Commonality-Driven Unit Test Generation

Björn Evers, Pouria Derakhshanfar, Xavier Devroey^(✉), and Andy Zaidman

Delft University of Technology, Delft, The Netherlands
b.evers@student.tudelft.nl, {p.derakhshanfar, x.d.m.devroey,a.e.zaidman}@tudelft.nl

Abstract. Various search-based test generation techniques have been proposed to automate the generation of unit tests fulfilling different criteria (*e.g.,* line coverage, branch coverage, mutation score, *etc.*). Despite several advances made over the years, search-based unit test generation still suffers from a lack of guidance due to the limited amount of information available in the source code that, for instance, hampers the generation of complex objects. Previous studies introduced many strategies to address this issue, *e.g.,* dynamic symbolic execution or seeding, but do not take the internal execution of the methods into account. In this paper, we introduce a novel secondary objective called *commonality score*, measuring how close the execution path of a test case is from reproducing a *common* or *uncommon* execution pattern observed during the operation of the software. To assess the commonality score, we implemented it in EvoSuite and evaluated its application on 150 classes from JabRef, an open-source software for managing bibliography references. Our results are mixed. Our approach leads to test cases that indeed follow *common* or *uncommon* execution patterns. However, if the commonality score can have a positive impact on the structural coverage and mutation score of the generated test suites, it can also be detrimental in some cases.

Keywords: Search-based software testing · Automated unit testing · Common paths coverage · Secondary objective

1 Introduction

Search-based test case generation enables the automated generation of test cases according to predefined criteria. Among the different approaches, white-box search-based test case generation [13, 15, 29] relies on evolutionary algorithms to generate test cases from source code and maximize structural coverage [15, 29] or mutation score [16]. Previous research has shown that automatically generated tests are effective for coverage and fault finding [3, 23, 28], can also find real faults [2], and are useful for debugging [5].

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-3-030-59762-7_9) contains supplementary material, which is available to authorized users.

Despite several advances, search-based unit test generation still faces many challenges. Among those are (i) the crafting of complex objects and values used during test generation [2], and (ii) the indirect coverage of encapsulated elements (*e.g.,* private methods and class attributes) through the invocation of specific paths in public methods [32]. Various approaches address those challenges by relying on *dynamic symbolic execution* to generate complex objects and values using constraint solvers [18,19,21,24]; *seeding* to identify objects and values from the application source and test code that are later reused during the search [31]; or class usages, learned from static analysis of the source code [8,17] and dynamic execution of the existing tests [8], and used to generate realistic objects.

However, if complex objects and values can indeed lead to an improvement in the coverage, it does not always succeed in covering all the elements of a class under test. For instance, if the indirect coverage of a private method requires specific executions paths in a public method, the current fitness functions will not be able to provide sufficient guidance to the search process [32].

In this paper, we hypothesize that common and uncommon execution paths, observed during the actual operation of the system, can lead to better guidance of the search process, and hence, better coverage. Complementing previous work on seeding [31], which is aimed at triggering different execution paths in the methods under test, we consider the *commonality* of those execution paths. For that, we approximate commonality using weights for the different code blocks, and define a secondary objective called the *commonality score*, denoting how close an execution path is from common or uncommon executions of the software.

We implemented the commonality score in EvoSuite [13] and evaluated it on 150 classes from JabRef, an open-source bibliography reference manager, for common and uncommon behaviors. We compare the commonality score (**RQ.1**), the structural coverage (**RQ.2**), and the fault-finding capabilities (**RQ.3**) of the thus generated tests to tests generated by the standard EvoSuite implementation. Our results are mixed but show that this secondary objective significantly improves the number of covered common paths in 32.6% of the classes. Although the average structural coverage remains stable, the commonality score significantly improves the line (resp. branch) coverage in three (resp. four) classes, but also negatively impacts the coverage for eight (resp. nine) classes. Finally, the commonality score impacts the number of killed mutants for 22 classes (11 positively and 11 negatively). Our implementation is openly available at https://github.com/STAMP-project/evosuite-ramp, and the replication package of our evaluation and data analysis have been uploaded to Zenodo [11,12].

2 Background and Related Work

2.1 Search-Based Unit Test Generation

Search-based unit test generation has been extensively investigated by prior studies [13,15,29]. These studies have confirmed that it achieves a high level of coverage [15,29], detects faults in real-world applications [2,16], and reduces the debugging costs [30]. Most search-based unit test generation approaches abstract the source code of a method to a control flow graph:

Definition 1 (Control Flow Graph (CFG) [1]). *A control flow graph for a method m is a directed graph $G = (B, E)$, where B is the set of* basic blocks *(i.e., sequences of statements that do not have any branch before the last statement), E is the set of* control flow edges *connecting the basic blocks.*

For instance, for the method with the pseudo-code presented in Fig. 1(a), the corresponding CFG for this method is depicted in Fig. 1(b).

