Noname manuscript No.
(will be inserted by the editor)

Toward Granular Search-Based Automatic Unit Test Case Generation

Fabiano Pecorelli · Giovanni Grano · Fabio Palomba · Harald C. Gall · Andrea De Lucia

Received: date / Accepted: date

Abstract Unit testing verifies the presence of faults in individual software components. Previous research has been targeting the automatic generation of unit tests through the adoption of random or search-based algorithms. Despite their effectiveness, these approaches aim at creating tests by solely optimizing metrics like code coverage, without ensuring that the resulting tests have granularities that would allow them to verify both the behavior of individual production methods and the interaction between methods of the class under test. To address this limitation, we propose a two-step systematic approach to the generation of unit tests: we first force search-based algorithms to create tests that cover individual methods of the production code, hence implementing the so-called *intra-method tests*; then, we relax the constraints to enable the creation of *intra-class tests* that target the interactions among production code methods. The assessment of our approach is conducted through a mixed-method research design that combines statistical analyses with a user study. The key results report that our approach is able to keep the same level of code

Fabiano Pecorelli
*corresponding author
SeSa Lab - University of Salerno, Italy
E-mail: fpecorelli@unisa.it

Giovanni Grano
LocalStack, Switzerland
E-mail: me@giograno.com

Fabio Palomba and Andrea De Lucia
SeSa Lab - University of Salerno, Italy
E-mail: fpalomba@unisa.it
E-mail: adelucia@unisa.it

Harald C. Gall
SEAL Lab - University of Zurich, Switzerland
E-mail: harald.gall@uzh.ch

and mutation coverage while providing test suites that are more structured, more understandable and aligned to the design principles of unit testing.

Keywords Search-based Software Testing, Test Code Quality, Automatic Test Case Generation

1 Introduction

Software testing is the process adopted to verify the presence of faults in production code [47]. The first step of this process consists of assessing the quality of individual production code units [3], *e.g.*, classes of an Object-Oriented project. Previous studies [17, 71] have shown that unit testing alone may identify up to 20% of a project's defects and reduce up to 30% the costs connected with development time. Despite the undoubted advantages given by unit testing, things are worse in reality: most developers do not actually practice testing and tend to over-estimate the time spent in writing, maintaining, and evolving unit tests, especially when it comes to regression testing [10].

To support developers during unit testing activities, the research community has been developing automated mechanisms—relying on various methodologies like random or search-based software testing [4]—that aim at generating regression test suites targeting individual units of production code. For instance, Fraser and Arcuri [21] proposed a search-based technique, implemented in the EvoSuite toolkit,[1] able to optimize whole test suites based on the coverage achievable on production code by tests belonging to the suite. Later on, Panichella *et al.* [51] built on top of EvoSuite to represent the search process in a multi-objective, dynamic fashion that allowed them to outperform the state-of-the-art approaches. Further techniques in literature proposed to (1) optimize code coverage along with other secondary objectives (*i.e.*, performance [18, 30, 59], code metrics [49, 50], and others [42]) or (2) empower the underlying search-based algorithms by working on their configuration [6, 40, 72]. Yet, these approaches often fail to generate tests that are well-designed, easily understandable, and maintainable [21]. In addition, existing approaches do not explicitly follow well-established methodologies that suggest taking *test case granularity* into account [58]. In particular, when developing unit test suites, two levels of granularity should be preserved [35, 48, 58]: first, the creation of tests covering single methods of the production code should be pursued, *i.e.*, *intra-method* [58] or *basic-unit* testing [48]; afterwards, tests exercising the interaction between methods of the class should be developed in order to verify additional execution paths of the production code that would not be covered otherwise, *i.e.*, *intra-class* [58] or *unit* testing [48].

In this paper, we target the problem of granularity in automatic test case generation, advancing the state of the art by pursuing the first steps toward the integration of a systematic strategy within the inner-working of automatic test case generation approaches that might possibly support the production of

[1] http://www.evosuite.org

more effective and understandable test suites. We build on top of MOSA [51] to devise an improved technique, coined GRANULAR-MOSA (G-MOSA hereafter), that implements the concepts of intra-method and intra-class testing. Our technique splits the overall search budget in two. In the first half, G-MOSA forces the search-based algorithm to generate intra-method tests by limiting the number of production calls to one. In the second half, the standard MOSA implementation is executed so that the generation can cover an arbitrary number of production methods, hence producing intra-class test cases that exercise the interaction among methods.

We envision the proposed approach to be useful in multiple scenarios. On the one hand, intra-method testing allows the isolation of issues, supporting regression testing of individual components. There are two specific use cases where this testing strategy would be particularly useful. First, the regression testing of changes targeting the evolution of individual methods: intra-method testing would indeed help developers in the detection of defects, logic errors, and exceptions that may be present within a single method. By testing a method in isolation, a developer may pinpoint issues without the complexity introduced by the interactions with other methods or classes, favoring a *quick resolution* of these issues. Second, intra-method testing would be essential when refactoring operations are applied at the level of individual methods, e.g., an *Inline Method* refactoring that aims at merging together the code of two original methods [19]: in such a use case, developers would aim at improving the design of the code without altering its functional behavior. Having a comprehensive suite of intra-method tests would provide a safety net that would support developers in ensuring that no regressions are introduced during the refactoring process, hence verifying that the refactoring process worked as expected. On the other hand, intra-class testing focuses on the interactions between methods within the same class. In the first place, it helps identify issues that arise when methods collaborate to achieve a higher-level functionality, thus targeting more complex behaviors than those considered with intra-method testing. Additionally, it is worth considering that some defects can only be detected by looking at the way methods of a class interact with each other, i.e., some defects are complex enough not to be spotted when verifying individual methods. As a consequence, intra-class tests are essential for catching such issues, ensuring that the class functions as intended. Last but not least, this category of test cases might also be relevant when verifying the outcome of refactoring operations affecting classes, e.g., a *Move Method* operation that moves a method from a class to another, affecting the way the methods in both original and target classes communicate with each other [19]. In this condition, the proper application of refactoring can only be tested through intra-class test cases, as the refactoring operation itself is not limited to individual methods but may affect the behavior of entire classes.

On the basis of the considerations above, we see the definition of an automated approach able to include both types of tests within automatically generated tests as instrumental to *enlarge the conceptual scope of test case generators and potentially lead to their higher adoption in practice.* In the first place, current approaches do not provide developers with test cases that can explicitly support the use cases mentioned above. In this sense, our approach may increase the confidence that developers have in automatically generated test cases by letting them experiment with tests that can cover multiple situations occurring when evolving a software system. In the second place, forcing the automated test case generators to design intra-method and intra-class tests may have implications for usability and readability: we indeed hypothesize that the test suite resulting from the adoption of a method that explicitly consider the two types of test cases may be more readable and understandable for developers, making these test cases more useful from their own perspective.

We evaluate G-Mosa in the context of an empirical study featuring both statistical analyses and a user study, in an effort of assessing its effectiveness under multiple parameters such as (1) branch and mutation coverage, (2) test suite size, (3) complexity and coupling of the generated suites, (4) number of test smells, and (5) developers' understandability. We conduct our empirical investigation on a dataset of 100 non-trivial classes which has been previously employed in similar studies. In doing so, we also compare G-Mosa against Mosa, so that we may have a measure of the effect size of our results.

Our key findings show that the defined systematic strategy actually allows G-Mosa to create intra-method and intra-class test cases. More importantly, the resulting suites have a lower size per test case, a lower presence of test smells, and a higher understandability than those generated by Mosa, yet having a statistically similar level of code and mutation coverage. In other terms, G-Mosa can advance the state of the art by providing developers with an automated strategy able to ensure similar coverage levels than previous approaches while improving the overall degree of maintainability and understandability of the generated test suites.

To sum up, our paper provides four main contributions:

1. The definition and implementation of a novel, granular approach for automatic test case generation;
2. An empirical assessment of the approach as well as its comparison with a baseline technique;
3. A user study that evaluates the understandability of the generated test suites compared to the selected baseline.
4. A publicly available appendix [5] including both the implementation of G-Mosa and the data/scripts used to assess it, that might be used by researchers to replicate our study and/or build on top of our findings.

Structure of the paper. Section 2 provides background required to properly understand our research. In Section 3 we present the algorithmic details of G-Mosa, while Section 4 overviews the research questions that we will address. In Section 5 we report on the experimental details of the evaluation of our technique. Section 6 reports and discusses the results achieved over our experimentation while Section 7 discusses the possible threats to validity of our study. Finally, Section 8 outlines our next steps.

2 Background and Related Work

This section reports the basic concepts on automated tools to generate unit test suites as well as a discussion on related work.

2.1 Automatic Unit Test Case Generation

The problem of automatically generating test data has been largely investigated in the last decade [45]. Search-based heuristics—genetic algorithms [28] in particular—have been successfully applied to solve such a problem [45] with the goal to generate tests with high code coverage. Single-target approaches have been the first techniques proposed in the context of white-box testing [64]. These approaches divide the search budget among all the targets (typically branches) and attempt to cover each of them at a time. To overcome the limitation of single-target approaches, Fraser and Arcuri [21] proposed a multi-target approach, called *whole suite test generation (WS)*, that tackles all the coverage targets at the same time. Building on such idea, Panichella *et al.* [51] proposed a many-objective algorithm called MOSA. While WS is guided by an aggregate suite-level fitness function, MOSA evaluates the overall fitness of a test suite based on a vector of n objectives, one for each branch to cover. The basic working of MOSA can be summarized as follows. At first, an initial population of randomly generated tests is initialized. Such a population is then evolved through consecutive generations: new offsprings are generated by selecting two parents in the current population and then both crossover and mutation operators are applied [51]. MOSA introduced a novel *preference-sorting* algorithm to focus the search toward uncovered branches. This heuristic solves the problem of selecting non-dominated solutions that typically occurs in many-objective algorithms [43].

Algorithm 1: Random Generation of the Initial Population of Tests

```
Input: $M = \{m_1, m_2, ..., m_i\}$: methods of the CUT we want to cover
       Maximum attempts $A$
       Maximum size $L$
Result: $T\{s_1, s_2, ..., s_n\}$: test case with with $n$ statements
begin
    $T \leftarrow \emptyset$
    $r \leftarrow$ RANDOM-NUMBER(1, $L$)
    while not(max attempts reached) AND ($|T| \leq L$) do
        $p \leftarrow$ RANDOM-NUMBER(0, 1)
        if $p \leq$ INSERTION-UUT then
            INSERT-CALL-ON-CUT($T$)
        else
            $v \leftarrow$ SELECT-VALUE($T$)
            INSERT-CALL-ON-VALUE($T$, $v$)
    return $T$
```

Random Test Case Generation. To provide the reader with the necessary context, we introduce the basics of the mechanism used by EvoSuite [20] to randomly initialize the first generation of tests. More details can be found in the paper by Fraser and Arcuri [21]. A tests case is represented in EvoSuite by a sequence of statements $T = \{s_1, s_2, ..., s_l\}$ where $|T| = l$. Each s_i has a particular value $v(s_i)$ of type τ. The pseudo-code for the random test cases generation is showed in Algorithm 1. At first, EvoSuite chooses a random $r \in (1, L)$ where L is the test maximum length (*i.e.*, number of statements) (line 3 of Algorithm 1). Thus, EvoSuite initializes an empty test and tries to add new statements to it. Such a logic is implemented in the `RandomLengthTestFactory` class. EvoSuite defines five different kinds of statements [21]: (i) primitive statements (S_p), *e.g.*, creating an Integer or a String variable, (ii) constructor statements (S_c), that instantiate an object of a given type, (iii) field statements (S_f) that access public member variables, (iv) method statements (S_m), *i.e.*, method invocations on objects (or static method calls), and (v) assignment statements (S_a) that assign a value to a defined variable. The value v and the type τ of each statement depend on the generated statement itself, *e.g.*, the value and type of method statement will depend on the return value of the invoked method. In the preprocessing phase, a test cluster [70] analyzes the entire SUT (*system under test*) and identifies all the available classes Ω. $\forall c \in \Omega$, the test cluster defines a set of $\{\mathcal{C}, \mathcal{M}, \mathcal{F}\}$, where $\mathcal{C}$ is the set of constructors, $\mathcal{M}$ if the set of instance method and $\mathcal{F}$ is the set of instance fields available for a class c, respectively.

EvoSuite tries to repetitively generate new statements (the loop from line 4 to line 10 in Algorithm 1) and add them to a test. The process continues until the test hits the maximum random length or the maximum number of attempts (a parameter in EvoSuite set to 1,000 by default) is reached (line 4 in Algorithm 1). EvoSuite can insert two main kinds of statements. With a probability lower than INSERTION-UUT (a property defined as 0.5 by default), EvoSuite generates a random call of either a constructor of the class under test (CUT) or a member class, *i.e.*, instance field of method (lines 6-7 in Algorithm 1). Alternatively, the tool can generate a method call to a value $v(s_j)$ where $j \in (0, i]$ and i is the position on which the statements will be added (lines 9-10 in Algorithm 1). In other words, EvoSuite invokes a method on a value of a statement already inserted into the test. Such a value is randomly selected among all the values from the statements from the position 0 to the actual position (line 9 in Algorithm 1) EvoSuite also takes care of the parameters or the callee objects needed to generate a given statement. For example, a call to an instance method of the CUT requires (i) the generation of a statement instantiating the CUT itself and (ii) the generation of a statement defining values needed as argument for the method call. The values for such parameters can either (i) be selected among the sets of values already in the test, (ii) set to `null`, or (iii) generated randomly.

To better understand the generation process, let consider the test case in Listing 1, which has been generated for the class `JavaParserTokenManager`. To create this test, Evosuite works as follows. Starting from an empty test, it

```
StringReader stringReader0 = new StringReader("#<z-K~+*04@s^W");
char[] charArray0 = new char[1];
stringReader0.read(charArray0);
JavaCharStream javaCharStream0 =
        new JavaCharStream(stringReader0, (-1273), 1, 77);
JavaParserTokenManager javaParserTokenManager0 =
        new JavaParserTokenManager(javaCharStream0);
Token token0 = javaParserTokenManager0.getNextToken();
```
Listing 1 Example of a test generated by Evosuite

decides with a certain random probability to insert a statement invoking an instance method of the CUT: in our example, the `getNextToken()` method (line 6 of Listing 1). However, Evosuite needs first to generate two other statements, *i.e.*, line 5 and 6 of Listing 1, respectively: a statements returning a value of type `JavaParserTokenManager` (*i.e.*, the callee of the method) and a statement returning a value of type `JavaCharStream` (*i.e.*, the parameter of the method). In turn the constructor of `JavaCharStream` will need a value of type `StringReader` (line 1 of Listing 1). Line 3 of Listing 1 is instead the result of the other kind of possible insertion, *i.e.*, a method call to a value already present in the test: the `stringReader0` object in this case. Similarly, the tool will generate the primitive statement at line 2 of Listing 1 to provide the parameter needed by such a call.

2.2 Related Work

During the last decades, researchers have been working on the definition of search-based solutions that automate the generation of test data [2]. Most of the proposed approaches target branch coverage as primary goal to achieve [46], but more recent investigations have attempted to consider additional goals that would be desirable for making automatic test case generation more practical and aligned to what testers would like to have: in this direction, techniques have been proposed to complement code coverage with memory consumption [42], oracle cost [18], execution time [59, 30], total amount of number of test cases [49], and code quality [50]. Rojas *et al.* [62] also proposed to combine multiple code coverage criteria during the generation process.

A more recent trend is represented by the adoption of natural language models to increase the overall readability of the generated tests [1]. As an example, Daka *et al.* [14] proposed a post-processing method that optimizes the readability of test cases by mutating them through a domain-specific model of unit test readability based on human judgment. Further strategies include the optimization of `assert` statements relying on mutation analysis [21].

Our paper builds upon the research conducted so far and proposes the introduction of a systematic approach to the generation of test cases. In this sense, the technique proposed can be applied on top of all the approaches

Algorithm 2: G-Mosa Algorithm

```
Input: $B = \{\tau_1, ..., \tau_m\}$: set of coverage targets of a program
       Population size $M$
Result: A test suite $T$
begin
    $T \leftarrow \emptyset$
    $\alpha \leftarrow$ intra-method-testing($M$)
    $\gamma \leftarrow$ MOSA($M$)
    $T_\alpha, B_\alpha \leftarrow$ GENERATE-TESTS($\alpha, B$)            /* half search budget */
    if $B_\alpha == \emptyset$ then
        return $T_\alpha$
    $T \leftarrow T \bigcup T_\alpha$
    $T_\gamma, B_\gamma \leftarrow$ GENERATE-TESTS($\gamma, B_\gamma$)          /* half search budget */
    $T \leftarrow T \bigcup T_\gamma$
    return $T$
```

mentioned above. In the context of our research, we selected Mosa as baseline since this represents a state of the art technique that has been shown to overcome other approaches reported in literature [53]; yet, the underlying idea of building intra-method tests first is general and can be complemented by the optimization of any primary/secondary objective.

It is also worth to mention the many empirical studies conducted on test cases automatically generated [2]. Researchers have indeed empirically compared the performance of multiple approaches to the generation [69], other than investigating on a large-scale the performance of those tools [22, 23], the usability of testing tools in practice [12, 26, 63], and their quality characteristics [31, 29, 55].

The empirical study discussed in this paper clearly has a different connotation, as it aims to assess the capabilities of the proposed technique. Yet, it contributes to the body of knowledge since we also evaluated how test code maintainability can be improved by means of the systematic strategy implemented within our approach.

