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

DOI: xxx/xxxx

RESEARCH PAPER

Exploring the Interaction of Code Coverage and Non-Coverage Objectives in Search-Based Test Generation

Afonso Fontes, Gregory Gay*, Robert Feldt

[1]Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Sweden

Correspondence
*Gregory Gay, Email: greg@greggay.com

Abstract

Context: Search-based test generation typically targets structural coverage of source code. Past research suggests that targeting coverage alone is insufficient to yield tests that achieve common testing goals (e.g., discovering situations where a class-under-test throws exceptions) or detect faults. A suggested alternative is to perform multi-objective optimization targeting both coverage and additional objectives directly related to the goals of interest. However, it is not fully clear how coverage and goal-based objectives interact during the generation process and what effects this interaction will have on the generated test suites.

Objectives: We assess five hypotheses about multi-objective test generation and the relationships between coverage-based and goal-based objectives, focusing on the effects on coverage, goal attainment, fault detection, test suite size, test case length, and the impact of the search budget.

Methods: We generate test suites using the EvoSuite framework targeting Branch Coverage, three testing goals—Exception Count, Output Coverage, and Execution Time—and combinations of coverage and goal-based objectives.

Results: Targeting multiple objectives does not reduce code coverage, yields no or minor reductions in goal attainment, and detects more faults compared to single-target configurations. It produces larger test suites, but test case length is not increased. The benefits of multi-objective optimization are often more limited than hypothesized in past research, but improved fault detection is still sufficient to recommend multi-objective optimization over targeting coverage or testing goals alone.

Conclusion: Our study offers insights and guidance into how coverage and goal-based objectives interact during multi-objective test generation.

KEYWORDS:
Automated Test Generation, Search-Based Test Generation, Coverage Criteria, Adequacy Criteria, Branch Coverage

1 | INTRODUCTION

Structural coverage criteria measure the percentage of the source code that has been executed according to a set of criterion-specific rules regarding (a) which code structures should be executed, and (b) how those structures should be executed [6, 12, 22]. Two of the most common criteria include Statement Coverage—which mandates that all code statements be executed, but

places no constraints on how they are executed—and Branch Coverage—which mandates that all control-diverting statements (e.g., `if`, `case`, and loop conditions) evaluate to each of their possible outcomes [11].

Coverage measurement is a common advisory activity for testers [6]. The current percentage of coverage attained can serve as an approximation of "how much testing" has been conducted, and missed coverage goals can serve as the targets of additional test cases. Because the attainment of most coverage criteria can be automatically measured through program instrumentation and execution analysis, such criteria have also become the de facto basis of automated test generation—especially techniques such as search-based test generation, fuzzing, and symbolic or concolic execution [5, 37].

Consider, for example, search-based test generation. In search-based test generation, metaheuristic optimization algorithms sample from the space of possible test inputs to identify those that maximize or minimize *fitness functions*—numeric scoring functions representing properties of interest [37]. Coverage criteria serve as natural fitness functions, often associating each code structure of interest with a score representing how close an execution came to executing that structure in the manner prescribed by the criterion—e.g., how much x would need to change for the condition ($x == 0$) to evaluate to `true` within a particular control structure [7].

Coverage-directed testing is ubiquitous in automated test generation because structural coverage is easy to measure, easy to translate into an optimization target, and is hypothesized to have a correlation to the probability of fault detection [11]. However, concerns have been raised about its use as the primary target of automated generation [22, 24, 26]. We have previously conducted large-scale case studies on coverage-directed test generation, focusing on model and search-based test generation [19, 22, 51]. These studies have yielded important observations about the efficiency and effectiveness of coverage-directed test generation—at least, in the manner it is generally employed.

First, we have observed that achieving structural coverage is a reasonable *starting point* for effective automated test generation. For example, we observed that coverage was the single strongest predictor of the likelihood of fault discovery [51]. That is, if we want to detect potential faults, *we must execute the code*. The same basic observation holds for many other goals a tester may have. If we want to expose situations where the code can crash, *we must execute the code*. If we want to show that performance or reliability targets are met, *we must execute the code*. Other testing goals—e.g., diversity, exposing interaction faults, and more—similarly benefit from exploration of the codebase. Targeting code coverage during search-based test generation is an effective and efficient method of exploring a wide range of program behaviors [51]. Therefore, even if a testers' goals lie beyond code coverage, *coverage is generally required to achieve those goals*.

However, we also observed that code coverage *alone* is a poor basis for producing test suites that meet these goals. In our past work, coverage only had a moderate correlation to the likelihood of fault detection [51], and was often weaker than random generation at detecting code mutations [22]. *Many different inputs can generally cover the same coverage goals*. While some coverage criteria are stricter than others, the majority impose few or no constraints on how code is executed [22, 23, 39, 51].

"How" is important. Testers rarely design tests for the sole purpose of attaining coverage [24, 27]. In practice, tests are designed around specifications and coverage is used to identify clear weaknesses in the suite [6]. That is—coverage serves an advisory role for testers, rather than the primary basis of test design. If we want to expose crashing code, we select input with a high probability of triggering a crash. If we want to violate performance requirements, we select input with a high probability of slowing program execution. If there are multiple bugs in a branch, we typically need diverse inputs to uncover them all, as well as to cover the specification [18, 17]. In other words, while research in automated test generation has predominately focused on code coverage, *coverage alone is not enough to ensure that testing goals are met*.

Search-based test generation offers potential solutions to this challenge. First, rather than optimizing fitness functions related to code coverage, one could attempt to optimize fitness functions based directly on goals of interest—for example, there are fitness functions that directly reward discovery of crashes or that assess performance. Second, rather than choosing coverage or goal optimization, one could attempt *multi-objective optimization*, where the combination of structural coverage and additional goal-based fitness functions are simultaneously targeted.

Multi-objective optimization is a particularly compelling solution, as each fitness function optimized shapes the resulting test suite. Our past research suggests that such a pairing can lead to better test suites than targeting coverage *or* a goal of interest alone [19, 51]. Consider a common testing goal—identifying situations where the system-under-test (SUT) throws an exception. This is a non-trivial goal, as we rarely know up-front which exceptions could be thrown. Targeting coverage may not satisfy this goal, as exception-triggering input will only be chosen if it uniquely enhances coverage. We could alternatively try to directly maximize the number of exceptions thrown. However, this count offers no feedback, more exceptions will only be discovered by random chance. We observed situations where targeting both offered feedback missing when targeting either alone—with the exception count biasing the input used to attain coverage, and branch coverage offering a means to explore the code base.

These observations suggest the potential benefit of blending code coverage and goal-based fitness functions. While multi-objective test generation has been previously proposed and attempted (e.g., [52, 42, 51]), the interaction between objectives during this optimization—in particular, the interaction between coverage and goal-based fitness functions—has not been investigated previously. Understanding this interaction is important, as this understanding can influence the selection of optimization targets, the design of new fitness functions, and the development of new test generation tools.
Therefore, in this study, our goal is to assess and explore five hypotheses about this interaction:

Hypothesis 1: The inclusion of goal-based fitness functions as additional generation targets will not have an impact on the attainment of code coverage, as compared to targeting coverage alone.

That is, the hypothesis that targeting multiple objectives will not affect the evolution of code coverage during the test generation process—raising or lowering the final quantity of coverage attained, changing the specific coverage goals covered, or affecting the rate at which coverage is attained during the generation process.

Hypothesis 2: Targeting both coverage and a goal-based fitness function will not have an impact on the attainment of goal-based fitness functions, as compared to targeting coverage or a goal-based fitness function alone.

Similar to Hypothesis 1, this hypothesis states that targeting multiple objectives will not affect the final fitness values of the goal-based objectives—raising or lowering goal attainment when compared to targeting a goal-based or a coverage-based objective alone.

Hypothesis 3: Targeting both coverage and a goal-based fitness function will not have an impact on the fault detection of generated test suites, as compared to targeting coverage or a goal-based fitness function alone.

This hypothesis states that targeting multiple objectives will not increase or decrease the likelihood that the generated test suites detect faults or the number of tests that fail when a fault is detected.

Hypothesis 4: Targeting both coverage and a goal-based fitness function will not have an impact on the size of the test suite and the average test length, as compared to targeting coverage or a goal-based fitness function alone.

Targeting multiple objectives could increase the number of test cases in the generated suites or increase the number of interactions in individual test cases, as each targeted objective adds additional obligations that the test suite must achieve. These obligations each may require distinct test input and setup to achieve, leading to the need for more or longer test cases.

Hypothesis 5: An increase in the search budget will not lead to increased attainment of each objective.

This hypothesis considers the effect of the search budget on fitness attainment. We further hypothesize that the effects that we observed when exploring the previous hypotheses will hold at higher search budgets. For example, if multi-objective optimization leads to higher fault detection at a limited search budget than single-objective optimization, we hypothesize that it will also do so at a higher search budget.

To assess these hypotheses, we target Branch Coverage—the most common structural coverage criterion [6]—as well as three specific testing goals:

- We further explore the goal of discovering situations where the SUT can crash.
- The discovery of situations that could violate performance goals—based on the maximization of execution time [40].
- Ensuring that test suites maximize coverage of diverse behaviors [18], specifically output diversity of the tested functions, which has been hypothesized to lead to faster coverage attainment and higher likelihood of fault detection [4].

Our study offers insight into how coverage and goal-based objectives interact during multi-objective test generation, with a focus on how this interaction affects code coverage, goal attainment, fault detection, the size of the test suite, and the length

```
1         @Test
2         public void testPrintMessage() {
3             String str = "Test Message";
4             StringUtils tCase = new StringUtils(str);
5             tCase.removeWhitespace();
6             assertEquals("TestMessage", tCase.getString());
7         }
```

Figure 1 Example of a unit test case written using the JUnit notation for Java.

of test cases. This research offers a starting point for exploring how search-based test generation can be adapted for particular goals, product domains, execution scenarios, or code structures, enables guidance on how to use test generation to meet tester goals, and can influence the creation of more efficient and effective test generation techniques and tools.

2 | BACKGROUND

2.1 | Unit Testing

Testing can be performed at various levels of granularity. In this research, we focus on *unit testing*, where test cases target small segments of code that can be tested in isolation [47]. Unit tests are written as executable code, which can be re-executed on demand by developers. We refer to a purposefully grouped set of test cases as a *test suite*. Unit testing frameworks exist for many programming languages, such as JUnit for Java, and are integrated into most development environments.

An example of a unit test, written in JUnit, is shown in Figure 1. A unit test consists of a *test sequence (or procedure)*–a series of method calls to the class-under-test (CUT)–with *test input* provided to each method. Then, the test case will validate the output of the called methods and the class variables against a set of encoded expectations—the *test oracle*—to determine whether the test passes or fails. In a unit test, the oracle is typically formulated as a series of assertions on the values of method output and class attributes [9]. In the example in Figure 1, the *test input* consists of passing a string to the constructor of the `StringUtils` class, then calling its `removeWhitespace()` and `getString()` methods. We use an assertion to ensure that a space is removed from the input.

2.2 | Adequacy (Coverage) Criteria

When testing, developers must judge both whether the tests they have written are effective and whether they have created enough test cases. Adequacy criteria have been developed to provide developers with guidance regarding these topics [6].

Each adequacy criterion prescribes, for a given program, a set of goals—referred to as *test obligations*. Each obligation represents one constraint that the produced test suite must fulfill. If tests fulfill the test obligations, than testing is deemed "adequate" with respect to faults that manifest through the structures of interest to the criterion. Most adequacy criteria are based on execution of structural elements of the software. In such cases, an obligation may be expressed as the selection of an individual element of the source code—e.g., a statement, a branch of the software's control flow, or a boolean expression—and the conditions under which that element must be executed—e.g., the chosen expression evaluates to `true` or `false` [47, 22]. However, there are also adequacy criteria based, e.g., on coverage of formal requirements through test cases [53] or detection of synthetic faults (mutants) planted in the source code [19].

Adequacy criteria have seen widespread use in software development. Structural coverage is routinely measured as part of automated build processes [23][1], and is mandated by safety standards in critical domains such as automotive [14] and avionics [41]. It is easy to understand the appeal of adequacy criteria. They offer clear checklists of testing goals that can be objectively evaluated and automatically measured through program instrumentation and execution analysis [23]. These same qualities make adequacy criteria ideal for use as automated test generation targets [7].

One of the most common adequacy criteria is *Branch Coverage*. A branch refers to an outcome of any program statement that can cause program execution to diverge down a particular control flow path, such as the conditions in `if`, `case`, or loop

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

```
1   public class StringUtils{
2       private String str;
3       ...
4       public void removeWhitespace(){
5           String modified = "";
6
7           for (int index = 0; index < this.str.length(); index++) {
8               char ch = this.str.charAt(index);
9               if (!Character.isWhitespace(ch)) {
10                  modified += ch;
11              }
12          }
13
14          this.str = modified;
15      }
16      ...
17  }
```

Figure 2 Subset of the class-under-test in Figure 1.

```
1   public void removeWhitespace();
2     Code:
3        0: ldc           #7              // String
4        2: astore_1
5        3: iconst_0
6        4: istore_2
7        5: iload_2
8        6: aload_0
9        7: getfield      #9              // Field str:Ljava/lang/String;
10      10: invokevirtual #15             // Method java/lang/String.length:()I
11      13: if_icmpge     46
12      16: aload_0
13      17: getfield      #9              // Field str:Ljava/lang/String;
14      20: iload_2
15      21: invokevirtual #21             // Method java/lang/String.charAt:(I)C
16      24: istore_3
17      25: iload_3
18      26: invokestatic  #25             // Method java/lang/Character.isWhitespace:(C)Z
19      29: ifne          40
20      32: aload_1
21      33: iload_3
22      34: invokedynamic #31,  0         // InvokeDynamic #0:makeConcatWithConstants:
23                                        // (Ljava/lang/String;C)Ljava/lang/String;
24      39: astore_1
25      40: iinc          2, 1
26      43: goto          5
27      46: aload_0
28      47: aload_1
29      48: putfield      #9              // Field str:Ljava/lang/String;
30      51: return
```

Figure 3 Java Bytecode of the `removeWhitespace()` method from Figure 2.

definitions. Branch Coverage requires that all outcomes of all control-diverging statements are executed at least once by the test suite under assessment.

To give an example, consider the `removeWhitespace()` method being tested in Figure 1, whose code is depicted in Figure 2. In this method, there are two program statements that affect the control flow—the loop condition on line 7 and the `if`-condition on line 9. To achieve branch coverage over this method, both conditions must evaluate to true and false at least once when the test suite is executed. In other words, there are four test obligations that must be fulfilled.

