Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

Choosing The Fitness Function for the Job: Automated Generation of Test Suites that Detect Real Faults

Alireza Salahirad, Hussein Almulla, Gregory Gay

[1]Department of Computer Science & Engineering, University of South Carolina, Columbia, SC, United States of America

Correspondence
*Email: alireza@email.sc.edu, halmulla@email.sc.edu, greg@greggay.com

Summary

Search-based unit test generation, if effective at fault detection, can lower the cost of testing. Such techniques rely on fitness functions to guide the search. Ultimately, such functions represent test goals that approximate—but do not ensure—fault detection. The need to rely on approximations leads to two questions—*can fitness functions produce effective tests and, if so, which should be used to generate tests?* To answer these questions, we have assessed the fault-detection capabilities of unit test suites generated to satisfy eight white-box fitness functions on 597 real faults from the Defects4J database. Our analysis has found that the strongest indicators of effectiveness are a high level of code coverage over the targeted class and high satisfaction of a criterion's obligations. Consequently, the branch coverage fitness function is the most effective. Our findings indicate that fitness functions that thoroughly explore system structure should be used as primary generation objectives—supported by secondary fitness functions that explore orthogonal, supporting scenarios. Our results also provide further evidence that future approaches to test generation should focus on attaining higher coverage of private code and better initialization and manipulation of class dependencies.

KEYWORDS:
Search-based Test Generation, Automated Test Generation, Unit Testing, Adequacy Criteria, Search-based Software Engineering

1 | INTRODUCTION

Proper verification practices are needed to ensure that developers deliver reliable software. *Testing* is an invaluable, widespread verification technique. However, testing is a notoriously expensive and difficult activity [51], and with exponential growth in the complexity of software, the cost of testing has risen accordingly. Means of lowering the cost of testing without sacrificing verification quality are needed.

Much of that cost can be traced directly to the human effort required to conduct most testing activities, such as producing test input and expected output. One way of lowering such costs may lie in the use of automation to ease this manual burden [5]. Automation has great potential in this respect, as much of the invested human effort is in service of tasks that can be framed as *search* problems [34]. For example, unit test case generation can naturally be seen as a search problem [5]. There are hundreds of thousands of test cases that could be generated for any particular class under test (CUT). Given a well-defined testing goal, and a numeric scoring function denoting *closeness to the attainment of that goal*—called a *fitness function*—optimization algorithms can systematically search the space of possible test inputs to locate those that meet that goal [45].

The effective use of search-based generation relies on the performance of two tasks—selecting a measurable test goal and selecting an effective fitness function for meeting that goal. Adequacy criteria offer checklists of measurable test goals based on the program source code, such as the execution of branches in the control-flow of the CUT [40, 55, 56]. Because such criteria are based on source code elements, we refer to them as "white-box" test selection criteria. Often, however, goals such as "coverage of branches" are an approximation of a goal that is harder to quantify—we really want tests that will reveal faults [2]. "Finding faults" is not a goal that can be measured, and cannot be translated into a distance function.

To generate effective tests, we must identify criteria—and corresponding fitness functions—that are correlated with an increased probability of fault detection. If branch coverage is, in fact, correlated with fault detection, then—even if we do not care about the concept of branch coverage itself—we will end up with effective tests. However, the need to rely on approximations leads to two questions. First, *can common fitness functions produce effective tests?* If so, *which of the many available fitness functions should be used to generate tests?* Unfortunately, testers are faced with a bewildering number of options—an informal survey of two years of testing literature reveals 28 viable white-box fitness functions—and there is little guidance on when to use one criterion over another [29].

While previous studies on the effectiveness of adequacy criteria in test generation have yielded inconclusive results [52, 48, 36, 29], two factors allow us to more deeply examine this problem—particularly with respect to search-based generation. First, tools are now available that implement enough fitness functions to make unbiased comparisons. The EvoSuite framework offers over twenty options, and uses a combination of eight fitness functions as its default configuration [20]. Second, more realistic examples are available for use in assessment of suites. Much of the previous work on adequacy effectiveness has been assessed using mutants—synthetic faults created through source code transformation [38]. Whether mutants correspond to the types of faults found in real projects has not been firmly established [31]. However, the Defects4J project offers a large database of real faults extracted from open-source Java projects [39]. We can use these faults to assess the effectiveness of search-based generation on the complex faults found in real software.

In this study, we have used EvoSuite and eight of its white-box fitness functions (as well as the default multi-objective configuration and a combination of branch, exception, and method coverage) to generate test suites for the fifteen systems, and 593 of the faults, in the Defects4J database. In each case, we seek to understand *when and why* generated test suites were able to detect—or not detect—faults. Such understanding could lead to a deeper understanding of the strengths and limitations of current test generation techniques, and could inspire new approaches. Thus, in each case, we have recorded the proportion of suites that detect the fault and a number of factors—related to suite size, obligation satisfaction, and attained coverage. We have recorded a set of traditional *source code metrics*—sixty metrics related to cloning, complexity, cohesion, coupling, documentation, inheritance, and size metrics—for each class associated with a fault the Defects4J dataset. By analyzing these generation factors and metrics, we can begin to understand not only the real-world applicability of the fitness options in EvoSuite, but—through the use of machine learning algorithms—the factors correlating with a high or low likelihood of fault detection. To summarize our findings:

- Collectively, 51.26% of the examined faults were detected by generated test suites.
- Branch coverage is the most effective criterion—detecting more faults than any other single criterion and demonstrating a higher likelihood of detection for each fault than other criteria (on average, a 22.60-25.24% likelihood of detection, depending on the search budget).
- Regardless of overall performance, most criteria have situational applicability, where their suites detect faults no other criteria can detect. Exception, output, and weak mutation coverage—in particular—seem to be effective for particular types of faults, even if their average efficacy is low.
- While EvoSuite's default combination performs well, the difficulty of simultaneously balancing eight functions prevents it from outperforming all individual criteria.
- However, a combination of branch, exception, and method coverage has an average 24.03-27.84% likelihood of fault detection—outperforming each of the individual criteria. It is more effective than the default eight-way combination because it adds lightweight situationally-applicable criteria to a strong, coverage-focused criterion.
- Factors that strongly indicate a high level of efficacy include high line or branch coverage over either version of the code and high coverage of their own test obligations.
- Coverage does not ensure success, but it is a prerequisite. In situations where achieved coverage is low, the fault does not tend to be found.
- The most important factor differentiating cases where a fault is occasionally detected and cases where a fault is consistently detected is satisfaction of the chosen criterion's test obligations. Therefore, the best suites are ones that both explore the code and fulfill their own goals, which may be—in cases such as exception coverage—orthogonal to code coverage.
- Test generation methods struggle with classes that have a large number of private methods or attributes, and thrive when a large portion of the class structure is accessible.
- Generated suites are more effective at detecting faults in well-documented classes. While the presence of documentation should not directly assist automated test generation, its presence may hint at the maturity, testability, and understandability of the class.
- Faults in classes with a large number of dependencies are more difficult to detect than those in self-contained classes, as the generation technique must initialize and manipulate multiple complex objects during generation.

Theories learned from the collected metrics suggest that successful criteria thoroughly explore and exploit the code being tested. The strongest fitness functions—branch, direct branch, and line coverage—all do so. We suggest the use of such criteria as *primary* fitness functions. However, our findings also indicate that coverage does not guarantee success. The fitness function must still execute the code in a manner that triggers the fault, and ensures that it manifests in a failure. Criteria such as exception, output, and weak mutation coverage are situationally useful, and should be applied as *secondary* testing goals to boost the fault-detection capabilities of the primary criterion—either as part of a multi-objective approach or through the generation of a separate test suite.

This work extends a prior conference publication [24]. The earlier paper looked at the same core research questions. However, in order to undergo a deeper investigation into the topic, we have contributed an additional 240 faults, from fifteen new systems, to Defects4J—almost doubling the size of the database. Our updated study includes suites generated over those new case examples, adding further observations and points of discussion. We have also used the findings of our separate research into combinations of fitness functions [25] to reformulate and extend our experiments and discussion of the effects of combining criteria. In addition, we have changed how we build and classify data in our treatment learning analysis, added the source code metric analysis, and have included a far deeper examination of the factors indicating success or lack thereof in test generation. Our observations provide evidence for the anecdotal findings of other researchers [8, 24, 23, 60, 7] and motivate improvements in how test generation techniques understand the behavior of private methods or manipulate environmental dependencies. While more research is still needed to better understand the factors that contribute to fault detection, and the joint relationship between the fitness function, generation algorithm, and CUT in determining the efficacy of test suites, our findings in this revised and extended case study offer lessons in understanding the use, applicability, and combination of common fitness functions.

2 | BACKGROUND

2.1 | Search-Based Software Test Generation

Test case creation can naturally be seen as a search problem [34]. Of the thousands of test cases that could be generated for any SUT, we want to select—systematically and at a reasonable cost—those that meet our goals [45, 2]. Given a well-defined testing goal, and a scoring function denoting *closeness to the attainment of that goal*—called a *fitness function*—optimization algorithms can sample from a large and complex set of options as guided by a chosen strategy (the *metaheuristic*) [10]. Metaheuristics are often inspired by natural phenomena, such as swarm behavior [15] or evolution [35].

While the particular details vary between algorithms, the general process employed by a metaheuristic is as follows: (1) One or more solutions are generated, (2), The solutions are scored according to the fitness function, and (3), this score is used to reformulate the solutions for the next round of evolution. This process continues over multiple generations, ultimately returning the best-seen solutions. By determining how solutions are evolved and selected over time, the choice of metaheuristic impacts the quality and efficiency of the search process [16].

Due to the non-linear nature of software, resulting from branching control structures, the search space of a real-world program is large and complex [2]. Metaheuristic search—by strategically sampling from that space—can scale to larger problems than many other generation algorithms [43]. Such approaches have been applied to a wide variety of testing goals and scenarios [2].

2.2 | Adequacy Metrics and Fitness Functions

When testing, developers must judge: (a) whether the produced tests are effective and (b) when they can stop writing additional tests. These two factors are linked. If existing tests have not surfaced any faults, is the software correct, or are the tests *inadequate*? The same question applies when adding new tests—if we have not observed new faults, have we not yet written *adequate* tests?

The concept of adequacy provides developers with the guidance needed to test effectively. As we cannot know what faults exist without verification, and as testing cannot—except in simple cases—conclusively prove the absence of faults, a suitable approximation must be used to measure the adequacy of tests. The most common methods of measuring adequacy involve coverage of structural elements of the software, such as individual statements, branches of the software's control flow, and complex boolean conditional statements [40, 55, 56].

Each adequacy criterion embodies a set of lessons about effective testing—requirements tests must fulfill to be considered adequate. These requirements are expressed as a series of *test obligations*—properties that must be met by the corresponding test suite. For example, the branch coverage criterion requires that each program expression that can cause control flow to diverge—i.e., loop conditions, switch statements, and if-conditions—evaluate to each possible outcome. In this case, a test obligation would indicate a particular expression and a targeted outcome for the evaluation of that expression. If tests fulfill the list of obligations prescribed by the criterion, than testing is deemed "adequate" with respect to faults that manifest through the structures of interest to the criterion.

Adequacy criteria have seen widespread use in software development, and is routinely measured as part of automated build processes [32][1]. It is easy to understand the popularity of adequacy criteria. They offer clear checklists of testing goals that can be objectively evaluated and automatically measured [57]. These very same qualities make adequacy criteria ideal for use as automated test generation targets. In search-based testing, the fitness function needs to capture the testing objective and guide the search. Through this guidance, the fitness function has a major impact on

[1] For example, see https://codecov.io/.

the quality of the solutions generated. Functions must be efficient to execute, as they will be calculated thousands of times over a search. Yet, they also must provide enough detail to differentiate candidate solutions and guide the selection of optimal candidates. Adequacy criteria are ideal optimization targets for automated test case generation as they can be straightforwardly transformed into efficient, informative fitness functions [6]. Search-based generation often can achieve higher coverage than developer-created tests [22].

3 | STUDY

To generate unit tests that are effective at finding faults, we must identify criteria and corresponding fitness functions that increase the probability of fault detection. As we cannot know what faults exist before verification, such criteria are approximations—intended to increase the probability of fault detection, but offering no guarantees. Thus, it is important to turn a critical eye toward the choice of fitness function used in search-based test generation. We wish to know whether commonly-used fitness functions produce effective tests, and if so, why—and under what circumstances—do they do so?

More empirical evidence is needed to better understand the relationships between adequacy criteria, fitness functions and fault detection [32]. Many criteria exist, and there is little guidance on when to use one over another [29]. To better understand the real-world effectiveness, use, and applicability of common fitness functions and the factors leading to a higher probability of fault detection, we have assessed the EvoSuite test generation framework and eight of its fitness functions (as well as the default multi-objective configuration) against 593 real faults, contained in the Defects4J database. In doing so, we wish to address the following research questions:

1. How capable are generated test suites at detecting real faults?
2. Which fitness functions have the highest likelihood of fault detection?
3. Does an increased search budget improve the effectiveness of the resulting test suites?
4. Under what situations can a combination of criteria outperform a single criterion?
5. What factors correlate with a high likelihood of fault detection?

The first three questions allow us to establish a basic understanding of the effectiveness of each fitness function—are *any* of the functions able to generate fault-detecting tests and, if so, are any of these functions more effective than others at the task? However, these questions presuppose that only one fitness function can be used to generate test suites. Many search-based generation algorithms can simultaneously target *multiple* fitness functions [25]. Therefore, we also ask question 4—when does it make sense to employ a set of fitness functions instead of a single function?

Finally, across all criteria, we also would like to gain insight into the factors that influence the likelihood of detection. To inspire new research advances, we desired a deeper understanding of when generated suites are likely to detect a fault, and when they will fail. We have made use of *treatment learning*—a machine learning technique designed to take classified data and identify sets of attributes, along with bounded values of such attributes, that are highly correlated with particular outcomes. In our case, these “outcomes” are associated with whether generated suites from each fitness function detect a fault or not. We have gathered factors from two broad sets:

- **Test Generation Factors** are related to the test suites produced—identifying coverage attained, suite size, and obligation satisfaction.
- **Source Code Metrics** examine the faulty classes being targeted, and ask whether factors related to the classes themselves—i.e., the number of private methods or cloned code—can impact the test generation process.

We have created datasets based off of both sets of factors and applied treatment learning to assess which factors strongly affected the outcome of test generation. We make these datasets, as well as the new Defects4J case examples, available to other researchers to aid in future advances (see Sections 3.1 and 3.5.3).

In order to investigate these questions, we have performed the following experiment:

1. **Collected Case Examples:** We have used 353 real faults, from five Java projects, as test generation targets (Section 3.1).
2. **Developed New Case Examples:** We have also mined an additional 240 faults from ten new projects, and added these faults to the Defects4J database (Section 3.1).
3. **Recorded Source Code Metrics:** For each affected class (both faulty and fixed versions), we measure a series of sixty source code metrics, related to cloning, cohesion, coupling, documentation, inheritance, and class size (Section 3.2).
4. **Generated Test Suites:** For each fault, we generated 10 suites per criterion using the fixed version of each CUT. We performed with both a two-minute and a ten-minute search budget per CUT (Section 3.3).
5. **Generated Test Suites for Combinations of Criteria:** We perform the same process for EvoSuite’s default configuration—a combination of eight criteria—and a combination of branch, exception, and method coverage (Section 3.3).

6. **Removed Non-Compiling and Flaky Tests:** Any tests that do not compile, or that return inconsistent results, are removed (Section 3.3).
7. **Assessed Fault-finding Effectiveness:** We measure the proportion of test suites that detect each fault to the number generated (Section 3.4).
8. **Recorded Generation Statistics:** For each suite, fault, and budget, we measure factors that may influence suite effectiveness, related to coverage, suite size, and obligation satisfaction (Section 3.4).
9. **Prepare Datasets**: Datasets were prepared for learning purposes by by adding classifications based on fault detection to each entry in the generation factor and code metric datasets. Separate datasets were prepared for each generation budget and fitness function, as well as sets based on overall fault detection (across all fitness functions and function combinations) for each budget (Section 3.5).
10. **Performed Treatment Learning:** We apply the TAR3 learner to identify factors correlated to each classification for each dataset (Section 3.5).

3.1 | Case Examples

Defects4J is an extensible database of real faults extracted from Java projects [39][2]. The original dataset consisted of 357 faults from five projects: Chart (26 faults), Closure (133 faults), Lang (65 faults), Math (106 faults), and Time (27 faults). For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose the fault, and a list of classes and lines of code modified by the patch that fixes the fault.

Each fault is required to meet three properties. First, a pair of code versions must exist that differ only by the minimum changes required to address the fault. The “fixed” version must be explicitly labeled as a fix to an issue, and changes imposed by the fix must be to source code, not to other project artifacts such as the build system. Second, the fault must be reproducible—at least one test must pass on the fixed version and fail on the faulty version. Third, the fix must be isolated from unrelated code changes such as refactorings.

In order to expand our study to a larger set of case examples, we have added an additional ten systems to Defects4J. The process of adding new faults is semi-automated, and requires the development of build files that work across project versions. The commit messages of the project’s version control system are scanned for references to issue identifiers. These versions are considered to be candidate “fixes” to the referenced issues. The human-developed test suite for that version is then applied to previous project versions. If one or more test cases pass on the “fixed” version and fail on the earlier version, then that version is retained as the “faulty” variant. The code differences between versions are captured as a patch file, which must then be manually minimized to remove any differences that are not required to reproduce the fault.

Following this process, we added 240 faults—bringing the total to 597 faults from 15 projects. The new projects include: CommonsCLI (24 faults), CommonsCSV (12), CommonsCodec (22), CommonsJXPath (14), Guava (9), JacksonCore (13), JacksonDatabind (39), JacksonXML (5), Jsoup (64), and Mockito (38)[3]. The ten new systems were chosen because they are popular projects and have reached a reasonable level of maturity—meaning that the detected faults are often relatively complicated. Two of these systems, Guava and Mockito, were the subjects of recent research challenges at the Symposium on Search-Based Software Engineering [23, 3]. Four faults from the Math project were omitted due to complications encountered during suite generation, leaving 593 faults that we used in our study.

3.2 | Code Metric-based Characterization

When assessing the results of our study, we wish to gain understanding of *when and why* our test suites were able to detect—or not detect—faults. Such understanding could lead to a deeper understanding of the strengths and limitations of current test generation techniques, and could inspire new approaches. Gaining such understanding requires fine-grained information about the faults being targeted—and, more specifically, the classes being targeted for test generation. To assist in gaining this understanding, we have turned to traditional *source code metrics*. Using the SourceMeter framework[4], we have gathered a set of 60 cloning, complexity, cohesion, coupling, documentation, inheritance, and size metrics for each class associated with a fault included in the Defects4J dataset.