3 G-Mosa: A Two-Step Automatic Test Case Generation Approach

G-Mosa is defined as a two-step methodology that combines intra-method and intra-class unit testing [48, 58]. The pseudo-code of G-Mosa is outlined in Algorithm 2. The first step of the methodology generates tests that exercise the behavior of production methods in isolation: we indeed only allowed by design to generate intra-method tests (details in Section 3.1). The second step is based on the standard MOSA implementation [51] that performs intra-class unit testing by exercising a class trough a sequence of method call invocations. In the following, we detail each of these two steps.

Algorithm 3: Insert Random Call

```
Input: $T\{s_1, s_2, ..., s_n\}$: test case with with $n$ statements
       $S = \{s_1, s_2, ..., s_j\}$: setters of the CUT
Result: $T$: test case with with $n + 1$ statements
1  begin
2      $o \leftarrow$ GET-RANDOM-TEST-CALL
3      if $(o\ is\ a\ method)$ then
4          if $RANDOM\text{-}NUMBER(0,\ 1) \leq INSERTION\text{-}SET$ then
5              $T \leftarrow T \bigcup$ ADD-METHOD($s_j \in S$)
6              return $T$
7          $T \leftarrow T \bigcup$ ADD-METHOD($o$)
8          $T_c \leftarrow$ true
9      else
10         $T \leftarrow T \bigcup$ (ADD-COSTRUCTOR($o$) OR $T \bigcup$ ADD-FIELD($o$))
11     return $T$
```

3.1 Step I - Intra-Method Tests Generation

The intra-method testing process is the first step to be initialized (line 3 of Algorithm 2). Like any other test-case generation technique, a set of coverage targets B is given as input, namely the set of branches within the production class under test that the prospective test cases aim at covering. The intra-method process starts (line 5 of Algorithm 2) with B as target of the search and sets its search budget to the half of the overall budget available: in other words, if G-Mosa is given 180 seconds as budget, the intra-method testing process will run for 90 seconds. At the end of its search, the first step returns (i) T_α, the set of generated tests cases, and (ii) B_α, the set of uncovered targets. T_α and B_α will be used then as input for the second phase (see Section 3.2).

Intra-Method Code-Generation Engine G-Mosa is a variant of Mosa that applies first an intra-method testing methodology [48]: each generated test exercises a single production method of the CUT. To enable intra-method testing, we modified the code-generation engine used by EvoSuite to randomly generate new tests. In Section 2 we described such a mechanism: in a nutshell, EvoSuite inserts randomly generated statements (*e.g.*, calls to a class constructor or invocation of instance methods) in a test until a maximum number of statements is reached. This approach does not guarantee—nor has been designed to do it—any control on the number of instance method invocations of a test. As a consequence, tests might end up containing a sequence of method calls for the CUT and thus, perform intra-class unit testing.

To enable intra-method testing, we modified the algorithm described in Algorithm 1. With the current formulation, the insertion loop (from line 4 to line 10 in Algorithm 1) has two stopping conditions: either a maximum number of attempts or the maximum length L of the test is reached. We defined a third stopping criterion: as soon as a statement s_i representing a method invocation on a CUT object is inserted, we considered the test as complete. To store this information, in our implementation each test T has a property T_c, initially set to `false`, that indicates whether such a statement s_i has been inserted in T.

Therefore, we added $not(T_c)$ as additional stopping criterion for the insertion loop at line 4 of Algorithm 1. It is worth remarking that insertions of CUT instance methods are managed by the INSERT-CALL-ON-CUT procedure (line 7 of Algorithm 1). Thus, we re-implemented such a procedure to handle the newly defined stopping criterion.

Algorithm 3 shows our ad-hoc implementation of the INSERT-CALL-ON-CUT procedure. The algorithm takes as input a test T with $1 \leq n < L$ statements and a set $S \subseteq \langle \mathcal{M}_{CUT} \cup \mathcal{F}_{CUT} \rangle$ of setters for the CUT. For a class c, S is composed of all its instance fields $\mathcal{F}$ and of a subset of its instance methods $\mathcal{M}$. We defined the following heuristic to detect the instance method $\in S$ for the CUT. We considered as setter every $m \in \mathcal{M}$ whose method name has the $\langle \text{prefix} \rangle \langle \text{keyword} \rangle \langle \text{suffix} \rangle$ structure, with keyword $\in \{\text{set}, \text{get}, \text{put}\}$, if and only if $\exists\, m' \in M \mid \langle \text{keyword} \rangle' == \text{get}$ and $\langle \text{prefix} \rangle' == \langle \text{prefix} \rangle$ & $\langle \text{suffix} \rangle' == \langle \text{suffix} \rangle$. It is worth noting that the $\langle prefix \rangle$ part of the method name is optional. For instance, let consider the class `SimpleNode` of the JMCA project: this has two instance methods named `jjtSetParent` and `jjtGetParent`. According to our heuristic, the method `jjtSetParent` is considered as a setter method of the class `SimpleNode`.

The first step for generating a random call on the CUT is to extract a random call o in the set $\{\mathcal{C}, \mathcal{M}, \mathcal{F}\}$. This is done by the GET-RANDOM-TEST-CALL procedure (line 2 of Algorithm 3). If $o \in \{\mathcal{C} \cup \mathcal{F}\}$, a new statement s_i including a call to o is inserted into the test (as described in Section 2). In case $o \in M$—with a certain probability (set as property to 0.3 by default)—a new statement with a randomly selected setter is generated and inserted into T; therefore, the test is returned (lines from 4 to 6 in Algorithm 3). In the opposite case, o is added to the test T and its property T_c is set to `true` (lines 7 and 8 of Algorithm 3). As a consequence, the code-generation engine stops attempting new insertions: T_c is now `true` and the condition $not(T_c)$ is not met anymore. Our implementation of GET-RANDOM-TEST-CALL enables intra-method testing since it allows by design the invocation of a single instance method of the CUT. Note that our formulation does not consider setters as units under test since they are needed only to set the state of the CUT object required to properly exercise the method under test.

3.2 Step II - Intra-Class Tests Generation

The procedure described so far generate intra-method test cases, each of them targeting individual methods of the class under test. To better understand the following, intra-class test generation step, let us reason on the outcome of the intra-method testing and the implications it has.

A production method may have one or multiple branches, with each predicate of a branch being either *true* or *false*. In the case a production method has a single branch and this is fully covered during the intra-method testing procedure, this means that G-MOSA has been able to generate two unit tests that were able to verify both true and false predicates of the branch. In this situation, coverage testing would indicate the branch as covered, hence suggesting that no further test cases are required. As our approach exploits the concepts of coverage testing, methods in this category would not be considered further in the intra-class testing generation phase.

On the contrary, if a production method has branches that were not covered yet or branches not fully covered in the intra-method testing phase, this means that G-Mosa was unable to generate an appropriate number of test cases for the method: this might be either caused by (i) the inability of our approach to cover a branch or a predicate thereof or (ii) the necessity to generate more complex test cases that let the methods of the production class interact. As such, any branches that remained uncovered following the intra-method testing process were subsequently given as input for the second phase of the generation (i.e., intra-class testing), where we let the baseline MOSA work without any constraint on the amount of method calls that the test should contain. This step allows our approach to keep generating test cases for the production methods of the class under test in an effort to further increase the overall branch coverage obtained and generate tests that may be able to identify defects caused by the interaction of multiple method calls.

From an algorithmic standpoint, the GENERATE-TESTS procedure returns a set of generated tests (T_α) and a set of uncovered targets $B_\alpha \subseteq B$, where (i) T_α represents the set of intra-method test cases generated at the first step and (ii) B_α represents the production code branches that were not successfully covered within the first part of the generation process.

If $B_\alpha == \emptyset$, the intra-method testing process achieved full coverage on the CUT and T_α is returned (lines 6-7 of Algorithm 2). In the opposite case, T_α is added to T (line 8 of Algorithm 2). In the second step, Mosa is selected as algorithm for the search. This time, B_α is given as set of target to Mosa (line 9 of Algorithm 2). In other words, Mosa will attempt to cover only the targets that have not been covered in the first step. At the end of the GENERATE-TESTS procedure, the resulting T_γ is added to T and the final test suite T is returned (lines 9-10 of Algorithm 2). T is formed by two different kinds of tests: T_α generated by the intra-method process, that tests single production methods in isolation and T_γ generated by Mosa, that exercise a class by constructing sequences of method calls.

4 Research Questions and Objectives

The primary goal of the proposed approach is that of improving the structure and quality of the automatically generated test cases. As such, the ultimate *goal* of the empirical study is to analyze the quality implications of G-Mosa in terms of size, maintainability, and understandability, with the *purpose* of understanding how our approach can generate higher-quality unit test cases when compared to a state of the art automatic test case generation technique like Mosa. To address our goal, we set up three *research questions* (**RQ**s).

Before assessing the quality implications of G-Mosa, we target one of the risks associated with the mechanisms implemented within our approach that might have impacted its actual usefulness. By design, G-Mosa forces the generation of intra-method tests, possibly limiting its scope and lowering the number of tangentially covered branches. As a consequence, both code and mutation coverage might have been impacted. Should this be the case, our approach might be considered poorly useful in practice, as the improvement of test quality would be accompanied by a decrease of effectiveness. As such, we first assess the level of code and mutation coverage achieved and, only after verifying that our approach does not compromise them, we proceed with the analysis of additional perspectives. Our first **RQ** can therefore be seen as preliminary and instrumental to the quality analysis: it aims at comparing the effectiveness of test suites generated by G-Mosa and Mosa [51]. We consider Mosa as baseline because (1) previous techniques aimed at improving the quality of generated tests were compared to Mosa as well (*e.g.*, [50]) and (2) we built G-Mosa on top of Mosa, making the comparison required. We define the following research question:

RQ$_1$ - Effectiveness. *How does* G-Mosa *compare to MOSA in terms of branch and mutation coverage?*

Once assessed the implications of G-Mosa for the effectiveness of test cases, we investigate the potential benefits given by our technique. We take into account the size of the generated test cases: according to previous research in the field [51, 21, 32], this is an indicator that has been often used to estimate the effort that developers would spend to comprehend and interact with the tests, indeed, a number of previously proposed search-based automatic test case generation approaches used it as a metric to optimize [52, 49, 59]. Also in this case, we compare the size of test cases generated by G-Mosa and Mosa, addressing the following **RQ**:

RQ$_2$ - Size. *How does* G-Mosa *compare to MOSA in terms of test case size?*

While the size assessment could already provide insights into the comprehensibility of the generated test cases, in the context of our research we provide additional analyses to assess their potential usefulness from a maintainability perspective. In particular, once generated, test cases not only need to be manually validated by testers to verify assertions [1, 9], but also maintained to keep them updated as a consequence of the changes to the production code [50]. Hence, it is reasonable to assess the capabilities of our approach in this respect. We compare G-Mosa and Mosa in terms of the metrics that have been previously designed to describe the quality and maintainability of test cases and that we have surveyed in our previous work [56]. These pertain to (1) code complexity, as measured by the weighted method count of a test suite

[67]; (2) fan-out [37]; and (3) test smells, i.e., suboptimal design or implementation choices applied when developing test cases [27]. This lead to our third research question:

RQ$_3$ - Maintainability. *How does* G-MOSA *compare to MOSA in terms of maintainability of test cases?*

On the one hand, the quantitative measurements computed so far can provide a multifaceted view of how the proposed approach compares to state of the art in terms of performance. On the other hand, these analyses cannot quantify the actual gain given by G-MOSA in practice. For this reason, the last step of our methodology includes a user study where we inquiry developers on the understandability of the test cases output by G-MOSA when compared to those of MOSA. This leads to the formulation of our last research question:

RQ$_4$ - Understandability. *How does* G-MOSA *compare to MOSA in terms of understandability of test cases?*

5 Study Design

To answer our research questions, we aim to perform an empirical study on Java classes comparing G-MOSA to MOSA [51]. This section reports details about the experimental procedure planned to address our **RQs.**

5.1 Experimental Environment

We run G-MOSA and MOSA against a dataset of Java classes, collecting the generated tests and the corresponding code coverage indicators. In particular, we consider around 100 classes pertaining to the SF110 corpus [22]. This benchmark[2] contains a set of Java classes extracted from 110 projects of the SOURCEFORGE repository. We select it since this is typically used in automatic test case generation research [22, 51, 31, 21] and, therefore, can allow us to experiment our technique on a "standard" benchmark that would enable other researchers to build upon our findings and compare other techniques. As part of our online appendix [5], we provide a table reporting the name of the classes considered in our study—for the sake of readability, we could not report it in the paper. These classes are associated with a unique identifier (column "ID") that we use when reporting the results. In this stage, nine of those classes led the approaches to crash because of an internal error produced by EVOSUITE [53] and, for this reason, we had to exclude them from our analysis resulting in a final set of 91 classes.

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

To account for the intrinsic non-deterministic nature of genetic algorithms, we run each approach on each class in the dataset for 30 times, as recommended by Campos *et al.* [11]. We use the time criterion as search budget, allowing 180 seconds for the search [11]. In G-Mosa, this time is equally distributed amongst the two steps of the approach, *i.e.*, we reserve 90 seconds for intra-method and 90 for intra-class testing. Mosa could instead rely on the entire search budget to generate tests, as it does not have multiple steps.

To run the experimented approaches, we rely on the default parameter configuration given by Evosuite. As shown by Arcuri and Fraser [7], the parameter tuning process is long and expensive, other than not necessarily paying off in the end.

5.2 Collecting Performance Metrics

In the context of $\mathbf{RQ}_1$, we rely on code and mutation coverage. We select branch coverage to measure the proportion of a program's source code branches that is executed when a specific set of test cases is run. More specifically, a branch is defined as a code instruction, e.g., an `if` statement, that may cause a program to begin executing a different sequence of instructions based on the verification of a certain condition. The branch coverage is instead computed by dividing the number of branches executed by the code included within a test suite over the total number of branches available in the production code under test. As for mutation coverage, this is a metric that estimates the effectiveness of test suites in detecting the so-called *mutants*, namely artificial defects purposely introduced into the production code through small modifications (i.e., mutations) aiming at altering its original behavior. The metric is computed by dividing the number of mutants detected by the test suite over the total number of mutants within the production code under test. To compute these two metrics, we rely on the code and mutation coverage analysis engine of Evosuite [24]. We let the tool collect the branch coverage of each test in each of the 30 runs. Additionally, the tool also collects information on the mutation score: despite the existence of other tools able to perform mutation analysis (*e.g.*, PiTest[3]), we rely on the one made by Evosuite since it can effectively represent real defects [24] and has been used in a series of recent studies on automatic test case generation [30, 53, 54]. We perform the mutation analysis at the end of the search, once the unit tests have been generated for all the approaches. To obtain meaningful results we give an extra-budget of 5 minutes to the mutation analysis—this step is required to generate more mutants and to verify the ability of tests to capture them [24].

As for $\mathbf{RQ}_2$, we start from the set of test suites output by the search process for the two experimented approaches and first compute their overall size, *i.e.*, the lines of code of the generated test classes. As shown by previous work in the field [21, 53], this metric represents an indicator of the usability of the test

[3]The PiTest analyzer: `https://pitest.org`.

suites given by the tools. While recognizing the value of this perspective, we also know that such a validation could be excessively unfair in our case. By design, G-Mosa aims at creating a larger amount of test cases with respect to Mosa, with a first set of many small tests implementing the concept of intra-method testing and a second set composed of larger tests that implement the concept of intra-class testing. On the contrary, Mosa does not explicitly target the creation of maintainable test cases, hence possibly generating a fewer amount of tests that account for a lower overall test suite size while reaching high branch coverage. As a consequence, the assessment of the overall test suite size could be too simplistic, other than providing coarse-grained considerations on the usefulness of test suites, *i.e.*, in practice, developers rarely look at the *entire* test suite while fixing defects [12]. Hence, we aim to complement the overall test suite size assessment with an analysis of the properties of the individual test cases: we compute the *mean size per test case*, namely the average amount of lines of code of the automatically generated test cases within a test suite. Such a measurement can allow us to verify whether our approach could provide developers with smaller units that might better align to the actual effort required by a developer to deal with the tests generated by G-Mosa when compared to our baseline Mosa [12].

