TerzoN: Human-in-the-Loop Software Testing with a Composite Oracle

MATTHEW C. DAVIS, Carnegie Mellon University, USA
AMY WEI, University of Michigan, USA
BRAD A. MYERS, Carnegie Mellon University, USA
JOSHUA SUNSHINE, Carnegie Mellon University, USA

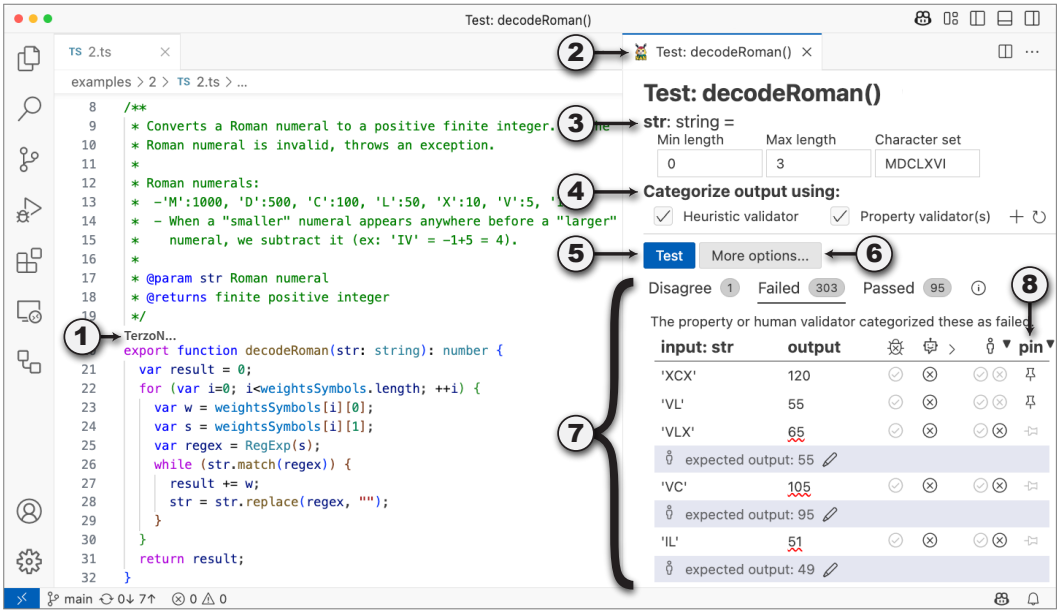

Fig. 1. The TerzoN user interface in the Visual Studio Code IDE provides one-click initial test generation for TypeScript programs. The user may gradually refine the tests in various ways. Key UI elements: (1) Button that starts TerzoN; (2) TerzoN's testing window beside the program under test; (3) Customizable input parameters; (4) Options for categorizing test results; (5) Button that starts testing; (6) Advanced options; (7) Test results organized to display likely bugs more prominently; (8) Button to add test cases to the persistent test suite.

Software testing is difficult, tedious, and may consume 28%–50% of software engineering labor. Automatic test generators aim to ease this burden but have important trade-offs. Fuzzers use an implicit oracle that can detect obviously invalid results, but the oracle problem has no general solution, and an implicit oracle

Authors' addresses: Matthew C. Davis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, mcd2@cs.cmu.edu; Amy Wei, University of Michigan, Ann Arbor, Michigan, USA, weia@umich.edu; Brad A. Myers, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, bam@cs.cmu.edu; Joshua Sunshine, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, sunshine@cs.cmu.edu.

ACM 2994-970X/2025/7-ARTFSE089
https://doi.org/10.1145/3729359

cannot automatically evaluate correctness. Test suite generators like EvoSuite use the program under test as the oracle and therefore cannot evaluate correctness. Property-based testing tools evaluate correctness, but users have difficulty coming up with properties to test and understanding whether their properties are correct. Consequently, practitioners create many test suites manually and often use an example-based oracle to tediously specify correct input and output examples. To help bridge the gaps among various oracle and tool types, we present the Composite Oracle, which organizes various oracle types into a hierarchy and renders a single test result per example execution. To understand the Composite Oracle's practical properties, we built TerzoN, a test suite generator that includes a particular instantiation of the Composite Oracle. TerzoN displays all the test results in an integrated view composed from the results of three types of oracles and finds some types of test assertion inconsistencies that might otherwise lead to misleading test results. We evaluated TerzoN in a randomized controlled trial with 14 professional software engineers with a popular industry tool, fast-check, as the control. Participants using TerzoN elicited 72% more bugs ($p < 0.01$), accurately described more than twice the number of bugs ($p < 0.01$) and tested 16% more quickly ($p < 0.05$) relative to fast-check.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Human-centered computing → User studies**.

Additional Key Words and Phrases: Empirical software engineering, user study, software testing, human subjects, experiments, usable testing, automatic test generation, composite oracle

ACM Reference Format:
Matthew C. Davis, Amy Wei, Brad A. Myers, and Joshua Sunshine. 2025. TerzoN: Human-in-the-Loop Software Testing with a Composite Oracle. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE089 (July 2025), 23 pages. https://doi.org/10.1145/3729359

1 INTRODUCTION

Software testing often intends to prevent bugs from affecting users and operations [33, 47]. However, testing is labor-intensive [2] and may represent 28%–50% of software engineering labor [6, 64]. Many tools aim to ease the burden by automatically generating tests but have important trade-offs.

