Encoding the Certainty of Boolean Variables to Improve the Guidance for Search-Based Test Generation

Sebastian Vogl
University of Passau
Passau, Germany

Sebastian Schweikl
University of Passau
Passau, Germany

Gordon Fraser
University of Passau
Passau, Germany

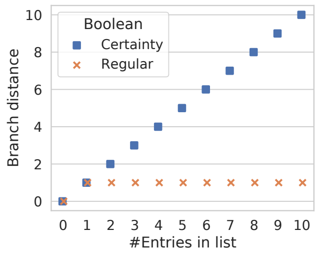
ABSTRACT

Search-based test generation commonly uses fitness functions based on branch distances, i.e., estimations of how close conditional statements in a program are to evaluating to true or to false. When conditional statements depend on Boolean variables or Boolean-valued methods, the branch distance metric is unable to provide any guidance to the search, causing challenging plateaus in the fitness landscape. A commonly proposed solution is to apply testability transformations, which transform the program in a way that avoids conditional statements from depending on Boolean values. In this paper we introduce the concept of *Certainty Booleans*, which encode how certain a `true` or `false` Boolean value is. Using these Certainty Booleans, a basic testability transformation allows to restore gradients in the fitness landscape for Boolean branches, even when Boolean values are the result of complex interprocedural calculations. Evaluation on a set of complex Java classes and the EvoSuite test generator shows that this testability transformation substantially alters the fitness landscape for Boolean branches, and the altered fitness landscape leads to performance improvements. However, Boolean branches turn out to be much rarer than anticipated, such that the overall effects on code coverage are minimal.

ACM Reference Format:
Sebastian Vogl, Sebastian Schweikl, and Gordon Fraser. 2021. Encoding the Certainty of Boolean Variables to Improve the Guidance for Search-Based Test Generation . In *2021 Genetic and Evolutionary Computation Conference (GECCO '21), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3449639.3459339

1 INTRODUCTION

The importance of software tests is widely recognized in both academia and industry. As writing good tests with high coverage and high potential to expose faults can be a time consuming and daunting task, the research field of automatic test generation is concerned with increasing the quality and applicability of the generated software tests. Many state of the art test generation tools such as EvoSuite [10] or Pynguin [18] employ *search-based software testing (SBST)* techniques. SBST uses meta-heuristic search algorithms to sample a search space in order to find candidate test suites that optimize a given set of search criteria, which are encoded as *fitness functions*. These are responsible for ranking candidates and thus greatly impact the effectiveness of the search algorithm. In SBST, the fitness functions are often based on the *branch distance* [15] metric which estimates how close the constraints that lead to the execution of a branch in the control flow are to being satisfied. For example, in Figure 1b the distance to the condition `size == 0` evaluating to true can be calculated as |`size`−0|.



(a) Branch distance to the `true` branch.

```
class MyList{
  boolean isEmpty() {
    return size == 0;
  }
}

void testMe(MyList list) {
  if (list.isEmpty()) {
    // Target
  }
}
```

(b) Code containing problematic branching.

Figure 1: Comparison between regular and Certainty Booleans as return type regarding the branch distance metric.

A challenge for test generation is the possible loss of information during runtime [8], which negatively affects the effectiveness of the branch distance metric. Branching conditions are often made on Boolean variables or returned values, and even though rich information may be available when calculating the Boolean value, the only information present at the branching point is whether the value is `true` or `false`. For example, the if-condition in function `testMe` in Figure 1b depends on the `list` parameter being empty. The call to `isEmpty` returns a regular Boolean, and the resulting branch distance can only encode whether the Boolean is `true` (branch distance 0) or `false` (branch distance 1), which results in a flat fitness landscape (c.f. the crosses in Figure 1a) and thus no guidance for the test generator. Testability transformation [11], i.e., the transformation of code to improve the effectiveness of automated test generation, has been proposed as a remedy, but is mostly concerned with intra-procedural information loss, and is not yet commonly integrated in search-based test generation tools.

In this paper we introduce a testability transformation using the notion of *Certainty Booleans*, which represent not only a truth value but also the certainty of that truth value, in order to prevent the loss of relevant information for the test generation. Certainty Booleans are encoded as integers, and the testability transformation thus replaces the `booleans` in a Java class with `ints` while corresponding Boolean predicates are transformed to integer comparisons on the certainty values. This enables the branch distance metric to capture estimates even for Boolean branches. For example, the Certainty

GECCO '21, July 10–14, 2021, Lille, France
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8350-9/21/07.
https://doi.org/10.1145/3449639.3459339

Boolean returned by a transformed version of the function `isEmpty` in Figure 1b would encode the branch distance of the comparison between `size` and `0`, and as a result the branch distance at the target branch in `testMe` produces a gradient based on that certainty as shown by the squares in Figure 1a. This gradient can guide the test generator to producing a test that passes an empty datastructure as parameter `list`, thus covering the target branch.

We have implemented this testability transformation for Certainty Booleans on Java bytecode, integrated it into the EvoSuite test generation tool, and use it to study the effects empirically on a set of complex open-source classes in terms of the resulting coverage and effects on the fitness landscape. We observe clear effects on the fitness landscape of transformed branches, which in turn lead to increased success rates for these branches. However, in the grand scheme of things, the overall effects on the coverage achieved by EvoSuite are surprisingly small, indicating that Boolean branches are only one puzzle piece in the complex fitness landscape of object oriented test generation.

2 TESTABILITY TRANSFORMATION WITH CERTAINTY BOOLEANS

While a Boolean only contains dichotomous information about its value (i.e., `true` or `false`), a *Certainty Boolean* additionally encodes information about the conditions that led to the assignment of the specific value or could have led to an alternative assignment. This information can be used in order to observe whether different executions leading to the same Boolean value differ in terms of the underlying conditions, which in turn allows us to guide the search towards changing the value. Since it captures the distance towards changing the value, we refer to this distance as the *certainty* of the variable. As the calculation of certainties is based on the branch distance metric, the certainty of integer comparisons is easier to infer than the certainty of complex type comparisons.