To answer our third research question (**RQ**$_3$), we compute three metrics which have been previously associated with maintainability and that might affect the way developers interact with test cases [65, 56, 32, 33]. Weighted Method Count of a Test Suite (TWMC) [67] represents a complexity metric whose computation implies the sum of the complexity values of the individual test methods of a test class. The metric provides an estimation of how complex a test class would be to understand for a developer [16, 33]. We compute TWMC as the sum of the cyclomatic complexity of all test cases in a test suite.. In the second place, we compute the fan-out metric [37], which provides an estimation of outgoing dependencies of the test cases in a test suite. It quantifies the number of dependencies that exist between a module/class and other modules/classes. Keeping coupling under control is a key concern when writing test cases, as an excessive dependence among tests might potentially lead to some sort of flakiness [34]. Finally, we detect the number of test smells per test suite: these smells have been often associated to the decrease of maintainability and effectiveness of test suites [65, 31] and likely represent the most suitable maintainability aspect to verify within the test code. In this respect, it is worth remarking that automatically generated test code is by design affected by certain test smells: for instance, the generated tests come without assertion messages and, therefore, are naturally affected by the smell known as *Assertion Roulette* [27], which arises when a test has no documented assertions. At the same time, automatic tests might not suffer from other types of smells. For example, external resources are mocked by the Evosuite framework, making the emergence of a test smell like *Mystery Guest* [27]—which has to do with the use of external resources—not possible. As such, comparing the experimented approaches based on the presence of these smells would not make sense. Hence, we only consider the test smells whose presence can be actually measured. Specifically we computed the following test smells:

– *Eager Test:* occurs when a test case tries to cover multiple scenarios or test multiple functionalities in one go instead of being focused on a specific behavior or functionality of the system under test.
– *General Fixture:* occurs when a test case relies on a common setup or configuration for multiple test scenarios, making it difficult to isolate and identify specific issues in the system.
– *Lazy Test:* occurs when a test case does not adequately cover all possible scenarios or behaviors of the system under test, leading to potential gaps in test coverage and inadequate identification of issues or bugs.
– *Sensitive Equality:* occurs when a test case compares two values using direct equality checks, such as equals(), without considering the possible tolerance for slight variations in values.
– *Indirect Testing:* occurs when a test case indirectly tests the functionality of the system under test by relying on the behavior of other components or dependencies.

In more practical terms, we employ the tool by Spinellis [66] to compute TWMC and EC metrics. As for test smells, we rely on TsDetect [57], which is a tool able to identify more than 25 different types of test smells—in this case, however, we limit the detection to the test smells that might actually arise in automatically generated tests.

5.3 Collecting Understandability Metrics

The last step of our experimentation concerns with the assessment of the actual gain provided by G-Mosa in practice. We therefore conducted an online experiment where we (1) involved developers in tasks connected to the understandability of the test cases generated by our approach and (2) compared our approach with the baseline Mosa.

Experimental setting. We designed a user study that allowed participants to first provide demographics information and then provide indications about the level of understandability of the test classes generated by the two approaches compared, i.e., G-Mosa and Mosa. To run the experiment, we used an online platform we have recently developed, which allows external participants to (1) navigate and interact with source code elements and (2) answer closed and open questions.

More specifically, the participants were first asked to answer demographic questions that will serve to address their background and level of expertise in software development and testing. We also inquired them about the type of development they use to do, e.g., whether they consider themselves as industrial or open-source developers. In addition, we asked to report how frequently the participants are involved in unit testing tasks with respect to other types of testing activities: in this way, we could assess the suitability of participants with the goal of our study, which was to assess the maintainability/understandability of unit test classes.

In the second place, participants were asked to perform the same task twice. They were provided with the source code of two Java test classes aiming at exercising the same production class. One of them generated by G-Mosa and the other one by Mosa. In each task, after reading each of the two test classes participants were asked to (1) rate the overall understandability of the class with a 5-points Likert scale (from 1, which indicates poorly understandable code, to 5, which indicates fully understandable code); (2) explain the reasons for the rating provided; (3) write the assertion and corresponding assertion message for two methods randomly selected from the test class under consideration. While the responses to the first two questions were used to assess the perceived understandability of test cases, the responses to the last question were used to verify the validity of the assertions produced by developers.

Table 1 User study configurations. The class ID refers to the table with all the considered classes reported in our online apppendix [5].

Configuration ID	1st Treatment	2nd Treatment	Class ID
Configuration 1	Mosa	G-Mosa	Class #5
Configuration 2	G-Mosa	Mosa	Class #9
Configuration 3	Mosa	G-Mosa	Class #2
Configuration 4	G-Mosa	Mosa	Class #34

The pairs of test classes were randomly selected from the dataset employed to address the previous research questions. We selected 4 pairs of test classes and prepared 4 different configurations of the study (one for each class). This was done to avoid biased interpretations of the results due to specific characteristics of a selected class. We had to limit the scope of the study to few classes in order to preserve the compromise between having enough information to address $\mathbf{RQ}_4$ and design a *short-enough* user study that allowed the participation of a large amount of developers—and that, therefore, would have allowed us to draw statistically significant conclusions. It is worth remarking that the choice of selecting four pairs of test classes for the user study was not random, but driven by the results of a pilot study, which revealed that this amount of test classes was the optimal choice for the kind of assessment participants should have done. More details on the pilot study and the results obtained are discussed in Section 7.4.

As for the order of the test classes, half of the participants first engaged with a test class generated by Mosa and then with the one generated by G-Mosa. Conversely, the other half of the participants read the two test classes in the reverse order. Through the experiment, we assessed the extent to which developers can understand and deal with the information provided by the test cases generated by the two approaches. Table 1 reports an overview of the four resulting user study configurations.

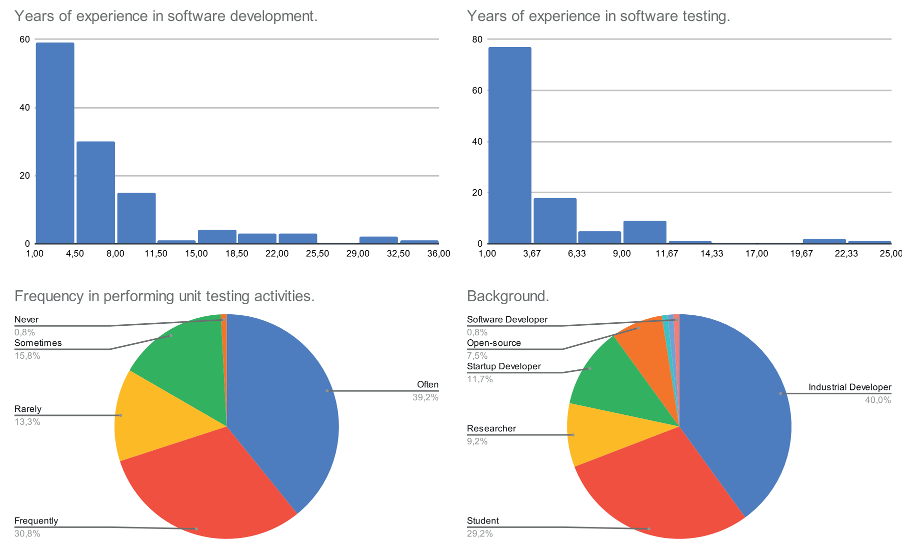
Participant's recruitment. We recruited developers using various channels. In the first place, we invited the original open-source developers of the classes considered in the study. This has been done via e-mail. Of course, we only approached the developers who have publicly released their e-mail address on GitHub. In a complementary manner, we recruited participants through Prolific[4] by carefully considering the guidelines recently proposed by Reid et al. [61]. This is a research-oriented web-based platform that enables researchers to find participants for user studies: in particular, it allowed to preset the desired number of responses (in our case, 140) and automatically closes the survey once this target is met—because of these characteristics, it would not be accurate to report a response rate. One of the features of Prolific is the specification of constraints over participants, which in our case enabled to limit the participation to software developers that are knowledgeable about Java development and unit testing. It is important to point out that Prolific implements an *opt-in* strategy [38], meaning that participants get voluntarily involved. This might potentially lead to self-selection or voluntary response bias [36]. To mitigate this risk, we introduced an incentive of 2 pounds per valid respondent. Once we received the answers, we filtered out the answers coming from participants who did not take the task seriously—this was done by manually validating the answers received, looking for cases where participants clearly replied to questions in a shallow manner or just for the sake of getting the experiment done within the lowest amount of time possible. Overall, we discarded 20 responses out of the 140 received.

We could rely on a total of 120 valid responses. Unfortunately, we did not receive any reply from the original developers (response rate=0%)—this was likely due to a reflection of the issues raising when involving developers from GitHub, who are typically overwhelmed by requests coming from researchers and which, because of that, are less and less prone to be involved [8]. On the contrary, we could get a notable amount of answers from developers contributing to Prolific. Figure 1 reports the background of the respondents, as self-assessed by themselves when filling the survey out. As shown, they indicated a programming experience between 1 and 35 years and an experience with unit testing ranging between 1 and 24 years. Perhaps more importantly, 70% of the participants reported that they often or frequently conduct unit testing activities, hence being qualified enough to take part of our study. In addition, most of the participants were industrial developers (40%).

From the analysis of the background information reported by our participants, we could conclude that our sample was mainly composed of industrial developers with a solid knowledge of unit testing and that perform such an activity quite often during their daily work activities. As such, we deemed the sample valid for the goals of our study.

[4]Prolific website: `https://www.prolific.co/`.

Fig. 1 Background of survey respondents.



5.4 Data Analysis

After collecting the metrics, we ran statistical tests to verify whether the differences observed between G-Mosa and Mosa are statistically significant. More specifically, we employed the non-parametric Wilcoxon Rank Sum Test [13] (with $\alpha == 0.05$) on the distributions of (1) code coverage, (2) mutation coverage, (3) size per test case, (4) weighted method count of a test suite, (5) fan-out, (6) number of test smells, and (7) understandability scores assigned by developers in the user study. In this respect, we formulated the following null hypotheses:

Hn 1. There is *no significant difference* in terms of *branch coverage* achieved by G-Mosa and MOSA.

Hn 2. There is *no significant difference* in terms of *mutation coverage* achieved by G-Mosa and Mosa.

Hn 3. There is *no significant difference* in terms of *size per unit* achieved by G-Mosa and Mosa.

Hn 4. There is *no significant difference* in terms of *weighted method count of a test suite* achieved by G-Mosa and Mosa.

Hn 5. There is *no significant difference* in terms of *fan-out* achieved by G-Mosa and Mosa.

Hn 6. There is *no significant difference* in terms of the *number of test smells* achieved by G-Mosa and Mosa.

Hn 7. There is *no significant difference* in terms of the *understandability scores* achieved by G-Mosa and Mosa.

From a statistical perspective, we have to take into account the fact that, if one of the null hypothesis is rejected, then one between G-Mosa and Mosa is statistically better than the other. Hence, we defined a set of alternative hypotheses such as the following:

An 1. The *branch coverage* achieved by G-Mosa and MOSA is *statistically different.*

An 2. The *mutation coverage* achieved by G-Mosa and Mosa is *statistically different.*

An 3. The *size per unit* of the unit test suites generated by G-Mosa and Mosa is *statistically different.*

An 4. The *weighted method count of a test suite* of the unit test suites generated by G-Mosa and Mosa is *statistically different.*

An 5. The *fan-out* of the unit test suites generated by G-Mosa and Mosa is *statistically different.*

An 6. The *number of test smells* of the unit test suites generated by G-Mosa and Mosa is *statistically different.*

An 7. The *understandability scores* of the unit test suites generated by G-Mosa and Mosa is *statistically different.*

We reject the null hypotheses if $Hn_i \iff p < 0.05$. In addition to the Wilcoxon Rank Sum Test, we rely on the Vargha-Delaney ($\hat{A}_{12}$) [68] statistical test to measure the magnitude of the differences in the distributions of the considered metrics. Based on the direction given by $\hat{A}_{12}$, we can make a practical sense to the alternative hypotheses. Should the $\hat{A}_{12}$ values be lower than 0.5, this would denote that the test suites generated by G-Mosa would be better than those provided by Mosa. For instance, a $\hat{A}_{12} < 0.50$ in the distribution of code coverage would indicate that the code coverage achieved by G-Mosa is higher than the one reached by the baseline. Similarly, a $\hat{A}_{12} > 0.50$ indicates the opposite, while $\hat{A}_{12} == 0.50$ points out that the results are identical.

```
public void testX() throws Throwable {
  Configuration configuration0 =
      Configuration.getSystemConfiguration(true);
  ConnectionConsumer<String> connectionConsumer0 = new
      ConnectionConsumer<String>(configuration0,
      "summa.configuration");
  connectionConsumer0.releaseConnection();
  .....
}
```

Listing 2 A test method, for which participants have provided both *valid* and *not valid* assertions.

Besides the statistical analysis of the distributions collected in our empirical study, we also proceeded with the verification of the assertions and assertion messages written by the user study participants. The first two authors of the paper acted as *inspectors* and assessed whether the reported assertions were in line with the actual behavior of the test cases. The two inspectors jointly performed the task in an effort of having two expert opinions on the validity of the assertions analyzed and immediately discuss and solve possible cases of disagreements. In the verification process, the inspectors exploited two main pieces of information: (1) the assertion message left by participants, which explained the rationale behind the assertion and the condition that the assertion was aimed at addressing; and (2) the path covered by the test, as indicated by JaCoCo, i.e., a code coverage analysis tool, which helped assess the match between the assertion, the assertion message, and the goals of the test case. Through these pieces of information, the inspectors marked an assertion as "valid" if it correctly captured the condition verified by the test case, "not valid" otherwise. t To better understand this criteria, let's examine the test method presented in Listing 2. This was one of the test methods utilized in our survey to solicit assertions from participants. For this test case, a participant reported the following assertion: "*assertNull('The connection should be null after calling releaseConnection', connectionConsumer0.getConnection());*". This case was considered "valid" because the assertion effectively verifies the intended outcome in this specific test case.

In contrast, another survey respondent provided the following assertion: "*assertNotNull(configuration0);*". Since "*configuration0*" is merely a variable used in the test case to instantiate the connection consumer and the goal of the test is not to determine whether this variable is null or not, this case was marked as "not valid." In addition, we made use of the free answers provided by participants when explaining the reasons for the understandability score (question #2 of the task) to identify the reasons for the correct/wrong assertion definitions. We finally provided an overview of the (dis)advantages of each test case generation tool with respect to the understandability of the resulting test cases.

5.5 Publication of generated data

G-Mosa source code, as well as all the other data generated from our study are publicly available in our online appendix [5].

We also released the scripts to automatically generate the test suites, other than the data collected and used for the statistical and content analysis that we present in the paper.

6 Analysis of the Results

This section discusses the results achieved while addressing our three research questions.

Table 2 Branch coverage achieved by Mosa and G-Mosa, with p-values resulting from the Wilcoxon test and Vargha-Delaney $A_1 2$ effect size. We use N, S, M, and L to indicate negligible, small, medium and large effect size respectively. Significant p-values are reported in bold-face.

Branch Coverage									
	Average		Mosa vs. G-Mosa			Average		Mosa vs. G-Mosa	
ID	Mosa	G-Mosa	p-value	$\hat{A}_{12}$	ID	Mosa	G-Mosa	p-value	$\hat{A}_{12}$
1	0.08	0.08	1.00	0.48 (N)	47	0.95	0.91	**0.01**	0.67 (M)
2	0.37	0.17	**<0.01**	0.80 (L)	48	0.70	0.56	0.14	0.61 (S)
3	0.86	0.84	0.08	0.62 (S)	49	0.94	0.94	0.71	0.53 (N)
4	0.23	0.24	0.09	0.40 (S)	50	0.91	0.91	1.00	0.52 (N)
5	0.88	0.89	**<0.01**	0.33 (M)	51	0.43	0.42	0.11	0.53 (N)
6	0.03	0.03	1.00	0.52 (N)	52	0.92	0.90	0.13	0.63 (S)
7	0.88	0.85	0.14	0.60 (S)	53	0.07	0.07	1.00	0.52 (N)
8	1.00	1.00	1.00	0.52 (N)	54	0.97	0.96	0.13	0.60 (S)
9	1.00	0.97	**<0.01**	0.77 (L)	55	0.87	0.84	0.08	0.60 (S)
10	0.93	0.91	0.15	0.60 (S)	56	0.47	0.46	0.58	0.54 (N)
11	0.61	0.61	0.48	0.58 (S)	57	0.77	0.68	**<0.01**	0.87 (L)
12	0.22	0.22	0.46	0.55 (N)	58	0.83	0.83	1.00	0.53 (N)
13	0.12	0.11	0.69	0.55 (N)	59	0.93	0.93	0.52	0.56 (N)
14	0.52	0.52	0.19	0.58 (S)	60	0.93	0.93	0.41	0.58 (S)
15	0.98	0.94	**<0.01**	0.86 (L)	61	1.00	1.00	1.00	0.52 (N)
16	0.32	0.35	**<0.01**	0.20 (L)	62	0.83	0.79	0.06	0.63 (S)
17	0.95	0.97	**<0.01**	0.31 (M)	63	1.00	1.00	1.00	0.52 (N)
18	0.76	0.73	**0.04**	0.63 (S)	64	0.73	0.68	**0.04**	0.63 (S)
19	0.69	0.72	0.50	0.47 (N)	65	0.08	0.08	1.00	0.50 (N)
20	0.05	0.05	1.00	0.50 (N)	66	0.17	0.17	1.00	0.52 (N)
21	0.09	0.09	1.00	0.48 (N)	67	0.09	0.08	**<0.01**	0.75 (L)
22	1.00	1.00	1.00	0.50 (N)	68	0.51	0.50	0.07	0.63 (S)
23	0.22	0.22	0.76	0.54 (N)	69	0.44	0.43	**0.01**	0.65 (S)
24	0.96	0.96	0.49	0.47 (N)	70	0.78	0.77	0.76	0.56 (N)
25	0.98	0.95	**0.03**	0.65 (S)	71	0.14	0.14	1.00	0.52 (N)
26	1.00	1.00	0.49	0.50 (N)	72	0.92	0.93	0.68	0.49 (N)
27	0.87	0.84	**0.01**	0.69 (M)	73	0.01	0.01	1.00	0.50 (N)
28	0.21	0.22	0.64	0.47 (N)	74	0.76	0.76	1.00	0.50 (N)
29	0.03	0.03	0.49	0.47 (N)	75	0.68	0.65	**<0.01**	0.65 (S)
30	1.00	1.00	1.00	0.53 (N)	76	1.00	1.00	1.00	0.47 (N)
31	0.97	0.97	0.49	0.50 (N)	77	1.00	1.00	1.00	0.53 (N)
32	0.78	0.77	0.05	0.62 (S)	78	1.00	0.93	**0.02**	0.66 (S)
33	0.23	0.25	0.19	0.47 (N)	79	0.91	0.88	**0.01**	0.64 (S)
34	0.01	0.02	**0.04**	0.37 (S)	80	1.00	1.00	1.00	0.48 (N)
35	0.76	0.78	0.08	0.46 (N)	81	0.18	0.17	0.06	0.66 (M)
36	0.98	0.98	0.46	0.54 (N)	82	0.02	0.02	1.00	0.50 (N)
37	0.96	0.97	0.49	0.47 (N)	83	0.10	0.10	1.00	0.50 (N)
38	0.02	0.03	0.58	0.45 (N)	84	0.14	0.14	0.89	0.54 (N)
39	0.70	0.74	0.12	0.37 (S)	85	0.68	0.62	**<0.01**	0.78 (L)
40	0.83	0.85	**0.03**	0.34 (S)	86	0.00	0.00	1.00	0.50 (N)
41	0.04	0.04	**<0.01**	0.62 (S)	87	0.66	0.63	0.16	0.53 (N)
42	0.17	0.17	1.00	0.50 (N)	88	0.92	0.92	0.78	0.43 (N)
43	0.35	0.36	0.96	0.52 (N)	89	0.36	0.36	1.00	0.47 (N)
44	0.77	0.77	0.49	0.50 (N)	90	0.74	0.74	1.00	0.50 (N)
45	0.91	0.87	**0.02**	0.69 (M)	91	0.60	0.58	**<0.01**	0.74 (M)
46	0.39	0.39	1.00	0.50 (N)					

6.1 $\mathbf{RQ}_1$ - Effectiveness

We addressed $\mathbf{RQ}_1$ by comparing G-Mosa and Mosa effectiveness in terms of branch and mutation coverage. As for the former, Table 2 reports the average branch coverage achieved by the two experimented techniques over 30 independent runs as well as the results of the Wilcoxon and the Vargha-Delaney tests. Just looking at the average, the results seem to indicate that G-Mosa

Table 3 Mutation score achieved by Mosa and G-Mosa, with p-values resulting from the Wilcoxon test and Vargha-Delaney A_12 effect size. We use N, S, M, and L to indicate negligible, small, medium and large effect size respectively. Significant p-values are reported in bold-face.

