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

A Detailed Investigation of the Effectiveness of Whole Test Suite Generation

José Miguel Rojas · Mattia Vivanti · Andrea Arcuri · Gordon Fraser

Received: date / Accepted: date

Abstract A common application of search-based software testing is to generate test cases for all goals defined by a coverage criterion (e.g., lines, branches, mutants). Rather than generating one test case at a time for each of these goals individually, *whole test suite generation* optimizes entire test suites towards satisfying all goals at the same time. There is evidence that the overall coverage achieved with this approach is superior to that of targeting individual coverage goals. Nevertheless, there remains some uncertainty on (a) whether the results generalize beyond branch coverage, (b) whether the whole test suite approach might be inferior to a more focused search for some particular coverage goals, and (c) whether generating whole test suites could be optimized by only targeting coverage goals not already covered. In this paper, we perform an in-depth analysis to study these questions. An empirical study on 100 Java classes using three different coverage criteria reveals that indeed there are some testing goals that are only covered by the traditional approach, although their number is only very small in comparison with those which are exclusively covered by the whole test suite approach. We find that keeping an archive of already covered goals along with the tests covering them and focusing the search on uncovered goals overcomes this small drawback on larger classes, leading to an improved overall effectiveness of whole test suite generation.

Keyword: Automated test generation, unit testing, search-based testing, EvoSuite

J. M. Rojas / G. Fraser
Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello, S1 4DP, Sheffield, UK, E-mail: j.rojas@sheffield.ac.uk, gordon.fraser@sheffield.ac.uk

M. Vivanti
Università della Svizzera italiana (USI), Lugano, Switzerland, E-mail: mattia.vivanti@usi.ch

A. Arcuri
Scienta, Norway, and SnT Centre, University of Luxembourg, Luxembourg E-mail: aa@scienta.no

1 Introduction

Search-based software engineering has been applied to numerous different software development activities [16], and software testing is one of the most successful of these [1, 22]. One particular software testing task for which search-based techniques are well suited is the automated generation of unit tests. For example, there are search-based tools like AUSTIN for C programs [20] or EVOSUITE for Java programs [9].

In search-based software testing, the testing problem is cast as a search problem. For example, common scenarios are to generate a set of test cases maximizing their code coverage or maximizing their fault detection capability. A code coverage criterion describes a set of typically structural aspects of the system under test (SUT) which should be exercised by a test suite, for example all statements, lines or branches. Mutation testing is traditionally used to assess the fault detection capability of a test suite: artificial faults are inserted in the SUT one at a time and the ability of the test suite to detect such faults is measured. For both cases, the search space would consist of all possible data inputs for the SUT. A search algorithm (e.g., a genetic algorithm) is then used to explore this search space to find the input data that maximize the given objective (e.g., cover as many branches as possible or achieve the highest possible mutation score).

Traditionally, to achieve this testing objective a search is carried out on each individual coverage goal [22] (e.g., a branch). To guide the search, the fitness function exploits information like the approach level [28] and branch distance [18]. It may happen that during the search for a coverage goal there are other goals that can be "accidentally" covered, and by keeping such test data one does not need to perform search for those accidentally covered goals. However, there are several potential issues with such an approach:

- *Search budget distribution:* If a coverage goal is infeasible, then all search effort to try to cover it would be wasted (except for any other coverage goals accidentally covered during the search). Unfortunately, determining whether a goal is feasible or not is an undecidable problem. If a coverage goal is trivial, then it will typically be covered by the first random input. Given a set of coverage goals and an overall available budget of computational resources (e.g., time), how to assign a search budget to the individual goals to maximize the overall coverage?
- *Coverage goal ordering:* Unless some smart strategies are designed, the search for each coverage goal is typically independent, and potentially useful information is not shared between individual searches. For example, to cover a nested branch one first needs to cover its parent branch, and test data for the latter could be used to help the search for the nested branch (instead of starting from scratch). In this regard, the order in which coverage goals are sought can have a large impact on final performance.

To overcome these issues, previous work introduced the *whole test suite approach* [12, 13]. Instead of searching for a single test for each individual coverage goal in sequence, the search problem is changed to a search for a set of tests that covers all coverage goals at the same time; accordingly, the fitness function guides to cover all goals. The advantage of such an approach is that both the questions of how

to distribute the available search budget between the individual coverage goals, and in which order to target those goals, disappear. With the whole test suite approach, large improvements have been reported for both branch coverage [12] and mutation testing [13].

Despite this evidence of higher overall coverage, the question remains of how the use of whole test suite generation influences individual coverage goals. In particular:

- Even if the whole test suite approach covers more goals, those are not necessarily going to be a superset of those that the traditional approach would cover. Is the higher coverage due to more easy goals being covered? Is the coverage of other goals adversely affected? Are there goals that the traditional one goal at a time approach can cover and the whole approach can not? Although higher coverage might lead to better regression test suites, for testing purposes some coverage goals might be more "valuable" than others. So, from a practical point of view, preferring the whole test suite approach over the traditional one may not necessarily lead to improvement in testing effectiveness.
- When generating individual tests, once a coverage goal is satisfied, it is no longer involved in test generation, and resources are invested only on uncovered goals. In whole test suite generation, all goals, including those already covered during the search, are part of the optimization until the search ends. For example, after mutation in the genetic algorithm a test suite may cover a new branch, but if the mutation meant that two already covered goals are "lost" by that test suite, the fitness evaluation would not consider this new test an improvement. Does this affect whole test suite optimization in practice? An easy solution to overcome this problem would be to keep a test "archive" for the already covered goals, and focus the search only on those goals not yet covered.

In this paper, we aim to empirically study these two aspects in detail. We investigate whether there are specific coverage goals for which the traditional approach is better and, if that is the case, we attempt to characterise those scenarios. Based on an empirical study performed on 100 Java classes, our study shows that indeed there are cases in which the traditional approach provides better results. However, those cases are rare (nearly one hundred times less) compared to the cases in which only the whole test suite approach is able to cover the goals. Using an archive does improve performance on average, but it also has negative side-effects on some testing targets.

This paper is an extension to earlier work [5]. In particular, in this paper we analyze three different coverage criteria, namely line coverage, branch coverage and weak mutation, instead of just branch coverage. Furthermore, we investigate the effects of using a test archive during whole test suite generation by replicating the previous experiments [5] and performing another set of experiments on a sample of more complex classes.

The structure of this paper is as follows. Section 2 provides background information, whereas the whole test suite approach is discussed in details in Section 3. The performed empirical study is presented in Section 4. A discussion on the threats to the validity of the study follows in Section 5. Finally, Section 6 concludes the paper.

2 Background

Search-based techniques have been successfully used for test data generation (Ali *et al.* [1] and McMinn [22] amply surveyed this topic). The application of search for test data generation can be traced back to the 70s [23], and later the key concepts of *branch distance* [18] and *approach level* [28] were introduced to help search techniques in generating the right test data.

More recently, search-based techniques have also been applied to test object-oriented software (e.g., [14, 25–27]). One specific issue that arises in this context is that test cases are sequences of calls, and their length needs to be controlled by the search. Since the early work of Tonella [26], researchers have tried to deal with this problem, for example by penalizing the length directly in the fitness function. However, longer test sequences can lead to achieve higher code coverage [2], yet properly handling their growth/reduction during the search requires special care [11].

Most approaches described in the literature aim at generating test suites that achieve as high as possible branch coverage. In principle, any other coverage criterion is amenable to automated test generation. For example, mutation testing [17] is often considered a worthwhile test goal, and has been used in a search-based test generation environment [14].

When test cases are sought for individual goals in such coverage-based approaches, it is important to keep track of the accidental collateral coverage of the remaining goals. Otherwise, it has been proven that random testing would fare better under some scalability models [6]. Recently, Harman *et al.* [15] proposed a search-based multi-objective approach in which, although each coverage goal is still targeted individually, there is the secondary objective of maximizing the number of collateral goals that are accidentally covered. However, no particular heuristic is used to help covering these other coverage goals.

All approaches mentioned so far target a single test goal at a time – this is the predominant method. There are some notable exceptions in search-based software testing. The works of Arcuri and Yao [7] and Baresi *et al.* [8] use a single sequence of function calls to maximize the number of covered branches while minimizing the length of such a test case. A drawback of such an approach is that there can be conflicting testing goals, and it might be impossible to cover all of them with a single test sequence regardless of its length.

The whole test suite approach [12, 13] was devised to overcome those issues. In this approach, instead of evolving individual tests, whole test suites are evolved, with a fitness function that considers all the coverage goals at the same time. Promising results were obtained for both branch coverage [12] and mutation testing [13].

As an alternative to the whole test suite approach, Panichella *et al.* [24] recently reformulated the generation of test suites for branch coverage as a many-objective optimization problem. The idea is to simultaneously minimize the distance between a test case and each uncovered branch in the class under test. To this end, an archive of solutions is used to store test cases which cover new branches, continuing the search with only uncovered branches as target goals. To which extent the reported benefits stem from the many-objective reformulation of the problem or from the use of this

archiving mechanism remains unclear. The archive-based whole test suite approach presented in this paper opens the way for further empirical evaluations.

3 Whole Test Suite Generation

To make this paper self-contained, in this section we provide a summarized description of the traditional approach used in search-based software testing, the whole test suite approach, and the archive-based extension of the whole test suite approach. For more details on the traditional approach, the reader can for example refer to McMinn [22] and Wegener *et al.* [28]. For the whole test suite approach, the reader can refer to Fraser and Arcuri [12, 13].

Given a SUT, assume X to be the set of coverage goals we want to automatically cover with a set of test cases T (i.e., a test suite). Coverage goals could be for example branches if we are aiming at branch coverage, or any other element depending on the chosen coverage criterion (e.g., mutants in mutation testing). In this paper, we consider three coverage criteria: The dominant coverage criterion in the literature is branch coverage, hence we include it as well. In practice, statement coverage often serves as a simpler alternative when practitioners measure the coverage of their tests. However, many modern bytecode-based tools [21] measure coverage on lines of code as a proxy for statement coverage. Consequently, we use line coverage as the second criterion in this paper. Finally, the third criterion we consider is weak mutation testing [13], where each mutants is expected to lead to a state change.

3.1 Generating Tests for Individual Coverage Goals

Given $|X| = n$ coverage goals, traditionally there would be one search for each of them. To give more gradient to the search (instead of just counting "yes/no" on whether a goal is covered), usually the approach level $\mathcal{A}(t,x)$ and branch distance $d(t,x)$ are employed for the fitness function [22, 28]. The approach level $\mathcal{A}(t,x)$ for a given test t on a coverage goal $x \in X$ is used to guide the search toward the target goal. It is determined as the minimal number of control dependent edges in the control dependency graph between the target goal and the control flow represented by the test case. The branch distance $d(t,x)$ is used to heuristically quantify how far a predicate in a branch x is from being evaluated as true. In this context, the considered predicate x_c is taken for the closest control dependent branch where the control flow diverges from the target branch.

Branch Coverage. The *Branch Coverage* fitness function to minimize the approach level and branch distance between a test t and a branch coverage goal x is defined as:

$$f(t,x) = \mathcal{A}(t,x) + \nu(d(t,x_c)) \,, \tag{1}$$

where ν is any normalizing function in the $[0,1]$ range [3]. For example, consider this trivial function:

```
public static void foo(int z){
      if(z > 0)
            if(z > 100)
                  if(z > 200)
                        ...; //target
}
```

With a test case t_{50} having the value $z = 50$, the execution would diverge at the second if-condition, hence the resulting fitness function for the target $x_{z>200}$ would be

$$f(t_{50},x_{z>200}) = 1 + \nu(|50 - 100| + 1) = 1 + \nu(51) \ , \tag{2}$$

where the first component is the approach level (i.e., 1) and the second component determines the distance to executing the "then" branch of the second if-condition. Now, the test with $z = 50$ would have higher fitness (i.e., it would be worse) than the following test case which uses $z = 101$, due to the lower approach level (i.e., 0) and the normalization of the branch distance values:

$$f(t_{101},x_{z>200}) = 0 + \nu(|101 - 200| + 1) = 0 + \nu(100) \ . \tag{3}$$

Line Coverage. Once the control flow graph of the SUT is constructed, the problem of determining the closeness to a line being covered boils down to determining how close the basic block it belongs to is from being covered. Hence, the definition of *Line Coverage* fitness function is identical to the one for *Branch Coverage*.

Weak Mutation Testing. Mutation testing consists in applying small changes, one at a time, in the code of the SUT and then checking if a test distinguishes between the original SUT and the changed versions, called *mutants*. Weak mutation considers a mutant covered ("killed") if the execution of the test on the mutant is observably different from its execution on the original SUT, that is, if *state infection* is reached by the test. Otherwise, the mutant remains uncovered ("alive"). The *Weak Mutation Testing* fitness function for a test t and a mutant μ is hence given by the sum of the approach level, branch distance and the minimal infection distance function $d_{inf}(t,\mu)$:

$$f(t,\mu) = \mathcal{A}(t,\mu) + \nu(d(t,\mu_c)) + \nu(d_{inf}(t,\mu)) \ , \tag{4}$$

where the $d_{inf}(t,\mu)$ estimates the distance to an execution of the mutant in which state infection occurs. Intuitively, if the mutation (i.e., the specific instruction where the mutant differs from the original SUT) is *not* executed, the infection distance is 1.0 (maximum). If the mutation is executed, on the other hand, the minimal state infection distance is specific to each mutation operator [14]. For simple operators such as for instance *Delete Statement* and *Insert Unary Operator*, the minimal infection distance solely depends on the execution distance (that is, it is 0.0 whenever the approach level and branch distances are 0.0 as well). For other operators, the infection distance depends on the comparison of the executions of the original and mutated code; for example, the minimal infection distance for the *Replace Arithmetic Operator* is 0.0 only when the outcome of the original arithmetic operation and the outcome of the mutation differ, and 1.0 when the outcomes are the same.