2.1 Certainty Booleans

We define Certainty Booleans as a finite set $\mathbb{C} \subset \mathbb{Z}$ of integers that can be partitioned into two non-empty sets $\mathbb{C}^+$ and $\mathbb{C}_0^-$ containing all values representing `true` and `false`, respectively. A mapping from a Certainty Boolean to a regular Boolean is given by

$$\mathsf{fromCertainty} : \mathbb{C} \to \mathbb{B}, \quad \mathsf{fromCertainty}(c) = c > 0$$

To use Certainty Booleans instead of Booleans in programs, a Boolean algebra for Certainty Booleans must be defined. Let $\neg_\mathsf{c}$ be the negation, $\wedge_\mathsf{c}$ be the conjunction and $\vee_\mathsf{c}$ be the disjunction, and $\top_\mathsf{c}$ and $\perp_\mathsf{c}$ be the neutral elements, then fromCertainty must be a homomorphism from Certainty Booleans to regular Booleans. This ensures that the semantics of the target program are not affected when replacing Booleans with Certainty Booleans:

$$\begin{aligned}
\mathsf{fromCertainty}(\perp_\mathsf{c}) &= \perp \\
\mathsf{fromCertainty}(\top_\mathsf{c}) &= \top \\
\mathsf{fromCertainty}(\neg_\mathsf{c} c) &= \neg\,\mathsf{fromCertainty}(c) \\
\mathsf{fromCertainty}(c_1 \wedge_\mathsf{c} c_2) &= \mathsf{fromCertainty}(c_1) \wedge \mathsf{fromCertainty}(c_2) \\
\mathsf{fromCertainty}(c_1 \vee_\mathsf{c} c_2) &= \mathsf{fromCertainty}(c_1) \vee \mathsf{fromCertainty}(c_2)
\end{aligned}$$

where c_1 and c_2 are Certainty Booleans. Furthermore, we define

$$\begin{aligned}
\neg_\mathsf{c} c &= \begin{cases} 1 - c & \text{if } c \le 0 \\ -(c-1) & \text{if } c > 0 \end{cases} \\
c_1 \wedge_\mathsf{c} c_2 &= \min(c_1, c_2) \\
c_1 \vee_\mathsf{c} c_2 &= \max(c_1, c_2)
\end{aligned} \tag{1}$$

The neutral elements of $\wedge_\mathsf{c}$ and $\vee_\mathsf{c}$ are relevant for the transformation. Since we take the maximum and minimum in our definition, the neutral elements are $\top_\mathsf{c} = \max(\mathbb{C})$ for $\wedge_\mathsf{c}$ and $\perp_\mathsf{c} = \min(\mathbb{C})$ for $\vee_\mathsf{c}$. It should be noted that $\mathbb{C}$ is not closed under arithmetic operations. For this reason, we require a case distinction for $\neg_\mathsf{c}$ in Equation 1 to ensure that the subtraction is well-defined.

2.2 Calculating Certainty

The value of a regular Boolean variable is only the truth value that was directly assigned to it. While the same holds for the truth-value of a Certainty Boolean c represented by fromCertainty(c), the certainty itself may change throughout a program execution. Two types of events influence the certainty: First, when a value is assigned to c then the certainty represents how certain this assignment was to happen, which encodes how close the execution was to *missing* the assignment. Between the assignment and uses of the variable later in the program the certainty may change depending on how close the execution was to altering the truth value. Overall, the certainty of c thus represents how close the program execution was to leading to an alternative Boolean value.

The certainty of a particular execution path to an instruction a being taken is captured by the branch distances of the control dependent branching statements of a along that path; if one of the control dependent branches changes its outcome, then a is not executed. We therefore consider all (acyclic) execution paths on the control dependence graph (CDG [13]), and define the certainty of an execution path p as the conjunction of its underlying conditions:

$$\mathsf{certainty}_\mathsf{path}(p) = \bigwedge_{\mathrm{cond}_i \in p}{}_\mathsf{c}\, \mathsf{certainty}_\mathsf{cond}(\mathrm{cond}_i) \tag{2}$$

Here, $\mathrm{cond}_1, \ldots, \mathrm{cond}_m$ are the control-dependent branching conditions present on path p, such that an execution in which these are satisfied will follow p (e.g., if the path follows the else-branch of an if-condition, then cond_i will be the negation of the conditional expression). The function $\mathsf{certainty}_\mathsf{cond}(\mathrm{cond}_i)$ returns the certainty for the condition cond_i evaluating to the desired outcome that follows the execution path, which is the branch distance [15] for that condition evaluating to true.

Given a statement a that assigns a constant value to c, the certainty of this assignment depends on a being reached. A point in the program can be reached by multiple paths. Let $\mathrm{paths}(a)$ denote the set of paths reaching a, then the certainty of reaching a is the disjunction of the certainties that any of these paths is executed:

$$\mathsf{reach}_\mathsf{c}(a) = \bigvee_{\mathrm{path}_i \in \mathrm{paths}(a)}{}_\mathsf{c}\, \mathsf{certainty}_\mathsf{path}(\mathrm{path}_i) \tag{3}$$

Between the assignment a of the value v to the variable c and some other point p in the program where the value c is read, the certainty is influenced by how close the variable was to being overwritten. In order to estimate this certainty, we determine the

```
1  boolean allPositive(int[] integers) {
2    boolean b = true;
3    int i = 0;
4    while (i < integers.length) {
5      int integer = integers[j];
6      if (integer <= 0) {
7        b = false; // Overwrite b
8        break;
9      } else i++;  // b is not overwritten
10   }
11   return b;
12 }
13
14 void setMemberVariable(int[] array) {
15   if (allPositive(array)) {
16     this.member = array;
17   } else {
18     throw new IllegalArgumentException("Negative Value");
19   }
20 }
```

Listing 1: The method `allPositive` checks whether all integers in the parameter `integers` are positive and `setMemberVariable` branches depending on the return value of `allPositive`.

set $A(c, a, p)$ of assignments of c between a and the current point p of the execution, with $a, p \notin A(c, a, p)$. We conservatively only consider the subsets $A_\top(c, a, p)$ and $A_\bot(c, a, p)$, which contain all instructions assigning a value in $\mathbb{C}^+$ or $\mathbb{C}_0^-$ to c, respectively.

