Combining Multiple Coverage Criteria in Search-Based Unit Test Generation

José Miguel Rojas[1], José Campos[1], Mattia Vivanti[2],
Gordon Fraser[1], and Andrea Arcuri[3]

[1] Department of Computer Science, The University of Sheffield, United Kingdom
[2] Università della Svizzera Italiana (USI), Switzerland
[3] Scienta, Norway, and University of Luxembourg, Luxembourg

Abstract. Automated test generation techniques typically aim at maximising coverage of well-established structural criteria such as statement or branch coverage. In practice, generating tests only for one specific criterion may not be sufficient when testing object oriented classes, as standard structural coverage criteria do not fully capture the properties developers may desire of their unit test suites. For example, covering a large number of statements could be easily achieved by just calling the `main` method of a class; yet, a good unit test suite would consist of smaller unit tests invoking individual methods, and checking return values and states with test assertions. There are several different properties that test suites should exhibit, and a search-based test generator could easily be extended with additional fitness functions to capture these properties. However, does search-based testing scale to combinations of multiple criteria, and what is the effect on the size and coverage of the resulting test suites? To answer these questions, we extended the EvoSuite unit test generation tool to support combinations of multiple test criteria, defined and implemented several different criteria, and applied combinations of criteria to a sample of 650 open source Java classes. Our experiments suggest that optimising for several criteria at the same time is feasible without increasing computational costs: When combining nine different criteria, we observed an average decrease of only 0.4% for the constituent coverage criteria, while the test suites may grow up to 70%.

1 Introduction

To support developers in creating unit test suites for object-oriented classes, automated tools can produce small and effective sets of unit tests. Test generation is typically guided by structural coverage criteria; for example, the search-based unit test generation tool EvoSuite by default generates test suites optimised for branch coverage [4], and these tests can achieve higher code coverage than manually written ones [8]. However, although manual testers often *check* the coverage of their unit tests, they are usually not *guided* by it in creating their test suites. In contrast, automated tools are only guided by code coverage, and do not take into account *how* this coverage is achieved. As a result, automatically generated unit tests are fundamentally different to manually written ones, and may not satisfy the expectations of developers, regardless of coverage benefit.

```
public class ArrayIntList
      extends RandomAccessIntList
      implements IntList, Serializable {
 public int set(int index, int element) {
   checkRange(index);
   incrModCount();
   int oldval = _data[index];
   _data[index] = element;
   return oldval;
 }
}
```

```
@Test
public void test9() throws Throwable {
 ArrayIntList arrayIntList0 = new ArrayIntList();
 // Undeclared exception!
 try {
   int int0 = arrayIntList0.set(200, 200);
   fail("Expecting IndexOutOfBoundsException");
 } catch(IndexOutOfBoundsException e) {
   // Should be at least 0 and less than 0, found 200
 }
}
```

(a) Source code excerpt.   (b) Test case generated by EvoSuite.

Fig. 1: This example shows how EvoSuite covers method `set` of the class `ArrayIntList`: the method is called, but statement coverage is not achieved.

```
public class Complex {
 public Complex log() {
   if (isNaN) {
     return NaN;
   }
   return createComplex(FastMath.log(abs()),
           FastMath.atan2(imaginary, real));
 }

 public Complex pow(double x) {
   return this.log().multiply(x).exp();
 }
 ...
}
```

```
@Test
public void test1() throws Throwable {
 Complex complex0 = new Complex(Double.NaN);
 Complex complex1 = complex0.pow(Double.NaN);
 assertEquals(Double.NaN, complex1.getArgument(),0.01D);
}

@Test
public void test2() throws Throwable {
 Complex complex0 = Complex.ZERO;
 Complex complex1 = complex0.pow(complex0);
 assertFalse(complex1.isInfinite());
 assertTrue(complex1.isNaN());
}
```

(a) Source code excerpt.   (b) Test cases generated by EvoSuite.

Fig. 2: This example shows how EvoSuite covers method `log`, even though there is no test that directly calls the method.

For example, consider the excerpt of class `ArrayIntList` from the Apache Commons Primitives project in Figure 1a. Applying EvoSuite results in a test suite including the test case in Figure 1b: The test calls `set`, but with parameters that do not pass the input validation by `checkRange`, such that an exception is thrown. Nevertheless, EvoSuite believes `set` is covered with this test, and adds no further tests, thus not even satisfying statement coverage in the method. The reason is that EvoSuite follows common practice in bytecode-based coverage analysis, and only checks if branching statements evaluated to true and false [13].

To cover method `set` fully, one would also need to aim at covering all instructions. However, when optimising test suites to cover branches *and* instructions, automated techniques may find undesired ways to satisfy the target criteria. For example, consider the excerpt of class `Complex` from the Apache Commons Math project shown in Figure 2a: EvoSuite succeeds to cover method `log`, but because `log` is called by `pow`, in the end often only tests calling `pow` (see Figure 2b) are retained, which makes it hard to check the behaviour of `log` independently (e.g., with test assertions on the return value of `log`), or to debug problems caused by faults in `log`. Thus, a good test suite has different properties, which cannot easily be captured by any individual structural coverage criterion.

