The Fitness Function for the Job: Search-Based Generation of Test Suites that Detect Real Faults

Gregory Gay
Department of Computer Science & Engineering
University of South Carolina, USA
greg@greggay.com

Abstract— **Search-based 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 the EvoSuite framework and eight of its fitness functions on 353 real faults from the Defects4J database. Our analysis has found that the strongest indicator of effectiveness is a high level of code coverage. 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 vary the scenarios explored.

I. 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 [32], 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. We believe the key to lowering such costs lies in the use of automation to ease this manual burden [3]. 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 [21].

Test case generation can naturally be seen as a search problem [3]. 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 [28].

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, such as the execution of branches in the control-flow of the CUT [26], [34], [35] 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 [1]. "Finding faults" is not a goal that can be measured, and cannot be cleanly 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 search-based testing literature reveals 28 viable fitness functions—and there is little guidance on when to use one criterion over another [16].

While previous studies on the effectiveness of adequacy criteria have yielded inconclusive results [33], [30], [23], [16], two factors now 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 in its default configuration [11]. 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 [24]. Whether mutants correspond to the types of faults found in real projects has not been firmly established [19]. However, the Defects4J project offers a large database of real faults extracted from open-source Java projects [25]. We can use these faults to assess the effectiveness of search-based generation on the complex faults found in real software.

We have used EvoSuite and eight of its fitness functions (as well as the default multi-objective configuration) to generate test suites for the five systems, and 353 of the faults, in the Defects4J database. In each case, we recorded the proportion of suites that detect the fault and a number of factors—related to suite size, obligation satisfaction, and attained coverage. By analyzing these factors, we can begin to understand not only the real-world applicability of the fitness options in EvoSuite, but—through the use of learning algorithms to build *theories*—the factors indicating a high likelihood of fault detection. To summarize our findings:

- Collectively, 51.84% of the examined faults were detected by generated test suites.
- EvoSuite's branch coverage criterion is the most effective—detecting more faults than any other criterion and demonstrating a 11.06-245.31% higher likelihood of detection for each fault than other criteria.
- There is evidence that combinations of criteria could be more effective than a single criterion—almost all criteria uniquely detect one or more faults, and in cases where the top-scoring criterion performs poorly, at least one other criterion would more capably detect the fault.
- Yet, while EvoSuite's default combination performs well, the difficulty of simultaneously balancing eight functions prevents it from outperforming all criteria.
- High levels of coverage over the fixed version of the CUT and over patched lines of code are the factors that most strongly predict suite effectiveness.
- While others have found that test suite size correlates to mutant detection, we found that larger suites are not necessarily more effective at detecting real faults.

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, line, and weak mutation 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. Almost all of the criteria were useful in *some* case and could 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. 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. However, our findings represent a step towards understanding the use, applicability, and combination of common fitness functions.

II. Background

A. Search-Based Software Test Generation

Test case creation can naturally be seen as a search problem [21]. 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 [28], [1]. 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*) [5]. Metaheuristics are often inspired by natural phenomena, such as swarm behaviors [7] or species evolution [22].

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 [1]. Metaheuristic search—by strategically sampling from that space—can scale to larger problems many other generation algorithms [27]. Such approaches have been applied to a wide variety of testing goals and scenarios [1].

B. 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 [26], [34], [35]. Each adequacy criterion embodies a set of lessons about effective testing—requirements tests must fulfill to be considered adequate. If tests execute elements in the manner prescribed by the criterion, than testing is deemed "adequate" with respect to faults that manifest through such structures. Adequacy criteria have seen widespread use in software development, and is routinely measured as part of automated build processes [20][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 [36]. These very same qualities make adequacy criteria ideal for use as automated test generation targets, as they can be straightforwardly transformed into distance functions that guide to the search to better solutions [4]. Search-based generation have even managed to achieve higher coverage than developer-created tests [13].

III. Study

To generate tests that are effective at finding faults, then 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

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

and fault detection [20]. Many criteria exist, and there is little guidance on when to use one over another [16]. 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 353 real faults, contained in the Defects4J database. In doing so, we wish to address the following research questions:

1) Are generated test suites able to detect 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?

These 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. Therefore, we also ask:

4) Are there situations where a combination of criteria could outperform a single criterion?
5) Does EvoSuite's default configuration—a combination of eight criteria—outperform any single criterion?

Finally, across all criteria, we also would like to answer:

6) What factors predict a high fault detection likelihood?

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 III-A).
2) **Generated Test Cases:** For each fault, we generated 10 suites per criterion, as well as EvoSuite's default configuration, using the fixed version of each CUT. We performed with both a two-minute and a ten-minute search budget per CUT (Section III-B).
3) **Removed Non-Compiling and Flaky Tests:** Any tests that do not compile, or that return inconsistent results, are automatically removed (Section III-B).
4) **Assessed Fault-finding Effectiveness:** For each fault, we measure the proportion of test suites that detect the fault to the number generated (Section III-C).
5) **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 III-C).