Search-based software unit test generation approaches use meta-heuristics to *evolve* a set of test cases. These techniques start with generating an *initial population* of randomly produced test cases. The *fitness* of each individual in the population is evaluated using a fitness function, which is usually defined according to the coverage in the CFGs of the target class. Next, a subset of the fittest individuals is *selected* for evolution and leads to the generation of a new population. The evolving process contains three steps: (i) *crossover*, which randomly mixes two selected individuals to generate new offspring; (ii) *mutation*, which randomly changes, adds, or removes a statement in an individual; and (iii) *insertion*, which reinserts the modified individuals into the population for the next iteration of the algorithm. This process continues until either satisfactory individuals are found, or the time budget allocated for the search is consumed. Among the different approaches, EvoSuite [13] uses genetic algorithms to evolve Java test suites in order to cover a class under test.

MOSA [27] and *DynaMOSA* [29] are two new many-objectives genetic algorithms proposed for unit test generation. These algorithms consider test cases as individuals and incorporate separate fitness functions for separate coverage goals (*e.g.,* covering each branch in the CFGs will be an independent search objective). They use non-dominated fronts to generate test cases in the direction of multiple coverage goals in parallel, and thereby generate tests aiming to cover specific goals, while not letting the test generation be trapped for covering a single goal. Panichella *et al.* [27] show that *MOSA* outperforms the original EvoSuite approach in terms of structural coverage and mutation score.

In this paper, we use *MOSA* to automatically generate test cases according to the collected logs during the production phase. Future work includes the evaluation of our approach with other multi and many-objectives algorithms, like *DynaMOSA*.

2.2 Usage-Based Test Generation

The majority of search-based test generation techniques aim to achieve high coverage for various metrics (*e.g.,* line coverage, branch coverage, or more recently mutation coverage). Despite their considerable achievements, they do not consider the execution patterns observed in production use for automatic generation of unit tests. Hence, Wang *et al.* [37] investigated how developer-written tests and automatically generated tests represent typical execution patterns in production. Their study confirms that these tests are not a proper representation of real-world execution patterns.

The behavior of actual users may reveal faults, which are not detected by the existing test cases. For instance, a piece of code in the software under test that is not often used in practice may be left relatively untested because it is rarely exercised in production. A recent method from Wang *et al.* [38], based on symbolic execution, recreates users behaviors using log data from a system run in production, which has allowed to find the same faults in a system encountered by a user. This paper aims to expand upon generating tests based on the actual usage of a system at the unit level. In contrast to Wang *et al.* [38], where the aim is to replicate a full behavior executed by a user by using symbolic execution, we aim to guide the search process in a genetic algorithm towards executing common or uncommon behaviors. In the same vein as Wang et al. [38], log data is used to determine the execution counts of code branches.

Other approaches consider user feedback [20], or *usage models* of the application and *statistical testing* [9,22,34,35] to generate and prioritize test cases at the system level. A usage model consists in a state machine where transitions have been labelled with a probability of being executed. Unlike those approaches, we consider test case generation at the unit level.

3 Test Generation for Common and Uncommon Behaviors

Intuitively, *commonality* describes to what extent a test exercises code branches that are executed often during the normal operation of the system under test. If a test executes branches that are often (respectively rarely) executed in practice, it will have a high (respectively low) *commonality score*. The commonality score has a value between 0 and 1 and is computed based on an annotated control flow graph [1]:

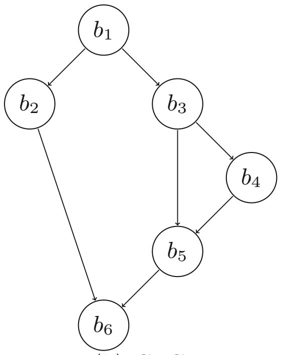
Definition 2 (Annotated Control Flow Graph). *An annotated control flow graph is a directed graph $G = (B, E, \gamma)$, where $G = (B, E)$ is a control flow graph, and $\gamma : B \rightarrow \mathbb{R}$ is a labelling function giving for the basic blocks in B an* execution weight *denoting how often the block is executed during operations.*

The *execution weights* can be derived from the operation logs of the system, an instrumented version of the system (like in our evaluation), or assigned manually.

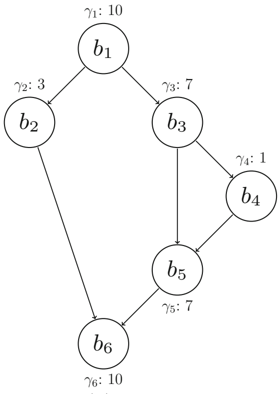
Let us define the *commonality score*. For a test case, its commonality score depends only on the branches it covers and on the highest and lowest execution weights in the class under test. Branches without execution weights are ignored and branches covered multiple times (*e.g.,* in a loop) are counted only once.

```
/* Branch 1     */
if condition1 then
    /* Branch 2  */
    ...;
else
    /* Branch 3  */
    ...;
    if condition2
     then
        /* Branch 4
           */
        ...;
    /* Branch 5  */
    ...;
/* Branch 6     */
...;
```

(a) Pseudo-code.