By default, coverage obligations are formulated over the source code. However, in Java, test obligations are often instead formulated and measured over the bytecode representation as this form is easier and more efficient to instrument and monitor. The bytecode representation of `removeWhitespace()` is shown in Figure 3. The same control-altering expressions are present, on lines 11 and 19. Branch Coverage requires that both lines evaluate to true and false.

Branch Coverage is arguably the most commonly used coverage criterion, with ample tool support and industrial adoption [46]. For example, branch coverage measurement is built into the popular IntelliJ IDEA development environment. Therefore, we focus on Branch Coverage in this study as a representation of structural coverage criteria.

2.3 | Search-Based Test Generation

Manual creation of a large volume of test cases can be tedious and expensive. Automation of aspects of test creation, such as test input selection, can reduce and focus the required manual effort [5]. Search-based test generation frames input selection as a search problem, where metaheuristic optimization algorithms attempt to identify test input that best embody properties that testers seek in their test cases [37, 5].

These properties are assessed using one or more fitness functions—numeric scoring functions. The metaheuristic embeds a strategy for sampling solutions from the space of possible inputs, often based on a natural process such as evolution or swarm behavior [25]. In test generation, a "solution" is often either a single test case or a full test suite. The metaheuristic uses the selected fitness functions to assess solution quality, offering feedback to guide the selection and improvement of solutions over a series of generations. Search-based test generation has proven to be a flexible [1], scalable [33], and competitive [51, 52] method of automated test generation.

The most common metaheuristics for search-based test generation are genetic algorithms, which are modeled after the natural evolution of a population [16]. While specific aspects vary, a "typical" test generation follows these steps:

- An initial population of solutions is randomly generated. Each solution represents a test suite, containing test cases.
- Each generation, the fitness score of each solution is calculated and a new population is created. This population is formed through four sources of solutions:
  - One of the best solutions may be carried over to the new population intact.
  - At a certain probability, elements of two solutions will be combined to create two "children" (crossover). For example, the children may blend test cases from the parents.
  - At a certain probability, a solution can be mutated—e.g., a test case may be modified.
  - At a certain probability, a new randomly generated solution will be added to the population to maintain diversity.
- When the search budget—typically expressed in time or number of generations—expires, the best solution is returned.

When multiple fitness functions are targeted by an optimization algorithm, each fitness function and its individual sub-goals collectively influence the final solution generated. The interaction between fitness functions—or even between the sub-goals of one or more fitness functions—can be either positive or negative.

In the positive case, one or more fitness functions or sub-goals can provide missing feedback that is needed to optimize other fitness functions or sub-goals. This is common in situations where the fitness landscape is relatively flat, as is the case for functions based on simple counts—e.g., the number of exceptions thrown. Such fitness functions offer little feedback for improving solutions, e.g., for detecting new exceptions. However, in previous work, we observed cases where pairing such functions with more informative fitness functions (e.g., ones based on distance measurements) offered the missing feedback needed to identify additional exceptions [19, 51]. Even informative fitness functions can have a positive symbiotic relationship. For example, there are multiple methods of formulating a fitness function for Branch Coverage, each with differing fitness landscapes. We previously observed situations where targeting multiple forms of Branch Coverage simultaneously yielded improved performance over targeting any one form [20].

In the negative case, there are situations where optimization of a subset of fitness functions or sub-goals can lower the attainment of other functions or sub-goals—i.e., there are implicit trade-offs between these functions or sub-goals. As an example, EvoSuite offers a fitness function that rewards executing methods without an exception being triggered. This function directly contradicts the earlier-mentioned function that rewards suites where more exceptions are thrown. It is possible to optimize both functions by constructing a test suite where these functions are optimized in *separate* test cases [19, 51]. However, overcoming the contradiction between functions requires investing additional time in the search process, and can result in larger and more cumbersome test suites.

This theoretical relationship between fitness functions and sub-goals of fitness functions is not well understood in the context of search-based test generation. While we have discussed examples observed in prior work [19, 29, 51], the exact nature of this relationship has not been deeply explored. One of the goals of this study is to further empirically investigate this relationship.

3 | RELATED WORK

Multi and many-objective optimization algorithms have become increasingly common in search-based test generation [48]. Even if the goal of the test generation process is solely code coverage, coverage can quickly be gained by representing each test obligation as an independent objective and applying multi or many-objective optimization [43]. Multiple studies have compared different algorithms for multi and many-objective optimization in terms of coverage achieved (e.g., [8, 10, 32, 45]). However, these studies focused solely on coverage-based fitness and have not examined the interaction between coverage-based and goal-based fitness functions.

The number of exceptions or crashes discovered is a common secondary objective in search-based test generation, optimized in conjunction with coverage-based fitness functions (e.g., [3, 30, 49, 51]). Others have explored combinations of coverage criteria with non-functional criteria during test generation or test suite minimization, such as memory consumption [31] or execution time [58]. While these represent multi-objective optimization of coverage and goal-based fitness functions, these studies do not examine how these fitness functions interact, e.g., how the combinations affect coverage or fault detection.

Rojas et al. examined multi-objective optimization of Line Coverage—a structural coverage criterion—and additional fitness functions [49]. Relevant to our work, they also include two of the same goal-based objectives that we focus on, exception count and output diversity. They found that adding additional fitness functions led to only a minimal loss in the final percentage of Line Coverage achieved. They also found that coverage of secondary criteria increased over when Line Coverage was targeted alone. Therefore, there is a partial overlap in our focus. However, they only examined the final level of coverage and focused on different aspects of test generation. We address a broader set of hypotheses and examine coverage attainment more deeply.

Palomba et al. examine optimization of Branch Coverage and fitness functions intended to improve test quality based on the cohesion and coupling of test cases [42]. They found that targeting these quality objectives could increase code coverage over targeting coverage alone.

Weiglhofer et al. showed that coverage-directed test generation can be used to complement test generation based on testing goals [56]. In their approach, humans develop "test purposes", specifications used in conjunction with formal models to generate test cases. Coverage-directed testing is then used to generate tests for parts of systems not covered by the test purposes. They do not apply multi-objective optimization, but the core concept is similar.

We previously examined the likelihood of fault detection of test suites generated targeting various fitness functions [19, 51]. Much of this research focused on single-objective optimization. However, we did find that some combinations of objectives, such as Branch Coverage and the exception count, had a higher likelihood of fault detection than targeting Branch Coverage or exception count alone [51]. This past work partially addresses one of our hypotheses, but we replicate this work and examine that hypothesis more closely in this study.

Zhou et al. propose an approach, "smart selection", for selecting a subset of test obligations when targeting multiple coverage-based fitness functions for test generation [60]. Their approach reduces redundancy between fitness functions and eases optimization difficulty. McMinn et al. have also proposed using search techniques to evolve new coverage criteria that combine features of existing criteria [38]. In previous work, we also used reinforcement learning to dynamically select the fitness functions targeted during multi-objective test generation [2]. We demonstrated that fitness functions could be identified that increased attainment of common testing goals for particular classes-under-test. However, these studies do not examine the interaction between coverage and goal-based fitness functions.

4 | METHODS

Our aim in this research is to examine the interaction between coverage-based and goal-based fitness functions during multi-objective test generation. In Section 1, we raised five informal hypotheses about how these objectives could interact. We assess those hypotheses by addressing the following specific research questions:

- **RQ1:** How is the Branch Coverage of generated test suites influenced by targeting additional goal-based fitness functions, compared to targeting Branch alone?
  - **RQ1.1:** How is the final percentage of attained Branch Coverage influenced?
  - **RQ1.2:** How is the set of satisfied Branch Coverage obligations influenced?
  - **RQ1.3:** How is the evolution of coverage attainment influenced?

- **RQ2:** How is the attainment of testing goals by generated test suites influenced by targeting Branch Coverage in addition to a goal-based fitness function, compared to targeting coverage or a goal-based fitness function alone?
- **RQ3:** How is the fault detection of generated test suites influenced by targeting Branch Coverage in addition to a goal-based fitness function, compared to targeting coverage or a goal-based fitness function alone?
- **RQ4:** How is the suite size and test case length of generated test suites influenced by targeting Branch Coverage in addition to a goal-based fitness function, compared to targeting coverage or a goal-based fitness function alone?
- **RQ5:** What influence does the search budget have on Branch Coverage, goal attainment, fault detection, test suite size, and test case length attained by suites targeting different fitness function configurations?

As discussed in Section 1, we focus on three concrete testing goals: (1) discovery of scenarios where exceptions are thrown ("Exception Count"), (2) discovery of scenarios where the execution time may violate performance goals ("Execution Time"), and (3), maximization of output diversity ("Output Coverage"). To address these research questions, we have performed the following experiment, targeting Branch Coverage, these three goals, and combinations of Branch Coverage with each goal:

1. **Collected Case Examples:** We have selected 93 case examples from the Defects4J fault dataset, from 14 Java projects (Section 4.1).
2. **Defined Test Generation Configurations:** We selected three single-objective configurations (Branch Coverage, Exception Count, Output Coverage) and three multi-objective configurations (Branch Coverage plus each testing goal listed above) and two search budgets (180 and 300 seconds) to target in our experiments (Section 4.2).
3. **Generated Test Suites:** For each class modified by each case example, fitness function configuration, and search budget, we generated 10 test suites using EvoSuite. We target the fixed version of each class-under-test (CUT) (Section 4.2).
4. **Monitored Coverage Evolution:** We monitor how satisfaction of Branch Coverage obligations changes over the course of each invocation of EvoSuite (Section 4.3).
5. **Recorded Generation Statistics:** For each suite generated, at the end of the generation process, we record information on Branch Coverage obligation satisfaction, fitness values for each targeted function, test suite size, and test case length (Section 4.3).
6. **Removed Non-Compiling and Flaky Tests:** Any tests that do not compile, or that return inconsistent results, are removed (Section 4.3).
7. **Assessed Fault-finding Effectiveness:** We measure the number of faults detected, the proportion of test suites that detect each fault to the number generated (likelihood of fault detection), and number of failing tests in each suite (Section 4.3).
8. **Analyzed the Collected Data:** We address the research questions using the data gathered above (Section 4.4).

We make a replication package available containing the data collected in this experiment: https://doi.org/10.5281/zenodo.11047567
We also make available our modified version of EvoSuite:

- Code: https://github.com/afonsohfontes/evosuite
- Executable: https://github.com/afonsohfontes/defects4j/tree/master/framework/lib/test_generation/generation

4.1 | Case Example Selection

Defects4J is an extensible database of real faults extracted from Java projects [28][2]. The current dataset, version 2.0.1, consists of 835 faults from 17 Java projects. To control experiment costs, in this study, we aimed to select a sample of approximately 100 faults, chosen to reflect the proportion of faults-per-project in the full dataset. To select this sample, we initially selected **206**

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

Project	Faults Selected	Total
Chart	7, 6, 10, 8, 3, 5	6
Cli	27, 7, 29, 28, 1, 10, 3, 40, 2, 5	10
Closure	161, 74, 19, 154, 162, 164, 37, 55, 41, 70, 12, 71, 5	13
Codec	7, 6, 17, 1, 2	5
Collections	25, 26	2
Compress	2, 47, 46	3
Csv	1	1
Gson	3, 4, 5, 6	4
JacksonCore	11	1
JacksonDatabind	62, 93, 111, 112	3
Jsoup	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 55, 60, 77	16
Lang	4, 5, 6, 8, 9, 10, 11, 12, 41, 55, 64, 65	12
Math	95, 11, 87, 81, 100, 39, 90, 41, 3, 49, 40, 2	12
Mockito	6, 8, 37, 15, 2	5
	Total	**93**

Table 1 Subset of Defects4J faults selected for this study.

faults at random, sampled based on the number of faults-per-project in the full dataset. We then generated test suites targeting Branch Coverage and the three multi-objective configurations following the procedure described in Section 4.2, and omitted faults where either:

- Errors prevented the completion of 10 valid trials for all configurations, where a test suite was generated and all data collection completed successfully.
- Where the average Branch Coverage was below 5%—we judged that the research questions could not be reliably answered without a minimal level of coverage being reached over the classes-under-test.

This filtering process ultimately resulted in a set of **93** faults used in this study, listed in Table 1. 59 faults were excluded due to issues with the test generation or data collection processes, while 54 faults were excluded due to low Branch Coverage.

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

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

4.2 | Test Generation Configuration

In this study, we make use of the EvoSuite unit test generation framework for Java [50]. EvoSuite is mature, actively maintained, and has been successfully applied to a wide variety of projects [51, 52]—even winning multiple tool competitions (e.g., [55]). Specifically, we make use of a modified version of EvoSuite version 1.2.1, where we have added an additional fitness function—Execution Time—as well as additional monitoring and data collection capabilities.

Test Generation Algorithm

We make use of EvoSuite's "whole test suite generation" Genetic Algorithm [50][3]. In this implementation of whole test suite generation, each solution represents a full test suite—in contrast to approaches where a solution represents a single test case. Then, rather than targeting one obligation (sub-goal) of each fitness function one-by-one, fitness is measured over all obligations of each fitness function at the same time.

[3]This implementation of whole test suite generation has been replaced as the default optimization algorithm in EvoSuite by DynaMOSA, a many-objective optimization algorithm [44]. While DynaMOSA has been shown to achieve better coverage than whole test suite generation in some experiments [10], we use whole test suite generation to enable clearer comparison to our past research [19, 51]. Further, the DynaMOSA algorithm explicitly considers the code structure, and goal-based fitness functions cannot be decoupled from structural coverage.

In traditional multi-objective optimization algorithms, such as NSGA-II [15], an attempt is made to balance fitness function attainment, and each fitness function is treated as independent. In contrast, in this implementation of whole test suite generation, a single aggregate fitness score is calculated. The fitness for a test suite T over the class-under-test C is:

$$\text{fitness}(T, C) = \sum_{f \in F} \hat{f}(T, C) \tag{1}$$

That is, the aggregate fitness is the sum of the normalized score of each fitness function. EvoSuite treats all optimizations as *minimization* problems, where lower fitness scores represent better solutions.

Fitness Function Configurations

We execute EvoSuite for each case example utilizing six fitness function configurations, representing three single-objective configurations (Branch Coverage, Exception Count, and Output Coverage) and three multi-objective configurations (Branch Coverage with Exception Count, Output Coverage, and Execution Time)[4]. The fitness functions are defined as follows:

- **Branch Coverage**: As defined in Section 2, Branch Coverage requires that all outcomes of all control-diverging statements are executed at least once by a test suite. For search-based test generation to be most effective, a fitness score should offer feedback to help guide the identification of better solutions. To that end, the most effective fitness functions tend to encode information about the *distance* to satisfying any unsatisfied goals. Therefore, rather than simply measuring whether each test obligation is covered or not, the fitness calculation for Branch Coverage instead embeds information—for each test obligation—on *how close execution came to satisfying that obligation*.