While implementing this traditional approach, we tried to derive a faithful representation of current practice, which means that there are some optimizations proposed in the literature which we did not include:

- New test cases are only generated for goals that have not already been covered through collateral coverage of previously created test cases. However, we do not evaluate the collateral coverage of all individuals during the search, as this would add a significant overhead, and it is not clear what effects this would have given the fixed timeout we used in our experiments.
- When applying the one goal at a time approach, a possible improvement could be to use a *seeding* strategy [28]. During the search, we could store the test data that have good fitness values on coverage goals that are not covered yet. These test data can then be used as starting point (i.e., for seeding the first generation of a genetic algorithm) in the successive searches for those uncovered goals. However, we decided not to implement this, as Wegener *et al.* [28] do not provide sufficient details to reimplement the technique, and there is no conclusive data regarding several open questions; for example, potentially a seeding strategy could reduce diversity in the population, and so in some cases it might in fact reduce the overall performance of the search algorithm.
- The order in which coverage goals are selected might also influence the result. As in the literature usually no order is specified (e.g., [15, 26]), we selected the branches in random order. However, in the context of procedural code approaches to prioritize coverage goals have been proposed, e.g., based on dynamic information [28]. However, the goal of this paper is neither to study the impact of different orders, nor to adapt these prioritization techniques to object-oriented code.
- In practice, when applying a single goal strategy, one might also bootstrap an initial random test suite to identify the trivial test goals, and then use a more sophisticated technique to address the difficult goals; here, a difficult, unanswered question is when to stop the random phase and start the search.

3.2 Whole Test Suite Generation

For the whole test suite approach, we used exactly the same implementation used by Fraser and Arcuri [12, 13]. In the *Whole* approach, the approach level $\mathcal{A}(t,x)$ is not needed in the fitness function, as generally all control dependencies are included in the optimization target, and thus the approach level is optimized to 0 for all cases automatically.

The *Branch Coverage* fitness function to minimize for a set of test cases T on a set of branches B is:

$$f_{BC}(T,B) = \sum_{b \in B} d(T,b) \ , \tag{5}$$

where $d(T,b)$ is defined as:

$$d(T,b) = \begin{cases} 0 & \text{if branch } b \text{ has been covered,} \\ \nu(d_{min}(t \in T,b)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases} \quad (6)$$

Similarly, the *Line Coverage* fitness function to minimize for a set of test cases T on a set of source code lines L is:

$$f_{LC}(T,L) = \nu(|L| - |L_{Covered}|) + \sum_{b_k \in B_{\text{CUT}}} d(T,b) \ , \quad (7)$$

where L is the set of all non-comment lines of code in the CUT, $L_{Covered}$ is the subset of non-comment lines of code *covered* with the execution of all the tests in T, and B_{CUT} is the set of branches that are control dependencies (i.e., branches that have child nodes in the control dependence tree).

The *Weak Mutation* fitness function optimizes test suites for weak mutation score:

$$f_{WM}(T,\mathcal{M}) = \sum_{\mu \in \mathcal{M}} d_w(T,\mu) \ , \quad (8)$$

where $\mathcal{M}$ is the set of all mutants generated for the SUT using a set of mutation operators [13] and $d_w(T,\mu)$ guides the search by calculating the state infection distance to a mutant μ:

$$d_w(T,\mu) = \begin{cases} \nu(d_{inf}(T,\mu)) & \text{if mutant } \mu \text{ was reached,} \\ 1 & \text{otherwise.} \end{cases} \quad (9)$$

Note that the total set of coverage goals for any of the fitness functions defined could be considered as different objectives. Instead of linearly combining them in a single fitness score, a multi-objective algorithm could be used. However, a typical class can have hundreds if not thousands of objectives (e.g., branches), making a multi-objective algorithm not ideal due to scalability problems. Specialized many-objective fitness functions may be applicable [24], but have only been considered for branch coverage so far.

3.3 Archive-based Whole Test Suite Generation

The fitness functions shown above all assume that all coverage goals are targeted at the same time. That is, even when we already have a test for a coverage goal, it will still influence the search, as an optimal test suite needs to consist of tests for all goals. This potentially can have adverse effects on the effectiveness of the search: A share of the search budget will be devoted to covering goals that have already been covered in the past, and the search may at times focus less on exploring new, completely uncovered code, but more on exploiting existing coverage. For example, even if a

newly covered branch is an important control dependency required to cover a large section of the code, if the mutation that led to coverage of this branch "lost" coverage of several other branches already covered in the past, then the fitness function will not reward this individual. On the other hand, a change that reduces the branch distance towards several uncovered goals may result in a better fitness value even if this reduces coverage (this effect can be observed by EVOSUITE's progress bar not increasing monotonically, but sometimes jumping back to lower coverage values).

These issues can be overcome by using a concept that is common in multi-objective optimization: an archive of individuals. During fitness evaluation, each time we discover that a new branch (or line or mutant) has been covered, we add the covering test and the covered goal to an archive. The fitness function is modified to no longer take these covered goals into account. However, it is important that this modification of the fitness function is not done in the middle of the fitness evaluation of one test suite, or the creation of a new population in a standard genetic algorithm approach, as it would make fitness values between individuals inconsistent. Therefore, we modify the fitness function only at the end of an iteration, after the fitness of all individuals has been evaluated. The modification simply consists of removing already covered goals. For example, for branch coverage the fitness function becomes:

$$f_{BC}(T,B) = \sum_{b \in B \setminus C} d(T,b) \; , \tag{10}$$

where C is the set of goals already covered in the archive.

The impact of not taking already covered goals into account for fitness computation can be demonstrated in the following example. Assume there are two test suites T_1 and T_2 and a set of branch goals $B = \{b_1,b_2\}$, such that $d(T_1,b_1) = 0$, $d(T_1,b_2) = 0.7$, and $d(T_2,b_1) = 1$, $d(T_2,b_2) = 0.1$. Using the fitness function defined in Equation 5, we have $f_{BC}(T_1,B) = 0+0.7 = 0.7$ and $f_{BC}(T_2,B) = 1+0.1 = 1.1$. Hence, T_1, with a lower fitness value, is a fitter individual than T_2 and has higher chances of remaining in the population in further generations than T_2, even though T_2 is much closer than T_1 to covering branch b_2. In contrast, by using the archive-based fitness function definition (Equation 10), the branch b_1 is omitted in the fitness computation and we have $f_{BC}(T_1,B) = 0.7$ and $f_{BC}(T_2,B) = 0.1$. Therefore, T_2 is a fitter individual, and by selecting it the search is more likely to evolve a test suite covering b_2 in future generations.

At the end of the search, by definition the best individual of the genetic algorithm population will not cover any of the remaining coverage objectives (it may cover any number of other goals already covered in the archive). Thus, the test suite used by EVOSUITE for its postprocessing steps (e.g., minimization, assertion generation) is not taken from the search population, but is a test suite consisting of all the tests in the archive.

The simplicity of the fitness functions for whole test suite generation is based on the assumption that all coverage goals are targeted at the same time. That is, the approach level is not included in the fitness function because all control dependencies are part of the optimization. If this assumption no longer holds and the fitness function only targets a subset of the coverage goals, then this potentially results in a lack

of guidance, for example if some of the control dependencies of code not yet covered are no longer optimized for. There are different ways to address this issue; for example, the approach level could be included in the fitness function to provide the missing guidance. A simpler solution is provided by ensuring that the genetic material covering the control dependencies is not lost, even if the coverage goals are removed from the fitness function. EvoSuite achieves this in two ways. In the simplest case, EvoSuite keeps the tests in the population even after removing a coverage goal. Moreover, EvoSuite's search operators further try to exploit the archive by creating, with a certain probability, new tests by mutating tests from the archive rather than always adding new random tests. These two simple mechanisms suffice, although more intelligent ways of exploiting the archive, e.g., alternative seeding strategies, could be explored in the future.

Besides the fitness function, the search operators (e.g., mutation of tests) are also intended to optimize for all coverage goals. For example, in EvoSuite, if a new statement is inserted during mutation or generation of a random test, then with a probability of 50% this is a call on an existing object in the test, else the new statement is a call to a method of the CUT. The choice of CUT method is made randomly with a uniform distribution, based on the assumption that all methods need to be covered. However, once all branches (or lines or mutants) in a method have been covered, then inserting a call to the method will not lead to an improvement of the fitness value (unless the call leads to a state change, for which the other 50% probability of call insertion is intended). Therefore, as an optimization of the search operators, we modify the selection of CUT methods to a random choice out of only those methods that are not yet fully covered.

4 Empirical Study

In this paper, we carried out an empirical study to compare the traditional one goal at a time approach (*OneGoal*), the whole test suite approach (*Whole*) and the whole test suite approach using the archive (*Archive*). For each of the test coverage criteria *Branch*, *Line* and *Mutation*, we aim at answering the following research questions:

RQ1: Are there coverage goals in which *OneGoal* performs better than *Whole*?

RQ2: How many coverage goals found by *Whole* get missed by *OneGoal*?

RQ3: Which factors influence the relative performance of *Whole* and *OneGoal*?

RQ4: How does using an archiving solution, *Archive*, influence the performance of *Whole*?

4.1 Experimental Setup

For the experiments, we randomly chose 100 Java classes from the SF100 corpus [10], which is a collection of 100 projects randomly selected from the SourceForge open source software repository. We randomly selected from SF100 to avoid possible bias in the selection procedure, and to have higher confidence to generalize our results to other Java classes as well. The domain of the selected classes varies;

for example, there are classes dealing with lexical analysis (*XPathLexer* in project 24_saxpath), visual components (e.g., *SiteListPanel* in 35_corina), multi-threading (*BlockThread* in 78_caloriecount) and complex differential geometry operations (*LocalDifferentialGeometry* in 89_jiggler). In total, the selected 100 classes contain 2,383 branches, 3,811 lines and 12,473 mutants, which we consider as test goals.

The SF100 corpus contains more than 11,000 Java classes. We only used 100 classes instead of the entire SF100 due to the type experiments we carried out: a large number of classes, multiple configurations and a large number of repetitions. In particular, for each class in the selected sample we ran EVOSUITE in three modes:

(a) The one goal at a time approach (*OneGoal*)
(b) The whole test suite approach (*Whole*)
(c) The archive-based whole test suite approach (*Archive*)

The three modes were used in combination with the three test coverage criteria under study, i.e, *Branch*, *Line* and *Mutation*, which are implemented as fitness functions in EVOSUITE, giving a total of nine configurations. Each experiment consists in running one particular configuration on one of the sampled classes. To take randomness into account, each experiment was repeated 500 times, for a total of $100 \times 9 \times 500 = 450{,}000$ runs of EVOSUITE.

When choosing how many classes to use in a case study, there is always a tradeoff between the number of classes and the number of repeated experiments. On one hand, a higher number of classes helps to generalize the results. On the other hand, a higher number of repetitions helps to better study in detail the differences on specific classes. For example, given the same budget to run the experiments, we could have used 10,000 classes and 5 repetitions. However, as we want to study the "corner cases" (i.e., when one technique completely fails while the other compared one does produce results), we gave more emphasis on the number of repetitions to reduce the random noise in the final results.

Each experiment was run for up to two minutes (the search on a class was also stopped once 100% coverage was achieved). Therefore, in total the entire case study had an upper bound of $450{,}000 \times 2/(24 \times 60) = 625$ days of computational resources, which required a large cluster to run[1]. When running the *OneGoal* approach, the search budget (i.e., the two minutes) is equally distributed among the coverage goals in the SUT. When the search for a coverage goal finishes earlier (or a goal is accidentally covered by a previous search), the remaining budget is redistributed among the other goals still to cover.

Observe that the experimental setup used in this journal extension differs from the one used for the experiments reported in the original paper [5]. First, the number of configurations under study increased from two (*OneGoal* vs *Whole*, both for *Branch Coverage* only) to nine ($\{$*OneGoal*, *Whole*, *Archive*$\} \times \{$*Branch Coverage*, *Line Coverage*, *Weak Mutation Score*$\}$). To accommodate for this setup, less repetitions were run for each configuration, namely 500 instead of 1,000. Furthermore, the search budget used for each run was reduced from three to two minutes. Importantly, the

[1] The high computational power needed to run these exhaustive experiments does not reflect the actual requirements of the approach in a day-to-day industrial scenario, where unit tests for a class under test can be automatically generated within seconds using the available tool interfaces.

EvoSuite tool has evolved, bugs were fixed and improvements were made, hence it is expected that the exact coverage values may vary slightly with respect to the earlier study.

Given the nature of the *Archive* approach, it is expected that it will deliver larger benefits when applied to more complex classes. To investigate whether that is actually the case, we conducted a second experiment of reduced magnitude targeting a more challenging set of classes. We selected from each project in the SF100 corpus the class with the highest number of branches (full list with details is in Table 8 in the Appendix). The three approaches were evaluated for each of the test criteria on this case study as well, but only 30 repetitions were run. As discussed before, the results obtained for a lower number of repetitions may be influenced by randomness. However, in this case even this small number of repetitions is expected to suffice in demonstrating the effects of using the *Archive* approach, as they are only used to provide more support to our main results. In total, this added a further $100 \times 9 \times 30 = 27{,}000$ runs of EvoSuite.

To properly analyse the randomized algorithms used in this paper, we followed the guidelines proposed by Arcury and Briand [4]. In particular, when branch coverage values were compared, statistical differences were measured with the Wilcoxon-Mann-Whitney U-test, where the effect size was measured with the Vargha-Delaney $\hat{A}_{12}$. An $\hat{A}_{12} = 0.5$ means no difference between the two compared algorithms.