(b) CFG.



(c) ACFG.

Fig. 1. Example of pseudo-code and its corresponding annotated control flow graph. The γ_i indicate to the execution weight of the node.

Definition 3 (Commonality score). *For a test case t executing n basic blocks b_i labelled by a function γ, the highest execution weight in the class under test h, the lowest execution weight in the class under test l, the commonality score of t, denoted $c(t)$ is defined as:*

$$c(t) = \frac{\sum_{i=1}^{n} (\gamma b_i - l)}{n \times (h - l)}$$

The commonality score for a *test suite* s is defined as the average of the commonality scores of its test cases: $c(s) = \left(\sum_{t_i \in s} c(t_i)\right) / |s|$.

For instance, considering a class containing a single method with the pseudo-code presented in Fig. 1(a), the corresponding annotated control flow graph in Fig. 1(b), and a test suite containing three test cases t_1 covering (b_1, b_2, b_6), t_2 covering (b_1, b_3, b_5, b_6), and t_3 covering (b_1, b_3, b_4, b_5, b_6). The commonality scores are $c(t_1) = ((10-1)) + (3-1) + (10-1)) / (3 \times (10-1)) = 20/27 \approx 0.741$, $c(t_2) = 5/6 \approx 0.833$, and $c(t_3) = 2/3 \approx 0.667$.

3.1 Commonality as a Secondary Objective

Secondary objectives are used to choose between different test cases in case of a tie in the main objectives. For instance, the default secondary objective used by MOSA [27] minimizes the test case length (*i.e.,* the number of statements) when two test cases satisfy the same main objectives (*e.g.,* cover the same branches). Using test case length minimization as a secondary objective addresses the *bloating effect* [33] by preventing the search process from always generating longer test cases. Since this is a desirable property, we combine the test case length minimization with the commonality of the test case using a weighted sum when comparing two test cases.

Definition 4 (Commonality secondary objective). *For two test cases t_1, t_2 with lengths l_1, l_2, the comparison between the two test cases is done using the following formula:*

$$common(t_1, t_2) = \frac{\left(\alpha\left(\frac{l_1}{l_2}\right) + \beta\left(\frac{1-c(t_1)}{1-c(t_2)}\right)\right)}{(\alpha + \beta)}$$

If $common(t_1, t_2) \leq 1$, then t_1 is kept, otherwise t_2 is kept.

Similarly, for the uncommonality between two test cases, we will have the following definition.

Definition 5 (Uncommonality secondary objective). *For two test cases t_1, t_2 with lengths l_1, l_2, the comparison between the two test cases is done using the following formula:*

$$uncommon(t_1, t_2) = \frac{\left(\alpha\left(\frac{l_1}{l_2}\right) + \beta\left(\frac{c(t_1)}{c(t_2)}\right)\right)}{(\alpha + \beta)}$$

If $uncommon(t_1, t_2) \leq 1$, then t_1 is kept, otherwise t_2 is kept.

In our evaluation, we use commonality and uncommonality with MOSA to answer our different research questions.

4 Empirical Evaluation

To assess the usage of commonality as a secondary objective for test case generation, we performed an empirical evaluation using 150 classes from JabRef[1], an open source bibliography reference manager, to answer the following research questions:

RQ.1. How does the *commonality score* of the generated tests compare when using the *common*, *uncommon*, and *default* secondary objectives?
RQ.2. How does the *line* and *branch coverage* of the generated tests compare when using the *common*, *uncommon*, and *default* secondary objectives?
RQ.3. How does the *mutation score* of the generated tests compare when using the *common*, *uncommon*, and *default* secondary objectives?

We implemented the secondary objectives from Sect. 3 in EvoSuite [13], a state-of-the-art white-box unit test generator tool for Java. Our implementation is openly available at https://github.com/STAMP-project/evosuite-ramp, and the replication package of our evaluation and data analysis have been uploaded to Zenodo [11,12].

[1] https://www.jabref.org.

4.1 Subject and Execution Weights

Collecting Execution Weights. For our evaluation, we choose JabRef (46 KLOC), an open-source Java bibliography reference manager with a graphical user interface working with BibTex files. To determine the execution weights of the different branches, we instrumented JabRef using Spoon[2] and added log statements producing a message with a unique identifier each time a branch is executed. These identifiers are then mapped to a source code location, identified by the *class name*, the *method name*, and the *line number*. Furthermore, the number of occurrences of the identifier in the log messages is established. We then asked five people (including the first author) to use our modified JabRef implementation to perform various tasks (adding a reference, updating a reference, removing a reference, *etc.*) and collected the produced logs. In an industrial context, operations logs can be analyses and traced back to the source code to identify the execution weights [39].