  The branch coverage fitness function is a minimization of the following, where T refers to the test suite and B represents the set of test obligations. Each test obligation, $b \in B$, represents a control-diverging program statement and a desired outcome for that statement (`true` or `false`).

  $$\text{fitness}(T, B) = \sum_{b \in B} d(T, b) \tag{2}$$

  where $d(T, b)$ is defined as:

  $$d(T, b) = \begin{cases} 0 & \text{if } b \text{ has been satisfied} \\ \overline{d_{min}(t \in T, b)} & \text{if } b \text{ has been evaluated at least twice} \\ 1 & \text{otherwise} \end{cases} \tag{3}$$

  In the case where an obligation has not been satisfied, $\overline{d_{min}(t \in T, b)}$ represents the *branch distance*—the magnitude of change in execution that would be needed to achieve the targeted outcome for that control-diverging statement. The branch distance is determined based on how the condition has been formulated, following a standard set of formulae [7]. In this case, the minimal observed value of the branch distance is used in the fitness calculation, and is normalized to be between 0–1.

- **Exception Count:** This fitness function represents the goal of causing the class-under-test to throw as many exceptions as possible—either declared or undeclared. The fitness function is a minimization of the following formula, where T refers to the test suite, $E_{discovered}$ represents the number of exceptions discovered during the current generation process to date, and E_{thrown} represents the number thrown by the current solution T.

  $$\text{fitness}(T) = 1 - \frac{E_{thrown}}{E_{discovered}} \tag{4}$$

  $E_{discovered}$ is the number of unique exceptions thrown by *all* assessed solutions to date, while E_{thrown} is the number of unique exceptions thrown by the current test solution-under-assessment, which may not throw exceptions that were thrown in previous solutions. Therefore, $E_{thrown} \leq E_{discovered}$. As the number of possible exceptions that a class can throw cannot be known ahead of time, the number of test obligations may change each time EvoSuite is executed on a CUT.

- **Execution Time:** This fitness function represents a scenario where we seek test suites that could uncover potential violations of performance requirements, manifested as a test suite—containing individually short test cases—that take

[4]Due to technical details of its implementation, we are unable to target Execution Time without also targeting Branch Coverage. Therefore, Execution Time cannot currently be targeted as a single-objective configuration.

excessive time to execute. We have added a new fitness function to EvoSuite for this purpose, which is calculated as follows:

$$\text{fitness}(T) = 1 - \frac{Time_{current}}{Time_{max}} + penalty \tag{5}$$

Where $Time_{current}$ represents the execution time of the solution currently under assessment and $Time_{max}$ is the largest execution time of any solution assessed during the current execution of the test generation framework.

One avenue to generate test suites with long execution times is simply to generate excessively long test cases that call many methods. Therefore, to prevent the generation of overly bloated test cases, the fitness calculation applies a penalty based on the average test case length within the suite:

$$penalty = 0.1 \times \frac{Length_{current}}{Length_{max}} \tag{6}$$

- **Output Coverage:** This configuration represents the goal of generating test suites that cover many different types of outcomes of the methods of the CUT. A tester may seek such diversity for two reasons. First, increased output coverage is hypothesized to lead to earlier and potentially higher code coverage [18, 17], and second, to potentially increase fault detection over pure white-box techniques [4].

  Output coverage rewards diversity in the method output by mapping return types to a list of abstract values—Alshahwan and Harman provide a detailed explanation, including fitness formulae [4]. A test suite satisfies output coverage if, for each public method in the CUT that returns a data type covered by the fitness function, at least one test yields a concrete return value matching each abstract value. For numeric data types, distance functions similar to the branch distance offer feedback using the difference between the chosen value and the targeted abstract values.

We have selected these three goal-based objectives because they reflect three different, non-overlapping, goals that a tester may have. Exception Count reflects the desire of testers to discover situations where the software can crash, as unexpected crashes can indicate a lack of robustness in the code-under-test. Crashes can occur in any software, can be detected without the need for specialized test oracles, and are the target of many test generation tools (e.g., fuzzers [35]). Output Coverage indicates that many different program behaviors have been triggered, potentially ensuring that the actual requirements of the code have been thoroughly exercised [4]. Finally, the execution time represents an important non-functional property of software—its performance. Even if the correct execution outcome occurs, slow performance may degrade user satisfaction [59]. While other goal-based fitness functions exist, we believe that these three exemplify three important, common, and distinct testing goals.

Search Budgets

Two search budgets were used—180 seconds and 300 seconds per class. This allows us to examine how an increased search budget affects the test suites produced by each single and multi-objective configuration.

Generation Procedure

Test suites are generated individually for each of the classes modified to fix each fault chosen from Defects4J. We repeat generation a fixed number of times for each class, fitness function configuration, and search budget.

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

Test suite generation and execution were performed on virtual machines, each configured with 4 vCPUs, 8GB of RAM, and 20GB of storage, running a server version of Ubuntu 18.04.4 LTS. Each virtual machine was dedicated to executing experiments for a specific subset of faults and fitness function configurations, ensuring that experiments remained isolated and independent to ensure result reliability.

To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. We also disable test suite reduction, an optional procedure that removes redundant test cases at the end of the generation process. We do this to maintain traceability between intermediate and final test suites during suite evolution. All other settings were kept at their default values. As results may vary, we performed 10 trials for each CUT, fitness function configuration, and search budget. This resulted in the generation of 11160 test suites (two budgets, ten trials, six configurations, 93 faults).

Generation tools may generate flaky (unstable) tests [52]. For example, a test case that makes assertions about the system time will only pass during generation. We automatically remove flaky tests. First, all non-compiling test suites are removed. Then, each remaining test suite is executed on the fixed version of the CUT. If the test results are inconsistent, the test case is removed. This process is repeated until all tests pass five times in a row. On average, less than 1% of test cases were removed from each suite.

4.3 | Data Collection

To answer our research questions, we capture the following data during and after generation:

- **Final Fitness Function Values:** For each test suite, we record the final fitness values for all four fitness functions considered in this experiment (Branch Coverage, Exception Count, Execution Time, and Output Coverage).
- **Branch Coverage Obligation Satisfaction:** Given a CUT, achieving Branch Coverage requires satisfying a set of test obligations, as defined in Section 2. We record information on the satisfaction of Branch Coverage obligations, including:
  - **Number of Test Obligations:** For each class-under-test, we record the number of Branch Coverage obligations.
  - **Percentage of Obligations Satisfied:** For each final test suite, we record the percentage of Branch Coverage obligations satisfied.
  - **Specific Obligations Satisfied:** For each final test suite, we record the specific obligations satisfied.
  - **Evolution of Branch Coverage During Generation:** To understand the dynamic evolution of coverage over the course of each invocation of EvoSuite, we tracked the percentage of obligations and specific obligations covered by the best test suite in the population once per second during the generation process.
- **Fault Detection:** To evaluate the fault-finding effectiveness of the generated test suites, we execute each test suite against the faulty version of each CUT. We then record the following:
  - **Likelihood of Fault Detection:** Across all trials for a particular fault, fitness function configuration, and search budget, we record the proportion of trials where the fault was detected to the total number of trials for that configuration.
  - **Number of Failing Tests:** For each test suite, we record the number of test cases that detect that fault (pass on the fixed version and fail on the faulty version).
- **Test Suite Size:** We recorded the number of tests in each test suite.
- **Average Test Case Length:** Each test consists of one or more method calls, variable initializations, and assertions. We record the average number of lines in each test case.

4.4 | Data Analysis

We answer each research question using the data gathered, comparing results attained by each fitness function configuration, split based on the search budget. To analyze the data, we employ a combination of descriptive statistics, distribution comparisons, and effect size tests when distributions are found to differ. Further explanation is provided in Section 5. Here, we provide a general overview of the data analysis procedure.

Descriptive Statistics

Descriptive statistics provide an initial overview of the collected data.

1. **Data Analysis**: Basic statistical measures such as the average, median, standard deviation, and percentiles are calculated for data appropriate for answering each research question. This provides an initial understanding of the data distribution and central tendencies [54].
2. **Data Visualization**: We utilize box plots as a graphical representation to offer a visual insight into the result distribution across different configurations [36].

Distribution Comparisons

We are interested in assessing whether the observed differences between two fitness function configurations at a particular search are significantly different.

For each research question, we select data relevant to that question (e.g., Branch Coverage attainment in RQ1). Then, for each pair of fitness function configurations, we formulate a hypothesis and null hypothesis in the following format:

- H: Generated test suites have different distributions of X depending on the targeted fitness function configuration.
- $H0$: Observations of X for both configurations are drawn from the same distribution.

The informal hypotheses raised in Section 1 correspond to the null hypotheses used to answer each research question. Our observations for each of the collected data items defined above are drawn from an unknown distribution. To evaluate the null hypothesis without any assumptions on distribution, we use the Wilcoxon rank-sum test [57], a non-parametric test. We apply the test with $\alpha = 0.05$. A p-value less than α indicates a statistically significant difference [34].

Effect Size

The Wilcoxon test determines if there are significant differences between the distributions of two configurations. To understand cases where there are differences, we use Cliff's delta to measure the effect size of these differences, providing a clearer understanding of their magnitude and practical significance [13]. We apply the standard interpretation of *Cliff's delta* (δ):

- $\delta > 0$ indicates that observations of configuration A are more likely to have higher values than observations of configuration B.
- $\delta < 0$ indicates that observations of configuration A are more likely to have lower values than observations of configuration B.
- The absolute value of δ is categorized as follows for further interpretation:
  - $|\delta| < 0.15$: Negligible effect
  - $0.15 \leq |\delta| < 0.33$: Small effect
  - $0.33 \leq |\delta| < 0.47$: Medium effect
  - $|\delta| \geq 0.47$: Large effect

5 | RESULTS

5.1 | Effect on Structural Coverage (RQ1)

In this section, we address the following hypothesis:

> **Hypothesis 1:** The inclusion of goal-based fitness functions as additional generation targets will not have an impact on the attainment of code coverage, as compared to targeting coverage alone.

Often, targeting multiple objectives can have *some* effect on each individual objective targeted, as compared to targeting a single objective on its own. If objectives are contradictory, or if too many objectives are targeted at once, then the final attainment of each may be lowered [49, 51]. However, targeting one objective may also offer feedback that enhances attainment of another [2, 19]. Therefore, we wish to assess—first—the impact that targeting additional goal-based objectives has on code coverage-based objectives. We examine three aspects of coverage: (1) the final percentage of coverage attained, (2) the specific coverage obligations covered by the final test suites, and (3), the evolution of coverage attainment over evolution. For this evaluation, we compare suite generated targeting Branch Coverage alone to suite targeting Branch Coverage and an additional goal-based fitness function.

Figure 4 Boxplots of the Branch Coverage attained by test suites, divided by budget.

Configuration	Budget	Mean	Std Dev	Min	25th %	Median	75th %	Max
Branch	180	0.618	0.326	0.000	0.383	0.690	0.906	1.000
Branch & Exception	180	0.614	0.333	0.000	0.367	0.690	0.912	1.000
Branch & Execution Time	180	0.621	0.324	0.000	0.389	0.704	0.900	1.000
Branch & Output	180	0.620	0.325	0.000	0.383	0.704	0.900	1.000
Branch	300	0.665	0.317	0.000	0.484	0.726	0.929	1.000
Branch & Exception	300	0.665	0.325	0.000	0.507	0.749	0.936	1.000
Branch & Execution Time	300	0.659	0.322	0.000	0.484	0.742	0.932	1.000
Branch & Output	300	0.653	0.324	0.000	0.475	0.740	0.934	1.000

Table 2 Descriptive statistics of Branch Coverage across different test generation configurations and search budgets.

Attained Branch Coverage

Table 2 offers descriptive statistics on the final attainment of Branch Coverage by generated test suites. Figures 4 and 5 also depict the attained Branch Coverage overall, and by project from Defects4J, respectively. Table 2 and Figure 4 do not demonstrate any clear differences between configuration, with regard to the final attained Branch Coverage. The distribution of results is visually similar for each configuration, and the mean and median Branch Coverage attained by each configuration is within a narrow range. An increase in search budget yields an increase in the average Branch Coverage, as well as less variance in the final results—seen in a rise in the 25th percentile. However, this improvement seems largely consistent across configurations.

In Figure 5, we do see some differences between configurations for particular projects. However, there are few clear trends and only a small number of bugs were drawn from many of these projects. Still, we note some observations from the projects with over 10 included bugs. First, for project Cli, we see a higher median Branch Coverage for the combination of Branch Coverage and Execution Time at both search budgets. For the project JSoup, we see that the combination of Branch and Output Coverage yields a slightly higher median coverage at both search budgets. We will investigate both of these observations—as well as potential differences for other projects—more closely in future work.

To confirm our initial inspection, we performed a Wilcoxon Rank-Sum test to assess pairwise comparisons between different test generation configurations for the two search budgets. The results of this test are shown in Table 3, where we see that no comparison demonstrated statistically significant differences—that is, no p-value was below 0.05.