In this paper, we define different criteria and their fitness functions to guide search-based test suite generation, and investigate the effects of combining these during test generation. Such a combination of multiple optimisation criteria

raises concerns about the effects on the size of resulting test sets, as well as on the effectiveness of the test generators used for this optimisation. To investigate these concerns, we performed a set of experiments on a sample of 650 open source classes. In detail, the contributions of this paper are as follows:

– Identification of additional criteria to guide unit test suite generation.
– Implementation of these criteria as fitness functions for a search-based test suite optimisation.
– An empirical study of the effects of multiple-criterion optimisation on effectiveness, convergence, and test suite size.

Our experiments suggest that optimising for several criteria at the same time is feasible without increasing computational costs, or sacrificing coverage of the constituent criteria. The increase in size depends on the combined criteria; for example, optimising for line and branch coverage instead of just line coverage increases test suites by only 10% in size., while optimising for nine different criteria leads to an increase of 70% in size. The effects of the combination of criteria on the coverage of the constituent criteria are minor; for criteria with fine-grained fitness functions the overall coverage may be reduced slightly (0.4% in our experiments), while criteria with coarse fitness functions (e.g. method coverage) may benefit from the combination with other criteria.

2   Whole Test Suite Generation for Multiple Criteria

In principle, the combination of multiple criteria is independent of the underlying test generation approach. For example, dynamic symbolic execution can generate test suites for any coverage criteria as by-product of the path exploration [10]. However, our initial usage scenario lies in unit testing for object oriented classes, an area where search-based approaches have been shown to perform well. In search-based testing, the test generation problem is cast as a search problem, such that efficient meta-heuristic search algorithms can be applied to create tests.

2.1   Whole Test Suite Generation

Whole test suite generation refers to the generation of test suites, which has been shown to be more effective than iteratively generating individual test cases [5]. When applying search-based testing for this task, a common technique is to use a genetic algorithm, which starts with a population of random test suites, and then evolves these using standard evolutionary operators [5]. The evolution is guided by a *fitness function* that estimates how close a candidate solution is to the optimal solution; i.e., 100% coverage in coverage-oriented test generation.

A *test suite* is a collection of *unit tests* for a target Class Under Test (CUT). The CUT comprises a set of methods, each of which consists of a list of statements. Each statement can be a conditional statement (e.g., `if`), a method call or a regular statement. A conditional statement results in two branches depending on the evaluation of its predicate. A *unit test* is an executable function which sets up a test scenario, calls some methods in the CUT, and checks that the observed behaviour matches the expected one. For simplicity, a unit test can be regarded as a sequence of calls to methods of the CUT. Executing a unit

test yields an *execution trace*, i.e., a sequence of executed statements which can either end normally with a regular statement, or with an uncaught exception.

2.2 Fitness Functions

In search-based test suite generation, a fitness function measures how good a test suite is with respect to the search optimisation objective, which is usually defined according to a test coverage criterion. Importantly, a fitness function usually also provides additional search guidance leading to satisfaction of the goals. For example, just checking in the fitness function whether a coverage target is achieved would not give any guidance to help covering it.

Method Coverage. Method Coverage is the most basic criterion for classes and requires that all methods in the CUT are executed by a test suite at least once, either via a direct call from a unit test or via indirect calls.

Top-level Method Coverage. For regression test suites it is important that each method is also invoked directly (cf. Figure 2). Top-level Method Coverage requires that all methods are covered by a test suite such that a call to the method appears as a statement in a test case.

No-exception Top-level Method Coverage. In practice, classes often consist of many short methods with simple control flow. Often, a generated test suite achieves high levels of coverage by calling these simple methods in an invalid state or with invalid parameters (cf. Figure 1). To avoid this, No-exception Top-level Method Coverage requires that all methods are covered by a test suite via direct invocations from the tests and considering only normal-terminating executions (i.e., no exception).

The fitness functions for Method Coverage, Top-level Method Coverage and No-exception Top-level Method Coverage are *discrete* and thus have no possible guidance. Fitness values are simply calculated by counting the methods that have been covered by a test suite. Let *TotalMethods* be the set of all public methods in the CUT and *CoveredMethods* be the set of methods covered by the test suite, then:

$$f_{crit}(Suite) = \mid TotalMethods \mid - \mid CoveredMethods_{crit} \mid$$

Line Coverage. A basic criterion in procedural code is statement coverage, which requires all statements to be executed. Modern test generation tools for Java or C# often use the bytecode representation for test generation, and bytecode instructions may not directly map to source code statements. Therefore, a more common alternative in coverage analysis tools, and the de-facto standard for most Java bytecode-based coverage tools, is to consider coverage of *lines* of code. Each statement in a class has a defined line, which represents the statement's location in the source code of the class. The source code of a class consists of non-comment lines, and lines that contain no code (e.g., whitespace or comments). A unit test suite satisfies the Line Coverage criterion only if it covers each non-comment source code line of the CUT with at least one of its tests. Line Coverage is very easy to visualise, interpret, and to implement in an analysis tool; all these reasons probably contribute to its popularity.