When checking how often a goal was covered, because whether or not a goal is covered is a binary variable, we used the Fisher exact test. As effect size, we used the odds ratios, with a $\delta = 1$ correction to handle the zero occurrences. When there is no difference between two algorithms, then the odds ratio is equal to one. Note, in some of the graphs we rather show the natural logarithm of the odds ratios, and this is done only to simplify their representation.

4.2 RQ1: Are there coverage goals in which *OneGoal* performs better than *Whole*?

Tables 1-3 show the average coverage results obtained for each of the test criteria on the selected 100 Java classes. The results in Table 1 confirm previous results [12]: the *Whole* test suite approach leads to higher branch coverage. In this case, the average branch coverage increases from 63% to 78%, with a 0.67 effect size. The *Whole* approach leads to significantly higher branch coverage for 45% of the classes, and to lower coverage in none of them.

Results for *Line Coverage* are also positive: Table 2 shows that the *Whole* approach leads to a 16% average increase, from 62% to 78% with a 0.69 effect size. We observed an individual statistically significant improvement for 47 classes. Whereas there is no class with significantly lower line coverage, class *JSListSubstitution* in project 85_shop is worth looking into since it shows a decrease in coverage with an effect size of 0.47. Figure 1 lists the relevant code from this class.

By default, EvoSuite uses the length of the test suite as a secondary optimization objective for *Whole* test suite generation. The general advantage is that resulting

```
class JSListSubstitution extends Vector
{
  ...
  public void print()
  {
     JSUtil.print("(");

     JSSubstitution s1;
     Enumeration s = elements();
     while(s.hasMoreElements())
     {
      s1 = (JSSubstitution) s.nextElement();
      if( s1!=null)
       s1.print();
     }
     JSUtil.println(")");
  }
}
```

Fig. 1: Excerpt of class *JSListSubstitution*: Using test suite sizes as a secondary optimization objective makes it harder for *Whole* to generate a data structure that satisfies the condition of the if-statement, hence often failing to cover the "then" branch.

Table 1: For each class with statistically significant results, the table reports the average *Branch Coverage* obtained by the *OneGoal* approach and by the *Whole* approach.

Project	Class	OneGoal	Whole	$\hat{A}_{12}$	p-value
13_jdbacl	AbstractTableMapper	0.20	0.71	0.97	< 0.001
18_jsecurity	IniResource	0.11	0.73	0.99	< 0.001
18_jsecurity	ResourceUtils	0.26	0.80	1.00	< 0.001
18_jsecurity	DefaultWebSecurityManager	0.02	0.42	1.00	< 0.001
21_geo-google	AddressToUsAddressFunctor	0.00	0.55	0.99	< 0.001
24_saxpath	XPathLexer	0.10	0.70	1.00	< 0.001
32_httpanalyzer	ScreenInputFilter	0.58	0.83	0.77	< 0.001
35_corina	TRML	0.00	0.20	1.00	< 0.001
35_corina	SiteListPanel	0.00	0.00	0.94	< 0.001
43_lilith	EventIdentifier	0.98	1.00	0.53	< 0.001
43_lilith	LogDateRunnable	0.36	0.59	0.69	< 0.001
44_summa	FacetMapSinglePackedFactory	0.01	0.46	0.99	< 0.001
45_lotus	Phase	0.97	1.00	0.54	< 0.001
46_nutzenportfolio	KategorieDaoService	0.00	0.16	1.00	< 0.001
54_db-everywhere	Select	0.00	0.08	0.95	< 0.001
58_fps370	MouseMoveBehavior	0.08	0.53	0.99	< 0.001
61_noen	DataType	0.00	1.00	1.00	< 0.001
61_noen	WatchDog	0.22	0.54	0.87	< 0.001
62_dom4j	STAXEventReader	0.00	0.01	1.00	< 0.001
62_dom4j	CloneHelper	0.77	1.00	0.82	< 0.001
62_dom4j	PerThreadSingleton	0.66	0.85	0.63	< 0.001
66_openjms	SecurityConfigurationDescriptor	0.27	0.68	0.98	< 0.001
66_openjms	And	0.90	1.00	0.68	< 0.001
66_openjms	BetweenExpression	0.23	0.87	0.98	< 0.001
69_lhamacaw	CategoryStateEditor	0.00	0.08	0.95	< 0.001
70_echodep	PackageDissemination	0.00	0.09	0.99	< 0.001
74_fixsuite	ListView	0.06	0.10	0.71	< 0.001
75_openhre	User	0.19	0.98	1.00	< 0.001
75_openhre	HL7SegmentMapImpl	0.99	1.00	0.51	< 0.001
78_caloriecount	BudgetWin	0.01	0.12	0.99	< 0.001
78_caloriecount	ArchiveScanner	0.01	0.63	1.00	< 0.001
78_caloriecount	RecordingEvent	0.75	1.00	0.96	< 0.001
78_caloriecount	BlockThread	0.46	0.82	0.93	< 0.001
79_twfbplayer	BattlefieldCell	0.00	0.36	0.95	< 0.001
80_wheelwebtool	Block	0.08	0.81	0.98	< 0.001
84_ifx-framework	ChkOrdInqRs_Type	0.92	1.00	0.69	< 0.001
84_ifx-framework	PassbkItemInqRs_Type	0.98	1.00	0.52	< 0.001
85_shop	JSListSubstitution	0.90	0.97	0.59	< 0.001
88_jopenchart	InterpolationChartRenderer	0.00	0.05	0.96	< 0.001
89_jiggler	SignalCanvas	0.19	0.97	1.00	< 0.001
89_jiggler	ImageOutputStreamJAI	0.05	0.79	0.99	< 0.001
89_jiggler	LocalDifferentialGeometry	0.03	0.32	0.99	< 0.001
92_jcvi-javacommon	PhdFileDataStoreBuilder	0.21	0.80	0.99	< 0.001
93_quickserver	SimpleCommandSet	0.66	0.83	0.65	< 0.001

Continued on next page

Table 1: Continued.

Project	Class	OneGoal	Whole	$\hat{A}_{12}$	p-value
93_quickserver	AuthStatus	0.12	0.33	0.80	< 0.001
Average		0.63	0.78	0.67	

* There were 53 classes with no statistically significant difference

Table 2: For each class with statistically significant results, the table reports the average *Line Coverage* obtained by the *OneGoal* approach and by the *Whole* approach.

Project	Class	OneGoal	Whole	$\hat{A}_{12}$	p-value
13_jdbacl	H2Util	0.64	0.81	0.62	< 0.001
13_jdbacl	AbstractTableMapper	0.22	0.68	0.99	< 0.001
18_jsecurity	IniResource	0.20	0.80	0.99	< 0.001
18_jsecurity	ResourceUtils	0.50	0.97	0.99	< 0.001
18_jsecurity	DefaultWebSecurityManager	0.04	0.43	1.00	< 0.001
21_geo-google	AddressToUsAddressFunctor	0.00	0.53	0.99	< 0.001
24_saxpath	XPathLexer	0.13	0.86	1.00	< 0.001
32_httpanalyzer	ScreenInputFilter	0.56	0.92	0.94	< 0.001
35_corina	TRML	0.01	0.43	0.99	< 0.001
35_corina	SiteListPanel	0.00	0.00	0.96	< 0.001
44_summa	FacetMapSinglePackedFactory	0.24	0.49	0.95	< 0.001
46_nutzenportfolio	KategorieDaoService	0.01	0.19	1.00	< 0.001
54_db-everywhere	Select	0.04	0.29	1.00	< 0.001
57_hft-bomberman	RoundTimeOverMsg	0.46	0.66	0.65	< 0.001
58_fps370	MouseMoveBehavior	0.12	0.66	0.99	< 0.001
58_fps370	Teder	0.12	0.33	0.80	< 0.001
6_jnfe	CST_COFINS	0.23	0.57	0.95	< 0.001
61_noen	DataType	0.00	1.00	1.00	< 0.001
61_noen	WatchDog	0.27	0.55	0.91	< 0.001
62_dom4j	STAXEventReader	0.00	0.01	1.00	< 0.001
62_dom4j	CloneHelper	0.11	0.38	0.99	< 0.001
62_dom4j	PerThreadSingleton	0.67	0.86	0.83	< 0.001
63_objectexplorer	LoggerFactory	0.64	0.80	0.60	< 0.001
66_openjms	SecurityConfigurationDescriptor	0.30	0.66	1.00	< 0.001
66_openjms	And	0.94	1.00	0.66	< 0.001
66_openjms	BetweenExpression	0.84	0.99	0.86	< 0.001
70_echodep	PackageDissemination	0.00	0.12	1.00	< 0.001
74_fixsuite	ListView	0.53	0.55	0.51	< 0.001
75_openhre	User	0.27	0.98	1.00	< 0.001
75_openhre	HL7SegmentMapImpl	0.99	1.00	0.52	< 0.001
78_caloriecount	BudgetWin	0.03	0.26	0.99	< 0.001
78_caloriecount	ArchiveScanner	0.02	0.68	1.00	< 0.001
78_caloriecount	RecordingEvent	0.70	0.97	1.00	< 0.001
78_caloriecount	BlockThread	0.28	0.62	0.91	< 0.001
79_twfbplayer	BattlefieldCell	0.04	0.41	0.92	< 0.001
79_twfbplayer	CriticalHit	0.81	0.99	0.60	< 0.001
80_wheelwebtool	Block	0.04	0.70	0.99	< 0.001
83_xbus	ByteArrayConverterAS400	0.00	0.06	0.97	< 0.001
84_ifx-framework	ChkOrdInqRs_Type	0.80	1.00	0.83	< 0.001
84_ifx-framework	PassbkItemInqRs_Type	0.92	0.99	0.62	< 0.001
85_shop	JSListSubstitution	0.97	0.98	0.47	0.019
88_jopenchart	InterpolationChartRenderer	0.00	0.02	0.97	< 0.001
89_jiggler	SignalCanvas	0.13	0.89	1.00	< 0.001
89_jiggler	ImageOutputStreamJAI	0.09	0.91	0.99	< 0.001
89_jiggler	LocalDifferentialGeometry	0.11	0.48	0.99	< 0.001
92_jcvi-javacommon	PhdFileDataStoreBuilder	0.29	0.80	0.99	< 0.001
93_quickserver	SimpleCommandSet	0.75	0.87	0.67	< 0.001
93_quickserver	AuthStatus	0.02	0.16	0.91	< 0.001
Average		0.62	0.78	0.69	

* There were 50 classes with no statistically significant difference

Table 3: For each class with statistically significant results, the table reports the average *Weak Mutation Score* obtained by the *OneGoal* approach and by the *Whole* approach.

Project	Class	OneGoal	Whole	$\hat{A}_{12}$	p-value
13_jdbacl	H2Util	0.78	0.93	0.67	< 0.001

Continued on next page

Table 3: Continued.

