To Call, or Not to Call: Contrasting Direct and Indirect Branch Coverage in Test Generation *

Gregory Gay
University of South Carolina
Columbia, SC, United States
greg@greggay.com

ABSTRACT

While adequacy criteria offer an end-point for testing, they do not mandate *how* targets are covered. Branch Coverage may be attained through *direct* calls to methods, or through *indirect* calls between methods. Automated generation is biased towards the rapid gains offered by indirect coverage. Therefore, even with the same end-goal, humans and automation produce very different tests. Direct coverage may yield tests that are more understandable, and that detect faults missed by traditional approaches. However, the added burden for the generation framework may result in lower coverage and faults that emerge through method interactions may be missed.

To compare the two approaches, we have generated test suites for both, judging efficacy against real faults. We have found that requiring direct coverage results in lower achieved coverage and likelihood of fault detection. However, both forms of Branch Coverage cover code and detect faults that the other does not. By isolating methods, Direct Branch Coverage is less constrained in the choice of input. However, traditional Branch Coverage is able to leverage method interactions to discover faults. Ultimately, both are situationally applicable within the context of a broader testing strategy.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Search-based software engineering**; *Software verification and validation*;

KEYWORDS

Adequacy Criteria, Automated Test Generation, Branch Coverage

ACM Reference Format:
Gregory Gay. 2018. To Call, or Not to Call: Contrasting Direct and Indirect Branch Coverage in Test Generation . In *SBST'18: SBST'18:IEEE/ACM 11th International Workshop on Search-Based Software Testing , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3194718.3194719

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

SBST'18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5741-8/18/05…$15.00
https://doi.org/10.1145/3194718.3194719

```
public int[] add(int[] values, int valueToAdd){
  for(int i = 0; i < values.size(); i++){
    if(valueToAdd >= 0){
      values[i] = faultyAdd(values[i], valueToAdd);
    }
  }
  return values;
}

public int faultyAdd(int value, int valueToAdd){
  if (valueToAdd <= 0){ // FAULT, should be ==
    return value;
  }
  return value + valueToAdd;
}
```

Figure 1: Sample code where coverage can be attained directly or indirectly over method `faultyAdd`.

1 INTRODUCTION

As we cannot know what faults exist in software, dozens of criteria—ranging from the measurement of structural coverage to the detection of synthetic faults [9, 10]—have been proposed to judge the *adequacy* of software testing efforts. Such *adequacy criteria* provide advice to developers, and can be used as optimization targets for automated test generation [8].

Regardless of the process used to create test cases—automated or manual—adequacy criteria offer a measurable goal, a point at which test creation can stop. Consider Branch Coverage—arguably the most common criterion used in research and practice [5]. At various points in a class, the decision of which block of statements to execute depends on the outcome of a *branch predicate*. Such branching points—contained within `if` and `switch` statements and loop conditions—determine the flow of control. Branch Coverage mandates that each predicate evaluate to all possible outcomes, ensuring that the correct statements are executed.

No conditions are placed on *how coverage is achieved*. As a result, even though they may have the same end-goal, humans and automation produce very *different* test cases. Consider the two methods in Figure 1. Method `add` iterates over an array of integers. If the value to add to each is ≥ 0, then method `faultyAdd` is called to add that amount. Method `faultyAdd` has a fault in it, where—if the value to add is ≤ 0—we return the original value. The expression should state == 0, which means that negative numbers are incorrectly handled. Because the two methods are linked through the call from `add` to `faultyAdd`, Branch Coverage of `faultyAdd` can be attained *indirectly* by providing test input to `add`. Automated test generation algorithms, designed to reward efficient attainment of coverage, may never call `add` directly as its branches can be covered indirectly.

The use of adequacy criteria in automated generation contrasts how such criteria are used by humans. For a human, a criterion such as Branch Coverage typically serves an advisory role—as a way to point out gaps in existing efforts. Yet, in automated generation,

coverage is typically *the* goal, and generators will single-mindedly climb towards that goal. A human tester would not stop after testing `add`, just because Branch Coverage has been attained. They would still write unit tests for `faultyAdd` to ensure that it works in isolation. By stopping after attaining indirect coverage, automated test suites *cannot* discover the fault in Figure 1—the indirect call can only cover the faulty code with a value of 0, but a negative value is needed to trigger a failure. This is impossible without a direct call. While a human may not discover this fault either, they are more likely to attempt such input. Although this is a simple example, more complex representations of the same situation are common during development (we cover real-world examples in Section 4.3).

