**	0.21	L	**<0.01**	0.97	L
27	0.97	0.97	0.96	0.74	0.52	-	**<0.01**	0.74	L
28	1.00	1.00	0.99	0.06	0.40	S	**0.02**	0.66	S
29	0.72	0.72	0.72	-	0.50	-	-	0.50	-
30	0.74	0.76	0.73	**0.03**	0.34	S	0.10	0.63	S
31	1.00	1.00	1.00	-	0.50	-	-	0.50	-
32	1.00	1.00	1.00	-	0.50	-	-	0.50	-
33	0.95	0.97	0.97	**0.04**	0.40	S	0.21	0.43	-
34	0.31	0.31	0.30	0.46	0.44	-	0.06	0.64	S
35	0.65	0.68	0.59	0.11	0.38	S	**<0.01**	0.77	L
36	0.21	0.22	0.23	0.24	0.41	S	**<0.01**	0.27	M
37	0.06	0.06	0.06	**0.04**	0.57	-	**<0.01**	0.85	L
38	0.66	0.66	0.62	0.33	0.57	S	**<0.01**	0.95	L
39	0.81	0.81	0.78	0.78	0.46	-	0.09	0.67	M
40	0.29	0.29	0.30	0.43	0.56	-	0.09	0.37	S
41	0.75	0.75	0.52	0.25	0.41	S	**<0.01**	1.00	L
42	0.64	0.54	0.53	**<0.01**	0.76	L	**<0.01**	0.92	L
43	0.84	0.85	0.85	0.68	0.47	-	0.79	0.48	-

Table 4 we also observe that the quality-based MOSA outperforms Whole Suite in 17 out of 43 cases, inheriting its effectiveness from the standard MOSA algorithm [35]. Finally, no statistically significant difference is observed for other 22 classes. There are some cases where the Wilcoxon test reveals statistically significant differences, that deserve a more detailed analysis. For five classes in our sample (lines 7, 8, 13, 37, and 42 in Table 4), the usage of quality metrics has a significant positive effect on branch coverage, with an improvement up to 10% for the class `Component`. To have a better understanding of such effects, Figure 3 depicts the variation of branch coverage achieved by the three algorithms (i.e., MOSA, quality-based MOSA and Whole Suite) over time for the class `Component`. To have a broader view, we consider a larger time window of five minutes as search budget. From Figure 3, we can observe that for the first 60 seconds MOSA is particularly effective compared to the other alternatives. However, after the first 90 seconds it is not able to cover any additional branches. On the other hand, the quality-based MOSA is less effective in the first 90 seconds. We conjecture that this is due to the overhead required for computing cohesion and coupling for tests in each generation. However, after consuming 30% of search budget our quality-based MOSA becomes more effective leading to a final higher branch coverage (+10%) at the end. Hence,

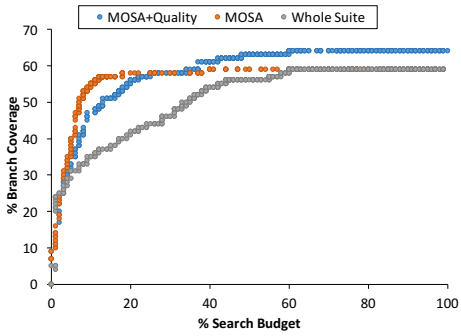


Figure 3: Branch coverage over time for the class *Component* with a search budget of five minutes.

the usage of cohesion and quality metrics seems to avoid the premature convergence of MOSA for this particular class: without quality metrics the same algorithm is not able to cover further branches. According to Figure 3, Whole Suite is less effective at reaching high coverage with respect to MOSA (either with or without cohesion and coupling) for most of the search time, confirming the high effectiveness of customized many-objective search in test case generation [35]. However, after 60% of the time simple MOSA and Whole Suite reached the same coverage score and both are not able to cover further branches for the rest of the search budget. For branch coverage we also observe that there are eight classes (lines 2, 12, 15, 19, 22, 26, 30, and 33 in Table 4) for which cohesion and coupling have a negative effect on MOSA with a coverage decrease ranging between -1% and -4%. One such class is `SimpleNode` from *jmca*, for which MOSA achieves 85% branch coverage against 81% for the quality-based MOSA. One possible explanation for this results is that computing quality metrics requires further overhead in addition to the computational complexity of genetic operators, e.g., the computational cost required to compute the textual similarity between tests (chromosomes) in each population. To provide a more in depth analysis, Figure 4 depicts the branch coverage scores achieved by the three experimented algorithms over time on `SimpleNode` but using a larger search budget of five minutes instead of 2 minutes used in our original study. We can notice that at the beginning of the search (up to 2 minutes) MOSA is, indeed, faster in reaching higher coverage as also reported in line 12 of Table 4. However, when increasing the search budget to over 2 minutes, MOSA and the quality-based MOSA are indistinguishable. At the end of the search (five minutes) our quality-based MOSA is able to cover more branches leading to an average increase of coverage of +2%. Therefore, the usage of quality metrics can help in improving branch coverage by avoiding premature convergence but only when giving more time to the search. On the other hand, when limited search budget is given, not including quality metrics is more effective because it will not increase the overall overhead for each generation. Potentially, better performance may be achieved by developing an adaptive strategy to incorporate cohesion and quality into MOSA's main loop according to the time assigned for the search, e.g., by disabling the computation of our metrics when the search budget is lower than a specific threshold. This is part of our future agenda. Finally, Table 5 reports the average test suite size