Classes Under Test. We sampled 150 classes. We excluded classes from the `org.-jabref.gui` and `org.jabref.logic.importer.fileformat` packages as they respectively work with JavaFX and perform input-output operations. From the remaining classes and following the best practices of the search-based unit testing community [26], we selected 75 classes with the highest cyclomatic complexity, as classes with a higher cyclomatic complexity are harder to process for unit test generation tools and 38 classes with the largest number of lines of code. Additionally, we selected 37 classes that were executed the most by our modified JabRef implementation.

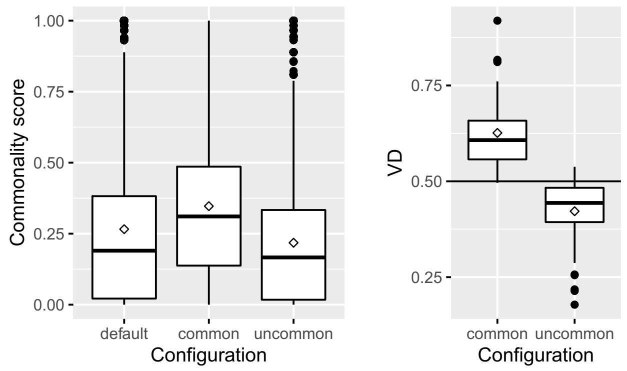
Configuration Parameters. We ran EvoSuite with the default coverage criteria (line, branch, exception, weak mutation, input, output, method, method without exceptions, and context branch) and three different secondary objectives: (i) *default*, minimizing the test case length, (ii) *commonality*, as described in Definition 4, and (iii) *uncommonality*, as described in Definition 5. We executed EvoSuite on each class under test 30 times with the MOSA algorithm [27] and a search budget of three minutes, offering a good compromise between runtime and coverage [15,28]. All other configuration parameters were left to their default value.

4.2 Data Analysis

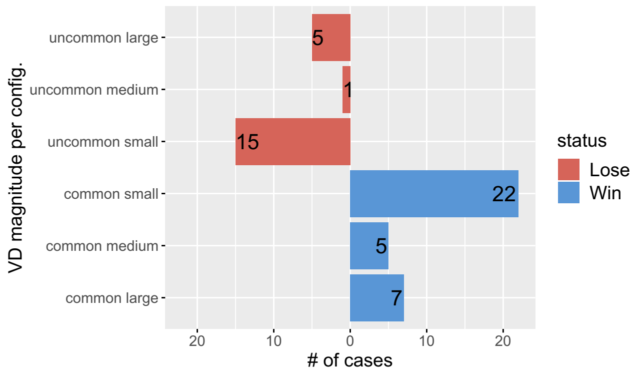
For each of the 13,500 execution (150 classes $\times$ 30 repetitions $\times$ 3 configurations), we collected the commonality score and structural coverage information from EvoSuite. Additionally, we performed a mutation analysis of the generated test suites using Pit [7]. For 46 classes (out of 150), EvoSuite could not complete 30 executions using our different configurations. We excluded those classes to keep the comparison fair and performed our analysis on the 104 remaining classes.

2 http://spoon.gforge.inria.fr/.

To compare the commonality score, the structural coverage, and the mutation score, we used the non-parametric Wilcoxon Rank Sum test, with $\alpha = 0.05$ for Type I error, and the Vargha-Delaney statistic $\widehat{A}_{12}$ [36] to evaluate the effect size between two configurations. An $\widehat{A}_{12}$ value lower than 0.5 for a pair of configurations (A,B) indicates that A increases the score or coverage compared to B, and a value higher than 0.5 indicates the opposite. The Vargha-Delaney magnitude measure also allows partitioning the results into three categories having large, medium, and small impact [36].



(a) Test cases commonality scores. (b) Effect sizes $\widehat{A}_{12}$.



(c) Effect sizes $\widehat{A}_{12}$ magnitudes.

Fig. 2. Test cases commonality values and comparison to *default*. Diamonds ($\diamond$) indicate mean values and horizontal bars (–) indicate median values.