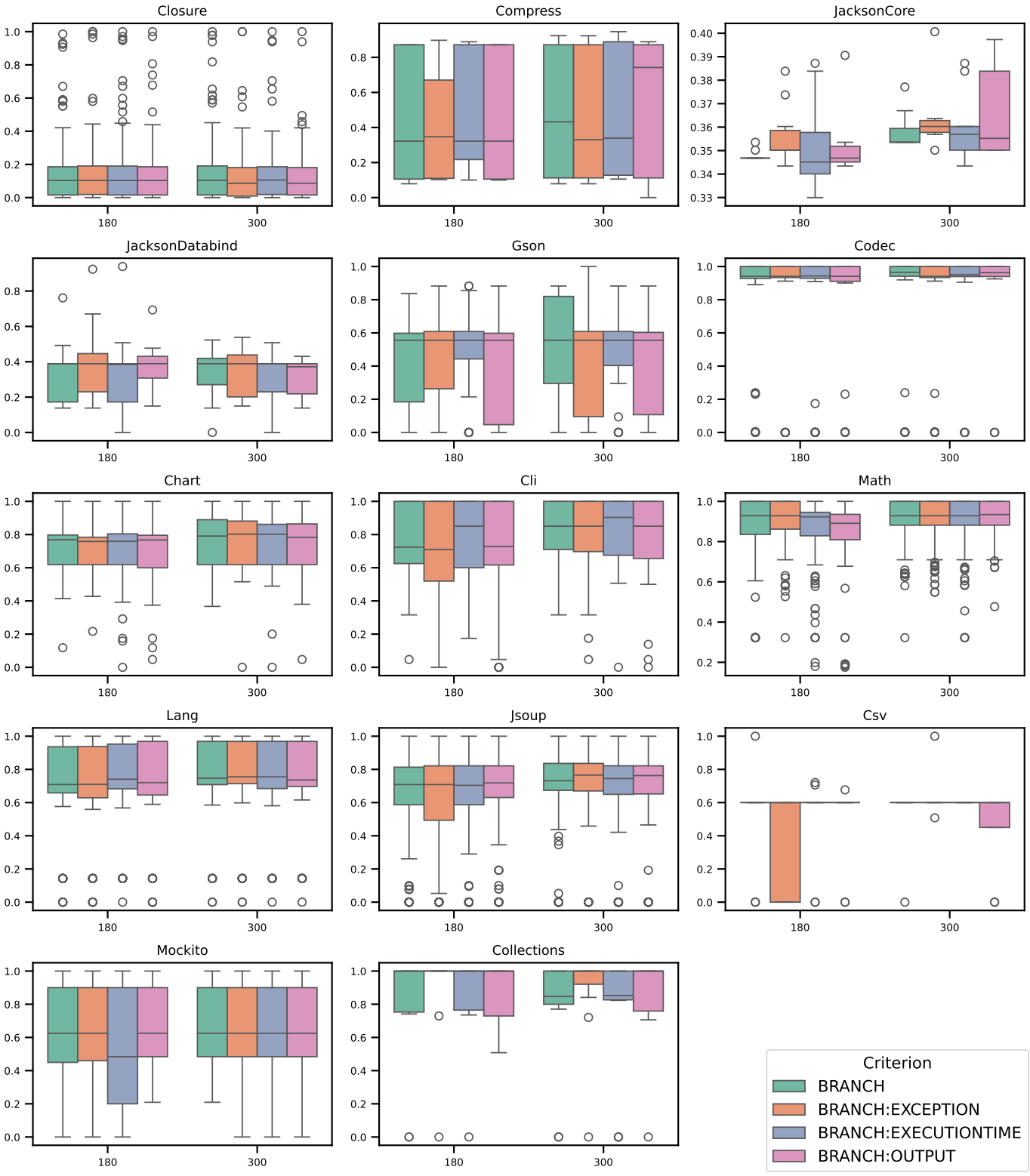

Figure 5 Boxplots of the Branch Coverage, divided by both budget and project from Defects4J. The X-axis reports the budget and the Y-axis the Branch coverage.

Comparison	Budget	P-value	Significant	Cliff's δ	Effect Size
Branch vs Branch & Execution Time	180	0.909	No	-0.005	Negligible
Branch vs Branch & Exception	180	0.362	No	-0.007	Negligible
Branch vs Branch & Output	180	0.647	No	0.005	Negligible
Branch vs Branch & Exception	300	0.985	No	-0.017	Negligible
Branch vs Branch & Execution Time	300	0.685	No	-0.000	Negligible
Branch vs Branch & Output	300	0.347	No	0.023	Negligible

Table 3 Calculated p-values from comparisons of attained Branch Coverage by different configurations, split by search budget.

Criterion	Budget	Mean	Std	Min	25%	50%	75%	Max
Branch & Exception	180	0.0033	0.0373	-0.0625	-0.0010	0.0000	0.0038	0.3194
Branch & Execution Time	180	-0.0015	0.0286	-0.1427	-0.0048	0.0000	0.0039	0.1226
Branch & Output	180	-0.0008	0.0527	-0.3875	-0.0043	0.0000	0.0058	0.2097
Branch & Exception	300	-0.0003	0.0232	-0.1111	-0.0051	0.0000	0.0048	0.0482
Branch & Execution Time	300	-0.0030	0.0311	-0.1936	-0.0063	0.0000	0.0073	0.1014
Branch & Output	300	-0.0011	0.0310	-0.1660	-0.0076	0.0000	0.0073	0.0871

Table 4 Descriptive statistics of the difference in the average coverage of each obligation between branch and another configuration, split by configuration and budget.

Comparison	Budget	Branch Wins (%)	Branch+X Wins (%)	Ties (%)
Branch vs Branch & Exception	180	3.7%	5.1%	91.2%
Branch vs Branch & Exec. Time	180	5.6%	6.4%	88.0%
Branch vs Branch & Output	180	6.0%	5.7%	88.3%
Branch vs Branch & Exception	300	4.6%	4.6%	90.8%
Branch vs Branch & Exec. Time	300	5.5%	4.9%	89.7%
Branch vs Branch & Output	300	5.2%	5.3%	89.5%

Table 5 Proportion of coverage obligations where one configuration outperforms another.

RQ1.1 (Attained Branch Coverage): Optimizing a second goal-based fitness function does not have a significant impact on the final Branch Coverage attained by test suites.

Attained Coverage Obligations

During and after the test generation process, we collected information on which specific Branch Coverage obligations were covered by generated test suites. To assess whether different configurations tend to cover distinct test obligations, we calculated the average coverage of each obligation for each class-under-test across all trials conducted for each configuration, search budget, and bug. For example, if four of the ten trials targeting Branch Coverage and Output Coverage for bug Chart-3 covered the first Branch Coverage obligation for the targeted class, then the average coverage of that obligation would be 0.40.

The resulting averages were then used to compare targeting Branch Coverage alone to targeting Branch Coverage and a second goal-based fitness function. For example, if there were four coverage obligations for a class:

- When targeting Branch Coverage alone, the average coverage of each obligation was 0.4, 0.7, 0.8, and 0.4.
- When targeting Branch and Exception Count, the average coverage of each obligation was 0.5, 0.7, 0.8, and 0.3.
- The resulting difference between the two would be -0.1, 0.0, 0.0, and 0.1.

Figure 7 visualizes the result of this comparison for one class from one bug and search budget, as an example of the differences that can emerge. In this example, positive spikes show cases where targeting Branch alone performed better for a particular obligation versus the compared configuration and vice versa.

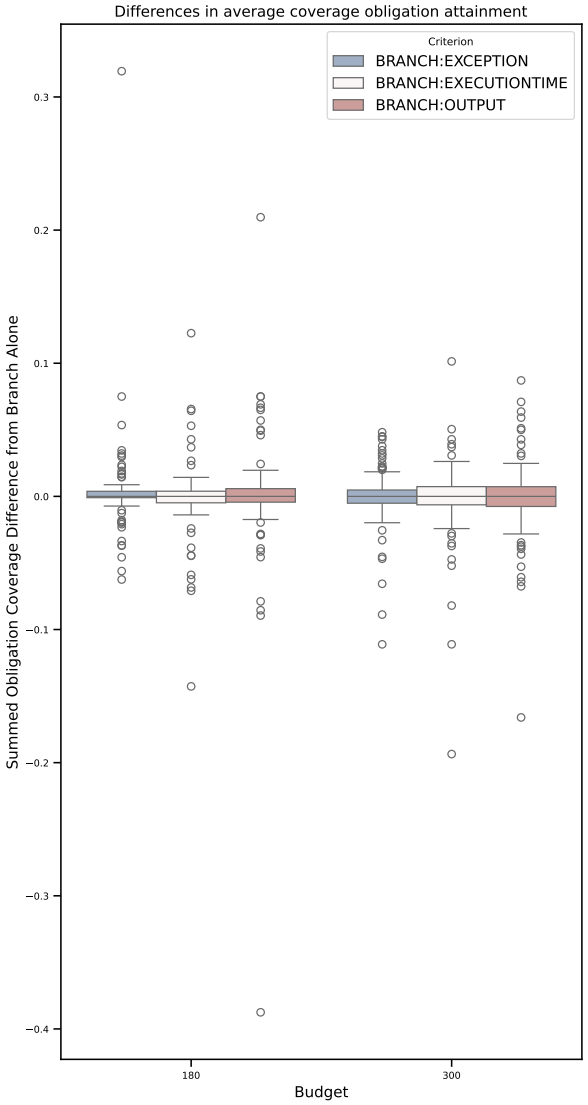

Figure 6 Boxplots of the differences in covered obligations between targeting Branch Coverage alone versus Branch and goal-based fitness function.

To generalize the assessment across all bugs, we calculated the sum of these differences for each class for each bug. Figure 6 plots the difference between each configuration across all bugs, split by search budget. Table 4 includes descriptive statistics on the difference in the average coverage of each obligation.

Figure 6 shows the vast majority of the summed differences are close to zero, with a median of 0.00 for all comparisons. This means that—in most cases—there are few major differences in the obligations covered by each configuration. There are differences in the 25th and 75th percentiles between configurations, but relatively narrow ones.

Table 5 offers a complementary analysis, where we indicate the proportion of individual coverage obligations where one configuration outperformed the other in terms of the average coverage of each obligation (a “win”) or where their performance

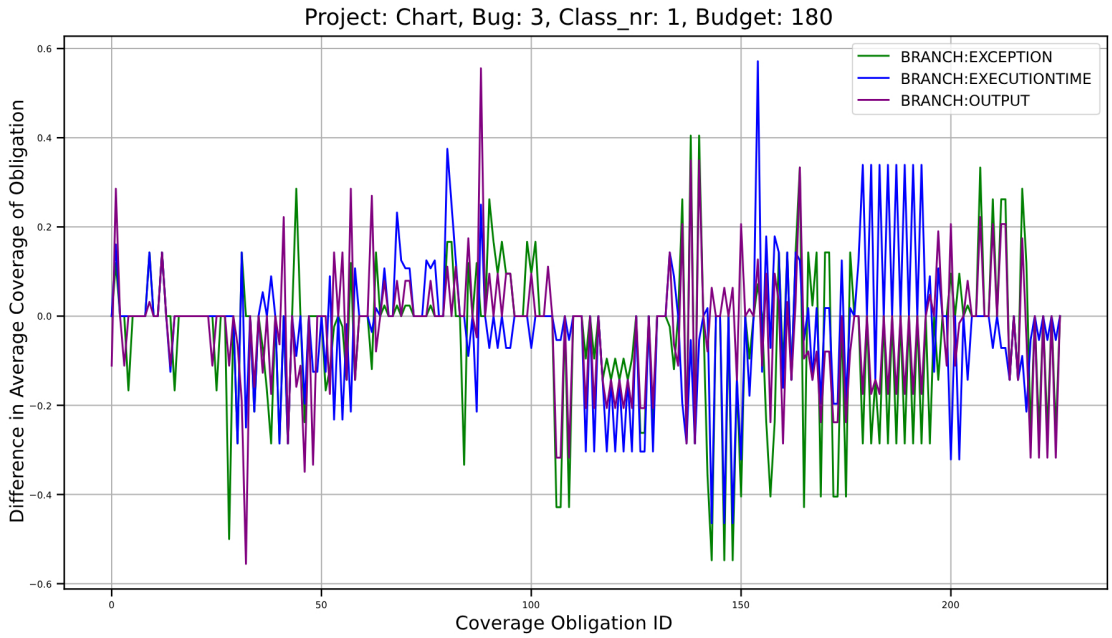

Figure 7 Difference in the average coverage of each obligation by Branch alone against Branch & Exception (green), Branch & Execution Time (Blue), and Branch & Output (purple) for Chart-3, Class 1, when the budget is set to 180 seconds.

was identical (a "tie"). The results show that for the vast majority of obligations (88%–91%), the single- and multi-objective strategies perform identically, resulting in a tie. For the small fraction of branches where performance differs, there is no consistent winner—both Branch alone and multi-objective configurations win a similarly small percentage of the time. This reinforces that adding a secondary objective rarely changes which specific branches get covered.

RQ1.2 (Obligations Covered): In the majority of cases, the addition of a goal-based fitness function does not change the likelihood of covering particular test obligations. Almost all differences in the average coverage of individual obligations are within 10% of when Branch Coverage is targeted alone.

Figure 6 shows that there are a number of outliers. Most outliers are clustered within -0.1 to 0.1, i.e., within 10% difference in average coverage. However, it is possible that some of these outliers offer information that could improve the results of test generation. In particular, negative outliers are interesting, as they suggest cases where the addition of a goal-based fitness function improved Branch Coverage.

We inspected the negative outliers, with a particular focus on the six most extreme cases—Math-81 at the 180 second budget for both Branch and Execution Time and Branch and Output Coverage, JSoup-9 for Branch and Exception Count and Branch and Execution Time at the 300 second budget, Math-11 for Branch and Execution Time at the 300 second budget, and Closure-164 for Branch and Output Coverage at the 300 second budget. Overall, there were few clear and actionable conclusions that we could draw from these outliers. However, we share some interesting observations.

The class modified in Math-81 for Branch and Output Coverage at the 180 second budget was the most extreme outlier, with an average difference of 38.75% in coverage. The explanation for this difference is relatively straightforward. Many of the methods of the class-under-test have numeric return values. It is likely that Output Coverage, by placing emphasis on returning diverse results for these methods, helped to steer test generation towards covering Branch Coverage obligations in these methods and in methods indirectly called through these methods. This improvement disappears at the 300 second search budget, suggesting that Branch Coverage alone is eventually effective. However, the addition of Output Coverage speeds coverage attainment.

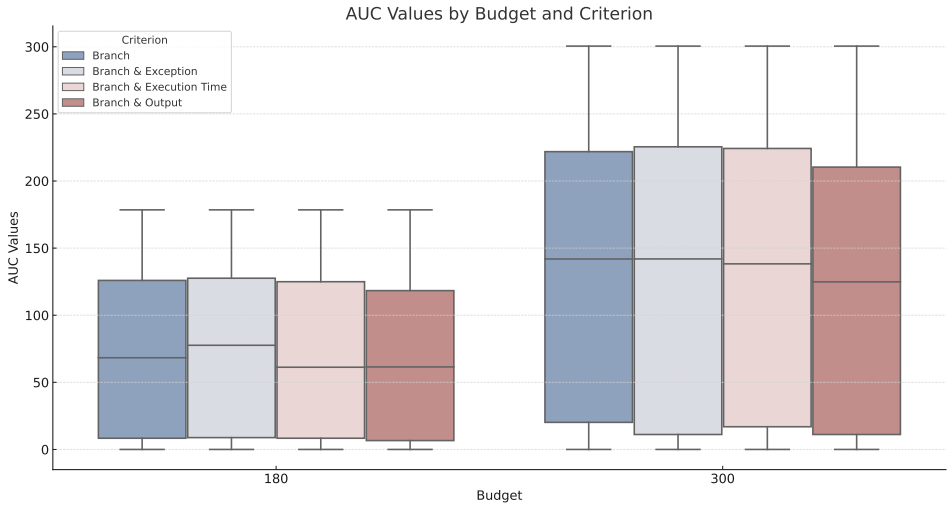

Figure 8 Boxplot for Area Under the Curve (AUC) of the branch coverage evolution, split by budget and Criterion.