Indirect coverage of branches also carries a cognitive cost for human developers. In a user study, Fraser et.al found that developers dislike tests that cover branches indirectly, because they are harder to understand and extend with assertions [3]. This imposes a high *human oracle cost* that may outweigh the benefits of automated generation [1]. The understandability benefits of direct coverage may help alleviate the concerns of developers with the readability of generated unit tests [2].

Recent updates to the EvoSuite test generation framework allow the use of both traditional Branch Coverage, where indirect attainment is allowed, and Direct Branch Coverage, where branches must be covered through direct method calls [10]. Direct Branch Coverage should carry a lower human oracle cost, and may detect faults that require direct calls. However, because indirect coverage does not contribute to the total Branch Coverage, the generator must make additional method calls to cover branches that traditional Branch Coverage could handle indirectly. This will likely result in a dip in coverage unless additional time is offered to the generation process. If there is enough of a coverage loss, then generated suites may have lower fault-detection potential as well. Additionally, while direct coverage is required to detect the example in Figure 1, other faults may only—or more easily—emerge by focusing on the interactions between methods. Therefore, it is not clear whether the benefits of direct coverage outweigh the costs of ignoring indirect coverage.

In order to study the costs and benefits of each approach, we have used EvoSuite to generate test suites using both variants of Branch Coverage, with efficacy judged against the Defects4J fault database [7]. By examining the attained coverage and fault-detection capabilities of both variants, we can determine the impact of the choice of fitness function on the generated test suites and explore situations where one form of Branch Coverage may be more appropriate than the other. To summarize our findings:

- Given a two-minute search budget, traditional Branch Coverage discovers 10.40% more faults and has a 13.59% higher average likelihood of fault detection than Direct Branch Coverage. With a ten-minute budget, traditional Branch Coverage discovers 4.32% more faults and has a 7.61% higher average likelihood of fault detection.
- Similarly, traditional Branch Coverage attains an average 7.94-9.00% higher Line Coverage and 9.09-10.20% higher Branch Coverage over the code, as well as 8.06-9.46% higher coverage over the *faulty* lines of code.
- However, each method covers portions of the code and detects faults that the other does not. By examining methods in isolation, Direct Branch Coverage is less constrained in the input it uses to cover each method. Traditional Branch Coverage is able to leverage the context in which methods interact to detect faults that emerge from those interactions.

We have found that requiring direct coverage imposes a cost in terms of coverage and likelihood of fault detection. As long as the human oracle benefits of Direct Branch Coverage outweigh the need to offer additional time for generation, practitioners may find value in requiring direct coverage. Ultimately, there are clear situations where each form of coverage is more suited to detecting a particular fault than the other. Importantly, both also have important limitations not possessed by the other. This indicates that both variants have value as part of a broader testing strategy, and that future approaches to test generation could leverage the strengths of each approach.

2 BACKGROUND

Test case creation can naturally be seen as a search problem [8]. 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 [8]. 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*). Metaheuristics are often inspired by natural phenomena, such as swarm behavior or evolution.

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

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 [9]. 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 [5][1].

Adequacy criteria offer clear checklists of testing goals that can be objectively evaluated and automatically measured [9]. These very same qualities make adequacy criteria ideal for use as automated test generation targets. In search-based testing, the fitness function needs to capture the testing objective and guide the search. Through this guidance, the fitness function has a major impact on the quality of the solutions generated. Adequacy criteria can be straightforwardly transformed into distance functions that effectively guide to the search to better solutions [8].

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

3 STUDY

While coverage criteria mandate an end-goal for testing, they impose no restrictions on how that goal is attained. Most test generation approaches count *indirect* coverage of the code in called methods towards the total. However, we could restrict counted coverage to that attained through direct calls to methods [10]. Direct Branch Coverage may offer benefits in terms of the understandability of test cases, and may contribute to fault discovery. However, it is not clear whether those benefits outweigh the potential loss in coverage—and potentially fault-detection capability—that would result from the additional demands imposed on the generation framework.

In order to study the costs and benefits of both forms of Branch Coverage, we have used EvoSuite to generate test suites using both variants, with efficacy judged against the Defects4J fault database [7]. By examining the coverage and fault-detection capabilities of suites generated using both forms of coverage, we can determine the impact of this choice on the automated test generation process and explore situations where one form of Branch Coverage may be more appropriate than the other. In particular, we wish to address the following research questions:

(1) Given a fixed time budget, which form of Branch Coverage detects the most faults, and which has the highest likelihood of fault detection?
(2) Given a fixed time budget, does the additional difficulty of attaining Direct Branch Coverage result in a lower final level of attained coverage?
(3) How does an increased search budget impact the performance gap between the two forms of Branch Coverage?
(4) Are there particular types of faults that certain forms of Branch Coverage are better suited to detect?