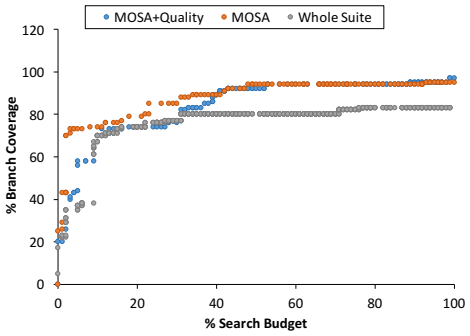


Figure 4: Branch coverage over time for the class *SimpleNode* with a search budget of five minutes.

Table 5: Suite sizes achieved by Quality-based MOSA, MOSA and Whole Suite, together with p-values resulting from the Wilcoxon test and Vargha-Delaney $\hat{A}_{12}$ effect size. We use S, M, and L to indicate small, medium and large effect sizes respectively. Significantly p-values are reported in boldface.

N.	Suite Size			Quality-based MOSA vs. MOSA			Quality-based MOSA vs. Whole Suite		
	Q-based MOSA	MOSA	Whole Suite	p-value	A_{12}		p-value	A_{12}	
1	15	14	14	**0.05**	0.54	S	**0.01**	0.58	M
3	12	12	12	0.06	0.59	S	0.21	0.46	-
4	21	20	20	**<0.01**	0.27	M	0.02	0.33	M
5	5	5	5	0.73	0.52	-	0.73	0.52	-
6	114	141	139	**<0.01**	0.01	L	**<0.01**	0.03	L
12	70		68		-	L	0.22	0.51	-
14	41	44	44	**<0.01**	0.17	L	**<0.01**	0.17	L
16	185	202	204	**0.03**	0.34	S	**<0.01**	0.26	L
17	70	75		**<0.01**	0.21	L			
18	3	3	3		0.50	-	-	0.50	-
20	27	27		0.74	0.52	-			
21	98	90	89	**<0.01**	0.86	L	**<0.01**	0.97	L
22	148		160				**0.01**	0.31	M
23	168	176		**<0.01**	0.97	L			
24	73	73	73	0.73	0.52	-	0.89	0.49	-
25	102	110	114	0.13	0.38	S	0.15	0.39	S
27	94	96		0.42	0.56	-			
28	65	65		0.94	0.49	S			
29	31	35	35	**<0.01**	0.03	L	**<0.01**	0.17	L
30	165		172				**<0.01**	0.94	L
31	74	87	86	**<0.01**	1.00	L	**<0.01**	0.04	L
32	49	53	53	**<0.01**	0.13	L	**<0.01**	0.13	L
33	58		62				**<0.01**	0.29	M
34	214	203	127	0.77	0.52	-	**<0.01**	0.06	L
35	108	105		0.53	-	S			
36	172	185		**<0.01**	0.04	L			
38	127	128		0.38	0.57	-			
39	165	186	162	**0.04**	0.16	L	0.12	0.56	S
40	81	75	79	**<0.01**	0.64	S	0.61	0.54	-
41	845	860		**0.02**	0.33	M			
42	383	479		0.70	0.54	-			
43	221	234	219	**0.03**	0.34	M	0.18	0.60	S