Project	Class	OneGoal	Whole	$\hat{A}_{12}$	p-value
13_jdbacl	AbstractTableMapper	0.13	0.67	0.99	< 0.001
18_jsecurity	Md2CredentialsMatcher	0.00	1.00	1.00	< 0.001
18_jsecurity	IniResource	0.12	0.70	0.99	< 0.001
18_jsecurity	ResourceUtils	0.23	0.84	1.00	< 0.001
18_jsecurity	DefaultWebSecurityManager	0.06	0.46	0.99	< 0.001
21_geo-google	AddressToUsAddressFunctor	0.00	0.63	1.00	< 0.001
21_geo-google	PremiseNumberSuffix	0.00	1.00	1.00	< 0.001
24_saxpath	XPathLexer	0.08	0.70	1.00	< 0.001
27_gangup	MapCell	0.00	1.00	1.00	< 0.001
32_httpanalyzer	ScreenInputFilter	0.50	0.86	0.92	< 0.001
35_corina	TRML	0.00	0.22	0.99	< 0.001
35_corina	SiteListPanel	0.00	0.00	0.94	< 0.001
43_lilith	EventIdentifier	0.70	1.00	1.00	< 0.001
43_lilith	LogDateRunnable	0.26	0.55	0.75	< 0.001
44_summa	FacetMapSinglePackedFactory	0.22	0.49	0.94	< 0.001
45_lotus	Phase	0.25	1.00	1.00	< 0.001
46_nutzenportfolio	KategorieDaoService	0.00	0.09	1.00	< 0.001
52_lagoon	Wildcard	0.66	0.99	1.00	< 0.001
54_db-everywhere	Select	0.00	0.04	1.00	< 0.001
57_hft-bomberman	RoundTimeOverMsg	0.00	1.00	1.00	< 0.001
58_fps370	MouseMoveBehavior	0.02	0.39	1.00	< 0.001
58_fps370	Teder	0.00	0.50	1.00	< 0.001
6_jnfe	CST_COFINS	0.55	0.88	0.95	< 0.001
61_noen	WatchDog	0.43	0.60	0.85	< 0.001
61_noen	MeasurementReport	0.00	1.00	1.00	< 0.001
61_noen	OperationResult	0.00	1.00	1.00	< 0.001
62_dom4j	STAXEventReader	0.00	0.00	1.00	< 0.001
62_dom4j	CloneHelper	0.10	0.64	0.99	< 0.001
62_dom4j	PerThreadSingleton	0.28	0.69	0.87	< 0.001
63_objectexplorer	LoggerFactory	0.00	1.00	1.00	< 0.001
66_openjms	SecurityConfigurationDescriptor	0.12	0.54	1.00	< 0.001
66_openjms	And	0.46	0.98	1.00	< 0.001
66_openjms	BetweenExpression	0.22	0.93	0.99	< 0.001
69_lhamacaw	CategoryStateEditor	0.00	0.04	0.89	< 0.001
70_echodep	PackageDissemination	0.00	0.13	1.00	< 0.001
74_fixsuite	ListView	0.09	0.09	0.54	< 0.001
75_openhre	User	0.17	0.97	1.00	< 0.001
75_openhre	HL7SegmentMapImpl	0.45	0.98	1.00	< 0.001
78_caloriecount	BudgetWin	0.03	0.15	0.94	< 0.001
78_caloriecount	ArchiveScanner	0.01	0.62	1.00	< 0.001
78_caloriecount	RecordingEvent	0.52	0.98	1.00	< 0.001
78_caloriecount	BlockThread	0.34	0.83	0.99	< 0.001
79_twfbplayer	BattlefieldCell	0.03	0.38	0.96	< 0.001
79_twfbplayer	CriticalHit	0.63	0.97	0.88	< 0.001
80_wheelwebtool	Block	0.00	0.84	0.99	< 0.001
80_wheelwebtool	JSONStringer	0.00	1.00	1.00	< 0.001
84_ifx-framework	ChkAcceptAddRs_Type	0.00	1.00	1.00	< 0.001
84_ifx-framework	ChkInfo_Type	0.00	1.00	1.00	< 0.001
84_ifx-framework	ChkOrdInqRs_Type	0.00	1.00	1.00	< 0.001
84_ifx-framework	CreditAdviseRs_Type	0.00	1.00	1.00	< 0.001
84_ifx-framework	DepAcctStmtInqRq_Type	0.00	1.00	1.00	< 0.001
84_ifx-framework	EMVCardAdviseRs_Type	0.00	1.00	1.00	< 0.001
84_ifx-framework	ForExDealMsgRec_Type	0.00	1.00	1.00	< 0.001
84_ifx-framework	PassbkItemInqRs_Type	0.00	1.00	1.00	< 0.001
84_ifx-framework	RecPmtCanRq_Type	0.00	1.00	1.00	< 0.001
84_ifx-framework	StdPayeeId_Type	0.00	1.00	1.00	< 0.001
84_ifx-framework	SvcAcctStatus_Type	0.00	1.00	1.00	< 0.001
84_ifx-framework	TINInfo_Type	0.00	1.00	1.00	< 0.001
85_shop	JSListSubstitution	0.22	0.88	0.99	< 0.001
86_at-robots2-j	RobotScoreKeeper	0.69	1.00	1.00	< 0.001
88_jopenchart	InterpolationChartRenderer	0.00	0.02	0.91	< 0.001
89_jiggler	SignalCanvas	0.37	0.92	0.99	< 0.001
89_jiggler	ImageOutputStreamJAI	0.07	0.82	0.99	< 0.001
89_jiggler	LocalDifferentialGeometry	0.03	0.32	0.99	< 0.001
9_falselight	falselight	0.00	1.00	1.00	< 0.001
92_jcvi-javacommon	PhdFileDataStoreBuilder	0.17	0.77	1.00	< 0.001
93_quickserver	SimpleCommandSet	0.00	0.91	1.00	< 0.001
93_quickserver	AuthStatus	0.00	0.16	1.00	< 0.001
Average		0.36	0.77	0.83	

* There were 29 classes with no statistically significant difference

test suites tend to be shorter using this secondary objective. However, optimizing the test suite's size can sometimes have undesired effects, e.g., less randomness and less diversity. For class *JSListSubstitution* in particular, the use of the test suite size as secondary objective seems to hamper the generation of a data structure that can satisfy the condition *if (s1!=null)*, as the fitness function in general provides no incentive to add more values into the vector data-structure. To verify this conjecture, we ran

Table 4: For each criterion, we report how often (number of goals) the *Whole* approach is better (higher effect size) than *OneGoal*, how often they are equivalent, and how often it is *OneGoal* that is better. We also report the number of comparisons that are statistically significant at 0.05 level, and when only one of the two techniques ever managed to cover a goal out of the 500 repeated experiments.

Criterion		# of Targets	Statistically at 0.05	Never Covered by the Other
Branch Coverage	Whole is better:	1410	1239	385
	Equivalent:	717		
	OneGoal is better:	255	3	0
	Total:	2382		
Line Coverage	Whole is better:	2292	1978	468
	Equivalent:	1457		
	OneGoal is better:	62	4	2
	Total:	3811		
Weak Mutation Testing	Whole is better:	9335	8634	3202
	Equivalent:	3137		
	OneGoal is better:	1	1	1
	Total:	12473		

EvoSuite 30 times for this configuration (class *JSListSubstitution*, mode *Whole* and criterion *Line Coverage*) with and without the secondary objective. The result of this comparison indeed indicates that not using the secondary objective leads to higher *WholeLine Coverage* on this class, with a 0.6 effect size and 0.04 p-value.

The largest increase in coverage is observed for the *Weak Mutation Testing* criterion. As shown in Table 3, *Whole* significantly outperforms *OneGoal* on 69 classes, leading to an average increase of 0.41 ($0.77 - 0.36$) with an effect size of 0.83. Recall that *OneGoal* has an inherent limitation related to the number of goals and the search budget distribution. The overall search budget, two minutes in our experiments, must be evenly distributed to target each coverage goal at a time. When the number of goals is high, as in our sample where 125 mutants are generated per class on average, the small time budget assigned to each goal is often not sufficient for a genetic algorithm to find any solution at all, i.e., zero coverage.

To study the difference between *OneGoal* and *Whole* at a finer grained level, Table 4 shows on how many coverage goals (i.e., respectively branches, lines and mutants) one approach is better than the other. These results indicate that there are only three goals for which *OneGoal* leads to better *Branch Coverage* results; however, all of them are covered by *Whole* at least once. A closer look reveals that all three goals are in the same class *JSONStringer*, which in fact is fully covered by both techniques in all their runs; but there are six datapoints for *Whole* missing, which is sufficient for the Fisher test to claim a significant difference. This may happen with experiments of this type, e.g., if there are problems on the cluster computer, or competing processes. This is the reason why we used as many as possible repetitions for our analysis. As this is a random effect, one would expect same odds regardless of the technique, which means that the number of cases where *Whole* is better because of crashes in *OneGoal* runs should be the same on average.

The results for *Line Coverage* and *Weak Mutation Testing* are similar. For *Line Coverage*, *OneGoal* is significantly better than *Whole* at covering four lines, but in this case *Whole* never manages to cover two of them. Also for *Weak Mutation Testing*, there is one mutant that is only covered by the *OneGoal* approach. These lines and the mutant are both in the class *BlockThread* (where all techniques resulted in 500 runs), which only has a single conditional branch, all other methods contain just sequences of statements (in EVOSUITE, a method without conditional statements is counted as a single branch, based on the control flow graph interpretation). However, the class spawns a new thread, and several of the methods synchronize on this thread (e.g., by calling *wait()* on the thread). EVOSUITE uses a timeout of five seconds for each test execution, and any test case or test suite that contains a timeout is assigned the maximum (worst) fitness value, and not considered as a valid solution in the final coverage analysis. In *BlockThread*, many tests lead to such timeouts, and a possible conjecture for the worse performance of the *Whole* approach may be that the chances of having an individual test case without timeout are simply higher than the chances of having an entire test *suite* without timeouts.

RQ1*: In our experiments, there are* 3 *out of* 2,382 *branches,* 4 *out of* 3,811 *lines and* 1 *out of* 12,473 *mutants for which OneGoal obtains better results than Whole. Of them,* 2 *lines and* 1 *mutant are never covered by Whole.*

4.3 RQ2: How many coverage goals found by *Whole* get missed by *OneGoal*?

Let us now quantify the goals on which *Whole* outperformed *OneGoal*. There are 11,851 goals (out of 18,666) in which *Whole* gives statistically better results: 1,239 branches, 1,978 lines and 8,634 mutants. For 4,055 of them (385 branches, 468 lines and 3,202 mutants), the *OneGoal* approach *never* managed to generate any results in any of the 500 runs. In other words, even if there are some (i.e., three) goals that only *OneGoal* can cover, there are many more goals that only *Whole* does cover.

RQ2*: Whole test suite generation is able to cover* 385 *branches,* 468 *lines and* 3,202 *mutants that OneGoal never covers.*

4.4 RQ3: Which factors influence the relative performance of *Whole* and *OneGoal*?

Having showed that the *Whole* approach leads to higher coverage, it is important to investigate the conditions in which this improvement is obtained. For each coverage criterion and for each goal, we calculated the odds ratio between *Whole* and *OneGoal* (i.e., we quantified what are the odds that *Whole* has higher chances to cover the goal compared to *OneGoal*). For each odds ratio, we studied its correlation with three different properties: (1) the $\hat{A}_{12}$ effect size between *Whole* and *OneGoal* on the class the goal belongs to; (2) the raw average coverage obtained by *OneGoal* on the class the goal belongs to; and, finally, (3) the size of the class, measured as number of

Table 5: For each criterion, correlation analyses between the odds ratios for each goal and three different properties. We used three different correlation measures: Pearson's r, Kendall's τ and Spearman's ρ. For each analysis, we report the obtained correlation value, its confidence interval at 0.05 level (only for Pearson's r) and the obtained p-value (of the test whether the correlation is different from zero).

Criterion	Property	r	Confidence Interval	p-value	τ	p-value	ρ	p-value
Branch Coverage	Whole vs. OneGoal	0.53	[0.50, 0.55]	≤ 0.001	0.50	≤ 0.001	0.62	≤ 0.001
	OneGoal coverage	-0.38	[-0.41, -0.34]	≤ 0.001	-0.02	0.117	-0.06	0.007
	# of branches	0.42	[0.39, 0.45]	≤ 0.001	0.34	≤ 0.001	0.46	≤ 0.001
Line Coverage	Whole vs. OneGoal	0.40	[0.37, 0.43]	≤ 0.001	0.35	≤ 0.001	0.44	≤ 0.001
	OneGoal coverage	-0.20	[-0.23, -0.17]	≤ 0.001	0.03	0.020	0.07	≤ 0.001
	# of lines	0.17	[0.14, 0.20]	≤ 0.001	0.16	≤ 0.001	0.23	≤ 0.001
Weak Mutation Testing	Whole vs. OneGoal	0.23	[0.21, 0.24]	≤ 0.001	0.46	≤ 0.001	0.56	≤ 0.001
	OneGoal coverage	0.29	[0.28, 0.31]	≤ 0.001	0.35	≤ 0.001	0.45	≤ 0.001
	# of mutants	-0.19	[-0.21, -0.18]	≤ 0.001	-0.11	≤ 0.001	-0.15	≤ 0.001

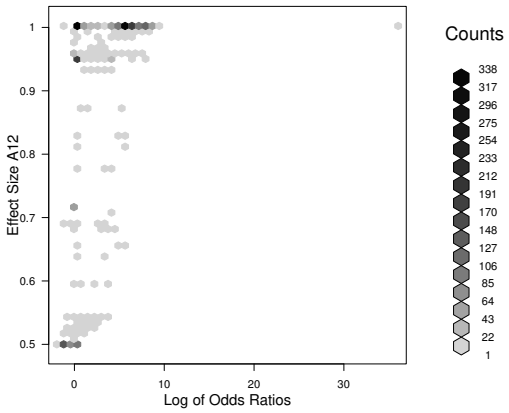
coverage targets (branches, lines, mutants) in it. Table 5 shows the results of these correlation analyses for the three coverage criteria.

We used Pearson's r correlation coefficient, as well as Kendall's τ and Spearman's ρ. The strengths of correlation are interpreted as follows: negligible ($.01$ to $.19$), weak ($.20$ to $.29$), moderate ($.30$ to $.39$), strong ($.40$ to $.69$), very strong ($.70$ to 1), and similarly on the negative range (-1 to -0.1) [19]. All three correlation coefficients are generally in agreement, except for the case of correlation with the *OneGoal* coverage for *Branch Coverage* and *Line Coverage*. There, it is weak/moderate for Pearson's r and negligible for the other two.

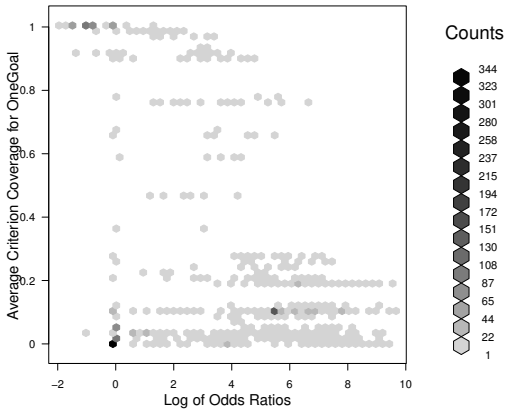
There is a positive correlation between the odds ratios and the $\hat{A}_{12}$ effect sizes for the three criteria. This is expected: on a class in which the *Whole* approach obtains higher coverage on average, it is also more likely that *Whole* will have higher coverage on each goal in isolation. This correlation is moderate for *Branch Coverage* and *Line Coverage*, at 53% and 40% respectively, and weak for *Weak Mutation Testing*, at only 23%.

On classes with many infeasible branches (or too difficult to cover for both *Whole* and *OneGoal*), one could expect higher results for *Whole* (as it is not negatively affected by infeasible branches [12]). It is not possible to determine for all branches if they are feasible or not. However, we can estimate the difficulty of a class by the obtained code coverage for the considered coverage criterion. Furthermore, one would expect better results of the *Whole* approach on larger, more complex classes. This is confirmed by the negative correlation of the odds ratios with the obtained average *OneGoal* coverage for *Branch Coverage* and *Line Coverage*. However, this is not the case for *Weak Mutation Testing*, which is the criterion with most targets. Similarly, the higher number of targets (so the class would likely be more difficult) leads to better performance for *Whole*. This is shown by a positive 42% correlation for *Branch Coverage* and 17% for *Line Coverage*. However, again we see the opposite trend for *Weak Mutation Testing*, i.e., a negative -19% correlation.