A. Case Examples

Defects4J is an extensible database of real faults extracted from Java projects [25][2]. Currently, it consists of 357 faults from five projects: JFreeChart (26 faults), Closure compiler (133 faults), Apache Commons Lang (65 faults), Apache Commons Math (106 faults), and JodaTime (27 faults). Four faults from the Math project were omitted due to complications encountered during suite generation, leaving 353 that we used in our study.

[2]Available from `http://defects4j.org`

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. For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose the faults, and a list of classes and lines of code modified by the patch that fixes the fault.

B. Test Suite Generation

The EvoSuite framework applies a genetic algorithm in order 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 [38]. In this study, we used EvoSuite version 1.0.3, and the following fitness functions:

Branch Coverage (BC): A test suite satisfies branch coverage if all control-flow branches are taken by at least one test case—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 [4].

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

Method	Budget	Total Obligations	% Obligations Satisfied	Suite Size	Suite Length	# Tests Removed	% LC (Fixed)	% LC (Faulty)	% BC (Fixed)	% BC (Faulty)	% Patch Coverage
Default	120	1834.43	50.50%	44.06	360.03	0.25	50.61%	49.82%	41.00%	40.00%	45.51%
	600		57.59%	58.50	563.74	0.47	55.41%	53.85%	47.00%	46.00%	49.94%
Branch Coverage (BC)	120	327.91	54.28%	36.35	225.33	0.32	56.44%	55.78%	48.00%	47.00%	50.31%
	600		61.58%	43.71	305.13	0.46	61.16%	60.30%	54.00%	53.00%	54.04%
Direct Branch (DBC)	120	327.91	49.97%	37.31	244.36	0.26	52.29%	51.80%	44.00%	43.00%	45.96%
	600		57.66%	47.10	347.13	0.43	56.11%	55.11%	49.00%	48.00%	50.02%
Exception Coverage (EC)	120	11.08	99.48%	11.03	28.18	0.07	20.89%	20.91%	11.00%	11.00%	16.41%
	600		99.40%	11.04	28.34	0.07	21.10%	21.08%	11.00%	11.00%	16.92%
Line Coverage (LC)	120	340.56	58.28%	30.64	186.78	0.25	56.91%	56.37%	44.00%	43.00%	50.25%
	600		63.75%	34.15	232.10	0.31	61.27%	60.12%	49.00%	48.00%	53.59%
Method Coverage (MC)	120	31.12	79.56%	21.82	72.33	0.06	36.09%	36.33%	21.00%	21.00%	29.39%
	600		84.48%	24.52	87.06	0.07	37.62%	37.81%	22.00%	22.00%	30.87%
Method, No Exception (MNEC)	120	31.12	77.96%	21.60	71.61	0.07	37.33%	37.41%	22.00%	22.00%	31.18%
	600		83.35%	24.00	89.32	0.07	39.29%	39.32%	23.00%	23.00%	32.89%
Output Coverage (OC)	120	205.71	46.66%	31.02	153.89	0.17	37.55%	37.15%	27.00%	27.00%	34.21%
	600		51.98%	36.86	197.23	0.20	39.85%	39.49%	29.00%	28.00%	36.18%
Weak Mutation (WMC)	120	560.04	52.57%	28.42	198.53	0.15	51.02%	50.76%	41.00%	41.00%	45.68%
	600		59.30%	35.32	300.84	0.27	56.25%	55.59%	48.00%	47.00%	50.93%

TABLE I

STATISTICS ON GENERATED TEST SUITES. VALUES ARE AVERAGED OVER ALL 353 FAULTS. LC=LINE COVERAGE, BC=BRANCH COVERAGE AS MEASURED BY COBERTURA. PATCH COVERAGE IS LINE COVERAGE OVER THE LINES ALTERED BY THE PATCH THAT FIXES THE FAULT.

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 [2]. 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.

Weak Mutation Coverage (WMC): Test effectiveness is often judged using mutants [24]. 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 [12].

Rojas et al. provide a primer on each of these fitness functions [37]. We have also used EvoSuite's default configuration—a combination of all of the above methods—to generate test suites. Test suites are generated that target the classes reported as relevant to the fault by Defects4J. Tests are generated from the fixed version of the CUT and applied to the faulty version in order to eliminate the oracle problem. In practice, this translates to a regression testing scenario, where tests guard against future issues.

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 63,540 test suites (two budgets, ten trials, nine configurations, 353 faults).

Generation tools may generate flaky (unstable) tests [38]. 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, less than one percent of the tests are removed from each suite (see Table I).

