Generating Effective Test Suites by Combining Coverage Criteria

Gregory Gay

University of South Carolina, Columbia, SC, USA,
greg@greggay.com

Abstract. A number of criteria have been proposed to judge test suite adequacy. While search-based test generation has improved greatly at criteria coverage, the produced suites are still often ineffective at detecting faults. Efficacy may be limited by the single-minded application of one criterion at a time when generating suites—a sharp contrast to human testers, who simultaneously explore multiple testing strategies. We hypothesize that automated generation can be improved by selecting and simultaneously exploring multiple criteria.

To address this hypothesis, we have generated multi-criteria test suites, measuring efficacy against the Defects4J fault database. We have found that multi-criteria suites can be up to 31.15% more effective at detecting complex, real-world faults than suites generated to satisfy a single criterion and 70.17% more effective than the default combination of all eight criteria. Given a fixed search budget, we recommend pairing a criterion focused on structural exploration—such as Branch Coverage—with targeted supplemental strategies aimed at the type of faults expected from the system under test. Our findings offer lessons to consider when selecting such combinations.

Keywords: Search-based Test Generation, Automated Test Generation, Adequacy Criteria, Search-based Software Engineering

1 Introduction

With the exponential growth in the complexity of software, the cost of testing has risen accordingly. One way to lower costs without sacrificing quality may lie in automating the generation of test input [1]. Consider search-based generation—given a testing goal, and a scoring function denoting *closeness to attainment of that goal*, optimization algorithms can search for input that achieves that goal [12].

As we cannot know what faults exist a priori, dozens of criteria—ranging from the measurement of structural coverage to the detection of synthetic faults [14]—have been proposed to judge testing *adequacy*. In theory, if such criteria are fulfilled, tests should be *adequate* at detecting faults. Adequacy criteria are important for search-based generation, as they can guide the search [12].

Search techniques have improved greatly in terms of achieved coverage [2]. However, the primary goal of testing is not coverage, but fault detection. In this regard, automated generation often does not produce human competitive results [2, 3, 5, 15].

If automation is to impact testing practice, it must match—or, ideally, outperform—manual testing in terms of fault-detection efficacy.

The current use of adequacy criteria in automated generation sharply contrasts how such criteria are used by humans. For a human, coverage typically serves an advisory role—as a way to point out gaps in existing efforts. Human testers build suites in which adequacy criteria contribute to a *multifaceted* combination of testing strategies. Previous research has found that effectiveness of a criterion can depend on factors such as how expressions are written [4] and the types of faults that appear in the system [6]. Humans understand such concepts. They build and vary their testing strategy based on the needs of their current target. Yet, in automated generation, coverage is typically *the* goal, and a single criterion is applied at a time.

However, search-based techniques need not be restricted to one criterion at a time. The test obligations of multiple criteria can be combined into a single score or simultaneously satisfied by multi-objective optimization algorithms. We hypothesize that the efficacy of automated generation can be improved by applying a targeted, multifaceted approach—where multiple testing strategies are selected and simultaneously explored.

In order to examine the efficacy of suites generated by combining criteria, we have used EvoSuite and eight coverage criteria to generate multi-criteria test suites—as suggested by three selection strategies—with efficacy judged against the Defects4J fault database [10]. Based on experimental observations, we added additional configurations centered around the use of two criteria, Exception and Method Coverage, that performed poorly on their own, but were effective in combination with other criteria.

By examining the proportion of suites that detect each fault for each configuration, we can examine the effect of combining coverage criteria on the efficacy of search-based test generation, identify the configurations that are more effective than single-criterion generation, and explore situations where particular adequacy criteria can effectively cooperate to detect faults. To summarize our findings:

- For all systems, at least one combination is more effective than a single criterion, offering efficacy improvements of 14.84-31.15% over the best single criterion.
- The most effective combinations pair a structure-focused criterion—such as Branch Coverage—with supplemental strategies targeted at the class under test.
  - Across the board, effective combinations include Exception Coverage. As it can be added to a configuration with minimal effect on generation complexity, we recommend it as part of any generation strategy.
  - Method Coverage can offer an additional low-cost efficacy boost.
  - Additional targeted criteria—such as Output Coverage for code that manipulates numeric values or Weak Mutation Coverage for code with complex logical expressions—offer further efficacy improvements.

Our findings offer lessons to consider when selecting such combinations, and a starting point in discovering the best combination for a given system.

2 Background

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 testing efforts. Common methods of measuring adequacy involve coverage of structural elements of the software, such as individual statements, points where control can branch, and complex boolean expressions [7].

The idea of measuring adequacy through coverage is simple, but compelling: unless code is executed, many faults are unlikely to be found. 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, as they offer clear checklists of testing goals that can be objectively evaluated and automatically measured [7]. Importantly, they offer *stopping criteria*, advising on when testing can conclude. These very same qualities make adequacy criteria ideal for use as automated test generation targets.