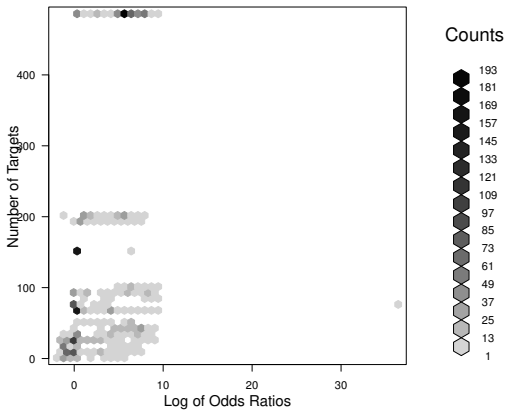
The analyses presented in Table 5 numerically quantify the correlations between the odds ratios and the different studied properties. To study them in more details, we present scatter plots for the $\hat{A}_{12}$ effect sizes, for the *OneGoal* average coverage



(a) $\hat{A}_{12}$ effect sizes.

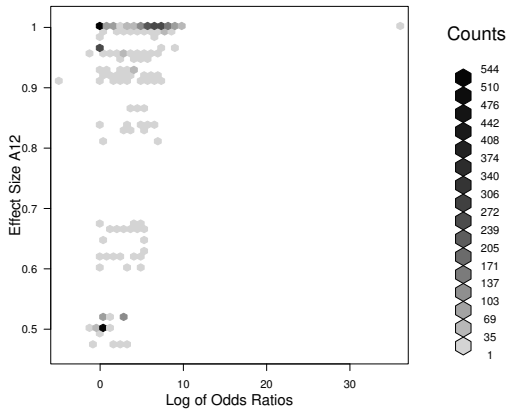


(b) Average class coverage obtained by *OneGoal*

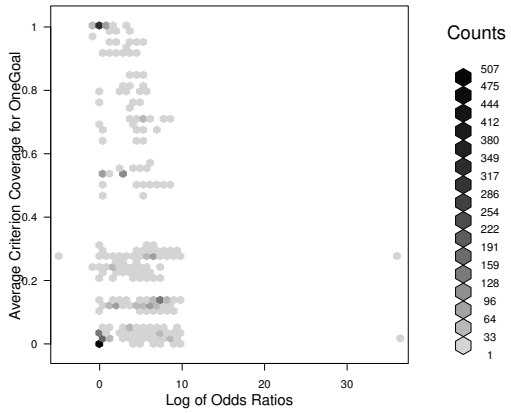


(c) Total number of branches

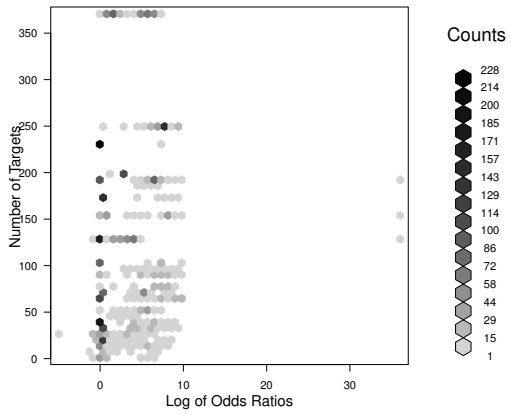
Fig. 2: For *Branch Coverage*, scatter plots of the (logarithm of) odds ratios compared to the $\hat{A}_{12}$ effect sizes (2a), average class coverage obtained by *OneGoal* (2b), and the total number of coverage goals (2c)



(a) $\hat{A}_{12}$ effect sizes

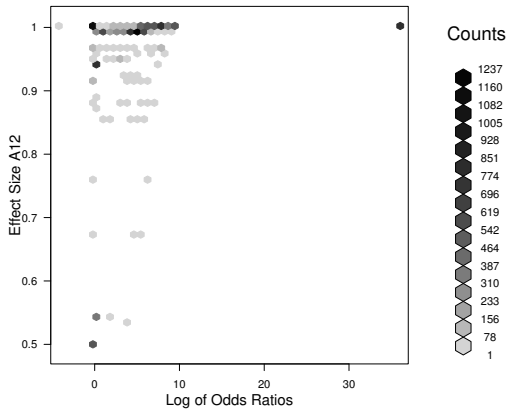


(b) Average class coverage obtained by *OneGoal*

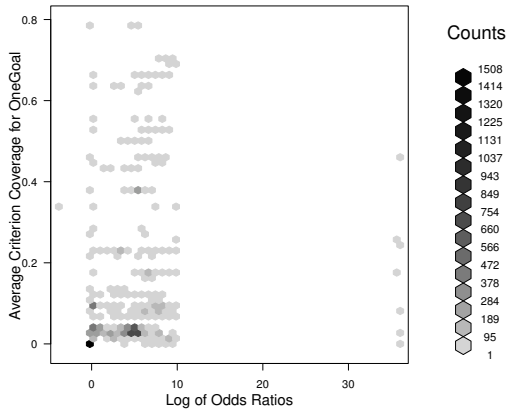


(c) Total number of lines

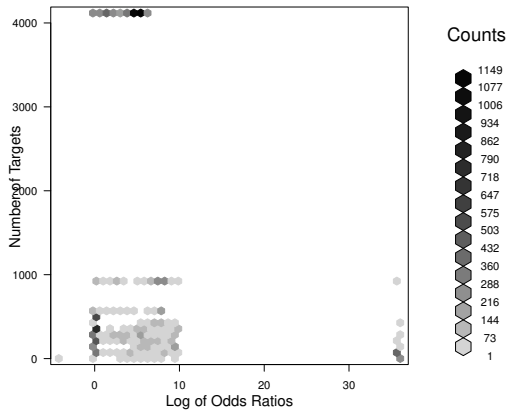
Fig. 3: For *Line Coverage*, scatter plots of the (logarithm of) odds ratios compared to the $\hat{A}_{12}$ effect sizes (3a), average class coverage obtained by *OneGoal* (3b), and the total number of coverage goals (3c)



(a) $\hat{A}_{12}$ effect sizes



(b) Average class coverage obtained by *OneGoal*



(c) Total number of mutants

Fig. 4: For *Weak Mutation Testing*, scatter plots of the (logarithm of) odds ratios compared to the $\hat{A}_{12}$ effect sizes (4a), average class coverage obtained by *OneGoal* (4b), and the total number of mutants (4c)

and for the number of target goals. Figure 2 presents said plots for *Branch Coverage*, Figure 3 for *Line Coverage* and Figure 4 for *Weak Mutation Testing*.

Figures 2a, 3a, 4a are in line with the positive correlation values shown in Table 5 for *Whole* vs. *OneGoal*. In these figures, there are clear major clusters at $\hat{A}_{12}$ close to 1 for logarithms of odds greater than 0. These are classes where *Whole* is very likely to lead to better results, although the number of coverage goals on which it improves are only small.

Regarding the comparisons with the difficulty of a class, Figure 2b for *Branch Coverage* and Figure 3b for *Line Coverage* show a similar trend. Most classes are located for low values of *OneGoal* coverage, with high odds ratios. For higher coverage, odds ratios decrease. However, in both cases, there is a consistent number of classes for which *OneGoal* achieves high coverage (clusters in the top-left borders in those figures). This is not the case for *Weak Mutation Testing*, as shown in Figure 4b, where, such a cluster does not appear. This explains the correlation values in Table 5. For *Weak Mutation Testing*, there is no simple class for which both *OneGoal* and *Whole* achieve very good results, and that would skew the correlations by creating that kind of cluster. Furthermore, for many classes, *OneGoal* achieves very low coverage (recall Table 4). By looking at the number of targets in Figure 4c, we can see there are many classes with high number of mutations. On these classes, we can hence infer that *OneBranch* achieves low coverage, regardless of whether those targets are difficult or not. This is due to how the search budget is split: trying to give same budget to all targets would result in very little budget per target if those are many, so low that even simple targets would not be covered. It would likely be more effective to use a higher budget per target, but addressing just a subset of them. In other words, the effectiveness of *OneGoal* on *Weak Mutation Testing* on complex classes is not a good indicator of how *Whole* will perform on those.

Still, there is a negative -19% correlation between the odds ratios and the number of targets for *Weak Mutation Testing* (recall Table 5). In Figure 4c, there are three main clusters: (a) odds ratios between 0 and 10 for less than 1000 mutations, (b) similar odds ratios but for approximately 4000 mutations, and (c) very high odds ratios (above 30) for low number of targets. The odds ratio in cluster (b) are slightly smaller than in (a) and, considering (c), that would lead to the negative correlation. Although the number of mutants in these classes does not seem particularly high, it seems that nevertheless the overall search budget per mutant is too low in the *OneGoal* approach. For example, even a lower number of mutants can be problematic if the time EvoSuite requires to generate the tests is high, or if the execution time of the tests is high. For example, class *wheel.components.Block* has only 46 mutants, yet EvoSuite struggles to create dependency classes: When creating dependency objects, EvoSuite calls methods/constructors to create these objects, and then recursively creates objects for parameters of these calls. In the case of *wheel.components.Block* the depth of this recursion frequently hits the maximum (which is 10 by default), in which case EvoSuite aborts the recursion and starts over with creating the same object. As a result, in the time given per mutant when targeting individual mutants, EvoSuite barely manages to generate sufficient tests to even fill the initial population, whereas *Whole* can spend more time on generating the initial population.

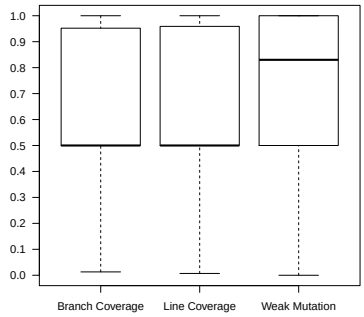


Fig. 5: For each criterion, boxplot of effect size in the comparison between *OneGoal* and *Whole*.

Table 6: For each criterion, we report how often (number of goals) the *Archive* approach is better (higher effect size) than *Whole*, how often they are equivalent, and how often it is *Whole* that is better. We also report the number of comparisons that are statistically significant at 0.05 level, and when only one of the two techniques ever managed to cover a goal out of the 500 repeated experiments for the random sample of 100 classes.

Criterion		# of Targets	Statistically at 0.05	Never Covered by the Other
Branch Coverage	Archive is better:	530	370	20
	Equivalent:	703		
	Whole is better:	1149	254	42
	Total:	2382		
Line Coverage	Archive is better:	986	444	12
	Equivalent:	1984		
	Whole is better:	841	263	124
	Total:	3811		
Weak Mutation Testing	Archive is better:	5479	4076	310
	Equivalent:	5139		
	Whole is better:	1855	271	126
	Total:	12473		

RQ3: *In general, the more complex a class, the better results will Whole achieve compared to OneBranch.*

To summarize in a graphical way the results obtained in the experiments conducted to address **RQ1-3**, Figure 5 presents the boxplots of the effect sizes in the comparison between *OneGoal* and *Whole* for *Branch Coverage*, *Line Coverage* and *Weak Mutation Testing*, which show that *Whole* overall performed better than *OneGoal* for the three criteria.

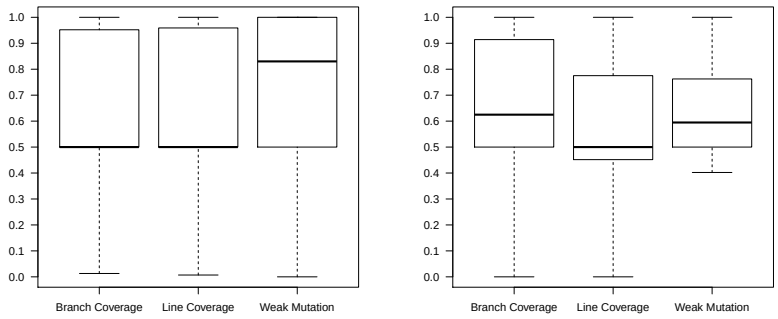
Table 7: For each criterion, we report how often (number of goals) the *Archive* approach is better (higher effect size) than *Whole*, how often they are equivalent, and how often it is *Whole* that is better. We also report the number of comparisons that are statistically significant at 0.05 level, and when only one of the two techniques ever managed to cover a goal out of the 30 repeated experiments for the selection of 100 classes with highest number of branches (one per SF100 project).

Criterion		# of Targets	Statistically at 0.05	Never Covered by the Other
Branch Coverage	Archive is better:	6288	2039	711
	Equivalent:	23566		
	Whole is better:	4196	1726	2588
	Total:	34050		
Line Coverage	Archive is better:	8466	5109	1391
	Equivalent:	32651		
	Whole is better:	3934	930	1376
	Total:	45051		
Weak Mutation Testing	Archive is better:	57621	20358	8327
	Equivalent:	106946		
	Whole is better:	14270	167	1896
	Total:	178837		

4.5 RQ4: How does using an archiving solution, *Archive*, influence the performance of *Whole*?

In order to answer **RQ4**, we now compare the *Archive* and the *Whole* test suite generation approaches; Table 6 shows the results of this comparison. There are more cases where using *Archive* is significantly better than *Whole*, demonstrating the beneficial effects of the archive. However, there are also many cases where using *Archive* achieves worse results compared to *Whole*. The fact that this is less often the case for *Weak Mutation Testing* suggests that the benefit achieved by *Archive* is larger for more complex classes. To verify this conjecture, we replicated our experiments comparing *Archive* and *Whole* for a selection of classes with higher complexity than the random sample used so far. The results of these analyses are shown in Table 7, and here the number of cases where *Archive* is significantly better is much higher in general. This is also confirmed by Figure 6, which summarizes in boxplots the effect sizes between *Archive* and *Whole*. Clearly, *Archive* is generally better, and for larger classes the benefit is larger. More per class details are in the Appendix, in Tables 9-14.