To cover each line of source code, we need to ensure that each basic code block is reached. In traditional search-based testing, this reachability would be expressed by a combination of approach-level and branch distance [14]. The approach-level measures how far an individual execution and the target statement are in terms of the control dependencies (i.e., distance between point of diversion and target statement in control dependence graph). The branch distance estimates how far a predicate is from evaluating to a desired target outcome. For example, given a predicate `x == 5` and an execution with value 3, the branch distance to the predicate evaluating to true would be $|3 - 5| = 2$, whereas an execution with value 4 is closer to being true with a branch distance of $|4 - 5| = 1$. Branch distances can be calculated by applying a set of standard rules [12, 14].

In contrast to test case generation, if we optimise a test suite to execute all statements then the approach level is not necessary, as all statements will be executed by the same test suite. Thus, we only need to consider the branch distance of all branches that are control dependencies of any of the statements in the CUT. That is, for each conditional statement that is a control dependency for some other statement in the code, we require that the branch of the statement leading to the dependent code is executed. Thus, the Line Coverage fitness value of a test suite can be calculated by executing all its tests, calculating for each executed statement the minimum branch distances $d_{min}(b, Suite)$ among all observed executions to every branch b in the set of control dependent branches B_{CD}, i.e., the distances to all the branches which need to be executed in order to reach such a statement. The Line Coverage fitness function is thus defined as:

$$f_{LC}(Suite) = \nu(|\ NCLs\ | - |\ CoveredLines\ |) + \sum_{b \in B_{CD}} \nu(d_{min}(b, Suite))$$

where $NCLs$ is the set of all non-comment lines of code in the CUT, $CoveredLines$ is the total set of lines covered by the execution traces of every test in the suite, and $\nu(x)$ is a normalising function in $[0, 1]$ (e.g., $\nu(x) = x/(x + 1)$) [2].

Branch Coverage. The concept of covering branches is also well understood in practice and implemented in popular tools, even though the practical definition of branch coverage may not always match the more theoretical definition of covering all edges of a program's control flow. Branch coverage is often interpreted as maximising the number of branches of conditional statements that are covered by a test suite. Hence, a unit test suite is said to satisfy the Branch Coverage criterion if and only if for every branch statement in the CUT, it contains at least one unit test whose execution evaluates the branch predicate to *true*, and at least one unit test whose execution evaluates the branch predicate to *false*.

The fitness function for the Branch Coverage criterion estimates how close a test suite is to covering all branches of the CUT. The fitness value of a test suite is measured by executing all its tests, keeping track of the branch distances $d(b, Suite)$ for each branch in the CUT. Then:

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

Here, $d(b, Suite)$ for branch $b \in B$ (where B is the set of all branches in the CUT) on the test suite is defined as follows:

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

Note that a predicate must be executed at least twice, because we need to cover the true and false evaluation of the predicate; if the predicate were only executed once, then the search could theoretically oscillate between true and false.

Direct Branch Coverage. When a test case covers a branch in a public method indirectly, i.e., without directly invoking the method that contains the branch, it is more difficult to understand how the test relates to the branch it covers (cf. Figure 2). Anecdotal evidence, from previous work with EVOSUITE, also indicates that developers dislike tests that cover branches indirectly, because they are harder to understand and to extend with assertions [8]. Direct Branch Coverage requires each branch in a public method of the CUT to be covered by a direct call from a unit test, but makes no restriction on branches in private methods. The fitness function is the same as the Branch Coverage fitness function, but only methods directly invoked by the test cases are considered for the fitness and coverage computation of branches in public methods.

Output Coverage. Class `ArrayIntList` from Figure 1 has a method `size` that simply returns the value of a member variable capturing the size of the internal array; class `Complex` from Figure 2 has methods `isNaN` or `isInfinite` returning boolean member values. Such methods are known as *observers* or *inspectors*, and method, line, or branch coverage are all identical for such methods. Developers in this case sometimes write unit tests to cover not only in the input values of methods, but also in the output (return) values they produce; indeed output diversity can help improve the fault detection capability [1].

To account for output uniqueness and diversity, the following function maps method return types to abstract values that serve as output coverage goals:

$$output(Type) = \begin{cases} \{true, false\} & if \ Type \equiv Boolean \\ \{-, 0, +\} & if \ Type \equiv Number \\ \{alphabetical, digit, *\} & if \ Type \equiv Char \\ \{null, \neq null\} & otherwise \end{cases}$$

A unit test suite satisfies the Output Coverage criterion only if for each public method M in the CUT and for each $V_{abst} \in output(type(M))$, there is at least one unit test whose execution contains a call to method M for which the concrete return value is characterised by the abstract value V_{abst}.