Such metrics, commonly used as part of research on effort estimation [50] and defect prediction [64], are considered to have substantial predictive power. By characterizing the classes that host the Defects4J faults using these code metrics, we can better understand the results of our research. Using the SourceMeter framework, we have measured 60 source code metrics for each class related to a fault in the Defects4J dataset. These metrics are recorded for both the faulty and fixed versions of each affected class. The metrics may be divided into the following categories:

- **Clone Metrics** are used to measure the occurrence of Type-2 clones in the class—code fragments that are structurally identical, but may differ in variable names, literals, and identifiers [59].

[2]Available from http://defects4j.org
[3]The new faults have been submitted to Defects4J as a pull request. Until they are accepted, they can be found at the *additional-faults-1.4* branch of https://github.com/Greg4cr/defects4j.
[4]Available from https://www.sourcemeter.com.

Category	Abbreviation	Metric	Median	Standard Deviation
Clone	CC	Clone Coverage	0.00	0.16
	CCL	Clone Classes	0.00	10.80
	CCO	Clone Complexity	0.00	453.26
	CI	Clone Instances	0.00	27.86
	CLC	Clone Line Coverage	0.00	0.09
	CLLC	Clone Logical Line Coverage	0.00	0.14
	LDC	Lines of Duplicated Code	0.00	161.06
	LLDC	Logical Lines of Duplicated Code	0.00	147.78
Cohesion	LCOM5	Lack of Cohesion in Methods 5	1.00	7.31
Complexity	NL	Nesting Level	4.00	2.94
	NLE	Nesting Level Else-If	3.00	1.93
	WMC	Weighted Methods per Class	55.00	149.58
Coupling	CBO	Coupling Between Object Classes	8.00	11.79
	CBOI	Coupling Between Object Classes Inverse	5.00	61.38
	NII	Number of Incoming Invocations	16.00	172.67
	NOI	Number of Outgoing Invocations	14.00	31.31
	RFC	Response Set For Class	37.50	56.43
Documentation	AD	API Documentation	1.00	0.32
	CD	Comment Density	0.38	0.18
	CLOC	Comment Lines of Code	127.00	460.47
	DLOC	Documentation Lines of Code	102.50	443.12
	PDA	Public Documented API	7.00	28.31
	PUA	Public Undocumented API	0.00	8.77
	TCD	Total Comment Density	0.36	0.17
	TCLOC	Total Comment Lines of Code	144.50	465.29
Inheritance	DIT	Depth of Inheritance Tree	1.00	1.15
	NOA	Number of Ancestors	1.00	1.71
	NOC	Number of Children	0.00	2.07
	NOD	Number of Descendants	0.00	3.39
	NOP	Number of Parents	1.00	0.86
Size	LLOC	Logical Lines of Code	208.50	462.19
	LOC	Lines of Code	382.50	879.00
	NA	Number of Attributes	8.00	14.33
	NG	Number of Getters	3.00	14.96
	NLA	Number of Local Attributes	6.00	10.24
	NLG	Number of Local Getters	2.00	8.58
	NLM	Number of Local Methods	21.00	35.03
	NLPA	Number of Local Public Attributes	0.00	4.35
	NLPM	Number of Local Public Methods	9.00	29.50
	NLS	Number of Local Setters	0.00	5.23
	NM	Number of Methods	28.50	50.96
	NOS	Number of Statements	109.50	279.94
	NPA	Number of Public Attributes	0.00	6.17
	NPM	Number of Public Methods	14.00	44.25
	NS	Number of Setters	0.00	11.33
	TLLOC	Total Logical Lines of Code	275.50	503.28
	TLOC	Total Lines of Code	495.00	919.51
	TNA	Total Number of Attributes	10.00	17.34
	TNG	Total Number of Getters	4.00	20.96
	TNLA	Total Number of Local Attributes	8.00	14.20
	TNLG	Total Number of Local Getters	2.00	10.62
	TNLM	Total Number of Local Methods	27.00	44.33
	TNLPA	Total Number of Local Public Attributes	0.00	4.67
	TNLPM	Total Number of Local Public Methods	12.00	34.82
	TNLS	Total Number of Local Setters	0.00	6.20
	TNM	Total Number of Methods	37.00	79.02
	TNOS	Total Number of Statements	143.00	293.37
	TNPA	Total Number of Public Attributes	0.00	6.36
	TNPM	Total Number of Public Methods	19.00	62.78
	TNS	Total Number of Setters	0.00	14.01

TABLE 1 List of metrics gathered for each class, separated by category. The median and standard deviation are listed for each.

- **Cohesion Metrics** assess the level of cohesion in a class—whether attributes and the operations that use them are organized into one class, and whether there are methods that are unrelated to the other methods and attributes in the class [33].
- **Complexity Metrics** assess the complexity of the class, using information such as the depth of nesting and the number of control-flow paths through methods [13]. Complexity is often used as part of defect prediction, as more complex methods are expected to contain more defects than simpler methods.

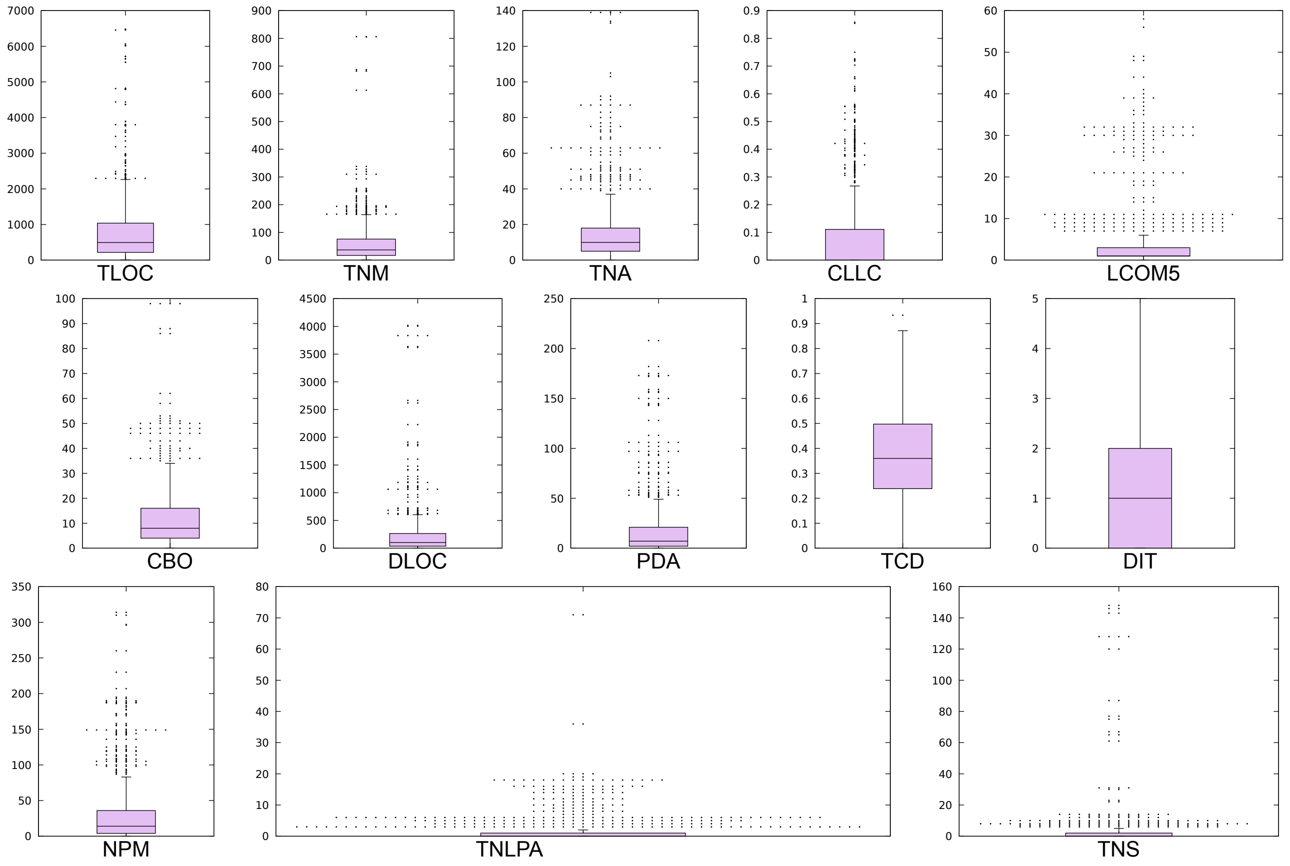


FIGURE 1 Boxplots illustrating the median, first, and third quartile values for select metrics from the dataset.

- **Coupling Metrics** assess the level of dependency between classes [9]. When designing a system, developers are cautioned to minimize the level of coupling—to make each class as independent as possible. Coupling metrics are used to identify design weaknesses and architectural bottlenecks.
- **Documentation Metrics** measure the degree that a class is documented by its developers [42]. Well-documented code is often thought to be higher quality code, and these metrics can identify classes that may have received less attention.
- **Inheritance Metrics** tracks the relationships between parent and child classes along the class hierarchy [9]. As inheritance defines a form of dependence, such metrics are useful for identifying how code changes can propagate through a system.
- **Size Metrics** characterize the size and complexity of a class based on structural elements such as the number of lines of code, methods, attributes, setters, and getters [1]. Such metrics can be used, at a glance, to identify some of the more complex classes in a system.

Table 1 lists the gathered metrics. Detailed definitions may be found on the SourceMeter documentation [63]. Table 1 notes the median and standard deviation for all metrics.

3.2.1 | Characterizing the "Average" Class

To help illustrate the "average" class from Defects4J, we have included boxplots for several of the measured metrics in Figure 1. Each box depicts the first and third quartiles, as well as the median value. Outliers—points more than 1.5 times the interquartile range—are depicted as well. Rather than depict all 60 metrics, we show a subset indicated as important in our case study to help characterize the studied classes. First, to set context:

- **TLOC (Total Lines of Code)** indicates the amount of code is in a class, including comments and whitespace. TLOC include lines in anonymous, nested, and local classes. The median TLOC is 495, but a large standard deviation (919.51) indicates that classes have a wide range of sizes. Studied classes tend to cluster between 0-1,000 TLOC, and the largest class has 6,481 TLOC.

- **TNM (Total Number of Methods)** is the number of methods in a class, including those in anonymous, nested, and local classes, as well as those inherited from a parent. The median TNM is 37, with the maximum being 806. Classes in Defects4J tend to have less than 100 methods. Again, however, there are a number of outliers.
- **TNA (Total Number of Attributes)** is the number of attributes in a class, including those in anonymous, nested, and local classes, as well as those inherited from a parent. The median TNA is 10—with values tending to cluster between 5-20—and the maximum is 139.

The following metrics were indicated in our case study as being able to explain why generated test suites are able—or not able—to detect faults. We will discuss the implications of these findings in Section 4.5. Here, we use these metrics to further characterize the "average" class in Defects4J.

- **CLLC (Clone Logical Line Coverage)** is the ratio of code covered by code duplications in the class to the size of the class, expressed in terms of logical lines of code (non-empty, non-comment lines). **Clone Coverage** is the same measurement, except that it includes comments and whitespace. The CLLC and CC are both largely concentrated towards the low end of the scale, with a median of 0 for both—no code being duplicated—and a relatively low standard deviation (0.14).
- **LCOM5 (Lack of Cohesion in Methods 5)** measures the lack of cohesion and computes how many coherent classes the class could be split into. Although there are many outliers, the LCOM5 is largely concentrated towards the low end of the scale—with a median of 1—indicating that classes tend to be highly cohesive.
- **CBO (Coupling Between Objects)** indicates the number of classes that serve as dependencies of the target class (by inheritance, method call, type reference, or attribute reference). Classes dependent on many other classes are very sensitive to the changes in the system, and can be harder to test or evolve. The median CBO is 8 and standard deviation is 11.79, indicating that—while classes are connected—the average class does not overly depend on the rest of the system.
- **DLOC (Documented Lines of Code)** simply measures the number of line of code that are comments (in-line or standalone). The median number of documented lines is 102. However, there are a large range of values, with the maximum DLOC at a staggering 4,017.
- **PDA (Public Documented API)** is the number of public methods with documentation. The median PDA is 7. Again, however, there is a large variance in PDA, with a standard deviation of 28.31 and a large number of outlying values.
- **TCD (Total Comment Density)** is the ratio of the comment lines to the sum of its comment and logical lines of code, including nested, anonymous, and local classes. The **CD (Comment Density)**, also noted as important, is the same measurement excluding nested, anonymous, and local classes. The median TCD is 0.36 and the median CD is 0.38, indicating that the "average" Defects4J class is approximately one-third documentation.
- **DIT (Depth of Inheritance Tree)** measures the length of the path from the class to its furthest ancestor in the inheritance tree. The median DIT is 1—many classes have a parent. The maximum DIT is 5, but this is an extreme outlier—the standard deviation is 1.15 and almost all classes have 0-2 levels of ancestors.
- **NPM (Number of Public Methods)** indicates the number of methods that are publicly-accessible in the class, including inherited methods. Closely related is the **TNLPM (Total Number of Local Public Methods)**, which includes nested, anonymous, and local classes, but excludes inherited methods. The median NPM is 14, with the majority concentrated below 50. The median TNLPM is slightly lower—12. Both are lower than the TNM, indicating that many classes have a large number of private methods.
- **NLPA (Number of Local Public Attributes)**, similarly, indicates the number of publicly-accessible attributes in the class, excluding inherited attributes. The **TNLPA (Total Number of Local Public Attributes)** includes nested, anonymous, and local classes. Both metrics are less consistent than many of the others, with a large number of outlying values. However, the median for both is 0—indicating that there are often no public attributes in a class. This is notably lower than the TNA, indicating that class attributes are generally private.
- **TNS (Total Number of Setters)** records the number of setter methods in the class, including inherited methods. Our study also indicated the **NS (Number of Setters)**—excluding nested, anonymous, and local classes—and **TNLS (Total Number of Local Setters)**—excluding inherited setters—as important. This set of attributes also has a relatively large number of outliers, but is concentrated towards the low end of the scale. All three of these metrics have a median value of 0, indicating that most classes in Defects4J have no setter methods.

3.3 | Test Suite Generation

The EvoSuite framework uses a genetic algorithm to evolve test suites over a series of generations, forming a new population by retaining, mutating, and combining the strongest solutions. It is actively maintained and has been successfully applied to a variety of projects [60]. In this study, we used EvoSuite version 1.0.5, and the following fitness functions:

Branch Coverage (BC): A test suite satisfies branch coverage if all control-flow branches are taken during test execution—the test suite contains at least one test whose execution evaluates the branch predicate to `true`, and at least one whose execution evaluates the predicate to `false`. To guide the search, the fitness function calculates the *branch distance* from the point where the execution path diverged from the targeted branch. If an undesired branch is taken, the function describes how "close" the targeted predicate is to being true, using a cost function based on the predicate formula [6].

Direct Branch Coverage (DBC): Branch coverage may be attained by calling a method *directly*, or *indirectly*—calling a method within another method. When a test covers a branch indirectly, it can be more difficult to understand how coverage was attained. Direct branch coverage requires each branch to be covered through a direct method call.

Line Coverage (LC): A test suite satisfies line coverage if it executes each non-comment source code line at least once. To cover each line of source code, EvoSuite tries to ensure that each basic code block is reached. The branch distance is computed for each branch that is a control dependency of any of the statements in the CUT. For each conditional statement that is a control dependency for some other line in the code, EvoSuite requires that the branch of the statement leading to the dependent code is executed.

Exception Coverage (EC): The goal of exception coverage is to build test suites that force the CUT to throw exceptions—either declared or undeclared. As the number of possible exceptions that a class can throw cannot be known ahead of time, the fitness function rewards suites that throw more exceptions. As this function is based on the number of discovered exceptions, the number of "test obligations" may change each time EvoSuite is executed on a CUT.

Method Coverage (MC): Method Coverage simply requires that all methods in the CUT are executed at least once, through direct or indirect calls. The fitness function for method coverage is discrete, as a method is either called or not called.

Method Coverage (Top-Level, No Exception) (MNEC): Generated test suites sometimes achieve high levels of method coverage by calling methods in an invalid state or with invalid parameters. MNEC requires that all methods be called directly and terminate without throwing an exception.

Output Coverage (OC): Output coverage rewards diversity in the method output by mapping return types to a list of abstract values [4]. A test suite satisfies output coverage if, for each public method in the CUT, at least one test yields a concrete return value characterized by each abstract value. For numeric data types, distance functions offer feedback using the difference between the chosen value and target abstract values.

Weak Mutation Coverage (WMC): Test effectiveness is often judged using mutants [38]. Suites that detect more mutants may be effective at detecting real faults as well. A test suite satisfies weak mutation coverage if, for each mutated statement, at least one test detects the mutation. The search is guided by the *infection distance*, a variant of branch distance tuned towards reaching and discovering mutated statements [21].

Rojas et al. provide a primer on each of these fitness functions [58]. In order to study the effect of combining fitness functions, we also generate test suites using two combinations. The first is EvoSuite's default configuration—a combination of all of the above methods (called the **"Default Combination"**). The second is a combination of branch, exception, and method coverage (called the **"BC-EC-MC Combination"**). This combination was identified as an effective baseline in our prior work studying combination efficacy on the five original systems from Defects4J [25].

Test suites are generated that target the classes reported as relevant to the fault by Defects4J. Tests are generated using the fixed version of the CUT and applied to the faulty version in order to eliminate the oracle problem. EvoSuite generates assertion-based oracles. Generating oracles based on the fixed version of the class means that we can confirm that the fault is actually detected, and not just that there are coincidental differences in program output. In practice, this translates to a regression testing scenario, where tests are generated using a version of the system understood to be "correct" in order to guard against future issues. Tests that fail on the faulty version, then, detect behavioral differences between the two versions.

Two search budgets were used—two minutes and ten minutes per class. This allows us to examine whether an increased search budget benefits each fitness function. To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. All other settings were kept at their default values. As results may vary, we performed 10 trials for each fault and search budget. This resulted in the generation of 118,600 test suites (two budgets, ten trials, ten configurations, 593 faults).

Generation tools may generate flaky (unstable) tests [60]. For example, a test case that makes assertions about the system time will only pass during generation. We automatically remove flaky tests. First, all non-compiling test suites are removed. Then, each remaining test suite is executed on the fixed version five times. If the test results are inconsistent, the test case is removed. This process is repeated until all tests pass five times in a row. On average, fewer than one test tends to be removed from each suite (see Table 2).