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 [12]. Given scoring functions denoting *closeness to the attainment of those goals*—called *fitness functions*—optimization algorithms can sample from a large and complex set of options as guided by a chosen strategy (the *metaheuristic*). Metaheuristics are often inspired by natural phenomena. For example, genetic algorithms evolve a group of candidate solutions by filtering out bad "genes" and promoting fit solutions [2].

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. Metaheuristic search—by strategically sampling from that space—can scale effectively to large problems. Such approaches have been applied to a wide variety of testing scenarios [1].

While adequacy has been used in almost all generation methods, it is particularly relevant to metaheuristic search-based generation. In search-based generation, the fitness function must capture the testing objective and provide feedback to guide the search. Through this guidance, the fitness function has a major impact on the quality of the solutions generated. Adequacy criteria are common optimization targets for automated test case generation, as they can be straightforwardly transformed into distance functions that lead to the search to better solutions [12].

3 Study

We hypothesize that the efficacy of automated generation can be improved by selecting and simultaneously exploring a combination of testing strategies. In particular—in this project—we are focused on combinations of common adequacy criteria.

Rojas et al. previously found that multiple fitness functions could be combined with minimal loss in coverage of any single criterion and with a reasonable increase in test suite size [14]. Indeed, recent versions of the EvoSuite framework[1] now, by default, combine eight coverage criteria when generating tests. However, their work did not assess the effect of combining criteria on the fault-detection efficacy of the generated suites. We intend to focus on the performance of suites generated using a combination of criteria. In particular, we wish to address the following research questions:

1. Do test suites generated using a combination of two or more coverage criteria have a higher likelihood of fault detection than suites generated using a single criterion?

[1] Available from `http://evosuite.org`

2. For each system, and across all systems, which combinations are most effective?
3. What effect does an increased search budget have on the studied combinations?
4. Which criteria best pair together to increase the likelihood of fault detection?

The first two questions establish a basic understanding of the effectiveness of criteria combinations—given fixed search budgets, are *any* of the combinations more effective at fault detection than suites generated to satisfy a single criterion? Further, we hypothesize that the most effective combination will vary across systems. We wish to understand the degree to which results differ across the studied systems, and whether the search budget plays a strong role in determining the resulting efficacy of a combination. In each of these cases, we would also like to better understand *why* and *when* particular combinations are effective.

In order to examine the efficacy of suites generated using such combinations, we have first applied EvoSuite and eight coverage criteria to generate test suites for the systems in the Defects4J fault database [10]. We have performed the following:

1. **Collected Case Examples:** We have used 353 real faults, from five Java projects, as test generation targets (Section 3.1).
2. **Generated Test Cases:** For each fault, we generated 10 suites per criterion using the fixed version of each class-under-test (CUT). We use both a two-minute and a ten-minute search budget per CUT (Section 3.2).
3. **Removed Non-Compiling and Flaky Tests:** Any tests that do not compile, or that return inconsistent results, are automatically removed (Section 3.2).
4. **Assessed Fault-finding Effectiveness:** For each fault, we measure the proportion of test suites that detect the fault to the number generated.

Following single-criterion generation, we applied three different selection strategies to build sets of multi-criteria configurations for each system (described in Section 3.3). We generated suites, following the steps above, using each of the configurations suggested by these strategies, as well as EvoSuite's default eight-criteria configuration. Based on our initial observations, we added additional configurations centered around two criteria—Exception and Method Coverage—that performed poorly on their own, but were effective in combination with other criteria (See Section 4).

3.1 Case Examples

Defects4J is an extensible database of real faults extracted from Java projects [10][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.

3.2 Test Suite Generation

EvoSuite 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.

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

It is actively maintained and has been successfully applied to a variety of projects [15]. We used the following fitness functions, corresponding to common coverage criteria[3]:

Branch Coverage (BC): A test suite satisfies BC 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 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 [14].

Direct Branch Coverage (DBC): Branch Coverage may be attained by calling a method *directly*, or *indirectly*—calling a method within another method. DBC requires each branch to be covered through a direct call.

Line Coverage (LC): A test suite satisfies LC if it executes each non-comment line of code at least once. To cover each line, 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, EvoSuite requires that the branch of the statement leading to the dependent code is executed.

Exception Coverage (EC): EC rewards test suites that force the CUT to throw more 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 the largest observed number of exceptions.

Method Coverage (MC): MC simply requires that all methods in the CUT be executed at least once, either directly or indirectly.

Method Coverage (Top-Level, No Exception) (MNEC): Test suites sometimes achieve MC while 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): OC rewards diversity in method output by mapping return types to abstract values. A test suite satisfies OC if, for each public method, at least one test yields a concrete value characterized by each abstract value. For numeric data types, distance functions guide the search by comparing concrete and abstract values.