The fitness function for the Output Coverage criterion is then defined as:

$$f_{OC}(Suite) = \sum_{g \in G} \nu(d_o(g, Suite))$$

where G is the total set of output goals for the CUT and $d_o(g, Suite)$ is an output distance function that takes as input a goal $g = \langle M, V_{abst} \rangle$:

$$d_o(g, Suite) = \begin{cases} 0 & \text{if } g \text{ is covered by at least one test,} \\ \nu(d_{num}(g, Suite)) & \text{if } type(M) \equiv Number \text{ and } g \text{ is not covered,} \\ 1 & \text{otherwise.} \end{cases}$$

In the case of methods declaring numeric return types, the search algorithm is guided with normalised numeric distances (d_{num}). For example, if a call to a method m with integer return type is observed in an execution trace and its return value is 5 (positive integer), the goal $\langle m, + \rangle$ has been covered, and the distances 5 and 6 are computed for goals $\langle m, 0 \rangle$ and $\langle m, - \rangle$, respectively.

Weak Mutation. Test generation tools typically include values generated to satisfy constraints or conditions, rather than values developers may prefer; in particular, anecdotal evidence suggests developers like boundary cases. Test generation can be forced to produce such values using weak mutation testing, which applies small code modifications to the CUT, and then checks if there exists a test that can distinguish between the original and the *mutant*. In weak mutation, a mutant is considered to be covered ("killed") if the execution of a test on the mutant leads to a different state than the execution on the CUT. A unit test suite hence satisfies the Weak Mutation criterion if and only if for each mutant for the CUT at least one its tests reaches state infection.

The fitness function for the Weak Mutation criterion guides the search using infection distances with respect to a set of mutation operators [7]. We assume a minimal infection distance function $d_{min}(\mu, Suite)$ exists and define:

$$d_w(\mu, Suite) = \begin{cases} 1 & \text{if mutant } \mu \text{ was not reached,} \\ \nu(d_{min}(\mu, Suite)) & \text{if mutant } \mu \text{ was reached.} \end{cases}$$

This results in the following fitness function for weak mutation testing:

$$f_{WM}(Suite) = \sum_{\mu \in \mathcal{M}_C} d_w(\mu, Suite)$$

where $\mathcal{M}_C$ is the set of all mutants generated for the CUT.

Exception Coverage. One of the most interesting aspects of test suites not captured by standard coverage criteria is the occurrence of actual faults. If exceptions are directly thrown in the CUTs with a `throw` statement, those will be retained in the final test suites if for example we optimise for line coverage. However, this might not be the case if exceptions are unintended (e.g., a null-pointer exception when calling a method on a null instance) or if thrown in the body of external methods called by the CUT. Unfortunately, it is not possible to know ahead of time the total number of feasible undeclared exceptions (e.g., null-pointer exceptions), in particular as the CUT could use custom exceptions that extend the ones in the Java API.

As coverage criterion, we consider all possible exceptions in each method of the CUT. However, in contrast to the other criteria, it cannot be defined with a percentage (e.g., we cannot say a test suite covers 42% of the possible exceptions). We rather use the sum of all unique exceptions found per CUT method as metric to maximise. The fitness function for Exception Coverage is thus also discrete, and is calculated in terms of the number of exceptions N_E, explicit and implicit, that have been raised in the execution of all the tests in the suite:

$$f_{EC}(Suite) = \frac{1}{1 + N_E}$$

2.3   Combining Fitness Functions

All criteria considered in this paper are non-conflicting: we can always add new tests to an existing suite to increase the coverage of a criterion without decreasing the coverage of the others. However, with limited time it may be necessary to balance the criteria, e.g., by prioritising weaker ones to avoid over-fitting for just some of the criteria involved. Thus, multi-objective optimisation algorithms based on Pareto dominance are less suitable than a linear combination of the different objectives, and we can define a combined fitness function for a set of n non-conflicting individual fitness functions $f_1 \ldots f_n$ as: $f_{comp} = \sum_{i=1}^{n} w_i \times f_i$, where $w_1 \ldots w_n$ are weights assigned to each individual function which allow for prioritisation of the fitness functions involved in the composition. Given enough time, a combined fitness search is expected to have the same result for each involved non-conflicting fitness function as if they were optimised for individually.

For some of the above-defined fitness functions, a natural partial order exists. For instance, Method Coverage subsumes Top-level Method Coverage. The intuition is that we first want to cover all methods, independently of whether they are invoked directly from a test case statement or not. In turn, Top-level Method Coverage subsumes No-Exception Top-level Method Coverage, that is, covering all methods with direct calls from test cases is more general than covering all methods with direct calls from test cases which do not raise any exception. However, there is no natural order between other functions like for instance Output Coverage and Weak Mutation. In this paper, we arbitrarily assign $w_i = 1$ for all i and leave the question of what are optimal w_i values for future work.

3   Experimental Evaluation

In order to better understand the effects of combining multiple coverage criteria, we empirically aim to answer the following research questions:

RQ1 What are the effects of adding a second coverage criterion on test suite size and coverage?