In testing, an **oracle** determines whether the output of a **program under test (PUT)** is correct. Fuzzers [39] like AFL [67] use an **implicit oracle** that automatically detects likely invalid results such as crashes or **NaN (Not-a-Number)** outputs. But the oracle problem has no general solution [5]; thus, fuzzers cannot automatically evaluate correctness of the PUT. **Automatic Test sUite Generators (ATUGs)** like EvoSuite [23, 24] use the PUT as the oracle and, therefore, also cannot evaluate a PUT's correctness. **Property-based testing (PBT)** tools like Hypothesis [38], fast-check [19], and Quickcheck [11] use a **property-based oracle**, which often requires the user to manually write snippets of code to check the correctness of many inputs and outputs. However, other researchers say that users find it difficult to come up with properties to test and to understand whether the properties they came up with are correct [7, 26, 27, 30, 66]. Further, automatic testing tools have usability problems [3, 7, 27, 36, 45, 50, 53, 61] that may impair user adoption, and many users continue to create test suites manually [3, 20, 24, 35] by tediously selecting example inputs and outputs. Consequently, an important problem is how to provide *effective* and *efficient* automation support for users generating test cases and test suites that evaluate correctness.

To help bridge these important gaps, we present the **Composite Oracle**, which allows a user to, e.g., generate initial test cases using an implicit oracle and then refine test cases by annotating the correctness of interesting execution examples and/or by writing snippets of code to test properties of the PUT. The Composite Oracle may also alert the user to inconsistencies among various correctness assertions, such as property-based and example-based assertions that yield contradictory results.

To understand whether the Composite Oracle might benefit users, we extended our prior tool, NaNofuzz [14], to build **TerzoN**, which organizes implicit, example-based, and property-based oracles into a particular Composite Oracle with a unique **user interface (UI)** that allows the user

to select and see the final and intermediate test results of the Composite Oracle. TerzoN's unique UI was refined over many rounds of rapid prototyping and user tests. Finally, we conducted a randomized controlled human trial [43, 54] with 14 professional software engineers using TerzoN as the intervention treatment and a popular industry PBT tool, fast-check [17, 19], as the control treatment. Participants using TerzoN elicited 72% more bugs ($p < 0.01$), accurately described more than twice the number of bugs ($p < 0.01$) and tested 16% more quickly ($p < 0.05$) relative to fast-check.

Contributions of this paper (C)

(C1) A precise definition and algorithm describing the **Composite Oracle** (Section 2)

(C2) The open-source **TerzoN ATUG**, which surfaces a particular instantiation of the Composite Oracle in its unique user interface (Section 3)

(C3) An **empirical evaluation** that provides evidence that TerzoN's particular Composite Oracle and UI may improve users' ability to efficiently generate effective test cases (Section 4)

This first study of the Composite Oracle provides evidence that TerzoN's particular Composite Oracle and UI helped study participants generate bug-finding tests more efficiently and effectively. Our study design carries the important limitation that apportioning the effect among TerzoN's particular Composite Oracle and UI is not possible. Consequently, further studies would be needed to explore the advantages and limitations of the Composite Oracle in various other contexts.

2 THE COMPOSITE ORACLE

Various types of oracles have trade-offs such that no single type of oracle may be optimal for every situation. The central insight of this paper is that the strengths and weaknesses of various types of oracles may complement one another when composed in a thoughtful way. Below we describe some trade-offs of three types of oracles used in our system and then provide an illustrative example.

An **implicit oracle** detects outputs such as crashes or `NaN`s that may indicate likely bugs. As an implicit oracle only considers the PUT's output, it cannot evaluate the PUT's correctness relative to specific inputs. However, this type of oracle requires minimal effort from a user and can find actual bugs, as evidenced by the lengthy empirical success record of the fuzzing community [39].

An **example-based oracle** detects outputs that differ from a specific example input/output pair previously determined to be correct. Creating a single example test case may be low effort, but the effort required to identify and specify *many* examples to test can be tedious in practice.

A **property-based oracle** detects outputs that violate a specified correctness property for a given PUT. A property-based oracle is particularly powerful in that it can evaluate correctness for a diverse set of execution examples. However, identifying the property to test requires distilling the program's specific behavior into a generalized property, which in practice can be a difficult task for users to do correctly even with significant time and effort [26, 27, 66].