The certainty $\mathsf{overwrite}_\mathsf{C}(c, v, a, p)$ that c is overwritten between a and p is then the certainty of any condition that would lead to assigning $\neg v$ to c being fulfilled. It is thus defined as:

$$\mathsf{overwrite}_\mathsf{C}(c, v, a, p) = \begin{cases} \bigvee_{\mathsf{C}\, x \in A(c,a,p) \setminus A_\top(c,a,p)} \mathsf{reach}_\mathsf{C}(x) & \text{if } v = \mathtt{true} \\ \bigvee_{\mathsf{C}\, x \in A(c,a,p) \setminus A_\bot(c,a,p)} \mathsf{reach}_\mathsf{C}(x) & \text{otherwise} \end{cases} \quad (4)$$

The certainty of c being v after assigning it at a and not overwriting it until p is then the conjunction of the certainty that a is reached and the certainty that no alternative assignments is executed between a and p:

$$\mathsf{certainty}(c, v, a, p) = \begin{cases} \mathsf{reach}_\mathsf{C}(a) \wedge_\mathsf{C} \neg_\mathsf{C} \mathsf{overwrite}_\mathsf{C}(c, v, a, p) & \text{if } v = \mathtt{true} \\ \neg_\mathsf{C}(\mathsf{reach}_\mathsf{C}(a) \wedge_\mathsf{C} \neg_\mathsf{C} \mathsf{overwrite}_\mathsf{C}(c, v, a, p)) & \text{otherwise} \end{cases} \quad (5)$$

In the case that $p = a$, the result of $\mathsf{overwrite}_\mathsf{C}(c, v, a, p)$ is $\bot_\mathsf{C}$ since the set of assignments between p and a $A(c, a, p)$ will be empty. Consequently, whenever a variable is written, the certainty of the variable is equal to the certainty that the instruction is reached.

2.3 Testability Transformation

While Section 2.2 defined the certainty in terms of all possible executions, in practice we measure the certainty for concrete executions, such that certainty values are used to calculate branch distances for branching conditions depending on Booleans. In order to make programs use Certainty Booleans instead of Booleans, several transformation steps are necessary: First, we update all declarations

```
int allPositive(int[] integers) {
  // certainty of: i >= integers.length
  int certainty_1 = Integer.MIN_VALUE;
  // certainty of: integer <= 0
  int certainty_2 = Integer.MIN_VALUE;
  boolean isChanged = true; // Monitor if update is needed
  int b = Integer.MAX_VALUE; // Initial Assignment
  int i = 0;
  while (true) {
    certainty_1 = intCmpGe(i,integers.length);
    if (i >= integers.length) break;
    int integer = integers[i];
    // If b is not written in previous iteration -> update b
    if (!isChanged)
      b = update(b, certainty_2, false);
    // b will be updated if no write is executed
    isChanged = false;

    certainty_2 = intCmpLe(integer,0);
    if (integer <= 0) { // Begin of dependent update.
      // Overwrite b
      b = neg(lor(land(neg(certainty_1),certainty_2)));
      isChanged = true; // Mark as written
      break;
    } else i++; // b is not overwritten
  }
  // Update if b is not written in last iteration
  if (!isChanged)
    b = update(b, certainty_2, false);
  isChanged = true; // Necessary for nested loops.
  return b;
}
```

Listing 2: `allPositive` after completing the transformation.

and signatures in the program to use Certainty Booleans instead of regular Booleans. Second, we update conditional statements to make use of the certainty value and to store the branch distance values at runtime. Third, we instrument the program with additional information that calculates and updates the certainty values of the Certainty Booleans along the execution path taken.

As a running example, Listing 1 shows the function `allPositive`, where `true` is initially assigned to a Boolean variable in Line 2. Then the value may be overwritten in Line 7, depending on whether the parameter `integers` contains at least one non positive integer. Assume we want to generate tests for the method `setMemberVariable`, which has an `if`-statement that depends on the return value of the function `allPositive`. In order to provide guidance for covering either branch of this `if`, the certainty of `b` must be encoded in the return value of `allPositive`.

2.3.1 Transforming Signatures and Declarations. As the datatype `boolean` is not able to encode values other than `true` or `false`, we alter all variables of type `boolean` to `int`. In Java bytecode, local `boolean` variables are already implemented as `int`s, so no actual transformation is necessary to change the type. However, arrays, fields, method and constructor parameters, as well as method return types are updated. In particular, transformation of method signatures is a prerequisite to enable certainty values to be propagated interprocedurally. Whenever a signature is changed, a method with the original signature that only converts the types and calls the instrumented method is added to ensure the compatibility between instrumented and non-instrumented code. In the function allPositive (Listing 1), our transformation changes the type of the variable b and the return type from boolean to int.

2.3.2 Transforming Branching Conditions. When transforming programs to use Certainty Booleans rather than Booleans, any branches depending on Boolean values can be transformed such that the branching condition is expressed in terms of the certainty value, rather than the Boolean value. This makes it possible to calculate a branch distance on the certainty, and thus provides guidance to search algorithms. Given a Boolean $b \in \mathbb{B}$ and the corresponding Certainty Boolean $c \in \mathbb{C}$, the conditions are transformed as follows:

$$\begin{aligned} b &\mapsto c > 0 \\ \neg b &\mapsto c \leq 0 \\ b_1 = b_2 &\mapsto \max(\min(\neg_{\mathsf{c}} c_1, c_2), \min(c_1, \neg_{\mathsf{c}} c_2)) \leq 0 \\ b_1 \neq b_2 &\mapsto \max(\min(\neg_{\mathsf{c}} c_1, c_2), \min(c_1, \neg_{\mathsf{c}} c_2)) > 0 \end{aligned} \tag{6}$$

At the level of Java bytecode, instructions that may operate on Booleans are IFEQ, IFNE, IF_ICMPEQ and IF_ICMPNE. As Booleans are represented as integers that only have the values 0 and 1 in Java bytecode, we first determine the type of the top of the operand stack (*tos*) using a simple dataflow analysis, and only apply a transformation if *tos* is a Boolean. The instructions IFEQ and IFNE compare the *tos* with 0 and branch depending on the result of the comparison. If *tos* is a Boolean, we replace these instructions according to Equation 6 with IFGT and IFLE. The instructions IF_ICMPEQ and IF_ICMPNE compare the two values on the top of the operand stack for equality. This comparison can be expressed as *XOR* in Boolean algebra. As two Certainty Booleans may represent the same truth value but with different certainty, we transform the equality to a logical expression over the truth values using

$$c_1 \oplus_{\mathsf{c}} c_2 = (\neg_{\mathsf{c}} c_1 \wedge_{\mathsf{c}} c_2) \vee_{\mathsf{c}} (c_1 \wedge_{\mathsf{c}} \neg_{\mathsf{c}} c_2)$$

This preserves the semantics of the program, and a certainty value is derived by replacing the logical operators with the certainty operators in Equation 6.