We have performed the following experiment:

(1) **Collected Case Examples:** We have used 353 real faults, from five Java projects, as test generation targets (Section 3.1).
(2) **Generated Test Cases:** For each class, we generated 10 suites satisfying each form of Branch Coverage. We performed this task using a two-minute and a ten-minute search budget per CUT (Section 3.2).
(3) **Assessed Fault-finding Effectiveness** (Section 3.3).
(4) **Recorded Generation Statistics:** For each suite, fault, and budget, we measure factors that allow us to compare suites, related to coverage, suite size, and suite fitness (Section 3.3).

3.1 Case Examples

Defects4J is an extensible database of real faults extracted from Java projects [7][2]. Currently, the "stable" dataset consists of 357 faults from five projects: Chart (26 faults), Closure (133), Lang (65), Math (106), and Time (27). Four faults from the Math project were omitted due to complications encountered during suite generation, leaving 353 that we used in our study.

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 and developer-written test cases that expose the fault.

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

3.2 Test Suite Generation

The EvoSuite framework uses a genetic algorithm to evolve test suites over a series of generations, forming a new population by retaining, mutating, and combining the strongest solutions. It is actively maintained and has been successfully applied to a variety of projects [4, 11]. In this study, we used EvoSuite version 1.0.3 and its implementations of Branch Coverage and Direct Branch Coverage.

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. The fitness value of a test suite is measured by executing all of its tests while tracking the branch distances $d(b, Suite)$ for each branch.

$$F_{BC}(Suite) = \sum_{b \in B} v(d(b, Suite)) \tag{1}$$

Note that $v(...)$ is a normalization of the distance $d(b, Suite)$ between 0-1. The value of $d(b, Suite)$, then, is calculated as follows:

$$d(b, Suite) = \begin{cases} 0 & \text{if the branch is covered,} \\ v(d_{min}(b, Suite)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases} \tag{2}$$

The cost function used to attain the distance value follows a standard formulation based on the branch predicate [8].

The fitness function for Direct Branch Coverage is the same as that used for Branch Coverage [10], but only methods directly invoked by the test cases are considered for the fitness and coverage computation of branches in public methods. Private methods may be covered indirectly (as they cannot be called directly).

Test suites are generated that target the classes reported as relevant to the fault by Defects4J. Tests are generated using the fixed version of the CUT and applied to the faulty version in order to eliminate the oracle problem. 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, and is comparable to similar testing experiments [11]. 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.

Generation tools may generate flaky (unstable) tests [11]. For example, a test case that makes assertions about the system time will only pass during generation. We automatically remove flaky and

	Budget	Chart	Closure	Lang	Math	Time	Overall
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

Table 1: Number of faults detected by each Branch Coverage variant. Totals are out of 26 faults (Chart), 133 (Closure), 65 (Lang), 102 (Math), 27 (Time), and 353 (Overall).

	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%
	% 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%

Table 2: Average likelihood of fault detection (proportion of suites that detect the fault to those generated), broken down by coverage type, budget, and system. "% Change" indicates how results change when moving to a larger search budget.

non-compiling tests. On average, less than one percent of the tests are removed from each suite [4].

3.3 Data Collection

To evaluate the fault-finding effectiveness of the generated test suites, we execute each test suite against the faulty version of each CUT. The **effectiveness** of each fitness function, for each fault, is the proportion of suites that successfully detect the fault to the total number of suites generated for that fault. We also collected the following for each test suite:

Achieved Branch and Line Coverage: Using the Cobertura tool, we have measured the Line and Branch Coverage achieved by each suite over 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.

Test Suite Size: Suites containing more tests are often thought to be more effective [6]. Even if two suites achieve the same coverage, the larger may be more effective simply because it exercises more combinations of input.

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

4 RESULTS & DISCUSSION

4.1 Comparing Fault Detection Capabilities

In Table 1, we list the number of faults detected by each variant of Branch Coverage (BC is traditional Branch Coverage, DBC denotes Direct Branch Coverage), broken down by system and search budget. 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 coverage type. In total, traditional Branch Coverage detects 158 faults (44.76% of the examples), while Direct Branch Coverage only detects 151 (42.78%). From these results, we can see that our initial hypothesis—that Direct Branch Coverage will be more difficult to satisfy due to the requirement for direct coverage—has some truth to it. At the two-minute budget, BC detects 10.40% more faults than DBC. This gap narrows at the ten-minute budget, where BC only detects 4.32% more faults.