obtained over 30 independents runs (**RQ**$_3$) as well as the statistical significance according to the Wilcoxon test and the Vargha-Delaney measure. In this case, values of $\hat{A}_{12} < 0.5$ indicate that our quality-based MOSA leads to shorter test suites (with less statements) than other algorithms. Comparisons in terms of test suite size are reported only for those cases where we do not observe a significant improvement in terms of branch coverage (**RQ**$_2$). The Wilcoxon test reveals that our quality-based MOSA produces significantly shorter test suites with respect to both MOSA and Whole Suite. Indeed, in 11 classes out of 30, the test suite generated by the quality-based MOSA are statistically shorter then those generated by MOSA, with an average size reduction raging between 1% and 19%. Vice versa, MOSA provides test suites significantly shorter than its quality-based version in only four cases. Similar results are also obtained from the comparison with the Whole Suite: for ten classes the generated test suites are statically shorter when using the quality-based MOSA. These results are quite surprising if we consider that for all three algorithms —i.e., MOSA, quality-based MOSA, and Whole Suite— the generated test suites are post-processed for minimization. Indeed, despite this minimization, the usage of cohesion and coupling seems to provide a complementary support toward the generation of shorter tests likely improving readability.

In Summary. The incorporation of cohesion and coupling into the main loop of test case generation tools help in improving the overall quality of generated tests, which are statistically more cohesive and less coupled (**RQ**$_1$). We also discover that such quality metrics have positive effects both on branch coverage and test suite size. Indeed, when giving more time to the search test quality metrics may reduce the probability of early convergence increasing the final branch coverage (**RQ**$_2$). Finally, the generated tests tend to be shorter, suggesting the possibility to complement existing post-search minimization strategies (**RQ**$_3$).

6. THREATS TO VALIDITY

This section describes the threats that can affect the validity of our empirical study.

Construct Validity. An important threat related to the relationship between theory and observation is due to imprecisions/errors in the measurements we performed. For the first study, we defined two software quality metrics based on textual analysis. Although we are aware that these metrics might not be the best ones for measuring test code quality, several previous studies demonstrated, on the one hand, the usefulness of textual analysis for measuring code quality [39, 42] and, on the other hand, the ability of textual-based techniques to measure code quality with higher accuracy with respect to structural metrics [26, 39, 42]. However, the analysis of different quality metrics, including structural ones, is part of our future agenda. As for the evaluation of the performances of the GA, we adopted widely used metrics. Indeed, we used branch coverage and size of the resulting test suite. In the context of test data generation, such metrics gave a good measure of the efficiency and the effectiveness of the test data generation techniques. Finally, for measuring the post-process quality of the test suites, we used the same textual metrics defined in Section 2.

Internal Validity. Considering the way the test code quality metrics are computed, the main threat is represented by the settings used for the IR process. During the preprocessing, we filtered the textual corpus by using well known standard procedures: stop word list, identifiers splitting, stemmer and the *tf-idf* weighting schema [8]. As for LSI, we used the heuristics defined by Kuhn *et al.* [24] for choosing the number of concepts (k). Regarding the second study, a potential threat that might affect our results is represented by the inherent randomness of GA. To limit such threat, we repeated each execution 30 times. Moreover, another factor could be represented by the parameters used for setting up the GA. We are aware that different configurations might result in different results, however finding the best configuration is an extremely difficult task. Furthermore, in practice the gains to be made do not always pay off compared to using default configurations widely used in literature [6]. Thus, we rely on a basic setting using default values as suggested

in literature.

Conclusion Validity. As for threats related to the relationship between the treatment and the outcome, in the analysis of the results we used appropriate statistical tests coupled with enough repetitions of the experiments to enable the statistical tests. In particular, we used the Wilcoxon test for testing significance in the differences, while the Vargha-Delaney effect size statistic for estimating the magnitude of the observed difference. Moreover, we tried to draw conclusions only when statistically significant results were available.

External Validity. Threats to the external validity regard the generalization of our findings. During the first study, we carried out a large-scale empirical study, involving 110 different software projects in order to evaluate the quality of test cases automatically generated. Instead, our second study involves 43 Java classes from different projects, having different size and different number of branches (ranging between 3 and 7,938). Thus, the selected classes exhibit a reasonable degree of diversity. However, the replication of the study on a larger number of classes is desirable and therefore part of our research agenda.

