Causal Testing: Understanding Defects' Root Causes

Brittany Johnson
University of Massachusetts Amherst
Amherst, MA, USA
bjohnson@cs.umass.edu

Yuriy Brun
University of Massachusetts Amherst
Amherst, MA, USA
brun@cs.umass.edu

Alexandra Meliou
University of Massachusetts Amherst
Amherst, MA, USA
ameli@cs.umass.edu

ABSTRACT

Understanding the root cause of a defect is critical to isolating and repairing buggy behavior. We present Causal Testing, a new method of root-cause analysis that relies on the theory of counterfactual causality to identify a set of executions that likely hold key causal information necessary to understand and repair buggy behavior. Using the Defects4J benchmark, we find that Causal Testing could be applied to 71% of real-world defects, and for 77% of those, it can help developers identify the root cause of the defect. A controlled experiment with 37 developers shows that Causal Testing improves participants' ability to identify the cause of the defect from 80% of the time with standard testing tools to 86% of the time with Causal Testing. The participants report that Causal Testing provides useful information they cannot get using tools such as JUnit. Holmes, our prototype, open-source Eclipse plugin implementation of Causal Testing, is available at http://holmes.cs.umass.edu/.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Causal Testing, causality, theory of counterfactual causality, software debugging, test fuzzing, automated test generation, Holmes

ACM Reference Format:
Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2020. Causal Testing: Understanding Defects' Root Causes. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380377

1 INTRODUCTION

Debugging and understanding software behavior is an important part of building software systems. To help developers debug, many existing approaches, such as spectrum-based fault localization [21, 41], aim to automatically localize bugs to a specific location in the code [6, 18]. However, finding the relevant line is often not enough to help fix the bug [56]. Instead, developers need help identifying and understanding the root cause of buggy behavior. While techniques such as delta debugging can minimize a failing test input [74] and a set of test-breaking changes [73], they do not help explain *why* the code is faulty [40].

To address this shortcoming of modern debugging tools, this paper presents *Causal Testing*, a novel technique for identifying root causes of failing executions based on the theory of counterfactual causality. Causal Testing takes a manipulationist approach to causal inference [71], modifying and executing tests to observe causal relationships and derive causal claims about the defects' root causes.

Given one or more failing executions, Causal Testing conducts *causal experiments* by modifying the existing tests to produce a small set of executions that differ minimally from the failing ones but do not exhibit the faulty behavior. By observing a behavior and then purposefully changing the input to observe the behavioral changes, Causal Testing infers causal relationships [71]: The change in the input *causes* the behavioral change. Causal Testing looks for two kinds of minimally-different executions, ones whose inputs are similar and ones whose execution paths are similar. When the differences between executions, either in the inputs or in the execution paths, are small, but exhibit different test behavior, these small, causal differences can help developers understand what is causing the faulty behavior.

Consider a developer working on a web-based geo-mapping service (such as Google Maps or MapQuest) receiving a bug report that the directions between "New York, NY, USA" and "900 René Lévesque Blvd. W Montreal, QC, Canada" are wrong. The developer replicates the faulty behavior and hypothesizes potential causes. Maybe the special characters in "René Lévesque" caused a problem. Maybe the first address being a city and the second a specific building caused a mismatch in internal data types. Maybe the route is too long and the service's precomputing of some routes is causing the problem. Maybe construction on the Tappan Zee Bridge along the route has created flawed route information in the database. There are many possible causes to consider. The developer decides to step through the faulty execution, but the shortest path algorithm coupled with precomputed-route caching and many optimizations is complex, and it is not clear how the wrong route is produced. The developer gets lost inside the many libraries and cache calls, and the stack trace quickly becomes unmanageable.

Suppose, instead, a tool had analyzed the bug report's test and presented the developer with the information in Figure 1. The developer would quickly see that the special characters, the first address being a city, the length of the route, and the construction are not the root cause of the problem. Instead, all the failing test cases have one address in the United States and the other in Canada, whereas all the passing test cases have both the starting and ending addresses in the same country. Further, the tool found a passing and a failing input with minimal execution trace differences: the failing execution contains a call to the `metricConvert(pathSoFar)` method but the passing one

```
Failing: New York, NY, USA to
        900 René Lévesque Blvd. W Montreal, QC, Canada
Failing: Boston, MA, USA to
        900 René Lévesque Blvd. W Montreal, QC, Canada
Failing: New York, NY, USA to
        1 Harbour Square, Toronto, ON, Canada
Passing: New York, NY, USA to
        39 Dalton St, Boston, MA, USA
Passing: Toronto, ON, Canada to
        900 René Lévesque Blvd. W Montreal, QC, Canada
Passing: Vancouver, BC, Canada to
        900 René Lévesque Blvd. W Montreal, QC, Canada
-----------------------------------------------------------------------
                 Minimally-different execution traces:
Failing:                                Passing:
[...]                                   [...]
findSubEndPoints(sor6, tar7);           findSubEndPoints(sor6, tar7);
findSubEndPoints(sor7, tar8);           findSubEndPoints(sor7, tar8);
metricConvert(pathSoFar);
findSubEndPoints(sor8, tar9);           findSubEndPoints(sor8, tar9);
[...]                                   [...]
```

Figure 1: Passing and failing tests for a geo-mapping service application, and test execution traces.

does not.[1] Armed with this information, the developer is now better equipped to find and edit code to address the root cause of the bug.