Weak Mutation Coverage (WMC): Suites that detect more mutants may be effective at detecting real faults as well. A test suite satisfies WMC 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 detecting mutated statements.

To generate for multiple criteria, EvoSuite calculates the fitness score as a linear combination of the objectives for all of the criteria [14]. No ordering is imposed on the criteria when generating—combinations such as BC-LC and LC-BC are equivalent.

Test suites are generated for each class reported as faulty, using the fixed version of the CUT. They are applied to the faulty version in order to eliminate the oracle problem. This translates to a regression testing scenario, where tests guard against future issues.

[3] Rojas et. al provide a primer on each fitness function [14].

Two search budgets were used—two minutes and ten minutes per class–allowing us to examine the effect of increasing the search budget. 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, generating an initial pool of 56,480 test suites.

Generation tools may generate flaky (unstable) tests [15]. For example, a test case that makes assertions about the system time will only pass during generation. We automatically remove flaky tests. First, non-compiling test cases are removed. Then, each test is executed on the fixed CUT five times. If results are inconsistent, the test case is removed. On average, less than one percent of the tests are removed from each suite.

3.3 Selecting Criteria Combinations

	Budget	Chart	Closure	Lang	Math	Time	Overall
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%
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%
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%
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%
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%
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%
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%
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%

Table 1. Average likelihood of fault detection for single-criterion generation, divided by budget and system.

Overall, suites generated to satisfy a single criterion detect 180 (50.99%) of the 353 studied faults. The average likelihood of fault detection is listed for each single criterion, by system and budget, in Table 1.

BC is the most effective single criterion, detecting 158 of the 353 faults. BC suites have an average likelihood of fault detection of 22.07% given a two-minute search budget and 25.61% given a ten-minute budget. Line and Direct Branch Coverage follow with a 19.92-22.78% and 19.43-23.80% likelihood of detection. DBC and WMC benefit the most from an increased search budget, with average improvements of 22.45% and 22.33%. Criteria with distance-driven fitness functions—BC, DBC, LC, WMC, and, partially, OC—improve the most given more time.

We seek to identify whether combinations of criteria can outperform the top single criterion for each system—either BC or DBC. Studying all possible combinations is infeasible—even restricting to combinations of four criteria would result in 1,680 configurations. To control experiment cost, we have employed three strategies to suggest combinations. We have also used Evosuite's default configuration of all eight criteria. We perform selection using each strategy for each system and search budget, then build combinations using the entire pool of faults in order to suggest general combinations. The suggested combinations are then used to generate suites. All combinations are applied ten times per search budget. To converse space, we do not list all combinations. However, the top combinations are named and discussed in Section 4. The three selection strategies include:

"Top-Ranked" Strategy: This simple strategy build combinations by selecting the top two to four coverage criteria, as seen in Table 1, and combining them.

"Boost" Strategy: This strategy aims to select secondary criteria that "back up" a central criterion—the criterion that performed best on its own—in situations where that central criterion performs poorly. To select the additional criteria, the pool of faults is filtered for examples where that central criterion had $<= 30\%$ likelihood of fault detection. Then, two to four top-ranked criteria from that filtered pool are selected for the combination. Criteria are only selected if they are more effective than the central criterion post-filtering.

"Unique Faults" Strategy: This greedy strategy favors the criteria that detect the highest number of unique faults. Combinations of criteria are selected by choosing the criterion that produced suites that detected the most faults, removing those faults from consideration, and choosing from the remaining faults and criteria. Ties are broken at random. Selection stops once four criteria are chosen, or if no faults remain.

Together, these three strategies (and EvoSuite's default) yielded 12 combinations for Chart, 12 for Closure, 17 for Lang, 14 for Math, and 11 for Time—resulting in the generation of 94,760 test suites.

4 Results & Discussion

Tables 2-3 show—for each system, and across all systems—the combinations that are more effective than the top single criterion for the two-minute and ten-minute search budgets. We also list the performance of EvoSuite's default combination of eight criteria. Combinations outperformed by the top criterion, except for the default, are omitted. For the overall results, only combinations generated for all systems are considered. The abbreviations for each criterion are listed in Section 3.2. For each combination, we note the strategies that suggested the combination and the average efficacy. In Table 3, we also note the improvement from the increased budget.

For all systems and both budgets, at least one combination outperforms the top single criterion. This validates our core hypothesis—the likelihood of fault detection can be increased by combining criteria. As can be seen in Tables 2 and 3, EvoSuite's default combination of all eight criteria rarely manages to outperform the top single criterion. As the number of criteria expands, the difficulty of the search process also grows. While criteria can work together to produce more effective suites, there is a point where the generation task becomes too difficult to achieve within the selected budget. Our observations, instead, point towards the wisdom of choosing a *targeted* set of criteria for the CUT.