Mutation Score									
	Average		Mosa vs. G-Mosa			Average		Mosa vs. G-Mosa	
ID	**Mosa**	**G-Mosa**	**p-value**	A_{12}	**ID**	**Mosa**	**G-Mosa**	**p-value**	A_{12}
1	0.00	0.00	1.00	0.48 (N)	47	0.13	0.14	**<0.01**	0.35 (S)
2	0.01	0.00	1.00	0.50 (N)	48	0.47	0.26	0.06	0.64 (S)
3	0.37	0.31	0.91	0.51 (N)	49	0.53	0.54	0.26	0.41 (S)
4	0.01	0.01	0.19	0.40 (S)	50	0.00	0.00	1.00	0.52 (N)
5	0.46	0.47	0.64	0.47 (N)	51	0.57	0.55	0.18	0.43 (N)
6	0.00	0.00	1.00	0.52 (N)	52	0.61	0.61	0.43	0.58 (S)
7	0.48	0.41	0.28	0.58 (S)	53	0.00	0.00	1.00	0.52 (N)
8	0.74	0.74	1.00	0.52 (N)	54	0.17	0.15	0.43	0.56 (N)
9	0.24	0.23	0.30	0.59 (S)	55	0.32	0.32	0.17	0.57 (N)
10	0.74	0.74	0.61	0.54 (N)	56	0.05	0.05	1.00	0.50 (N)
11	0.19	0.18	0.53	0.58 (S)	57	0.44	0.44	0.51	0.45 (N)
12	0.07	0.07	0.92	0.51 (N)	58	0.55	0.50	**<0.01**	0.79 (L)
13	0.03	0.04	0.34	0.45 (N)	59	0.48	0.48	0.70	0.55 (N)
14	0.78	0.78	1.00	0.52 (N)	60	0.66	0.67	0.89	0.53 (N)
15	0.71	0.74	0.62	0.54 (N)	61	0.51	0.53	0.21	0.45 (N)
16	0.05	0.05	0.86	0.51 (N)	62	0.64	0.65	0.42	0.45 (N)
17	0.52	0.54	0.50	0.43 (N)	63	0.28	0.28	1.00	0.52 (N)
18	0.28	0.28	0.43	0.44 (N)	64	0.42	0.39	0.05	0.63 (S)
19	0.97	1.00	0.49	0.50 (N)	65	0.00	0.00	1.00	0.50 (N)
20	0.00	0.00	1.00	0.50 (N)	66	0.06	0.06	1.00	0.52 (N)
21	0.00	0.00	1.00	0.48 (N)	67	0.04	0.03	**<0.01**	0.78 (L)
22	0.53	0.53	1.00	0.50 (N)	68	0.27	0.27	0.63	0.54 (N)
23	0.08	0.11	**<0.01**	0.33 (M)	69	0.37	0.37	0.33	0.58 (S)
24	0.63	0.63	0.21	0.41 (S)	70	0.30	0.33	**<0.01**	0.12 (L)
25	0.20	0.20	1.00	0.50 (N)	71	0.00	0.00	1.00	0.52 (N)
26	0.15	0.18	**<0.01**	0.23 (L)	72	0.37	0.38	0.19	0.45 (N)
27	0.25	0.24	**0.03**	0.67 (M)	73	0.00	0.00	1.00	0.50 (N)
28	0.01	0.00	1.00	0.52 (N)	74	0.02	0.02	1.00	0.50 (N)
29	0.05	0.05	0.73	0.47 (N)	75	0.33	0.33	**0.02**	0.57 (N)
30	0.70	0.70	0.98	0.53 (N)	76	0.73	0.74	0.07	0.36 (S)
31	0.70	0.74	**<0.01**	0.30 (M)	77	0.31	0.36	**<0.01**	0.26 (L)
32	0.53	0.53	1.00	0.50 (N)	78	0.48	0.54	0.26	0.48 (N)
33	0.00	0.00	1.00	0.52 (N)	79	0.66	0.62	0.30	0.57 (N)
34	0.00	0.00	1.00	0.50 (N)	80	0.21	0.25	**0.04**	0.38 (S)
35	0.08	0.08	**<0.01**	0.23 (L)	81	0.09	0.08	0.50	0.58 (S)
36	0.49	0.50	0.48	0.43 (N)	82	0.00	0.00	1.00	0.50 (N)
37	0.71	0.74	**<0.01**	0.10 (L)	83	0.00	0.00	1.00	0.50 (N)
38	0.00	0.00	1.00	0.50 (N)	84	0.00	0.00	1.00	0.55 (N)
39	0.37	0.39	0.15	0.38 (S)	85	0.11	0.10	**<0.01**	0.79 (L)
40	0.57	0.58	0.12	0.37 (S)	86	0.00	0.00	1.00	0.50 (N)
41	0.00	0.00	**0.04**	0.54 (N)	87	0.32	0.41	**<0.01**	0.19 (L)
42	0.00	0.00	1.00	0.50 (N)	88	0.67	0.66	0.94	0.46 (N)
43	0.00	0.00	1.00	0.52 (N)	89	0.20	0.20	0.59	0.52 (N)
44	0.46	0.46	0.70	0.48 (N)	90	0.28	0.28	0.78	0.47 (N)
45	0.38	0.37	0.80	0.54 (N)	91	0.36	0.37	0.15	0.39 (S)
46	0.01	0.00	**0.01**	0.68 (M)					

and Mosa achieve very similar performance in terms of branch coverage. Indeed, the great majority of rows show $\hat{A}_{12}$ values around 0.5, reinforcing the observation above. Only in 23 out of 91 cases, ($\approx 25\%$) there is a statistically significant difference in the performance achieved, while in the remaining 75% of cases there is no statistical difference between the branch coverage achieved by the two approaches. Based on these results, we cannot reject the null hypothesis **Hn 1**. Therefore, we can claim that there is *no statistically significant difference* in terms of *branch coverage* achieved by G-Mosa and Mosa.

Table 3 reports the average mutation score achieved by G-Mosa and Mosa together with the results of Wilcoxon and Vargha-Delaney statistical tests. The first interesting observation is that in 67 out of 91 cases ($\approx 74\%$), both approaches achieve low performance, *i.e.*, average mutation score < 0.5. While this represents a scientifically relevant result, we could not provide a detailed explanation behind the poor mutation capabilities of the experimented approaches. In particular, the mutation analysis is performed as part of the inner-working of EvoSuite, i.e., the framework G-Mosa and Mosa build upon, and is based on the application of multiple mutation operators (e.g., statement deletion) which are individually used to modify the production code under test and assess the extent to which the corresponding test case is able to detect the artificial defect introduced. Such a mutation analysis is performed multiple times for each test case considered and for each of the 30 runs of both G-Mosa and Mosa. Also, the algorithms behind the test case generators are inherently non-deterministic, which means that for each of their executions there might have been a different reason leading to miss mutants. These reasons make the mutation analysis step hardly explainable - at least until an explainable model able to work under these conditions would not be available.

Hence, we could limit ourselves to the observation of the overall mutation score obtained by the approaches, interpreting the conceptual causes of this result and the tangible implications that such a low mutation score may have.

In terms of conceptual causes, it is worth remarking that both G-Mosa and Mosa have branch coverage as main target, while they are not designed to optimize the mutation score. This may possibly explain why the good level of branch coverage is not accompanied by adequate mutation coverage. As for the implications of the low mutation coverage achieved by the approaches, our findings suggest that the automated test case generation approaches are still unable to satisfactorily detect artificial defects. This seems to be a common limitation and, in this sense, our work outlines a limitation that further research may want to address.

As for the comparison, similarly to what happened for branch coverage, there were only a few cases highlighting a clear statistical difference in the distributions of G-Mosa and Mosa. Specifically, this happened only for 17 out of 91 classes ($\approx 19\%$), 14 ($\approx 15\%$) if we exclude those with negligible or small effect size. Of these 14, 8 indicated G-Mosa as the best performing technique ($\hat{A}_{12} < 0.5$), while Mosa achieved higher performance in the remaining 6 cases ($\hat{A}_{12} > 0.5$). These results do not allow to reject the null hypothesis **Hn 2**. thus indicating that there is no *statistically significant difference* in the mutation coverage achieved by G-Mosa and Mosa.

Besides the statistical analysis, we aimed at collecting qualitative insights that could better delineate strengths and weaknesses of the devised technique. For this reason, we dived into the quantitative results and manually analyzed the classes for which the performance indicators computed revealed a significant difference, either in favor of G-Mosa or Mosa. This qualitative investigation was mainly conducted by the first author of this paper, who acted as a code inspector: the task was that of performing a *code review* of the selected classes aiming at understanding the main code quality aspects influencing the branch coverage achieved by the corresponding test cases and the differences observed in the way the two approaches generated test cases. In doing so, the inspector could rely on the metric values computed on the production classes, which supported the analysis of the code. During the review task, the inspector took notes reporting the main insights and observations coming from the analysis. These notes were later used as a basis for a larger discussion which was opened with the second and third authors of the article. More particularly, the three authors jointly navigated the source code of the classes considered and discussed on the notes of the first author, deriving insights that can be well summarized through the following three qualitative examples.

As a first discussion point, let consider the classes `org.gudy.azureus2.ui.console.commands.Show` (id. 16) and `de.progra.charting.render.PieChartRenderer` (id. 2). The former is characterized by a total number of 356 branches, thus being very complex [44]: when generating tests for such a class, G-Mosa achieved a *significantly better* branch coverage with a *large* effect size. The latter is characterized by 12 branches: unlike the previous case, Mosa performed *significantly better* with a *large* effect size. These two examples indicate that, while in most cases the two techniques perform similarly in terms of branch coverage, G-Mosa can act better when testing more complex classes. This observation could be attributed to the fact that the granular nature of G-Mosa can lead to simplifying the testing of complex classes. In the first step, all the tests that cover more fine-grained cases are generated. Therefore, in the second step, the remaining search budget is spent solely on those branches that are more difficult to cover, thus resulting in higher coverage. In other words, half of the total search budget—the one relative to step 2—is completely dedicated to covering hard targets. This trend was also confirmed when looking at other test suites of the dataset, hence potentially indicating additional capabilities of our approach. We plan to investigate this aspect further as part of our future research agenda, especially by conducting larger qualitative investigations into the peculiarities of G-Mosa.

Similar conclusions could be drawn when considering the mutation score. As an example, on the class `portlet.shopping.model.ShoppingCategoryWrapper` (id. 26), G-Mosa had a *significantly higher* mutation score with a *large* effect size. This class is characterized by 53 methods and 2,384 lines, being one of the largest in our dataset. Differently, when considering smaller classes, Mosa achieved better performance. This is the case of the class `weka.core.tokenizers.AlphabeticTokenizer` (id. 58) that only contains 7 lines of code. As such, it seems that our approach cannot only produce better results on large classes in terms of code coverage, but also in terms of mutation score.

Table 4 Lines of code in test classes generated by Mosa and G-Mosa, with p-values resulting from the Wilcoxon test and Vargha-Delaney $A_1 2$ effect size. We use N, S, M, and L to indicate negligible, small, medium and large effect size respectively. Significant p-values are reported in bold-face.

Lines of code									
	Average		Mosa vs. G-Mosa			Average		Mosa vs. G-Mosa	
ID	**Mosa**	**G-Mosa**	**p-value**	$\hat{A}_{12}$	**ID**	**Mosa**	**G-Mosa**	**p-value**	$\hat{A}_{12}$
1	9.70	13.00	<**0.01**	0.32 (M)	47	12.00	11.00	<**0.01**	1.00 (L)
2	361.48	660.79	<**0.01**	0.00 (L)	48	183.07	220.97	<**0.01**	0.01 (L)
3	50.43	59.03	<**0.01**	0.19 (L)	49	124.10	181.24	<**0.01**	0.01 (L)
4	235.71	330.27	<**0.01**	0.00 (L)	50	12.00	13.10	0.08	0.45 (N)
5	191.27	194.69	0.36	0.58 (S)	51	118.72	152.07	<**0.01**	0.00 (L)
6	178.37	174.93	0.62	0.54 (N)	52	612.73	922.50	<**0.01**	0.00 (L)
7	66.31	74.22	<**0.01**	0.20 (L)	53	164.64	176.17	0.12	0.38 (S)
8	394.86	578.60	<**0.01**	0.00 (L)	54	118.53	127.37	**0.03**	0.34 (S)
9	66.55	93.77	<**0.01**	0.15 (L)	55	824.63	1036.67	<**0.01**	0.00 (L)
10	521.97	731.90	<**0.01**	0.00 (L)	56	34.00	37.80	<**0.01**	0.03 (L)
11	228.07	248.50	<**0.01**	0.28 (M)	57	187.00	177.13	<**0.01**	0.80 (L)
12	263.20	321.67	<**0.01**	0.13 (L)	58	765.33	897.93	<**0.01**	0.01 (L)
13	45.37	56.57	<**0.01**	0.01 (L)	59	585.67	1117.17	<**0.01**	0.00 (L)
14	144.14	173.43	<**0.01**	0.01 (L)	60	35.41	46.00	<**0.01**	0.00 (L)
15	57.53	103.77	<**0.01**	0.00 (L)	61	71.90	105.93	<**0.01**	0.00 (L)
16	234.10	311.27	<**0.01**	0.01 (L)	62	209.00	240.72	<**0.01**	0.16 (L)
17	12.00	20.10	<**0.01**	0.12 (L)	63	586.33	680.17	<**0.01**	0.12 (L)
18	58.89	61.40	0.14	0.39 (S)	64	232.26	318.64	<**0.01**	0.00 (L)
19	37.13	41.33	0.06	0.36 (S)	65	65.60	107.83	<**0.01**	0.00 (L)
20	4.00	22.00	<**0.01**	0.00 (L)	66	171.10	174.10	**0.03**	0.34 (S)
21	60.81	99.53	<**0.01**	0.05 (L)	67	1117.55	1846.27	<**0.01**	0.00 (L)
22	456.55	534.80	<**0.01**	0.21 (L)	68	117.79	124.77	**0.01**	0.31 (M)
23	114.90	135.17	<**0.01**	0.04 (L)	69	572.50	578.90	0.53	0.45 (N)
24	248.00	371.92	<**0.01**	0.03 (L)	70	175.17	316.97	<**0.01**	0.00 (L)
25	754.21	1078.38	<**0.01**	0.05 (L)	71	6.00	583.60	<**0.01**	0.00 (L)
26	224.75	145.38	<**0.01**	0.95 (L)	72	702.73	826.37	<**0.01**	0.04 (L)
27	312.33	322.76	0.53	0.45 (N)	73	212.28	334.70	<**0.01**	0.00 (L)
28	448.90	344.47	<**0.01**	1.00 (L)	74	1217.43	2325.00	<**0.01**	0.00 (L)
29	11.72	12.69	0.09	0.45 (N)	75	310.14	421.34	<**0.01**	0.00 (L)
30	84.70	107.97	<**0.01**	0.06 (L)	76	252.62	166.20	<**0.01**	0.97 (L)
31	90.77	79.83	<**0.01**	0.84 (L)	77	243.31	274.33	<**0.01**	0.09 (L)
32	41.43	52.57	<**0.01**	0.05 (L)	78	46.47	69.37	<**0.01**	0.02 (L)
33	90.77	132.37	<**0.01**	0.00 (L)	79	51.14	77.83	0.07	0.36 (S)
34	764.00	821.93	0.15	0.39 (S)	80	61.93	101.20	<**0.01**	0.00 (L)
35	44.86	41.10	<**0.01**	0.83 (L)	81	19.00	17.00	<**0.01**	1.00 (L)
36	21.00	20.00	<**0.01**	1.00 (L)	82	14.00	14.00	1	0.50 (N)
37	246.40	129.78	**0.01**	0.73 (M)	83	92.87	111.40	<**0.01**	0.02 (L)
38	420.55	821.53	<**0.01**	0.00 (L)	84	351.83	238.63	<**0.01**	0.72 (M)
39	195.47	246.80	<**0.01**	0.15 (L)	85	2126.70	2210.73	<**0.01**	0.15 (L)
40	733.38	1344.70	<**0.01**	0.00 (L)	86	763.21	973.23	<**0.01**	0.03 (L)
41	113.45	139.77	<**0.01**	0.02 (L)	87	143.07	162.18	<**0.01**	0.16 (L)
42	74.33	65.55	<**0.01**	0.80 (L)	88	4517.48	3494.41	<**0.01**	0.26 (L)
43	5.10	8.93	**0.01**	0.35 (S)	89	31.87	33.93	0.74	0.47 (N)
44	4.00	75.10	**0.02**	0.42 (S)	90	12.00	14.33	**0.01**	0.38 (S)
45	5.40	7.17	0.13	0.41 (S)	91	152.04	194.79	<**0.01**	0.00 (L)
46	11.73	21.31	<**0.01**	0.03 (L)					

Summing Up: There is *no statistically significant difference* in terms of branch and mutation coverage achieved by G-Mosa and Mosa. G-Mosa seems to perform better when generating tests for more complex classes. On the contrary, Mosa achieves better coverage when testing simpler classes.