We implement Causal Testing in an open-source, proof-of-concept Eclipse plug-in, Holmes, that works on Java programs and interfaces with JUnit. Holmes is publicly available at http://holmes.cs.umass.edu/. We evaluate Causal Testing in two ways. First, we use Holmes in a controlled experiment. We asked 37 developers to identify the root causes of real-world defects, with and without access to Holmes. We found that developers could identify the root cause 86% of the time when using Holmes, but only 80% of the time without it. Second, we evaluate Causal Testing's applicability to real-world defects by considering defects from real-world programs in the Defects4J benchmark [45]. We found that Causal Testing could be applied to 71% of real-world defects, and that for 77% of those, it could help developers identify the root cause.

A rich body of prior research aims to help developers debug faulty behavior. Earlier-mentioned fault localization techniques [3, 6, 18, 21, 32, 33, 41, 47, 48, 70, 75] rank code locations according to the likelihood that they contain a fault, for example using test cases [41] or static code features [47, 48]. The test-based rankings can be improved, for example, by generating extra tests [6, 75] or by applying statistical causal inference to observational data [7, 8]. Automated test generation can create new tests, which can help discover buggy behavior and debug it [29, 30, 35, 42], and techniques can minimize test suites [38, 54, 68] and individual tests [34, 73, 74] to help deliver the most relevant debugging information to the developer. These techniques can help developers identify *where* the bug is. By contrast, Causal Testing focuses on explaining *why* buggy behavior is taking place. Unlike these prior techniques, Causal Testing generates *pairs of very similar tests* that nonetheless exhibit different behavior. Relatedly, considering tests that exhibit minimally different behavior, BugEx focuses on tests that differ slightly in branching behavior [60] and Darwin generates tests that pass a version of the program without the defect but fail a version with the defect [58]. Unlike these techniques, Causal Testing requires only a single, faulty version of the code, and only a single failing test, and then conducts causal experiments and uses the theory of counterfactual causality to produce minimally-different tests and executions that help developers *understand* the cause of the underlying defect.

The rest of this paper is structured as follows. Section 2 illustrates how Causal Testing can help developers on a real-world defect. Sections 3 and 4 describe Causal Testing and Holmes, respectively. Section 5 evaluates how useful Holmes is in identifying root causes and Section 6 evaluates how applicable Causal Testing is to real-world defects. Section 7 discusses the implications of our findings and limitations and threats to the validity of our work. Finally Section 8 places our work in the context of related research, and Section 9 summarizes our contributions.

2 MOTIVATING EXAMPLE

Consider Amaya, a developer who regularly contributes to open source projects. Amaya codes primarily in Java and regularly uses the Eclipse IDE and JUnit. Amaya is working on addressing a bug report in the Apache Commons Lang project. The report comes with a failing test (see ① in Figure 2).

Figure 2 shows Amaya's IDE as she works on this bug. Amaya runs the test to reproduce the error and JUnit reports that an exception occurred while trying to create the number `0Xfade` (see ② in Figure 2). Amaya looks through the JUnit failure trace, looking for the place the code threw the exception (see ③ Figure 2). Amaya observes that the exception comes from within a `switch` statment, and that there is no case for the `e` at the end of `0Xfade`. To add such a case, Amaya examines the other `switch` cases and realizes that each case is making a different kind of number, e.g., the case for `l` creates either a `long` or `BigInteger`. Since `0Xfade` is 64222, Amaya conjectures that this number fits in an `int`, and creates a new method call to `createInteger()` inside of the case for `e`. Unfortunately, the test still fails.

Using the debugger to step through the test's execution, Amaya sees the `NumberFormatException` thrown on line 545 (see ③ in Figure 2). She sees that there are two other locations the input touches (see ④ and ⑤ in Figure 2) during execution that could be affecting the outcome. She now realizes that the code on lines 497–545, despite being where the exception was thrown, may not be the location of the defect's cause. She is feeling stuck.

But then, Amaya remembers a friend telling her about Holmes, a Causal Testing Eclipse plug-in that helps developers debug. Holmes tells her that the code fails on the input `0Xfade`, but passes on input `0xfade`. The key difference is the lower case `x`. Also, according to the execution trace provided by Holmes, these inputs differ in the execution of line 458 (see ④ in Figure 2). The `if` statement fails to check for the `0X` prefix. Now, armed with the cause of the defect, Amaya turns to the Internet to find out the hexadecimal specification and learns that the test is right, `0X` and `0x` are both valid prefixes for hexadecimal numbers. She augments the `if` statement and the bug is resolved!

Holmes implements Causal Testing, a new technique for helping understand root causes of behavior. Holmes takes a failing test case (or test cases) and perturbs its inputs to generate a pool of

[1]Note that prior work, such as spectrum-based fault localization [21, 41], can identify the differences in the traces of existing tests; the key contribution of the tool we describe here is generating the relevant executions with the goal of minimizing input and execution trace differences.