In addition to these transformations, we further instrument *all* branch conditions such that the certainty of the condition is stored (cached), because this information is important to compute the certainty of a path as described previously. The function $\mathsf{certainty}_{\mathsf{cond}}(c)$ returns the cached value for the condition c. If a condition has not been executed yet, the certainty is $\perp_{\mathsf{c}}$. In the allPositive function in Listing 1, the certainty of the conditions of the while loop (i < integers.length in Line 4) and the if statement (integer <= 0 in Line 6) are cached. In the function setMemberVariable, the condition of the if statement in Line 15 is transformed to allPositive(array) > 0 according to Equation 6.

2.3.3 Transforming Boolean Assignments. Assignments of the constants true and false are transformed to assignments of Certainty Booleans according to Section 2.2. That is, the certainty of a variable at an assignment is the certainty that the assignment is reached via any of the paths leading to it. The instrumentation of branches described in Section 2.3.2 ensures that the value of $\mathsf{certainty}_{\mathsf{cond}}$ is known at runtime for all conditions that have been executed. The path actually taken at runtime will have a certainty that corresponds to true, such that the logical disjunction of the certainties of all relevant paths involved in the certainty calculation will also represent true. If the constant being assigned is false, the certainty is negated according to Equation 1.

In the allPositive example (Listing 1) there are two Boolean constants: The initial assignment of true in Line 2, and the value false in Line 7. The initial assignment of b is only control dependent on the method entry and no other conditions. Therefore, the replacement for the true constant is $\top_{\mathsf{c}}$. When b is overwritten in Line 7, two control dependent conditions must hold: On the one hand, there must be elements remaining in the parameter integers (while-condition in Line 4) and on the other hand, the current integer must be less than zero (if-condition in Line 6). There is only one path from the method entry to the overwrite satisfying these two conditions, and the certainty of the replacement is therefore:

$$\begin{aligned} &\mathsf{certainty}_{\mathsf{cond}}(\mathtt{i < integers.length}) \\ \wedge_{\mathsf{c}}\ &\mathsf{certainty}_{\mathsf{cond}}(\mathtt{integer} \leq \mathtt{0}) \end{aligned}$$

Finally, this certainty value is negated because b was originally assigned the constant value false in Line 7 of Listing 1.

2.3.4 Transforming Dependent Updates. Assume we are testing Listing 1 with arrays that contain only positive numbers. Intuitively, a test case with an array of [2021,31415] is "further away" from making the branching condition in Line 6 false than a test with [3,1]. In order to express this difference in terms of the Certainty Boolean returned by allPositive, we also include instrumentation that updates the certainty for paths that could have changed the truth value of b (cf. Section 2.2); in other words, the certainty should include how close the if-condition in Line 6, which checks if a value is negative, was from evaluating to true.

To achieve this, the testability transformation needs to add instrumentation to update the certainty value according to Equation 4 in cases where the variable is *not* assigned a new value. This instrumentation is placed in branches that are *exclusive* to the branch that contains a variable assignment. A branch b_1 is considered to be *exclusive* to a branch b_2, if and only if there is an instruction i, such that b_1 and b_2 are control dependent on i and the execution of either branch ensures that the other one is not executed until i is executed again (e.g., the *then* and corresponding *else* block of an if-statement are exclusive to each other with i being the if statement). Updates that are control dependent on at least one branching condition and for which at least one exclusive branch that does not write the variable exists, are referred to as *dependent updates*. Executing a dependent update is equal to adding an element to the set of assignments $A(c, a, p)$ in Section 2.2.

One update routine is added before the most common post-dominator [7] of the branching condition and the writing statement, e.g., after the else branch. Another update is added before the branching condition of the dependent update to tackle loops with dependent updates. Based on Section 2.2, the update does not change the truth value of the Certainty Boolean, but the conjunction ensures that the certainty after the update is less or equal than the certainty before the update.

In the allPositive example in Listing 1, the overwrite of b is dependent on the if-condition. The most common post-dominator of the if-condition and the reassignment of b is the end of the loop.

2.3.5 Instrumentation Example. The transformation described in this section is fully automatic and implemented at the level of Java bytecode. Listing 2 shows the complete instrumentation of `allPositive` at the level of the source code:

- The return type and the type of `b` are changed to `int` (Section 2.3.1).
- The variables `certainty_1` and `certainty_2` cache the certainty values for the `while`-loop and `if`-condition, respectively, and these conditions are transformed to calculate the corresponding values (Section 2.3.2).
- The assignments to `b` in Line 7 and 22 are updated to represent the certainty of reaching the corresponding assignments (Section 2.3.3).
- Dependent updates are handled by monitoring when `b` is not assigned a value (encoded in the Boolean `isChanged`), and calling the method `update` to calculate the dependent updates where needed (Section 2.3.4).