RQ2 How does combining of multiple coverage influence the test suite size?

RQ3 Does combining multiple coverage criteria lead to worse performance of the constituent criteria?

RQ4 How does coverage vary with increasing search budget?

3.1   Experimental Setup

Unit test generation tool. We have implemented the discussed criteria in the EvoSuite [4] tool for automatic unit test suite generation. EvoSuite uses a genetic algorithm where each individual is a test suite [5]. Once a test suite has been generated, EvoSuite applies *minimisation* in order to optimise the size of the resulting test suite both in terms of total number of lines of code and in number of unit tests. For each coverage goal defined by the selected criterion, a test that covers this goal is selected from the test suite. Then, on a copy of that test, all statements that do not contribute to satisfaction of the goal are successively removed. When minimising for multiple criteria, the order in which each criterion is evaluated may influence the resulting minimised test suite. In particular, if criterion C_1 subsumes criterion C_2, then minimising for criterion C_2 first and then for C_1 may lead to tests being added during minimisation for C_2, but made redundant later, by tests added during minimisation for C_1. EvoSuite counters this problem with a second minimisation pass where a final minimised test suite with no redundant tests is produced.

Subject Selection. We used the SF110 corpus [6] of Java classes for our experimental evaluation. SF110 consists of more than 20,000 classes in 110 projects; running experiments on all classes would require an infeasibly large amount of resources. Hence, we decided to select a stratified random sample of 650 classes. That is, we constructed the sample iteratively such that in each iteration we first selected a project at random, and then from that project we selected a class and added it to the sample. As a result, the sample contains classes from all 110 projects, totalling 63,191 lines of code.

Experiment Procedure. For each selected class, we ran EvoSuite with ten different configurations: 1) All fitness functions combined; 2) Only Line Coverage (baseline); 3-10) For each fitness function f defined in Section 2.2 (except Line Coverage) a fitness function combining f and Line Coverage. Combining the other criteria with Line Coverage instead of using each of them in isolation allows a more objective evaluation, since not all the fitness functions for these other criteria can provide guidance to the search on their own. Each configuration was run using two time values for the search: 2 and 10 minutes. To take the randomness of the genetic algorithm into account, we repeated the two minutes experiments 40 times, and the 10 minute experiments five times.

Experiment Analysis. We used coverage as the main measurement of effectiveness, for all the test criteria under study. Furthermore, we also analysed the size of the resulting test suites; as the number of unit tests could be misleading, we analysed the size of a test suite in terms of its total number of statements. Statistical analysis follows the guidelines discussed in [3]: We use the Wilcoxon-Mann-Whitney statistical symmetry test to assess the performance of different experiments. Furthermore, we use the Vargha-Delaney $\hat{A}_{ab}$ to evaluate if a particular configuration a used on experiments performed better than another configuration b. E.g, a $\hat{A}_{ab}$ value of 0.5 means equal performance between configurations; when $\hat{A}_{ab}$ is less than 0.5, the first configuration (a) is worse; and when $\hat{A}_{ab}$ is more than 0.5, the second configuration (b) is worse.

Table 1: Coverage results for each configuration, average of all runs for all CUTs. Size is measured in number of statements in the final minimised test suites.

Criteria	Lines	Branches	D. Branches	Methods	Top Methods	M. No Exc.	Exceptions	Mutation	Output	Size
ALL	0.78	0.75	0.75	0.87	0.90	0.88	1.35	0.75	0.64	38.01
Lines	0.78	0.73	0.22	0.81	0.74	0.71	0.45	0.69	0.27	22.25
L. & Branches	0.78	0.77	0.24	0.81	0.74	0.72	0.47	0.70	0.27	24.92
L. & D. Branches	0.78	0.76	0.76	0.87	0.85	0.82	0.48	0.70	0.27	26.73
L. & Methods	0.79	0.73	0.22	0.87	0.80	0.77	0.46	0.70	0.27	22.33
L. & Top Methods	0.78	0.73	0.22	0.87	0.89	0.86	0.48	0.70	0.27	24.89
L. & M. No Exc.	0.78	0.73	0.23	0.87	0.89	0.88	0.40	0.69	0.27	25.26
L. & Exceptions	0.78	0.72	0.22	0.81	0.78	0.70	1.93	0.70	0.27	28.00
L. & Mutation	0.79	0.75	0.23	0.81	0.75	0.72	0.50	0.76	0.27	27.45
L. & Output	0.77	0.71	0.21	0.80	0.77	0.75	0.36	0.69	0.64	23.98

3.2   Results and Discussion

RQ1: What are the effects of adding a second coverage criterion on test suite size and coverage? Table 1 shows the results of the experiments when using a two minute timeout for the search. Considering line coverage as baseline, adding a further coverage criterion does not increase test suite size by a large amount. For example, adding branch coverage only increases average test suite size from 22.25 statements to 24.92 (a relative $\frac{24.92-22.25}{22.25} = 12\%$ increase). The largest increase is for the Exception Coverage testing criterion, which adds a further $28.00 - 22.25 = 5.75$ statements on average to the test suites.