We proposed three strategies to suggest combinations—the Top-Ranked, Boosting, and Unique Faults strategies. Examining the results of this experiment, the Unique Faults strategy seems to produce the best overall results, suggesting nine of the top strategies for the two-minute budget and 24 for the ten-minute budget. The Boosting strategy suggests only three for the two-minute budget and 17 for the ten-minute budget, and the Top Ranked strategy suggests seven for the two-minute budget and 14 for the ten-minute budget.

However, while these strategies have yielded effective combinations, we cannot recommend any of them as a general-purpose strategy for testing new systems. Each also suggested a large number of ineffective combinations. With a two-minute budget, only

System	Combination	Strategy	Efficacy
Chart	Default	-	47.30%
	BC	-	45.00%
Closure	BC-LC	TR, UF	6.00%
	DBC	-	5.10%
	Default	-	4.50%
Lang	BC-EC-LC-MC	UF	40.00%
	BC-EC	UF	39.40%
	BC-LC	TR, BS, UF	36.50%
	BC-EC-LC	UF	35.70%
	BC-DBC	TR, BS, UF	35.50%
	BC-LC-DBC	TR, BS	34.00%
	BC	-	34.00%
	Default	-	23.80%
Math	BC-LC-OC-EC	UF	32.40%
	BC-LC	TR, UF	31.90%
	BC	-	27.90%
	Default	-	25.80%
Time	DBC-BC-LC	TR, BS	35.20%
	BC	-	34.80%
	Default	-	25.90%
Overall	BC-LC	TR, UF	24.00%
	BC	-	22.10%
	Default	-	19.00%

Table 2. Efficacy (average likelihood of fault detection) for the initial set of combinations, when generated with a two-minute search budget. TR = Top-Ranked Strategy, BS = Boosting Strategy, UF = Unique Faults Strategy.

System	Combination	Strategy	Efficacy	Improvement From Budget
Chart	BC-LC-WMC-EC	UF	57.30%	35.46%
	BC-EC-DBC	BS	55.80%	28.28%
	BC-EC	BS	54.60%	23.53%
	BC-DBC-WMC-OC	UF	49.20%	54.23%
	BC-LC-WMC	TR, UF	48.90%	49.71%
	BC-WMC-MC	BS	48.90%	27.01%
	BC-WMC	BS, UF	48.80%	39.43%
	BC	-	48.50%	7.78%
	Default	-	48.10%	1.69%
Closure	BC-LC	TR, UF	7.6%	26.67%
	BC-LC-DBC	TR	7.30%	48.98%
	BC-LC-WMC-EC	UF	7.20%	94.59%
	BC-DBC	TR, BS, UF	7.10%	51.05%
	Default	-	7.10%	57.78%
	BC-WMC-DBC-LC	TR, BS	7.00%	42.86%
	DBC-WMC	BS, UF	6.80%	61.90%
	BC-WMC	BS	6.50%	54.76%
	DBC-WMC-BC	TR, BS, UF	6.40%	60.00%
	DBC	-	6.10%	19.61%
Lang	BC-EC	UF	47.50%	20.56%
	BC-EC-LC-MC	UF	45.70%	14.25%
	BC-EC-LC	UF	45.70%	28.00%
	BC-LC-DBC	TR, BS	42.30%	24.41%
	BC-LC-WMC-EC	UF	41.70%	44.79%
	BC-DBC	TR, BS, UF	41.10%	12.60%
	BC-LC	TR, BS, UF	40.90%	12.05%
	BC	-	40.20%	18.24%
	Default	-	32.60%	36.97%
Math	BC-LC-OC-EC	UF	38.00%	17.28%
	BC-OC-LC	BS, UF	35.80%	31.62%
	BC-OC	BS	34.70%	24.82%
	BC-OC-LC-WMC	BS, UF	33.70%	22.99%
	BC-LC	TR, UF	31.90%	3.76%
	BC-LC-WMC-EC	UF	33.00%	21.32%
	BC	-	32.80%	17.56%
	Default	-	32.80%	27.13%
Time	DBC-BC	TR, BS, UF	43.30%	39.22%
	DBC-BC-LC	TR, BS	41.50%	17.90%
	DBC	-	40.40%	29.90%
	Default	-	33.33%	28.68%
Overall	BC-LC-WMC-EC	UF	26.90%	33.83%
	BC-LC	TR, UF	26.40%	10.00%
	BC-DBC	TR, BS, UF	25.80%	20.00%
	BC-LC-DBC	TR	25.60%	28.00%
	BC	-	25.60%	15.84%
	Default	-	24.20%	27.39%

Table 3. Efficacy for the initial set of combinations (ten-minute search budget).

28% of the the Top Ranked strategy combinations were effective. For the Boosting strategy, the total was only 8%, and for the Unique Faults strategy, the total was only 19%. With a ten-minute budget, 56% of the combinations for the Top Ranked strategy were effective. For the Boosting strategy, this was 45%, and for the Unique Faults strategy, the total was 50%. Therefore, while the basic hypothesis—that combinations *can* outperform a single criterion—seems to be valid, more research is needed in how to determine the best combination.