Example combination of three oracle types: Imagine a user is generating tests for the `decodeRoman` function shown in Figure 1, which is supposed to convert roman numerals as strings into standard integers. The user may generate an initial set of tests in a few seconds using an implicit oracle. This initial set of tests may detect some or no bugs. Regardless, the initial set of tests provides a rich set of execution examples from which the user may recognize interesting examples to annotate for correctness using the example-based oracle. Combining these two oracles changes the task from one of *thinking up* interesting tests to one of *recognizing* interesting tests from among examples. Like Nielsen's "recognition over recall" [46], we hypothesize that recognizing a useful test is easier

than thinking one up. Let's say the user notices that `decodeRoman("VLX")` incorrectly returns 65. They first might create a standard unit test asserting the expected result. They next might realize that an important property to test is that `decodeRoman(toRoman(n))=n` for $\forall n \in \mathbb{Z}_{>0}$. In other words, if a natural number is first translated to roman numerals and then back, then the input and return value should be equal. Once the user has identified a property, it may take the user a few tries to correctly write a code snippet to specify it. Therefore, it is important to compare the property-based oracle's results to that of the example-based oracle to find examples where the two oracles disagree, which may indicate errors in either the property's specification or the example-based annotation.

Oracle Precedence: Test results of the example- and property-based oracles are naturally more authoritative than those of the implicit oracle given that they encode aspects of the PUT's expected behavior. Consequently, test results from the example- and property-based oracles may be allocated a higher precedence than those of the implicit oracle such that a passing example-based result may override a failing implicit result. E.g., a PUT may correctly crash or return `NaN` for certain inputs.

Judgment Values: Different oracles might render conflicting judgments. In addition to *pass* and *fail*, a third judgment value, *unknown*, represents cases where a judgment cannot be rendered; e.g., when contradictory judgments cannot be resolved according to the rules of precedence.

Together, we call the above concepts a **Composite Oracle**, which we precisely define below:

Definition: Composite Oracle

Let a **Test Oracle** (O) be a function that evaluates the inputs ($input_{0..m}$) and outputs ($output_{0..n}$) of a PUT execution and produces a **Judgment** (J) indicating whether the execution is valid. A judgment may be one of: *fail*, *unknown*, or *pass*.

$$J \in \mathbb{Z} \tag{1}$$

$$fail = \mathbb{Z}_{<0} \qquad unknown = 0 \qquad pass = \mathbb{Z}_{>0} \tag{2}$$

$$O\big(\{input_1, ..., input_m\}, \{output_1, ..., output_n\}\big) \mapsto J \tag{3}$$

Let a **Judgment Hierarchy** ($JH_{i=0..k}$) be a recursive tuple of k depth such that the tuple's first element is the set of judgments with a higher precedence (i) than those of the second tuple element's judgment hierarchy (JH_{i+1}). JH_k is the empty set, $\emptyset$, at the bottom of the hierarchy.

$$JH_k := \emptyset \qquad JH_{i=0..k-1} := \big\{\{J_{i,1}, .., J_{i,l}\}, JH_{i+1}\big\} \tag{4}$$

Let a **Composite Oracle** (O_C) be a function that produces a single judgment (J') from a Judgment Hierarchy (JH) according to the precedence of judgments in the hierarchy.

$$O_C(JH_i) := \left\{ \begin{array}{lll} unknown & \text{if} & \exists x, y \in \{1, ..., l\} \mid J_{i,x} \in pass \wedge J_{i,y} \in fail \\ \Sigma[J_{i,1}, ..., J_{i,l}] & & \exists x \in [1, l] \mid J_{i,x} \neq unknown \\ pass & & JH_{i+1} = \emptyset \\ O_C(JH_{i+1}) & & \text{otherwise} \end{array} \right\} \mapsto J' \tag{5}$$

O_C returns *unknown* to signal disagreement when judgments of the current precedence level include both *pass* and *fail*. Excluding *unknown*, if there are *pass* xor *fail* Judgments, then return their sum. If we have reached the bottom of the hierarchy ($\emptyset$), return *pass*. Otherwise, recursively process the next-lower level of the Judgment Hierarchy.

The Composite Oracle renders a single judgment from the hierarchy of oracle judgments for a given example execution. For example, an implicit, example-based, and property-based oracle may respectively render *pass*, *fail*, and *fail* judgments for the incorrect example execution, `decodeRoman("VLX")=65`. Given a Judgment Hierarchy where the implicit oracle has the lowest precedence, the Composite Oracle would render a *fail* judgment.

3 THE TERZON AUTOMATIC TEST GENERATOR

To explore the Composite Oracle's properties, we built TerzoN, which surfaces within its UI a particular instantiation of a Composite Oracle with three different oracle types: implicit, example-based, and property-based. Other instantiations of the Composite Oracle are possible. We built TerzoN on top of the NaNofuzz ATUG [14] for a number of reasons. First, representing the Composite Oracle's state in NaNofuzz' graphical UI might be more usable than in the terminal-based interface provided by many testing tools. Second, NaNofuzz supports TypeScript, which we believe is a language that justifies more attention from the testing research community given that a recent IEEE survey [10] reported that TypeScript is a top-5 programming language. Third, Davis et al. [14] showed that NaNofuzz' unique UI of tabs and grids helped users efficiently generate bug-finding test suites using an easy-to-use input generator, implicit oracle, and coherent organization of test results. In the following subsections, we discuss how TerzoN generates test inputs, categorizes test outputs using the Composite Oracle, presents test results, and how the user interacts with TerzoN.

3.1 Generating Inputs

Our study focuses on oracles and test validation and not on input generation. Therefore, TerzoN uses NaNofuzz' input generator, which heuristically derives an input generator from the PUT's formal parameters and types. This generator produces random inputs within a set of ranges or constraints that the user may modify using the UI shown at (3) of Figure 1.