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—the proportion of detecting suites to the total number of suites generated for that fault. The average likelihood of fault detection is listed for each coverage type, by system and budget, in Table 2. We also list the change in likelihood between budgets. We largely observe the same trends as above. Given a fixed budget, traditional Branch Coverage has an overall average likelihood of fault detection of 22.07% given a two-minute search budget and 25.61% given a ten-minute budget. Direct Branch Coverage follow with a 19.92% chance of detection given a two-minute budget and a 22.78% chance given a ten-minute budget. Again, traditional Branch Coverage outperforms direct coverage, with a 13.59% higher overall chance of detection with two minutes and 7.61% given ten minutes.

> Given a fixed time budget, traditional Branch Coverage outperforms Direct Branch Coverage, detecting 10.40%/4.32% more faults with a 13.59%/7.61% higher average likelihood of detection (two/ten-minute budget).

We can perform statistical analysis to assess our observations. We formulate hypothesis H and its null hypothesis, $H0$:

- H: Given a fixed budget, suites generated to satisfy traditional Branch Coverage will have a higher likelihood of fault detection than suites generated to satisfy Direct Branch Coverage.
- $H0$: Observations of fault detection likelihood for both criteria are drawn from the same distribution.

Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate $H0$ without any assumptions on distribution, we use a one-sided (strictly greater) Mann-Whitney-Wilcoxon rank-sum test [12]. Due to the limited number of faults for the Chart and Time systems, we have analyzed results across the combination of all systems (353 observations per budget, per criterion). We apply the test for each pairing of fitness function and search budget with $\alpha = 0.05$.

The application of this test results in a p-value of 0.13 at the two-minute budget, and 0.27 at the ten-minute budget. Therefore, we fail to reject the null hypothesis in both cases. Although Branch Coverage has a higher average performance:

> Traditional Branch Coverage fails to outperform Direct Branch Coverage with statistical significance.

Further, while traditional Branch Coverage is more effective than DBC, the gap between the two narrows as the search budget increases. The differences in the number of faults detected (Table 1) and likelihood of detection (Table 2) both decrease at the ten-minute budget. We can also see this from the "% Change" rows in Table 2.

	Budget	Chart	Closure	Lang	Math	Time	Overall
BC	120	55.26%	17.00%	73.00%	64.62%	74.00%	48.00%
	600	70.28%	23.00%	79.00%	67.19%	85.00%	54.00%
	% Change	27.18%	35.29%	8.22%	3.98%	14.87%	12.50%
DBC	120	49.77%	14.00%	64.00%	61.37%	71.00%	44.00%
	600	60.23%	18.00%	71.00%	65.16%	79.00%	49.00%
	% Change	21.02%	28.57%	10.94%	6.12%	11.28%	11.36%

Table 3: Average Branch Coverage attained by generated suites, broken down by coverage type, budget, and system.

	Budget	Chart	Closure	Lang	Math	Time	Overall
BC	120	65.19%	27.00%	79.00%	70.77%	83.00%	56.44%
	600	75.37%	34.00%	82.00%	72.39%	89.00%	61.16%
	% Change	15.59%	25.93%	3.80%	2.29%	7.23%	8.93%
DBC	120	56.67%	24.00%	71.00%	68.88%	81.00%	52.29%
	600	65.01%	28.00%	75.00%	71.11%	84.00%	56.11%
	% Change	14.11%	16.67%	5.63%	3.24%	3.70%	7.69%

Table 4: Average Line Coverage attained by generated suites, by coverage type, budget, and system.

Direct Branch Coverage benefits far more from the increase in search budget than traditional Branch Coverage does for several systems—DBC sees an average overall improvement of 22.45%, while Branch Coverage only improves by 16.05%.

These results confirm that the "direct" coverage requirement of Direct Branch Coverage does impose additional burden on the test generation framework. There is a dip in average performance at both budget levels, but not a statistically significant difference in either case. The performance gap is not enough of a deterrent to recommend the use of traditional Branch Coverage in situations where a testing practitioner could derive human oracle benefit from the understandable test cases generated using Direct Branch Coverage.

> As the gap between traditional and Direct Branch Coverage narrows at a higher search budget, we recommend its use—while allocating a longer budget—in situations where DBC may yield understandability benefits.

Additionally, although Branch Coverage detects more faults, it does not necessarily detect the *same* faults. Branch Coverage detects ten faults not detected by Direct Branch Coverage—again maintaining a slight edge. However, Direct Branch Coverage is able to uniquely detect three faults that are missed by traditional Branch Coverage. There is some variation in the performance of each technique between systems. For the Chart system, traditional Branch Coverage earns far better results, with 18.86-31.46% higher likelihood of detection. In general, indirect Branch Coverage maintains the edge, albeit with closer margins. However, there are also a few cases where Direct Branch Coverage has a slightly higher chance of fault detection—namely, for the Closure system at both budget levels (5.18-9.66% improvement) and Time at the ten-minute level (a modest 2.82% improvement). This indicates that:

> Both techniques, regardless of overall performance, have some level of situational applicability.