C. 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.

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

Number of Test Obligations: Given a CUT, each fitness function will calculate a series of test obligations—goals—to cover. The number of obligations is informative of the difficulty of the generation, and impacts the size and formulation of tests [15].

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 often thought to be more effective [17], [23]. Even if two suites achieve the same

	Budget	Chart	Closure	Lang	Math	Time	Total
Default	120	15	17	29	48	14	123
	600	17	20	35	57	14	143
	Total	18	22	36	59	16	151
BC	120	17	16	36	53	16	138
	600	20	19	35	54	17	145
	Total	21	21	41	57	18	158
DBC	120	14	16	32	48	15	125
	600	19	19	36	47	18	139
	Total	19	22	40	52	18	151
EC	120	8	7	12	13	6	46
	600	10	5	13	12	5	45
	Total	10	8	15	15	6	54
LC	120	15	12	31	50	15	123
	600	18	14	32	52	14	130
	Total	18	17	37	55	15	142
MC	120	10	6	9	25	5	55
	600	10	10	11	24	5	45
	Total	12	10	14	27	6	69
MNEC	120	9	8	10	29	5	61
	600	11	6	13	27	3	60
	Total	11	9	13	32	6	71
OC	120	9	7	13	36	5	70
	600	13	9	17	33	6	78
	Total	13	12	18	38	9	89
WMC	120	13	15	31	42	14	115
	600	18	19	32	48	14	131
	total	18	22	37	51	16	143
Any criterion	120	18	23	44	61	16	162
	600	23	31	34	63	18	180
	Total	23	32	46	64	18	183

TABLE II
NUMBER OF FAULTS DETECTED BY EACH FITNESS FUNCTION.

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.

Code Coverage: As the premise of many adequacy criteria and their corresponding fitness functions 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 the Cobertura tool[3], we have measured the line and branch coverage achieved by each suite over both the faulty and fixed versions of each CUT.

Patch Coverage: A high level of coverage does not necessarily indicate that the lines relevant to the fault are covered. We also record line coverage over the program statements modified by the patch that fixes the fault—the lines of code that differ between the faulty and fixed version.

Table I records, for each fitness function and budget, the average values attained for each of these measurements over all faults. Note that the number of test obligations is dependent on the CUT, and does not differ between budgets.

IV. RESULTS & DISCUSSION

In Table II, 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*. Due to the stochastic search, a higher budget does not guarantee detection of the same faults found under a lower search budget. Therefore, we also list the total number of faults detected by each fitness function.

[3]Available from `http://cobertura.github.io/cobertura/`

Function	Number of Faults
Default	3
BC	6
DBC	3
EC	2
LC	0
MC	1
MNEC	1
OC	1
WC	3

TABLE III
NUMBER OF FAULTS UNIQUELY DETECTED BY EACH FITNESS FUNCTION.

The criteria are capable of detecting 183 (51.84%) of the 353 studied faults.

While there is clearly room for improvement, these results are encouraging. Generated tests are able to detect a variety of complex, real-world faults. In the following subsections, we will assess the results of our study with respect to each research question. In Section IV-A, we compare the capabilities of each fitness function. In Section IV-B, we explore combinations of criteria. Finally, in Section IV-C, we explore the factors that indicate effectiveness.

A. *Comparing Fitness Functions*

From Table II, we can see that suites differ in effectiveness between criteria. Overall, branch coverage produces the best suites, detecting 158 faults. Branch is closely followed by direct branch (151 faults), weak mutation (143), and line coverage (142). These four fitness functions are trailed by the other four, with exception coverage showing the weakest results (54 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 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.

Table III depicts the number of faults uniquely detected by each fitness function. A total of twenty faults can only be detected by a single criterion. Branch coverage is again the most effective in this regard, uniquely detecting six faults. Direct branch coverage and weak mutation also perform well, each detecting three faults that no other criterion can detect.

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 the likelihood of detection. To do so, we record the proportion of detecting suites to the total number of suites generated for