Table 5 Number of methods in test classes generated by Mosa and G-Mosa, with p-values resulting from the Wilcoxon test and Vargha-Delaney A_12 effect size. We use N, S, M, and L to indicate negligible, small, medium and large effect size respectively. Significant p-values are reported in bold-face.

Number of methods									
	Average		**Mosa vs. G-Mosa**			**Average**		**Mosa vs. G-Mosa**	
ID	**Mosa**	**G-Mosa**	**p-value**	$\hat{A}_{12}$	**ID**	**Mosa**	**G-Mosa**	**p-value**	$\hat{A}_{12}$
1	1.00	1.00	NaN	0.50 (N)	47	1.00	1.00	NaN	0.50 (N)
2	38.45	65.50	<**0.01**	0.00 (L)	48	19.07	23.83	<**0.01**	0.01 (L)
3	7.97	10.14	<**0.01**	0.09 (L)	49	15.20	21.03	<**0.01**	0.00 (L)
4	24.50	30.67	<**0.01**	0.00 (L)	50	1.00	1.10	0.08	0.45 (N)
5	12.80	13.56	0.24	0.40 (S)	51	19.28	25.43	<**0.01**	0.00 (L)
6	24.77	28.20	<**0.01**	0.15 (L)	52	65.77	92.03	<**0.01**	0.00 (L)
7	4.83	5.33	**0.01**	0.29 (M)	53	22.32	27.10	<**0.01**	0.12 (L)
8	42.79	59.83	<**0.01**	0.00 (L)	54	22.43	28.00	<**0.01**	0.00 (L)
9	3.83	5.69	<**0.01**	0.07 (L)	55	104.93	122.37	<**0.01**	0.02 (L)
10	80.43	78.28	**0.01**	0.69 (M)	56	3.90	4.00	0.33	0.48 (N)
11	33.23	35.70	<**0.01**	0.19 (L)	57	32.37	38.03	<**0.01**	0.03 (L)
12	38.13	46.50	<**0.01**	0.06 (L)	58	111.80	106.59	<**0.01**	0.86 (L)
13	4.00	4.97	<**0.01**	0.02 (L)	59	39.00	62.67	<**0.01**	0.00 (L)
14	14.59	17.83	<**0.01**	0.01 (L)	60	7.31	8.83	<**0.01**	0.04 (L)
15	5.80	9.07	<**0.01**	0.01 (L)	61	8.90	13.80	<**0.01**	0.00 (L)
16	20.60	27.93	<**0.01**	0.00 (L)	62	27.67	33.83	<**0.01**	0.00 (L)
17	1.00	1.73	<**0.01**	0.13 (L)	63	70.17	70.93	0.63	0.46 (N)
18	5.86	7.00	<**0.01**	0.17 (L)	64	27.11	37.57	<**0.01**	0.00 (L)
19	3.80	4.53	<**0.01**	0.28 (M)	65	7.97	12.72	<**0.01**	0.00 (L)
20	1.00	2.00	<**0.01**	0.00 (L)	66	22.10	27.03	<**0.01**	0.00 (L)
21	6.37	9.60	<**0.01**	0.03 (L)	67	157.34	217.67	<**0.01**	0.00 (L)
22	58.31	71.33	<**0.01**	0.04 (L)	68	15.55	19.40	<**0.01**	0.01 (L)
23	12.07	14.93	<**0.01**	0.03 (L)	69	78.33	79.83	**0.03**	0.34 (S)
24	27.83	32.00	<**0.01**	0.19 (L)	70	16.73	26.90	<**0.01**	0.00 (L)
25	60.17	80.12	<**0.01**	0.02 (L)	71	1.00	62.23	<**0.01**	0.02 (L)
26	21.96	14.62	<**0.01**	0.96 (L)	72	101.07	108.70	<**0.01**	0.16 (L)
27	26.80	31.17	<**0.01**	0.18 (L)	73	25.59	32.47	<**0.01**	0.01 (L)
28	45.70	48.07	<**0.01**	0.20 (L)	74	108.40	179.00	<**0.01**	0.00 (L)
29	1.00	1.07	0.16	0.47 (N)	75	33.03	44.00	<**0.01**	0.00 (L)
30	9.77	10.33	<**0.01**	0.30 (M)	76	33.31	25.17	<**0.01**	0.99 (L)
31	15.30	14.27	<**0.01**	0.73 (M)	77	31.17	38.30	<**0.01**	0.02 (L)
32	3.80	4.63	<**0.01**	0.15 (L)	78	6.53	7.37	<**0.01**	0.28 (M)
33	12.53	19.30	<**0.01**	0.00 (L)	79	5.17	6.70	**0.02**	0.33 (M)
34	44.57	40.93	0.05	0.65 (S)	80	6.93	9.73	<**0.01**	0.01 (L)
35	8.59	10.03	<**0.01**	0.19 (L)	81	2.00	2.00	NaN	0.50 (N)
36	2.00	2.00	NaN	0.50 (N)	82	1.00	1.00	1	0.50 (N)
37	10.30	14.72	<**0.01**	0.08 (L)	83	11.40	14.67	<**0.01**	0.00 (L)
38	52.14	81.63	<**0.01**	0.00 (L)	84	25.50	35.03	<**0.01**	0.00 (L)
39	21.73	29.07	<**0.01**	0.07 (L)	85	150.37	145.86	0.06	0.66 (S)
40	69.03	118.33	<**0.01**	0.00 (L)	86	97.24	114.41	<**0.01**	0.03 (L)
41	14.79	19.33	<**0.01**	0.00 (L)	87	16.39	20.05	<**0.01**	0.01 (L)
42	10.73	10.17	0.14	0.60 (S)	88	175.24	168.07	<**0.01**	0.72 (M)
43	1.00	1.03	0.33	0.48 (N)	89	3.33	3.30	0.88	0.51 (N)
44	1.00	8.27	**0.02**	0.42 (S)	90	1.00	1.23	**0.01**	0.38 (S)
45	1.00	1.03	0.33	0.48 (N)	91	18.04	22.52	<**0.01**	0.00 (L)
46	1.00	1.93	<**0.01**	0.03 (L)					

6.2 **RQ$_2$** - Size

We addressed **RQ$_2$** by first computing the overall size of the test classes generated by the experimented approaches. Tables 4 and 5 report the average values and the comparison between the two approaches in terms of lines of code (LOCs) and number of methods respectively. As expected, G-Mosa produced test classes having a statistically higher size than Mosa when considering lines

Table 6 Mean methods length achieved by Mosa and G-Mosa, with p-values resulting from the Wilcoxon test and Vargha-Delaney $A_1 2$ effect size. We use N, S, M, and L to indicate negligible, small, medium and large effect size respectively. Significant p-values are reported in bold-face.

Mean Test Cases Length									
	Average		Mosa vs. G-Mosa			Average		Mosa vs. G-Mosa	
ID	**Mosa**	**G-Mosa**	**p-value**	A_{12}	**ID**	**Mosa**	**G-Mosa**	**p-value**	A_{12}
1	1.00	1.20	0.49	0.47 (N)	47	3.68	3.38	<**0.01**	0.93 (L)
2	4.86	3.59	<**0.01**	0.80 (L)	48	4.68	3.48	**0.03**	0.68 (M)
3	3.29	3.31	0.16	0.39 (S)	49	2.24	2.16	<**0.01**	0.88 (L)
4	4.53	3.41	<**0.01**	0.97 (L)	50	2.44	2.45	0.97	0.51 (N)
5	2.51	2.21	<**0.01**	1.00 (L)	51	2.53	2.46	<**0.01**	0.72 (M)
6	1.00	6.94	<**0.01**	0.03 (L)	52	2.38	2.41	**0.02**	0.69 (M)
7	3.28	3.02	<**0.01**	0.92 (L)	53	2.63	2.53	<**0.01**	0.78 (L)
8	1.23	1.30	<**0.01**	0.08 (L)	54	3.24	3.16	<**0.01**	0.78 (L)
9	4.43	4.27	<**0.01**	0.87 (L)	55	3.25	2.96	<**0.01**	0.72 (M)
10	4.15	4.00	<**0.01**	0.79 (L)	56	2.71	2.46	<**0.01**	0.86 (L)
11	3.84	3.31	<**0.01**	1.00 (L)	57	3.77	3.41	<**0.01**	0.99 (L)
12	3.98	4.14	**0.01**	0.31 (M)	58	2.83	2.75	<**0.01**	0.83 (L)
13	4.75	4.70	0.34	0.59 (S)	59	2.81	2.69	0.97	0.52 (N)
14	2.61	1.59	<**0.01**	0.97 (L)	60	3.20	2.65	<**0.01**	0.74 (L)
15	8.10	5.61	<**0.01**	0.81 (L)	61	5.15	5.03	<**0.01**	0.74 (M)
16	3.41	2.36	0.05	0.66 (S)	62	3.22	2.97	<**0.01**	0.84 (L)
17	3.22	2.91	<**0.01**	0.80 (L)	63	3.04	2.81	<**0.01**	0.99 (L)
18	3.48	3.25	**0.04**	0.65 (S)	64	5.54	4.83	<**0.01**	0.76 (L)
19	2.05	2.07	**0.02**	0.35 (S)	65	1.00	1.00	1.00	0.50 (N)
20	2.00	2.00	1.00	0.50 (N)	66	3.00	2.68	<**0.01**	0.98 (L)
21	1.00	1.00	1.00	0.48 (N)	67	4.16	4.36	<**0.01**	0.26 (L)
22	3.01	2.90	<**0.01**	0.87 (L)	68	4.07	3.75	<**0.01**	0.93 (L)
23	4.98	5.26	**0.01**	0.33 (M)	69	2.54	2.68	<**0.01**	0.25 (L)
24	2.67	2.63	<**0.01**	0.78 (L)	70	3.19	2.88	<**0.01**	0.97 (L)
25	4.77	4.55	**0.01**	0.68 (M)	71	2.00	2.00	1.00	0.52 (N)
26	2.88	2.80	<**0.01**	0.85 (L)	72	2.64	2.49	<**0.01**	0.95 (L)
27	2.53	2.55	0.77	0.50 (N)	73	1.00	1.00	1.00	0.50 (N)
28	2.87	2.83	0.14	0.39 (S)	74	3.74	3.42	<**0.01**	0.78 (L)
29	3.67	3.53	**0.01**	0.69 (M)	75	3.45	3.21	<**0.01**	0.83 (L)
30	3.13	3.02	0.28	0.61 (S)	76	3.28	3.10	<**0.01**	0.97 (L)
31	3.14	2.89	<**0.01**	0.86 (L)	77	4.02	3.65	<**0.01**	1.00 (L)
32	2.27	2.24	0.27	0.58 (S)	78	4.55	3.94	**0.01**	0.74 (L)
33	3.86	3.24	<**0.01**	0.99 (L)	79	5.47	4.73	<**0.01**	0.90 (L)
34	4.00	4.83	0.96	0.37 (S)	80	5.11	4.90	<**0.01**	0.77 (L)
35	3.31	3.22	0.07	0.65 (S)	81	6.17	6.28	0.74	0.51 (N)
36	2.78	2.78	0.84	0.50 (N)	82	1.00	2.25	0.05	0.42 (S)
37	2.99	2.68	<**0.01**	1.00 (L)	83	1.75	1.89	<**0.01**	0.00 (L)
38	2.00	2.00	1.00	0.50 (N)	84	6.75	9.00	<**0.01**	0.10 (L)
39	4.16	3.85	<**0.01**	0.69 (M)	85	3.65	3.09	<**0.01**	0.89 (L)
40	3.10	2.89	<**0.01**	0.78 (L)	86	1.40	2.10	0.81	0.30 (M)
41	2.47	7.61	<**0.01**	0.00 (L)	87	11.43	6.41	<**0.01**	0.80 (L)
42	1.84	1.75	**0.03**	0.66 (S)	88	3.36	2.72	**0.01**	0.64 (S)
43	4.22	4.30	<**0.01**	0.82 (L)	89	5.98	5.72	<**0.01**	0.84 (L)
44	2.51	2.37	<**0.01**	0.96 (L)	90	2.51	2.48	**0.02**	0.66 (S)
45	7.54	7.09	**0.02**	0.68 (M)	91	3.02	2.83	<**0.01**	0.91 (L)
46	3.16	2.72	<**0.01**	0.90 (L)					

of code and number of test methods. Specifically, for ≈73% of the classes under test, G-Mosa generates significantly larger test classes than Mosa. A similar results is achieved when considering the average number of test methods generated by the two approaches. The results highlight a statistical significant difference in 82% of cases, 77% in favor of Mosa and the remaining 5% in favor of G-Mosa. This is a clear evidence that test classes generated by Mosa are significantly smaller. Looking deeper at this result, G-Mosa tends to generate larger test suites having both a higher count of methods and increased total lines of code. This is due to the intrinsic design of the approach. The larger method count can be readily understood by recognizing that G-MOSA places emphasis on producing a set of tests that covers individual branches of the production methods. This step influences the quantity of intra-method test cases produced, since the approach does not allow tests to tangentially cover multiple branches but require single tests to cover single branches. As such, more tests are required to cover branches individually. This design choice has an immediate impact on the higher volume of total lines of code: more test cases naturally lead to have additional lines of code in the form of method signatures, variable definitions/initializations, and single assertion statements.

However, to make a more fair comparison, we also computed the *size per test case*, namely the mean lines of code of each test method generated by the experimented approaches. Table 6 reports the results achieved for this analysis. $\hat{A}_{12} > 0.5$ indicates that test cases generated by G-Mosa are, on average, smaller than those of Mosa. The results highlighted a clear difference in the size of tests generated by the two approaches. In particular, for 68 out of 91 classes ($\approx 75\%$) there is a statistically significant difference in the mean length of generated test cases. Of these 68 classes, G-Mosa produced smaller tests than Mosa in 58 cases ($\approx 85\%$), 51 with *large* or *medium* effect size with an average size reduction raging between $\approx 1\%$ and $\approx 44\%$. Such results led us to reject the null hypothesis **Hn 3**, thus accepting the alternative hypothesis **An 3** in favor of G-Mosa: it generates test methods having a *size* significantly lower than Mosa. In spite of the statistical results, the average size per test case of G-Mosa and Mosa looks similar if we consider the absolute number of lines of code of the tests generated. This may potentially limit the relevance of our findings in practice, as both the approaches tend to generate small test cases, with G-MOSA able to further minimize the size. In this respect, it is worth remarking that the generation of statistically smaller test cases may have implications on their overall maintainability and understandability. This is what we aim at assessing in the context of $\mathbf{RQ}_3$ and $\mathbf{RQ}_4$, where our goal will be to evaluate whether the difference in terms of size per test case, which may seem marginal at a first glance, has serious implications in practice.

While G-Mosa produces test methods of smaller size than Mosa, it is worth remarking that, in rare cases, the baseline outperformed our technique. This is the case of the class `azureus2.core3.disk.impl.resume.RDResumeHandler` (id. 41), which is characterized by 300 branches. When generating tests for such a class, Mosa was able to *significantly reduce* the mean methods size of the generated test class (*i.e.*, $\approx 67\%$ of average size reduction over the 30 runs). By manually investigating the class, we observed that the high cyclomatic complexity influenced G-Mosa- the McCabe cyclomatic complexity measured 97 in this case, hence confirming the complexity of testing the class. By construction, our technique equally splits the search budget between the two steps: this may clearly impact the intra-class testing process, namely the one responsible to exercise the target class by employing multiple calls of the production code.