Criterion	Budget	Average	Median
Branch	180	74.06	68.32
Branch & Exception	180	75.44	77.59
Branch & Execution Time	180	70.73	61.25
Branch & Output	180	70.30	61.48
Branch	300	134.19	141.89
Branch & Exception	300	135.65	141.93
Branch & Execution Time	300	130.55	138.27
Branch & Output	300	127.60	124.85

Table 6 Average and median values for AUC of coverage evolution for different configurations, split by budget.

Comparison	Budget	P-value	Significant	Cliff's δ	Effect Size
Branch vs Branch & Execution Time	180	0.909	No	-0.005	Negligible
Branch vs Branch & Exception	180	0.362	No	-0.007	Negligible
Branch vs Branch & Output	180	0.647	No	0.005	Negligible
Branch vs Branch & Exception	300	0.985	No	-0.017	Negligible
Branch vs Branch & Execution Time	300	0.685	No	-0.001	Negligible
Branch vs Branch & Output	300	0.347	No	0.023	Negligible

Table 7 P-values and effect size on pairwise comparisons of AUC by different configurations, split by search budget.

A similar observation can be made for Branch and Output Coverage for the class-under-test in Closure-164, where there was an average coverage difference of 16.60%. Most methods in this class return Boolean values. It is possible the Output Coverage helped to encourage Branch Coverage by ensuring that the methods returned both possible values. Here, this difference increased with the search budget.

A potential explanation for the outliers for Branch and Execution Time is that the Execution Time fitness function encourages the generation of longer test cases, with more program interactions. This function penalizes test cases that are too long, but the average test case length is still higher than when Branch Coverage is targeted alone. This may encourage improvement in coverage as well, in a small number of cases.

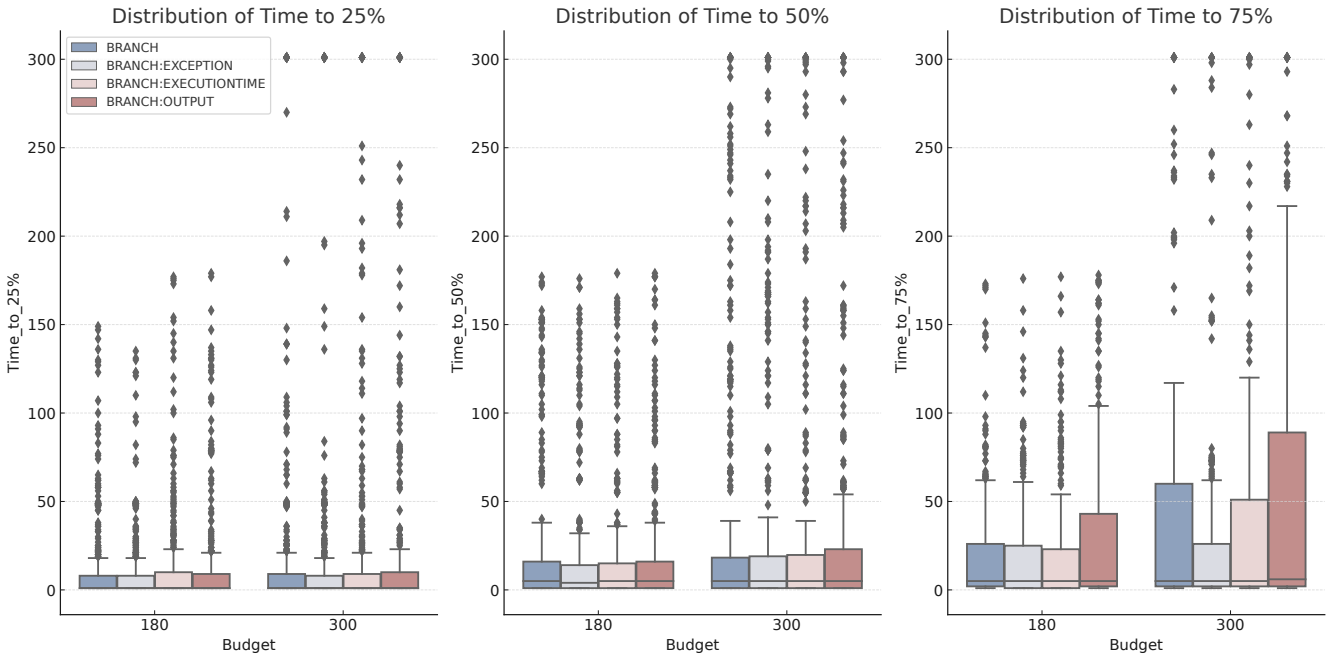

Figure 9 Boxplots of the time taken to achieve 25%, 50%, and 75% Branch Coverage for each configuration, split by budget.

Evolution of Coverage Attainment

We collected the evolution of coverage during the test generation process, based on the Branch Coverage achieved by the best test suite in the evolving population. A snapshot of coverage is captured each second during the generation process. This allows us to calculate the AUC (Area Under the Curve) and the time the search took to achieve 25%, 50%, and 75% coverage during each trial for each configuration.

Figure 8 and Table 6 shows the statistics for AUC for each configuration and search budget. Higher AUC values indicate that Branch Coverage evolved early while lower means the search took more time to achieve coverage.

We observe that the median AUC is lower for both search budgets for Branch and Execution Time as well as for Branch and Output Coverage than for Branch alone, potentially indicating slightly slower coverage attainment. However, the 25th and 75th percentiles are similar. We also observe that the median AUC is slightly higher for Branch and Exception Coverage than for Branch alone at the 180 second budget, potentially indicating a slight improvement in the rate of coverage attainment.

In Table 7, we show the results of statistical testing on the AUC. All p-values are considerably above the 0.05 threshold. Figure 9 and Table 8 report the time needed to reach 25, 50, and 75% Branch Coverage. The median time to reach each landmark is very similar across all configurations, regardless of search budget. The largest differences between configurations can be seen in the 75th percentile for each configuration. Here, we often see a higher 75th percentile for Branch and Output, as compared to the higher configurations, indicating again that coverage attainment may be slightly slowed with the inclusion of Output Coverage as a goal. However, there is no evidence that this effect is significant.

While the median remains relatively consistent across search budgets, the 75% percentile is often higher at the 300 second budget, especially at the 75% coverage threshold. The average also raises across search budgets. This is due to a small number of additional test suites reaching these thresholds later in the generation process under the higher budget. Again, the median is relatively consistent across all budgets and configurations.

RQ1.3 (Coverage Evolution): There are almost no significant differences between configurations with regard to the rate of attainment of Branch Coverage. Branch and Exception Coverage at a 180 second budget shows statistically significant improvement, but with only a negligible effect size.

Criterion	Budget	Count	Average	Median
Time to 25%				
Branch	180	544	10.33	1.00
Branch & Exception	180	527	9.12	1.00
Branch & Execution Time	180	501	12.06	1.00
Branch & Output	180	512	12.45	1.00
Branch	300	551	19.00	1.00
Branch & Exception	300	513	17.37	1.00
Branch & Execution Time	300	479	19.76	1.00
Branch & Output	300	492	21.34	1.00
Time to 50%				
Branch	180	433	22.55	5.00
Branch & Exception	180	416	18.70	4.00
Branch & Execution Time	180	376	19.67	5.00
Branch & Output	180	391	20.74	5.00
Branch	300	448	35.64	5.00
Branch & Exception	300	417	34.98	5.00
Branch & Execution Time	300	378	33.36	5.00
Branch & Output	300	391	37.39	5.00
Time to 75%				
Branch	180	233	24.29	5.00
Branch & Exception	180	236	22.08	5.00
Branch & Execution Time	180	209	23.15	5.00
Branch & Output	180	217	33.29	5.00
Branch	300	244	41.32	5.00
Branch & Exception	300	231	35.94	5.00
Branch & Execution Time	300	222	43.28	5.00
Branch & Output	300	229	57.98	6.00

Table 8 Descriptive statistics on the time (in seconds) to reach coverage thresholds, split by configuration and budget. "Count" indicates the number of trials that reached this threshold.

Criterion	Budget	Execution Time		Output Coverage		Exception Coverage	
		Avg	Median	Avg	Median	Avg	Median
Branch	180	41.73	31.00	0.31	0.35	0.29	0.22
Branch & Exception	180	42.69	31.00	0.32	0.37	0.62	0.79
Branch & Execution Time	180	42.90	33.00	0.31	0.36	0.31	0.25
Branch & Output	180	42.89	31.00	0.43	0.49	0.30	0.25
Exception	180	27.48	21.00	0.19	0.11	0.59	0.71
Output	180	23.13	16.00	0.45	0.50	0.13	0.08
Branch	300	39.83	30.00	0.33	0.36	0.30	0.25
Branch & Exception	300	39.11	28.00	0.35	0.39	0.68	0.88
Branch & Execution Time	300	40.37	30.00	0.33	0.36	0.32	0.25
Branch & Output	300	40.38	30.00	0.46	0.50	0.32	0.25
Exception	300	26.87	20.00	0.19	0.11	0.59	0.71
Output	300	23.57	16.00	0.47	0.52	0.14	0.09

Table 9 Averages and median values for Execution Time, Output Coverage, and Exception Count, by configuration and budget.

5.2 | Impact on Goal-Based Objectives (RQ2)

Our second hypothesis was the following:

> **Hypothesis 2:** Targeting both coverage and a goal-based fitness function will not have an impact on the attainment of goal-based fitness functions, as compared to targeting coverage or a goal-based fitness function alone.

Similar to RQ1, targeting multiple objectives could affect the final fitness values of the goal-based fitness functions—e.g., raising or lowering goal attainment when compared to targeting a goal-based or a structure-based fitness function alone.

In addition to examining this hypothesis directly, there is a secondary hypothesis of interest. One of the reasons for the prevalence of structural coverage in search-based test generation is that structural coverage can be translated effectively into

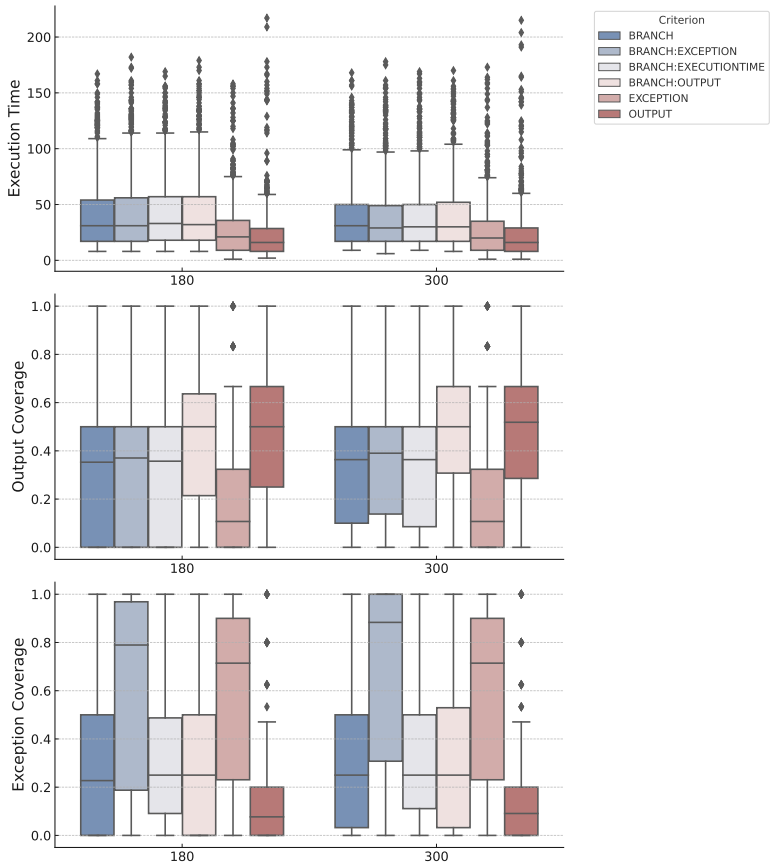

Figure 10 Boxplots for Execution time, Exception Count, and Output Coverage divided by budget and configuration.

distance-based fitness functions, e.g., the branch distance used for optimizing Branch Coverage [7]. This means that tests can be efficiently generated that widely explore the codebase. Goal-based fitness functions often lack distance-based fitness functions [2]. Consequently, they may offer less feedback to the optimization process. As a result, targeting both coverage and goal-based objectives could potentially result in higher attainment of goal-based fitness by offering an additional feedback mechanism [2, 51]. Past research has not assessed this hypothesis. In this experiment, the Exception Count is one such example. In contrast, Output Coverage does have a distance-based fitness function, so such benefits may not be observed in this case.

During the experiment, we recorded the final attainment of each goal-based fitness function for all generated test suites. Note that the Execution Time fitness function cannot be executed without also targeting Branch Coverage, so we were unable to generate test suites targeting Execution Time alone. In addition, note that the Exception Count is normalized between 0–1 for all bugs, based on the largest number of exceptions seen in any trial for that bug, as the number of possible exceptions differs between bugs. Figure 10 shows boxplots for Exception Count, Output Coverage, and Execution Time for each fitness function configuration and search budget. Average and median values are reported in Table 9. Tables 10–12 report p-values and effect sizes for comparisons between single-objective generation versus multi-objective optimization.

First, we observe that no goal-based fitness function can serve as a proxy for another goal-based fitness function. Targeting Output Coverage yields a low Exception Count and Execution Time. Similarly, targeting Exception Coverage yields low Output Coverage and Execution Time. If one targets a goal-based fitness function *alone*, they should not expect high attainment of goals other than the one that function was designed for.

Targeting Branch Coverage *alone* yields better performance at each goal than targeting a fitness function designed for a different goal, suggesting that coverage of the code base will always lead to *some* degree of goal attainment. However, as shown in Tables 10–11, these suites are also significantly worse at attaining Output Coverage or Exception Count than targeting either goal directly or targeting multiple objectives—with large effect size for Exception Count at both search budgets, small–medium effect size for Output Coverage at the 180-second search budget, and medium–large effect size for Output Coverage at the

Comparison	Budget	P-value	Significant	Cliff's δ	Category
Branch vs Branch & Exception	180	5.28×10^{-72}	Yes	-0.48	Large
Branch vs Exception	180	2.68×10^{-84}	Yes	-0.56	Large
Branch & Exception vs Exception	180	0.176	No	0.01	Negligible
Branch vs Branch & Exception	300	3.06×10^{-80}	Yes	-0.51	Large
Branch vs Exception	300	1.49×10^{-79}	Yes	-0.56	Large
Branch & Exception vs Exception	300	0.152	No	0.10	Negligible

Table 10 Significance tests and effect size for comparisons of **Exception Count** between single and multi-objective configurations across different budgets.