The observation that there are cases where using an archive leads to a negative effect suggests that the search operators need to be further optimized in order to accommodate for the archive. In particular, when mutating test suites, EvoSuite either changes existing tests, or adds new tests. New tests are generated randomly, and over time the search would be expected to focus on more difficult coverage goals when using an archive. However, random tests are less likely to cover these goals. The minimization used as a secondary objective would thus gradually remove these additional tests, and the search may prematurely converge on sub-optimal individuals. Using specialized search operators helps in alleviating this problem by sampling tests



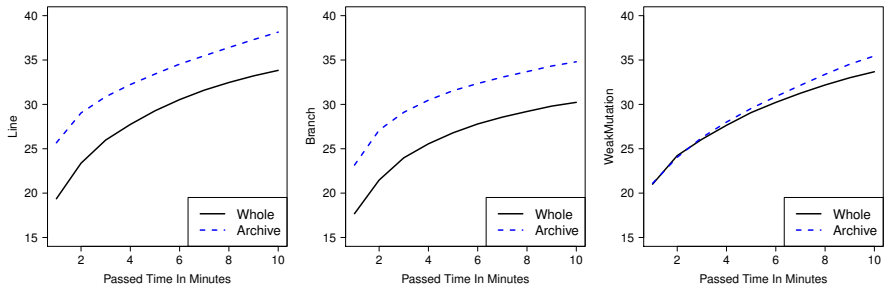
(a) Random sample of 100 classes (b) 100 classes with highest number of branches

Fig. 6: Boxplots of effect sizes in the comparisons between *Whole* and *Archive* for the two selection of classes (random sample of 100 classes, and selection of 100 clases, one per SF100 project, with highest number of branches).

from the archive and mutating them instead of constantly generating random tests. Other optimisations, like for instance applying seeding strategies, could help increase the benefits of using the archive.

Extended Search Budget. Intuitively, it is conceivable that the improvements observed for *Archive* compared to *Whole* might be less apparent with an increased search budget. The *Archive* approach builds up an artificial test suite by collecting test cases across evolving test suites. Given a more generous amount of time, could *Whole* converge towards a best individual with similar coverage to the one produced by *Archive*? To address this notion, we ran an experiment on a selection of the 10 classes with the highest number of branches in SF100 (ten largest classes in Table 8 in the Appendix). For this experiment, we used an extended search budget of 10 minutes and 50 repetitions. Figure 7 compares the performance of *Archive* and *Whole* on *Branch Coverage*, *Line Coverage* and *Weak Mutation Testing* over time. The plots show that the improvement of performance by the archive persist over time for *Branch Coverage* and *Line Coverage* and increases in the case of *Weak Mutation Testing*. It is important to notice, however, that coverage on these large classes is not saturated even with a 10 minutes budget. A plausible conjecture is that *Whole* would have the opportunity to catch up once *Archive* achieves full coverage; before that, though, a steady advantage can be expected with the *Archive* approach.

Figure 7 shows the manifest improvements in coverage achieved by the *Archive* approach, but where does this overall increase in performance stem from? While the data we collected does not comprise specific timing information about the phases of the search, we take the number of generations and fitness evaluations as proxy metrics. In this experiment with extended search budget we observed that on average the *Archive* approach evolved more generations (4,186 vs. 3,020) and performed more fitness evaluations (209,370 vs. 151,005) than the *Whole* approach. Our interpretation of these results is that in the *Archive* approach, individuals tend to be smaller



(a) Line (b) Branch (c) Weak Mutation

Fig. 7: Comparison of the performance of *Whole* and *Archive* for the 10 classes with the highest number of branches overall (at most one per SF100 project), using an increased search budget of 10 minutes.

and their fitness evaluation cheaper, allowing for a more exhaustive search than in the *Whole* approach.

RQ4: *Archive leads to better results, which persist over time, particularly on larger classes, but it may have negative effects on smaller classes without specialized search operators.*

5 Threats to Validity

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our testing framework, it has been carefully tested. But it is well known that testing alone cannot prove the absence of defects. Furthermore, randomized algorithms are affected by chance. To cope with this problem, we ran each experiment 500 times, and we followed rigorous statistical procedures to evaluate their results. To enable fair comparisons and to avoid possible confounding factors when different tools are used, the three approaches, *OneGoal*, *Whole* and *Archive*, were implemented in the same tool (i.e., EvoSuite). Furthermore, the same default values were used for all relevant parameters of the tool, e.g., population size, mutation rates and test length.

When addressing the difficulty of covering certain goals in our experiments, we rely on the underlying assumption that a high number of branches in a class implies that it contains more difficult coverage goals. Often in practice, the larger a class, the more complex behaviour it represents, and the more complex behaviour it represents, the more difficult it gets for an automated technique to to recreate specific states and configurations to cover certain branches. Nevertheless, since this assumption may not always hold, it represents a threat to the *construct validity* of our study.

There is the threat to *external validity* regarding the generalization to other types of software, which is common for any empirical analysis. Because of the large number of experiments required (in the order of hundreds of days of computational resources), we only used 100 classes for our in depth evaluations. These classes were randomly chosen from the SF100 corpus, which is a random selection of 100 projects

from SourceForge. We only experimented for Java software using branch, line and mutation coverage. Whether our results do generalize to other programming languages and testing criteria is a matter of future research.

6 Conclusions

Existing research has shown that the whole test suite approach can lead to higher code coverage [12, 13]. However, there was a reasonable doubt on whether it would still perform better on particularly difficult coverage goals when compared to a more focused approach.

To shed light on this potential issue, in this paper we performed an in-depth analysis to study if such cases do indeed occur in practice. Based on a random selection of 100 Java classes in which we aim at automating test generation for different testing criteria (branch, line and mutation coverage) with the EvoSuite tool, we found out that there are indeed coverage goals for which the whole test suite approach leads to worse results. However, these cases are very few compared to the cases in which better results are obtained (nearly two orders of magnitude in difference), and all coverage goals that were covered by *OneGoal* but not covered by *Whole* at all turned out to be special cases, rather than general deficiencies of the approach.

Our experiments also showed that the use of an archive does lead to better results on average, but it may have some negative side-effects on some testing targets, as the use of an archive would require specialized search operators that use the archive; designing these search operators will require further research. Furthermore, the incorporation of the archive as part of the *Whole* approach raises the question of whether it can still be regarded as evolution of "test suites". Whereas from the practical point of view we have demonstrated the usefulness of this optimization, there might be theoretical implications related to constructing the resulting test suite incrementally from multiple test suites instead of just producing the fittest individual in the final population. Since these aspects escape the scope of this paper, we plan to investigate them in future work.

The results presented in this paper provide more support to the validity and usefulness of the whole test suite approach in the context of test data generation. Whether such an approach could be successfully adapted also to other search-based software engineering problems and whether the whole approach is more suitable than the traditional per-goal approach for industrial testing practitioners will be a matter of future research.

To learn more about EvoSuite, visit our website at:

`http://www.evosuite.org`

Acknowledgements

This project has been funded by the EPSRC project "EXOGEN" (EP/K030353/1), a Google Focused Research Award on "Test Amplification", and by the National Research Fund, Luxembourg (FNR/P10/03).

References

1. Ali, S., Briand, L., Hemmati, H., Panesar-Walawege, R.: A systematic review of the application and empirical investigation of search-based test-case generation. IEEE Transactions on Software Engineering (TSE) **36**(6), 742–762 (2010)
2. Arcuri, A.: A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage. IEEE Transactions on Software Engineering (TSE) **38**(3), 497–519 (2012)
3. Arcuri, A.: It really does matter how you normalize the branch distance in search-based software testing. Software Testing, Verification and Reliability (STVR) **23**(2), 119–147 (2013)
4. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability (STVR) **24**(3), 219–250 (2014)
5. Arcuri, A., Fraser, G.: On the effectiveness of whole test suite generation. In: International Symposium on Search Based Software Engineering (SSBSE), pp. 1–15. Springer International Publishing (2014)
6. Arcuri, A., Iqbal, M.Z., Briand, L.: Random testing: Theoretical results and practical implications. IEEE Transactions on Software Engineering (TSE) **38**(2), 258–277 (2012)
7. Arcuri, A., Yao, X.: Search based software testing of object-oriented containers. Inform. Sciences **178**(15), 3075–3095 (2008)
8. Baresi, L., Lanzi, P.L., Miraz, M.: Testful: an evolutionary test approach for java. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 185–194 (2010)
9. Fraser, G., Arcuri, A.: EvoSuite: Automatic test suite generation for object-oriented software. In: ACM Symposium on the Foundations of Software Engineering (FSE), pp. 416–419 (2011)
10. Fraser, G., Arcuri, A.: Sound empirical evidence in software testing. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 178–188 (2012)
11. Fraser, G., Arcuri, A.: Handling test length bloat. Software Testing, Verification and Reliability (STVR) **23**(7), 553–582 (2013)
12. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering (TSE) **39**(2), 276–291 (2013)
13. Fraser, G., Arcuri, A.: Achieving scalable mutation-based generation of whole test suites. Empirical Software Engineering (EMSE) **20**(3), 783–812 (2015)
14. Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. IEEE Transactions on Software Engineering (TSE) **28**(2), 278–292 (2012)
15. Harman, M., Kim, S.G., Lakhotia, K., McMinn, P., Yoo., S.: Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In: International Workshop on Search-Based Software Testing (SBST) (2010)
16. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys (CSUR) **45**(1), 11 (2012)
17. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. Technical Report TR-09-06, CREST Centre, King's College London, London, UK (2009)
18. Korel, B.: Automated software test data generation. IEEE Transactions on Software Engineering (TSE) **16**(8), 870–879 (1990)
19. Kotrlik, J.W., Williams, H.A.: The incorporation of effect size in information technology, learning, and performance research. Information Technology, Learning, and Performance **21**(1), 1–7 (2003)
20. Lakhotia, K., McMinn, P., Harman, M.: An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. J. Syst. Softw. **83**(12) (2010)
21. Li, N., Meng, X., Offutt, J., Deng, L.: Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report). In: IEEE International Symposium on Software Reliability Engineering (ISSRE), pp. 380–389 (2013)
22. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability (STVR) **14**(2), 105–156 (2004)
23. Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. IEEE Transactions on Software Engineering (TSE) **2**(3), 223–226 (1976)
24. Panichella, A., Kifetew, F.M., Tonella, P.: Reformulating branch coverage as a many-objective optimization problem. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 1–10 (2015)
25. Ribeiro, J.C.B.: Search-based test case generation for object-oriented Java software using strongly-typed genetic programming. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 1819–1822. ACM (2008)

26. Tonella, P.: Evolutionary testing of classes. In: ACM Int. Symposium on Software Testing and Analysis (ISSTA), pp. 119–128 (2004)
27. Wappler, S., Lammermann, F.: Using evolutionary algorithms for the unit testing of object-oriented software. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 1053–1060. ACM (2005)
28. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Information and Software Technology **43**(14), 841–854 (2001)

Additional Supporting Material

Table 8: For each project in the SF100 corpus, selection of the class with the highest number of branch goals.

Project	Class	Public Methods	LOC	Branch Goals
1_tullibee	EClientSocket	39	55	350
2_a4j	ProductDetails	102	101	130
3_gaj	GAAlgorithm	11	13	14
4_rif	RIFInvoker	3	5	47
5_templateit	Poi2ItextUtil	11	11	78
6_jnfe	TransportKeyStoreBean	12	10	28
7_sfmis	Loader	20	19	82
8_gfarcegestionfa	ModifTableStockage	10	9	123
9_falselight	Services	6	9	9
10_water-simulator	SuiteGUI	3	8	47
11_imsmart	MContentManagerFileNet	6	6	26
12_dsachat	Challenge	9	14	136
13_jdbacl	SQLParser	78	152	2195
14_omjstate	Transition	11	12	30
15_beanbin	LuceneIndexManager	8	23	91
16_templatedetails	JoomlaOutput	9	87	18
17_inspirento	MonthlyCalendar	45	50	150
18_jsecurity	AuthorizingRealm	29	44	193
19_jmca	JavaParser	122	477	7910
20_nekomud	Connection	5	6	19
21_geo-google	ObjectFactory	87	86	86
22_byuic	TokenStream	1	34	1030
23_jwbf	SimpleArticle	28	28	72
24_saxpath	XPathLexer	4	44	484
25_jni-inchi	INCHI_KEY	4	5	47
26_jipa	Main	14	16	129
27_gangup	AbstractMap	8	9	166
28_greencow	Main	3	1	1
29_apbsmem	Main	10	23	275
30_bpmail	EmailFacadeImpl	20	25	78
31_xisemele	WriterEditorImpl	23	25	48
32_httpanalyzer	HttpAnalyzerView	5	66	84
33_javaviewcontrol	JVCParserTokenManager	8	63	2380
34_sbmlreader2	SBMLGraphReader	6	6	38
35_corina	GrapherPanel	42	50	290
36_schemaspy	Config	115	124	408
37_petsoar	Pet	20	20	28
38_javabullboard	PropertyUtils	31	31	366
39_diffi	StringIncrementor	6	4	35
40_glengineer	Scheme	15	46	241
41_follow	FollowAppAttributes	55	68	116
42_asphodel	DefaultRepositoryManager	12	14	42
43_lilith	MainFrame	68	158	411
44_summa	DatabaseStorage	21	91	533
45_lotus	Phase	4	2	28
46_nutzenportfolio	AuswahlfeldDaoService	34	38	218
47_dvd-homevideo	GUI	14	89	184
48_resources4j	AbstractResources	55	55	176
49_diebierse	Drink	46	45	81
50_biff	Scanner	6	38	817
51_jiprof	MethodWriter	25	43	824
52_lagoon	XMLSerializer	30	41	287
53_shp2kml	GeomConverter	12	10	27
54_db-everywhere	MysqlTableStructure	16	14	161
55_lavalamp	DeviceProperties	18	17	22
56_jhandballmoves	HandballModel	66	78	317
57_hft-bomberman	ServerGameModel	6	9	128

Continued on next page

Table 8: Continued.