	Budget	Chart	Closure	Lang	Math	Time	Total
Default	120	47.31%	4.51%	23.85%	25.78%	25.93%	19.01%
	600	48.08%	7.07%	32.66%	32.84%	33.33%	24.25%
	% Change	1.62%	56.67%	36.77%	27.38%	28.57%	27.57%
BC	120	45.00%	4.66%	34.00%	27.94%	34.82%	22.07%
	600	48.46%	5.79%	40.15%	32.75%	39.26%	25.61%
	% Change	7.69%	24.19%	18.10%	17.19%	12.77%	16.05%
DBC	120	34.23%	5.11%	30.00%	24.51%	31.11%	19.43%
	600	40.77%	6.09%	38.77%	28.63%	40.37%	23.80%
	% Change	19.10%	19.12%	29.23%	16.80%	29.76%	22.45%
EC	120	22.31%	1.35%	7.54%	6.37%	9.26%	6.09%
	600	21.54%	0.98%	9.23%	7.06%	9.63%	6.43%
	% Change	-3.45%	-27.78%	22.45%	10.77%	4.00%	5.58%
LC	120	38.85%	4.14%	31.23%	25.78%	30.00%	19.92%
	600	46.15%	4.81%	34.31%	29.22%	36.67%	22.78%
	% Change	18.81%	16.36%	9.85%	13.31%	22.22%	14.37%
MC	120	30.77%	1.58%	7.54%	10.98%	8.15%	8.05%
	600	30.77%	2.26%	7.69%	10.88%	8.15%	8.30%
	% Change	0.00%	42.86%	2.04%	-0.89%	0.00%	3.17%
MNEC	120	23.46%	2.18%	6.62%	12.16%	6.67%	7.79%
	600	30.77%	1.88%	7.54%	12.06%	5.19%	8.24%
	% Change	31.15%	-13.79%	13.95%	-0.81%	-22.22%	5.82%
OC	120	21.15%	2.03%	7.85%	16.57%	9.63%	9.29%
	600	23.85%	2.56%	10.92%	16.76%	12.22%	10.51%
	% Change	12.73%	25.93%	39.22%	1.18%	26.92%	13.11%
WMC	120	38.08%	4.44%	24.15%	23.04%	25.19%	17.51%
	600	46.15%	5.56%	32.15%	27.45%	27.04%	21.42%
	% Change	21.21%	25.42%	33.12%	19.15%	7.35%	22.33%

TABLE IV

AVERAGE LIKELIHOOD OF FAULT DETECTION, BROKEN DOWN BY FITNESS FUNCTION, BUDGET, AND SYSTEM.

that fault. The average likelihood of fault detection is listed for each criterion, by system and budget, in Table IV.

We largely observe the same trends as above. Branch coverage has the highest overall likelihood of fault detection, with 22.07% of suites detecting faults given a two-minute search budget and 25.61% of suites detecting faults given a ten-minute budget. Line and direct branch coverage follow with a 19.92-22.78% and 19.43-23.80% 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.

Branch coverage is the most effective criterion, detecting 158 faults—six of which were only detected by this criterion. Branch coverage suites have, on average, a 22.07-25.61% likelihood of fault detection.

From Table IV, 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 22.45% and 22.33% in effectiveness. In general, all distance-driven criteria—branch, direct branch, line, weak mutation, and, partially, output coverage—improve given more time. Discrete fitness functions benefit less from an increased search budget.

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

	Default	BC	DBC	LC	WM
Default	-	0.89	0.55	0.54	0.25
BC	0.11	-	0.13	0.13	**0.03**
DBC	0.45	0.87	-	0.50	0.22
LC	0.46	0.87	0.50	-	0.22
WC	0.75	0.97	0.78	0.78	-

TABLE V

P-VALUES FOR MANN-WHITNEY-WILCOXON COMPARISONS OF "TOP-CLUSTER" CRITERIA (TWO-MINUTE SEARCH BUDGET).

We can perform statistical analysis to assess our observations. For each pair of criteria, we formulate hypotheses:

- H_1: Given a fixed search budget, test suites generated using criterion *A* will have a higher likelihood of fault detection than suites generated using criterion *B*.
- $H0_1$: 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 $H0_1$ without any assumptions on distribution, we use a one-sided (strictly greater) Mann-Whitney-Wilcoxon rank-sum test [40], a non-parametric hypothesis test for determining if one set of observations is drawn from a different distribution than another set of observations. Due to the limited number of faults for the Chart and Time 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 $\alpha = 0.05$.