4.1 Additional Configurations

Following test generation, we noticed that (a) Exception Coverage was frequently selected as part of combinations, despite performing poorly on its own, and (2), that these combinations are often highly effective. For example, the top combinations for Chart, Lang, and Math in Tables 2-3 all contain EC. Of the studied criteria, EC is unique in that it does not prescribe static test obligations. Rather, it simply rewards suites that

System	Combination	Efficacy
Chart	*Default*	47.30%
	BC	45.00%
Closure	*BC-LC*	6.00%
	BC-LC-EC	5.70%
	DBC-EC	5.10%
	DBC	5.10%
	Default	4.50%
Lang	*BC-EC-LC-MC*	40.00%
	BC-EC*	39.40%
	BC-LC-EC*	35.70%
	BC	34.00%
	Default	23.80%
Math	*BC-LC-OC-EC*	32.40%
	BC-EC	31.70%
	BC	27.90%
	Default	25.80%
Time	BC-EC	39.60%
	DBC-BC-LC-EC	39.30%
	DBC-EC	37.80%
	DBC-BC-LC	35.20%
	BC	34.80%
	Default	25.90%
Overall	BC-EC	24.50%
	BC-LC	24.00%
	BC-LC-EC	22.40%
	BC	22.10%
	Default	19.00%

Table 4. Efficacy of Exception Coverage-based combinations (two-minute budget). The top combination from Table 2, top single criterion, and EvoSuite's default combination are shown for context. A * means that the combination was also suggested by a previous strategy.

System	Combination	Efficacy	Improvement From Budget
Chart	*BC-LC-WMC-EC*	57.30%	35.46%
	BC-EC*	54.60%	23.53%
	BC-LC-EC	50.40%	14.81%
	BC	48.50%	7.78%
	Default	48.10%	1.69%
Closure	*BC-LC*	7.6%	26.67%
	BC-LC-EC	7.40%	29.82%
	BC-EC	7.10%	47.92%
	Default	7.10%	57.78%
	DBC	6.10%	19.61%
Lang	BC-EC*	47.50%	20.56%
	BC-LC-EC*	45.70%	28.00%
	BC	40.20%	18.24%
	Default	32.60%	36.97%
Math	*BC-LC-OC-EC*	38.00%	17.28%
	BC-EC	34.00%	7.25%
	BC	32.80%	17.56%
	Default	32.80%	27.13%
Time	BC-EC	44.80%	13.13%
	DBC-BC-EC	44.10%	38.25%
	DBC-BC	43.30%	39.22%
	DBC-BC-LC-EC	42.60%	8.40%
	BC-LC-EC	41.10%	18.10%
	DBC-EC	41.10%	8.73%
	DBC	40.40%	29.90%
	Default	33.33%	28.68%
Overall	BC-EC	28.70%	17.14%
	BC-LC-EC	27.50%	22.77%
	BC-LC-WMC-EC	26.90%	33.83%
	BC	25.60%	15.84%
	Default	24.20%	27.39%

Table 5. Efficacy of EC-based combinations (ten-minute budget). The top combination from Table 3, top single criterion, and EvoSuite's default combination are shown for context.

cause more exceptions to be thrown. This means that it can be added to a combination with little increase in search complexity.

To further study the potential of EC as a "low-cost" addition to combinations, we added a set of additional combinations to our study. Specifically, we generated tests for all systems using the BC-EC and BC-LC-EC combinations—the combination of EC with the overall best single criterion, and the combination of EC with the best overall combination seen to this point. As DBC outperformed BC for the Closure and Time systems, we also generated tests for the DBC-EC combination in those two cases. Finally, as the top-ranked combination Time lacked EC, we added the DBC-BC-LC-EC and DBC-BC-EC combinations for that system. In total, this adds one new configuration for Chart, three for Closure, zero for Lang, two for Math, and five for Time—resulting in the generation of an additional 15,280 test suites.

Results can be seen in Tables 4 and 5 for the two budgets. From these results, we can see that—almost universally—the best observed combination of criteria includes Exception Coverage. In fact, the best overall configuration—up to this point—is a simple combination of BC and EC. The simplicity of EC explains its poor performance as the *primary* criterion. It lacks a feedback mechanism to drive generation towards exceptions. However, EC appears to be effective *when paired with criteria that effectively explore the structure of the CUT*, such as Branch or Line Coverage. Exception Cover-