Comparison	Budget	P-value	Significant	Cliff's δ	Category
Branch vs Branch & Output	180	9.72×10^{-29}	Yes	-0.32	Small
Branch vs Output	180	7.3×10^{-47}	Yes	-0.47	Medium
Branch & Output vs Output	180	2.53×10^{-9}	Yes	-0.16	Small
Branch vs Branch & Output	300	1.82×10^{-29}	Yes	-0.36	Medium
Branch vs Output	300	1.53×10^{-44}	Yes	-0.49	Large
Branch & Output vs Output	300	2.1×10^{-4}	Yes	-0.11	Negligible

Table 11 Significance tests and effect size for comparisons of **Output Coverage** between single and multi-objective configurations across different budgets.

Comparison	Budget	P-value	Significant	Effect Size	Category
Branch vs Branch & Execution Time	180	0.350	No	-0.01	Negligible
Branch vs Branch & Execution Time	300	0.882	No	-0.01	Negligible

Table 12 Significance tests and effect size for comparisons of **Execution Time** between single and multi-objective configurations across different budgets.

300-second budget. In other words, code coverage is a weak proxy for a goal-based fitness function—as noted in Section 1, coverage alone is not enough to ensure goal attainment.

RQ2 (Goal Coverage): Targeting code coverage *alone* leads to worse goal attainment than directly targeting a goal-based objective.

Next, we compare targeting a goal alone versus multi-objective optimization. Table 11 shows that, at both budgets, there is a significant difference in Output Coverage between targeting Output Coverage alone and targeting both Branch and Output Coverage, in favor of targeting Output Coverage alone. However, the effect size is only small at the 180-second budget and negligible at the 300 second budget. Similarly, Table 10 shows that, at both budgets, there is no significant difference between targeting Branch and Exception Count and targeting Exception Count alone.

In our past research, we observed situations where both Branch Coverage and the Exception Count offer the other function missing feedback—with the Exception Count steering Branch Coverage towards input that triggers exceptions and Branch Coverage offering feedback on how to further explore the code base [51]. Such cases are rare, but may explain the higher median seen for multi-objective generation in Figure 10.

We were unable to generate suites targeting Execution Time alone due to limitations in the implementation. However, from Table 9, we can see that there is a slight improvement in the average (at both budgets) and the median (at the 180 second budget) Execution Time from targeting Branch and Execution Time simultaneously. However, there are no statistically significant differences between targeting both objectives versus targeting Branch Coverage alone (Table 12). No configuration was significantly better at yielding tests with high execution times. It is possible that the examples chosen from Defects4J had few or no performance issues that could be exposed through unit testing.

Criterion	Budget	Min	25%	50%	75%	Max	Avg	# of bugs detected
Branch	180	0.00	0.00	0.00	0.38	1.00	0.24	28
Branch & Exception	180	0.00	0.00	0.00	0.54	1.00	0.27	30
Branch & Execution Time	180	0.00	0.00	0.00	0.41	1.00	0.25	34
Branch & Output	180	0.00	0.00	0.00	0.89	1.00	0.28	35
Exception	180	0.00	0.00	0.00	0.10	0.22	0.04	23
Output	180	0.00	0.00	0.00	0.10	0.44	0.09	27
Branch	300	0.00	0.00	0.00	0.43	1.00	0.26	32
Branch & Exception	300	0.00	0.00	0.00	0.75	1.00	0.30	35
Branch & Execution Time	300	0.00	0.00	0.00	0.57	1.00	0.28	39
Branch & Output	300	0.00	0.00	0.00	0.85	1.00	0.31	33
Exception	300	0.00	0.00	0.00	0.10	0.20	0.06	28
Output	300	0.00	0.00	0.00	0.10	0.33	0.13	31

Table 13 Descriptive statistics on the **likelihood of fault detection**, split by budget and configuration.

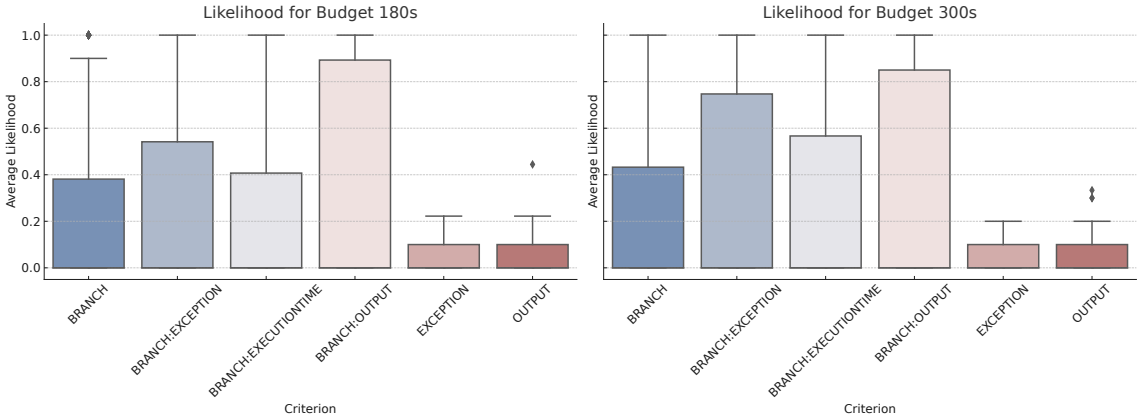

Figure 11 Boxplots of the **likelihood of fault detection**, divided by budget and configuration.

RQ2 (Goal Coverage): Targeting code coverage and a goal-based objective simultaneously results in no, or only a limited (negligible–small), drop in goal-based fitness compared to targeting a goal-based objective alone.

5.3 | Impact on Fault Detection (RQ3)

The third hypothesis that we raised was the following:

Hypothesis 3: Targeting both coverage and a goal-based fitness function will not have an impact on the fault detection of generated test suites, as compared to targeting coverage or a goal-based fitness function alone.

Regardless of the impact on the code coverage or attainment of non-coverage testing goals, targeting multiple objectives could change the specific inputs applied to the class-under-test. As a result, there could be a change to the fault-revealing power of those test suites—either increased due to a change in the versatility of the test suite [19, 49, 51] or, even, a potential decrease.

To assess this hypothesis, we consider two aspects of fault detection. First, the *likelihood of fault detection*—for each fault, the proportion of suites (for a particular configuration) that detect the fault to those generated for that configuration. Second, we consider the *number of failing tests*—how many test cases detect the fault when it is detected. We consider both so that we can examine both how likely a fault is to be detected and how much information exists to understand and debug the fault. If two configurations have the same likelihood of detection, one may offer more failing tests to use in the debugging process.

Comparison	Budget	P-value	Significant	Cliff's δ	Category
Branch vs Branch & Exception	180	0.657	No	-0.035	Negligible
Branch vs Branch & Execution Time	180	0.895	No	-0.014	Negligible
Branch vs Branch & Output	180	0.457	No	-0.052	Negligible
Branch vs Exception	180	6.39×10^{-5}	Yes	0.668	Large
Branch vs Output	180	1.72×10^{-4}	Yes	0.625	Large
Branch & Exception vs Exception	180	8.32×10^{-5}	Yes	0.792	Large
Branch & Output vs Output	180	5.60×10^{-5}	Yes	0.810	Large
Branch vs Branch & Exception	300	0.384	No	-0.061	Negligible
Branch vs Branch & Execution Time	300	0.612	No	-0.046	Negligible
Branch vs Branch & Output	300	0.461	No	-0.058	Negligible
Branch vs Exception	300	1.41×10^{-5}	Yes	0.729	Large
Branch vs Output	300	6.07×10^{-5}	Yes	0.669	Large
Branch & Exception vs Exception	300	6.28×10^{-6}	Yes	0.937	Large
Branch & Output vs Output	300	2.92×10^{-5}	Yes	0.838	Large

Table 14 Significance tests and effect size for comparisons of **likelihood of fault detection** between single and multi-objective configurations across different budgets.

Figure 11 illustrates the likelihood of fault detection, and Table 13 offers descriptive statistics for each configuration and search budget. Table 13 lists, for each budget and fitness function configuration, the lowest, highest, average, median, 25^{th} quartile, and 75^{th} quartile likelihood of fault detection observed for the assessed faults, as well as the total number of faults detected over the set of 93 considered in this experiment. Immediately, we can see that the multi-objective configurations detect *more* faults—with Branch and Output detecting the most at the 180 second budget and Branch and Execution Time detecting the most at the 300 second budget. The multi-objective configurations are followed by Branch Coverage, then Output Coverage, then the Exception Count.

RQ3 (Fault Detection): Suites targeting multi-objective configurations detected more faults than single-objective configurations. Further, suites targeting Branch Coverage alone detected more faults than goal-based suites alone.

However, as the majority of faults are never detected, the median likelihood of fault detection is also zero for all configurations. The average, skewed by cases where faults are detected, is somewhat more informative. Targeting Branch and Output Coverage yields the highest average likelihood of fault detection at both search budgets (28 and 31%), followed at both budgets by Branch and Exception Count (27 and 30%) and Branch and Execution Time (25 and 28%). This same ordering can be seen in the 75th percentile in Figure 11. Again, targeting the goal-based functions alone yields the lowest average likelihood of fault detection—with the worst performance from targeting the Exception Count alone.

We note that these results replicate the general trends observed in our previous work [51, 19]. While the exact results differ due to changes made to EvoSuite and the stochastic nature of search-based test generation, we previously observed that targeting Branch Coverage alone yielded higher likelihood of fault detection than targeting Exception Count or Output Coverage alone, and that multi-objective combinations had an even higher average likelihood of fault detection.

Table 14 includes significance tests and effect sizes for the likelihood of fault detection. At both budgets, the multi-objective configurations do *not* yield significantly different results from targeting Branch Coverage alone in the likelihood of fault detection. However, targeting Branch Coverage yields better results (with large effect size) than targeting a goal-based fitness function at both budgets. Targeting Branch and Exception Count simultaneously also yields better results than targeting Exception Count alone, with large effect size, at both budgets. Finally, targeting Branch Coverage and Output Coverage yields better results than targeting Output Coverage alone, with large effect sizes at both budgets.

RQ3 (Fault Detection): Suites targeting Branch Coverage alone or a multi-objective configuration outperform suites targeting a goal-based objective in the likelihood of fault detection with medium–large effect size.

Criterion	Budget	Avg	Min	25%	50%	75%	Max
Branch	180	0.89	0.00	0.00	0.00	0.00	29.00
Branch & Exception	180	1.13	0.00	0.00	0.00	1.00	30.00
Branch & Execution Time	180	1.00	0.00	0.00	0.00	0.00	33.00
Branch & Output	180	0.88	0.00	0.00	0.00	1.00	37.00
Exception	180	0.04	0.00	0.00	0.00	0.00	1.00
Output	180	0.09	0.00	0.00	0.00	0.00	1.00
Branch	300	1.08	0.00	0.00	0.00	1.00	41.00
Branch & Exception	300	1.28	0.00	0.00	0.00	1.00	39.00
Branch & Execution Time	300	1.04	0.00	0.00	0.00	1.00	39.00
Branch & Output	300	0.98	0.00	0.00	0.00	1.00	38.00
Exception	300	0.06	0.00	0.00	0.00	0.00	1.00
Output	300	0.13	0.00	0.00	0.00	0.00	1.00

Table 15 Descriptive statistics on **number of failing tests**, split by budget and configuration.

Comparison	Budget	P-value	Significant	Cliff's δ	Category
Branch vs Branch & Exception	180	0.043	No	-0.043	Negligible
Branch vs Branch & Execution Time	180	0.518	No	-0.019	Negligible
Branch vs Branch & Output	180	0.053	No	-0.049	Negligible
Branch vs Exception	180	2.96×10^{-37}	Yes	0.35	Medium
Branch vs Output	180	7.43×10^{-31}	Yes	0.35	Medium
Branch & Exception vs Exception	180	1.90×10^{-36}	Yes	0.41	Medium
Branch & Output vs Output	180	4.16×10^{-35}	Yes	0.36	Medium
Branch vs Branch & Exception	300	0.014	No	-0.069	Negligible
Branch vs Branch & Execution Time	300	0.124	No	-0.050	Negligible
Branch vs Branch & Output	300	0.012	No	-0.059	Negligible
Branch vs Exception	300	2.90×10^{-45}	Yes	0.33	Medium
Branch vs Output	300	2.62×10^{-37}	Yes	0.33	Medium
Branch & Exception vs Exception	300	2.29×10^{-38}	Yes	0.40	Medium
Branch & Output vs Output	300	3.34×10^{-38}	Yes	0.41	Medium

Table 16 Significance tests and effect size for comparisons of **number of failing tests** between single and multi-objective configurations across different budgets.

RQ3 (Fault Detection): Suites targeting a multi-objective configuration fail to outperform suites targeting Branch Coverage alone with significance in the likelihood of fault detection. However, targeting a multi-objective configuration does increase the average and 75th percentile performance.

Figure 12 illustrates the number of failing tests, and Table 15 offers descriptive statistics for each configuration and search budget. Table 16 includes significance tests and effect sizes, when significance is found.

Here we see largely similar trends to the likelihood of fault detection, with the median number of failing tests being 0 for all configurations. Targeting Branch and Exception Count yields the largest average number of failing tests at both budgets (1.13 and 1.28), followed by Branch and Execution Time (1.00) at the 180 second budget and Branch alone (1.08) at the 300 second budget. However, these results are in a relatively narrow range, and no multi-objective configuration is an outlier in terms of the number of tests that fail when a fault is detected. As shown in Table 16, targeting Branch Coverage alone or a multi-objective configuration yields a larger number of failing tests than targeting a goal-based objective alone, with medium effect size.

RQ3 (Fault Detection): Suites targeting Branch Coverage alone or a multi-objective configuration outperform suites targeting a goal-based objective, in the number of failing tests, with medium effect size.

RQ3 (Fault Detection): Suites targeting a multi-objective configuration fail to outperform suites targeting Branch Coverage alone with significance in the number of failing tests.