The method `update(c,d,v)` takes three parameters: c is the current Certainty Boolean, d is the cached certainty of the condition that would lead to overwriting c being met and v is the value that would be assigned, if the condition would be met. The Boolean value of fromCertainty(d) will always be `false`, because otherwise the assignment would be executed.

$$update(c,d,v) = \begin{cases} c & \text{if fromCertainty}(c) = v \\ c \wedge_{\mathsf{C}} \neg_{\mathsf{C}} d & \text{if } v = \texttt{false} \\ c \vee_{\mathsf{C}} d & \text{otherwise} \end{cases} \tag{7}$$

There are two locations in the code where dependent updates of the certainty of `b` are required: First, an update is added before the branching condition (Lines 14–15). Second, an update is added before the most common post-dominator (Lines 28–29). Each of those updates the certainty of `b` only if `isChanged` is `false`. When entering the dependent update block, `isChanged` is set to `false`. In order to ensure that the certainty is only updated if `b` is not written, `isChanged` is set to `true` (Line 22) after every write in between the branching condition and the post-dominator.

3 EVALUATION

In order to evaluate the effects of the proposed testability transformation, we aim to answer the following research questions:

RQ 1: *Does the testability transformation lead to higher code coverage?*

RQ 2: *How does the testability transformation influence the fitness landscape?*

3.1 Experimental Setup

3.1.1 Dataset. As targets for test generation we used the corpus of 64 Java classes introduced in earlier work [21] for the evaluation of the Multi-Objective Search Algorithm (MOSA). For 7 of these classes the testability transformation seemed to interact with EvoSuite's bytecode instrumentation such that EvoSuite could no longer generate tests. We excluded these classes, leaving a total of 57 classes for experiments. We did not further select classes with many Boolean branches in order to increase the external validity of our experiments. Note that the testability transformation is not only applied to the class under test, but also its transitive dependencies: In order to find the relevant classes for the instrumentation, the *call tree* [26] of the target class is analyzed and every class that might be called is instrumented. Internal classes of Java (package `java.*`) and classes that are part of EvoSuite or the testability transformation itself are excluded.

3.1.2 RQ1. In order to measure the overall improvement in coverage we generated test suites with and without the testability transformation enabled for each class in the dataset. We then compare the branch coverage of the generated test suites. We used EvoSuite with MOSA and branch coverage as target criterion. Since search algorithms are inherently stochastic, every configuration is executed 30 times. The search budget was set to 600 s to ensure the search has enough time to converge and to evaluate the effects of the transformation over time. To avoid conflating effects or interactions, other testability transformations as well as seeding [9] were disabled during the experiments. All other parameters were left at their established defaults [3].

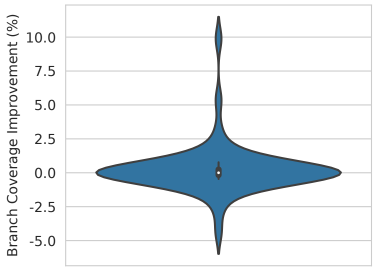
3.1.3 RQ2. In order to investigate how the testability transformation affects the fitness landscapes of individual target branches, we measure the *success rate (SR)* of a branch b as the ratio of number of times the branch was covered in the 30 runs of the RQ1 experiment, and the *success rate improvement* (SRI) as the difference between success rate with testability transformation and without.

The *fitness landscape* describes the topological structure of the search space [1]. The key features of a fitness landscape impacting the performance of a genetic algorithm are *ruggedness* and the *neutrality*. The former describes the number of local optima in the landscape [22], while the latter tells the number of adjacent samples having the same fitness value [23]. In order to measure the impact of the proposed testability transformation on the ruggedness of the fitness landscape, we employ the following four metrics: Autocorrelation (AC), Information Content (IC), Partial Information Content (PIC) and Density-Basin Information (DBI). To measure the impact on the neutrality we use the Neutrality Distance (ND) and Neutrality Volume (NV). All the above metrics are based on measurements of the fitness values for every step in a random walk [1]. The random walks were limited to 1000 steps and 600 s, and repeated 30 times. Branches with a success rate of either 0.0% or 100.0% were excluded since no knowledge can be derived from the measurements. After this, there were 1761 of 11,268 ($\sim$ 15.6%) branches left.

For every metric the branches are grouped by whether the value for the metric has *increased*, *decreased* or stayed *equal*. For this grouping procedure, the p-value was computed according to the Mann-Whitney U-test [19] and the effect size $\hat{A}_{12}$ according to Vargha and Delaney [27] was used to determine whether the metric has increased or decreased.

3.2 Threats to Validity

A threat to the construct validity of our experiments arises from the metrics used to measure the testability. We used branch coverage and success rate, which are commonly used metrics. To counter the threat to internal validity arising from the stochastic nature of search-based approaches, we repeated all experiments 30 times and



(a) Distribution of the branch coverage improvement.

Metric	Value (%)
min	−4.4
1st Quartile	−0.2
median	0.0
3rd Quartile	0.2
max	9.9
mean	0.2
std	1.8

(b) Distribution metrics.

Figure 2: Distribution of difference in class-wise mean branch coverage. Positive values indicate that the testability transformation improved the mean branch coverage.

applied statistical analyses. The parameters of a search-based technique can potentially have an impact on the performance. Therefore, we used established default parameters [3]. We used the Mann-Whitney U-test to test the significance of the results. Additionally, we used the Vargha Delaney effect size [27] to estimate the differences. We tested our approach on 57 real world Java classes from different open source projects. While these classes provide a wide diversity of branches, further experiments with more classes would increase the confidence in our experiments. Future changes of the Java language may require changes in the implementation, but the principle of Certainty Booleans is not limited to one Java version.

3.3 RQ1: Effects on Branch Coverage

Figure 2 shows the distribution of the difference in mean coverage for all classes. For most target classes, the difference is ~0%. The median difference is exactly 0.0%, indicating that the number of classes for which the testability transformation increased the mean coverage is similar to the number of classes for which the testability transformation lead to a decrease in mean coverage. However, some individual outliers, for which the testability transformation increases the guidance, do exist.

Table 1 lists the mean coverage. The p-value is computed with the Mann-Whitney U-test. In total, 15 classes show a statistically significant difference, but out of these only 8 show an increase in coverage, whereas in the other cases the coverage is decreased.

The highest improvement was measured for `DoubleMetaphone`[1], where the coverage increased from 69% to 79%. The branches for which the success rate was improved the most within this class were inside `private` methods that take a Boolean as parameter; in those private methods branching decisions were made on these Boolean parameters. In all cases, the passed value for the parameter was computed by a call to another `private` method `isSlavoGermanic`. In `isSlavoGermanic` the disjunction of four conditions is returned. In this case, the testability transformation was capable of reducing the loss of information caused by the Booleans and provide information how close the Boolean variable is to flip. Other improved classes only showed a slight increase, e.g., the difference measured for `Conversion` is only one branch.