System	Combination	Efficacy
Chart	*Default*	47.30%
	BC	45.00%
Closure	*BC-LC*	6.00%
	BC-LC-EC	5.70%
	BC-EC-MC	5.60%
	BC-LC-MC	5.30%
	DBC	5.10%
	Default	4.50%
Lang	BC-EC-MC	40.50%
	BC-EC-LC-MC*	40.00%
	BC-EC	39.40%
	BC-LC-MC	38.00%
	BC	34.00%
	Default	23.80%
Math	BC-LC-OC-EC-MC	32.90%
	BC-LC-OC-EC	32.40%
	BC-EC	31.70%
	BC-EC-MC	30.40%
	BC-EC-LC-MC	30.20%
	BC	27.90%
	Default	25.80%
Time	*BC-EC*	39.60%
	BC-EC-MC	35.90%
	DBC-BC-LC	35.20%
	BC	34.80%
	Default	25.90%
Overall	*BC-EC*	24.50%
	BC-EC-MC	24.30%
	BC-LC	24.00%
	BC-EC-LC-MC	23.60%
	BC-LC-MC	22.20%
	BC	22.10%
	Default	19.00%

Table 6. Efficacy of Method Coverage-based combinations (two-minute budget). The top combinations from Tables 2 and 4, top single criterion, and EvoSuite's default combination are are shown for context. A * means that the combination was also suggested by a previous strategy.

System	Combination	Efficacy	Improvement From Budget
Chart	*BC-LC-WMC-EC*	57.30%	35.46%
	BC-EC	54.60%	23.53%
	BC-EC-MC	53.90%	25.06%
	BC-LC-EMC-EC-MC	53.50%	19.96%
	BC	48.50%	7.78%
	Default	48.10%	1.69%
Closure	BC-EC-MC	8.00%	42.86%
	BC-EC-LC-MC	7.70%	54.00%
	BC-LC	7.6%	26.67%
	BC-LC-EC	7.40%	29.82%
	Default	7.10%	57.78%
	DBC	6.10%	19.61%
Lang	BC-EC-MC	48.20%	19.01%
	BC-EC	47.50%	20.56%
	BC-EC-LC-MC*	45.70%	14.25%
	BC-LC-MC	43.70%	15.00%
	BC	40.20%	18.24%
	Default	32.60%	36.97%
Math	BC-LC-OC-EC-MC	39.00%	18.54%
	BC-LC-OC-EC	38.00%	17.28%
	BC-EC-MC	34.30%	12.83%
	BC-EC-LC-MC	34.10%	12.91%
	BC-EC	34.00%	7.25%
	BC	32.80%	17.56%
	Default	32.80%	27.13%
Time	BC-EC-MC	47.00%	30.92%
	BC-EC	44.80%	13.13%
	DBC-BC	43.30%	39.22%
	BC-EC-LC-MC	41.10%	18.10%
	DBC	40.40%	29.90%
	Default	33.33%	28.68%
Overall	BC-EC-MC	29.40%	20.99%
	BC-EC	28.70%	17.14%
	BC-EC-LC-MC	27.80%	17.80%
	BC-LC-WMC-EC	26.90%	33.83%
	BC-LC-MC	25.60%	15.31%
	BC	25.60%	15.84%
	Default	24.20%	27.39%

Table 7. Efficacy of MC-based combinations (ten-minute budget). Top combinations from Tables 3 and 5, top single criterion, and EvoSuite's default combination are shown for context.

age adds little cost in terms of generation difficulty, and almost universally outperforms the use of Branch Coverage alone.

An example of effective combination can be seen in fault 60 for Lang[4]—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 BC and EC 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, EC alone never detects the fault. BC provides the necessary means to drive program execution to the correct location. However, two suites with an equal coverage score are considered equal. BC alone may prioritize suites with slightly higher (or different) coverage, missing the fault. By combining

[4] `https://github.com/apache/commons-lang/commit/a8203b65261110c4a30ff69fe0da7a2390d82757`.

the two, exception-throwing tests are prioritized and retained, succeeding where either criterion would fail alone.

Given that EC can boost the likelihood of fault detection without a substantial cost increase, it seems reasonable to look for other "low-cost" criteria that could provide a similar effect. The two forms of Method Coverage used in this project are ideal candidates. In general, a class will not have a large number of methods, and methods are either covered or not covered. Additionally, MC also appears in some of the top combinations—such as those for Lang—despite poor performance on its own.

Therefore, we have also generated tests for the BC-EC-MC, BC-LC-MC, and BC-EC-LC-MC combinations for all systems. We have also added MC to the top combination for any system that did not already have one of the above as the resulting combination, adding BC-LC-WMC-EC-MC for Chart and BC-LC-OC-EC-MC for Math. In total, this adds four new combinations for Chart, three for Closure, two for Lang, four for Math, and three for Time—yielding 22,400 additional test suites.

The results of these combinations can be seen in Tables 6-7. With a two-minute budget, the addition of Method Coverage can improve results—as seen in Lang, where BC-EC-MC outperforms BC-EC, and Math, where BC-LC-OC-EC-MC outperforms BC-LC-OC-EC. However, in other cases—such as with Closure and Time—the addition of MC decreases efficacy. Results improve across the board with a ten-minute budget, where the top combinations for Closure, Lang, Math, and Time all contain MC. Overall, with a ten-minute budget, the combination of BC-EC-MC outperforms any other blanket policy. It seems that MC can improve a combination, but does not have the same impact as EC. Given a high enough search budget, we do recommend its inclusion. An example where the addition of MC could boost efficacy can be seen in Lang fault 34[5]. This fault resides in two small (1-2 line) methods. Calling either method will reveal the fault, but BC can easily overlook them.