Project	Class	Public Methods	LOC	Branch Goals
58_fps370	Fps370Panel	18	23	151
59_mygrid	Job	24	24	108
60_sugar	SCLLexer	21	24	207
61_noen	EFSMGenerator	26	35	142
62_dom4j	XMLWriter	57	93	426
63_objectexplorer	ExplorerFrameEventConverter	20	42	175
64_jtailgui	JTailLogger	32	34	125
65_gsftp	RemoteFileBrowser	14	18	167
66_openjms	URI	30	44	470
67_gae-app-manager	QuotaDetailsParser	3	6	47
68_biblestudy	ServletConnection	34	35	60
69_lhamacaw	SQLVariableManager	39	58	179
70_echodep	HaSMETSValidator	15	18	792
71_ext4j	Functions	28	28	154
72_battlecry	bcGenerator	4	19	281
73_fim1	ModernChatServer	53	60	369
74_fixsuite	TreeView	6	18	62
75_openhre	LdapService	15	19	132
76_dash-framework	Main	3	5	7
77_io-project	ClientGroup	8	12	66
78_caloriecount	WindowHelper	325	337	387
79_twfbplayer	BattleStatistics	37	44	156
80_wheelwebtool	MethodWriter	25	44	838
81_javathena	UserManagement	60	66	328
82_ipcalculator	IPv4	3	62	133
83_xbus	RecordTypeDescriptionChecker	11	15	281
84_ifx-framework	BankSvcRq_Type	304	304	304
85_shop	JSTerm	19	22	192
86_at-robots2-j	AtRobotLineLexer	6	14	119
87_jaw-br	JanelaPrincipal	3	70	62
88_jopenchart	CoordSystemUtilities	14	13	92
89_jiggler	ImageOps	3	7	337
90_dcparseargs	ArgsParser	10	9	80
91_classviewer	ClassInfo	19	27	153
92_jcvi-javacommon	Nucleotide	14	15	240
93_quickserver	QuickServer	146	167	725
94_jclo	JCLO	27	37	129
95_celwars2009	Entity	18	20	287
96_heal	MetadataDAO	44	46	392
97_feudalismgame	Battle	9	8	788
98_trans-locator	FoxHuntFrame	3	16	26
99_newzgrabber	Downloader	19	20	268
100_jgaap	jgaapGUI	3	4	23
Total		3049	4609	32794

Table 9: For each class with statistically significant results, the table reports the average *Branch Coverage* obtained by the *Whole* approach and by the *Archive* approach.

Project	Class	Whole	Archive	$\hat{A}_{12}$	p-value
13_jdbacl	AbstractTableMapper	0.71	0.68	0.40	< 0.001
18_jsecurity	IniResource	0.73	0.79	0.75	< 0.001
18_jsecurity	DefaultWebSecurityManager	0.42	0.46	0.66	< 0.001
24_saxpath	XPathLexer	0.70	0.86	1.00	< 0.001
32_httpanalyzer	ScreenInputFilter	0.83	0.87	0.60	< 0.001
35_corina	TRML	0.20	0.22	0.72	< 0.001
44_summa	FacetMapSinglePackedFactory	0.46	0.49	0.59	< 0.001
58_fps370	MouseMoveBehavior	0.53	0.56	0.65	< 0.001
61_noen	WatchDog	0.54	0.57	0.56	< 0.001
66_openjms	BetweenExpression	0.87	0.94	0.69	< 0.001
78_caloriecount	ArchiveScanner	0.63	0.58	0.33	< 0.001
78_caloriecount	BlockThread	0.82	1.00	0.95	< 0.001
79_twfbplayer	BattlefieldCell	0.36	0.42	0.58	< 0.001
80_wheelwebtool	Block	0.81	0.68	0.37	< 0.001
85_shop	JSListSubstitution	0.97	0.98	0.53	< 0.001
89_jiggler	SignalCanvas	0.97	0.99	0.71	< 0.001
89_jiggler	ImageOutputStreamJAI	0.79	0.63	0.31	< 0.001
89_jiggler	LocalDifferentialGeometry	0.32	0.45	0.82	< 0.001
92_jcvi-javacommon	PhdFileDataStoreBuilder	0.80	0.84	0.88	< 0.001
Average		0.78	0.78	0.52	

* There were 79 classes with no statistically significant difference

Table 10: For each class with statistically significant results, the table reports the average *Line Coverage* obtained by the *Whole* approach and by the *Archive* approach.

Project	Class	Whole	Archive	$\hat{A}_{12}$	p-value
13_jdbacl	AbstractTableMapper	0.68	0.65	0.30	< 0.001
18_jsecurity	IniResource	0.80	0.74	0.21	< 0.001
18_jsecurity	ResourceUtils	0.97	0.96	0.25	< 0.001
18_jsecurity	DefaultWebSecurityManager	0.43	0.48	0.67	< 0.001
21_geo-google	AddressToUsAddressFunctor	0.53	0.19	0.17	< 0.001
24_saxpath	XPathLexer	0.86	0.92	0.99	< 0.001
35_corina	TRML	0.43	0.38	0.23	< 0.001
44_summa	FacetMapSinglePackedFactory	0.49	0.45	0.19	< 0.001
61_noen	WatchDog	0.55	0.56	0.55	< 0.001
62_dom4j	STAXEventReader	0.01	0.01	0.50	< 0.001
66_openjms	SecurityConfigurationDescriptor	0.66	0.66	0.48	< 0.001
66_openjms	BetweenExpression	0.99	0.99	0.51	0.009
75_openhre	User	0.98	0.97	0.34	< 0.001
78_caloriecount	ArchiveScanner	0.68	0.57	0.10	< 0.001
78_caloriecount	BlockThread	0.62	0.80	1.00	< 0.001
79_twfbplayer	BattlefieldCell	0.41	0.45	0.52	0.018
80_wheelwebtool	Block	0.70	0.43	0.08	< 0.001
85_shop	JSListSubstitution	0.98	0.93	0.26	< 0.001
89_jiggler	SignalCanvas	0.89	0.92	0.75	< 0.001
89_jiggler	ImageOutputStreamJAI	0.91	0.73	0.11	< 0.001
89_jiggler	LocalDifferentialGeometry	0.48	0.52	0.64	< 0.001
92_jcvi-javacommon	PhdFileDataStoreBuilder	0.80	0.76	0.02	< 0.001
93_quickserver	SimpleCommandSet	0.87	0.89	0.58	< 0.001
Average		0.78	0.77	0.47	

* There were 75 classes with no statistically significant difference

Table 11: For each class with statistically significant results, the table reports the average *Weak Mutation Score* obtained by the *Whole* approach and by the *Archive* approach.

Project	Class	Whole	Archive	$\hat{A}_{12}$	p-value
13_jdbacl	H2Util	0.93	0.86	0.00	< 0.001
13_jdbacl	AbstractTableMapper	0.67	0.65	0.22	< 0.001
18_jsecurity	IniResource	0.70	0.69	0.40	< 0.001
18_jsecurity	ResourceUtils	0.84	0.89	0.98	< 0.001
18_jsecurity	DefaultWebSecurityManager	0.46	0.54	0.77	< 0.001
21_geo-google	AddressToUsAddressFunctor	0.63	0.89	0.96	< 0.001
24_saxpath	XPathLexer	0.70	0.81	0.99	< 0.001

Continued on next page

Table 11: Continued.

Project	Class	Whole	Archive	$\hat{A}_{12}$	p-value
32_httpanalyzer	ScreenInputFilter	0.86	0.90	0.84	< 0.001
35_corina	TRML	0.22	0.21	0.72	< 0.001
35_corina	SiteListPanel	0.00	0.00	0.50	< 0.001
43_lilith	LogDateRunnable	0.55	0.50	0.50	< 0.001
44_summa	FacetMapSinglePackedFactory	0.49	0.56	0.80	< 0.001
46_nutzenportfolio	KategorieDaoService	0.09	0.02	0.00	< 0.001
52_lagoon	Wildcard	0.99	0.98	0.00	< 0.001
54_db-everywhere	Select	0.04	0.00	0.00	< 0.001
58_fps370	MouseMoveBehavior	0.39	0.25	0.00	< 0.001
58_fps370	Teder	0.50	0.00	0.00	< 0.001
6_jnfe	CST_COFINS	0.88	0.77	0.00	< 0.001
61_noen	WatchDog	0.60	0.66	0.76	< 0.001
62_dom4j	STAXEventReader	0.00	0.00	0.00	< 0.001
62_dom4j	CloneHelper	0.64	0.28	0.00	< 0.001
62_dom4j	PerThreadSingleton	0.69	0.52	0.00	< 0.001
66_openjms	SecurityConfigurationDescriptor	0.54	0.40	0.00	< 0.001
66_openjms	And	0.98	0.97	0.00	< 0.001
66_openjms	BetweenExpression	0.93	0.99	0.87	< 0.001
69_lhamacaw	CategoryStateEditor	0.04	0.00	0.00	< 0.001
70_echodep	PackageDissemination	0.13	0.18	1.00	< 0.001
74_fixsuite	ListView	0.09	0.09	0.00	< 0.001
75_openhre	User	0.97	0.96	0.37	< 0.001
75_openhre	HL7SegmentMapImpl	0.98	0.96	0.00	< 0.001
78_caloriecount	BudgetWin	0.15	0.18	1.00	< 0.001
78_caloriecount	ArchiveScanner	0.62	0.66	0.58	< 0.001
78_caloriecount	RecordingEvent	0.98	0.96	0.03	< 0.001
78_caloriecount	BlockThread	0.83	0.93	0.95	< 0.001
79_twfbplayer	BattlefieldCell	0.38	0.31	0.13	< 0.001
79_twfbplayer	CriticalHit	0.97	0.93	0.05	< 0.001
80_wheelwebtool	Block	0.84	0.80	0.24	< 0.001
88_jopenchart	InterpolationChartRenderer	0.02	0.00	0.00	< 0.001
89_jiggler	SignalCanvas	0.92	0.90	0.18	< 0.001
89_jiggler	ImageOutputStreamJAI	0.82	0.80	0.65	< 0.001
89_jiggler	LocalDifferentialGeometry	0.32	0.64	0.97	< 0.001
92_jcvi-javacommon	PhdFileDataStoreBuilder	0.77	0.74	0.35	< 0.001
93_quickserver	SimpleCommandSet	0.91	1.00	1.00	< 0.001
93_quickserver	AuthStatus	0.16	0.00	0.00	< 0.001
Average		0.77	0.77	0.43	

* There were 54 classes with no statistically significant difference

Table 12: For each *top* class, the table reports the average *Branch Coverage* obtained by the *Whole* approach and by the *Archive* approach.

Project	Class	Whole	Archive	$\hat{A}_{12}$	p-value
1_tullibee	EClientSocket	0.16	0.17	0.95	< 0.001
100_jgaap	jgaapGUI	0.54	0.59	0.71	0.003
11_imsmart	MContentManagerFileNet	0.21	0.22	0.62	0.016
13_jdbacl	SQLParser	0.25	0.32	0.93	< 0.001
15_beanbin	LuceneIndexManager	0.28	0.30	0.84	< 0.001
16_templatedetails	JoomlaOutput	0.90	0.95	0.94	< 0.001
17_inspirento	MonthlyCalendar	0.62	0.71	0.91	< 0.001
18_jsecurity	AuthorizingRealm	0.72	0.75	0.68	0.014
19_jmca	JavaParser	0.12	0.00	0.00	< 0.001
2_a4j	ProductDetails	0.82	0.83	0.83	< 0.001
22_byuic	TokenStream	0.45	0.52	0.96	< 0.001
23_jwbf	SimpleArticle	0.98	0.96	0.32	0.014
24_saxpath	XPathLexer	0.71	0.86	1.00	< 0.001
25_jni-inchi	INCHI_KEY	0.98	1.00	0.72	< 0.001
26_jipa	Main	0.34	0.47	0.99	< 0.001
29_apbsmem	Main	0.00	0.00	0.79	< 0.001
33_javaviewcontrol	JVCParserTokenManager	0.14	0.20	0.71	0.003
36_schemaspy	Config	0.77	0.85	0.98	< 0.001
38_javabullboard	PropertyUtils	0.65	0.84	1.00	< 0.001
40_glengineer	Scheme	0.58	0.64	0.92	< 0.001
43_lilith	MainFrame	0.00	0.00	0.53	< 0.001
44_summa	DatabaseStorage	0.01	0.01	0.59	< 0.001
46_nutzenportfolio	AuswahlfeldDaoService	0.13	0.13	0.75	< 0.001
49_diebierse	Drink	0.91	1.00	1.00	0.003
50_biff	Scanner	0.15	0.16	0.97	< 0.001
51_jiprof	MethodWriter	0.26	0.34	0.91	< 0.001
53_shp2kml	GeomConverter	0.97	1.00	0.67	< 0.001
54_db-everywhere	MysqlTableStructure	0.35	0.45	0.83	< 0.001
55_lavalamp	DeviceProperties	0.99	0.97	0.34	0.003
56_jhandballmoves	HandballModel	0.62	0.73	0.98	< 0.001
57_hft-bomberman	ServerGameModel	0.09	0.10	0.66	0.018

Continued on next page

Table 12: Continued.