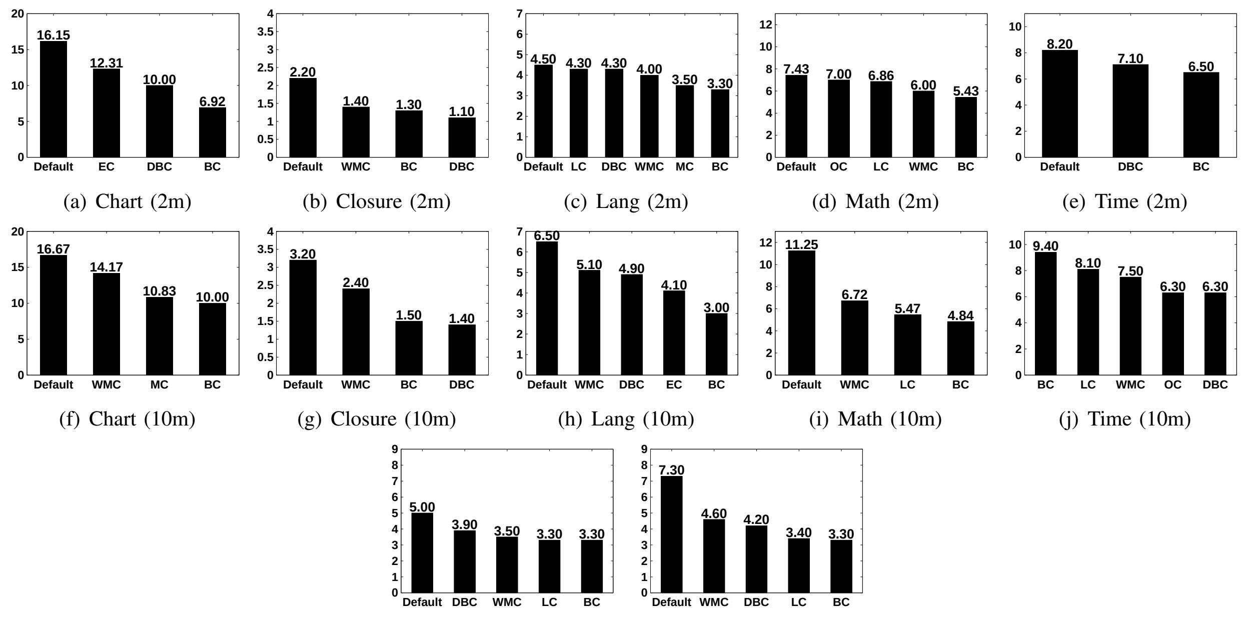
The tests further validate the "two clusters" observation. For the four criteria in the top cluster—branch, direct branch, line, and weak mutation coverage—we can always reject the null hypothesis with regard to the remaining four criteria in the bottom cluster. Within each cluster, we generally cannot reject the null hypothesis. P-values within the top cluster, when a two-minute search budget is used, are shown in Table V. We can use these results to infer a partial ordering—Branch coverage outperforms weak mutation coverage with significance. While there were no other cases where $H0_1$ can be rejected, p-values are much lower for branch coverage against the other criteria than in any other pairing[4]. This suggests a slight advantage in using branch coverage, consistent with previous results. Similarly, in the lower cluster, we can reject $H0_1$ for output coverage against exception coverage (two-minute budget) and against exception, method, and method (no exception) with a ten-minute budget. We cannot reject $H0_1$ in the other comparisons.

Branch coverage outperforms, with statistical significance, weak mutation (2m budget), and method, MNEC, output, and exception coverage (both budgets).

B. Combinations of Fitness Functions

The analysis above presupposes that only one fitness function can be used to generate test suites. However, many

[4]P-values for a ten-minute budget are omitted due to space constraints, but suggest similar conclusions.



(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)

(k) Overall (2m) (l) Overall (10m)

Fig. 1. 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.

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. In theory, suites generated through a combination of fitness functions could be more effective than suites generated through any one objective. 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. To better understand the potential of combined criteria, we answer two questions. First, are there situations where the most effective criterion is outperformed by another, secondary, criterion? Second, does EvoSuite's default combination outperform the individual criteria?

From Table III, we can see that there are a total of twenty faults only detected by a single configuration. Thus, it is clear that no one fitness function can detect all faults. Almost all criteria—regardless of their overall effectiveness—can detect something the others cannot. This does not automatically mean that combinations of criteria can detect these faults either—the default configuration does not detect the 17 faults detected by the other individual criteria. It does, however, detect three faults not found otherwise. Still, combinations of criteria, either explored over multiple, independent generations—each for a single criterion—or through a single multi-objective generation, may be able to detect faults missed by a single criterion.

To further understand the situations where combinations of criteria may be useful, we filter the set of faults for those that the top-scoring criterion is ineffective at detecting. In Figure 1, we have taken the faults for each system, isolated any where the "best" criterion for that system (generally branch coverage, see Table IV) 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 likelihood of fault detection for any criterion that outperforms the best from the full set.

From these plots, we can see that there are always 2-5 criterion that are more effective in these situations. The exact criteria depend strongly on the system, and likely, on the types of faults examined. 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. Despite its poor overall performance, exception coverage also often does well in these situations—demonstrating the wisdom of favoring suites that throw more exceptions.

These results imply that a combination of criteria could outperform a single criterion. Indeed, from Figure 1, we can see that EvoSuite's default configuration outperforms all other criteria for the examined subsets of faults. In situations where the single "best" criterion performs poorly, a multi-objective solution achieves improved results.

In situations where the criterion that is the most effective overall has $< 30\%$ likelihood of fault detection, other criteria and combinations of criteria more effectively detect the fault. The effective secondary criteria vary by system.