[1]https://github.com/apache/commons-codec

Table 1: Mean coverage for every class. $\hat{A}_{12} > 0.5$ means the testability transformation ("TT") improved the coverage.

Target Class	No TT	TT	p-value	$\hat{A}_{12}$	Branches: transformed/	unchanged
AbstractSimplex	0.00	0.00	–	0.50	4	54
ArrayUtils	1.00	1.00	**0.0000**	0.23	84	1104
AttributeList	0.87	0.89	**0.0000**	0.82	14	126
BasePeriod	0.96	0.96	0.9887	0.50	8	58
BasicMonthOfYearDateTimeField	0.98	0.98	0.0552	0.38	8	58
BigIntegerMath	0.92	0.92	0.5080	0.55	14	130
BooleanUtils	0.96	0.95	**0.0014**	0.26	56	256
BrentOptimizer	0.07	0.07	–	0.50	20	74
CacheBuilderSpec	0.96	0.96	0.4233	0.56	36	122
CompareToBuilder	0.96	0.95	0.1507	0.40	32	240
Conversion	0.97	0.97	**0.0058**	0.64	4	760
DfpDec	0.06	0.06	0.3337	0.52	26	122
DoubleMetaphone	0.69	0.79	**0.0007**	0.76	248	484
EnglishStemmer	0.62	0.61	**0.0476**	0.35	94	286
Expression	0.86	0.85	**0.0004**	0.26	114	168
FunctionUtils	0.17	0.17	0.6432	0.54	0	94
HashCodeBuilder	0.91	0.91	0.0738	0.62	34	94
HelpFormatter	0.93	0.93	0.9325	0.51	38	112
IntervalsSet	1.00	1.00	–	0.50	26	46
ItalianStemmer	0.56	0.56	0.8930	0.51	104	226
JDOMResult	0.60	0.58	0.1607	0.47	6	26
LimitChronology	0.01	0.01	–	0.50	20	68
LinearMath	0.81	0.83	0.4669	0.56	0	260
MatrixUtils	0.83	0.83	0.7670	0.52	4	146
Monitor	0.10	0.10	–	0.50	94	176
MultivariateNormalMixtureExp.	0.33	0.33	0.9226	0.51	2	60
MutablePeriod	0.97	0.97	–	0.50	0	8
NamespaceStack	0.81	0.81	–	0.50	16	64
Option	1.00	1.00	–	0.50	18	72
Partial	0.54	0.54	0.2814	0.43	12	116
PeriodFormatterBuilder	0.92	0.92	0.0645	0.64	146	644
RandomAccessByteList	0.96	0.96	–	0.50	20	58
SAXOutputter	0.80	0.80	–	0.50	28	64
SchurTransformer	0.82	0.87	0.0818	0.62	12	92
SequencesComparator	0.97	0.97	–	0.50	8	82
SimpleCharStream	0.98	0.98	0.6684	0.53	6	66
SpecialMath	0.87	0.87	**0.0018**	0.72	0	186
StrBuilder	1.00	1.00	0.5838	0.46	24	514
TByteFloatHashMap	0.98	0.97	**0.0066**	0.30	90	238
TByteIntHash	0.67	0.67	0.6999	0.48	2	76
TByteObjectHashMap	0.95	0.96	**0.0067**	0.70	66	196
TCharHash	0.94	0.94	0.9155	0.49	2	56
TDoubleLinkedList	0.93	0.95	**0.0002**	0.77	132	244
TDoubleShortMapDecorator	0.73	0.73	0.6543	0.48	26	50
TFloatCharHash	0.67	0.66	0.2862	0.45	2	76
TFloatDoubleHash	0.66	0.66	0.7582	0.52	2	76
TFloatObjectHashMap	0.93	0.89	0.2857	0.58	66	196
TShortByteMapDecorator	0.73	0.73	0.1003	0.42	26	50
TShortHash	0.94	0.93	**0.0345**	0.38	2	56
TreeBidiMap	0.78	0.82	0.2868	0.58	96	362
TreeList	0.94	0.95	**0.0000**	0.83	30	198
TricubicSplineInterpolatingFun.	0.96	0.94	**0.0071**	0.30	0	80
Utf8	0.90	0.91	0.3337	0.52	0	62
Validate	0.99	0.99	0.1607	0.47	32	118
Verifier	0.87	0.88	**0.0097**	0.69	56	272
XMLElement	0.80	0.80	0.6673	0.47	54	266
XMLOutputter	0.95	0.95	–	0.50	0	18
Overall	0.77	0.78	–	0.51	2064	10006

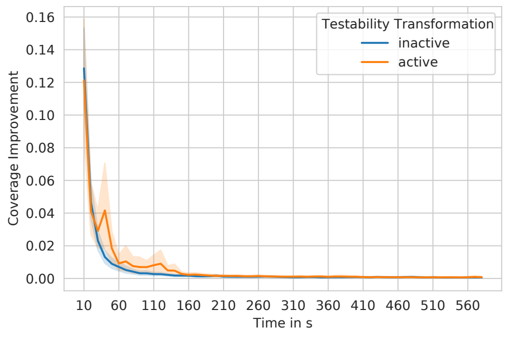


Figure 3: Convergence of experiments with a 95 % confidence interval. The plot shows the improvement of the branch coverage over the previous 10 s of the search.

There are classes for which a significant decrease of the branch distance was measured, e.g., `TricubicSplineInterpolatingFun`. Since no Boolean variables are contained in this class, we anticipated no increase in coverage. The decrease can be explained by the execution overhead introduced by the testability transformation. There are two points where overhead is added, irrespective of whether there are Boolean variables: First, the caching of branching conditions adds additional instructions and (in our implementation) an additional method call. Second, the testability transformation is executed for every class in the transformation scope. For `TricubicSplineInterpolatingFun` the effects can be observed when considering the number of statements executed within the search budget: Without transformation the search executed an average of ∼2.2 million statements, while this reduced to ∼1.6 million (∼72%) after the transformation.