3.4 | Test Generation Data Collection

To evaluate the fault-finding effectiveness of the generated test suites, we execute each test suite against the faulty version of each CUT. The **effectiveness** of each fitness function, for each fault, is the proportion of suites that successfully detect the fault to the total number of suites generated for that fault. We refer to this as the likelihood of fault detection.

To better understand the generation factors that influence effectiveness, we also collected the following for each test suite:

Method	Budget	Total Obligations	% Obligations Satisfied	Suite Size	Suite Length	# Tests Removed	% Line Coverage (Fixed)	% Line Coverage (Faulty)	% Branch Coverage (Fixed)	% Branch Coverage (Faulty)
Branch Coverage (BC)	120	295.15	58.32%	36.33	195.96	0.38	61.98%	62.04%	58.96%	58.87%
	600		65.00%	44.27	268.95	0.75	67.35%	67.23%	65.44%	65.19%
Direct Branch (DBC)	120	295.15	54.60%	38.32	217.55	0.35	60.01%	59.59%	56.37%	55.93%
	600		61.79%	48.27	310.52	0.73	65.53%	64.96%	63.32%	62.65%
Exception Coverage (EC)	120	12.47	99.41%	11.99	35.54	0.23	21.35%	21.36%	15.82%	15.99%
	600	12.57	99.38%	12.12	36.09	0.26	21.60%	21.63%	15.97%	16.15%
Line Coverage (LC)	120	329.90	61.29%	30.32	162.08	0.28	62.27%	61.69%	53.60%	53.09%
	600		66.79%	34.73	207.65	0.45	67.53%	66.90%	59.11%	58.51%
Method Coverage (MC)	120	31.92	78.99%	22.00	73.78	0.05	37.51%	37.40%	29.18%	29.26%
	600		83.30%	24.32	86.62	0.09	38.91%	38.93%	30.36%	30.52%
Method, No Exception (MNEC)	120	31.92	77.59%	21.68	74.54	0.05	39.06%	38.99%	30.39%	30.48%
	600		82.19%	23.79	88.88	0.10	40.84%	40.78%	31.75%	31.91%
Output Coverage (OC)	120	185.89	42.78%	29.04	133.04	0.12	39.00%	38.82%	32.90%	32.88%
	600		46.17%	32.68	161.32	0.26	40.81%	40.67%	34.47%	34.47%
Weak Mutation (WMC)	120	508.38	56.20%	26.48	164.51	0.16	56.02%	55.87%	49.73%	49.50%
	600		62.94%	32.60	246.57	0.42	61.58%	61.31%	56.19%	55.80%
Default Combination	120	1673.19	53.92%	48.71	358.93	0.55	58.25%	58.05%	53.15%	52.95%
	600	1681.19	60.72%	64.57	550.03	1.08	64.65%	64.07%	60.91%	60.30%
BC-EC-MC Combination	120	345.59	63.81%	50.73	262.03	1.06	62.91%	62.22%	59.95%	59.12%
	600	354.84	69.69%	65.10	368.83	2.04	68.00%	67.12%	66.30%	65.24%

TABLE 2 Statistics on generated test suites (each statistic is explained in Section 3.4). Values are averaged over all faults (i.e., the average number of obligations, average number of tests removed, etc.). "Default Combination" is a combination of the eight individual fitness functions. "BC-EC-MC Combination" combines the branch, exception, and method coverage fitness functions.

Number of Test Obligations: Given a CUT, each fitness function will calculate a series of test obligations to cover (as defined in Section 2). The number of obligations is informative of the difficulty of the generation, and impacts the size and formulation of tests [28]. Note that the number of test obligations is dependent on the CUT, and does not differ between budgets except in the case of exception coverage. As exception coverage simply counts the number of observed exceptions, it does not have a consistent set of obligations each time generation is performed.

Percentage of Obligations Satisfied: This factor indicates the ability of a fitness function to cover its goals. A suite that covers 10% of its goals is likely to be less effective than one that achieves 100% coverage.

Test Suite Size: We have recorded the number of tests in each test suite. Larger suites are often thought to be more effective [30, 36]. Even if two suites achieve the same coverage, the larger may be more effective simply because it exercises more combinations of input.

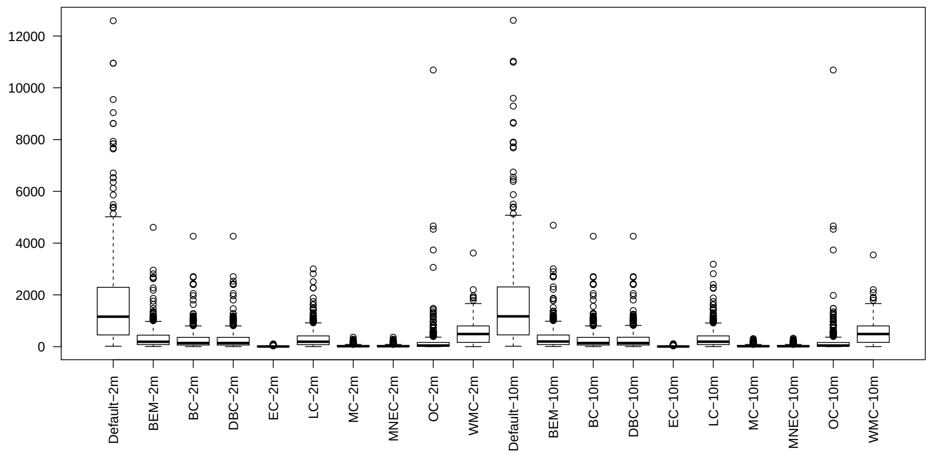
Test Suite Length: Each test consists of one or more method calls. Even if two suites have the same number of tests, one may have much *longer* tests—making more method calls. In assessing the effect of suite size, we must also consider the length of each test case.

Number of Tests Removed: Any tests that do not compile, or that return inconsistent results, are automatically removed. We track the number removed from each suite.

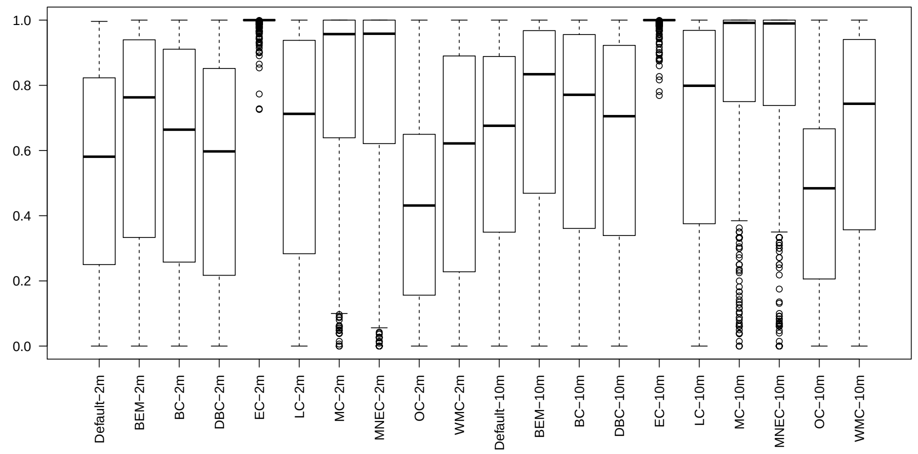
Code Coverage: As the premise of many adequacy criteria is that faults are more likely to be detected if structural elements of the code are thoroughly executed, the resulting coverage of the code may indicate the effectiveness of a test suite. Using EvoSuite's coverage measurement capabilities, we have measured the line and branch coverage achieved by each suite when executed over both the faulty and fixed versions of each CUT. Due to instrumentation issues, we were unable to measure coverage over two systems—JacksonDatabind and Mockito. However, we were able to measure coverage over the remaining 516 faults.

Table 2 records, for each fitness function and budget, the average values attained for each of these measurements over all faults for which we were able to take measurements. In Figure 2, we show boxplots of the total obligations and % of obligations satisfied for suites generated for each fitness configuration and search budget. Branch and direct branch coverage will always have the same number of obligations. Line coverage tends to operate in the same approximate range. Exception, method, and MNEC have the fewest obligations, while weak mutation coverage tends to have the most obligations of the individual fitness functions. Naturally, the two combinations have more obligations—the combination of their member functions. In terms of satisfaction, the three fitness functions with the fewest obligations—exception, method, and MNEC—all also have the highest satisfaction rate. Output coverage has the lowest average, and generally lower, satisfaction rates. For all functions other than Exception Coverage, there tends to be large variance in the satisfaction rate.

In Figure 3, we show boxplots of the suite size, length, and line coverage of suites generated for each fitness configuration and search budget. Most fitness functions yield similar median suite size and variance in results. Exception coverage tends to yield the smallest suites, owing to its small number of test obligations. Method coverage and MNEC yield smaller suites than other fitness functions, but not significantly so. Output coverage



(a) Total Obligations



(b) % Obligations Satisfied

FIGURE 2 Boxplots of the total obligations and % of obligations satisfied for suites generated for each fitness configuration and search budget.

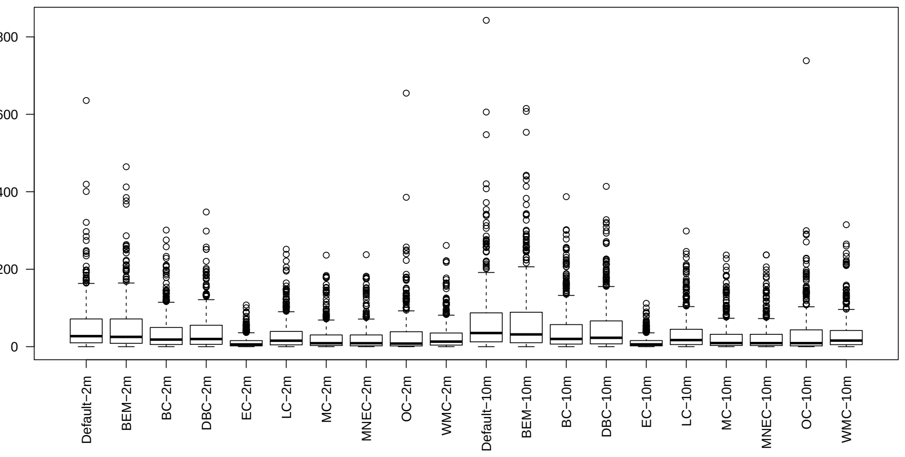
tends to have relatively large test suites—comparable in size to branch and direct branch coverage. Again, the two combinations tend to yield larger test suites, but not significantly larger than those for branch, direct branch, and weak mutation coverage. Test suite length largely offers similar observations. However, we do note that the "default combination" tends to yield very *long* test suites, composed of more method calls than suites for other fitness configurations. This is not the case for the BC-EC-MC combination.

Like with obligation satisfaction, there is a large variance in the line coverage attained by test suites, regardless of fitness function. Exception coverage tends to achieve both the lowest coverage and the least variance in coverage. This is reasonable, as the fitness function for exception coverage has no mechanism to encourage class exploration. Naturally, branch, direct branch, line, and weak mutation coverage tend to attain high coverage rates over classes, as all four use coverage-based fitness mechanisms. Exception, method, MNEC, and output coverage are based on source code elements, but all have fitness representations that are not based on control flow. As a result, they tend to attain lower coverage levels.

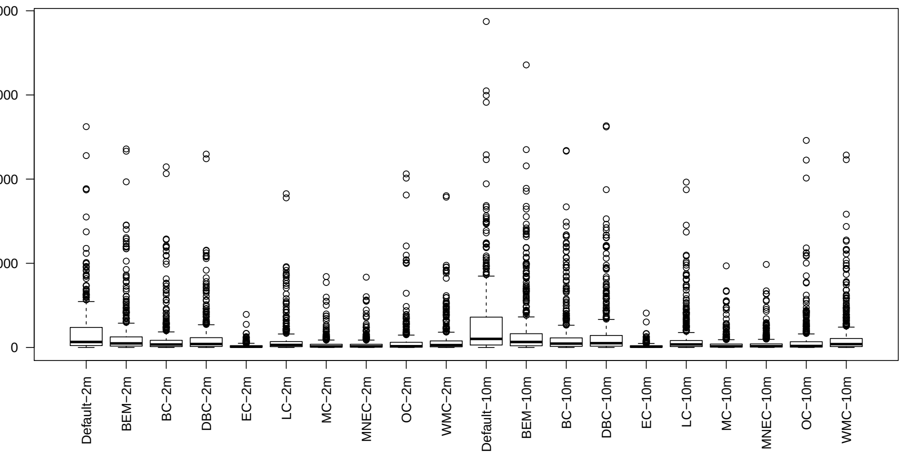
3.5 | Dataset Preparation for Treatment Learning

To understand the factors leading to detection—or lack of detection—of a fault, we have collected two basic sets of data for each fault: **test generation factors** related to the test suites produced and **source code metrics** examining the classes being targeted for unit test generation.

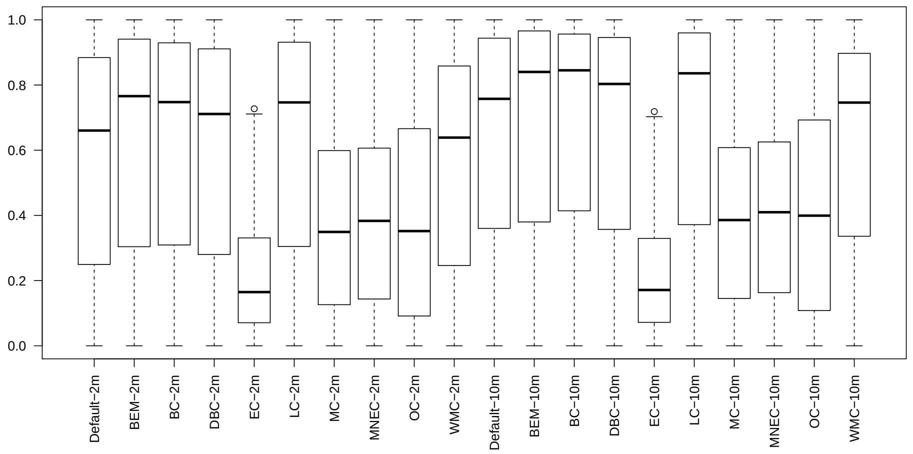
A standard practice in machine learning is to *classify data*—to use previous experience to categorize new observations [49]. We are instead interested in the reverse scenario. Rather than attempting to categorize new data, we want to work backwards from classifications to discover *which*



(a) Suite Size

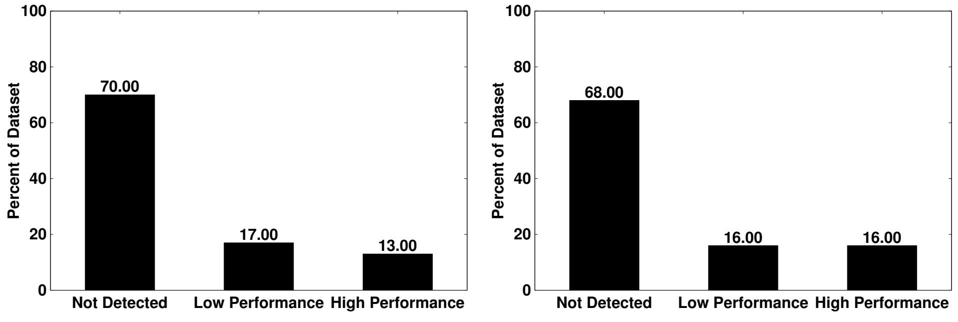


(b) Suite Length



(c) % Line Coverage (Fixed)

FIGURE 3 Boxplots of the suite size, length, and line coverage of suites generated for each fitness configuration and search budget.



(a) 2m Budget (b) 10m Budget

FIGURE 4 Baseline class distribution of the generation factors datasets used for treatment learning.

factors correspond most strongly to a class of interest—a process known as treatment learning [47]. Treatment learning approaches take the classification of an observation and reverse engineers the evidence that led to that categorization. Such learners produce a *treatment*—a small set of attributes and value ranges for each that, if imposed, will identify a subset of the original data skewed towards the target classification. In this case, a treatment notes the metrics—and their values—that indicate that generated test suites will detect a fault.

For example, the treatment $[\text{NS} = [0.00..1.00), \text{TCD} = [0.52..0.93]]$—derived from the code metric-based dataset—indicates that the subset of the data where the Number of Setters is less than 1 and the Total Comment Density is between 52-93% has a higher percentage of "Yes" classifications ("Yes" implying that the fault is detected, while a "No" classification indicates that the fault was not detected) than the base dataset. Within this subset, "Yes" classifications account for 72.00% of the class distribution of the subset—compared to 47.44% of the base distribution.

Using the TAR3 treatment learner [27], we have generated five treatments from each dataset that target each of the applied verdicts. A user can specify the minimum number of examples that may make up the data subset matching a treatment in order to ensure minimum support for a treatment. We require the subset to contain at least 20% of the total dataset. In addition, a limit can be placed on the number of attributes chosen for a treatment. It is thought that large treatments—those recommending more than five attribute-range pairs—may not have more explanatory power than smaller treatments [27]. Therefore, we also limit the treatment size to five attribute-value pairings.

3.5.1 | Generation-Based Datasets

As discussed in Section 3.4—to better understand the combination of factors correlating with effective fault detection—we have collected the following statistics for each generated test suite: the number of obligations, the percent satisfied, suite size, suite length, number of tests removed, as well as branch (BC) and line coverage (LC) over the fixed and faulty versions of the CUT.

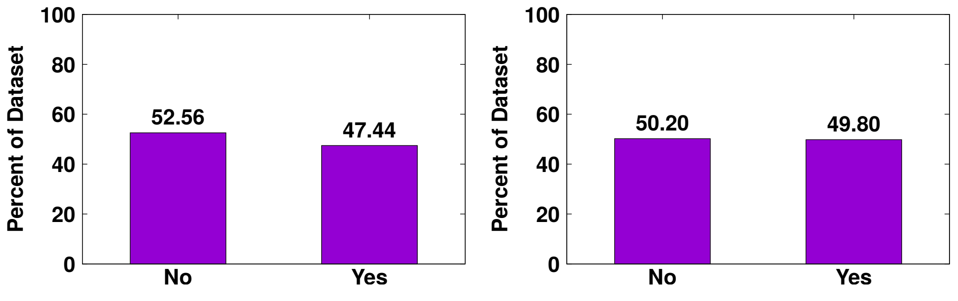
This collection of factors forms a dataset that can be used to analyze the impact of these factors on efficacy. Each record in the dataset corresponds to the values of these attributes for each fault, and for each criterion or combination of criteria. In total, this dataset contains 5160 records (516 faults for which we could collect all statistics, times the ten fitness configurations). We build separate datasets for each search budget. We can then use the likelihood of fault detection (D) as the basis for a class variable—discretized into three values: "not detected" ($\text{D} = 0$), "low performance" ($\text{D} < 70\%$), and "high performance" ($\text{D} \geq 70\%$). The class distribution of each dataset is shown in Figure 4.