Overall, EvoSuite's default configuration performs well, but fails to outperform all individual criteria. Table II shows that the default configuration detects 151 faults—fewer than branch coverage, but a tie with direct branch coverage. As previously mentioned, it also uniquely detects three faults (see Table III). Again, this is fewer than branch coverage, but tied with direct branch and weak mutation coverage. According to Table IV, the default configuration's average overall likelihood of fault detection is 19.01% (2m budget)-24.25% (10m budget). At the two-minute level, this places it below branch, line, 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 further statistical tests.

In theory, a combination of criteria could detect more faults than any single criterion. In practice, combining *all* criteria results in suites that are quite effective, but fail to reliably outperform individual criterion. From Table IV, we can see that the performance of the default configuration also varies quite a bit between systems, and by search budget. For Chart and Math, the default configuration performs almost as well as branch coverage. With a higher budget, it performs as well or better than Branch for Math and Closure. However, for the Lang and Time systems, the default configuration is less effective than branch, line, and direct branch coverage even with a larger budget.

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 I, the default configuration must attempt to satisfy, on average, 1,834 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 27.57% improvement in efficacy.

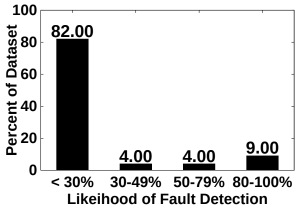
EvoSuite's default configuration has an average 19.01-24.25% likelihood of fault detection—in the top cluster, but failing to outperform all individual criteria.

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 subset of criteria to combine during test generation. More research is needed to understand which combinations are most effective, and whether system-specific combinations may yield further improvements.

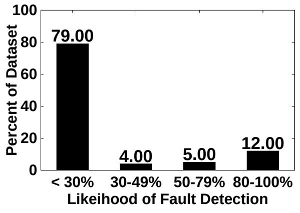
C. Understanding the Factors Leading to Fault Detection

As discussed in Section III-C—to better understand the combination of factors leading to 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, branch (BC) and line coverage (LC) over the fixed and faulty versions of the CUT, and coverage of the lines changed to fix the fault (patch coverage, or PC).

This collection of factors forms a dataset where, for each fault, we have recorded the average of each statistic for the suites generated to detect that fault. We can then use



(a) 2m Budget



(b) 10m Budget

Fig. 2. Baseline class distribution of the dataset used for treatment learning. Likelihood of fault detection is discretized into four classes—our target is the 80-100% class.

the likelihood of fault detection (D) as the class variable—discretized into four values: $D < 30\%$, $30 \leq D < 50$, $50 \leq D < 80$, $D \geq 80$. We have created two datasets, dividing data by search budget. The class distribution of each dataset is shown in Figure 2.

A standard practice in machine learning is to *classify data*—to use previous experience to categorize new observations [31]. 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 factors correspond most strongly to a class of interest*—a process known as treatment learning [29]. Treatment learning approaches take the classification of an observation and reverse engineer the evidence that led to that categorization. Such learners produce a *treatment*—a small set of attributes and value ranges that, if imposed, will identify a subset of the original data matching skewed towards the target classification. In this case, a treatment notes the factors—and their values—that contribute to a high likelihood of fault detection. Using the TAR3 treatment learner [14], we have generated five treatments from each dataset. The treatments are scored according to their impact on class distribution, and top-scoring treatments are presented first. We extracted the following treatments:

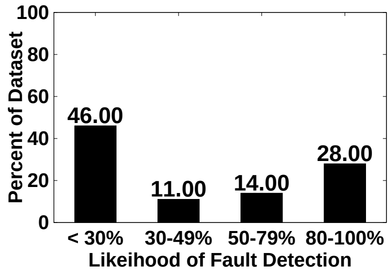
Two-Minute Dataset:

1) BC (fixed) $>$ 67.70%, PC $>$ 66.67%
2) LC (fixed) $>$ 82.19%, PC $>$ 66.67%, BC (fixed) $>$ 67.70%
3) PC $>$ 66.67%, LC (fixed) $>$ 82.19%
4) 69.40 $\geq$ Length $\leq$ 169.80, BC (fixed) $>$ 67.70%
5) BC (fixed) $>$ 67.70%, % of obligations satisfied $>$ 86.38%

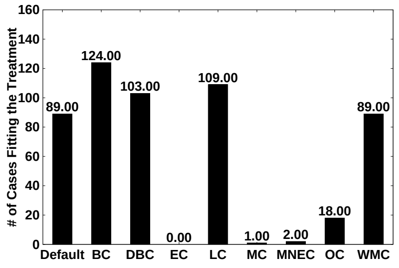
Ten-Minute Dataset:

1) BC (fixed) $>$ 76.08%, PC $>$ 70.00%
2) PC $>$ 70.00%, LC (fixed) $>$ 85.92%
3) BC (fixed) $>$ 76.08%, % of obligations satisfied $>$ 93.30%
4) BC (fixed) $>$ 76.08%, LC (fixed) $>$ 85.92%,% of obligations satisfied $>$ 93.30%
5) PC $>$ 70.00%, LC (fixed) $>$ 85.92%, BC (fixed) $>$ 76.08%

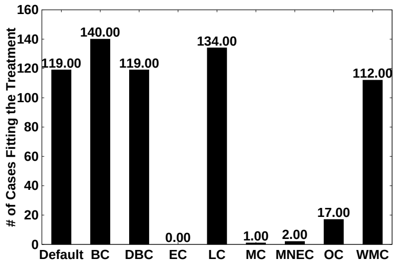
In Figure 3(a), we plot the class distribution of the subset fitting the highest-ranked treatment learned from the two-minute dataset. In Figure 4, we do the same for the top treatment from the ten-minute dataset. Comparing the plots in Figure 2 to the subsets in Figures 3-4, we can see that the treatments do impose a large change in the class distribution—a lower percentage of cases have the $< 30\%$ classification, and more have the other classifications,



(a) Class Distribution

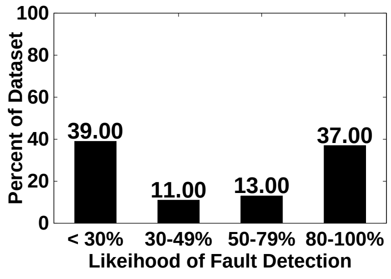


(b) # of Cases Fitting the Treatment (2m)

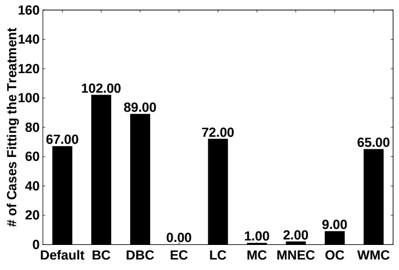


(c) # of Cases Fitting the Treatment (10m)

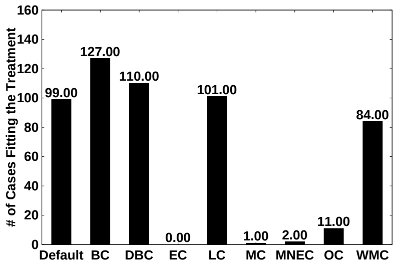
Fig. 3. For the top treatment learned from the 2m dataset: Class distribution of the data subset fitting the treatment, the number of cases that fit the treatment for each fitness function from the 2m dataset, and the number of cases that fit the treatment for each fitness function from the 10m dataset.



(a) Class Distribution



(b) # of Cases Fitting the Treatment (2m)



(c) # of Cases Fitting the Treatment (10m)

Fig. 4. For the top treatment learned from the 2m dataset: Class distribution of the data subset fitting the treatment, the number of cases that fit the treatment for each fitness function from the 2m dataset, and the number of cases that fit the treatment for each fitness function from the 10m dataset.

particularly $80 - 100\%$. 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 more likely to be.

We can make several observations. First, the most common factors selected as indicative of efficacy are all coverage-related factors. For both datasets, the top-ranked treatment specifically indicates that branch coverage over the fixed version of the CUT and patch coverage are the most important factors in determining suite efficacy. 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. Coverage is recommended by all of the selected treatments—generally over the fixed version of the CUT. While patch coverage is important, overall branch and line coverage over the faulty version is less important than coverage over the fixed version. It seems to be important that the suite thoroughly explore the program is it generated on, and that it still covers lines patched in the faulty version.

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 [23]. However, a size attribute (suite length) only appears in one of the treatments produced for either dataset. This seems to indicate that, unlike with mutants, larger test suites are not necessarily more effective at detecting real faults.

The other factor noted as indicative of efficacy is the percent of obligations satisfied. For coverage-based fitness functions like branch and line coverage, a high level of satisfied obligations likely correlates with a high level of branch or line coverage. For other fitness functions, the correlation may not be as strong, but it is likely that suites satisfying more of their obligations also explore more of the structure of the CUT.

Factors that strongly indicate efficacy include a high level of branch coverage over the fixed CUT and patch coverage. Coverage is favored over factors related to suite size or test obligations.

We have recorded how effective each fitness function is at producing suites that meet the conditions indicated by the top-ranked treatments learned from each dataset in Figures 3(b) and (c) and 4(b) and (c). From these plots, we can immediately see that the top cluster of fitness functions met those conditions for a large number of faults. The bottom cluster, on the other hand, rarely meet those conditions.