Regarding coverage of the criteria, already a basic criterion like line coverage can achieve reasonable results. For example, targeting also branch coverage explicitly only increases it by 3% (from 73% to 77%). For other criteria, improvements are higher. For example, we obtain a $88 - 71 = 17\%$ coverage improvement of No-exception Top-level Method Coverage, although with the need of $25.26 - 22.25 = 3.01$ more statements. Of particular interest is the case of Output coverage, where a $64 - 27 = 37\%$ increase is achieved with only slightly larger test suites (less than two statements). The Direct Branch Coverage criterion shows the largest increase ($76 - 22 = 54\%$), which confirms that in the traditional approach code is often covered through indirect calls; this increase comes at the cost of $26.73 - 22.25 = 4.48$ statements on average.

> **RQ1:** *In our experiments, adding a second criterion increased test suites size by 14%, and coverage by 20% over line coverage test suites.*

RQ2: How does combining of multiple coverage influence the test suite size? When combining all criteria together, test suite sizes increase substantially, from 22.25 to 38.01 statements. However, we argue that the resulting test suites could still be manageable for developers: Their size is still less than twice the size of the average baseline test suite. Interestingly, this increase of 15.76 ($38.01-22.25$) is also less than the sum of the increases observed for each criterion in isolation (25.56). This shows that the criteria are related and lead to coincidental coverage, where tests covering one particular goal may lead to coverage of other goals.

> **RQ2:** *In our experiments, combining all nine criteria increased test suites size by 70%.*

Table 2: For each criterion, we compare the "All" configuration for that criterion with the configuration for that criterion and line coverage. Averaged effect sizes are reported with p-values of the statistical tests of symmetry around 0.5.

Criterion	All	Just Line & Criterion	Avg. $\hat{A}_{12}$	p-value
Line	0.78	0.78	0.47	≤ 0.001
Branch	0.75	0.77	0.47	≤ 0.001
Direct Branch	0.75	0.76	0.47	≤ 0.001
Exception	1.35	1.93	0.43	≤ 0.001
Method	0.87	0.87	0.50	0.015
Top Method	0.90	0.89	0.50	0.025
Method No Exc.	0.88	0.88	0.51	≤ 0.001
Mutation	0.75	0.76	0.46	≤ 0.001
Output	0.64	0.64	0.51	≤ 0.001

RQ3: Does combining multiple coverage criteria lead to worse performance of the constituent criteria? When combining different criteria together, the test generation becomes more complicated. Given the same amount of time, it could even happen that for some criteria we would get lower coverage compared to just targeting those criteria in isolation. For example, the class `Auswahlfeld` in the SF110 project `nutzenportfolio` consists of 29 methods, each consisting of only a single line. There are only 15 mutants, and when optimising for line coverage and weak mutation all mutations are easily covered within two minutes. However, when using all criteria, then the number of additional test goals based on the many methods (many of which return primitive types) means that on average after two minutes of test generation only seven mutations are covered.

On the other hand, it is conceivable that coverage criteria can "help each other", in the sense that they might smooth the search landscape. For example, the `NewPassEventAction` class from the `jhandballmoves` project in SF110 has two complex methods with nested branches, and the `if` statements have complex expressions with up to four conditions. When optimising method calls without exceptions, after two minutes the constructor is the only method covered without exceptions, as the search problem is a needle-in-the-haystack type search problem. However, if optimising for all criteria, then branch coverage helps reaching test cases where both methods are called without exceptions.

Table 2 shows the comparison of the "All" configuration on each criterion with the configuration that optimises line coverage and each particular criterion. For each class, we calculated the Vargha-Delaney $\hat{A}_{12}$ effect size [3]. For each configuration comparison, we calculated the average $\hat{A}_{12}$ and ran a Wilcoxon-Mann-Whitney symmetry test on 0.5, to see if a configuration leads to better or worse results on a statistically higher number of classes.