3.5.2 | Code Metric-Based Datasets

As with the test generation factors, we have created separate source code metric datasets split by the search budget used for test generation (two minutes per class or ten minutes per class). Source code metrics do not differ based on the fitness function used in test generation. Therefore, rather than merging all fitness configurations into a single dataset, we have produced separate datasets for each fitness function. We also produced "overall" datasets, classified by whether *any* fitness function or combination detected a fault. In total, this process produced a set of 18 datasets for treatment learning: two based on overall results—split by search budget—and two for each of the eight studied coverage criteria.

In order to learn which metrics predict whether or not a fault is detected, we have added classifications to each characterization dataset. In this case, we have used two classification values—*yes* or *no*, based on whether or not the fault was detected by the generated test suites[5] For each fault,

[5]Initially, we used a three-option classification like with the generation factors dataset. However, this did not yield enough examples to yield detailed treatments in many cases. Instead, we elected to use a two-class outcome.



(a) 2m Budget (b) 10m Budget

FIGURE 5 Baseline class distributions for the "overall" code metrics datasets used for treatment learning.

the characterization dataset has an entry for each class fixed as part of patching the fault. We apply a "yes" classification if *any* test suite generated targeting that class, under that search budget, detects the fault. If no test suites detected that fault under that search budget, we apply a "no" classification. The class distributions for the two overall datasets are shown in Figure 5. In both cases, the two classes each make up roughly half of the dataset, with a slight edge to the number of "no" classifications.

3.5.3 | Accessing Datasets

We have made all datasets used in this study openly available as community resources. They can be downloaded from:

https://github.com/Greg4cr/coverage-exp-data

4 | RESULTS & DISCUSSION

In Section 4.1, we will outline the basic fault detection capabilities of the generated test suites. Section 4.2 examines the efficacy of each individual fitness function, while Section 4.3 outlines the effect of combining fitness functions. In Section 4.4, we examine the generation factors contributing to fault detection. Finally, in Section 4.5, we examine the source code metrics that impact detection efficacy.

4.1 | Overall Fault-Detection Capability

In Table 3, we list the number of faults detected by each fitness function, broken down by system and search budget. We also list the number of faults detected by *any criterion*, including and excluding the combinations (which we will discuss further in Section 4.3). Due to the stochastic search, a higher budget does not guarantee detection of the same faults found under a lower search budget. Therefore, we list the number of faults found under either budget, as well as the total number of faults detected by each fitness function. These results offer a baseline for further discussion. In our experiments:

The individual criteria are capable of detecting 304 (51.26%) of the 593 faults. Combinations detect a further 17 faults.

While there is clearly room for improvement, these results are encouraging. The studied faults are actual faults, reported by the users of real-world software projects. The generated tests are able to detect a variety of complex programming issues. Ultimately, our results are consistent with previous studied involving Defects4J—for instance, Shamshiri et al. found that a combination of test generation tools—including suites generated using EvoSuite's branch fitness function—could detect 55.7% of the faults from the original five systems from Defects4J [60].

Shamshiri's work—as well as our studies on Guava [3] and Mockito [23]—offer explanation of the broad reasons why test generation fails to detect particular faults. Some of these reasons include a general inability to gain coverage—particularly over private methods—challenges with initialization of complex data types, and a general lack of the context needed to set up sophisticated series of method and class interactions.

In this work, we are focused on the capabilities and applicability of common fitness functions. In the following subsections, we will assess the results of our study with respect to each research question. In Section 4.2, we compare the capabilities of each fitness function. In Section 4.3, we

	Budget	Chart	Closure	CommonsCLI	CommonsCodec	CommonsCSV	CommonsJXPath	Guava	JacksonCore	JacksonDatabind	JacksonXML	Jsoup	Lang	Math	Mockito	Time	Total
BC	2min	17	16	10	12	10	8	3	10	8	3	21	36	53	4	16	227
	10min	20	19	12	12	10	7	2	9	8	3	23	35	54	4	17	235
	Total	21	21	13	13	11	9	3	10	9	3	25	41	57	4	17	257
DBC	2min	14	16	12	12	11	5	2	9	8	1	20	32	48	4	15	209
	10min	19	19	11	13	10	6	2	10	10	2	22	36	47	4	18	229
	Total	19	22	13	13	11	7	2	10	10	2	26	40	52	4	18	249
EC	2min	8	7	5	4	2	0	0	5	4	1	9	12	13	3	6	79
	10min	10	5	4	3	2	2	0	6	3	1	8	13	12	3	5	77
	Total	10	8	6	4	2	2	0	6	4	1	9	16	15	3	6	92
LC	2min	15	12	7	11	11	4	2	8	9	3	14	31	50	3	15	196
	10min	18	14	10	11	9	8	2	8	10	3	23	32	52	3	14	217
	Total	18	17	11	13	11	8	2	8	11	3	24	37	55	3	15	236
MC	2min	10	6	5	7	5	0	1	5	2	1	7	9	25	2	5	90
	10min	10	10	4	6	4	0	2	4	2	1	7	11	24	2	5	92
	Total	12	10	6	7	5	0	2	5	2	1	8	14	27	2	6	107
MNEC	2min	9	8	5	7	3	1	2	5	5	1	6	10	29	2	5	98
	10min	11	6	2	5	4	2	2	4	4	0	9	13	27	2	3	94
	Total	11	9	5	7	4	2	2	5	5	1	9	13	21	2	6	113
OC	2min	9	7	6	8	2	1	2	4	5	2	7	13	36	2	5	109
	10min	13	9	5	9	2	2	2	3	4	2	8	17	33	2	6	117
	Total	13	12	6	9	2	2	2	4	5	2	9	18	38	2	8	132
WMC	2min	13	15	6	13	8	4	2	10	7	3	19	31	42	3	14	190
	10min	18	19	9	15	8	6	3	9	7	2	22	32	48	3	14	215
	Total	18	22	9	15	8	6	3	10	9	3	25	37	51	3	15	234
Default	2min	15	17	11	14	10	4	2	9	9	1	17	29	48	5	14	205
	10min	17	20	11	14	11	7	2	10	11	1	22	35	57	5	14	237
	Total	18	22	11	14	11	7	2	10	11	1	23	36	59	5	15	245
BC-EC-MC	2min	16	19	10	13	11	5	2	10	9	1	21	38	53	4	15	227
	10min	20	30	12	12	11	6	3	10	9	1	27	40	55	4	21	261
	Total	20	31	12	13	11	6	3	11	10	1	29	41	56	4	21	269
Any criterion (no combinations)	2min	18	23	15	14	11	9	3	10	10	3	31	43	61	5	16	272
	10min	23	29	15	17	11	10	3	10	11	3	33	45	59	5	18	292
	Total	23	31	16	17	11	10	3	10	12	3	37	46	62	5	18	304
Any criterion (w. combinations)	2min	18	24	15	15	11	10	3	10	10	3	33	44	62	5	16	279
	10min	23	37	15	17	11	10	3	11	12	3	35	46	64	5	21	313
	Total	23	37	16	17	11	10	3	11	13	3	39	47	65	5	21	321

TABLE 3 Number of faults detected by each fitness function. Totals are out of 26 faults (Chart), 133 (Closure), 24 (CommonsCLI), 22 (CommonsCodec), 12 (CommonsCSV), 14 (CommonsJXPath), 9 (Guava), 13 (JacksonCore), 39 (JacksonDatabind), 5 (JacksonXML), 64 (JSoup), 65 (Lang), 102 (Math), 38 (Mockito), 27 (Time), and 593 (Overall).

explore combinations of criteria. In Section 4.4, we explore the generation factors that indicate efficacy or lack of efficacy. Finally, in Section 4.5, we explore the source code metrics that indicate efficacy or lack of efficacy.

4.2 | Comparing Fitness Functions

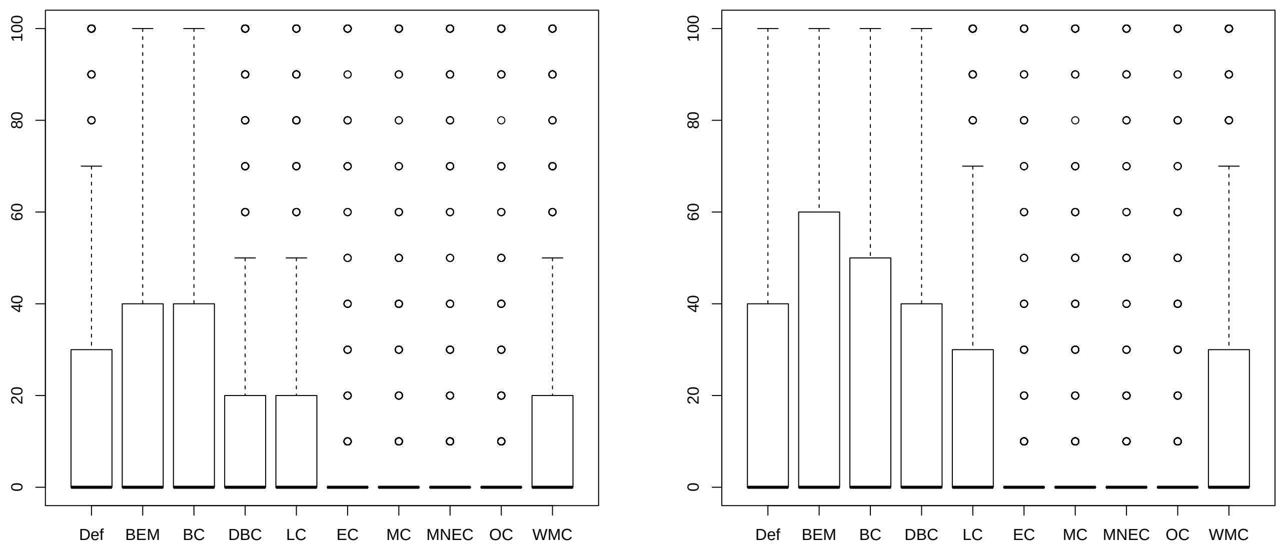
From Table 3, we can see that suites differ in effectiveness between criteria. Overall, branch coverage outperforms the other criteria, detecting 257 faults. Branch is closely followed by direct branch (249 faults), line coverage (236), and weak mutation coverage (234). These four fitness functions are trailed by the other four, with exception coverage showing the weakest results (92 faults). These rankings do not differ much on a per-system basis. At times, ranks may shift—for example, direct branch coverage occasionally outperforms branch coverage—but we can see two clusters form among the fitness functions. The first cluster contains branch, direct branch, line, and weak mutation coverage—with branch and direct branch leading the other two. The second cluster contains exception, method, method (no-exception), and output coverage—with output coverage producing the best results and exception coverage producing the worst.

Due to the stochastic nature of the search, one suite generated by EvoSuite may not always detect a fault detected by another suite—even if the same criterion is used. To more clearly understand the effectiveness of each fitness function, we must not track only whether a fault was detected, but how *reliably* it is detected. We are interested in the likelihood of detection—if a fresh suite is generated, how likely is it to detect a particular fault. To measure the likelihood, we record the proportion of detecting suites to the total number of suites generated for that fault. The average likelihood of fault detection is listed for each criterion, by system and budget, in Table 4. Figure 6 shows boxplots of the likelihood of detection for each fitness function and combination of functions.

We largely observe the same trends as above. Branch coverage has the highest overall likelihood of fault detection, with 22.60% of suites detecting faults given a two-minute search budget and 25.24% of suites detecting faults given a ten-minute budget. Direct branch coverage and line coverage follow with a 20.62-23.88% and 19.87-22.24% success rate, respectively. While the effectiveness of each criterion varies between system—direct branch outperforms all other criteria for Closure, for example—the two clusters noted above remain intact. Branch, line, direct branch, and weak mutation coverage all perform well, with the edge generally going to branch coverage. On the lower side of the scale, output, method, method (no exception), and exception coverage perform similarly, with a slight edge to output coverage.

	Budget	Chart	Closure	CommonsCLI	CommonsCodec	CommonsCSV	CommonsJXPath	Guava	JacksonCore	JacksonDatabind	JacksonXML	Jsoup	Lang	Math	Mockito	Time	Overall
BC	2min	45.00 %	4.66 %	26.67 %	33.18 %	57.50 %	25.00 %	17.78 %	50.77 %	14.62 %	60.00 %	18.13 %	34.00 %	27.94 %	9.21 %	34.81 %	22.60 %
	10min	48.46 %	5.79 %	28.33 %	31.36 %	60.83 %	25.00 %	20.00 %	60.00 %	13.08 %	60.00 %	21.72 %	40.15 %	32.75 %	8.42 %	39.26 %	25.24 %
	% Change	7.69 %	24.19 %	6.25 %	-5.48 %	5.80 %	0.00 %	12.50 %	18.18 %	-10.53 %	0.00 %	19.83 %	18.10 %	17.19 %	-8.57 %	12.77 %	11.72 %
DBC	2min	34.23 %	5.11 %	27.50 %	35.91 %	60.00 %	18.57 %	20.00 %	55.38 %	14.62 %	20.00 %	16.56 %	30.00 %	24.51 %	8.16 %	31.11 %	20.62 %
	10min	40.77 %	6.09 %	26.67 %	36.82 %	65.83 %	25.71 %	17.78 %	54.62 %	14.36 %	22.00 %	20.31 %	38.77 %	28.63 %	8.42 %	40.37 %	23.88 %
	% Change	19.10 %	19.12 %	-3.03 %	2.53 %	9.72 %	38.46 %	-11.11 %	-1.39 %	-1.75 %	10.00 %	22.64 %	29.23 %	16.80 %	3.23 %	29.76 %	15.78 %
EC	2min	22.31 %	1.35 %	5.83 %	12.27 %	16.67 %	0.00 %	0.00 %	20.00 %	3.33 %	20.00 %	6.41 %	7.54 %	6.37 %	6.05 %	9.26 %	6.56 %
	10min	21.54 %	0.98 %	5.83 %	12.27 %	16.67 %	1.43 %	0.00 %	22.31 %	1.79 %	20.00 %	5.94 %	9.23 %	7.06 %	5.26 %	9.63 %	6.64 %
	% Change	-3.45 %	-27.78 %	0.00 %	0.00 %	0.00 %	- %	- %	11.54 %	-46.15 %	0.00 %	-7.32 %	22.45 %	10.77 %	-13.04 %	4.00 %	1.29 %
LC	2min	38.85 %	4.14 %	21.25 %	22.73 %	59.17 %	20.71 %	21.11 %	51.54 %	15.13 %	60.00 %	12.03 %	31.23 %	25.78 %	5.79 %	30.00 %	19.87 %
	10min	46.15 %	4.81 %	22.08 %	21.36 %	50.00 %	25.71 %	20.00 %	53.08 %	17.95 %	56.00 %	17.50 %	34.31 %	29.22 %	5.79 %	36.67 %	22.24 %
	% Change	18.81 %	16.36 %	3.92 %	-6.00 %	-15.49 %	24.14 %	-5.26 %	2.99 %	18.64 %	-6.67 %	45.45 %	9.85 %	13.31 %	0.00 %	22.22 %	11.97 %
MC	2min	30.77 %	1.58 %	9.17 %	16.36 %	18.33 %	0.00 %	11.11 %	25.38 %	0.77 %	4.00 %	7.34 %	7.54 %	10.98 %	0.53 %	8.15 %	7.77 %
	10min	30.77 %	2.26 %	7.92 %	12.27 %	16.67 %	0.00 %	12.22 %	23.85 %	1.03 %	8.00 %	6.56 %	7.69 %	10.88 %	1.58 %	8.15 %	7.71 %
	% Change	0.00 %	42.86 %	-13.64 %	-25.00 %	-9.09 %	- %	10.00 %	-6.06 %	33.33 %	100.00 %	-10.64 %	2.04 %	-0.89 %	200.00 %	0.00 %	-0.87 %
MNEC	2min	23.46 %	2.18 %	9.17 %	14.55 %	12.50 %	0.71 %	11.11 %	20.00 %	4.10 %	2.00 %	7.50 %	6.62 %	12.16 %	1.05 %	6.67 %	7.59 %
	10min	30.77 %	1.88 %	7.50 %	15.00 %	12.50 %	4.29 %	12.22 %	24.62 %	5.90 %	0.00 %	7.50 %	7.54 %	12.06 %	0.79 %	5.19 %	8.09 %
	% Change	31.15 %	-13.79 %	-18.18 %	3.12 %	0.00 %	500.00 %	10.00 %	23.08 %	43.75 %	-100.00 %	0.00 %	13.95 %	-0.81 %	-25.00 %	-22.22 %	6.67 %
OC	2min	1.15 %	2.03 %	14.17 %	24.55 %	10.00 %	0.71 %	14.44 %	8.46 %	7.95 %	40.00 %	5.31 %	7.85 %	16.57 %	3.68 %	9.63 %	9.31 %
	10min	23.85 %	2.56 %	16.25 %	25.00 %	10.83 %	2.14 %	18.89 %	13.08 %	7.69 %	40.00 %	5.00 %	10.92 %	16.76 %	2.89 %	12.22 %	10.25 %
	% Change	12.73 %	25.93 %	14.71 %	1.85 %	8.33 %	200.00 %	30.77 %	54.55 %	-3.23 %	0.00 %	-5.88 %	39.22 %	1.18 %	-21.43 %	26.92 %	10.14 %
WMC	2min	38.08 %	4.44 %	19.17 %	31.36 %	41.67 %	15.00 %	16.67 %	44.62 %	8.97 %	26.00 %	15.00 %	24.15 %	23.04 %	5.79 %	25.19 %	17.59 %
	10min	46.15 %	5.56 %	20.00 %	33.64 %	45.00 %	17.86 %	18.89 %	42.31 %	7.69 %	18.00 %	18.28 %	32.15 %	27.45 %	5.53 %	27.04 %	20.34 %
	% Change	21.21 %	25.42 %	4.35 %	7.25 %	8.00 %	19.05 %	13.33 %	-5.17 %	-14.29 %	-30.77 %	21.88 %	33.12 %	19.15 %	-4.55 %	7.35 %	15.63 %
Default	2min	47.31 %	4.51 %	32.08 %	44.09 %	52.50 %	16.43 %	17.78 %	47.69 %	14.10 %	20.00 %	15.63 %	23.85 %	25.78 %	11.84 %	25.93 %	20.56 %
	10min	48.08 %	7.07 %	34.58 %	45.91 %	50.00 %	21.43 %	20.00 %	52.31 %	16.15 %	20.00 %	20.94 %	32.62 %	32.84 %	10.79 %	33.33 %	24.69 %
	% Change	1.63 %	56.67 %	7.79 %	4.12 %	-4.76 %	30.43 %	12.50 %	9.68 %	14.55 %	0.00 %	34.00 %	36.77 %	27.38 %	-8.89 %	28.57 %	20.10 %
BC-EC-MC	2min	43.08 %	5.64 %	30.42 %	37.27 %	60.00 %	20.71 %	17.78 %	50.00 %	14.87 %	20.00 %	19.38 %	40.46 %	30.39 %	10.26 %	35.93 %	24.03 %
	10min	53.85 %	8.05 %	30.83 %	38.18 %	50.00 %	26.43 %	21.11 %	56.15 %	14.62 %	20.00 %	25.00 %	48.15 %	34.31 %	10.53 %	47.04 %	27.84 %
	% Change	25.00 %	42.67 %	1.37 %	2.44 %	-16.67 %	27.59 %	18.75 %	12.31 %	-1.72 %	0.00 %	29.03 %	19.01 %	12.90 %	2.56 %	30.93 %	15.86 %

TABLE 4 Average likelihood of fault detection, broken down by fitness function, budget, and system. % Change indicates the average gain or loss in efficacy when moving from a two-minute to a ten-minute budget.