Note, however, that we still do not entirely understand the factors that indicate a high probability of fault detection. From Figures 3-4, we can see that the treatments radically alter the class distribution from the baseline in Figure 2. Still, suites fitting the top-ranked treatments are ineffective as often as they are highly effective. 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. 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 [39]) or combinations of fitness functions will better guide such a search. Further research is needed to better understand how to ensure a high probability of fault detection.

While coverage increases the likelihood of fault detection, it does not ensure that suites are effective.

V. 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 [20]. Researchers have attempted to address whether such a correlation exists for almost as long as such criteria have existed [33], [30], [6], [9], [10], [8], [18], [23], [16]. Inozemtseva et al. provide a good overview of work in this area [23]. 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.

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 [38]. 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. In our study, we have expanded the number of EvoSuite configurations to better understand the role of the fitness function in determining suite efficacy.

VI. Threats to Validity

External Validity: Our study has focused on a relatively small number of systems. Nevertheless, we believe that such systems are representative of—at minimum—other small to medium-sized open-source Java systems. We also 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.

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 still yielded 64,360 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 apriori, and normality cannot be assumed.

VII. 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 353 faults studied, we can conclude:

- EvoSuite's branch coverage fitness function is the most effective—detecting more faults than any other criterion, and having a higher likelihood of detection for each fault. Line, direct branch, and weak mutation coverage also perform well in both regards.
- There is evidence that multi-objective suite generation could be more effective than single-objective generation, and EvoSuite's default combination of eight functions performs relatively well. However, the difficulty of simultaneously balancing all eight functions decreases the average performance of this configuration, and it fails to outperform branch coverage.
- High levels of coverage over the fixed version of the CUT and over patched lines of code are the factors that most strongly predict suite effectiveness.
- While others have found that test suite size correlates to mutant detection, we found that larger suites are not necessarily more effective at detecting real faults.

Our findings represent a step towards understanding the use, applicability, and combination of common fitness functions. Our observations indicate that, while coverage seems to be a prerequisite to effective fault detection, it is not sufficient to ensure it. The fitness function must still execute the code in a manner that triggers the fault, and ensures that it manifests in a failure. We hypothesize that lightweight, perhaps system-specific, combinations of fitness functions may be more effective than single metrics in improving the probability of fault detection—both exploring the structure of the CUT and sufficiently varying input in a manner that improves the probability of fault detection. However, 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.

References

[1] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, 36(6):742–762, 2010.

[2] N. Alshahwan and M. Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 181–192, New York, NY, USA, 2014. ACM.

[3] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[4] A. Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability*, 23(2):119–147, 2013.

[5] L. Bianchi, M. Dorigo, L. Gambardella, and W. Gutjahr. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, 8(2):239–287, 2009.

[6] X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing profiles. In *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, A-MOST '05, pages 1–7, New York, NY, USA, 2005. ACM.

[7] M. Dorigo and L. M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, 1997.

[8] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '98/FSE-6, pages 153–162, New York, NY, USA, 1998. ACM.

[9] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, TAV4, pages 154–164, New York, NY, USA, 1991. ACM.

[10] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, Aug 1993.

[11] G. Fraser and A. Arcuri. Whole test suite generation. *Software Engineering, IEEE Transactions on*, 39(2):276–291, Feb 2013.

[12] G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2014.

[13] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 291–301, New York, NY, USA, 2013. ACM.

[14] G. Gay, T. Menzies, M. Davies, and K. Gundy-Burlet. Automatically finding the control variables for complex system behavior. *Automated Software Engineering*, 17(4):439–468, Dec. 2010.

[15] G. Gay, A. Rajan, M. Staats, M. Whalen, and M. P. E. Heimdahl. The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. *ACM Trans. Softw. Eng. Methodol.*, 25(3):25:1–25:34, July 2016.

[16] G. Gay, M. Staats, M. Whalen, and M. Heimdahl. The risks of coverage-directed test case generation. *Software Engineering, IEEE Transactions on*, PP(99), 2015.

[17] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 302–313, New York, NY, USA, 2013. ACM.

[18] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 302–313, New York, NY, USA, 2013. ACM.

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

[20] A. Groce, M. A. Alipour, and R. Gopinath. Coverage and its discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 255–268, New York, NY, USA, 2014. ACM.

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

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

[23] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.

[24] R. Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 433–436, New York, NY, USA, 2014. ACM.

[25] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

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

[27] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 436–439, Washington, DC, USA, 2011. IEEE Computer Society.

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

[29] T. Menzies and Y. Hu. Data mining for very busy people. *Computer*, 36(11):22–29, Nov. 2003.

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

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

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

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

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

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

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

[37] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. Combining multiple coverage criteria in search-based unit test generation. In M. Barros and Y. Labiche, editors, *Search-Based Software Engineering*, volume 9275 of *Lecture Notes in Computer Science*, pages 93–108. Springer International Publishing, 2015.

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

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

[40] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):pp. 80–83, 1945.