3.2 Categorizing Outputs

TerzoN's particular Composite Oracle may evaluate a single execution example multiple times to obtain judgments from 3 different oracles. Consequently, TerzoN generates an input, calls the PUT, and then provides both the input and the resulting output to the 3 test oracles that need to render judgments. A benefit of TerzoN's design is that users neither need to code multiple input generators nor code multiple calls to the PUT to obtain its output. Below we describe TerzoN's particular Judgment Hierarchy used by O_C to produce a single J' from the 3 different oracle judgments.

Definition: TerzoN's Judgment Hierarchy

TerzoN's Judgment Hierarchy is a particular instantiation of the Composite Oracle and is defined as follows: Let $J_{implicit}$ be the judgment of the implicit oracle, $J_{example}$ be the judgment of the example-based oracle, and $J_{property}$ be the judgment of the property-based oracle,

$$JH_1 := \{\{J_{implicit}\}, \emptyset\} \tag{6}$$

$$JH_0 := \{\{J_{example}, J_{property}\}, JH_1\} \tag{7}$$

Intuition: TerzoN's Judgment Hierarchy gives the example-based and property-based oracles equivalent precedence such that contradictory judgments among these two oracles result in $J' = unknown$. The implicit oracle has a lower precedence; thus, it is only a factor in J' when neither the example-based nor the property-based oracle renders a *pass* or *fail* judgment.

TerzoN includes the particular instantiation of the Composite Oracle described above; however, *other* instantiations are possible, and we describe some of these further possibilities in Section 9. Similar to other property-based testing tools used in industry, TerzoN users manually write snippets of code that specify the properties they would like to test: TerzoN does not automatically infer properties for the user. We describe in more detail this aspect of TerzoN and another tool, fast-check, in Section 4.3.

3.3 Presentation of Test Results

TerzoN approaches the presentation of test results differently from many testing tools, which hide passing test details by default. Our tool additionally departs from the default behavior of PBT tools, which typically stop testing upon encountering the first failing example. The intent behind hiding some details or limiting the number of failures displayed may be to avoid overwhelming the user with too much information and distracting the user from focusing on the failure. However, we are not aware that these widely-used defaults have an empirical basis, and hiding this information might reasonably make it harder for users to understand what inputs were tested, what outputs passed, and what *other* examples may have failed if testing continued. All of these details may be important to a user generating tests. For TerzoN, we decided to follow NaNofuzz' example: by default, TerzoN shows all test results and continues testing regardless of the number of failures. However, the user may change TerzoN's defaults in its Advanced Options panel if desired.

3.4 User Interface

In this section, we refer to the interface elements in Figure 1. Refining TerzoN's UI required multiple rounds of rapid prototyping and user tests, including paper prototypes and the "Wizard of Oz" HCI method [43] in early rounds. Aspects of the Composite Oracle are surfaced in **(4)** and **(7)**.

(1) TerzoN button. TerzoN decorates exported TypeScript functions displayed in the IDE's editor with a button that starts TerzoN in a side window **(2)**.

(2) TerzoN window. The testing window opens "to the side" of the PUT so that the user may simultaneously see the PUT, the test cases, and the test results.

(3) Input parameters. TerzoN analyzes the function signature to determine its inputs, types, and default input ranges. The human-in-the-loop user may adjust these ranges if desired. Inputs are displayed in a TypeScript-like format to minimize the user's cognitive load.

(4) Oracle selection. The user may use the checkboxes to select whether tests should be classified using the implicit (heuristic) and/or property-based oracles. The add (+) button creates a new property validator function where the user may type in code to test a property of the PUT. The list of property validator functions is displayed when hovering over the "Property validator(s)" label, and the user may force a refresh of this list by pressing the refresh button. The example-based (human) oracle is set at the test case level (see Figure 1) and, therefore, is always active.

(5) Test button. When pressed, TerzoN generates test inputs, executes the PUT, and categorizes results using the Composite Oracle. Testing runs until a stop condition defined in **(6)** is met: runtime, number of tests executed, or number of tests failed. When testing ends, results are displayed in **(7)**.

(6) More options. This button toggles the display of advanced options, which are organized into two tabs: reporting and stopping. The reporting tab allows the user to choose whether TerzoN reports all test results or only those that failed. By default, all test results are reported. The stopping tab allows the user to specify when testing should stop, e.g., after a maximum period of time, after a maximum number of tests, or after a maximum number of failed tests. By default, TerzoN runs up to 1,000 tests and for up to 3 seconds (with no limit on the number of failed tests) to ensure the tool provides rapid feedback to maintain the user's attention. Longer testing sessions may be configured.