(a) 2m Budget

(b) 10m Budget

FIGURE 6 Boxplots of the likelihood of detection for each fitness function and combination. "Def" = Default Combination, "BEM" = BC-EC-MC Combination.

The boxplots in Figure 6 echo these results. All methods have medians near 0%, reflecting that many faults are not detected. However, for both budgets, branch coverage has a larger upper quartile than other fitness functions—indicating a large variance in results, but also that branch coverage yields more suites with a higher likelihood of detection than other methods. At the two-minute mark, it has a higher upper whisker as well. Direct branch coverage, line coverage, and weak mutation coverage follow in terms of third quartile size and upper whisker.

	Default Combination	BC	DBC	EC	LC	MC	MNEC	OC	WMC
BC	0.87	-	-	-	-	-	-	-	-
DBC	1.00	0.97	-	-	-	-	-	-	-
EC	**< 0.01**	**< 0.01**	**< 0.01**	-	-	-	-	-	-
LC	1.00	0.59	1.00	**< 0.01**	-	-	-	-	-
MC	**< 0.01**	**< 0.01**	**< 0.01**	1.00	**< 0.01**	-	-	-	-
MNEC	**< 0.01**	**< 0.01**	**< 0.01**	1.00	**< 0.01**	1.00	-	-	-
OC	**< 0.01**	**< 0.01**	**< 0.01**	0.77	**< 0.01**	0.96	0.94	-	-
WMC	0.91	0.07	0.75	**< 0.01**	0.99	**< 0.001**	**< 0.01**	**< 0.01**	-
BC-EC-MC Combination	0.56	0.99	0.79	**< 0.01**	0.27	**< 0.01**	**< 0.01**	**< 0.01**	**0.02**

TABLE 5 P-values for Nemenyi comparisons of fitness functions (two-minute search budget). Cases where we can reject the null hypothesis are **bolded**.

Branch coverage is the most effective criterion, detecting 257 faults. Branch coverage suites have, on average, a 22.60-25.24% likelihood of fault detection (2min/10min budget).

From Table 4, we can see that almost all criteria benefit from an increased search budget. Direct branch coverage and weak mutation benefit the most, with average improvements of 15.78% and 15.63% in effectiveness. In particular, it is reasonable that direct branch coverage benefits more than traditional branch coverage. In traditional branch coverage, branches executed through indirect chains of calls to program methods contribute to the total coverage. In direct branch coverage, only calls made directly by the test cases count towards coverage. Therefore, the test generator requires more time, and more method calls, to attain the same level of branch coverage. As a result, direct branch coverage attains slightly worse results than traditional branch coverage given the same time budget.

In general, all distance-driven criteria—branch, direct branch, line, weak mutation, and, partially, output coverage—improve given more time. These criteria all have complex, informative fitness functions that are able to guide the search process. Discrete fitness functions, such as those used by method coverage or exception coverage, benefit less from the budget increase. In such cases, the fitness function is unable to guide the search towards better solutions. More time is of benefit—as the generator can make more guesses. However, such time is not guaranteed to be beneficial, and does not necessarily result in improved test suites.

Distance-based functions benefit from an increased search budget, particularly direct branch and weak mutation.

Further increases in generation time beyond ten minutes may yield further improvements in the likelihood of fault detection. However, there is likely to be a plateau due to test obligations that the generator cannot satisfy, due to limitations in coverage of private code or manipulation of complex objects. Further, if test generation requires more time than it takes for a human to write tests, then the benefits of automation are more limited. Therefore, there is likely to be a limit to the gain from increasing the search budget.

We can perform statistical analysis to assess our observations. For each pair of criteria, we formulate hypothesis H and its null hypothesis, H0:

- H: Given a fixed search budget, test suites generated using criterion *A* will have a different distribution of likelihood of fault detection results than suites generated using criterion *B*.
- H0: Observations of fault detection likelihood for both criteria are drawn from the same distribution.

Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate H0 without any assumptions on distribution, we use the Friedman non-parametric alternative to the parametric repeated measures ANOVA [**?**]. Due to the limited number of faults for several systems, we have analyzed results across the combination of all systems. We apply the test for each pairing of fitness function and search budget with $\alpha = 0.05$.

At both budgets, the Friedman test confirms with p-value < 0.001 that the results for all fitness functions are not drawn from the same distribution. To differentiate and rank methods, we apply the post-hoc Nemenyi test in order to assess all pairs of fitness functions. The resulting p-values are listed in Tables 5-6.

	Default Combination	BC	DBC	EC	LC	MC	MNEC	OC	WMC
BC	1.00	-	-	-	-	-	-	-	-
DBC	1.00	1.00	-	-	-	-	-	-	-
EC	**< 0.01**	**< 0.01**	**< 0.01**	-	-	-	-	-	-
LC	0.95	0.71	1.00	**< 0.01**	-	-	-	-	-
MC	**< 0.01**	**< 0.01**	**< 0.01**	1.00	**< 0.01**	-	-	-	-
MNEC	**< 0.01**	**< 0.01**	**< 0.01**	1.00	**< 0.01**	1.00	-	-	-
OC	**< 0.01**	**< 0.01**	**< 0.01**	0.50	**< 0.01**	0.90	0.95	-	-
WMC	0.73	0.38	0.92	**< 0.01**	1.00	**< 0.01**	**< 0.01**	**< 0.01**	-
BC-EC-MC Combination	0.47	0.81	0.23	**< 0.01**	**0.02**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**

TABLE 6 P-values for Nemenyi comparisons of fitness functions (ten-minute search budget). Cases where we can reject the null hypothesis are **bolded**.

	Two-Minute Budget		**Ten-Minute Budget**		**All Budgets**	
Function	**Number of Faults (W. Combinations)**	**Number of Faults (No Combinations)**	**Number of Faults (W. Combinations)**	**Number of Faults (No Combinations)**	**Number of Faults (W. Combinations)**	**Number of Faults (No Combinations)**
Branch Coverage	5	13	2	7	0	1
Direct Branch Coverage	4	6	3	6	0	1
Exception Coverage	3	4	3	5	1	2
Line Coverage	2	5	3	4	0	0
Method Coverage	0	0	0	1	0	0
Method, No Exception	0	1	1	1	0	0
Output Coverage	1	3	2	7	1	2
Weak Mutation Coverage	3	6	4	7	0	0
Default Combination	2	-	5	-	0	-
BC-EC-MC Combination	4	-	13	-	1	-

TABLE 7 Number of faults uniquely detected by each suites generated using each fitness function (with and without considering combinations) for each budget, and for the combination of budgets.

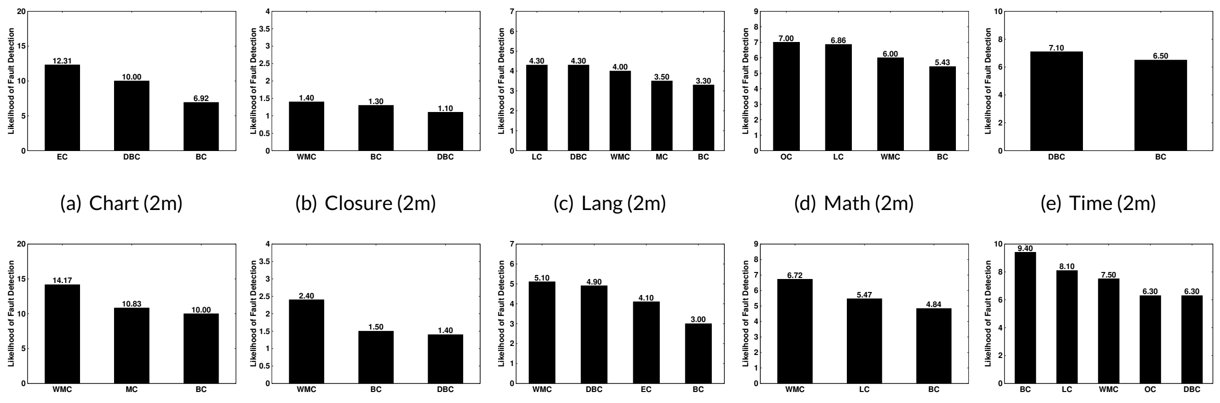
The results of these tests further validate the “two clusters” observation. For the four criteria in the top cluster—branch, direct branch, line, and weak mutation coverage—we can always reject the null hypothesis with regard to the remaining four criteria in the bottom cluster. This is also true in the opposite direction. The performance of the four criteria in the bottom cluster—exception, method, MNEC, and output coverage—is drawn from a different distribution to the criteria in the other cluster. Within each cluster, we usually fail to reject the null hypothesis.

Branch, direct branch, line, and weak mutation coverage outperform, with statistical significance, method, MNE, output, and exception coverage (both budgets).

Another way to consider performance is—regardless of overall performance—to look at whether a criterion leads to suites that detect faults that other criteria would not detect. Table 7 depicts the number of faults uniquely detected by each fitness function for each search budget, then the number of faults uniquely detected regardless of budget. Results are listed when combinations are considered, which we will discuss in Section 4.3, and when only considering the individual criteria.

From these results, we can see that almost all criteria clearly have *situational applicability*—that is, there are situations where their use leads to the detection of faults missed by other criteria. At both search budgets, a total of 38 faults are detected by a single criterion. Most interestingly, there are six faults that are—regardless of search budget—only detected by a single criterion.

The general efficacy of branch and direct branch coverage clearly can still be seen here, where each detects one fault that nothing else can detect, regardless of budget. However, criteria like exception coverage or output coverage—which have low average performance—can also detect faults that no other criterion can expose. Both criteria detect two faults, regardless of budget, that nothing else can catch. At each independent budget



(a) Chart (2m) (b) Closure (2m) (c) Lang (2m) (d) Math (2m) (e) Time (2m)

(f) Chart (10m) (g) Closure (10m) (h) Lang (10m) (i) Math (10m) (j) Time (10m)

FIGURE 7 Average % likelihood of fault detection for fitness functions once data is filtered for faults where the most effective function for that system has $< 30\%$ chance of detection.

level, exception, output, and weak mutation coverage each detect a number of faults that other criteria miss. This is especially worth noting at the ten-minute budget level, where suites generated to satisfy output and weak mutation coverage detect as many unique faults as suites generated to satisfy branch coverage.

These results suggest that—regardless of absolute efficacy—each criterion results in *different* test suites, each of which exercise the code under test in a distinct manner. Even if a criterion is not universally effective, it offers some form of situational applicability where it could be considered for use, and where it may have some value as part of a portfolio of testing tools.

For example, consider fault 100 for the Math project[6]. To address this fault, an estimation method switches from getting all parameters to only getting unbound parameters. Tests generated for exception coverage cause an exception by passing in a parameter with no measurements—i.e., an unbound parameter. This exception, even if triggered, is not retained by any other fitness function. Exception coverage, by prioritizing exceptions, ensures that the observed failure is retained and passed along to testers.

Output coverage is focused on coverage of abstract value classes for particular types of function output. It is particularly well-defined for numeric types. This makes it well-suited to discovering faults related to such numeric data types. Consider Mockito fault 26[7]. This fault lies in the Mockito framework's code for replicating Java's primitive datatypes. Illegal casts can be made from `integer` to other primitive types. Output coverage is able to trigger this fault by illegally casting `integer` variables to `double` variables. This *contextual* use of the class is not suggested by code coverage, and is not attempted by any of the other criteria.

To further understand the situational applicability of criteria, we filter the set of faults for those that the top-scoring criterion is ineffective at detecting. In Figure 7, we have taken the faults for five of the systems (Chart, Closure, Lang, Math, and Time), isolated any where the "best" criterion for that system (generally branch coverage, see Table 4) has $< 30\%$ likelihood of detection, and calculated the likelihood of fault detection for each criterion for that subset of the faults. In each subplot, we display the average likelihood of fault detection over the subset for any criterion that outperforms the best from the full set.

From these plots, we can see that there are always two-to-four criteria that are more effective in these situations. The exact criteria depend strongly on the system, and likely, on the types of faults examined. However, we frequently see the criteria mentioned above—including exception, weak mutation, and output coverage. Interestingly, despite the similarity in distance functions and testing intent, direct branch and line coverage are often more effective than branch coverage in situations where it has a low chance of detection. In these cases, the criteria drive tests to interact in such a way with the CUT that they are better able to detect the fault. The efficacy of alternative criteria in situations where the overall top performer offers poor results further emphasizes that:

Regardless of overall performance, most criteria have situational applicability, where their suites detect faults no other criteria can detect. Exception, output, and weak mutation coverage—in particular—seem to be effective for particular types of faults.

[6]https://github.com/Greg4cr/defects4j/blob/master/framework/projects/Math/patches/100.src.patch
[7]https://github.com/Greg4cr/defects4j/blob/master/framework/projects/Mockito/patches/26.src.patch

Given its strong overall performance, we would recommend that practitioners prioritize branch coverage—of the studied options—when generating test suites. However, we also stress that other criteria should not be ignored. Several—including exception and output coverage—can be quite effective at times, even if they are not effective on average. More research is clearly needed to codify the situations where such criteria can be effective and should be employed. Alternatively, such criteria could be used in *combination* with generally effective criteria such as branch coverage.

4.3 | Combinations of Fitness Functions

The analysis above presupposes that only one fitness function can be used to generate test suites. However, many search-based generation algorithms can simultaneously target multiple fitness functions. EvoSuite's default configuration, in fact, attempts to satisfy all eight of the fitness functions examined in this study [58]. In theory, suites generated through a combination of fitness functions could be more effective than suites generated through any one objective because—rather than focusing exclusively on one goal—they can simultaneously target multiple facets of the class under test.

For example, combining exception and branch coverage may result in a suite that both thoroughly explores the structure of the system (due to the branch obligations) and rewards tests that throw a larger number of exceptions. Such a suite may be more effective than a suite generated using branch or exception coverage alone. In fact, rather than generating tests for multiple independent criteria—when one or more of those criteria may only be effective situationally—a tester could, in theory, simultaneously generate for a combination of criteria in an attempt to produce suites effective in all situations.

To better understand the potential of combined criteria, we have generated tests for two combinations. The first is EvoSuite's default combination of the eight criteria that were the focus of this study. The second is a more lightweight combination of branch, exception, and method coverage—a combination of the most effective criterion overall with two that were selectively effective. In a recent study on combination of criteria on a subset of the Defects4J database, the BC-EC-MC combination was suggested as an effective baseline for new, untested systems [25].

Overall, EvoSuite's default configuration performs well, but fails to outperform all individual criteria in most situations. Table 3 shows that the default configuration detects 245 faults—fewer than branch and direct branch coverage, but more than the other individual criteria. It also uniquely detects two faults at the two-minute budget level and five at the ten-minute level (see Table 7). At the two minute level, this is worse than several individual criteria, but at the ten-minute level, it finds more faults than any individual criterion. According to Table 4, the default configuration's average overall likelihood of fault detection is 20.56% (2m budget)-24.69% (10m budget). At the two-minute level, this places it below branch, and direct branch coverage. At the ten-minute level, it falls below branch coverage, but above all other criteria. This places the default configuration in the top cluster—an observation confirmed by statistical tests (Tables 5 and 6).

However, it fails to outperform branch coverage in almost all situations. In theory, a combination of criteria should be able to detect more faults than any single criterion. In practice, combining *all* criteria results in suites that are fairly effective, but fail to reliably outperform individual criterion. The major reason for the less reliable performance of this configuration is the difficulty in attempting to satisfy so many obligations at once. As noted in Table 2, the default configuration must attempt to satisfy, on average, 1681 obligations. The individual criteria only need to satisfy a fraction of that total. As a result, the default configuration also benefits more than any individual criterion from an increased search budget—a 20.10% improvement in efficacy.

EvoSuite's default configuration has an average 20.56-24.69% likelihood of fault detection—in the top cluster, but failing to outperform all individual criteria.

Our observations imply that combinations of criteria could be more effective than individual criteria. However, a combination of all eight criteria results in unstable performance—especially if search budget is limited. Instead, testers may wish to identify a smaller, more targeted subset of criteria to combine during test generation. In fact, we can see the wisdom in such an approach by examining the results for the focused BC-EC-MC combination.

From Table 3, we can see that the BC-EC-MC combination finds a total of 269 faults—more than any individual criterion. From Table 4, this combination has a 24.03% (two-minute) to 27.84% (ten-minute) likelihood of detection. Again, this is better than any of the individual criteria. This combination also detects four faults uniquely at the two-minute budget, 13 at the ten-minute budget, and one fault regardless of the budget.The boxplots in Figure 6 show that the BC-EC-MC combination has a higher third-quartile box than any other method, indicating that it returned more results in that range than other methods. Statistical tests place this configuration in the top cluster, where it also outperforms weak mutation coverage with significance (Tables 5 and 6).

There are still situations where this combination can be outperformed by an individual criterion, particularly at the two-minute budget level. This combination requires fewer obligations than the eight-way default combination—around 354, on average—but still more than any individual criterion. This means that the combination benefits from a larger search budget (15.86% average improvement), and offers more stable performance at higher budgets. Still, this combination is clearly quite effective.