There is strong statistical difference in all the comparisons except Method Coverage and Top-level Method Coverage, which seem to consist of methods that are either trivially covered by all criteria, or never covered. For No-exception Top-level Method Coverage and Output Coverage there is a small increase in coverage; this is likely because these criteria provide little guidance and benefit
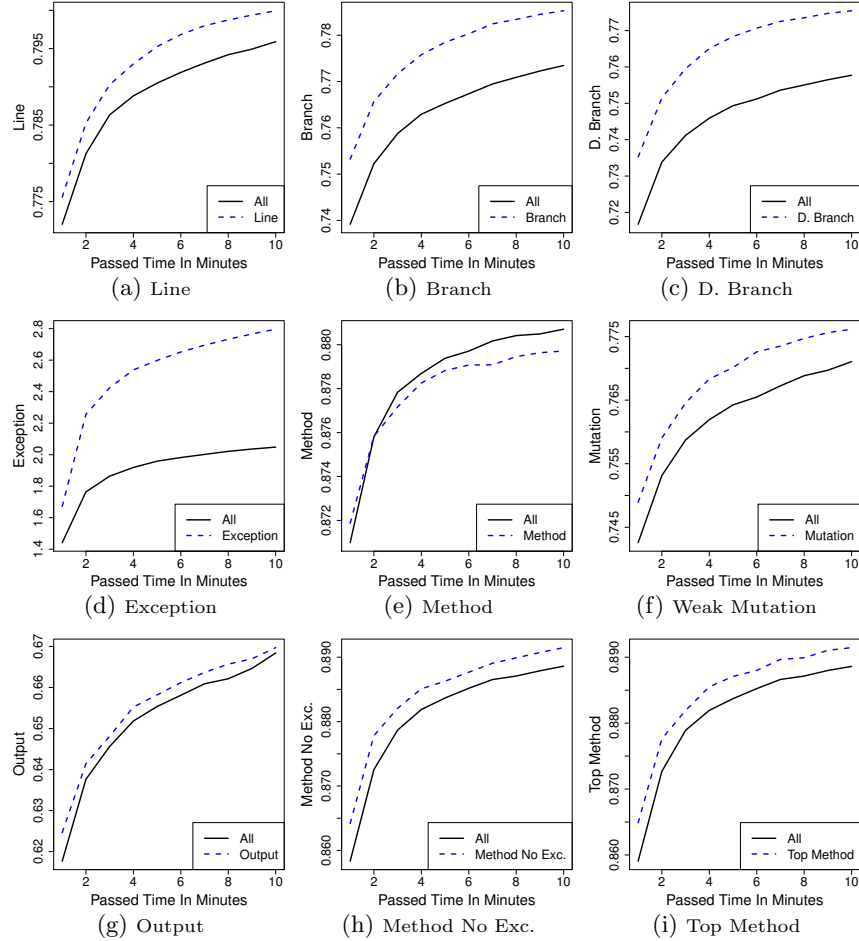
Fig. 3: Time analysis, per minute, for each criterion for the "All" configuration compared with just optimising Line Coverage together with each of those criteria, one at a time.

from the combination with criteria with better guidance. For Exception Coverage targeting all criteria decreases the average number of exceptions substantially from 1.35 to 1.93, which may be caused by the search focusing more on valid executions related to branches and mutants, whereas without that the search becomes more random. For all other criteria there is a decrease in coverage, although very small ($\leq 2\%$).

> **RQ3:** *Combining multiple criteria leads to a 0.4% coverage decrease on average; criteria with coarse fitness functions can benefit more from the combination than criteria with finer grained guidance.*

RQ4: How does coverage vary with increasing search budget? Figure 3 compares the performance of the "All" configuration with the ones of Line Coverage combined with each further criterion. Performance is measured with different coverage criteria in each subplot based on the type of comparison. For example, Branch Coverage is used as performance metric when "All" is compared with "Line & Branch" configuration, whereas Method Coverage is used as performance metric when "All" is compared with "Line & Method". Performance is reported through time, from one minute to ten. The vertical y axes are scaled between the minimum and maximum value each metric obtained.

Given enough time, the performance between each compared configuration should converge to the same value. In other words, given enough time, one could expect that the performance of "All" in each metric would become maximised and equal to just generating data for that criterion alone. Figure 3 shows that for the majority of criteria the performance of the "All" configuration remains slightly below the more focused search, and for Exception Coverage the more focused search even improves over time. For Output Coverage both configurations seem to converge around ten minutes and for Method Coverage the "All" configuration even takes a small lead. Overall, these results suggest that 10 minutes might not be a long enough time interval to see convergence for all criteria; possibly there might also be side-effects between the combination of criteria in the "All" configuration that generate fitness plateaus in the search landscape. Another possible conjecture is that, because the search in EvoSuite minimises size as a secondary objective, over time the amount of exploration in the search space will be reduced, making it more difficult to hit additional targets that are not closely related to what is already covered. This could in principle be overcome by keeping an archive of already covered goals and matching tests, and letting the fitness function focus on uncovered goals.

> **RQ4:** *The influence of combining criteria is not limited to early phases of the search but persists over longer time, and the combination does not catch up with focused search within ten minutes.*

Threats to Validity. To counter internal validity, we have carefully tested our framework, and we repeated each experiment several times and followed rigorous statistical procedures in the analysis. To cope with possible threats to external validity, the SF110 corpus was employed as case study, which is a collection of 100 Java projects randomly selected from SourceForge and the top 10 most popular projects [6]. We used only EvoSuite for experiments and did not compare with other tools; however, at least in terms of the generated tests EvoSuite is similar to other unit test generation tools. Threats to construct validity might result from our focus on coverage; for example, this does not take into account how difficult it will be to manually evaluate the test cases for writing assert statements (i.e., checking the correctness of the outputs).