4.2 Observations and Recommendations

We can address each research question in turn. First:

> For all systems and search budgets, at least one combination of criteria is more effective than a single criterion, with the top combination offering a 5.11-31.15% improvement in the likelihood of fault detection over the best single criterion and up to 70.17% improvement over the default combination of all eight criteria.

For each budget B, combination C, and individual criterion I, we formulate hypothesis H and null hypothesis $H0$:

- H: With budget B, test suites generated using X will have a higher likelihood of fault detection than suites generated using I.
- $H0$: Observations of efficacy for C and I are drawn from the same distribution.

Due to the limited number of faults for Chart and Time, we have focused on overall results. Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate $H0$ without

[5] https://github.com/apache/commons-lang/commit/496525b0d626dd5049528cdef61d71681154b660

any assumptions on distribution, we use a one-sided (strictly greater) Mann-Whitney-Wilcoxon rank-sum test, a non-parametric hypothesis test for determining if one set of observations is drawn from a different distribution than another set of observations. We apply the test for each pairing of fitness function and search budget with $\alpha = 0.05$.

To save space, we focus on the top combinations—BC-EC for the two-minute budget and BC-EC-MC for the ten-minute budget. At the two-minute search budget, we can reject the null hypothesis for MC, MNEC, EC, OC (all $<$0.001), and WMC (0.030). We cannot reject the null hypothesis for DBC (0.055), LC (0.057), or BC (0.304). At the ten-minute level, we can reject the null hypothesis for MC, MNEC, EC, OC, WMC, LC (all $<$ 0.001), DBC (0.017), and BC (0.040). Therefore, the BC-EC-MC combination significantly outperforms all individual criteria, given sufficient search budget.

From Tables 3, 5, and 7, we can see that the search budget affects the efficacy of combinations. At a higher budget, more combinations outperform individual criteria, and the performance gap between combinations and individual criteria widens. While combinations *can* outperform individual criteria at the two-minute budget, a larger budget clearly benefits combinations.

As more criteria are added, the generation task becomes more complex. There is a trade-off to be made in terms of the required search budget and the efficacy of the results. The default eight-way combination of criteria, even with a ten-minute budget, is ineffective in the majority of cases. While an even higher budget may help, we have seen that simple, *targeted* combinations can perform very well, even with a tight budget.

This leads, naturally, to the next question—which combinations are effective, in practice? At the two minute budget, a combination of all eight criteria is the most effective for Chart. BC-LC is best for Closure. For Lang, it is CB-EC-MC. For Math, it is BC-LC-OC-EC-MC. Finally, for Time, it is BC-EC. The best general policy, at that budget, is the BC-EC combination. More consensus is seen at the ten-minute budget level, where the BC-EC-MC combination is the best observed for Closure, Lang, and Time (and is the best general policy). For Chart, the top combination is BC-LC-WMC-EC. For Math, is is BC-LC-OC-EC-MC.

We do not wish to advocate these as the best *possible* combinations. Even for the studied systems, we did not exhaustively try all possibilities. Further, while performance gains are reasonably significant, better performance is likely possible. Rather, we wish to use this study to derive a starting point for those who wish to generate effective tests.

Either Branch or DBC was found to be the most effective single criterion. Tests that fail to execute faulty lines of code are highly unlikely to reveal a fault, so a criterion intended to achieve code coverage should form the core of a combination. However, code coverage is not sufficient on its own. Merely executing code does not ensure a failure—*how* that code is executed is important. From our results, we can observe that the most effective combinations pair a structure-focused criterion with a small number of supplemental strategies that can guide the structure-based criterion towards the *correct* input for the CUT.

Across the board, effective combinations include Exception Coverage. As EC can be added to a combination with minimal effect on generation complexity, we recommend it as part of any generation strategy. Although Method Coverage does not have the clear

symbiotic relationship with BC that EC has, it offers a slight boost to efficacy at a low cost. We recommend its inclusion in combinations with a longer search budget.

> We recommend a combination of Branch or Direct Branch Coverage with Exception and Method Coverage as a *base* approach to test generation. Additional criteria, targeted towards the CUT, may further improve efficacy.

We observed several situations where the central structure-based criterion is boosted by secondary criteria. First, Output Coverage often assists in revealing faults for Math. OC divides the data type of the method output into a series of abstract values, then rewards suites that cover each of those classes. In particular, OC offers the search feedback for numeric data types [14], explaining its utility for Math. For example, consider Fault 53[6]. The patch removes a misbehaving check for NaN. As the fixed version *removes* code, BC does not reveal the fault. However, Output Coverage ensures that method calls return a variety of values—raising the likelihood of fault detection.