(7) Results grid. This pane displays a set of relevant category tabs, each containing a grid of test cases and results that the user may sort by clicking on the column heading. Each test is assigned a category according to the Composite Oracle's Judgment (J') as follows: tests with conflicting results (*unknown*), failed tests (*fail*), and passed tests (*pass*). Category tabs containing no results are not displayed. To direct user attention to tests that are more likely to reveal errors, the tabs are shown in the order above. Each row of the grid contains a test case with its input, output, and individual oracle judgments. An oracle judgment is grayed out if a judgment at a higher level of the hierarchy (JH) has precedence, such as in (7) of Figure 1 where for input "`XCX`," the implicit oracle's `pass` judgment is overruled by the property-based oracle's `fail` judgment. The example-based oracle allows the user to mark a test as `passed` by clicking the test case's checkmark icon or as `failed` by clicking the `X` icon. When testing runs again, TerzoN re-checks the example oracle by matching the inputs. We opted to display test cases in table form to minimize the user's reading effort.

(8) Pin button. After TerzoN displays test results, the user may add any of the generated tests to the persistent test suite by pressing the Pin button next to the test result on the results grid **(7)**. The persistent test suite is in Jest [62] format. When the test button **(5)** is pressed again, pinned tests are re-executed with their test results displayed at the top of the grid above newly-generated tests. When a test is un-pinned, its corresponding Jest unit test is removed from the file system.

3.5 Scope and Limitations

TerzoN has a number of important limitations; however, we found it to be a useful vehicle for exploring the practical aspects of a Composite Oracle as well as for evaluating how a testing tool with a Composite Oracle might affect a user's ability to efficiently and effectively generate tests (see Section 4). TerzoN supports exported TypeScript functions that have parameters of types: finite numbers (integers and floats), strings, booleans, literal object types, n-dimensional arrays of any of the previous types, as well as optional and mandatory parameters. TerzoN does not implement test case minimization, its example-based oracle only allows definition of a single expected result, and the tool does not check for non-determinism. As with NaNofuzz, TerzoN's default input ranges are determined using type-based heuristics, and its input generator is much less expressive than those found in industrial PBT tools. In Section 9 we describe plans to broaden support as TerzoN matures.

4 EMPIRICAL EVALUATION

We evaluated TerzoN using a randomized controlled human trial [54]. An overview of the design is shown in Figure 2, and the data and methods are summarized in Table 1. Usability inquiries into software engineering tools are appropriate because software engineers are users, too [43]. ISO's Usability Definitions and Concepts (9241-11:2018) [32] defines **usability** as the "extent to which a system, product, or service can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use." In an industrial setting, Sadowski and Zimmerman [59] similarly emphasized the importance of evaluating software engineering tools along these three dimensions. Therefore, we aimed to answer three variants of the following question:

Research Questions (RQ)

Relative to standard practice, to what extent may TerzoN affect X, where the values of X are:

(RQ1) X = the number of bugs a user accurately identifies

(RQ2) X = the user's confidence in the test activity

(RQ3) X = the user's time on the testing task

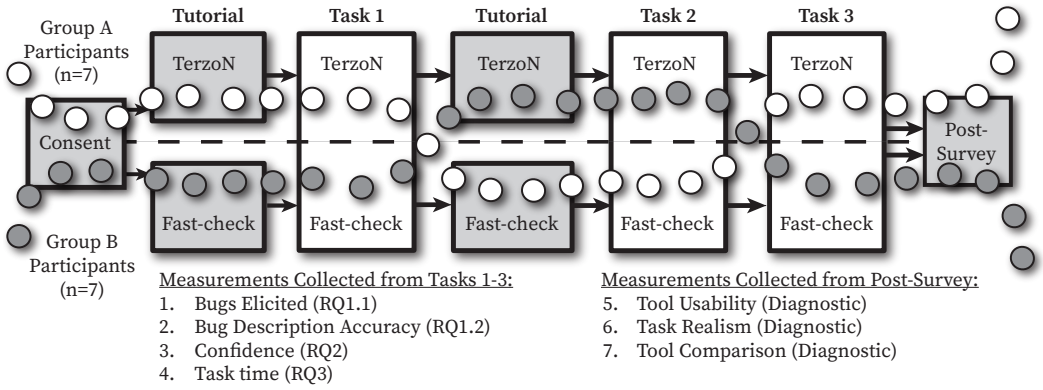

Fig. 2. Visualization of study task sequence (see Section 4). Participants were randomly assigned by pairs into Group A or Group B, which determined the treatment for each task. Each pair's task sequence was also randomized. Shaded tasks (e.g., tutorials) were not time-limited.

Hypothesis: We posited that TerzoN and its particular instantiation of the Composite Oracle would improve users' ability to effectively, confidently, and efficiently test programs.

Our hypothesis seemed reasonable for a number of reasons. First, TerzoN's initial set of tests can direct the user's attention to certain types of likely-incorrect execution examples (e.g., NaNs, crashes) without much user effort. Second, other researchers have said that users have difficulty coming up with properties to test [26, 27, 66], and the Composite Oracle provides users a way to continue refining a set of tests by annotating examples even if a testable property does not come to their minds. Finally, once the user writes code to test a property, the Composite Oracle uses the annotated examples to find executions where the properties pass an execution that should fail—and vice-versa.