4   Related Work

Coverage criteria are well established to estimate the quality of test sets [18], and combinations of criteria have been considered in the context of regression

testing [15]. For example, using multiple criteria can improve the fault detection ability after minimisation [11], and Yoo and Harman [16,17] combined coverage criteria with non-functional aspects such as execution time during minimisation. Non-functional aspects have also been considered during test generation; for example, Harman et al. [9] generated tests optimised for branch coverage and memory consumption. In contrast to this approach, we combine different non-conflicting functional criteria, and thus do not require specialised multi-objective optimisation algorithms. In fact, some of the criteria previously implemented in EvoSuite were already combinations of constituent criteria included in this paper. For example, the default branch coverage configuration [5] in EvoSuite combines method and branch coverage. Mutation coverage [7] combines branch coverage with the infection distances used in this paper.

5   Conclusions

Although structural coverage criteria are well established in order to evaluate existing test cases, they may be less suitable in order to guide test generation. As with any optimisation problem, an imprecise formulation of the optimisation goal will lead to unexpected results: For example, although it is generally desirable that a reasonable test suite covers all statements of a Class Under Test (CUT), the reverse may not hold — not every test suite that executes all statements is reasonable. Indeed the desirable properties of a test suite are multi-faceted.

In this paper, we have tried to identify standard criteria used in practice as well as functional aspects that are not captured by standard structural coverage criteria, but are still common practice in object oriented unit testing. We have implemented a search-based approach to generate test suites optimised for combinations of these criteria. Experiments with a sample of open source Java classes have shown that such a combination does neither mean that the test suite sizes become unreasonable, nor that the test generation performance suffers. In fact some aspects can even benefit from the combination, for example when search guidance in the case of search-based test generation is only coarse. An important question that remains to be answered in future work is which selection of criteria matches the expectations of practitioners; for this, we plan to perform controlled experiments with real programmers.

Besides the criteria used in our experiments, the same approach could also be applied in order to enhance test generation with other structural criteria, such as dataflow criteria. On the other hand, there are also non-functional properties of unit test suites that test generation will have to consider in future research, such as the readability of the generated unit tests. However, unlike combinations of functional criteria the inclusion of non-functional aspects may require dedicated multi-objective optimisation algorithms, as functional and non-functional goals may be conflicting (e.g., coverage vs. size).

References

1. Alshahwan, N., Harman, M.: Coverage and fault detection of the output-uniqueness test selection criteria. In: Proc. of ISSTA'14. pp. 181–192. ACM (2014)
2. Arcuri, A.: It really does matter how you normalize the branch distance in search-based software testing. Softw. Test. Verif. Reliab. 23(2), 119–147 (2013)
3. Arcuri, A., Briand, L.: A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. Softw. Test. Verif. Reliab. 24(3), 219–250 (2014)
4. Fraser, G., Arcuri, A.: EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In: Proc. of FSE'11. pp. 416–419. ACM (2011)
5. Fraser, G., Arcuri, A.: Whole Test Suite Generation. IEEE Trans. Softw. Eng. 39(2), 276–291 (2013)
6. Fraser, G., Arcuri, A.: A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite. ACM Trans. Softw. Eng. Methodol. 24(2), 8:1–8:42 (2014)
7. Fraser, G., Arcuri, A.: Achieving Scalable Mutation-based Generation of Whole Test Suites. Empirical Softw. Eng. 20(3), 1–30 (2014)
8. Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated white-box test generation really help software testers? In: Proc. of ISSTA'13. pp. 291–301. ACM (2013)
9. Harman, M., Lakhotia, K., McMinn, P.: A multi-objective approach to search-based test data generation. In: Proc. of GECCO'07. pp. 1098–1105. ACM (2007)
10. Jamrozik, K., Fraser, G., Tillman, N., De Halleux, J.: Generating test suites with augmented dynamic symbolic execution. In: Proc. of TAP'13, pp. 152–167. LNCS 7942, Springer (2013)
11. Jeffrey, D., Gupta, N.: Improving Fault Detection Capability by Selectively Retaining Test Cases During Test Suite Reduction. IEEE Trans. Softw. Eng. 33(2), 108–123 (2007)
12. Korel, B.: Automated software test data generation. IEEE Trans. Softw. Eng. 16(8), 870–879 (1990)
13. Li, N., Meng, X., Offutt, J., Deng, L.: Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report). In: Proc. of ISSRE'13. pp. 380–389. IEEE (2013)
14. McMinn, P.: Search-based software test data generation: A survey. Softw. Test. Verif. Reliab. 14(2), 105–156 (2004)
15. Sampath, S., Bryce, R., Memon, A.: A Uniform Representation of Hybrid Criteria for Regression Testing. IEEE Trans. Softw. Eng. 39(10), 1326–1344 (2013)
16. Yoo, S., Harman, M.: Pareto efficient multi-objective test case selection. In: Proc. of ISSTA'07. pp. 140–150. ACM (2007)
17. Yoo, S., Harman, M.: Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. Journal of Systems and Software 83(4), 689 – 701 (2010)
18. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. 29(4), 366–427 (1997)