4.2 Comparing Suite Characteristics

In Table 3, we list the average level of Branch Coverage attained by the final generated suites for each system, budget, and coverage variant. We do the same for Line Coverage in Table 4. Like with fault detection, traditional Branch Coverage has an edge over Direct Branch Coverage given a fixed budget. Overall, traditional Branch Coverage attains an average of 7.94% higher Line Coverage and 9.09% higher Branch Coverage than Direct Branch Coverage given a two-minute budget. With a ten-minute budget, traditional Branch Coverage attains 9.40% higher average Line Coverage and 10.41% higher Branch Coverage. Again, this effect can be explained by the additional work required to gain coverage if indirect calls do not count towards the total.

	Branch Cov. to Fault Detection		Line Cov. to Fault Detection	
	120	600	120	600
BC	0.37	0.35	0.35	0.33
DBC	0.31	0.34	0.29	0.33

Table 5: Correlation of coverage to likelihood of fault detection.

	Budget	Chart	Closure	Lang	Math	Time	Overall
BC	120	62.76%	19.72%	73.03%	63.88%	84.60%	50.31%
	600	70.80%	25.36%	75.22%	65.80%	91.60%	54.04%
	% Change	12.81%	28.60%	3.00%	3.00%	8.27%	7.41%
DBC	120	55.83%	16.28%	63.99%	61.24%	81.52%	45.96%
	600	64.26%	20.63%	68.73%	64.88%	85.70%	50.02%
	% Change	15.10%	26.72%	7.41%	5.94%	5.13%	8.83%

Table 6: Average patch coverage (coverage over the patched lines of code), by variant, budget, and system.

A gap in the level of coverage does not always predict a gap in terms of fault-detection capabilities. For Closure and Time, the two systems where Direct Branch Coverage outperformed traditional BC, the average attained Line and Branch Coverage are still lower than that attained by traditional BC. To further examine this effect, we have measured—for both metrics and budgets—the correlation between attained Line Coverage and attained Branch Coverage using the Kendall rank correlation. The resulting τ values are listed in Table 5, where we can see that, at most, attained coverage has a moderate-to-low correlation to the likelihood of fault detection for both versions of Branch Coverage. Lower coverage for Direct Branch Coverage does not entirely explain lower fault-detection efficacy.

Not all coverage is relevant to detecting faults. We can also analyze the "patch coverage"—the coverage attained over the lines of code related to the fault [4, 11]. The resulting patch coverage is listed in Table 6 for each coverage type, budget, and system. Overall, this table offers similar results, with traditional BC attaining 9.46% higher average coverage over patched lines with a two-minute budget and 8.06% with a ten-minute budget.

We can also see from Tables 3-6 that—unlike with the likelihood of fault detection—Direct Branch Coverage often benefits less than traditional BC from an increased search budget. In fact, the gap in overall attained coverage actually increases with the larger budget. However, the performance gap in attained *patch coverage* drops slightly (9.46% to 8.06%) as the budget increases. While this is not of the same significance as the improvement in fault-detection from a higher budget, it does suggest that increasing the budget tends to help Direct Branch Coverage cover the fault.

> Branch Coverage attains an average 7.94/9.40% higher Line Coverage and 9.09/10.41% higher Branch Coverage than Direct Branch Coverage (two/ten-minute budget), as well an average 9.46/8.06% higher Patch Coverage.

	Budget	Chart	Closure	Lang	Math	Time	Overall
Covered by BC, not DBC	120	13.52%	5.78%	11.39%	4.26%	5.22%	6.87%
	600	14.62%	8.17%	10.61%	3.75%	6.70%	7.66%
	% Change	8.14%	41.35%	-6.85%	-11.97%	28.35%	11.50%
Covered by DBC, not BC	120	5.29%	2.37%	3.21%	2.21%	2.76%	2.71%
	600	4.24%	2.24%	3.68%	2.33%	1.71%	2.63%
	% Change	-19.85%	-5.49%	14.64%	5.43%	-38.04%	-2.95%

Table 7: Average percent of code that one method covers and the other does not—broken down by budget and system.