The presence of Boolean branches does not guarantee an increase in coverage: There are classes in the data set, like `ItalianStemmer`, that have a relatively high proportion of Boolean branches, but do not show a significant change in coverage. It was also observed that not all Boolean branches are a challenge to cover. Classes like `IntervalSet` are fully covered even without transformation despite a high proportion of Boolean branches (26/46).

Figure 3 shows that the transformation seems to have effects only at the beginning of the search. After ∼150 s the search generally barely finds solutions to uncovered branches, regardless of whether the transformation is activated.

Overall, these effects are much smaller than we would have expected. It appears that the effect of Boolean branches in practice is not a dominant reason when EvoSuite does not achieve higher coverage; instead, other likely causes may include environmental dependencies (e.g., files), the challenge of creating valid configurations of complex classes, or branches that do not depend on Boolean variables but on reference or `null` comparisons.

Summary *(RQ1)* In our experiments, the testability transformation increased the coverage from 77% to 78% overall, but the majority of classes showed no change in coverage.

Table 2: Mean improvement of the success rate for every metric. Bold values are higher than the mean over all branches (∼ 3.78%).

	increased		equal		decreased	
Metric	#	SRI (%)	#	SRI (%)	#	SRI (%)
AC	263	**5.4**	1349	2.8	149	**9.7**
ND	124	**9.5**	1412	1.8	225	**13.2**
NV	222	**17.0**	1207	2.1	332	1.0
IC	221	**16.9**	1208	2.2	332	0.7
PIC	220	**17.0**	1207	2.2	334	0.7
DBI	213	**8.4**	1351	2.7	197	**6.3**

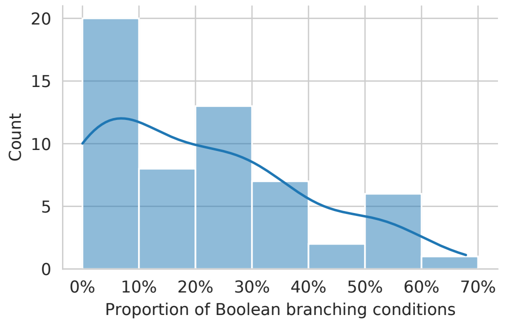


Figure 4: Distribution of the proportion of branches with a Boolean branching condition per class. The bars represent the count of the corresponding bin and the line is the kernel density estimation.

3.4 RQ2: Effects on the Fitness Landscape

RQ1 suggests that the transformation overall only has little effect. This raises the question whether the transformation is actually achieving its goal of transforming the fitness landscape. Figure 4 shows the proportion of branches per class that are actually Boolean branches and can be transformed in the first place; overwhelmingly, classes are dominated by branching conditions that are not based on Boolean comparisons, reducing the possible effects of the transformation. Consequently, the majority of branches in our dataset are either *always* covered in all runs, or *never* covered, regardless of the transformation; only 1761 of 11,268 branches lie in between these two categories. To study the effects on the fitness landscape, RQ2 focuses on these 1761 branches, for which, interestingly, the transformation increases the mean success rate by ∼3.78 %.

Table 2 groups the branches according to the changes in the landscapes caused by the testability transformation. Additionally, the mean success rate improvement for every group is shown. For the majority of branches, the fitness landscape has not significantly changed, which is in line with the number of transformations performed according to Figure 4, as most branching decisions are not based on Booleans. Interestingly, even these non-Boolean branches show a slight increase in the success rate. We conjecture that this is because the *equal* group contains branches where the fitness landscape of their control dependencies provides more guidance.

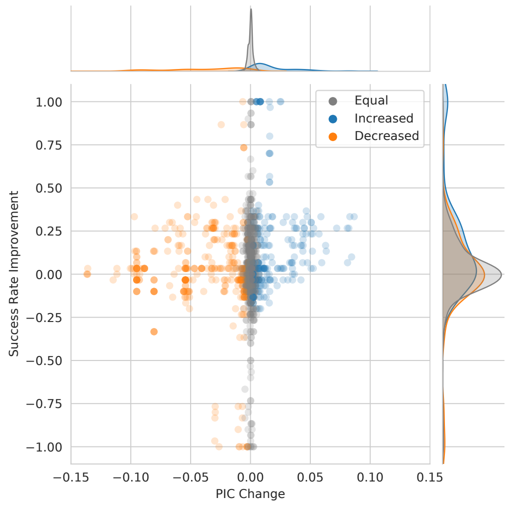


Figure 5: Success rate improvement vs. PIC change introduced by the testability transformation. The marginals of the plot are the kernel density estimations of the groups.

For those cases where the landscape metrics show a change due to the testability transformation, the guidance is overall affected positively. Thus, the fitness landscape without transformation is likely flat, and any alteration of the plateaus, regardless how they affect the metrics, improves the search. Consequently, the testability transformation is capable of improving the guidance.

The largest increase in the success rate can be seen for branches in which the partial information content metric (PIC) has increased. PIC is the ratio of local extrema to the length of the random walk and measures the modality of a landscape [1], and thus reinforces the conjecture that plateaus are removed. Figure 5 plots the success rate improvement against the PIC change introduced by the testability transformation. The *increased* group has nearly exclusively positive SRI, indicating that the testability of the branch has increased. The distribution regarding SRI for the *decreased* group is similar to the one of the *equal* group indicating that decreasing the PIC has not increased the testability of these branches. For the *equal* group no clear patterns are discernible.

Summary *(RQ2)* Transformed branches show a change in the fitness landscape which increases the success rate from 68% to 71%. However, the majority of branches does not depend on Booleans.

4 RELATED WORK

Since the introduction of the concept of testability transformation [12], many different specific transformations have been proposed. A main target of testability transformations are Boolean flags [11], and transformations have been applied also for specific scenarios such nesting [20] or loop-assigned flags [4, 5]. Most of these transformations target procedural code, and flags assigned and used within the same procedure, by substituting the conditions of a flag variable at the branch instruction. In contrast, we target object-oriented code, which usually consists of many small methods in classes that interact with complex call chains, thus requiring inter-procedural transformations.