4.1 Method

A randomized controlled human trial is recommended as the "gold standard" for evaluating whether users perform better with one tool or another [43, 54]. However, human evaluations in software engineering are rare [34], and recruiting a sufficient number of professional software engineers to achieve statistical power in a human evaluation is even more rare. One approach to reduce the number of participants required for statistical significance is to select a within-subjects design with

Table 1. Measurements, instruments, and methods

ID	Measurement	Instrument	Research Question	Analysis Method(s)
1	Bugs Elicited	Scoring Rubric	RQ1.1	ANOVA, Fisher's exact test* [54]
2	Bug Descr Accuracy	Scoring Rubric	RQ1.2	ANOVA, Fisher's exact test† [54]
3	Confidence	Likert Scale	RQ2	ANOVA, Fisher's exact test† [54]
4	Task Time	Elapsed Time	RQ3	ANOVA, paired t-test* [54]
5	Tool Usability	SUS [9]	Diagnostic	Direct comparison
6	Task Realism	Likert Scale	Diagnostic	Direct comparison
7	Tool Comparison	Free-form Survey	Diagnostic	Thematic Analysis [8]

*=two-tailed; †=Mehta and Patel r×c [40]

repeated measures such that participants use both treatments to complete the same or matched tasks. While such a design would reduce the recruiting burden, a within-subjects repeated-measures design is not suitable for this study due to the task's high learning effect: once a participant discovers a bug, they do not quickly forget it. Consequently, a between-subjects randomized controlled human trial design was more appropriate, despite its offering less statistical power for the same number of participants. In our design, participants used both treatments, but unlike a within-subjects design, they did so on *different* tasks. We could therefore ask participants to rate relative usability and provide comparative feedback about both treatments to provide diagnostic insight into why one treatment might have a measured effect. Since we collected timing data, a think-aloud protocol was not appropriate. Instead, we collected qualitative data in the post-survey. Fatigue and learning effects are important problems to address in such a design, and we controlled for such effects by randomizing the task and treatment sequence. To minimize influence on participants, we did not disclose that we created TerzoN; rather, we characterized both treatments as tools participants might have used previously. Due to practical limits of time and recruiting of professional software engineers [13], our study design evaluated the effects of TerzoN's unique user interface and its particular instantiation of the Composite Oracle at once with the trade-off that apportioning benefits between these two contributions is not possible. We piloted the study with 7 pilot participants in order to refine the tasks and instructions. This study was reviewed by our Institutional Review Board.

4.2 Treatments

TerzoN was the intervention treatment described in Section 3 that builds upon NaNofuzz [14] by implementing a particular instantiation of the Composite Oracle and by surfacing that Composite Oracle in the TerzoN user interface. Consequently, TerzoN users could assert correctness in various ways and find bugs in sophisticated and complex situations.

Fast-check [17, 19] was the control treatment we selected because it is particularly well-suited for the types of tasks in this study. Fast-check is a popular and powerful property-based testing tool for TypeScript with 3.7 million downloads per month as of August, 2024 [18] that is used successfully in industrial-scale software engineering projects such as TypeScript itself, Jest, Google Docs, React, and Jasmine [19]. Similar to TerzoN, fast-check supports property-based based oracles. Consequently, fast-check is an appropriate control treatment. To control for differences that we did not want to measure, we used the Jest Runner [63] Visual Studio Code extension so that participants could run fast-check via a GUI button in the IDE, similar to the way they executed TerzoN. We considered using NaNofuzz [14], the tool on which TerzoN was built, as the control treatment; however, NaNofuzz was very limited such that it only detected a few classes of bugs, such as exception, Not-a-Number (`NaN`), and `null`, which meant that NaNofuzz, critically, was unable to evaluate correctness. Given the foregone conclusion of NaNofuzz' failure to succeed at the more-sophisticated tasks in this study, NaNofuzz was not an appropriate control treatment.

4.3 Tasks

As shown in Figure 2, the experiment included a tutorial for each of the two treatments and three testing tasks. The three tasks varied: (a) the PUT (see Table 2) according to the participant's randomized task sequence and (b) the treatment (see Section 4.2) according to the participant's random group assignment and task sequence number. In each of the tasks, the participant used Visual Studio Code and the treatment to generate test cases and elicit bugs in the program under test. Upon starting the task, we verbally instructed the participant to open the PUT in the IDE and to find allowed program inputs, if any, that caused the program to behave contrary to its specification. We did not reveal the number of bugs in each program and implied that the program

Table 2. Programs under test used in the study tasks

Task Program	Found On	Lines	Error Class(es)	Bugs
`primed`	Rosetta Code	40	incorrect behavior	2
`roman`	Rosetta Code	13	incorrect behavior, out-of-range output	2
`normd`	RunJS	5	incorrect behavior, out-of-range output	2
`lcm`	Rosetta Code	11	incorrect behavior, out-of-range output	2

may or may not contain bugs; however, as shown in Table 2, each program had two bugs. We specified which treatment to use and, if fast-check, directed the participant to open the test file "to the side" so that the tests and the code under test were simultaneously visible. The comment at the top of the program specified the allowed input values, output values, and expected behavior. For each task we provided a working example property test. Once the participant indicated they were ready, the start time was recorded and the task began. Each participant's screen and audio were monitored and recorded via Zoom to ensure use of the intended treatment and PUT. During the task, the participant tested the program using the treatment and generated test cases. At the end of 15 minutes—or when the participant indicated they were done testing—the researcher recorded the stop time and verbally instructed the participant to complete the post-task survey, in which the participant recorded their understanding of the generalized input domains that elicited bugs and recorded their confidence in the testing activity according to a 5-point Likert scale. We provide our task materials in the supplementary materials.