Weak Mutation Coverage can also boost BC. Consider Lang fault 28[7]. BC alone fails to detect the fault, while WMC alone has a 40% chance of detection. A BC-WMC combination has a 90% chance of detection. The patched code includes an `if`-condition that can be mutated in several ways. BC assists in mutation detection by driving execution to, and into, the `if`-block. This combination is effective for other similar faults.

Combining structure-focused criteria seems potentially redundant. However, BC-LC and BC-DBC combinations can be effective (see Table 7). Consider Closure fault 94[8]. No single criterion detects the fault. However, at the ten-minute budget, the BC-LC combination has a 30% detection likelihood. The BC-LC combination is not only more effective, but also achieves higher levels of coverage. BC suites attain an average of 54.91% LC and 37.46% BC. LC-based suites attain 58.94% LC and 33.99% BC. Suites generated using the combination achieve 59.09% LC and 42.45% BC. By attaining higher coverage, the combination is more likely to execute the faulty code.

While more research is needed to identify situations where criteria work well together, developers should be able to produce more effective test cases using automated generation by considering the CUT and choosing criteria accordingly.

5 Related Work

Advocates of adequacy criteria hypothesize that there should exist a correlation between higher attainment of a criterion and fault detection efficacy [7]. Researchers have attempted to address whether such a correlation exists for almost as long as such criteria have existed [13, 5, 8]. Inozemtseva et al. provide a good overview [8].

Shamshiri et al. applied EvoSuite (Branch Coverage only), Randoop, and Agitar to each fault in Defects4J to assess the fault-detection capabilities of automated generation [15]. They found that the combination of tools could detect 55.70% of the faults.

[6] `https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/53.src.patch`

[7] `https://github.com/apache/commons-lang/commit/3e1afecc200d7e3be9537c95b7cf52a7c5031300`

[8] `https://github.com/rjust/defects4j/blob/master/framework/projects/Closure/patches/94.src.patch`

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. Our recent experiments expand on this work, comparing fitness functions from EvoSuite in terms of fault detection efficacy [3].

Rojas et al. previously found 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 [14]. Others have explored combinations of coverage criteria with non-functional criteria, such as memory consumption [11] or execution time [16]. Few have studied the effect of such combinations on fault detection. Jeffrey et al. found that combinations are effective following suite reduction [9].

6 Threats to Validity

External Validity: We have focused on five systems. We believe such systems are representative of small to medium-sized open-source Java systems, and that we have examined a sufficient number of faults to offer generalizable results.

We have used only one test generation framework. While other techniques may yield different results, no other framework offers the same variety of coverage criteria. Therefore, a more thorough comparison of tool performance cannot be made. While exact results may differ, we believe that general trends will remain the same, as the underlying criteria follow the same philosophy.

To control costs, we have only performed ten trials per combination of fault, budget, and configuration. Additional trials may yield different results. However, we believe that 134,300 suites is a sufficient number to draw conclusions.

Conclusion Validity: When using statistical analysis, we ensure base assumptions are met. We use non-parametric methods, as distribution characteristics are not known.

7 Conclusions

In this work, we have examined the effect of combining coverage criteria on the efficacy of search-based test generation, identified effective combinations, and explored situations where criteria can cooperate to detect faults. For all systems, we have found that at least one combination is more effective than individual criteria, with the top combinations offering up to a 31.15% improvements in efficacy over top individual criteria. The most effective combinations pair a criterion focused on structure exploration—such as Branch Coverage—with a small number of targeted supplemental strategies suited to the CUT. Our findings offer lessons to consider when selecting such combinations.

Although we recommend the combination of Branch, Exception, and Method Coverage as a starting point, further research is needed to determine how to select the best combination for a system. In future work, we plan to focus on automated means of selecting combinations, perhaps using hyperheuristic search.

8 Acknowledgements

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

References

1. S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, August 2013.
2. 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, pages 291–301, New York, NY, USA, 2013. ACM.
3. G. Gay. The fitness function for the job: Search-based generation of test suites that detect real faults. In *Proceedings of the 2017 International Conference on Software Testing*, ICST 2017. IEEE, 2017.
4. 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.
5. 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.
6. R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *25th International Symposium on Software Reliability Engineering*, pages 189–200, Nov 2014.
7. 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!'14, pages 255–268, New York, NY, USA, 2014. ACM.
8. 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.
9. D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering*, 33(2):108–123, Feb 2007.
10. 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.
11. K. Lakhotia, M. Harman, and P. McMinn. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1098–1105, New York, NY, USA, 2007. ACM.
12. P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.
13. 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.
14. 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.
15. 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.
16. S. Yoo and M. Harman. Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software*, 83(4):689 – 701, 2010.