In cases like the one of the example, the excessive cyclomatic complexity did not allow G-Mosa to generate effective intra-class tests, while Mosa could spend the entire search budget for the generation of those tests. This example highlights a possible limitation of our approach: the configuration of the search budget may have an influence on the results. While we plan to investigate how to best tune the approach in our followup research on the matter, we could still conclude that this is not something arising frequently, hence making G-Mosa a valid alternative for automatic test case generation.

Summing Up: Mosa generates test classes that are significantly smaller than G-Mosa. However, G-Mosa is able to generate tests that are significantly better in terms of *size per method* with respect to Mosa. Moreover, we observed that the configuration of the search budget might influence the resulting performance in some cases.

6.3 $\mathbf{RQ}_3$ - Maintainability

In the context of the $\mathbf{RQ}_3$ we compared the maintainability of test classes generated by Mosa and G-Mosa. To have a comprehensive view of test classes' maintainability we relied on three different metrics capturing different aspects of software maintainability, namely (i) the Weighted Methods Count (WMC) to measure class complexity, (ii) the Fan-out to measure class coupling, and (iii) the number of test smells contained in the generated test suites.

Table 7 reports the average WMC and the pairwise statistical analysis for the test classes generated by the two approaches. The results clearly highlight that test classes generated by Mosa have significantly lower complexity. Indeed, for 65 out of the 91 classes in our dataset (i.e., ≈71%) Mosa achieves significantly better results than G-Mosa with *large* or *medium* effect size. Therefore, we can reject the null hypothesis **Hn 4**, thus accepting the alternative hypothesis **An 4** in favor of Mosa. This result could immediately suggest that Mosa generates more maintainable test suites. Therefore, a deeper discussion is deserved. The metric we used for measuring code complexity is the Weighted Methods Count (WMC). This metric sums the complexities of all test methods in a test class, therefore, the higher the number of methods, the higher the overall complexity. In $\mathbf{RQ}_2$, we demonstrated that the test classes generated by G-Mosa are larger in terms of total size and number of methods: because of that, it is not really surprising to see that the statistical tests for this metric are in favor of Mosa. Nonetheless, it is also worth remarking that $\mathbf{RQ}_2$ showed that our approach tends to preserve the conciseness of individual test methods. As such, it might still be possible that the individual tests generated by G-Mosa are more understandable and maintainable. While the quantitative analysis made to answer $\mathbf{RQ}_3$ aims at addressing this question in a systematic manner, a manual investigation of the test cases generated by the two approaches already revealed some insights. More specifically, we can consider the case of class `com.lts.util.scheduler.NewScheduler` (id. 87) as an example. As reported in Table 7, G-Mosa generates more complex tests classes in this case. Manually inspecting the source code we were able

Table 7 Weighted Methods Count (WMC) of test classes generated by Mosa and G-Mosa, with p-values resulting from the Wilcoxon test and Vargha-Delaney A_12 effect size. We use N, S, M, and L to indicate negligible, small, medium and large effect size respectively. Significant p-values are reported in bold-face.

Weighted Methods Count (WMC)									
	Average		Mosa vs. G-Mosa			Average		Mosa vs. G-Mosa	
ID	Mosa	G-Mosa	p-value	$\hat{A}_{12}$	ID	Mosa	G-Mosa	p-value	$\hat{A}_{12}$
1	2.90	4.00	<**0.01**	0.32 (M)	47	4.00	3.00	<**0.01**	1.00 (L)
2	110.03	204.68	<**0.01**	0.00 (L)	48	47.97	59.20	<**0.01**	0.01 (L)
3	25.30	24.72	0.34	0.57 (N)	49	40.10	57.72	<**0.01**	0.00 (L)
4	86.57	121.13	<**0.01**	0.00 (L)	50	4.00	4.40	0.08	0.45 (N)
5	38.77	44.56	<**0.01**	0.13 (L)	51	50.69	70.90	<**0.01**	0.00 (L)
6	54.37	61.10	<**0.01**	0.19 (L)	52	169.83	268.93	<**0.01**	0.00 (L)
7	21.24	24.22	<**0.01**	0.19 (L)	53	60.07	64.63	0.05	0.35 (S)
8	130.86	193.93	<**0.01**	0.00 (L)	54	44.90	56.03	<**0.01**	0.00 (L)
9	19.00	23.38	0.18	0.37 (S)	55	253.77	325.13	<**0.01**	0.00 (L)
10	187.60	210.79	<**0.01**	0.08 (L)	56	13.60	11.97	<**0.01**	0.97 (L)
11	76.67	78.07	0.71	0.47 (N)	57	64.73	76.23	<**0.01**	0.03 (L)
12	81.93	105.10	<**0.01**	0.03 (L)	58	283.27	258.52	<**0.01**	0.98 (L)
13	16.00	19.87	<**0.01**	0.02 (L)	59	154.30	233.40	<**0.01**	0.00 (L)
14	52.21	55.93	<**0.01**	0.29 (M)	60	14.62	17.67	<**0.01**	0.04 (L)
15	20.27	34.27	<**0.01**	0.01 (L)	61	20.73	33.93	<**0.01**	0.00 (L)
16	71.90	100.93	<**0.01**	0.00 (L)	62	68.40	84.00	<**0.01**	0.03 (L)
17	4.00	6.93	<**0.01**	0.13 (L)	63	197.40	220.45	<**0.01**	0.12 (L)
18	18.71	18.87	0.83	0.48 (N)	64	77.78	111.96	<**0.01**	0.00 (L)
19	11.60	11.60	0.67	0.47 (N)	65	20.63	34.14	<**0.01**	0.00 (L)
20	1.00	8.00	<**0.01**	0.00 (L)	66	57.33	65.77	<**0.01**	0.01 (L)
21	21.59	30.30	<**0.01**	0.10 (L)	67	418.62	616.37	<**0.01**	0.00 (L)
22	161.66	197.07	<**0.01**	0.11 (L)	68	44.90	48.83	<**0.01**	0.12 (L)
23	42.14	43.50	0.78	0.48 (N)	69	165.13	173.03	<**0.01**	0.17 (L)
24	94.33	117.12	<**0.01**	0.09 (L)	70	51.40	101.50	<**0.01**	0.00 (L)
25	152.17	209.62	<**0.01**	0.00 (L)	71	2.00	190.07	<**0.01**	0.00 (L)
26	65.07	51.72	<**0.01**	0.82 (L)	72	222.37	248.96	<**0.01**	0.01 (L)
27	88.03	105.97	<**0.01**	0.15 (L)	73	78.79	111.33	<**0.01**	0.00 (L)
28	98.67	104.57	<**0.01**	0.14 (L)	74	399.93	655.50	<**0.01**	0.00 (L)
29	3.90	4.28	0.09	0.45 (N)	75	106.48	144.10	<**0.01**	0.00 (L)
30	23.37	26.83	<**0.01**	0.12 (L)	76	95.52	72.60	<**0.01**	0.93 (L)
31	36.87	34.57	<**0.01**	0.73 (M)	77	73.93	96.13	<**0.01**	0.00 (L)
32	11.53	15.47	<**0.01**	0.00 (L)	78	16.57	21.70	<**0.01**	0.04 (L)
33	28.63	44.03	<**0.01**	0.00 (L)	79	17.14	24.03	<**0.01**	0.24 (L)
34	113.89	117.37	0.63	0.46 (N)	80	19.53	31.30	<**0.01**	0.00 (L)
35	23.79	28.70	<**0.01**	0.09 (L)	81	6.00	5.00	<**0.01**	1.00 (L)
36	8.00	7.00	<**0.01**	1.00 (L)	82	5.00	5.00	1	0.50 (N)
37	28.13	38.72	<**0.01**	0.12 (L)	83	32.50	41.37	<**0.01**	0.00 (L)
38	154.48	260.97	<**0.01**	0.00 (L)	84	63.20	88.57	<**0.01**	0.00 (L)
39	64.37	85.20	<**0.01**	0.13 (L)	85	334.83	364.36	<**0.01**	0.26 (M)
40	225.59	398.17	<**0.01**	0.00 (L)	86	264.34	338.00	<**0.01**	0.00 (L)
41	36.55	49.00	<**0.01**	0.00 (L)	87	46.07	51.92	<**0.01**	0.12 (L)
42	23.47	21.86	0.14	0.60 (S)	88	390.48	404.72	0.05	0.35 (S)
43	1.30	2.30	**0.01**	0.35 (S)	89	9.23	9.07	0.64	0.53 (N)
44	1.00	18.67	**0.02**	0.42 (S)	90	4.00	4.93	**0.01**	0.38 (S)
45	1.50	2.13	0.13	0.41 (S)	91	60.54	78.45	<**0.01**	0.00 (L)
46	3.90	7.72	<**0.01**	0.03 (L)					

to confirm our above consideration. In particular, we noticed that test classes generated by G-Mosa have a mean number of ≈15 test methods per class while the average number of test methods per class is ≈7 for Mosa. As such, analyzing this case allows us to confirm that the higher complexity could be associated to the higher number of methods generated. Indeed, by checking the mean methods length for this example class in Table 6, we observe a value of 6.41 for G-Mosa compared to 11.43 of Mosa. In this specific situation,

Table 8 Fan-out of test classes generated by Mosa and G-Mosa, with p-values resulting from the Wilcoxon test and Vargha-Delaney $A_{1}2$ effect size. We use N, S, M, and L to indicate negligible, small, medium and large effect size respectively. Significant p-values are reported in bold-face.

	Fan-out								
	Average		**Mosa vs. G-Mosa**			**Average**		**Mosa vs. G-Mosa**	
ID	**Mosa**	**G-Mosa**	**p-value**	$\hat{\mathbf{A}}_{12}$	**ID**	**Mosa**	**G-Mosa**	**p-value**	$\hat{\mathbf{A}}_{12}$
1	1.27	2.00	**<0.01**	0.32 (M)	47	3.00	2.00	**<0.01**	1.00 (L)
2	122.07	184.89	**<0.01**	0.00 (L)	48	59.14	52.40	**<0.01**	0.87 (L)
3	18.43	15.86	**<0.01**	0.86 (L)	49	45.93	51.83	0.10	0.37 (S)
4	93.36	107.97	**<0.01**	0.07 (L)	50	2.00	2.20	0.08	0.45 (N)
5	117.70	103.25	**<0.01**	0.92 (L)	51	41.55	39.93	**0.02**	0.67 (M)
6	105.07	72.37	**<0.01**	0.98 (L)	52	266.33	227.40	**<0.01**	0.97 (L)
7	34.10	35.78	0.54	0.43 (N)	53	64.29	50.33	**<0.01**	0.95 (L)
8	99.48	129.33	**<0.01**	0.00 (L)	54	43.83	30.33	**<0.01**	0.97 (L)
9	34.07	47.46	**<0.01**	0.14 (L)	55	460.77	324.33	**<0.01**	0.95 (L)
10	233.27	249.86	**<0.01**	0.28 (M)	56	8.70	7.93	**<0.01**	0.97 (L)
11	109.73	91.20	**<0.01**	0.91 (L)	57	96.13	49.50	**<0.01**	1.00 (L)
12	126.23	109.27	**<0.01**	0.80 (L)	58	387.03	370.59	**<0.01**	0.78 (L)
13	9.00	11.10	**<0.01**	0.02 (L)	59	161.78	267.47	**<0.01**	0.00 (L)
14	52.00	48.63	**<0.01**	0.78 (L)	60	8.34	9.83	**<0.01**	0.12 (L)
15	14.67	36.07	**<0.01**	0.00 (L)	61	26.40	20.87	**<0.01**	0.89 (L)
16	50.83	53.70	**0.02**	0.33 (M)	62	79.50	46.72	**<0.01**	0.96 (L)
17	2.00	3.47	**<0.01**	0.13 (L)	63	219.30	248.79	**<0.01**	0.17 (L)
18	18.79	13.83	**<0.01**	0.91 (L)	64	114.00	92.61	**<0.01**	0.98 (L)
19	13.17	8.77	**<0.01**	0.91 (L)	65	15.53	18.79	**<0.01**	0.17 (L)
20	0.00	4.00	**<0.01**	0.00 (L)	66	72.63	40.80	**<0.01**	1.00 (L)
21	17.59	24.30	**<0.01**	0.16 (L)	67	403.10	526.37	**<0.01**	0.03 (L)
22	171.86	162.90	0.99	0.50 (N)	68	49.14	33.40	**<0.01**	1.00 (L)
23	41.59	35.90	**<0.01**	0.84 (L)	69	354.10	221.83	**<0.01**	1.00 (L)
24	81.97	129.73	**<0.01**	0.06 (L)	70	64.73	99.67	**<0.01**	0.01 (L)
25	371.55	449.38	**<0.01**	0.17 (L)	71	1.00	80.23	**<0.01**	0.00 (L)
26	112.54	52.31	**<0.01**	0.99 (L)	72	405.20	375.19	**<0.01**	0.78 (L)
27	140.67	96.34	**<0.01**	0.98 (L)	73	84.48	143.03	**<0.01**	0.00 (L)
28	277.67	119.00	**<0.01**	1.00 (L)	74	405.40	716.25	**<0.01**	0.00 (L)
29	1.93	2.14	0.09	0.45 (N)	75	99.90	127.45	**<0.01**	0.02 (L)
30	36.17	33.33	**<0.01**	0.77 (L)	76	92.00	56.03	**<0.01**	0.97 (L)
31	48.00	22.23	**<0.01**	1.00 (L)	77	105.66	77.77	**<0.01**	0.97 (L)
32	12.47	14.37	**<0.01**	0.26 (L)	78	16.53	19.07	**0.01**	0.29 (M)
33	33.53	36.53	0.07	0.36 (S)	79	15.69	20.43	0.55	0.46 (N)
34	578.71	513.43	0.05	0.65 (S)	80	17.53	16.03	**0.02**	0.67 (M)
35	18.90	11.77	**<0.01**	0.97 (L)	81	6.00	4.00	**<0.01**	1.00 (L)
36	5.00	4.00	**<0.01**	1.00 (L)	82	4.00	4.00	1	0.50 (N)
37	77.07	36.89	**<0.01**	0.86 (L)	83	35.53	31.80	**<0.01**	0.84 (L)
38	148.76	225.00	**<0.01**	0.00 (L)	84	106.80	63.17	**<0.01**	0.92 (L)
39	79.67	75.63	0.20	0.60 (S)	85	778.63	813.32	0.80	0.52 (N)
40	194.69	295.50	**<0.01**	0.00 (L)	86	344.28	356.86	0.31	0.42 (S)
41	54.62	55.37	0.30	0.58 (S)	87	69.50	61.58	**<0.01**	0.79 (L)
42	31.67	20.55	**<0.01**	0.97 (L)	88	1222.69	1022.03	**0.02**	0.68 (M)
43	0.20	0.93	**0.01**	0.35 (S)	89	7.90	6.97	0.15	0.61 (S)
44	0.00	20.57	**0.02**	0.42 (S)	90	3.00	3.70	**0.01**	0.38 (S)
45	0.40	0.90	0.13	0.41 (S)	91	46.32	56.14	**<0.01**	0.01 (L)
46	1.93	3.86	**<0.01**	0.03 (L)					

it seems that the two approaches generate classes having approximately the same overall size (class lines of code), however, the higher number of methods negatively influences the overall class complexity.

When it turns to assess classes' coupling, it seems that there is no a clear winner between the two approaches. According to the results reported in Table 8, conflicting considerations could be drawn based on each specific class under consideration. Indeed, by simply looking at the average values for the two

Table 9 Number of Test Smells in test classes generated by Mosa and G-Mosa, with p-values resulting from the Wilcoxon test and Vargha-Delaney $A_1 2$ effect size. We use N, S, M, and L to indicate negligible, small, medium and large effect size respectively. Significant p-values are reported in bold-face.