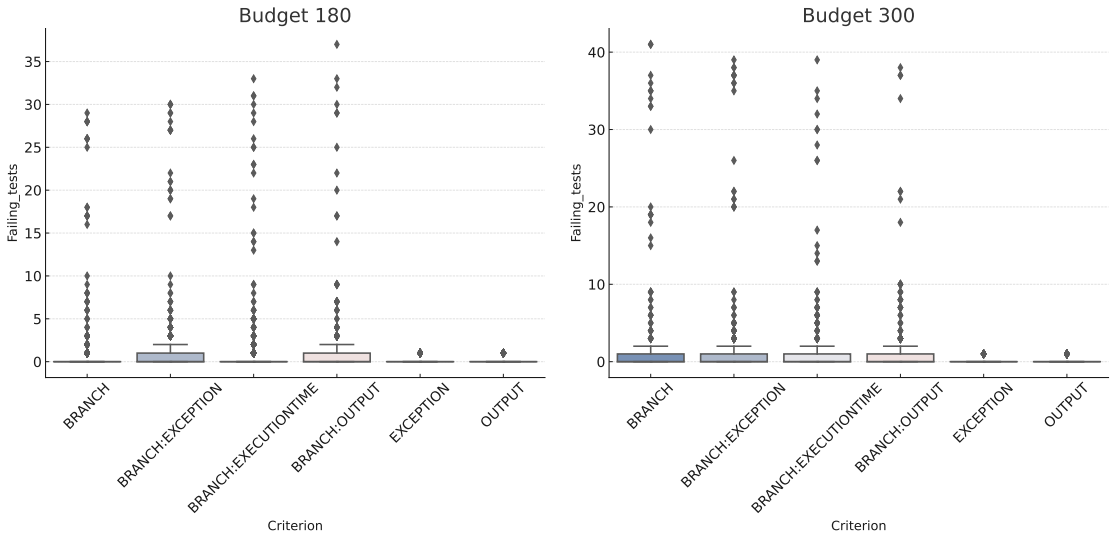

Figure 12 Boxplots of the **number of failing tests**, divided by budget and configuration.

Criterion	Budget	Suite Size		Test Case Length	
		Avg	Median	Avg	Median
Branch	180	18.36	14.00	34.51	21.43
Branch & Exception	180	23.45	19.00	33.48	21.38
Branch & Execution Time	180	19.76	15.00	35.12	21.82
Branch & Output	180	22.89	18.00	35.90	21.67
Exception	180	9.92	6.00	20.35	20.00
Output	180	10.23	5.00	21.76	19.75
Branch	300	20.23	14.00	32.35	21.29
Branch & Exception	300	25.64	19.00	31.06	21.19
Branch & Execution Time	300	22.41	16.00	34.88	21.73
Branch & Output	300	25.96	18.00	34.76	21.54
Exception	300	9.98	6.00	19.85	20.00
Output	300	10.65	6.00	20.45	19.75

Table 17 Average and median test suite size and average test case length, divided by budget and configuration.

5.4 | Impact on Test Suite Contents (RQ4)

Our fourth hypotheses was that:

> **Hypothesis 4:** Targeting both coverage and a goal-based fitness function will not have an impact on the size of the test suite and the average test length, as compared to targeting coverage or a goal-based fitness function alone.

Targeting multiple objectives could increase the suite size or average test case length. Each unit test case contains one or more interactions with the class-under-test. Each targeted objective imposes a set of obligations that must be covered in those interactions, and only particular input will ensure those obligations are met. Multi-objective optimization imposes a larger set of obligations than single-objective optimization.

Each test case can cover obligations of multiple criteria, meaning that one should not expect a linear increase in suite size or test case length during multi-objective optimization compared to single-objective optimization. However, some obligations require highly specific test input or setup, potentially necessitating additional specialized test cases or an increased number of program interactions. Therefore, some increase in suite size, test length, or both could occur.

Figure 13 shows boxplots for the test suite size and average test case length, with Table 17 reporting median and average values for both. Tables 18 and 19 report the results of significance tests and effect sizes (when significance is found) for both measurements, comparing single and multi-objective configurations.

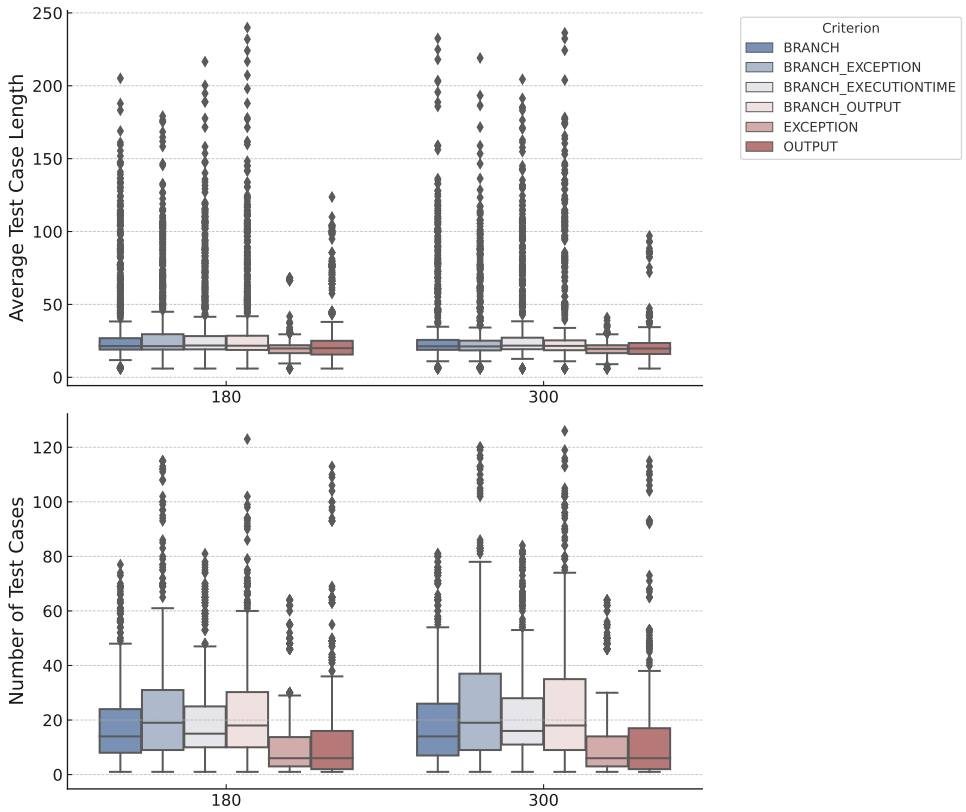

Figure 13 Boxplots for the test suite size and the average test case length, divided by budget and configuration.

Comparison	Budget	P-value	Cliff's δ	Effect Size
Branch vs Branch & Exception	180	1.41×10^{-7}	-0.29	Small
Branch vs Branch & Output	180	1.95×10^{-6}	-0.26	Small
Branch vs Branch & Execution Time	180	0.011	-0.12	Negligible
Branch & Exception vs Exception	180	4.97×10^{-69}	0.84	Large
Branch & Output vs Output	180	2.38×10^{-54}	0.68	Medium
Branch vs Branch & Exception	300	4.35×10^{-7}	-0.27	Small
Branch vs Branch & Output	300	1.80×10^{-6}	-0.27	Small
Branch vs Branch & Execution Time	300	0.001	-0.13	Negligible
Branch & Exception vs Exception	300	9.21×10^{-67}	0.92	Large
Branch & Output vs Output	300	1.96×10^{-51}	0.73	Medium

Table 18 Significance comparisons and effect sizes for **test suite size** between single and multi-objective configurations across different budgets. All p-values are significant.

From Figure 13 and Table 18, we can immediately see that the distributions of test suite sizes vary significantly between configurations. Targeting code coverage and a goal-based fitness function simultaneously results in larger test suites than targeting either alone, with medium–large effect size compared to targeting a goal-based objective and negligible small effect size compared to targeting Branch Coverage alone.

Figure 13 and Table 19 also show that the average test case length tends to increase with multi-objective optimization compared to targeting a goal-based objective alone, with small–medium effect size. However, the test length does not increase compared to targeting Branch Coverage alone—with only a negligible increase when targeting Branch and Execution Time.

Branch Coverage tends to have more obligations to cover than the Exception Count or Output Coverage, as a program will generally have more branches in control flow than output partitions or thrown exceptions. Covering the obligations of Branch

Comparison	Budget	P-value	Significant	Cliff's δ	Effect Size
Branch vs Branch & Exception	180	0.891	No	-0.001	Negligible
Branch vs Branch & Output	180	0.374	No	-0.021	Negligible
Branch vs Branch & Execution Time	180	0.490	No	-0.009	Negligible
Branch & Exception vs Exception	180	4.97×10^{-30}	Yes	0.63	Medium
Branch & Output vs Output	180	2.18×10^{-24}	Yes	0.45	Small
Branch vs Branch & Exception	300	0.691	No	-0.003	Negligible
Branch vs Branch & Output	300	0.300	No	-0.023	Negligible
Branch vs Branch & Execution Time	300	0.002	Yes	-0.09	Negligible
Branch & Exception vs Exception	300	4.30×10^{-26}	Yes	0.60	Medium
Branch & Output vs Output	300	6.61×10^{-21}	Yes	0.52	Medium

Table 19 Significance comparisons and effect sizes for **average test case length** between single and multi-objective configurations across different budgets.

Coverage requires more interactions with the class-under-test and requires that a larger number of specialized scenarios be set up and executed compared to a goal-based objective alone, increasing both suite size and test length.

There is a larger increase between Branch and Exception Count and Exception Count alone in both suite size and test length than between Branch and Output Coverage and Output Coverage alone. This is because the Exception Count depends on the number of exceptions discovered, which—in almost all cases—will be fewer than the number of required output partitions for the methods of the class-under-test. Further, tests that trigger exceptions may not achieve high coverage, as the execution path will end when the exception is triggered. If an exception is triggered early in the execution of a particular method, few coverage obligations will be achieved.

RQ4 (Test Suite Contents): Both the test suite size and average test length increase with multi-objective optimization compared to when a goal-based criterion is targeted alone. Branch Coverage tends to impose more obligations than goal-based objectives, leading to the increase.

There is only a small increase in test suite size—and no increase in average test length—between Exception Count and Branch Coverage and Output Coverage and Branch Coverage versus Branch Coverage alone. As discussed above, the number of obligations for the goal-based objectives is small compared to the number for Branch Coverage, so only a small increase in suite size would be expected.

RQ4 (Test Suite Contents): Targeting Exception or Output Coverage in addition to Branch Coverage leads to a small increase in test suite size compared to targeting Branch Coverage alone. However, there is no significant increase in test case length.

We see no or negligible increase in suite size and test length between Branch Coverage and Branch and Execution Time. The Execution Time fitness function differed from the others in that it had no "obligations". Rather, the goal was simply to find the maximum execution time for a test suite during the generation process. Therefore, one would not expect a significant impact on the test suite size. Some impact on test case length would be reasonable, however, as increasing the number of interactions will increase the execution time. That said, the fitness function imposed a high penalty on test case length to prevent the generation of bloated test cases. Further, as shown in Table 12, actual attainment of the Execution Time goal was limited.

RQ4 (Test Suite Contents): Targeting Execution Time in addition to Branch Coverage leads to no or negligible change in suite size or test case length compared to targeting Branch Coverage alone.

5.5 | Impact of Search Budget (RQ5)

Our final hypothesis was the following:

Hypothesis 5: An increase in the search budget will not lead to increased attainment of each objective.

An increased search budget could potentially increase the resulting attainment of each targeted objective. However, we must examine whether this is the case. In any case, we also hypothesize that the relative relationships between single and multi-objective optimization will not fundamentally differ.

We observe that attainment generally did increase. We also observe that the observed trends generally held true. With regard to attained Branch Coverage, the increased search budget led to higher coverage attainment (Table 2) and more suites reaching particular coverage thresholds (Table 8). However, this increase is approximately consistent across configurations, regardless of the targeted fitness functions. The same trends between configurations generally held at both search budgets with regard to total attained coverage, the particular obligations covered, and the rate of coverage attainment.

With regard to coverage of goal-based objectives, an increased search budget led to slightly higher median attainment of Exception Count and Output Coverage (Table 9. However, again, the same general trends were witnessed in comparisons of multi-objective and single-objective generation at both budgets for the most part. Two exceptions emerged (Tables 10 and 11). First, at a higher budget, targeting Branch and Exception yielded significantly better Exception Count than targeting Branch alone (when there was no significant difference at the lower budget). Second, a negligible difference at the lower search budget between targeting Branch and Output and targeting Output Coverage alone in terms of the achieved Output Coverage disappeared at a higher budget.

With regard to fault detection, the number of faults detected increased with the search budget. In addition, we see that the average and 75th percentile likelihood of fault detection also increased with the search budget (Table 13)—as well as the average number of failing tests (Table 15). The general relationships between single and multi-objective configurations held, except that some effect sizes increased at the higher budget (Table 14).

Finally, an increased search budget generally led to little-to-no change in the median test suite size or test case length—however, there was a minor increase in the average suite size (Table 17). The observations with regard to single versus multi-objective generation held across budgets, with small amplifications at the larger search budget (e.g., an increased effect size for Branch and Output versus Output alone at a 300 second budget for the average test case length).

RQ5 (Search Budget): An increased search budget leads to increased Branch Coverage, goal attainment, and fault detection, but does not substantially affect test suite size and average test length.

RQ5 (Search Budget): An increased search budget generally does not fundamentally change—but may increase the effect size of—relationships between single and multi-objective optimization.

6 | DISCUSSION

6.1 | Assessment of Hypotheses

Our study assessed five hypotheses about the relationships between coverage-directed test generation, goal-directed test generation, and multi-objective optimization targeting both coverage and testing goals. Here, we summarize our findings with regard to these hypotheses.

Hypothesis 1: The inclusion of goal-based fitness functions as additional generation targets will not have an impact on the attainment of code coverage, as compared to targeting coverage alone.

Ultimately, our observations **fail to refute this hypothesis**. We found that adding a second goal-based fitness function does not have a significant impact on the final Branch Coverage attained by test suites. Further, in the majority of cases, the addition of a goal-based fitness function does not change the likelihood of covering particular test obligations. Almost all differences in the average coverage of individual obligations are within 10% of when Branch Coverage is targeted alone. Finally, there are almost no significant differences between configurations with regard to the rate of attainment of Branch Coverage.

Hypothesis 2: Targeting both coverage and a goal-based fitness function will not have an impact on the attainment of goal-based fitness functions, as compared to targeting coverage or a goal-based fitness function alone.

Our observations **partially refute this hypothesis**. First, we observed that targeting code coverage *alone* leads to worse goal attainment than directly targeting a goal-based objective, adding evidence to our previous observations [51] that coverage is a prerequisite for goal attainment but does not guarantee attainment.

We observed that targeting Branch Coverage and Exception Count did not yield significant differences from targeting Exception Count alone. However, the multi-objective configuration did have a higher median performance. As observed in our previous work [19, 51], the addition of Branch Coverage can offer feedback that leads to the discovery of more exceptions. However, such cases are rare. Targeting Branch and Output Coverage did yield significant differences from targeting Output Coverage alone, with worse results from the multi-objective configuration. However, the effect size is small at the 180-second budget and negligible at the 300-second budget.

In short, we cannot reject this hypothesis in all situations. When we do, the results (i.e., for Output Coverage), there is a potential loss in performance, but only a small one.

Hypothesis 3: Targeting both coverage and a goal-based fitness function will not have an impact on the fault detection of generated test suites, as compared to targeting coverage or a goal-based fitness function alone.