Programs Under Test. Ko et al. suggests that researchers consider using "found" tasks in a tool evaluation to increase the validity of their study's results [34]. We found 11 incorrect TypeScript programs on Stack Overflow, Rosetta Code, RunJS, and LeetCode. We added type annotations where needed to avoid distracting IDE warnings and ensured each PUT included a description of its allowed inputs and expected outputs. User study tasks need to be sufficiently brief so as to be achievable within the time a recruited professional engineer might be available [13]. Consequently, we ran a series of pilots in which students used various combinations of the 11 buggy PUTs, and we eliminated PUTs that the pilot users could not finish testing in 15 minutes regardless of the treatment used. This resulted in the 4 PUTs shown in Table 2.

Task Infrastructure. To avoid some of the difficulties associated with deployment of experimental study software to remote software engineers [13], we hosted the tasks on GitHub Codespaces [25], which provides a web-based Visual Studio Code IDE and Linux virtual machine such that a remote software engineer may edit, run, test, and debug code using a complete IDE inside a web browser.

Tutorials. We designed similar tutorials for TerzoN and fast-check: both had three short exercises and participants could complete each tutorial in roughly ten minutes (see supplemental material).

Writing Property Tests. Fast-check and TerzoN both require the user to write TypeScript code to check whether a property holds across many input and output examples. This delegation to the user is typical of property-based tools; thus, study participants who wanted to check a property of a program in our study needed to write snippets of TypeScript code. As an example, for the `primed` task program that performs prime decomposition of an integer, $x \mid x \in \mathbb{Z}_{\geq 2}$, the correct output is an array of prime numbers, the product of which equals the program's input, x. Listings 1 and 2, below, show two property tests written by users in the study—one for TerzoN and one for fast-check—that pass if all elements of the output array are prime; otherwise, the test fails. Line 1 and the specific variable names used in each listing reflect differences among the two tools. However, as you can see in lines 2–6, both listings encode the property that the test passes if all elements in

the output array are prime. Otherwise, the test fails. What is noteworthy is that the property test code snippets were similar across both tools.

```
export function primeFactorizeValidator1(r: FuzzTestResult): boolean {
  for (let i=0; i<r.out.length; ++i) {
    if(!isPrime(r.out[i])) return false;
  }
  return true;
}
```

Listing 1. TerzoN primed property test written in TypeScript by P09

```
fc.property(fc.integer({min: 2, max:100}), (n) => {
  const arr = primeFactorize(n);
  arr.forEach(element => {
    expect(isPrime(element)).toEqual(true);
  });
}
```

Listing 2. Fast-check primed property test written in TypeScript by P20

4.4 Participants

Finding and recruiting a large and/or representative sample of professional software engineers is difficult [1, 4, 13, 34]. We recruited professional software engineers via LinkedIn, Mastodon, and e-mail—both via direct messages and via public postings that described our study and included a link to the screener survey, which screened for participants: (i) in the United States or Canada (as required by our IRB), (ii) who were over 18, (iii) had at least one year of professional programming experience, and (iv) had programming experience with TypeScript. We offered participants a $30 Amazon gift card and did not offer bonuses. When recruiting on LinkedIn, we selected participants located in the United States and Canada with TypeScript experience and more than one year of professional software engineering experience in their profiles. We asked e-mail and direct message recipients to recruit others they thought might be open to participating, though we did not offer incentives to do so. From August 9 to September 5, 2024, the screener survey collected 168 responses and automatically classified 38 responses as likely being eligible, of which 24 were eligible humans that scheduled sessions. 7 of the 24 were pilot participants, 1 was a pilot no-show, 14 were participants who completed the study, and 2 signed up but did not attend a session before the study period ended.

In the screener survey, potential participants self-reported: gender; professional software engineering experience; hours of coding per week; and experience with property-based testing tools, TypeScript, and VS Code. The screener included timed questions recommended by Danilova [12] to eliminate non-programmers. We intended to recruit a sample of professional TypeScript engineers, and researchers say that adoption of PBT tools is low among this population [7, 26, 27]; therefore, we accepted participants regardless of their level of PBT experience. We excluded participants who lacked TypeScript experience, and the consented participants were capable of writing and reading TypeScript code such that they could complete the study tasks. The final page of the screener allowed the participant to choose an available time slot.