Number of Test Smells									
	Average		Mosa vs. G-Mosa			Average		Mosa vs. G-Mosa	
ID	**Mosa**	**G-Mosa**	**p-value**	$\hat{\mathbf{A}}_{12}$	**ID**	**Mosa**	**G-Mosa**	**p-value**	$\hat{\mathbf{A}}_{12}$
1	0.00	0.00	NaN	0.50 (N)	47	0.00	0.00	NaN	0.50 (N)
2	3.00	1.00	<**0.01**	1.00 (L)	48	2.00	1.00	<**0.01**	1.00 (L)
3	1.00	0.00	<**0.01**	1.00 (L)	49	1.33	0.21	<**0.01**	0.93 (L)
4	1.11	0.00	<**0.01**	1.00 (L)	50	0.00	0.00	NaN	0.50 (N)
5	1.93	1.00	<**0.01**	0.97 (L)	51	1.00	0.03	<**0.01**	0.98 (L)
6	1.50	0.07	<**0.01**	0.98 (L)	52	1.00	0.00	<**0.01**	1.00 (L)
7	0.79	0.44	0.05	0.67 (M)	53	0.96	0.00	<**0.01**	0.98 (L)
8	1.52	1.13	0.27	0.58 (S)	54	0.97	0.00	<**0.01**	0.98 (L)
9	0.03	0.00	0.54	0.52 (N)	55	1.00	0.13	<**0.01**	0.93 (L)
10	1.67	0.34	<**0.01**	0.88 (L)	56	0.00	0.00	NaN	0.50 (N)
11	1.00	0.00	<**0.01**	1.00 (L)	57	1.00	0.00	<**0.01**	1.00 (L)
12	0.93	0.03	<**0.01**	0.95 (L)	58	1.00	0.00	<**0.01**	1.00 (L)
13	0.00	0.00	NaN	0.50 (N)	59	0.00	0.00	NaN	0.50 (N)
14	0.17	0.00	**0.04**	0.57 (N)	60	1.00	0.00	<**0.01**	1.00 (L)
15	0.77	0.00	<**0.01**	0.88 (L)	61	1.00	0.00	<**0.01**	1.00 (L)
16	1.17	0.00	<**0.01**	1.00 (L)	62	0.97	0.17	<**0.01**	0.90 (L)
17	0.00	0.00	NaN	0.50 (N)	63	1.00	0.00	<**0.01**	1.00 (L)
18	1.00	0.00	<**0.01**	1.00 (L)	64	1.00	0.00	<**0.01**	1.00 (L)
19	1.00	0.00	<**0.01**	1.00 (L)	65	1.00	0.07	<**0.01**	0.97 (L)
20	0.00	0.00	NaN	0.50 (N)	66	2.00	1.00	<**0.01**	1.00 (L)
21	1.00	0.07	<**0.01**	0.91 (L)	67	3.31	1.60	<**0.01**	0.96 (L)
22	1.17	0.37	<**0.01**	0.74 (L)	68	1.00	0.00	<**0.01**	1.00 (L)
23	1.00	0.00	<**0.01**	1.00 (L)	69	2.00	1.00	<**0.01**	1.00 (L)
24	0.97	1.08	0.33	0.47 (N)	70	0.67	0.03	<**0.01**	0.78 (L)
25	0.97	0.50	<**0.01**	0.73 (M)	71	1.00	0.03	<**0.01**	0.98 (L)
26	1.04	0.00	<**0.01**	1.00 (L)	72	1.07	0.04	<**0.01**	0.97 (L)
27	1.17	0.00	<**0.01**	1.00 (L)	73	1.00	1.00	NaN	0.50 (N)
28	1.00	0.07	<**0.01**	0.97 (L)	74	2.77	2.00	<**0.01**	0.90 (L)
29	0.00	0.00	NaN	0.50 (N)	75	1.03	0.00	<**0.01**	1.00 (L)
30	1.00	0.00	<**0.01**	1.00 (L)	76	1.00	0.00	<**0.01**	1.00 (L)
31	1.00	0.00	<**0.01**	1.00 (L)	77	1.00	0.00	<**0.01**	1.00 (L)
32	1.00	0.07	<**0.01**	0.97 (L)	78	1.03	0.03	<**0.01**	0.98 (L)
33	1.00	0.00	<**0.01**	1.00 (L)	79	1.03	0.00	<**0.01**	1.00 (L)
34	0.71	0.07	<**0.01**	0.81 (L)	80	1.00	0.00	<**0.01**	1.00 (L)
35	0.97	0.00	<**0.01**	0.98 (L)	81	1.00	0.00	<**0.01**	1.00 (L)
36	0.00	0.00	NaN	0.50 (N)	82	0.00	0.00	NaN	0.50 (N)
37	1.60	0.67	<**0.01**	0.86 (L)	83	1.00	0.00	<**0.01**	1.00 (L)
38	1.00	0.10	<**0.01**	0.95 (L)	84	1.03	0.03	<**0.01**	0.98 (L)
39	1.00	0.00	<**0.01**	1.00 (L)	85	1.40	0.23	<**0.01**	0.92 (L)
40	1.17	0.00	<**0.01**	0.98 (L)	86	1.48	0.36	<**0.01**	0.91 (L)
41	1.00	0.00	<**0.01**	1.00 (L)	87	1.71	0.00	<**0.01**	1.00 (L)
42	1.37	0.00	<**0.01**	1.00 (L)	88	1.41	0.21	<**0.01**	0.93 (L)
43	0.00	0.00	NaN	0.50 (N)	89	0.03	0.00	0.33	0.52 (N)
44	0.00	0.00	NaN	0.50 (N)	90	0.00	0.00	NaN	0.50 (N)
45	0.07	0.17	0.24	0.45 (N)	91	0.29	0.24	0.71	0.52 (N)
46	0.00	0.00	NaN	0.50 (N)					

approaches we could see that in ≈48% of the cases Mosa achieves a better (i.e., lower) coupling while G-Mosa performs better in the remaining ≈52%. From a statistical point of view, we can observe that for 80% of the classes in our dataset (i.e., 73 cases out of 91) there is a statistically significant difference between the coupling achieved by the two approaches. As such, the results leads to reject the null hypothesis **Hn 5** in favor of the alternative hypotesis **An 5**. By considering only these 73 classes in which there is a statistically significant

Table 10 Understandability scores of test classes generated by Mosa and G-Mosa, with p-values resulting from the Wilcoxon test and Vargha-Delaney $A_1 2$ effect size. We use N, S, M, and L to indicate negligible, small, medium and large effect size respectively. Significant p-values are reported in bold-face.

Understandability scores			
Average		**Mosa vs. G-Mosa**	
Mosa	**G-Mosa**	**p-value**	$\hat{\mathbf{A}}_{12}$
2.50	2.90	**0.01**	0.41 (S)

difference, we observe that for 30 of them Mosa performs better (i.e., ≈41%), while G-Mosa achieves a lower coupling for the remaining 43 (i.e., ≈59%). While these results could suggest that G-Mosa outperforms Mosa in terms of coupling, we cannot speculate on the results achieved, as they do not allow to make a definitive conclusion. In this sense, more investigations would be desirable.

As a last dimension to measure classes' maintainability, we considered the total number of test smells in classes generated by the two approaches. Table 9 reports the results achieved for this analysis. First and foremost, both approaches allow the generation of test classes having a limited number of test smells, with average values ranging between 0 and 3.31. Additionally, there are several cases in which both approaches generate classes with no test smells. These cases are easily recognizable by the **NaN** values in the p-value column (this is due to the Wilcoxon test failing in presence of ties).

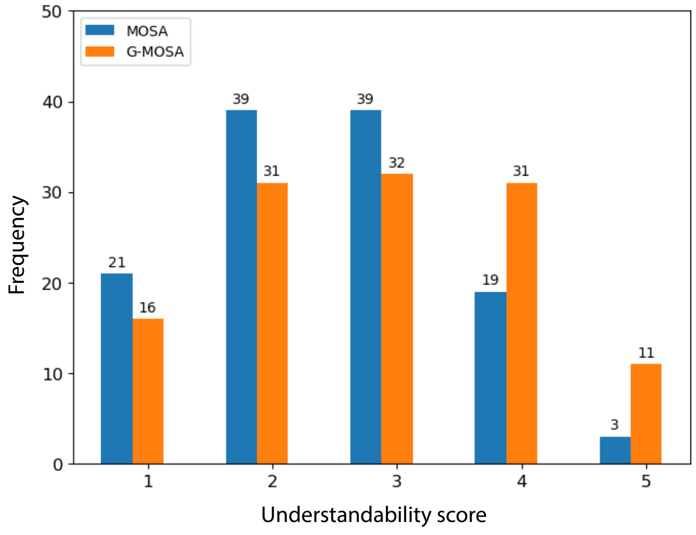
However, when it turns to the statistical comparison the results clearly highlight that G-Mosa outperforms Mosa. For 68 out of the 91 analyzed classes (≈75%) we have a p-value lower than 0.5 indicating a statistical significant difference. In all these 68 cases G-Mosa outperforms Mosa with a large effect size. Based on such considerations, we can reject the null hypothesis **Hn 6** and accept the alternative hypothesis **An 6** in favor of G-Mosa.

Summing Up: On the maintainability side, we could not reach a definitive conclusion. When considering WMC, our findings report that Mosa is statistically better than G-Mosa, even though we highlight that this may not necessarily indicate a lower level of understandability and maintainability by G-Mosa. In terms of coupling, there seems not be a clear winner. Finally, G-Mosa provides test cases with a significant lower amount of test smells. The follow-up analysis of the practitioners' perspective might provide further insights into the merit of the experimented techniques in terms of maintainability.

6.4 $\mathbf{RQ}_4$ - Understandability

To answer $\mathbf{RQ}_4$, we compared the understandability scores given to test cases generated by Mosa and G-Mosa. Figure 2 shows a plot reporting the understandability scores for both approaches. More particularly, the figure shows the amount of participants who scored the understandability of test cases produced by the experimented approaches from 1 (low understandability) to 5 (high understandability). As we can observe, tests generated by Mosa are

Fig. 2 Understandability scores achieved by MOSA and G-Mosa.



associated with lower understandability scores, as 99 out of the 120 (≈82%) respondents rated them with a score between 1 and 3. On the contrary, for G-Mosa the ratings are higher, with 42 participants (35%) giving ratings of 4 or 5. This result already provides an indication of the goodness of the test cases generated by a granular approach: according to our findings, G-Mosa is actually able to generate test classes which are perceived by practitioners are more understandable, overall.

Table 10 reports the results of the statistical analysis performed to compare the understandability scores of Mosa and G-Mosa. The tests confirmed the quantitative insights discussed above. On the one hand, the test classes generated by our approach have higher ratings on average (2.9 against 2.5). On the other hand, the Wilcoxon and Vargha-Delaney tests reported a p-value of 0.01, highlighting statistical significance with a *small* effect size. On the basis of these observations, we could reject the null hypothesis **Hn 7** and accept the alternative hypothesis **An 7** in favor of G-Mosa: our approach generates more understandable test cases with a statistically significant difference with respect to the baseline approach.

To further support our findings, we also looked at the assertions reported by participants for tests generated by the two approaches. As introduced in Section 5, we performed a manual analysis of all the assertion statements to check whether they were consistent with the corresponding test case. From the analysis, it turned out that for both approaches participants were able to write valid assertion statements in most of the cases. In particular, as for Mosa, at least one valid assertion was reported for 195 out of the 240 tests (≈ 81%). When considering test cases generated by G-Mosa, 220 cases with at least one valid assert statement were reported (≈92%). These results further corroborate the conclusion that the test cases generated by G-Mosa are, overall, more understandable than those generated by Mosa.

In literature, a higher number of assertions per single test case, i.e., a higher assertion density, has been often associated with an increased capability of test classes to identify faults in production code [41]. As such, the reader may possibly interpret the results so that, despite the lower understandability, the test cases generated by Mosa could still be more effective when employed to discover faults. While this perspective might be worth of assessment through a dedicated empirical investigation, we believe that our findings should be interpreted differently. By design, G-Mosa generates more test cases, but of smaller size and more cohesive when compared to the baseline. This implies that the developers involved in our survey study were called to analyze a larger amount of tests of smaller size: when analyzing the assert statements, we could realize that the developers were able to focus more the scope of the assertions, hence letting the tests focusing on more specific targets of the production code. In our view, this represents a valuable characteristic of our approach, as it allows developers to develop better test cases. In addition, it is also worth remarking that the results obtained on the number of assertions per test case have significant implications for fault localization and debugging. Indeed, test cases with less assertions but more focused on targets might allow developers to potentially diagnose the root causes of faults with a reduced effort.

To further investigate on the motivations behind the understandability ratings provided by the survey participants, we analyzed the comments left when assessing the understandability of test cases. We noticed some responses in which users assigned low ratings to both the test classes generated by the two approaches, however, these ratings were influenced by the lack of comments and assertions that are peculiarities of automatically generated test classes. More interestingly, we found that in several cases the participants appreciated the granular nature of our approach. Here we report two of these cases. The entire list of responses can be found on our online appendix [5].

This is the case of participant #21 who reported *"**Very difficult to understand the purpose of each unit test**. This can be inferred, but without assertions, new developers will have to assume the purpose and fix the code."* for Mosa (with a rating of 2), while they rated with a score of 4 the understandability of G-Mosa with the following comment: *"**Easy to understand the purpose of each unit test**, even with modules I do not have experience with. With more comments in the code itself, the unit tests would be fully understandable."*. Similarly, participant #42 reported the following comment for G-Mosa *"The unit tests were clear and written well **since they tested only one thing at a time**. I feel like more documentation, organization, or labeling would be better"*. Also in this case, the ratings reported were 4 for G-Mosa and 2 for Mosa with the following justification: *"This class was harder to understand because there were few assertions and **the code was more verbose**"*

🔍 **Summing Up:** Test cases generated by G-Mosa are significantly more understandable than those generated by Mosa. Participants of the survey were able to generate at least one valid assertion statement in a higher number of cases for G-Mosa. Moreover, test cases generated Mosa received a higher average number of asserts per single test case, indicating that a major effort is required to write assertions for this approach.

7 Threats to Validity

In this section, we discuss the main threats that might have affected the validity of our study and how we mitigated them.

7.1 Threats to construct validity

Threats in this category refer to the relation between theory and observations. Our context was originally composed of 100 classes but we only reported results for 91 of them since the remaining 9 in our sample led EvoSuite to fail due to internal errors. Nevertheless, the size of our experiment is inline with respect to previous work [2]. Another possible threat could be connected to the selection of the baseline technique on which we built G-Mosa. The selection of Mosa was driven by the fact that this was the technique we knew best and felt most confident with to modify. Yet, we believe that the selection of another baseline would have not had an important impact on the results obtained in the context of our study. In particular, our aim was to define a systematic approach and to improve the resulting structure of the generated test cases, independently from the baseline approach, *i.e.*, the methodology implemented in G-Mosa can be applied on any automatic test case generation technique. As such, the results achieved would not be influenced by the technique chosen as baseline. In any case, we already plan to replicate our study with different core techniques in order to verify this consideration.

7.2 Threats to internal validity

As for the intrinsic factors that could have influenced our findings, our approach and the baseline used for comparison were implemented within the same tool, *i.e.*, Evosuite [20]. As such, they relied on exactly the same underlying implementation of the genetic operators, avoiding possible confounding effects due to the use of different algorithms. The parameter configuration represents a second aspect possibly affecting our results. We used the default settings available in Evosuite on the basis of previous research in the field [7] which showed that the configuration of parameters is not only expensive but also possibly ineffective in improving the performance of search-based algorithms. To deal with the inherent randomness of genetic algorithms, we re-executed the experimented approaches 30 times—as recommended by previous research [11]—and reported their average performance when discussing the results. Finally, we equally split the search budget of our technique in two: this might have led G-Mosa to underperform with respect to the optimal case, *i.e.*, as noticed in our qualitative analysis, the effectiveness of the intra-class step could be negatively influenced in some cases. Nonetheless, our goal was that of investigating the feasibility of using a two-step approach for automatic test case generation; we plan to perform an extensive analysis aimed at identifying the optimal configuration for our technique in our followup research.

In the context of the user study conducted to assess the understandability of the generated test cases, we did not limit our recruitment to original developers, but we also employed a research-oriented platform like Prolific. On the one hand, we could not finally recruit any original developers: this implies that we could not assess the understandability of the test classes generated by the compared approaches from the perspective of the actual designers of the source code under test. While the opinions of the original developers might have revealed additional insights, the expertise and background of the participants who took part to the survey make us confident of the results reported. On the other hand, the choice of selecting Prolific might have potentially introduced some sort of selection bias [61]. To mitigate this risk, we have taken two main actions. First, we introduced an incentive of 2 pounds per valid respondent, which means that the participation was stimulated through the recognition rather than left to the willingness of developers. Second, we manually verified the actual validity of the answers received, in an effort of discarding the responses from participants who did not take the task seriously. In addition, it is also worth mentioning that, other than collecting background information by directly inquiring participants, the online platform used by participants to execute the study is able to keep track of the time spent by each participant on each answer: this enabled an improved analysis of the performance of the participants and supported us when spotting cases to discard. Nonetheless, we are aware of the limitations of an online experiment - yet, with the current pandemic situation, this was the only viable solution.

Another aspect that might have affected the internal validity of the user study concerns with the selection of the test classes shown to participants. To avoid any biased selection, we proceeded with a random selection from the entire set of classes considered in our study.

7.3 Threats to conclusion validity

Threats in this category concern with the relationship between treatment and outcome. In the comparison of G-Mosa and Mosa, we adopted well-known state-of-the-art metrics to assess their structure and performance. For example, we computed branch coverage when understanding the effectiveness of the tests generated by the two approaches. In addition, we employed appropriate statistical tests to verify the significance of the differences achieved by our approach and the baseline. Specifically, we first used the Wilcoxon Rank Sum

Test [13] for statistical significance and then the Vargha-Delaney effect size statistic [68] to estimate the magnitude of the observed difference.

7.4 Threats to external validity

Threats to the external validity regard the generalization of our findings. We conducted our study considering the SF110 benchmark dataset [22], which has been widely employed by previous researchers to conduct experimentations in the context of automatic test case generation [22, 51, 31, 21]. To increase the reliability of the reported results, we also filtered out trivial classes from the initial dataset, ending up with a sample of 100 classes that allowed us to analyze the results from a statistical point of view. Nevertheless, the re-execution of the study in other contexts, *e.g.*, the *XCorpus* dataset [15], might lead to different results. We plan to tackle this potential issue in our future work. Finally, we limited the study to classes written in Java because our tooling can only deal with them: as such, replications of our work on systems written in other languages would therefore be desirable.