Project	Class	Whole	Archive	$\hat{A}_{12}$	p-value
59_mygrid	Job	0.94	0.94	0.68	0.005
60_sugar	SCLLexer	0.43	0.51	0.82	< 0.001
61_noen	EFSMGenerator	0.35	0.38	0.84	< 0.001
62_dom4j	XMLWriter	0.51	0.61	0.98	< 0.001
64_jtailgui	JTailLogger	0.14	0.16	0.50	< 0.001
66_openjms	URI	0.74	0.83	0.99	< 0.001
69_lhamacaw	SQLVariableManager	0.07	0.09	1.00	< 0.001
7_sfmis	Loader	0.44	0.48	0.82	< 0.001
71_ext4j	Functions	0.73	0.91	1.00	< 0.001
74_fixsuite	TreeView	0.20	0.23	0.80	< 0.001
78_caloriecount	WindowHelper	0.27	0.89	1.00	< 0.001
8_gfarcegestionfa	ModifTableStockage	0.73	0.83	0.80	< 0.001
80_wheelwebtool	MethodWriter	0.27	0.33	0.89	< 0.001
81_javathena	UserManagement	0.12	0.16	0.98	< 0.001
83_xbus	RecordTypeDescriptionChecker	0.19	0.21	0.75	< 0.001
84_ifx-framework	BankSvcRq_Type	0.93	1.00	1.00	< 0.001
85_shop	JSTerm	0.34	0.46	0.92	< 0.001
88_jopenchart	CoordSystemUtilities	0.35	0.40	0.84	< 0.001
90_dcparseargs	ArgsParser	0.94	0.97	0.94	< 0.001
92_jcvi-javacommon	Nucleotide	0.82	0.99	1.00	< 0.001
93_quickserver	QuickServer	0.34	0.45	0.99	< 0.001
94_jclo	JCLO	0.55	0.61	0.98	< 0.001
95_celwars2009	Entity	0.13	0.13	0.62	0.006
96_heal	MetadataDAO	0.20	0.21	0.91	< 0.001
Average		0.40	0.43	0.68	

* There were 45 classes with no statistically significant difference

Table 13: For each *top* class, the table reports the average *Line Coverage* obtained by the *Whole* approach and by the *Archive* approach.

Project	Class	Whole	Archive	$\hat{A}_{12}$	p-value
1_tullibee	EClientSocket	0.12	0.14	0.98	< 0.001
12_dsachat	Challenge	0.54	0.49	0.33	< 0.001
13_jdbacl	SQLParser	0.12	0.38	1.00	< 0.001
15_beanbin	LuceneIndexManager	0.39	0.29	0.01	< 0.001
16_templatedetails	JoomlaOutput	0.92	0.93	0.72	0.001
17_inspirento	MonthlyCalendar	0.77	0.87	0.97	< 0.001
18_jsecurity	AuthorizingRealm	0.82	0.76	0.20	< 0.001
19_jmca	JavaParser	0.15	0.37	1.00	< 0.001
23_jwbf	SimpleArticle	0.98	0.79	0.12	< 0.001
24_saxpath	XPathLexer	0.87	0.92	1.00	< 0.001
26_jipa	Main	0.29	0.42	0.99	< 0.001
30_bpmail	EmailFacadeImpl	0.26	0.25	0.69	< 0.001
33_javaviewcontrol	JVCParserTokenManager	0.13	0.09	0.16	< 0.001
34_sbmlreader2	SBMLGraphReader	0.20	0.19	0.32	< 0.001
36_schemaspy	Config	0.85	0.88	0.95	< 0.001
38_javabullboard	PropertyUtils	0.66	0.83	1.00	< 0.001
41_follow	FollowAppAttributes	0.93	0.93	0.77	0.035
44_summa	DatabaseStorage	0.02	0.01	0.41	0.048
46_nutzenportfolio	AuswahlfeldDaoService	0.17	0.16	0.26	< 0.001
49_diebierse	Drink	0.93	0.95	0.77	< 0.001
50_biff	Scanner	0.12	0.11	0.06	< 0.001
51_jiprof	MethodWriter	0.33	0.25	0.10	< 0.001
53_shp2kml	GeomConverter	0.99	0.99	0.67	< 0.001
54_db-everywhere	MysqlTableStructure	0.41	0.37	0.27	0.002
55_lavalamp	DeviceProperties	0.99	0.95	0.04	< 0.001
60_sugar	SCLLexer	0.35	0.48	0.92	< 0.001
61_noen	EFSMGenerator	0.45	0.49	0.94	< 0.001
62_dom4j	XMLWriter	0.60	0.64	0.88	< 0.001
64_jtailgui	JTailLogger	0.53	0.53	0.50	< 0.001
7_sfmis	Loader	0.51	0.49	0.27	< 0.001
70_echodep	HaSMETSValidator	0.20	0.02	0.30	< 0.001
71_ext4j	Functions	0.84	0.91	0.97	< 0.001
72_battlecry	bcGenerator	0.05	0.35	0.80	< 0.001
74_fixsuite	TreeView	0.39	0.40	0.84	< 0.001
75_openhre	LdapService	0.45	0.45	0.62	0.039
77_io-project	ClientGroup	0.92	0.89	0.36	< 0.001
78_caloriecount	WindowHelper	0.27	0.67	1.00	< 0.001
8_gfarcegestionfa	ModifTableStockage	0.75	0.85	0.92	< 0.001
80_wheelwebtool	MethodWriter	0.30	0.26	0.21	< 0.001
81_javathena	UserManagement	0.14	0.26	1.00	< 0.001
82_ipcalculator	IPv4	0.42	0.41	0.26	< 0.001
83_xbus	RecordTypeDescriptionChecker	0.31	0.34	0.68	0.003
84_ifx-framework	BankSvcRq_Type	0.91	1.00	1.00	< 0.001
85_shop	JSTerm	0.41	0.44	0.72	0.002

Continued on next page

Table 13: Continued.

Project	Class	Whole	Archive	$\hat{A}_{12}$	p-value
88_jopenchart	CoordSystemUtilities	0.42	0.46	0.96	< 0.001
9_falselight	Services	0.84	0.85	0.58	0.046
91_classviewer	ClassInfo	0.90	0.82	0.01	< 0.001
92_jcvi-javacommon	Nucleotide	0.65	0.98	1.00	< 0.001
93_quickserver	QuickServer	0.45	0.57	1.00	< 0.001
94_jclo	JCLO	0.64	0.67	0.93	< 0.001
95_celwars2009	Entity	0.24	0.22	0.00	< 0.001
96_heal	MetadataDAO	0.21	0.22	0.97	< 0.001
Average		0.45	0.47	0.56	

* There were 48 classes with no statistically significant difference

Table 14: For each *top* class, the table reports the average *Weak Mutation Score* obtained by the *Whole* approach and by the *Archive* approach.

Project	Class	Whole	Archive	$\hat{A}_{12}$	p-value
1_tullibee	EClientSocket	0.19	0.23	0.69	< 0.001
10_water-simulator	SuiteGUI	0.00	0.00	0.50	< 0.001
11_imsmart	MContentManagerFileNet	0.17	0.14	0.62	< 0.001
12_dsachat	Challenge	0.43	0.53	0.51	< 0.001
13_jdbacl	SQLParser	0.30	0.41	0.84	< 0.001
14_omjstate	Transition	0.97	0.96	0.58	< 0.001
15_beanbin	LuceneIndexManager	0.29	0.29	0.75	0.030
16_templatedetails	JoomlaOutput	0.87	0.90	0.89	< 0.001
17_inspirento	MonthlyCalendar	0.64	0.76	0.75	< 0.001
18_jsecurity	AuthorizingRealm	0.74	0.81	0.76	< 0.001
19_jmca	JavaParser	0.14	0.20	0.99	0.028
2_a4j	ProductDetails	0.81	0.84	0.77	< 0.001
22_byuic	TokenStream	0.51	0.62	0.62	< 0.001
24_saxpath	XPathLexer	0.72	0.81	1.00	< 0.001
25_jni-inchi	INCHI_KEY	0.96	0.94	0.50	< 0.001
26_jipa	Main	0.35	0.59	0.95	< 0.001
27_gangup	AbstractMap	0.01	0.01	0.50	< 0.001
3_gaj	GAAlgorithm	0.83	0.75	0.50	< 0.001
30_bpmail	EmailFacadeImpl	0.17	0.13	0.70	< 0.001
31_xisemele	WriterEditorImpl	0.02	0.00	0.50	< 0.001
32_httpanalyzer	HttpAnalyzerView	0.01	0.00	0.50	< 0.001
33_javaviewcontrol	JVCParserTokenManager	0.16	0.20	0.57	0.010
34_sbmlreader2	SBMLGraphReader	0.62	0.89	0.51	< 0.001
35_corina	GrapherPanel	0.00	0.01	0.50	< 0.001
36_schemaspy	Config	0.80	0.88	0.84	< 0.001
37_petsoar	Pet	0.98	0.96	0.50	< 0.001
38_javabullboard	PropertyUtils	0.76	0.95	1.00	< 0.001
39_diffi	StringIncrementor	0.80	0.79	0.59	0.007
4_rif	RIFInvoker	0.05	0.04	0.51	< 0.001
40_glengineer	Scheme	0.67	0.82	0.82	< 0.001
41_follow	FollowAppAttributes	0.94	0.96	0.62	< 0.001
42_asphodel	DefaultRepositoryManager	0.01	0.00	0.50	< 0.001
43_lilith	MainFrame	0.00	0.00	0.53	< 0.001
46_nutzenportfolio	AuswahlfeldDaoService	0.10	0.08	0.86	< 0.001
49_diebierse	Drink	0.91	0.96	0.82	< 0.001
50_biff	Scanner	0.13	0.10	0.57	< 0.001
51_jiprof	MethodWriter	0.31	0.47	0.74	< 0.001
54_db-everywhere	MysqlTableStructure	0.48	0.61	0.53	< 0.001
55_lavalamp	DeviceProperties	0.98	0.97	0.50	< 0.001
56_jhandballmoves	HandballModel	0.65	0.71	0.67	< 0.001
57_hft-bomberman	ServerGameModel	0.42	0.75	0.49	< 0.001
58_fps370	Fps370Panel	0.01	0.02	0.50	< 0.001
59_mygrid	Job	0.92	0.90	0.48	< 0.001
6_jnfe	TransportKeyStoreBean	0.98	0.96	0.50	< 0.001
61_noen	EFSMGenerator	0.39	0.45	0.72	< 0.001
62_dom4j	XMLWriter	0.56	0.72	0.90	< 0.001
63_objectexplorer	ExplorerFrameEventConverter	0.01	0.01	0.50	< 0.001
64_jtailgui	JTailLogger	0.22	0.30	0.50	< 0.001
65_gsftp	RemoteFileBrowser	0.00	0.00	0.50	< 0.001
66_openjms	URI	0.79	0.89	0.63	< 0.001
67_gae-app-manager	QuotaDetailsParser	0.15	0.16	0.50	< 0.001
68_biblestudy	ServletConnection	0.01	0.00	0.50	< 0.001
69_lhamacaw	SQLVariableManager	0.04	0.15	1.00	< 0.001
7_sfmis	Loader	0.39	0.37	0.76	0.005
70_echodep	HaSMETSValidator	0.03	0.05	0.69	0.005
71_ext4j	Functions	0.76	0.86	0.92	< 0.001
73_fim1	ModernChatServer	0.00	0.00	0.50	< 0.001
74_fixsuite	TreeView	0.23	0.26	0.74	< 0.001
75_openhre	LdapService	0.19	0.27	0.59	< 0.001
76_dash-framework	Main	0.30	0.11	0.50	< 0.001

Continued on next page

Table 14: Continued.

Project	Class	Whole	Archive	$\hat{A}_{12}$	p-value
77_io-project	ClientGroup	0.93	0.94	0.67	< 0.001
78_caloriecount	WindowHelper	0.36	0.74	1.00	< 0.001
8_gfarcegestionfa	ModifTableStockage	0.84	0.91	0.67	< 0.001
80_wheelwebtool	MethodWriter	0.33	0.48	0.73	< 0.001
81_javathena	UserManagement	0.14	0.28	0.99	< 0.001
82_ipcalculator	IPv4	0.17	0.13	0.50	< 0.001
83_xbus	RecordTypeDescriptionChecker	0.31	0.44	0.72	< 0.001
84_ifx-framework	BankSvcRq_Type	0.86	1.00	0.50	< 0.001
85_shop	JSTerm	0.42	0.62	0.83	< 0.001
86_at-robots2-j	AtRobotLineLexer	0.14	0.15	0.50	< 0.001
87_jaw-br	JanelaPrincipal	0.00	0.00	0.50	< 0.001
88_jopenchart	CoordSystemUtilities	0.48	0.69	0.96	< 0.001
89_jiggler	ImageOps	0.14	0.20	0.50	< 0.001
9_falselight	Services	0.77	0.83	0.69	< 0.001
90_dcparseargs	ArgsParser	0.94	0.97	0.80	< 0.001
91_classviewer	ClassInfo	0.86	0.91	0.59	< 0.001
92_jcvi-javacommon	Nucleotide	0.88	0.99	0.98	< 0.001
93_quickserver	QuickServer	0.49	0.72	0.95	< 0.001
94_jclo	JCLO	0.54	0.55	0.78	0.014
95_celwars2009	Entity	0.37	0.63	0.50	< 0.001
96_heal	MetadataDAO	0.22	0.26	0.89	< 0.001
97_feudalismgame	Battle	0.08	0.17	0.50	< 0.001
98_trans-locator	FoxHuntFrame	0.01	0.00	0.50	< 0.001
99_newzgrabber	Downloader	0.19	0.26	0.46	< 0.001
Average		0.42	0.48	0.64	

* There were 16 classes with no statistically significant difference