Our observations **partially refute this hypothesis**. Suites targeting multi-objective configurations detected more faults than single-objective configurations. Suites targeting Branch Coverage alone detected fewer faults than multi-objective configurations, but they did detect more faults than suites targeting goal-based objectives.

In addition, suites targeting Branch Coverage alone or a multi-objective configuration outperform suites targeting a goal-based objective in the likelihood of fault detection with medium–large effect size. However, suites targeting a multi-objective configuration fail to outperform suites targeting Branch Coverage alone with significance in the likelihood of fault detection. That said, targeting a multi-objective configuration does increase the average and 75th percentile performance.

These findings reinforce our previous observations [19, 51] that coverage is needed to discover faults but does not guarantee the selection of the specific inputs needed to trigger a failure. Targeting coverage yields more failures than only targeting testing goals. Targeting a goal *in addition* to coverage resulted in the discovery of more faults than coverage alone by biasing the test input used in the generated suite. However, many faults still remain undetected. Future research should consider additional goal-based fitness functions, and aim to discover which functions can best shape coverage towards an increased likelihood of fault detection.

Hypothesis 4: Targeting both coverage and a goal-based fitness function will not have an impact on the size of the test suite and the average test length, as compared to targeting coverage or a goal-based fitness function alone.

Our observations **partially refute this hypothesis**. Both the test suite size and average test length significantly increase with multi-objective optimization compared to when a goal-based criterion is targeted alone. Branch Coverage tends to impose more obligations than goal-based objectives, leading to the increase.

Targeting Exception or Output Coverage in addition to Branch Coverage also leads to a negligible–small increase in test suite size compared to targeting Branch Coverage alone. However, there is no significant increase in test case length.

Targeting Execution Time in addition to Branch Coverage leads to no or negligible changes in suite size or test length compared to targeting Branch Coverage alone, as the Execution Time fitness function does not have distinct test obligations.

Hypothesis 5: An increase in the search budget will not lead to increased attainment of each objective.

Our observations **partially refute this hypothesis**. An increased search budget leads to increased Branch Coverage, goal attainment, and fault detection, but does not substantially affect suite size and test length. Additionally, an increased search budget does not fundamentally change—but may amplify—relationships between single and multi-objective optimization.

6.2 | Threats to Validity

External Validity

For this study, we focused on case examples of real faults from the Defects4J dataset. The use of this dataset introduces certain threats to external validity. First, the faults used in the study represent only 14 Java projects. This is a relatively small number of projects. Nevertheless, we believe that Defects4J offers enough case examples that our results are generalizable to, at minimum, other small to medium-sized Java projects. Further, as Defects4J is used extensively in search-based test generation research [21], the use of Defects4J examples enables comparisons of our results with other research, and eases replication.

The set of specific faults used from the Defects4J dataset may also introduce selection bias, as certain types of faults or certain projects may be overrepresented or underrepresented. While we lacked experimental resources to consider all faults, we worked to ensure that we drew a proportional sample. We initially selected 206 faults, then retained 93 faults in the final experiment. This set remains large enough to offer a broad range of case examples and we do not believe that any single project is overrepresented.

We have based our research on a single test generation framework, EvoSuite. There are many search-based methods of generating tests and these methods may yield different results. Unfortunately, no other generation framework offers the same variety of fitness functions, particularly goal-based fitness functions. Therefore, a more thorough comparison of tools cannot be made at this time. In addition, by focusing on a single generation framework, we ensure that all test suites are compared in a controlled and fair manner.

Within EvoSuite, we also only employed one multi-objective algorithm, whole suite generation. Other algorithms may yield different results, as search objectives may be targeted through different mechanisms. We chose this algorithm to enable comparison to past research, and chose to focus on a single algorithm to perform a focused and detailed analysis of the data collected. We believe that the general trends observed would hold regardless of algorithm, even if specific results varied. In future work, we will consider the influence of algorithm more closely.

Internal Validity

Evolutionary algorithms inherently introduce randomness, affecting result consistency. To mitigate this, we conducted multiple trials, aiming to average out randomness and stabilize outcomes. To control experiment cost, we only generated ten test suites for each class, budget, and fitness function configuration. A larger number of repetitions may yield different results. However, given the consistency of our results, we believe that this is a sufficient number of trials to draw stable conclusions from.

Conclusion Validity

Conclusion validity depends on our choice of statistical tests and the assumptions underlying those tests. Data segmentation allowed for targeted analysis of different budget and fitness function configurations, with descriptive statistics and box plots providing an initial overview that could be used to validate the results of statistical analyses. We have favored non-parametric methods, as distribution characteristics were not known a priori, and normality cannot be assumed.

7 | CONCLUSION

Past research has suggested the potential benefit of blending code coverage and goal-based fitness functions. While multi-objective generation has been previously studied, how these objectives interact—and, in particular, the interaction between coverage and goal-based fitness functions—has not been studied in depth. Therefore, in this study, we assessed and explored five hypotheses about this interaction and its effects on code coverage, goal attainment, fault detection, the size of the test suite, the length of test cases, and the impact of the search budget.

Ultimately, our observations suggest that there are more benefits than drawbacks in targeting multiple objectives over a single objective. Targeting multiple objectives does not reduce code coverage, and goal attainment is either not reduced, or only minorly reduced. At the same time, targeting multiple objectives can lead to the detection of more faults and a higher average likelihood of fault detection. Multi-objective optimization does lead to larger test suites, but imposes only a small increase over suites targeting code coverage alone, and test case length is not significantly increased.

The benefits of multi-objective optimization are often more limited than hypothesized in past research, but the improvements in fault detection are still sufficient enough to recommend multi-objective optimization over targeting coverage or testing goals

alone. Our study offers insight into how coverage and goal-based objectives interact during multi-objective test generation, offering guidance to researchers and testers and a starting point for future research on multi-objective test generation.

In future work, we would like to continue to explore these—and other—hypotheses with an expanded scope and and consideration of additional experimental variables. We will target a wider variety of projects and faults, and will also vary the metaheuristic algorithms used to perform multi-objective generation (e.g., contrasting whole suite generation and MOSA). In addition, we will consider combinations of more than two fitness functions. Our past research found that EvoSuite's default combination of eight fitness functions performed worse at fault detection than simply targeting Branch Coverage under the same budget, as competing objectives and overhead of calculating fitness limited test suite evolution. However, a subset of more than two and less than eight functions may yield highly effective results. Finally, we will also explore situations where fitness functions can be given different weights during optimization—e.g., targeting both a goal-based and coverage-based function and giving heavier weight to the goal-based function.

8 | ACKNOWLEDGMENTS

This research was supported by Vetenskapsrådet grants 2019-05275 and 2020-05272. Computing resources were provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by Vetenskapsrådet grant agreement 2022-06725.

References

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

[2] Almulla, H. and G. Gay, 2022: Learning how to search: generating effective test cases through adaptive fitness function selection. *Empirical Software Engineering*, **27**, no. 2, 38, doi:10.1007/s10664-021-10048-8.
URL https://doi.org/10.1007/s10664-021-10048-8

[3] Alshahwan, N., X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, 2018: Deploying search based software engineering with sapienz at facebook. *Search-Based Software Engineering*, Springer International Publishing, Cham, 3–45.

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

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

[6] Aniche, M., 2022: *Effective Software Testing: A developer's guide*. Simon and Schuster.

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

[8] — 2018: Test suite generation with the many independent objective (mio) algorithm. *Information and Software Technology*, **104**, 195–206, doi:https://doi.org/10.1016/j.infsof.2018.05.003.
URL https://www.sciencedirect.com/science/article/pii/S0950584917304822

[9] Barr, E., M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, 2015: The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, **41**, no. 5, 507–525, doi:10.1109/TSE.2014.2372785.

[10] Campos, J., Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, 2018: An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, **104**, 207–235, doi:https://doi.org/10.1016/j.infsof.2018.08.010.
URL https://www.sciencedirect.com/science/article/pii/S0950584917304858

[11] Chekam, T. T., M. Papadakis, Y. Le Traon, and M. Harman, 2017: An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, 597–608.

[12] Chen, Y. T., R. Gopinath, A. Tadakamalla, M. D. Ernst, R. Holmes, G. Fraser, P. Ammann, and R. Just, 2020: Revisiting the relationship between fault detection, test adequacy criteria, and test set size. *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, 237–249.

[13] Cliff, N., 1993: Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, **114**, no. 3, 494.

[14] de Normalización, O. I., 2011: *ISO 26262: Road Vehicles : Functional Safety*. ISO.
URL https://books.google.se/books?id=3gcAjwEACAAJ

[15] Deb, K., A. Pratap, S. Agarwal, and T. Meyarivan, 2002: A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*, **6**, no. 2, 182–197, publisher: IEEE.

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

[17] Feldt, R., S. Poulding, D. Clark, and S. Yoo, 2016: Test set diameter: Quantifying the diversity of sets of test cases. *2016 IEEE international conference on software testing, verification and validation (ICST)*, IEEE, 223–233.

[18] Feldt, R., R. Torkar, T. Gorschek, and W. Afzal, 2008: Searching for cognitively diverse tests: Towards universal test diversity metrics. *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, IEEE, 178–186.

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

[20] — 2018: To call, or not to call: Contrasting direct and indirect branch coverage in test generation. *Proceedings of the 11th International Workshop on Search-Based Software Testing*, ACM, New York, NY, USA, SBST 2018.

[21] Gay, G. and R. Just, 2020: Defects4j as a challenge case for the search-based software engineering community. *Search-Based Software Engineering*, A. Aleti and A. Panichella, Eds., Springer International Publishing, Cham, 255–261.

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

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

[24] Hemmati, H., 2015: How effective are code coverage criteria? *2015 IEEE International Conference on Software Quality, Reliability and Security*, 151–156.

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

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

[27] Istanbuly, D., M. Zimmer, and G. Gay, 2023: How do different types of testing goals affect test case design? *IFIP International Conference on Testing Software and Systems*, Springer, 97–114.

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

[29] Kanapala, A. and G. Gay, 2018: Mapping class dependencies for fun and profit. *Proceedings of the Symposium on Search-Based Software Engineering*, Springer Verlag, SSBSE 2018.

[30] Kechagia, M., X. Devroey, A. Panichella, G. Gousios, and A. van Deursen, 2019: Effective and efficient api misuse detection via exception propagation and search-based testing. *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Association for Computing Machinery, New York, NY, USA, ISSTA 2019, 192–203.
URL https://doi.org/10.1145/3293882.3330552

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

[32] Lukasczyk, S., F. Kroiß, and G. Fraser, 2023: An empirical study of automated unit test generation for python. *Empirical Software Engineering*, **28**, no. 2, doi:10.1007/s10664-022-10248-w.

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

[34] Mann, H. B. and D. R. Whitney, 1947: On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, 50–60.

[35] Manès, V. J., H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, 2021: The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, **47**, no. 11, 2312–2331, doi:10.1109/TSE.2019.2946563.

[36] McGill, R., J. W. Tukey, and W. A. Larsen, 1978: Variations of box plots. *The american statistician*, **32**, no. 1, 12–16.

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

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

[39] Meng, Y., G. Gay, and M. Whalen, 2018: Ensuring the observability of structural test obligations. *IEEE Transactions on Software Engineering*, 1–1, doi:10.1109/TSE.2018.2869146, available at http://greggay.com/pdf/18omcdc.pdf.

[40] Moghadam, M. H., M. Saadatmand, M. Borg, M. Bohlin, and B. Lisper, 2022: An autonomous performance testing framework using self-adaptive fuzzy reinforcement learning. *Software Quality Journal*, **30**, no. 1, 127–159, doi:10.1007/s11219-020-09532-z.
URL https://doi.org/10.1007/s11219-020-09532-z

[41] Moy, Y., E. Ledinot, H. Delseny, V. Wiels, and B. Monate, 2013: Testing or formal verification: Do-178c alternatives and industrial experience. *IEEE Software*, **30**, no. 3, 50–57, doi:10.1109/MS.2013.43.

[42] Palomba, F., A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, 2016: Automatic test case generation: What if test code quality matters? *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 130–141.

[43] Panichella, A., F. M. Kifetew, and P. Tonella, 2017: Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, **44**, no. 2, 122–158.

[44] — 2017: Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, **44**, no. 2, 122–158.

[45] — 2018: A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology*, **104**, 236–256, doi:https://doi.org/10.1016/j.infsof.2018.08.009.
URL https://www.sciencedirect.com/science/article/pii/S0950584917304950

[46] Parsai, A. and S. Demeyer, 2020: Comparing mutation coverage against branch coverage in an industrial setting. *International Journal on Software Tools for Technology Transfer*, **22**, no. 4, 365–388.

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

[48] Ramírez, A., J. R. Romero, and S. Ventura, 2019: A survey of many-objective optimisation in search-based software engineering. *Journal of Systems and Software*, **149**, 382–395, doi:https://doi.org/10.1016/j.jss.2018.12.015.
URL https://www.sciencedirect.com/science/article/pii/S0164121218302759

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

[50] Rojas, J. M., M. Vivanti, A. Arcuri, and G. Fraser, 2017: A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, **22**, no. 2, 852–893, doi:10.1007/s10664-015-9424-2.
URL 10.1007/s10664-015-9424-2

[51] Salahirad, A., H. Almulla, and G. Gay, 2019: Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Software Testing, Verification and Reliability*, **29**, no. 4-5, e1701, doi:10.1002/stvr.1701, e1701 stvr.1701.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1701

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

[53] Staats, M., M. W. Whalen, A. Rajan, and M. P. Heimdahl, 2010: Coverage metrics for requirements-based testing: Evaluation of effectiveness. *Proceedings of the Second NASA Formal Methods Symposium*, NASA.

[54] Tukey, J. W. et al., 1977: *Exploratory data analysis*, volume 2. Reading, MA.

[55] Vogl, S., S. Schweikl, G. Fraser, A. Arcuri, J. Campos, and A. Panichella, 2021: Evosuite at the sbst 2021 tool competition. *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, IEEE, 28–29.

[56] Weiglhofer, M., G. Fraser, and F. Wotawa, 2009: Using coverage to automate and improve test purpose based testing. *Information and Software Technology*, **51**, no. 11, 1601–1617.

[57] Wilcoxon, F., 1945: Individual comparisons by ranking methods. *Biometrics Bulletin*, **1**, no. 6, 80–83, doi:doi.org/10.2307/3001968.

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

[59] — 2010: Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software*, **83**, no. 4, 689–701.

[60] Zhou, Z., Y. Zhou, C. Fang, Z. Chen, and Y. Tang, 2023: Selectively combining multiple coverage goals in search-based unit test generation. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, Association for Computing Machinery, New York, NY, USA, ASE '22.
URL https://doi.org/10.1145/3551349.3556902