	Budget	Chart	Closure	Lang	Math	Time	Overall
BC	120	42.85	17.86	73.76	30.36	53.77	36.35
	600	52.52	27.36	82.84	32.74	61.99	43.71
	% Change	22.57%	53.19%	12.31%	7.84%	15.29%	20.25%
DBC	120	48.12	17.69	76.13	29.47	59.77	37.31
	600	65.11	27.12	90.89	34.36	70.87	47.10
	% Change	35.31%	53.31%	19.39%	16.59%	18.57%	26.24%

Table 8: Average suite size—in number of tests—broken down by coverage type, budget, and system.

	Budget	Chart	Closure	Lang	Math	Time	Overall
BC	120	326.46	113.58	565.86	127.91	226.59	225.33
	600	505.14	224.35	629.55	146.35	329.26	305.13
	% Change	54.73%	97.53%	11.26%	14.42%	45.31%	35.41%
DBC	120	366.97	119.96	533.94	178.40	291.18	244.36
	600	599.96	233.01	680.33	209.08	385.12	347.13
	% Change	63.49%	94.24%	27.42%	17.20%	31.58%	42.06%

Table 9: Average suite size—in number of method calls—broken down by coverage type, budget, and system.

Stating that traditional Branch Coverage outperforms Direct Coverage in terms of total coverage does not offer the complete picture—the two metrics also cover *different* targets. In Table 7, we list the average percent of the code that is covered by BC and not DBC as well as the average percent of the code that is covered by DBC and not BC. We can see that each metric covers targets that the other does not. An average of 6.87-7.66% of the lines covered by traditional Branch Coverage are not touched by DBC. However, the reverse is also true. On average, 2.71% of the code is covered by Direct Branch Coverage and remains untouched by BC within the two-minute budget. At the ten-minute budget, DBC covers 2.63% of the program that is never covered by BC.

It is clear that the requirement for direct coverage imposes additional burdens on the test generator. Given a fixed budget, Direct Branch Coverage will attain a lower final level of coverage. To a certain extent, this can be alleviated by offering additional time for generation. However, we can also see that the differences in the final results are not due simply to this burden. The requirement for direct coverage changes how the test generation framework creates test cases, directing the search in different directions. Not only is the total coverage different, but the targets covered differ as well. In many cases, traditional Branch Coverage benefits from the context offered by indirect method calls. There are also cases where Direct Branch Coverage benefits from being forced to make direct calls.

> Traditional Branch Coverage and Direct Branch Coverage each cover different targets, again suggesting that each technique has situational applicability.

One other factor that can be used to analyze test suites is the size of the resulting suites. Suite size has been a focus in recent work, with Inozemtseva et al. finding that the size has a stronger correlation to efficacy than coverage level [6] and Gay has found the opposite [4]. In Table 8, we measure size in terms of average number of unit tests per suite. In Table 9, we measure size in terms of the length—the average number of method calls per suite.

Direct Branch Coverage results in suites that are 2.60% larger at the two-minute budget and 7.80% larger at the ten-minute budget. These suites also have somewhat longer tests, in terms of number of calls—8.50% and 13.76% longer at the two and ten-minute budgets. These results reflect the requirement for direct coverage. As we cannot cover methods indirectly, test cases will call methods and may need more test cases to achieve the same results.

Although this is a natural result of requiring direct coverage, it is also a potential source of concern. Each test added to a suite carries a human oracle cost. Direct Branch Coverage should, in theory, carry a lower *total* cost by producing more understandable test cases [10]. However, this benefit may be reduced by also requiring that more test cases be produced in the first place. In most cases, the increase in suite size is relatively modest—and unlikely to outweigh the potential benefits. However, we cannot confirm this at this time. In the future, we would like to more closely examine the human oracle costs and benefits of each approach.

> Direct Branch Coverage produces test suites with 2.60-7.80% more tests and 8.50-13.76% more method calls.

4.3 Comparing Situational Applicability

Our results indicate that each version of Branch Coverage was able to detect faults that the other was not and covered code that the other did not. There are clearly differences between the two forms of coverage that are not merely a result of the search budget, but come down to fundamental differences in how each variant is driven to attain coverage. By examining these situations, we can come to understand the situations where each technique excels and each technique falls short.

Fundamentally, freely allowing a generation framework to count indirect coverage in its total—as is the current standard practice—will bias the generator *towards* indirect coverage. Counting indirect coverage will rapidly accelerate the attainment of coverage. Covering a method through indirect calls removes the need to form input for that method directly. This will result in tests that can differ greatly from those created by humans. Coverage is attained through indirect calls, but these indirect calls may constrain the range of input that is used to call a particular method. In such cases, tests generated using traditional Branch Coverage could miss a fault that a human—or tests generated through Direct Branch Coverage—could detect, by not attempting input that would be tried through direct coverage.