Of the studied criteria, exception coverage is unique in that it does not prescribe static test obligations. Rather, it simply rewards suites that cause more exceptions to be thrown. This means that it can be added to a combination with little increase in search complexity. The simplicity of exception coverage explains its poor performance as the *primary* criterion. It lacks a feedback mechanism to drive generation towards exceptions. However, exception coverage appears to be very effective *when paired with criteria that effectively explore the structure of the CUT*. Branch coverage gives exception coverage the feedback mechanism it needs to explore the code. Adding exception coverage to branch coverage adds little cost in terms of generation difficulty, and generally outperforms the use of branch coverage alone.

An example of effective combination can be seen in fault 60 for Lang[8]—a case where two methods can look beyond the end of a string. No single criterion is effective, with a maximum of 10% chance of detection given a two-minute budget and 20% with a ten-minute budget. However, combining branch and exception coverage boosts the likelihood of detection to 40% and 90% for the two budgets. In this case, if the fault is triggered, the incorrect string access will cause an exception to be thrown. However, this only occurs under particular circumstances. Therefore, exception coverage alone never detects the fault. Branch coverage provides the necessary means to drive program execution to the correct location. However, two suites with an equal coverage score are considered equal. Branch coverage alone may prioritize suites with slightly higher (or different) coverage, missing the fault. By combining the two, exception-throwing tests are prioritized and retained, succeeding where either criterion would fail alone.

Method coverage adds another "low-cost" boost. In general, a class will not have a large number of methods, and methods are either covered or not covered. Thus, even if method coverage is not a particularly helpful addition to a combination, its inclusion does not substantially increase the number of obligations that the test generator is tasked with fulfilling. An example where the addition of method coverage boosts efficacy can be seen in Lang fault 34[9]. This fault resides in two small (1-2 line) methods. Calling either method will reveal the fault, but branch coverage alone can easily overlook them because their invocation does not substantially improve branch coverage of the class as a whole. The addition of method coverage adds a useful "reminder" for the generator to invoke these simple methods.

A combination of branch, exception, and method coverage has an average 24.03-27.84% likelihood of fault detection—outperforming each of the individual criteria. It is more effective than the default eight-way combination because it adds lightweight situationally-applicable criteria to a strong, coverage-focused criterion.

It is unlikely that the BC-EC-MC combination is the strongest possible combination, and we can see some situations where a single criterion is still the most effective. In fact, it is unlikely that any one criterion or combination of criteria will ever universally be the "best". The most effective criteria depend on the type of system under test, and the types of faults that the developers have introduced into the code. Still, there is a powerful idea at the heart of this combination. When generating tests, a strong coverage-focused criterion should be selected as the primary criterion. Then, a small number of targeted, orthogonal criteria can be added to that primary criterion. More investigation is needed into the situational applicability of criteria in order to better understand when any one criterion or a combination of criteria will be effective.

4.4 | Understanding the Generation Factors Impacting Fault Detection

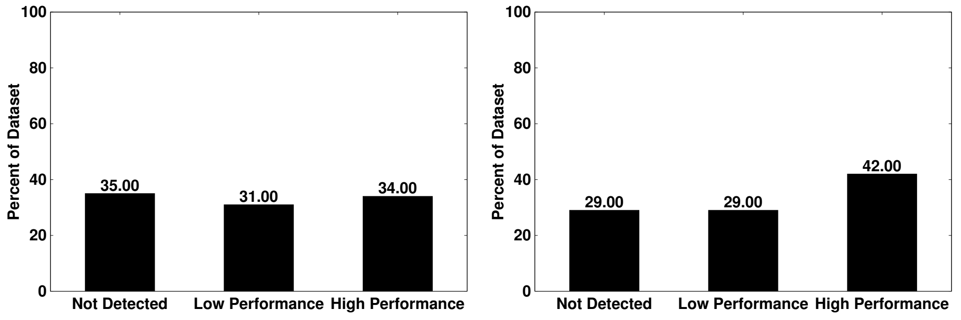
Using the TAR3 treatment learner [27], we have generated five treatments from each of the two generation factor datasets for each of the three classifications ("Not Detected", "Low Performance", and "High Performance"). The treatments are scored according to their impact on class distribution, and top-scoring treatments are presented first.

First, the following treatments indicate the factors pointing most strongly to a "high" likelihood of fault detection:

Two-Minute Dataset:

1. BC (fixed) $>$ 79.71%, LC (fixed) $>$ 87.89%, % of obligations satisfied $>$ 89.59%, BC (faulty) $>$ 79.85%
2. BC (fixed) $>$ 79.71%, LC (fixed) $>$ 87.89%, % of obligations satisfied $>$ 89.59%, LC (faulty) $>$ 88.09%
3. LC (fixed) $>$ 87.89%, % of obligations satisfied $>$ 89.59%, BC (faulty) $>$ 79.85%
4. BC (fixed) $>$ 79.71%, % of obligations satisfied $>$ 89.59%, LC (faulty) $>$ 88.09%

[8]https://github.com/apache/commons-lang/commit/a8203b65261110c4a30ff69fe0da7a2390d82757.
[9]https://github.com/apache/commons-lang/commit/496525b0d626dd5049528cdef61d71681154b660



(a) 2m Budget

(b) 10m Budget

FIGURE 8 Class distributions of the data subsets fitting the top treatments learned from each dataset for the "High Performance" class.

5. BC (fixed) $> 79.71\%$, LC (fixed) $> 87.89\%$, % of obligations satisfied $> 89.59\%$, BC (faulty) $> 79.85\%$, LC (faulty) $> 88.09\%$

Ten-Minute Dataset:

1. BC (faulty) $> 86.25\%$, % of obligations satisfied $> 94.34\%$, LC (faulty) $> 92.08\%$
2. BC (faulty) $> 86.25\%$, % of obligations satisfied $> 94.34\%$, LC (fixed) $> 92.23\%$
3. BC (faulty) $> 86.25\%$, % of obligations satisfied $> 94.34\%$, LC (faulty) $> 92.08\%$, LC (fixed) $> 92.23\%$
4. BC (faulty) $> 86.25\%$, % of obligations satisfied $> 94.34\%$, BC (fixed) $> 85.97\%$, LC (fixed) $> 92.23\%$
5. BC (faulty) $> 86.25\%$, % of obligations satisfied $> 94.34\%$, BC (fixed) $> 85.97\%$, LC (faulty) $> 92.08\%$

In Figure 8, we plot the class distribution of the subset fitting the highest-ranked treatment learned from both datasets. Comparing the plots in Figure 4 to the subsets in Figures 8, we can see that the treatments do impose a large change in the class distribution—a lower percentage of cases have the "Not Detected" class, and more have the other classifications. This shows that the treatments do reasonably well in predicting for success. Test suites fitting these treatments are not guaranteed to be successful, but are likely to be.

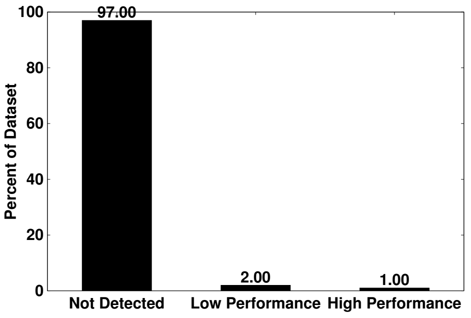
Note that some treatments are subsets of other treatments. For example, the third treatment for the two-minute dataset above is a subset of the top treatment. Each treatment indicates a set of attributes and value ranges for those attributes that, when applied *together*, tend to lead to particular outcomes. A smaller treatment that is a subset of a larger treatment, when applied, will lead to a different subset of the overall data set with a different class distribution to the larger treatment. In general, smaller treatments are easier for humans to understand—this is why we have limited treatment size to five attributes. However, within that limit, a larger treatment may have more explanatory power than a smaller treatment. In this case, as treatments are ranked by score, the larger treatment is more indicative of the target class and the additional factors offer additional explanatory power.

We can make several observations. First, the most common factors selected as indicative of efficacy are all coverage-related factors. Even if their goal is not to attain coverage, successful suites thoroughly explore the structure of the CUT. The fact that coverage is important is not, in itself, entirely surprising—if patched code is not well covered, the fault is unlikely to be discovered.

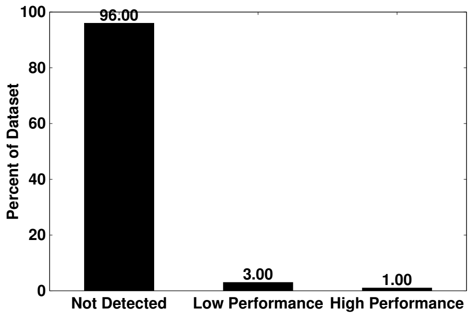
More surprising is how much weight is given to coverage. Suite size has been a focus in recent work, with Inozemtseva et al. (and others) finding that the size has a stronger correlation to efficacy than coverage level [36]. However, size attributes—number of tests and test length—do not appear in any of the generated treatments. Kendall correlation tests further reinforce this point. For both budgets and for both suite size and length, correlation strengths were all approximately 0.25—"weak to low" correlations.

However, rather than indicating that larger test suites are not necessarily more effective at detecting real faults, it is important to look at the suites themselves. As can be seen in Section 3.4, test suite sizes do not range dramatically between each of the fitness configurations. Within the size ranges of suites in this study, larger suites also do not necessarily outperform smaller suites. Suites for Output Coverage, which performs poorly on average, are often similar in size to those yielded for the higher-scoring fitness functions. The suites for the combinations are, naturally, the largest. However, the largest suites belong to the "default" combination—which is often outperformed by branch coverage and the BC-EC-MC combination. Exception Coverage, the poorest performing fitness function on average, does have the smallest test suites. However, other factors, such as its low coverage of source code, seem to play a larger role in determining suite efficacy than size alone.

The other factor noted as indicative of efficacy is the percent of obligations satisfied. This too seems reasonable. If a suite covers more of its test obligations, it will be better at detecting faults. For coverage-based fitness functions like branch and line coverage, a high level of satisfied obligations



(a) 2m Budget



(b) 10m Budget

FIGURE 9 Class distributions of the data subsets fitting the top treatments learned from each dataset for the "Not Detected" class.

naturally correlates with a high level of branch or line coverage. For other fitness functions, the correlation may not be as strong, but it is also likely that suites satisfying more of their obligations also tend to explore more of the structure of the CUT.

Factors that strongly indicate a high level of efficacy include line or branch coverage over either version of the code and high coverage of their own test obligations. Coverage and obligation satisfaction are favored over factors related to suite size or test obligations.

Note, however, that we still do not entirely understand the factors that indicate a high probability of fault detection. From Figure 8, we can see that the treatments radically alter the class distribution from the baseline in Figure 4. Still, we can also see that suites from the "Not Detected" class still form a significant portion of that class distribution. From this, we can conclude that factors predicted by treatments are a necessary precondition for a high likelihood of fault detection, but are not sufficient to ensure that faults are detected. Unless code is executed, faults are unlikely to be found. Thus, coverage is impotent. However, how code is executed matters, and execution alone does not guarantee that faults are triggered and observed as failures. The fitness function determines how code is executed. It may be that fitness functions based on stronger adequacy criteria (such as complex condition-based criteria [67]) or combinations of fitness functions will better guide such a search. While coverage increases the likelihood of fault detection, it does not ensure that suites are effective.

To better understand factors indicating success, we can also perform treatment learning for the opposite scenario—what indicates that we will *not* detect a fault? Factors that indicate a lack of success include:

Two-Minute Dataset:

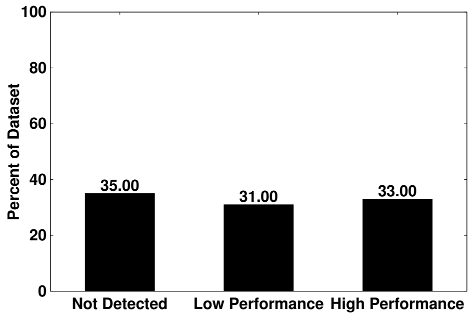
1. LC (faulty) $\leq 11.77\%$, LC (fixed) $\leq 12.41\%$, BC (faulty) $\leq 8.96\%$
2. LC (faulty) $\leq 11.77\%$, LC (fixed) $\leq 12.41\%$, BC (faulty) $\leq 8.96\%$, BC (fixed) $\leq 9.33\%$
3. LC (faulty) $\leq 11.77\%$, LC (fixed) $\leq 12.41\%$
4. LC (faulty) $\leq 11.77\%$
5. LC (faulty) $\leq 11.77\%$, BC (faulty) $\leq 8.96\%$

Ten-Minute Dataset:

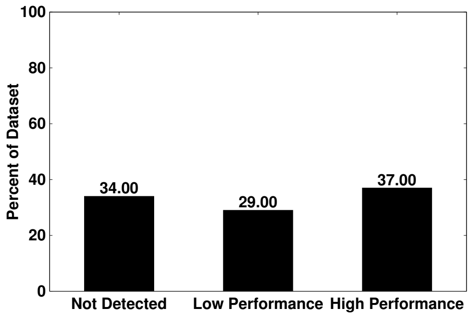
1. LC (fixed) $\leq 15.02\%$, BC (fixed) $\leq 12.12\%$
2. LC (fixed) $\leq 15.02\%$, BC (faulty) $\leq 11.91\%$
3. LC (fixed) $\leq 15.02\%$, BC (faulty) $\leq 11.91\%$, BC (fixed) $\leq 12.12\%$
4. BC (fixed) $\leq 12.12\%$
5. LC (fixed) $\leq 15.02\%$, LC (faulty) $\leq 14.54\%$

In Figure 9, we plot the class distribution of the subset fitting the highest-ranked treatment learned from both datasets for the "Not Detected" outcome. Comparing the plots in Figure 4 to the subsets in Figures 8, we can see a dramatic change in the class distribution. These treatments predict quite clearly a lack of success, with almost no data records from the other classes still matching the treatment.

The factors indicating a lack of success are entirely coverage-based. If coverage—line or branch—is less than approximately 15%, then the odds of effective fault detection are extremely low. This further reinforces the discussion above:



(a) 2m Budget



(b) 10m Budget

FIGURE 10 Class distributions of the data subsets fitting the top treatments learned from each dataset for the "Low Performance" class.

While coverage may not ensure success, it is a prerequisite. If the code is not exercised, then the fault will not be found.

Finally, we can examine one additional classification—"low" efficacy. What factors differentiate situations where a faulty is highly likely to be found from situations where it is still generally found, but with a low likelihood of detection? The factors that suggest this situation include:

Two-Minute Dataset:

1. LC (faulty) $> 88.09\%$, % of obligations satisfied $> 89.59\%$, LC (fixed) $> 87.89\%$
2. LC (faulty) $> 88.09\%$, % of obligations satisfied $> 89.59\%$
3. BC (faulty) $> 79.85\%$, BC (fixed) $> 79.71\%$, % of obligations satisfied $> 89.59\%$, LC (fixed) $> 87.89\%$
4. BC (fixed) $> 79.71\%$, LC (fixed) $> 87.89\%$, % of obligations satisfied $> 89.59\%$, BC (faulty) $> 79.85\%$, LC (faulty) $> 88.09\%$
5. BC (fixed) $> 79.71\%$, LC (faulty) $> 88.09\%$, % of obligations satisfied $> 89.59\%$, LC (fixed) $> 87.89\%$

Ten-Minute Dataset:

1. LC (faulty) $> 92.08\%$, BC (fixed) $> 85.97\%$
2. LC (faulty) $> 92.08\%$, BC (fixed) $> 85.97\%$, LC (fixed) $> 92.23\%$
3. BC (faulty) $> 86.25\%$, BC (fixed) $> 85.97\%$, LC (fixed) $> 92.23\%$
4. BC (faulty) $> 86.25\%$, BC (fixed) $> 85.97\%$, LC (faulty) $> 92.08\%$
5. BC (faulty) $> 86.25\%$, LC (fixed) $> 92.23\%$

These treatments, and their resulting class distributions—illustrated in Figure 10—are very similar to the factors predicting "high" performance. This is particularly true for the two-minute dataset, where we simply see a small downgrade in the number of "high" cases. From this, we can again see that coverage is needed to detect faults, but more data is needed to help ensure reliable detection.

However, we can make one interesting observation from the treatments learned from the ten-minute dataset. The ten-minute dataset includes more effective test suites from the start, allowing us to better differentiate high efficacy from low—but extant—efficacy. The treatments learned from the ten-minute dataset for "low efficacy" are lacking any reference to satisfaction of their obligations—a factor that is always present in the treatments learned for the "high efficacy" classification. We can also see a shift in the resulting class distribution in Figure 10 from that in Figure 8. The percent of "low" efficacy examples remains the same, but there are fewer "high" cases and more "not detected" cases.

The most important factor differentiating cases where a fault is occasionally detected and cases where a fault is consistently detected is satisfaction of the chosen criterion's test obligations.

Thus, we can observe that the most effective test suites are those that both cover a large percentage of their own obligations and thoroughly exercise the targeted code. In the case of criteria like branch coverage, these two go hand-in-hand. However, this also illustrates why criteria based on orthogonal factors to code coverage—like exceptions—tend to be best used in combination with coverage-based criteria. In future work, we will further explore such factors and others, and investigate how to best ensure effective test suite generation.

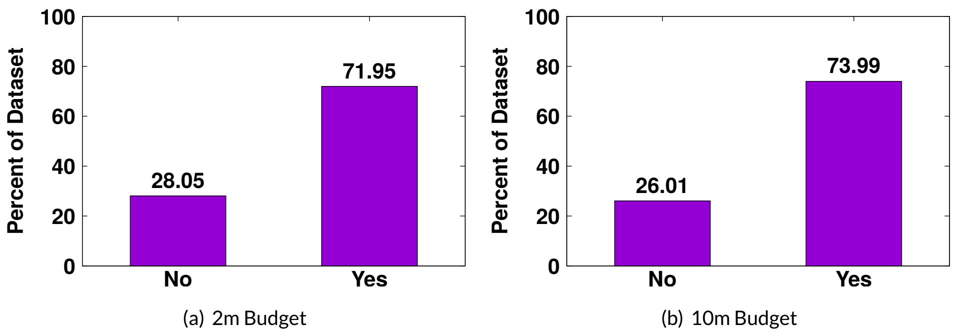


FIGURE 11 Class distributions for the subsets of the two overall datasets fulfilling the top-ranked **"Yes"** treatment for each.