5 Results

5.1 Commonality Score (RQ.1)

In this section we answer the question: *How does the commonality score of the generated tests compare when using the common, uncommon, and default secondary objectives?*

Figure 2 illustrates the impact of using *commonality* and *uncommonality*, as the secondary objective, on the commonality score of the generated test cases. Figure 2(a) shows that the average and median of the commonality score is improved by 8% and 12%, respectively, compared to *default* when using *commonality* as secondary objective. In parallel, using *uncommonality* as secondary objective reduces the commonality score by, on average, 5% (2.5% for median) compared to *default*. Moreover, Fig. 2(c) presents the number of cases (*i.e.,* classes used as the target class for unit testing), in which the application of *commonality* and *uncommonality* significantly (*p-value* < 0.05) changes the commonality score with effect size magnitude of large, medium, or small. As we can see in this figure, utilizing *commonality* always leads to a significant improvement in the commonality score (blue bars), and in contrast, using *uncommonality* always reduces this score (red bars). In total, *commonality* significantly improves the commonality score in 34 cases (32.6% of classes), and *uncommonality* significantly reduces this score in 21 classes (20.1% of cases). Figure 2(b) depicts the effect sizes of differences observed in these cases. Consistent with the previous figures, the average effect size ($\widehat{A}_{12}$) achieved by *commonality* is higher than 0.5 (*i.e.,* commonality score has been improved). However, this value is lower than 0.5 for *uncommonality*.

Summary. Using *commonality* as secondary objective in the EvoSuite search-based test case generation process leads to test cases that exhibit an improved commonality score. In parallel, the application of *uncommonality* leads to the reduction of the commonality score.

5.2 Structural Coverage (RQ.2)

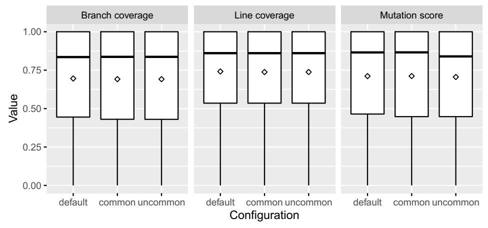
In this section we provide an answer to the following research question: *How does the line and branch coverage of the generated tests compare when using the common, uncommon, and default secondary objectives?*

Figure 3 shows the line and branch coverage achieved by using *commonality* and *uncommonality* as secondary objectives compared to *default*. Figure 3(a) indicates that the average coverage is the same for all of the assessed configurations.

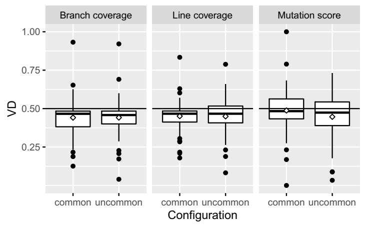
Looking at the comparison of the structural coverage values achieved by each secondary objective in each class, we can see that the line and branch coverage is significantly impacted by *commonality* and *uncommonality* in some cases. Figure 3(c) presents the number of cases that these secondary objectives significantly (*p-value* < 0.05) reduce ($\widehat{A}_{12} < 0.5$) or increase ($\widehat{A}_{12} > 0.5$) the line and branch coverage with effect size magnitude small, medium, or large. According to this figure, in general, utilizing *commonality* leads to a significant improvement for line and branch coverage in three and four classes, respectively. Nevertheless, this secondary objective reduced the line and branch coverage in eight and nine classes, respectively. A list of these classes is provided as the supplementary material to this submission.

Also, we can see a similar result for *uncommonality*: significant improvements in three and five classes and significant reductions in seven and nine cases for line and branch coverage. Since the number of cases in which *commonality* and *uncommonality* lead to a significantly lower structural coverage is higher than the number of cases in which we see a significant improvement in coverage, the average effect size of differences (Fig. 3(b)) is slightly less than 0.5 for both line (0.47 for both secondary objectives) and branch coverage (0.46 for both).

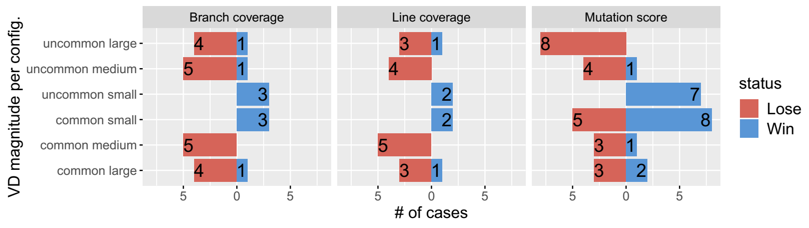
Summary. On average, using *commonality* or *uncommonality* does not impact the line and branch coverage. However, these two secondary objectives can significantly impact the structural coverage in specific cases.



(a) Test suites coverage and mutation score.



(b) Effect sizes $\widehat{A}_{12}$.



(c) Effect sizes $\widehat{A}_{12}$ magnitudes.

Fig. 3. Test suites coverage and mutation score, and comparison to *default*. Diamonds ($\diamond$) indicate mean values and horizontal bars ($-$) indicate median values.