7. RELATED WORK

Over the last years, the research community has spent a lot of effort on the definition of search based tools aimed at automatically generating test data by targeting branch coverage as the primary goal to achieve [27]. However, researchers have noticed that there are further goals that testers would like to achieve in addition to code coverage, such as memory consumption [25], etc. Most of approaches in this research thread have considered coverage and non-coverage criteria as equally important objectives to reach and, thus, they have used explicit multi-objective algorithms to optimize them. For example, Ferrer *et al.* [17] proposed a multi-objective search based algorithm that tries to balance coverage and oracle cost. Specifically, their approach target one branch at time, trying to maximize the coverage and minimize the cost. Pinto and Vergilio [37] also used the *single target* approach —i.e., targeting only one branch at time— as well as execution time and ability of tests to reveal faults as three distinct objectives to optimize using three-objective search algorithms. Similarly, Oster and Saglietti [31] applied bi-objective optimization for maximizing branch coverage and minimizing the number of test cases. Instead, Lakhotia *et al.* [25] considering dynamic memory consumption as second objective to consider in addition to branch coverage. However, all previous work that applies multi-objective approaches to combine code coverage with non-coverage based objectives reported no effect or even harmful effects on the final code coverage [17]. More recently, Afshan *et al.* [2] have noticed that *readability* is a critical non-coverage criteria to take into account since testers are required to manually check test data input and test results (e.g., assertions) in order to assess whether the software behaves as intended. Indeed, unreadable tests imply a substantial increase of human effort involved in manually checking the inputs produced [2]. Therefore, they propose to use natural language models to generate tests with readable string input data that are easy to comprehend for humans. Daka *et al.* [14] proposed a post-processing technique to optimize readability by mutating generated tests leveraging a domain-specific model of unit test readability based on human judgement. Other strategies to improve test readability also include the usage of mutation analysis in order to reduce the number of assertions [19]. In this paper, we observe that *test code quality* is a desirable and important goal in test case generation, since poorly designed tests have been proven to negatively impact future maintenance activities [7, 30, 47, 46]. Specifically, we incorporate two *test code quality* metrics into the loop of MOSA (Many-Objective Sorting Algorithm) [35], a novel many-objective algorithm that targets all branches in the code at once. As explained in Section 4, we added the quality of test code as *secondary non-coverage* criterion for selection, instead of the test case size used in the original implementation. Our empirical results show that, unlike previous attempts to combine coverage and non coverage-criteria into the search loop, code quality metrics have positive effect on code coverage reducing the probability of premature convergence. Furthermore, the generated tests tend to be shorter.

8. CONCLUSION

In this paper, we defined two textual-based test code metrics, i.e., the *Coupling Between Test Methods* (CBTM) and the *Lack of Cohesion of a Test Method* (LCTM), able to measure the degree of quality of a test case. In this first study we empirically investigated the quality of test cases automatically generated by EvoSuite on the *SF110* dataset, a set of 110 open source projects from SourceForge.

Our findings demonstrate that, even though measures have been taken to keep test code quality under control during the process of automatic test case generation (e.g., minimization), the design of the generated test cases can still be improved. For this reason, we incorporate the defined test code quality metrics into MOSA, a many-objective algorithm for automatic unit test generation [35].

In our approach, we put the quality indicator based on coupling and cohesion metrics as a secondary objective to be optimized by the search based algorithm implemented by MOSA. This contrasts the work of Afshan *et al.* [3] and Daka *et al.* [14], who address the problem of test code readability by applying linguistic models as a set of post-processing steps.

In the second study, we randomly selected 43 classes from the *SF110* dataset in order to evaluate the benefits provided by the quality-based automatic generation process. First of all, we observed that the generated test cases are statistically more cohesive and less coupled ($\mathbf{RQ}_1$). Moreover, the quality-based automatic generation process actually has a positive impact on branch coverage and test suite size ($\mathbf{RQ}_2$).

Finally, the size of the generated test cases tends to decrease, suggesting that our process can nicely complement existing post-search minimization strategies ($\mathbf{RQ}_3$). Our future work includes the evaluation of the impact of our quality-based algorithm on the effectiveness of test cases, as well as the evaluation of the effects on other maintainability factors, (e.g., readability). Moreover, we plan to assess possible gains (if any) from the application of other test code quality metrics in the automatic test case generation process.

9. REFERENCES

[1] *Results for EvoSuite - MOSA at the Third Unit Testing Tool Competition*, 2015.

[2] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *Proceedings International Conference on Software Testing, Verification and Validation (ICST)*, pages 352–361. IEEE, 2013.

[3] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *International Conference on Software Testing, Verification and Validation (ICST 2013)*, pages 352–361. IEEE, 2013.

[4] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 1–10. ACM, 2011.

[5] A. Arcuri and G. Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.

[6] A. Arcuri and G. Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering (EMSE)*, 18(3):594–623, 2014.