This is the same scenario alluded to in Section 1. Although that example was relatively trivial, similar situations exist in the case examples studied. For instance, we can see such a situation in fault 106 for the Closure project[3]. The faulty version of this class lacks a check for a null object. This method, `canCollapseUnannotatedChildNames()` is called in several other places. It has no formal parameters. Rather, the execution path depends on the current state of class attributes. As a result, the probability of detecting the fault strongly depends on the context that the method is called in. If the method is called indirectly, the object being examined is unlikely to be null,

[3] https://github.com/rjust/defects4j/blob/master/framework/projects/Closure/patches/106.src.patch

as it has been manipulated within another method first. As the code of interest checks for ! = *null*, this does mean that indirect coverage will be attained. However, to detect the fault, we actually want the object to be null. In tests that cover the same code, Direct Branch Coverage will also try to make the object not null. However, the need for a direct call *also* means that we see more test cases with a null object. Traditional Branch Coverage calls the method fewer times, with a smaller range of input values.

A similar example can be seen in Math fault 102[4]. The affected method `chiSquare(double[] expected, long[] observed)` is called elsewhere in the class, making indirect coverage possible. Both Branch and Direct Branch Coverage attain full coverage of the patched code in all cases. However, coverage is less important than choosing the right input. Indirect coverage results in a smaller range of input being passed to the affected method, and a lower likelihood of fault detection. Direct Branch Coverage calls the method in a wider variety of configurations.

> Indirect coverage can limit the range of input used to cover a method, missing faults detected through direct calls.

Coverage is a prerequisite to fault detection, but it is not enough to ensure that faults are detected [4, 5]. Context matters—*how* a method is covered is more important than *whether* it was covered. As calls come through another method, indirect coverage limits the range of input passed to the affected method. In addition to the potential understandability benefits, requiring direct coverage gives the generator more freedom to choose how each method is covered.

Direct Branch Coverage is still outperformed by traditional Branch Coverage in many situations. This is because Direct Branch Coverage *entirely* ignores the context offered by indirect coverage. Even though the coverage attained through indirect calls is not counted, those calls are still being made. Methods do not exist in a vacuum, even if we pretend they do when measuring coverage. As these methods work together to perform tasks, faults may be caught by examining how the methods *interact* that are missed by looking at each method in isolation.

Interaction context helps us find faults in two common situations. In the first case, indirect coverage of the faulty method *allows* us to detect the fault while direct coverage does not. An example of this can be seen in Chart fault 5[5]. The affected method `addOrUpdate(XYDataItem item)` is part of a series of `addOrUpdate(...)` methods that take in various numeric primitives and, ultimately, cast them into an instance of `XYDataItem` that is passed into the affected method. Due to the difficulty of creating this non-standard data structure, neither BC or DBC cover this method directly. DBC occasionally detects this fault by accident—through an indirect call. However, because traditional Branch Coverage can explore the affected method indirectly, it is able to more reliably cover and trigger the fault.

Another example can be seen in Closure fault 52[6]. The affected method, `isSimpleNumber(String s)`, is supposed to return `true` if *s* has a length > 0 and has '0' as the first character. In the faulty version, the requirement of '0' is missing from the check. This is a case where both traditional and Direct Branch Coverage should be on similar footing. However, the affected method is indirectly called by method `getSimpleNumber(String s)`. Coverage of `getSimpleNumber` requires that the input string be a simple number already, which also guarantees indirect coverage of `isSimpleNumber`. This situation gives traditional Branch Coverage an advantage. The context offered by tracking indirect coverage allows EvoSuite to evolve tests that simultaneously consider both methods, rather than requiring each to be covered in isolation. As a result, traditional Branch Coverage reliably produces input that triggers the fault (that has a '0' as the first character).

The second type of situation where this context matters are cases where the faulty method calls another method, and indirect coverage of the second—non-faulty—method helps expose the fault in the calling method. We can see an example of this situation in Math fault 35[7]. The faulty version of the constructor for the class `ElitisticListPopulation` assigns values to the attribute `elitismRate`, whereas the fixed version assigns this value through the method `setElitismRate(double elitismRate)`. The setter method performs bounds-checking, preventing the use of illegal rates—those below 0 or larger than 1. Because traditional Branch Coverage is able to consider indirect coverage of `setElitismRate(...)`, it evolves input that leads to fault detection. Direct Branch Coverage is able to cover `setElitismRate(...)` in isolation, but the lack of guidance when calling the faulty constructor prevents EvoSuite from generating the right input.

> By ignoring indirect coverage, Direct Branch Coverage misses faults that emerge when covering method interactions.