5.3 Mutation Analysis (RQ.3)

In the final research question we reflect on *How does the mutation score of the generated tests compare when using the common, uncommon, and default secondary objectives?*

Figure 3(a) depicts the mutation score achieved by using *commonality* and *uncommonality* compared to *default*. Like line and branch coverage, the average mutation scores achieved by these secondary objectives is similar to the one achieved by *default*. However, Fig. 3(c) shows that *commonality* and *uncommonality* can significantly ($p\text{-}value < 0.05$) impact the mutation score achieved by unit test generation. The *commonality* secondary objective significantly increases the number of mutants killed for 11 classes but, at the same time, also decreases the mutation score in another 11 cases. Moreover, *uncommonality* significantly changes the mutation score in 20 cases (8 wins against 12 losses). Figure 3(b) shows the effect size of differences in these cases for both *commonality* and *uncommonality* secondary objectives. According to this Figure, the average $\widehat{A}_{12}$ estimations are 0.49 and 0.47. Since these values are lower than 0.5, on average, the difference achieved by these two secondary objectives is negative. However, the outliers in this figure show us that the effect sizes of *commonality* above 0.75 in some specific cases. Hence, the graphs in Fig. 3(c) indicate that using *commonality* and *uncommonality* can improve the mutation score in specific cases.

Summary. On average, using *commonality* or *uncommonality* does not have any effect on the mutation score achieved by the generated test suites. However, these two secondary objectives can significantly change the killed mutants in some cases.

6 Discussion

6.1 Execution Weights

In our evaluation, we collected execution weights using an instrumented version of JabRef distributed to five different users. As a result, a large number of log messages allowed us to have execution weights for many different classes. Such data collection is not realistic in an industrial setting as the collection and analysis of log data is challenging for large applications [4]. It is likely that the collected data will not cover the complete system but only a subset of its classes.

However, we believe that the development of scalable *software analytics* practices [25] represents an opportunity to include information from the software operations environment in various development activities, including testing [4,6]. For instance, Winter *et al.* [39] recently brought information about the number of times a log statement is executed to the developer's IDE to raise awareness about the load it represented during the operations of the system. Similar information can be collected for seeding [8,31] or annotating a control flow graph, like in our approach, and allow developers to generate new tests from their IDE.

Finally, our current approach considers execution weights individually to approximate usages, which allows us to compute a commonality score quickly. Different definitions of the commonality score based, for instance, on full and partial execution paths identified from the operations logs are possible. Those finer grained definitions of commonality would allow to take multiple executions of the same code blocks (like loops) into account, at the expense of a higher computational cost. Exploration and evaluation of such definitions are left for future work.

6.2 Impact on Mutation Analysis

After manual investigation of the generated tests[3], we saw that the classes on which *commonality* performs relatively well are often executed. The class on which the performance was especially good compared to *default* was an enum class (`org.jabref.logic.util.StandardFileType`). In a large majority of the cases (25 out of 30), the tests generated using the *commonality* secondary objective contain a method sequence that is not present in the tests generated by *default*. We inspected the execution counts of individual branches stemming from the operational use of the system. From this inspection, we found a single branch in the code of method `getExtensions` that has been executed as part of the usage scenarios from all participants.

This method is consistently involved in the test cases that kill the mutants that the tests generated by *default* fail to kill most of the time. This supports our initial assumption that using *commonality* can drive the search process to cover the code in a different way, possibly finding different kinds of faults.

6.3 Usefulness for Debugging

The end-goal of any test suite is to identify faults in source code and help the developer during her debugging activities. Our evaluation measured the fault-finding capabilities of the generated test suites, but did not investigate their usefulness for debugging. Previous research has confirmed that automatically generated tests can find faults in real software [2] and are useful for debugging [5].

However, there remain several challenges, like the understandability of the generated tests [2,14]. Since the *commonality* and *uncommonality* secondary objectives aim to influence how the lines in a method are covered, we expect that it will also have an impact on the understandability of the generated tests. Future research will include the assessment of the debugging capabilities of the generated tests (*e.g.,* understandability, performance, readability, *etc.*).

6.4 Threats to the Validity

Internal Validity. We repeated each execution of EvoSuite 30 times to take the randomness of the generation process into account in our data analysis. We have tested our implementation of the *commonality* and *uncommonality* secondary objectives to reduce bugs' chance of influencing our results.

[3] The results and complete manual analysis are available online [10–12].

External Validity. We gathered execution data needed from a small number of people in a relatively structured manner. We cannot guarantee that those executions are representative of all the usages of JabRef. However, we believe that the diversity of the tasks performed by our users is enough for this evaluation. Also, the evaluation was performed using only one case study. Future research includes the repetition of the assessment on other Java applications.

Construct Validity. We relied on the reports produced by EvoSuite for structural coverage and the reports produced by Pit for mutation analysis to compare our different secondary objectives. The usage of those standard metrics allows the comparison of our results with other search-based unit test generation approaches.

Conclusion Validity. Our conclusions were only drawn based on statistically significant results with $\alpha = 0.05$. We used the standard non-parametric Wilcoxon Rank Sum test for significance and the Vargha-Delaney statistic for the effect size.