4.5 | Understanding the Code Metrics Impacting Fault Detection

The following treatments were reported by TAR3, from the "overall" (all fitness functions) code metric datasets, as indicative of situations where generated test suites—regardless of the targeted criterion—were able to detect a fault. The treatments are scored according to their impact on class distribution and the number of cases in the treatment-fulfilling subset of the data, and the top-scoring treatments are presented first. Definitions of metrics are listed in Table 1.

Two-Minute Overall Dataset:

1. TCD = [0.52..0.93], NOD = [0.00..1.00)]
2. TCD = [0.52..0.93], NS = [0.00..1.00)]
3. TCD = [0.52..0.93], TNLS = [0.00..1.00)]
4. TCD = [0.52..0.93], TNS = [0.00..1.00)]
5. TCD = [0.52..0.93], TNLS = [0.00..1.00)], NS = [0.00..1.00)]

Ten-Minute Overall Dataset:

1. TCD = [0.52..0.93], CD = [0.53..0.93]
2. TCD = [0.52..0.93]
3. PDA = [30.00..208.00], DLOC = [348.00..4017.00]
4. TNLPM = [38.00..287.00], DLOC = [348.00..4017.00]
5. CD = [0.53..0.93]

Figure 11 illustrates the shift in the class distribution for the subset of each overall dataset fitting the top-ranked treatment targeting the "Yes" classification for each respective budget. Comparing to the baseline distribution in Figure 5, we can see that the class distribution has shifted heavily in favor of the "Yes" class—from 47.44-71.95% and 49.80-73.90% respectively. As approximately 25% of the examples still have a "No" verdict, these treatments are not perfectly explanatory. Some classes may match the treatment and still evade fault detection. However, the shift in distribution still suggests that the metrics and value ranges named in the treatments have explanatory power. The treatments indicate that:

Generated test suites are effective at detecting faults in well-documented classes.

The most consistently-identified metric from the two-minute dataset—and one that appears in treatments for the ten-minute dataset as well—is that a high Total Comment Density (52-93%) tends to indicate that the fault is more likely to be found. This is above the 75th percentile of the results depicted in Figure 1. From the ten-minute dataset, we can also see that suites tend to detect faults in classes with a Comment Density of 52-93%, 348-4,017 Documented Lines of Code, and with 30-208 documented public methods (PDA).

Test generation will yield better results if more of the class is publicly accessible.

A Total Number of Local Public Methods from 38-287 indicates that generated suites are more likely to detect a fault. From Table 1, we can see that this is well above the median TNLPM (12.00), indicating that allowing direct access to more of your methods will yield better test generation

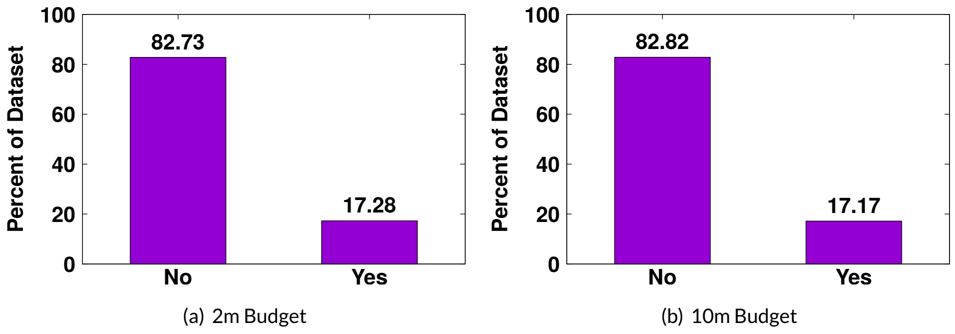


FIGURE 12 Class distributions for the subsets of the two "overall" datasets fulfilling the top-ranked **"No"** treatment for each.

results. A number of treatments also suggest a (Total) Number of (Local) Setters (TNS, NS, TNLS) of 0. Having no setters implies that all attributes are publicly-accessible. In addition, while Public Documented API is intended to capture how well-documented the class is, it also illustrates this point—a high PDA indicates not just that there are a large number of documented methods, but that a large number of methods are public as well.

The top treatment for the two-minute dataset also suggests a Number of Descendants (NOD) of 0. This value reflects the vast majority of classes, and also appears in the treatments for the "No" classification. Therefore, we consider it to be a coincidental factor.

The test suites for Exception, Method, Method (Top Level, No Exception), and Output Coverage did not detect enough faults to yield useful treatments for the "Yes" classification. Test suites for the remaining four criteria—Branch, Direct Branch, Line, and Weak Mutation Coverage—yielded treatments largely echoing the "overall" dataset. However, multiple treatments for those criteria-specific datasets included the metric-value pairings CLLC = [0.24..0.86] and CC = [0.27..0.91]. These metrics—Clone Logical Line Coverage (CLLC) and Clone Coverage (CC)—tell us that:

Faults are easier to detect if a large proportion of the class contains duplicate code.

These value ranges are the high end of the scale, and do not include the majority of classes, indicating that the importance of these metrics is not coincidental. Intuitively, if there is a lot of duplicate code, the overall class structure will be easier to cover. Test generation methods driven by code coverage will be able to quickly achieve high levels of coverage, making it easier to reach and execute the code containing the fault.

The following treatments were reported by TAR3, based on the all-fitness-function datasets, as indicative of situations where generated test suites—regardless of the targeted criterion—were **not able** to detect a fault. Figure 12 illustrates the shift in the class distribution for the subset of each overall dataset fitting the top-ranked treatment targeting the "No" classification for each.

Two-Minute Overall Dataset:

1. DIT = [1.00..2.00)], PDA = [1.00..4.00)]
2. NPM = [0.00..3.00)], NOC = [0.00..1.00)]
3. NPM = [0.00..3.00)], NOD = [0.00..1.00)]
4. NPM = [0.00..3.00)], TNLPA = [0.00..1.00)]
5. CBO = [17.00..98.00], NOC = [0.00..1.00)]

Ten-Minute Overall Dataset:

1. NPM = [0.00..3.00)], LCOM5 = [1.00..2.00)]
2. NS = [0.00..1.00)], CBO = [17.00..98.00]
3. NPM = [0.00..3.00)], NOC = [0.00..1.00)]
4. NPM = [0.00..3.00)], NOD = [0.00..1.00)]
5. NPM = [0.00..3.00)], NOD = [0.00..1.00)], NLPA = [0.00..1.00)]

Comparing to the baseline distribution in Figure 5, we can see that the class distribution has shifted heavily in favor of the "No" class—from 52.56-82.73% and 50.20-82.82% respectively. Once again, not all examples in the subset have a "No" classification. However, the sharp shift in distribution suggests that the treatments have explanatory power. From the produced treatments, we can see that:

Test generation methods struggle with classes that have a large number of private methods or attributes.

The majority of the treatments include a Number of Public Methods (NPM) between 0-3 methods. This is an extremely low number of public methods—far below the median—indicating that much of the class is private. In these cases, the generated suites are likely to have low overall code coverage, and are more likely to miss a fault. One treatment also includes a PDA of 1-3 methods. In this case, this metric's importance is likely not to be as an indicator of the level of documentation, but an indicator that there are a low number of public methods.

Treatments also include a (Total) Number of Local Public Attributes (TNLPA/NLPA) of 0—indicating that any extant attributes are private. Like with the "Yes" treatments, the attributes serve a different purpose than methods in test generation—serving as a way to configure class state and drive coverage of the code contained in the methods. A lack of public attributes limits the ability of the generation framework to control class state.

It can be more difficult to generate tests for classes with dependencies or inherited state.

A Coupling Between Objects of 17-98 is well above the mean of 8, indicating that generated suites are likely to miss a fault in a class coupled to a large number of other classes. In such cases, the test generation framework would need to set up dependencies and put them in the state needed to expose the fault in the targeted class—a non-trivial task.

A related metric is the Depth of the Inheritance Tree (DIT). A DIT of 1 indicates that the target class has a parent class. The existence of a parent class indicates that some methods and attributes are inherited from a parent. This, in itself, is not a problem as the test generator does not need to properly set up a parent like in the last case (and many classes in the dataset have a parent). While we have discussed the metric-value pairs in each treatment as largely being independent, the pairs in a treatment can be related. In this case, the treatment pairs the DIT of 1 with a low PDA. This implies situations where part of the class is inherited from a parent, and where the class has a low number of public methods of its own. If much of the complexity of the class is inherited and new functionality is largely private, the test generation framework may have difficulty in driving execution to the fault location.

Once again, the fact that the vast majority of classes have a NOC/NOD of 0 and that this metric-value pair appears for both targeted classes suggests that these two metrics are coincidental, and we have chosen to ignore them in our discussion. LCOM5 = 1 matches the vast majority of classes in the dataset, and is not particularly informative. Therefore, we also suspect that it is coincidental. All three of these metrics—NOC, NOD, and LCOM5—are paired with a low NPM. We suspect that the low NPM is the true indicator of test efficacy.

One treatment includes the metric-value pair $NS = 0$. This pairing also appeared for the "Yes" treatments, and captures a majority of case examples. We believe it is coincidental for this classification, but not the "Yes" classification, as the treatments for the "Yes" classification included a number of related metrics and often included this pairing. In this case, it was paired with a high CBO—a more informative metric.

The datasets for all eight coverage criteria offered very similar results to the overall datasets for the "No" classification. No additional metric-value pairs were observed.

We will summarize the trends observed among the identified metrics, and discuss their implications. First, **test generation methods struggle with classes that have a large number of private methods or attributes, and thrive when the class structure is accessible.** The clearest indication of this trend is in the treatments produced for the "No" classification, where a low number of public methods appeared in almost every treatment. In the "Yes" treatments, a large number of public methods is prescribed. This finding is further backed by the prevalence of the Public Documented API metric for both classifications, where the "Yes" treatments prescribe for a PDA of 30-208 methods, and in the "No" treatments, where the treatments prescribe 1-3 methods. PDA is a documentation metric, intended to highlight classes in need of more documentation. However, as it measures the proportion of public methods that are documented, it has a strong correlation to the Number of Public Methods (0.63, measured using the Kendall correlation test). In this experiment, PDA seems to further indicate the effect of private methods on test generation effectiveness.

This finding makes intuitive sense. The test generation technique explored in this experiment is driven by various coverage criteria—largely based on different ways of executing structural elements of the code. The obligations of such criteria will inevitably require that code structures within private methods be executed in the mandated manner. In practice, coverage can be measured over private code, and the feedback mechanisms that power search-based test generation will still reward greater coverage of that code. However, without the ability to directly call such methods, the generation technique will struggle to actually obtain coverage. Private methods must be covered *indirectly*, through calls to public methods. While the generation technique will attempt to adjust the input provided to the public methods to obtain higher indirect coverage of private methods, this indirect manipulation may prove unsuccessful due to constraints placed on how the public method can invoke the secondary private method or discontinuity in the scoring method introduced by this indirect manipulation.

Both the "Yes" and "No" treatments touch on the use of private attributes as well, through the (Total) Number of (Local) Setters and Total Number of Public Attributes. Private attributes do not directly prevent code from being covered in the same manner as private methods do. However, they may still result in lower coverage and missed faults by limiting the ability of the test generation technique to manipulate the state of the class.

Certain methods may only be coverable by setting the class attributes to particular values, and some failures may only be triggered under particular class states. If the class attributes are private, the test generation technique will need to find indirect means of manipulating those attributes. Again, this is a non-trivial task.

Authors have previously hypothesized that private methods are a reason for poor test generation results [60, 8, 24]. Our findings offer evidence supporting this hypothesis. In practice, we would not advocate that developers reduce the use of private methods or attributes—the protection offered through this feature is crucial. Instead, test generation techniques need to be augmented with better means of increasing coverage of private methods. For example, machine learning techniques may be able to form a behavioral model of the indirectly-called method that could offer better feedback than the existing scoring function used to guide search-based generation.

Second, **generated suites seem to be more effective at detecting faults in classes with more documentation**. The metrics that were the most common indicators of success are largely documentation-related metrics such as the (Total) Comment Density, the Documented Lines of Code, and—to a lesser extent—the Public Documented API. As discussed previously, Public Documented API is strongly correlated to the Number of Public Methods, and the documentation connection is likely to be a coincidence. TCD and DCLOC both have a moderate correlation to the PDA (strengths of 0.48 and 0.47, respectively). However, TCD is only weakly correlated to NPM (0.24) and DLOC is only moderately correlated (0.42). Therefore, TCD, CD, and DLOC are important indicators of efficacy in their own right, and are not merely indicative of a higher number of public methods.

There is no reason to expect the presence of documentation to assist automated test case generation techniques, as such techniques do not make use of documentation in any way. Instead, *it is important to consider what the presence of documentation implies*. One theory is that there is an unintended selection bias in how the case examples were gathered for Defects4J. The studied faults were identified by searching for project commits that referenced bug reports. Bug reports are more likely to be filed for classes that are frequently invoked by the users and developers of a project. In turn, classes that are more heavily used tend to be ones that developers spend more time refining and polishing. Classes that are expected to be used more heavily will, naturally, be better documented. As a result, it could be that well-documented classes are more likely to be identified as subjects for Defects4J than classes with low amounts of documentation.

However, this theory does not completely explain these results. The majority of classes in Defects4J *do not* have a high TCD or DLOC. The median TCD—36%—is likely to be higher than the median for all classes in the wild, but is still well below the TCD prescribed by the treatments—52-93%. The classes with a high TCD are also not necessarily smaller than other examples. The median LLOC—non-comment lines of code—for classes with a TCD greater than 52% is 288.50, compared to an overall median of 208.50. The median TNM is 37.50, slightly above the median of 37. This implies that the well-documented classes are actually larger than the average class in Defects4J. These are not simpler examples. Further, TCD and DLOC do not have a strong correlation with any non-documentation metric, meaning that there is not a simple explanation within the collected date.

Therefore, there must be some additional factor implied by the presence of high levels of documentation. More research is needed to understand the impact of documentation in these case examples. While the presence of documentation should not directly assist automated test case generation techniques, its presence may hint at the maturity, testability, and understandability of the class.

Third, **classes with a large number of dependencies are more difficult to test than more self-contained classes**. This is indicated in the "No" treatments by a Coupling Between Objects (CBO) of 17-98 classes, and—to a lesser extent—an inheritance tree depth (DIT) of 1, meaning that the target class inherits functionality from a parent. For a class to be included in Defects4J, it must be directly changed by the patch applied to fix the fault. This means that the fault lies in the class that depends on other classes, rather than being a fault *in* one of the dependencies.

If a class depends on other classes, then the test generation technique may also need to initialize and manipulate the dependencies properly as part of the test generation process. By not doing so properly, we may not be able to achieve high coverage of the target class. Further, we may fail to expose the fault even if the code is covered, as we are trying to make use of the target class outside of the "normal" use of the rest of its dependencies. Because of that, we may see the same *incorrect* behavior from both the working and faulty versions of the class, missing the fault. Researchers have discussed the problem of configuring complex dependencies as part of the broader challenges of controlling the execution environment when generating test cases [8, 7, 23]. Again, our observations provide clear motivation for further research on this topic.

Fourth, when structure-based criteria like Branch Coverage are targeted, **test generation techniques are more effective when a large proportion of the class is duplicated code**. This is supported by the prevalence of Clone Logical Line Coverage (CLLC) and Clone Coverage (CC) in the criteria-specific "Yes" treatments. This observation makes intuitive sense. If a large proportion of the code is identical, then that code will be easier to cover through automated generation. Even if the fault does not lie in the duplicated code, it will be easier to guide execution to the faulty code.

Code duplication is discouraged during development, as any changes will need to be made in multiple locations. Further, duplicating code rather than encapsulating it in one location and invoking it throughout the class will not have any benefit, as a high CLLC simply implies that there is not a significant quantity of non-duplicated code. Rather than offering actionable information, this observation simply indicates that classes with a lot of duplicate code and very little other functionality are easier to test than complex classes.

5 | RELATED WORK

Those who advocate the use of adequacy criteria hypothesize that criteria fulfillment will result in test suites more likely to detect faults—at least with regard to the structures targeted by that criterion. If this is the case, we should see a correlation between higher attainment of a criterion and the chance of fault detection for a test suite [32]. Researchers have attempted to address whether such a correlation exists for almost as long as such criteria have existed [52, 48, 11, 18, 19, 17, 30, 36, 29]. Inozemtseva et al. provide a good overview of work in this area [36]. Our focus differs—our goal is to examine the relationship between fitness function and fault detection efficacy for search-based test generation. However, fitness functions are largely based on, and intended to fulfill, adequacy criteria. Therefore, there is a close relationship between the fitness functions that guide test generation and adequacy criteria intended to judge the resulting test suites. In recent work, McMinn et al. have even proposed using search techniques to evolve new coverage criteria that combine features of existing criteria [46].

EvoSuite has previously been used to generate test suites for the systems in the Defects4J database. Shamshiri et al. applied EvoSuite, Randoop, and Agitar to each fault in the Defects4J database to assess the general fault-detection capabilities of automated test generation [60]. They found that the combination of all three tools could identify 55.7% of the studied faults. Their work identifies several reasons why faults were not detected, including low levels of coverage, heavy use of private methods and variables, and issues with simulation of the execution environment. In their work, they only used the branch coverage fitness function when using EvoSuite. Yu et al. used EvoSuite to generate tests for 224 of the faults in Defects4J, examining whether such tests could be used for program repair [69]. In our initial study, we expanded the number of EvoSuite configurations to better understand the role of the fitness function in determining suite efficacy [24]. We have also compared and contrasted test suites generated to achieve traditional branch coverage and direct branch coverage, noting that each fitness function detects different faults [26].

We used the Defects4J faults to understand the effect of combining fitness functions, identifying lightweight combinations of fitness functions that could effectively detect faults [25]. Rojas et al. also examined combining fitness functions, finding that, given a fixed generation budget, multiple fitness functions could be combined with minimal loss in coverage of any single criterion and with a reasonable increase in test suite size [58]. Others have explored combinations of coverage criteria with non-functional criteria, such as memory consumption [41] or execution time [68]. Few have studied the effect of such combinations on fault detection. Jeffrey et al. found that combinations are effective following suite reduction [37]. Recent efforts have been made to introduce many-objective search algorithms that can better balance and cover multiple coverage criteria simultaneously [12, 54].