Our takeaway from these observations is that both methods of Branch Coverage are flawed. The *intent* behind Direct Branch Coverage is reasonable, but ignoring the context offered by indirect coverage hobbles its performance. Methods do not exist in isolation, and the context offered by their interactions can help expose faults. Even if direct coverage offers human understandability benefits, we should not ignore indirect coverage entirely. Similarly, traditional Branch Coverage is driven towards indirect coverage to the point where the generated tests no longer resemble the tests created by human developers—raising the human oracle cost associated with their use, and potentially missing faults by constraining input choices. It is important to remember that a human tester's job is not done after indirect coverage is attained, and that a generation algorithm could benefit from direct examination of a method.

Fundamentally, coverage is not the true goal of testing. It is a means to judge progress, and a means to automate the creation of input, but what we really want are tests that *detect faults*. Fault detection requires not just coverage, but the right context—the input that will expose the fault. Both traditional and Direct Branch Coverage hold pieces of that context.

We believe that the test generation frameworks of the future should consider means of leveraging the benefits of each approach. For example, a new form of Branch Coverage could require direct

[4] https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/102.src.patch
[5] https://github.com/rjust/defects4j/blob/master/framework/projects/Chart/patches/5.src.patch
[6] https://github.com/rjust/defects4j/blob/master/framework/projects/Closure/patches/52.src.patch
[7] https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/35.src.patch

calls to consider each branch to be covered, but—rather than ignoring indirect coverage—the generator could use it as a means of weighting the fitness score. This type of approach may lend Direct Branch Coverage the context that it lacks on its own, and a small increase in generation budget may allow it to overcome the performance loss from the direct coverage requirement.

5 RELATED WORK

Advocates of adequacy criteria hypothesize that we should see a correlation between higher attainment of a criterion and the chance of fault detection for a test suite [5]. Researchers have attempted to address whether such a correlation exists for almost as long as such criteria have existed. Inozemtseva et al. provide a good overview of work in this area [6]. Branch Coverage is a common target for test generation [8, 10] and measurement [5].

Shamshiri et al. found that a combination of three state-of-the-art tools could identify 55.7% of the faults in the Defects4J database [11]. 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 traditional Branch Coverage to generate suites. Recent experiments by Gay compare a variety of fitness functions in terms of fault detection efficacy—including both Branch and Direct Branch Coverage [4]. They found that Branch Coverage is the most effective fitness function.

User studies conducted by Fraser et.al found that developers dislike tests that cover branches indirectly, because they are harder to understand and to extend with assertions [3]. Similarly, Almasi et al. and others have found that concern over the readability of generated suites has slowed industrial adoption of automated test generation [2]. Direct Branch Coverage is an attempt to address such issues [10]. Our study is the first to directly compare and contrast Direct and traditional Branch Coverage.

6 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 have used a single test generation framework, EvoSuite, as it is the only framework to implement both direct and indirect Branch Coverage. While results may differ between generation frameworks, we believe that the underlying trends would remain the same. Several of the observed differences between direct and indirect Branch Coverage are a natural result of the requirements of each method, not how they were implemented in EvoSuite. To control experiment cost, we have only generated ten test suites for each combination of fault, budget, and coverage variant. It is possible that larger sample sizes may yield different results. However, we believe that the 14,120 test suites used in analysis are sufficient to draw stable conclusions.

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

7 CONCLUSIONS

While adequacy criteria offer an end-point for testing, they do not mandate *how* targets are covered. Branch Coverage may be attained through *direct* calls to methods, or through *indirect* calls between methods. In order to study the costs and benefits of each approach, we have judged the efficacy of test suites generated using both variants of Branch Coverage against a set of real faults.

We have found that direct coverage imposes a cost in terms of coverage and likelihood of fault detection. However, traditional and Direct Branch Coverage cover portions of the code and detects faults that the other does not. By examining methods in isolation, Direct Branch Coverage is less constrained in the input it uses to cover each method. However, traditional Branch Coverage is able to leverage the context in which methods interact with each other to detect faults that emerge from those interactions. There are clear situations where each form of coverage is more suited to detecting a particular fault than the other. Importantly, both also have important limitations not possessed by the other. This indicates that both variants have value as part of a broader testing strategy, and that future approaches to test generation should leverage the strengths of both.

REFERENCES

[1] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 352–361, March 2013.

[2] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)—Software Engineering in Practice Track (SEIP)*, ICSE 2017, New York, NY, USA, 2017. ACM.

[3] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Trans. Softw. Eng. Methodol.*, 24(4):23:1–23:49, Sept. 2015.

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

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

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