7 Conclusion and Future Work

In this paper, we introduced the *commonality score*, denoting how close an execution path is from common or uncommon executions of the software in production, and the *commonality* and *uncommonality* secondary objectives for search-based unit test generation. We implemented our approach in EvoSuite and evaluated it on JabRef using execution data from real usages of the application. Our results are mixed. The *commonality* secondary objective leads to an increase of the commonality score, and the *uncommonality* secondary objective leads to a decrease of the score, compared to the *default* secondary objective (**RQ.1**). However, results also show that if the commonality score can have a positive impact on the structural coverage (**RQ.2**) and mutation score (**RQ.3**) of the generated test suites, it can also be detrimental in some cases. Future research includes a replication of our evaluation on different applications and using different algorithms (*e.g., DynaMOSA*) to gain a deeper understanding of when to apply *commonality* and *uncommonality* secondary objectives; the exploration and assessment of different definitions of commonality; and an assessment of the generated tests regarding their usefulness for debugging.

Acknowledgements. This research was partially funded by the EU Horizon 2020 ICT-10-2016-RIA "STAMP" project (No. 731529).

References

1. Allen, F.E.: Control flow analysis. ACM SIGPLAN Not. **5**(7), 1–19 (1970). https://doi.org/10.1145/390013.808479

2. Almasi, M.M., Hemmati, H., Fraser, G., Arcuri, A., Benefelds, J.: An industrial evaluation of unit test generation: finding real faults in a financial application. In: ICSE-SEIP 2017, pp. 263–272. IEEE, May (2017). https://doi.org/10.1109/ICSE-SEIP.2017.27
3. Campos, J., Ge, Y., Albunian, N., Fraser, G., Eler, M., Arcuri, A.: An empirical evaluation of evolutionary algorithms for unit test suite generation. IST **104**, 207–235 (2018). https://doi.org/10.1016/j.infsof.2018.08.010
4. Candido, J., Aniche, M., van Deursen, A.: Contemporary software monitoring: a systematic literature review (2019). http://arxiv.org/abs/1912.05878
5. Ceccato, M., Marchetto, A., Mariani, L., Nguyen, C.D., Tonella, P.: Do automatically generated test cases make debugging easier? An experimental assessment of debugging effectiveness and efficiency. TOSEM **25**(1), 1–38 (2015). https://doi.org/10.1145/2768829
6. Chen, B., Song, J., Xu, P., Hu, X., Jiang, Z.M.J.: An automated approach to estimating code coverage measures via execution logs. In: ASE 2018, no. 3, pp. 305–316. ACM Press (2018). https://doi.org/10.1145/3238147.3238214
7. Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A.: PIT: a practical mutation testing tool for Java. In: ISSTA 2016, pp. 449–452. ACM (2016). https://doi.org/10.1145/2931037.2948707
8. Derakhshanfar, P., Devroey, X., Perrouin, G., Zaidman, A., Deursen, A.: Search-based crash reproduction using behavioural model seeding. STVR **30**(3), e1733 (2020). https://doi.org/10.1002/stvr.1733
9. Devroey, X., Perrouin, G., Cordy, M., Samih, H., Legay, A., Schobbens, P.Y., Heymans, P.: Statistical prioritization for software product line testing: an experience report. SoSyM **16**(1), 153–171 (2017). https://doi.org/10.1007/s10270-015-0479-8
10. Evers, B.: Unit test generation for common and uncommon behaviors. Master thesis, Delft University of Technology (2020). http://resolver.tudelft.nl/uuid:6d8a1835-9054-4e4a-a85f-99ac592978da
11. Evers, B., Derakhshanfar, P., Devroey, X., Zaidman, A.: Unit test generation for common and uncommon behaviors: dataset, June 2020. https://doi.org/10.5281/zenodo.3894711
12. Evers, B., Derakhshanfar, P., Devroey, X., Zaidman, A.: Unit test generation for common and uncommon behaviors: replication package, June 2020. https://doi.org/10.5281/zenodo.3897513
13. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: ESEC/FSE 2011, p. 416. ACM (2011). https://doi.org/10.1145/2025113.2025179
14. Fraser, G., Arcuri, A.: EvoSuite: on the challenges of test case generation in the real world. In: ICST 2013, pp. 362–369. IEEE, March 2013. https://doi.org/10.1109/ICST.2013.51
15. Fraser, G., Arcuri, A.: A large-scale evaluation of automated unit test generation using EvoSuite. TOSEM **24**(2), 1–42 (2014). https://doi.org/10.1145/2685612
16. Fraser, G., Arcuri, A.: Achieving scalable mutation-based generation of whole test suites. EMSE **20**(3), 783–812 (2015). https://doi.org/10.1007/s10664-013-9299-z
17. Fraser, G., Zeller, A.: Exploiting common object usage in test case generation. In: ICST 2011, pp. 80–89. IEEE, March 2011. https://doi.org/10.1109/ICST.2011.53
18. Galeotti, J.P., Fraser, G., Arcuri, A.: Improving search-based test suite generation with dynamic symbolic execution. In: ISSRE 2013, pp. 360–369. IEEE, November 2013. https://doi.org/10.1109/ISSRE.2013.6698889