Object-oriented source code metrics have been used for a variety of purposes. For example, Chowdhury et al. used complexity, coupling, and cohesion metrics as early indicators of vulnerabilities [14]. Singh et al. tried to find the relationship between object-oriented metrics and fault-proneness at different fault severity levels [61]. Mansoor et al. used metrics to detect code smells [44]. Cinneide et al. used cohesion and cloning metrics to guide automated refactoring [53]. Tripathi et al. [66] developed models to predict the change-proneness of the classes using code metrics [66]. Relevant to this study, Toth et al. used SourceMeter to gather the same metrics that we used on classes from Java projects on GitHub [65]. They gathered metrics for multiple revisions, focusing on pairs of revisions related to faulty and working versions of the system. They used this dataset for defect prediction. While our purposes differ and there is no overlap in the studied systems, our dataset could potentially be used to augment their study. Recently, Sobreira et al. also assembled a dataset characterizing the faults in Defects4J [62]. Rather than focusing on class characteristics, they focus on the patches used to fix each fault, characterizing them in terms of size, spread, and the repair actions needed to perform automated program repair.

6 | THREATS TO VALIDITY

External Validity: Our study has focused on fifteen systems—a relatively small number. Nevertheless, we believe that such systems are representative of, at minimum, other small to medium-sized open-source Java systems. We believe that Defects4J offers enough fault examples that our results are generalizable to other, sufficiently similar projects.

We have used a single test generation framework. There are many search-based methods of generating tests and these methods may yield different results. Unfortunately, no other generation framework offers the same number and variety of fitness functions. Therefore, a more thorough comparison of tool performance cannot be made at this time. Still, our goal is to examine the coverage criteria, not the generation framework. By using the same framework to generate all test suites, we can compare criteria on an equivalent basis.

To control experiment cost, we have only generated ten test suites for each combination of fault, budget, and fitness function. It is possible that larger sample sizes may yield different results. However, this process still yielded 118,600 test suites to use in analysis. We believe that this is a sufficient number to draw stable conclusions.

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions behind these analyses are met. We have favored non-parametric methods, as distribution characteristics are not generally known a priori, and normality cannot be assumed.

Our learning results are based on a single learning technique. Treatment learning was used to analyze the gathered data, as it is designed to offer succinct, explanatory theories based on classified data [47]—fitting the goal of our work. TAR3 was thought to be appropriate, as it is the most common treatment learning approach and is competitive with other approaches [27].

7 | CONCLUSIONS

We have examined the role of the fitness function in determining the ability of search-based test generators to produce suites that detect complex, real faults. From the eight fitness functions and 593 faults studied, we can conclude:

- Collectively, 51.26% of the examined faults were detected by generated test suites.
- Branch coverage is the most effective criterion—detecting more faults than any other single criterion and demonstrating a higher likelihood of detection for each fault than other criteria (on average, a 22.60-25.24% likelihood of detection, depending on the search budget).
- Regardless of overall performance, most criteria have situational applicability, where their suites detect faults no other criteria can detect. Exception, output, and weak mutation coverage—in particular—seem to be effective for particular types of faults, even if their average efficacy is low.
- While EvoSuite's default combination performs well, the difficulty of simultaneously balancing eight functions prevents it from outperforming all individual criteria.
- However, a combination of branch, exception, and method coverage has an average 24.03-27.84% likelihood of fault detection—outperforming each of the individual criteria. It is more effective than the default eight-way combination because it adds lightweight situationally-applicable criteria to a strong, coverage-focused criterion.
- Factors that strongly indicate a high level of efficacy include high line or branch coverage over either version of the code and high coverage of their own test obligations.
- Coverage does not ensure success, but it is a prerequisite. In situations where achieved coverage is low, the fault does not tend to be found.
- The most important factor differentiating cases where a fault is occasionally detected and cases where a fault is consistently detected is satisfaction of the chosen criterion's test obligations. Therefore, the best suites are ones that both explore the code and fulfill their own goals, which may be—in cases such as exception coverage—orthogonal to code coverage.
- Test generation methods struggle with classes that have a large number of private methods or attributes, and thrive when a large portion of the class structure is accessible.
- Generated suites are more effective at detecting faults in well-documented classes. While the presence of documentation should not directly assist automated test generation, its presence may hint at the maturity, testability, and understandability of the class.
- Faults in classes with a large number of dependencies are more difficult to detect than those in self-contained classes, as the generation technique must initialize and manipulate multiple complex objects during generation.

Theories learned from the collected metrics suggest that successful criteria thoroughly explore and exploit the code being tested. The strongest fitness functions—branch, direct branch, and line coverage—all do so. We suggest the use of such criteria as *primary* fitness functions. However, our findings also indicate that coverage does not guarantee success. The fitness function must still execute the code in a manner that triggers the fault, and ensures that it manifests in a failure. Criteria such as exception, output, and weak mutation coverage are situationally useful, and should be applied as *secondary* testing goals to boost the fault-detection capabilities of the primary criterion—either as part of a multi-objective approach or through the generation of a separate test suite.

Our findings represent a step towards understanding the use, applicability, and combination of common fitness functions. Our observations provide evidence for the anecdotal findings of other researchers [8, 24, 23, 60, 7] and motivate improvements in how test generation techniques understand the behavior of private methods or manipulate environmental dependencies. More research is needed to better understand the factors that contribute to fault detection, and the joint relationship between the fitness function, generation algorithm, and CUT in determining the efficacy of test suites. In future work, we plan to further explore these topics.

8 | ACKNOWLEDGEMENTS

This work is supported by National Science Foundation grant CCF-1657299.

References

[1] Albrecht, A. J. and J. E. Gaffney, 1983: Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, **SE-9**, no. 6, 639–648, doi:10.1109/TSE.1983.235271.

[2] Ali, S., L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, 2010: A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, **36**, no. 6, 742–762.

[3] Almulla, H., A. Salahirad, and G. Gay, 2017: Using search-based test generation to discover real faults in Guava. *Proceedings of the Symposium on Search-Based Software Engineering*, Springer Verlag, SSBSE 2017.

[4] Alshahwan, N. and M. Harman, 2014: Coverage and fault detection of the output-uniqueness test selection criteria. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ACM, New York, NY, USA, ISSTA 2014, 181–192.
URL http://doi.acm.org/10.1145/2610384.2610413

[5] Anand, S., E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, et al., 2013: An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, **86**, no. 8, 1978–2001.

[6] Arcuri, A., 2013: It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability*, **23**, no. 2, 119–147.

[7] Arcuri, A., G. Fraser, and J. P. Galeotti, 2014: Automated unit test generation for classes with environment dependencies. *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ACM, New York, NY, USA, ASE '14, 79–90.
URL http://doi.acm.org/10.1145/2642937.2642986

[8] Arcuri, A., G. Fraser, and R. Just, 2017: Private api access and functional mocking in automated unit test generation. *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 126–137.

[9] Basili, V. R., L. C. Briand, and W. L. Melo, 1996: A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, **22**, no. 10, 751–761, doi:10.1109/32.544352.

[10] Bianchi, L., M. Dorigo, L. Gambardella, and W. Gutjahr, 2009: A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, **8**, no. 2, 239–287, doi:10.1007/s11047-008-9098-4.
URL http://dx.doi.org/10.1007/s11047-008-9098-4

[11] Cai, X. and M. R. Lyu, 2005: The effect of code coverage on fault detection under different testing profiles. *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, ACM, New York, NY, USA, A-MOST '05, 1–7.
URL http://doi.acm.org/10.1145/1082983.1083288

[12] Campos, J., Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, 2018: An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, doi:https://doi.org/10.1016/j.infsof.2018.08.010.
URL http://www.sciencedirect.com/science/article/pii/S0950584917304858

[13] Chidamber, S. R. and C. F. Kemerer, 1991: Towards a metrics suite for object oriented design. *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, ACM, New York, NY, USA, OOPSLA '91, 197–211.
URL http://doi.acm.org/10.1145/117954.117970

[14] Chowdhury, I. and M. Zulkernine, 2011: Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, **57**, no. 3, 294–313.

[15] Dorigo, M. and L. M. Gambardella, 1997: Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, **1**, no. 1, 53–66.

[16] Feldt, R. and S. Poulding, 2015: Broadening the search in search-based software testing: It need not be evolutionary. *Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on*, 1–7.

[17] Frankl, P. G. and O. Iakounenko, 1998: Further empirical studies of test effectiveness. *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, New York, NY, USA, SIGSOFT '98/FSE-6, 153–162.
URL http://doi.acm.org/10.1145/288195.288298

[18] Frankl, P. G. and S. N. Weiss, 1991: An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. *Proceedings of the Symposium on Testing, Analysis, and Verification*, ACM, New York, NY, USA, TAV4, 154–164.
URL http://doi.acm.org/10.1145/120807.120821

[19] — 1993: An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, **19**, no. 8, 774–787, doi:10.1109/32.238581.

[20] Fraser, G. and A. Arcuri, 2013: Whole test suite generation. *Software Engineering, IEEE Transactions on*, **39**, no. 2, 276–291, doi:10.1109/TSE.2012.14.

[21] — 2014: Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, **20**, no. 3, 783–812.

[22] Fraser, G., M. Staats, P. McMinn, A. Arcuri, and F. Padberg, 2013: Does automated white-box test generation really help software testers? *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ACM, New York, NY, USA, ISSTA, 291–301.
URL http://doi.acm.org/10.1145/2483760.2483774

[23] Gay, G., 2016: Challenges in using search-based test generation to identify real faults in mockito. *Search Based Software Engineering: 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*, Springer International Publishing, Cham, 231–237.
URL http://dx.doi.org/10.1007/978-3-319-47106-8_17

[24] — 2017: The fitness function for the job: Search-based generation of test suites that detect real faults. *Proceedings of the International Conference on Software Testing*, IEEE, ICST 2017.

[25] — 2017: Generating effective test suites by combining coverage criteria. *Proceedings of the Symposium on Search-Based Software Engineering*, Springer Verlag, SSBSE 2017.

[26] — 2018: To call, or not to call: Contrasting direct and indirect branch coverage in test generation. *Under submission, International Conference on Software Testing*, IEEE, ICST 2018, draft available from http://greggay.com/pdf/18dbc.pdf.

[27] Gay, G., T. Menzies, M. Davies, and K. Gundy-Burlet, 2010: Automatically finding the control variables for complex system behavior. *Automated Software Engineering*, **17**, no. 4, 439–468, doi:10.1007/s10515-010-0072-x.
URL http://dx.doi.org/10.1007/s10515-010-0072-x

[28] Gay, G., A. Rajan, M. Staats, M. Whalen, and M. P. E. Heimdahl, 2016: The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. *ACM Trans. Softw. Eng. Methodol.*, **25**, no. 3, 25:1–25:34, doi:10.1145/2934672.
URL http://doi.acm.org/10.1145/2934672

[29] Gay, G., M. Staats, M. Whalen, and M. Heimdahl, 2015: The risks of coverage-directed test case generation. *Software Engineering, IEEE Transactions on*, **PP**, no. 99, doi:10.1109/TSE.2015.2421011.

[30] Gligoric, M., A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, 2013: Comparing non-adequate test suites using coverage criteria. *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ACM, New York, NY, USA, ISSTA 2013, 302–313.
URL http://doi.acm.org/10.1145/2483760.2483769

[31] Gopinath, R., C. Jensen, and A. Groce, 2014: Mutations: How close are they to real faults? *25th International Symposium on Software Reliability Engineering*, 189–200.

[32] Groce, A., M. A. Alipour, and R. Gopinath, 2014: Coverage and its discontents. *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ACM, New York, NY, USA, Onward!'14, 255–268.
URL http://doi.acm.org/10.1145/2661136.2661157

[33] Gui, G. and P. D. Scott, 2006: Coupling and cohesion measures for evaluation of component reusability. *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ACM, New York, NY, USA, MSR '06, 18–21.
URL http://doi.acm.org/10.1145/1137983.1137989

[34] Harman, M. and B. Jones, 2001: Search-based software engineering. *Journal of Information and Software Technology*, **43**, 833–839.

[35] Holland, J. H., 1992: *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT press.

[36] Inozemtseva, L. and R. Holmes, 2014: Coverage is not strongly correlated with test suite effectiveness. *Proceedings of the 36th International Conference on Software Engineering*, ACM, New York, NY, USA, ICSE 2014, 435–445.
URL http://doi.acm.org/10.1145/2568225.2568271

[37] Jeffrey, D. and N. Gupta, 2007: Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering*, **33**, no. 2, 108–123, doi:10.1109/TSE.2007.18.

[38] Just, R., 2014: The major mutation framework: Efficient and scalable mutation analysis for java. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ACM, New York, NY, USA, ISSTA 2014, 433–436.
URL http://doi.acm.org/10.1145/2610384.2628053

[39] Just, R., D. Jalali, and M. D. Ernst, 2014: Defects4J: A database of existing faults to enable controlled testing studies for Java programs. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ACM, New York, NY, USA, ISSTA 2014, 437–440.
URL http://doi.acm.org/10.1145/2610384.2628055

[40] Kit, E. and S. Finzi, 1995: *Software Testing in the Real World: Improving the Process*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

[41] Lakhotia, K., M. Harman, and P. McMinn, 2007: A multi-objective approach to search-based test data generation. *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ACM, New York, NY, USA, GECCO '07, 1098–1105.
URL http://doi.acm.org/10.1145/1276958.1277175

[42] Linda, P. E., V. M. Bashini, and S. Gomathi, 2011: Metrics for component based measurement tools. *International Journal of Scientific & Engineering Research Volume 2, Issue 5*.

[43] Malburg, J. and G. Fraser, 2011: Combining search-based and constraint-based testing. *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society, Washington, DC, USA, ASE '11, 436–439.
URL http://dx.doi.org/10.1109/ASE.2011.6100092

[44] Mansoor, U., M. Kessentini, B. R. Maxim, and K. Deb, 2017: Multi-objective code-smells detection using good and bad design examples. *Software Quality Journal*, **25**, no. 2, 529–552.

[45] McMinn, P., 2004: Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, **14**, 105–156.

[46] McMinn, P., M. Harman, G. Fraser, and G. M. Kapfhammer, 2016: Automated search for good coverage criteria: Moving from code coverage to fault coverage through search-based software engineering. *Proceedings of the 9th International Workshop on Search-Based Software Testing*, ACM, New York, NY, USA, SBST '16, 43–44.
URL http://doi.acm.org/10.1145/2897010.2897013

[47] Menzies, T. and Y. Hu, 2003: Data mining for very busy people. *Computer*, **36**, no. 11, 22–29, doi:10.1109/MC.2003.1244531.
URL http://dx.doi.org/10.1109/MC.2003.1244531

[48] Mockus, A., N. Nagappan, and T. Dinh-Trong, 2009: Test coverage and post-verification defects: A multiple case study. *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, 291–301.

[49] Mohri, M., A. Rostamizadeh, and A. Talwalkar, 2012: *Foundations of Machine Learning*. MIT Press.

[50] Molokken, K. and M. Jorgensen, 2003: A review of software surveys on software effort estimation. *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.*, 223–230.

[51] Myers, G. J. and C. Sandler, 2004: *The Art of Software Testing*. John Wiley & Sons.

[52] Namin, A. and J. Andrews, 2009: *The influence of size and coverage on test suite effectiveness*.

[53] Ó Cinnéide, M., L. Tratt, M. Harman, S. Counsell, and I. Hemati Moghadam, 2012: Experimental assessment of software metrics using automated refactoring. *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ACM, 49–58.

[54] Panichella, A., F. M. Kifetew, and P. Tonella, 2018: Incremental control dependency frontier exploration for many-criteria test case generation. *Search-Based Software Engineering*, T. E. Colanzi and P. McMinn, Eds., Springer International Publishing, Cham, 309–324.

[55] Perry, W., 2006: *Effective Methods for Software Testing, Third Edition*. John Wiley & Sons, Inc., New York, NY, USA.

[56] Pezze, M. and M. Young, 2006: *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons.

[57] Rayadurgam, S. and M. Heimdahl, 2001: Coverage based test-case generation using model checkers. *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*, IEEE Computer Society, 83–91.

[58] Rojas, J. M., J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, 2015: Combining multiple coverage criteria in search-based unit test generation. *Search-Based Software Engineering*, M. Barros and Y. Labiche, Eds., Springer International Publishing, volume 9275 of *Lecture Notes in Computer Science*, 93–108.
URL http://dx.doi.org/10.1007/978-3-319-22183-0_7

[59] Roy, C. K. and J. R. Cordy, 2007: A survey on software clone detection research.

[60] Shamshiri, S., R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, 2015: Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, New York, NY, USA, ASE 2015.

[61] Singh, Y., A. Kaur, and R. Malhotra, 2010: Empirical validation of object-oriented metrics for predicting fault proneness models. *Software quality journal*, **18**, no. 1, 3.

[62] Sobreira, V., T. Durieux, F. Madeiral, M. Monperrus, and M. A. Maia, 2018: Dissection of a bug dataset: Anatomy of 395 patches from defects4j. *arXiv preprint arXiv:1801.06393*.

[63] SourceMeter, 2014: *Sourcemeter java documentation*. https://www.sourcemeter.com/resources/java/.

[64] Tóth, Z., P. Gyimesi, and R. Ferenc, 2016: A public bug database of github projects and its application in bug prediction. *Computational Science and Its Applications – ICCSA 2016*, O. Gervasi, B. Murgante, S. Misra, A. M. A. Rocha, C. M. Torre, D. Taniar, B. O. Apduhan, E. Stankova, and S. Wang, Eds., Springer International Publishing, Cham, 625–638.

[65] — 2016: A public bug database of github projects and its application in bug prediction. *Computational Science and Its Applications – ICCSA 2016*, O. Gervasi, B. Murgante, S. Misra, A. M. A. Rocha, C. M. Torre, D. Taniar, B. O. Apduhan, E. Stankova, and S. Wang, Eds., Springer International Publishing, Cham, 625–638.

[66] Tripathi, A. and K. Sharma, 2015: Improving software quality based on relationship among the change proneness and object oriented metrics. *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on*, IEEE, 1633–1636.

[67] Whalen, M., G. Gay, D. You, M. Heimdahl, and M. Staats, 2013: Observable modified condition/decision coverage. *Proceedings of the 2013 Int'l Conf. on Software Engineering*, ACM.

[68] Yoo, S. and M. Harman, 2010: Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software*, **83**, no. 4, 689 – 701, doi:http://dx.doi.org/10.1016/j.jss.2009.11.706.
URL http://www.sciencedirect.com/science/article/pii/S0164121209003069

[69] Yu, Z., M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, 2017: Test case generation for program repair: A study of feasibility and effectiveness. *CoRR*, **abs/1703.00198**.
URL http://arxiv.org/abs/1703.00198

How to cite this article: Alireza Salahirad, Hussein Almulla, and Gregory Gay (2017), Choosing The Fitness Function for the Job: Automated Generation of Test Suites that Detect Real Faults, *Software Testing, Verification and Reliability, 2018;00:1–XX*.