An inter-procedural transformation was proposed by Li and Fraser [16] at the level of Java bytecode. This transformation aims to replace Boolean values with distance values similar to our transformation. While the use of integers to encode distances is similar to our approach, the transformation rules are different and very intricate, and superseded by our Certainty Boolean transformation. Furthermore, the certainty value is also influenced by how close an execution comes to altering a Boolean value, and an important difference between the two approaches is that we consider all control dependencies of the Boolean flag variable to be replaced, rather than only consider the immediate branch condition. Lin et al. [17] address the problem of interprocedural Boolean flags by first calculating context-sensitive branch distance values for all branches in methods returning Boolean flags, and then aggregating the appropriate distance values for the corresponding call context. While this approach is not based on a program transformation, it has similar effects on the fitness landscape. Similar to our findings, the authors conclude that interprocedural flags are in most cases not the limiting factor for automatic test generation.

5 CONCLUSIONS

Although search-based test generation has been reported to produce test suites of high coverage [6] there are still fundamental limitations [25]. Recent analyses of the fitness landscape in search-based test generation [1, 2] have confirmed that in the domain of test generation for object-oriented programs the search faces particularly challenging fitness landscapes. While there are speculations about the reasons, improvement requires a deeper understanding of which are the factors that influence these landscapes. In this paper, we introduced a testability transformation that addresses one of the challenges posed by the information loss through Boolean flag variables. Our experiments indicate that the transformation successfully alters the fitness landscape when Boolean variables are used, but overall branching conditions depending on Booleans are less influential than expected. We expect similar results for other object-oriented programming languages than `Java`, but further work must be done to generalize our results for other programming languages.

The transformation applied to calculate Certainty Booleans could be generalized beyond the use case of Boolean branches. As next step in our work, we plan to investigate the influence of branches that compare references against `null` (`IFNULL`, `IFNONNULL`) and against other references (`IF_ACMPEQ`, `IF_ACMPNEQ`). Similar to the certainty of a Boolean variable being true we can use our transformation to calculate the certainty of a reference being `null`.

There is also potential to further refine the certainty calculation. For example, dependent updates are currently only performed as part of exclusive branches. However, inspired by the concept of the approach level [28], it might be possible to represent execution paths as sums of normalized branch distances, rather than just the most certain one. Besides refinement of the fitness function, orthogonal approaches to address the problem of challenging fitness landscapes lie in modifying the search operators that induce this landscape [14], or using multiple different fitness functions [24].

ACKNOWLEDGEMENTS

This work is supported by EPSRC project EP/N023978/2.

REFERENCES

[1] Nasser Albunian, Gordon Fraser, and Dirk Sudholt. 2020. Causes and effects of fitness landscapes in unit test generation. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 1204–1212.

[2] Aldeida Aleti, Irene Moser, and Lars Grunske. 2017. Analysing the fitness landscape of search-based software testing problems. *Automated Software Engineering* 24, 3 (2017), 603–621.

[3] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18, 3 (2013), 594–623.

[4] André Baresel, David Binkley, Mark Harman, and Bogdan Korel. 2004. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. *ACM SIGSOFT Software Engineering Notes* 29, 4 (2004), 108–118.

[5] David W Binkley, Mark Harman, and Kiran Lakhotia. 2011. FlagRemover: a testability transformation for transforming loop-assigned flags. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20, 3 (2011), 1–33.

[6] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235.

[7] TY Chen and YY Cheung. 1997. Structural properties of post-dominator trees. In *Proceedings of the Australian Software Engineering Conference (ASWEC)*. IEEE, 158–165.

[8] David Clark and Robert M Hierons. 2012. Squeeziness: An information theoretic measure for avoiding fault masking. *Inform. Process. Lett.* 112, 8-9 (2012), 335–340.

[9] Gordon Fraser and Andrea Arcuri. 2012. The seed is strong: Seeding strategies in search-based software testing. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 121–130.

[10] Gordon Fraser and Andrea Arcuri. 2012. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2012), 276–291.

[11] Mark Harman, André Baresel, David Binkley, Robert Hierons, Lin Hu, Bogdan Korel, Phil McMinn, and Marc Roper. 2008. Testability transformation–program transformation to improve testability. In *Formal methods and testing*. Springer, 320–344.

[12] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability transformation. *IEEE Transactions on Software Engineering* 30, 1 (2004), 3–16.

[13] Mary Jean Harrold, Gregg Rothermel, and Alex Orso. 2005. Representation and analysis of software. *Lecture Notes* (2005).

[14] Yue Jia, Myra B Cohen, Mark Harman, and Justyna Petke. 2015. Learning combinatorial interaction test generation strategies using hyperheuristic search. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 540–550.

[15] Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on Software Engineering* 16, 8 (1990), 870–879.

[16] Yanchuan Li and Gordon Fraser. 2011. Bytecode testability transformation. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 237–251.

[17] Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. 2020. Recovering fitness gradients for interprocedural Boolean flags in search-based testing. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 440–451.

[18] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated Unit Test Generation for Python. *arXiv preprint arXiv:2007.14049* (2020).

[19] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[20] Phil McMinn, David Binkley, and Mark Harman. 2009. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 18, 3 (2009), 1–27.

[21] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[22] Erik Pitzer and Michael Affenzeller. 2012. A comprehensive survey on fitness landscape analysis. In *Recent advances in intelligent engineering systems*. Springer, 161–191.

[23] Christian M Reidys and Peter F Stadler. 2001. Neutrality in fitness landscapes. *Appl. Math. Comput.* 117, 2-3 (2001), 321–350.

[24] Alireza Salahirad, Hussein Almulla, and Gregory Gay. 2019. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Software Testing, Verification and Reliability* 29, 4-5 (2019), e1701.

[25] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 201–211.

[26] Adam M Smith, Joshua Geiger, Gregory M Kapfhammer, and Mary Lou Soffa. 2007. Test suite reduction and prioritization with call trees. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 539–540.

[27] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[28] Joachim Wegener, André Baresel, and Harmen Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Information and Software Technology* 43, 14 (2001), 841–854.