In the user study, we had to limit the selection of the test classes to present to participants to two. Such a limited scope was required to ensure a reasonable compromise between the amount of classes to verify and the time required to participants. Before opting for the selection of two classes, we run a pilot study aimed at understanding the optimal amount of classes to consider in the study. The pilot was conducted with 10 software engineering researchers working within the lab of the third and last authors of the paper. The researchers have between 2 and 5 years of academic experience on software quality assurance and testing, with two of them who had previous experience in industry. In the pilot study, we verified the amount of time required by participants to assess five pairs of test classes generated by G-Mosa and MOSA. We could realize that after the first two pairs, not only the answers took significantly longer, but the overall quality of the assertions provided decreased. By interacting with the participants, we could understand that their level of attention significantly decreased after the first two evaluations due to the fatigue-effect. For this reason, we fixed the number of tasks for the actual participants to two. Nonetheless, further replications of the study aiming at corroborating our findings are already part of our future research agenda.

8 Conclusion

The ultimate goal of our research was to define a systematic strategy for the automatic generation of test code. In this paper, we started working toward this goal by implementing the concepts of intra-method and intra-class testing within a state-of-the-art automatic technique for test case generation like Mosa. One of the risks connected to these mechanisms is the decrease of effectiveness: by forcing our approach to generate intra-method tests we naturally limit its scope, potentially lowering the number of tangentially covered branches. It turned out that, instead, this was not the case. According to our results, G-Mosa provided test cases that are comparable in terms of both code and mutation coverage. Hence, it seems that it is actually possible to improve the inner-working of automatic test case generators by creating more granular tests that are still as effective as those produced by baseline techniques. The empirical results of our study also suggest that our approach to the generation provides rewards in terms of other desirable properties of test cases. The test cases produced by G-Mosa are indeed shorter, more maintainable, and understandable than those produced by Mosa. Hence, we can conclude that:

◎ Overall conclusion of our work.

The granular approach to the generation of test cases provides promising results. G-Mosa *tends to generate a larger amount of test cases with respect to the baseline, but these are significantly shorter. On the one hand, this property does not damage code and mutation coverage, which remain statistically comparable. On the other hand, it provides multiple benefits in terms of maintainability and understandability, potentially providing relevant implications to fields like fault localization and debugging.*

We consider this as a key result of our research, as it might potentially lead further researchers to consider the application of structured approaches that may generate test classes that are potentially more focused, comprehensible, and maintainable while keeping the same level of effectiveness.

The technique we proposed is also prepared to allow the generation at different granularity levels. Indeed, one can simply increase the number of production calls allowed in the first part of the generation, that we limit to one in this first concept, to generate tests at incremental levels of granularity. This would potentially have key implications, as the proposed strategy can be easily extended from a two-step (*i.e.*, *intra-method* + *intra-class*) to an n-step approach in which the number of calls allowed to methods of the class under test (CUT) is increased at each step. Since different number of calls to methods of the class under test corresponds to different paths on the state machine of the CUT, it would be possible to limit the length of the paths to execute on the state machine, thus providing shorter and more comprehensible tests for which it will be easier to generate an oracle. In this sense, our work poses the basis for the definition of a brand new way to generate test cases that might be of particular interest for the researchers working at intersection between software testing and software code quality.

Perhaps more importantly, when diving into the tests generated by the experimented techniques we found out that G-Mosa performed better than Mosa on large classes. In a real-case scenario, this becomes particularly important when a failing test must be diagnosed. As shown in literature [60, 73], developers use test cases to start the debugging activities and understand the nature of a failure: in this sense, the availability of smaller test cases that contain a lower amount of assertions might help developers in finding defects faster. More investigations into the implications of our technique for debugging are part of our future research agenda.

In addition, we also plan to exploit the granular nature of G-Mosa to perform multiple additional investigations. On the one hand, we plan to assess how the test cases generated by our techniques behave when considering the detection of real defects: in this respect, the use of Defects4J [39] as a database of real defects might be instrument, even though such an analysis might require some tuning and/or modifications to the inner-working of G-Mosa to fit computation constraints [25]. On the other hand, we plan to conduct further experimentation based on several granularity levels. Finally, we plan to implement our approach on top of a broader set of baselines as well as an *in-vivo* performance assessment involving real testing experts.

9 Credits

Fabiano Pecorelli: Technique design, Technique experimentation, User study design and execution, Data Curation, Data Analysis, Writing. **Giovanni Grano**: Technique design, Technique implementation, Technique experimentation, Data Curation, Writing. **Fabio Palomba**: Technique design, User study design and execution, Supervision, Writing. **Harald C. Gall**: Supervision, Writing - Review & Editing. **Andrea De Lucia**: Technique design, Supervision, Writing - Review & Editing.

10 Conflict of interest

The authors declare that they have no conflict of interest.

Data Availability Statement (DAS)

The data collected in the context of this study, along with the scripts used to analyze and generate data, charts, and plots discussed when addressing our research goals, are publicly available in our online appendix [5].

Acknowledgment

Fabio gratefully acknowledges the support of the Swiss National Science Foundation (SNSF) through the SNF Project No. PZ00P2_186090 (TED).

References

1. Afshan S, McMinn P, Stevenson M (2013) Evolving readable string test inputs using a natural language model to reduce human oracle cost. In: Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, IEEE Computer Society, Washington, DC, USA, ICST '13, pp 352–361, DOI 10.1109/ICST.2013.11, URL http://dx.doi.org/10.1109/ICST.2013.11
2. Ali S, Briand LC, Hemmati H, Panesar-Walawege RK (2009) A systematic review of the application and empirical investigation of search-based test case generation. IEEE Transactions on Software Engineering 36(6):742–762
3. Ammann P, Offutt J (2016) Introduction to software testing. Cambridge University Press
4. Anand S, Burke EK, Chen TY, Clark J, Cohen MB, Grieskamp W, Harman M, Harrold MJ, Mcminn P, Bertolino A, et al. (2013) An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software 86(8):1978–2001
5. Anonymous (2021) Toward granular automatic unit test case generation - online appendix https://figshare.com/s/34d39dae76ed68d57d18. URL https://figshare.com/s/34d39dae76ed68d57d18
6. Arcuri A (2019) Restful api automated test case generation with evomaster. ACM Transactions on Software Engineering and Methodology (TOSEM) 28(1):1–37
7. Arcuri A, Fraser G (2013) Parameter tuning or default values? an empirical investigation in search-based software engineering. Empirical Software Engineering 18(3):594–623
8. Baltes S, Diehl S (2016) Worse than spam: Issues in sampling software developers. In: Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement, pp 1–6
9. Barr ET, Harman M, McMinn P, Shahbaz M, Yoo S (2015) The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering 41(5):507–525, DOI 10.1109/TSE.2014.2372785
10. Beller M, Gousios G, Panichella A, Proksch S, Amann S, Zaidman A (2017) Developer testing in the ide: Patterns, beliefs, and behavior. IEEE Transactions on Software Engineering 45(3):261–284
11. Campos J, Ge Y, Fraser G, Eler M, Arcuri A (2017) An empirical evaluation of evolutionary algorithms for test suite generation. In: Proceedings of the 9th International Symposium on Search Based Software Engineering SSBSE 2017, pp 33–48, DOI 10.1007/978-3-319-66299-2_3
12. Ceccato M, Marchetto A, Mariani L, Nguyen CD, Tonella P (2015) Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency. ACM Transactions on Software Engineering and Methodology (TOSEM) 25(1):1–38
13. Conover W (1999) Practical nonparametric statistics, 3rd edn. Wiley series in probability and statistics, Wiley, New York, NY [u.a.]

14. Daka E, Campos J, Fraser G, Dorn J, Weimer W (2015) Modeling readability to improve unit tests. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, pp 107–118
15. Dietrich J, Schole H, Sui L, Tempero E (2017) Xcorpus–an executable corpus of java programs
16. Elish MO, Rine D (2006) Design structural stability metrics and post-release defect density: An empirical study. In: 2006 30th Annual International Computer Software and Applications Conference (COMPSAC'06 Supplement), IEEE, pp 1–8
17. Erdogmus H, Morisio M, Torchiano M (2005) On the effectiveness of the test-first approach to programming. IEEE Transactions on software Engineering 31(3):226–237
18. Ferrer J, Chicano F, Alba E (2012) Evolutionary algorithms for the multi-objective test data generation problem. Softw Pract Exper 42(11):1331–1362, DOI 10.1002/spe.1135, URL `http://dx.doi.org/10.1002/spe.1135`
19. Fowler M, Beck K (1999) Refactoring: improving the design of existing code. Addison-Wesley Professional
20. Fraser G, Arcuri A (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, New York, NY, USA, ESEC/FSE '11, pp 416–419, DOI 10.1145/2025113.2025179, URL `http://doi.acm.org/10.1145/2025113.2025179`
21. Fraser G, Arcuri A (2013) Whole test suite generation. IEEE Trans Softw Eng 39(2):276–291, DOI 10.1109/TSE.2012.14, URL `http://dx.doi.org/10.1109/TSE.2012.14`
22. Fraser G, Arcuri A (2014) A large-scale evaluation of automated unit test generation using evosuite. ACM Transactions on Software Engineering and Methodology (TOSEM) 24(2):1–42
23. Fraser G, Arcuri A (2015) 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. Empirical software engineering 20(3):611–639
24. Fraser G, Arcuri A (2015) Achieving scalable mutation-based generation of whole test suites. Empirical Software Engineering 20(3):783–812
25. Fraser G, Arcuri A (2016) Evosuite at the sbst 2016 tool competition. In: Proceedings of the 9th International Workshop on Search-Based Software Testing, pp 33–36
26. Fraser G, Staats M, McMinn P, Arcuri A, Padberg F (????) Does automated unit test generation really help software testers? a controlled empirical study. ACM Trans Softw Eng Methodol To Appear
27. Garousi V, Küçük B (2018) Smells in software test code: A survey of knowledge in industry and academia. Journal of systems and software 138:52–81
28. Goldberg DE (1989) Genetic Algorithms in Search, Optimization and Machine Learning, 1st edn. Addison-Wesley Longman Publishing Co., Inc.,

Boston, MA, USA
29. Grano G, Scalabrino S, Oliveto R, Gall H (2018) An empirical investigation on the readability of manual and generated test cases. In: Proceedings of the 26th International Conference on Program Comprehension, ICPC
30. Grano G, Laaber C, Panichella A, Panichella S (2019) Testing with fewer resources: An adaptive approach to performance-aware test case generation. IEEE Transactions on Software Engineering
31. Grano G, Palomba F, Di Nucci D, De Lucia A, Gall HC (2019) Scented since the beginning: On the diffuseness of test smells in automatically generated test code. Journal of Systems and Software 156:312–327
32. Grano G, De Iaco C, Palomba F, Gall HC (2020) Pizza versus pinsa: On the perception and measurability of unit test code quality. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 336–347
33. Gren L, Antinyan V (2017) On the relation between unit testing and code quality. In: 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, pp 52–56
34. Habchi S, Haben G, Papadakis M, Cordy M, Traon YL (2021) A qualitative study on the sources, impacts, and mitigation strategies of flaky tests. arXiv preprint arXiv:211204919
35. Harrold MJ, McGregor JD, Fitzpatrick KJ (1992) Incremental testing of object-oriented class structures. In: Proceedings of the 14th international conference on Software engineering, pp 68–80
36. Heckman JJ (1990) Selection bias and self-selection. In: Econometrics, Springer, pp 201–224
37. Henry S, Kafura D (1981) Software structure metrics based on information flow. IEEE transactions on Software Engineering (5):510–518
38. Hunt KJ, Shlomo N, Addington-Hall J (2013) Participant recruitment in sensitive surveys: a comparative trial of 'opt in'versus 'opt out'approaches. BMC Medical Research Methodology 13(1):1–8
39. Just R, Jalali D, Ernst MD (2014) Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 international symposium on software testing and analysis, pp 437–440
40. Knowles JD, Corne DW (2000) Approximating the nondominated front using the pareto archived evolution strategy. Evolutionary computation 8(2):149–172
41. Kudrjavets G, Nagappan N, Ball T (2006) Assessing the relationship between software assertions and faults: An empirical investigation. In: 2006 17th International Symposium on Software Reliability Engineering, IEEE, pp 204–212
42. Lakhotia K, Harman M, McMinn P (2007) A multi-objective approach to search-based test data generation. In: Proceedings of the 9th annual conference on Genetic and evolutionary computation, pp 1098–1105
43. von Lücken C, Barán B, Brizuela C (2014) A survey on multi-objective evolutionary algorithms for many-objective problems. Computational op-

timization and applications 58(3):707–756

44. McCabe TJ (1976) A complexity measure. IEEE Transactions on software Engineering (4):308–320
45. McMinn P (2004) Search-based software test data generation: A survey. Softw Test Verif Reliab 14(2):105–156, DOI 10.1002/stvr.v14:2, URL `http://dx.doi.org/10.1002/stvr.v14:2`
46. McMinn P (2004) Search-based software test data generation: a survey. Software Testing, Verification and Reliability 14(2):105–156
47. Myers GJ, Sandler C, Badgett T (2011) The art of software testing. John Wiley & Sons
48. Orso A, Silva S (1998) Open issues and research directions in object-oriented testing. In: Proceedings of the 4th International Conference on" Achieving Quality in Software: Software Quality in the Communication Society"(AQUIS'98)
49. Oster N, Saglietti F (2006) Automatic test data generation by multi-objective optimisation. In: International Conference on Computer Safety, Reliability, and Security, Springer, pp 426–438
50. Palomba F, Panichella A, Zaidman A, Oliveto R, De Lucia A (2016) Automatic test case generation: What if test code quality matters? In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA 2016, pp 130–141, DOI 10.1145/2931037.2931057, URL `http://doi.acm.org/10.1145/2931037.2931057`
51. Panichella A, Kifetew FM, Tonella P (2015) Reformulating branch coverage as a many-objective optimization problem. In: ICST, IEEE Computer Society, pp 1–10
52. Panichella A, Kifetew FM, Tonella P (2015) Reformulating branch coverage as a many-objective optimization problem. In: ICST, IEEE Computer Society, pp 1–10
53. Panichella A, Kifetew FM, Tonella P (2018) Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Transactions on Software Engineering 44(2):122–158
54. Panichella A, Kifetew FM, Tonella P (2018) Incremental control dependency frontier exploration for many-criteria test case generation. In: International Symposium on Search Based Software Engineering, Springer, pp 309–324
55. Papadakis M, Shin D, Yoo S, Bae DH (2018) Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, pp 537–548
56. Pecorelli F, Palomba F, De Lucia A (2021) The relation of test-related factors to software quality: A case study on apache systems. Empirical Software Engineering 26(2):1–42
57. Peruma A, Almalki K, Newman CD, Mkaouer MW, Ouni A, Palomba F (2020) Tsdetect: An open source test smells detection tool. In: Proceed-

ings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 1650–1654

58. Pezzè M, Young M (2008) Software testing and analysis: process, principles, and techniques. John Wiley & Sons
59. Pinto GH, Vergilio SR (2010) A multi-objective genetic algorithm to test data generation. In: 2010 22nd IEEE International Conference on Tools with Artificial Intelligence, IEEE, vol 1, pp 129–134
60. Ramler R, Wolfmaier K, Kopetzky T (2013) A replicated study on random test case generation and manual unit testing: How many bugs do professional developers find? In: 2013 IEEE 37th Annual Computer Software and Applications Conference, IEEE, pp 484–491
61. Reid B, Wagner M, d'Amorim M, Treude C (2022) Software engineering user study recruitment on prolific: An experience report. arXiv preprint arXiv:220105348
62. Rojas JM, Campos J, Vivanti M, Fraser G, Arcuri A (2015) Combining multiple coverage criteria in search-based unit test generation. In: Barros M, Labiche Y (eds) Search-Based Software Engineering, Springer International Publishing, Cham, pp 93–108
63. Rojas JM, Fraser G, Arcuri A (2015) 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, pp 338–349
64. Scalabrino S, Grano G, Di Nucci D, Oliveto R, De Lucia A (2016) Search-based testing of procedural programs: Iterative single-target or multi-target approach? In: Sarro F, Deb K (eds) Search Based Software Engineering, Springer International Publishing, Cham, pp 64–79
65. Spadini D, Palomba F, Zaidman A, Bruntink M, Bacchelli A (2018) On the relation of test smells to software code quality. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 1–12
66. Spinellis D (2005) Tool writing: a forgotten art?(software tools). IEEE Software 22(4):9–11
67. Subramanyam R, Krishnan MS (2003) Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. IEEE Transactions on software engineering 29(4):297–310
68. Van Deursen A, Moonen L, van den Bergh A, Kok G (2001) Refactoring test code. In: Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001), pp 92–95
69. Wang S, Offutt J (2009) Comparison of unit-level automated test generation tools. In: 2009 International Conference on Software Testing, Verification, and Validation Workshops, IEEE, pp 210–219
70. Wappler S, Lammermann F (2005) Using evolutionary algorithms for the unit testing of object-oriented software. In: Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, Association

for Computing Machinery, New York, NY, USA, GECCO ’05, pp 1053–1060, DOI 10.1145/1068009.1068187, URL `https://doi.org/10.1145/1068009.1068187`

71. Williams L, Kudrjavets G, Nagappan N (2009) On the effectiveness of unit test automation at microsoft. In: 2009 20th International Symposium on Software Reliability Engineering, IEEE, pp 81–89
72. Zamani S, Hemmati H (2020) A cost-effective approach for hyper-parameter tuning in search-based test case generation. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 418–429
73. Zeller A (2009) Why programs fail: a guide to systematic debugging. Elsevier