19. Gouraud, S.D., Denise, A., Gaudel, M.C., Marre, B.: A new way of automating statistical testing methods. In: ASE 2001, pp. 5–12. IEEE, November 2001. https://doi.org/10.1109/ASE.2001.989785
20. Grano, G., Ciurumelea, A., Panichella, S., Palomba, F., Gall, H.C.: Exploring the integration of user feedback in automated testing of Android applications. In: SANER 2018, pp. 72–83. IEEE, March 2018. https://doi.org/10.1109/SANER.2018.8330198
21. Inkumsah, K., Xie, T.: Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In: ASE 2008, pp. 297–306. IEEE, September 2008. https://doi.org/10.1109/ASE.2008.40
22. Kallepalli, C., Tian, J.: Measuring and modeling usage and reliability for statistical Web testing. TSE **27**(11), 1023–1036 (2001). https://doi.org/10.1109/32.965342
23. Kracht, J.S., Petrovic, J.Z., Walcott-Justice, K.R.: Empirically evaluating the quality of automatically generated and manually written test suites. In: QSIC 2014, pp. 256–265. IEEE, October 2014. https://doi.org/10.1109/QSIC.2014.33
24. Lakhotia, K., Tillmann, N., Harman, M., de Halleux, J.: FloPSy - search-based floating point constraint solving for symbolic execution. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 142–157. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16573-3_11
25. Menzies, T., Zimmermann, T.: Software analytics: so what? IEEE Softw. **30**(4), 31–37 (2013)
26. Molina, U.R., Kifetew, F., Panichella, A.: Java unit testing tool competition - sixth round Urko. In: SBST 2018, pp. 22–29. ACM (2018). https://doi.org/10.1145/3194718.3194728
27. Panichella, A., Kifetew, F.M., Tonella, P.: Reformulating branch coverage as a many-objective optimization problem. In: ICST 2015, pp. 1–10. IEEE, April 2015. https://doi.org/10.1109/ICST.2015.7102604
28. Panichella, A., Kifetew, F.M., Tonella, P.: A large scale empirical comparison of state-of-the-art search-based test case generators. IST **104**, 236–256 (2018). https://doi.org/10.1016/j.infsof.2018.08.009
29. Panichella, A., Kifetew, F.M., Tonella, P.: Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. TSE **44**(2), 122–158 (2018). https://doi.org/10.1109/TSE.2017.2663435
30. Panichella, S., Panichella, A., Beller, M., Zaidman, A., Gall, H.C.: The impact of test case summaries on bug fixing performance: an empirical investigation, pp. 547–558. ACM (2016). https://doi.org/10.1145/2884781.2884847
31. Rojas, J.M., Fraser, G., Arcuri, A.: Seeding strategies in search-based unit test generation. STVR **26**(5), 366–401 (2016). https://doi.org/10.1002/stvr.1601
32. Salahirad, A., Almulla, H., Gay, G.: Choosing the fitness function for the job: automated generation of test suites that detect real faults. STVR **29**(4–5), e1701 (2019). https://doi.org/10.1002/stvr.1701
33. Silva, S., Dignum, S., Vanneschi, L.: Operator equalisation for bloat free genetic programming and a survey of bloat control methods. Genet. Program. Evolvable Mach. **13**(2), 197–238 (2012). https://doi.org/10.1007/s10710-011-9150-5
34. Sprenkle, S.E., Pollock, L.L., Simko, L.M.: Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour. STVR **23**(6), 439–464 (2013). https://doi.org/10.1002/stvr.1496
35. Tonella, P., Ricca, F.: Statistical testing of Web applications. SMR **16**(1–2), 103–127 (2004). https://doi.org/10.1002/smr.284

36. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J. Educ. Behav. Stat. **25**(2), 101–132 (2000). https://doi.org/10.3102/10769986025002101
37. Wang, Q., Brun, Y., Orso, A.: Behavioral execution comparison: are tests representative of field behavior? In: ICST 2017. IEEE, March 2017. https://doi.org/10.1109/icst.2017.36
38. Wang, Q., Orso, A.: Mimicking user behavior to improve in-house test suites. In: ICSE 2019. IEEE, May 2019. https://doi.org/10.1109/icse-companion.2019.00133
39. Winter, J., Aniche, M., Cito, J., van Deursen, A.: Monitoring-aware IDEs. In: ESEC/FSE 2019, pp. 420–431. ACM (2019). https://doi.org/10.1145/3338906.3338926