[7] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. Test code quality and its relation to issue handling performance. *IEEE Trans. Software Eng.*, 40(11):1100–1125, 2014.

[8] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[9] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S.-I. Yoo. The oracle problem in software testing: A survey. 2015.

[10] Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[11] 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 Trans. Softw. Eng. Methodol.*, 25(1):5:1–5:38, 2015.

[12] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd edition edition, 1998.

[13] J. K. Cullum and R. A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*, volume 1, chapter Real rectangular matrices. Birkhauser, Boston, 1998.

[14] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 107–118, New York, NY, USA, 2015. ACM.

[15] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast elitist multi-objective genetic algorithm: NSGA-II. *IEEE Trans. on Evolutionary Computation*, 6:182–197, 2000.

[16] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, (41):391–407, 1990.

[17] J. Ferrer, F. Chicano, and E. Alba. Evolutionary algorithms for the multi-objective test data generation problem. *Softw. Pract. Exper.*, 42(11):1331–1362, Nov. 2012.

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

[19] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Trans. Softw. Eng.*, 39(2):276–291, Feb. 2013.

[20] G. Fraser and A. Arcuri. A large scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):8, 2014.

[21] G. Fraser and A. Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20(3):611–639, 2015.

[22] M. Greiler, A. Zaidman, A. van Deursen, and M. D. Storey. Strategies for avoiding text fixture smells during software evolution. In T. Zimmermann, M. D. Penta, and S. Kim, editors, *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 387–396. IEEE Computer Society, 2013.

[23] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.

[24] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Information & Software Technology*, 49(3):230–243, 2007.

[25] K. Lakhotia, M. Harman, and P. McMinn. A multi-objective approach to search-based test data generation. In *9th Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1098–1105. ACM, 2007.

[26] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 133–142. IEEE, 2005.

[27] P. McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[28] P. McMinn. Search-based software testing: Past, present and future. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 153–163, 2011.

[29] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[30] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink. On the interplay between software testing and evolution and its effect on program comprehension. In T. Mens and S. Demeyer, editors, *Software Evolution*, pages 173–202. Springer, 2008.

[31] N. Oster and F. Saglietti. Automatic test data generation by multi-objective optimisation. In

J. GÃşrski, editor, *Computer Safety, Reliability, and Security*, volume 4166 of *Lecture Notes in Computer Science*, pages 426–438. Springer Berlin Heidelberg, 2006.

[32] N. Oster and F. Saglietti. Automatic test data generation by multi-objective optimisation. In *Computer Safety, Reliability, and Security*, volume 4166 of *Lecture Notes in Computer Science*, pages 426–438. Springer Berlin Heidelberg, 2006.

[33] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia. On the diffusion of test smells in automatically generated test code: An empirical study. In *Proceedings of the 9th International Workshop on Search-based Software Testing*, SBST 2016, 2016.

[34] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia. Online appendix - automatic test case generation: What if test code quality matters? Technical report. https://github.com/fpalomba/issta16-test-code-quality-matters.

[35] A. Panichella, F. M. Kifetew, and P. Tonella. Reformulating branch coverage as a many-objective optimization problem. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015*, pages 1–10, 2015.

[36] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE 2016, 2016.

[37] G. Pinto and S. Vergilio. A multi-objective genetic algorithm to test data generation. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 1, pages 129–134, Oct 2010.

[38] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[39] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of the 22Nd IEEE International Conference on Software Maintenance*, ICSM '06, pages 469–478, Washington, DC, USA, 2006. IEEE Computer Society.

[40] 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*, ISSTA 2015, pages 338–349, New York, NY, USA, 2015. ACM.

[41] P. Tonella. Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128. ACM, 2004.

[42] B. Ujhazi, R. Ferenc, D. Poshyvanyk, and T. Gyimothy. New conceptual coupling and cohesion metrics for object-oriented systems. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, SCAM '10, pages 33–42, Washington, DC, USA, 2010. IEEE Computer Society.

[43] A. van Deursen, L. Moonen, A. Bergh, and G. Kok. Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, pages 92–95, 2001.

[44] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Trans. Software Eng.*, 33(12):800–817, 2007.

[45] A. Vargha and H. D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[46] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production & test code. In *First International Conference on Software Testing, Verification, and Validation (ICST)*, pages 220–229. IEEE, 2008.

[47] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[48] E. Zitzler, M. Laumanns, and L. Thiele. Spea2: Improving the strength pareto evolutionary algorithm. Technical report, 2001.