Participants were assigned to groups using matched pair random assignment and a physical coin flip. We classified participants by self-reported professional coding experience: 1–5 years, 6–10 years, and 11+ years. When a participant scheduled a session, we assigned a participant number and checked to see if a previous participant with the same experience level was awaiting a match. If no participant with the same experience level was awaiting a match, we flipped the coin to determine the participant's group and used Google RNG to randomize the participant's task

Table 3. Randomized controlled human trial participants ($n = 14$)

ID	Gender	Group	Professional Experience	Coding Hrs/Week	Self-Reported Expertise with: TypeScript	PBT Tools	VS Code
P09	Female	A	1-5 years	20–30	Beginner	No experience	Expert
P10	Female	A	1-5 years	30–40	Beginner	Intermediate	Advanced
P11	Male	B	1-5 years	20–30	Intermediate	No experience	Expert
P12	Male	A	1-5 years	1–5	Beginner	No experience	Advanced
P13	Female	B	1-5 years	30–40	Advanced	No experience	Advanced
P14	Male	B	1-5 years	1–5	Beginner	No experience	Advanced
P15	Male	B	1-5 years	5–10	Intermediate	No experience	Intermediate
P16	Male	B	1-5 years	>40	Beginner	No experience	Advanced
P17	Male	A	1-5 years	1–5	Intermediate	No experience	Advanced
P19	Male	B	1-5 years	>40	Advanced	Advanced	Expert
P20	Male	A	1-5 years	10–20	Intermediate	No experience	Intermediate
P21	Male	B	11+ years	10–20	Advanced	No experience	Expert
P23	Male	A	1-5 years	10–20	Intermediate	Beginner	Advanced
P24	Male	A	11+ years	>40	Beginner	Advanced	Advanced

P01–P08 represent pilot users. P18 and P22 did not attend a session prior to the close of the study period.

sequence. Then the participant was flagged as needing a match. When the next participant with the same experience level scheduled a time slot, the new participant was matched to the previous one such that one participant would be randomly assigned to group *A* or *B*, the other assigned to the other group, and the paired participants would have the same randomized task sequence.

Table 3 summarizes the participant demographics, which were as follows: 3 participants identified as female, 11 as male, 0 as non-binary, and 0 did not disclose; 2 participants had 10+ years of professional experience, 0 had 6—9 years, and 12 had 1—5 years; 5 participants reported spending 30+ hours coding per week, 5 reported spending 10–29 hours , and 4 reported 1–9 hours.

4.5 Measurements

The experiment included the following measurements, which are summarized in Table 1:

(1) Bugs elicited. Prior to the study, we created an unambiguous rubric that listed the bugs in each task program and the input sets that elicited each bug. The participants' tests were evaluated by the first author against this rubric to determine how many bugs were elicited.

(2) Bug description accuracy. While the prior measure was concerned with whether the tool elicited a bug, this measure was concerned with how accurately the user understood the bug from, e.g., the tool's output. After generating tests, the participant typed in the general inputs that elicited any bugs found. For example, suppose a program throws an exception for the set of inputs, $\mathbb{Z}_{>1}$. According to the rubric, a full score was given for identifying the entire set ($\mathbb{Z}_{>1}$). A half score was given for identifying only a subset (e.g., $\mathbb{Z}_{>2}$) of the rubric's described set. Finally, a half score was deducted if the participant's set included allowed inputs that did not elicit any bug (e.g., $\mathbb{Z}_{<1}$). Each task contained two bugs such that the participant's maximum possible score for each task was 4.

(3) Confidence. At the end of each task, the participant reported their confidence in identifying the inputs that elicited bugs using a 5-point Likert scale.

(4) Task time. At the start of each task, we recorded the begin time manually. When the participant finished creating tests or ran out of time, the researcher recorded the end time.

(5) Tool usability. The System Usability Scale (SUS) is a standard measure of system usability. In the post-survey, participants rated both treatments using the standard SUS questions, which were scored according to the procedure described by Brooke et al. [9]. The order of the treatments in the survey was randomized to counter-balance ordering effects.

(6) Task realism. Participants rated their agreement with the statement, "I thought the testing tasks in this study resemble tasks I might encounter outside the study," using a Likert scale.

(7) Tool comparison. Participants answered free-form comparative survey questions regarding each treatment's: (i) effectiveness in testing, (ii) ease of use, (iii) fit with the participant's test and debug workflow, and (iv) opportunities for improvement.

4.6 Data Analysis

As shown in Table 1, this mixed-methods experiment quantitatively analyzed task data to answer the research questions. We further designed a diagnostic explanatory analysis to help us explore the quantitative results. We discuss the analysis for each measurement from Figure 2 and Table 1 below.

(1) Bugs elicited. We used a randomized matched pair design (see Section 4.4) based on years of professional experience. From 14 total participants, this yielded 7 pairings. Within each pairing, we created two pseudo-participants, one by taking all the data from the pairing using fast-check, and another by taking all the data from the pairing using TerzoN. These pseudo-participants naturally partition into two groups, one exclusively using each treatment. We then performed a two-tailed Fisher's exact test on the categorical data. There were a total of 6 bugs that each of the 14 participants could find, so we first allocated $6 \cdot 14 = 84$ possible bugs, and then within the two groups (fast-check and TerzoN), counted up the number of bugs actually elicited by participants through a test case. For each task and treatment, we performed an ANOVA using the presence of the intervention as a factor and analyzed the effect of other independent variables: years of professional experience, experience with TypeScript, experience with PBT tools, experience with VS Code, and the task sequence.

(2) Bug description accuracy. This analysis is the same as (1), except we used categories corresponding to the recorded accuracy values of 0–4. We counted up the number of accuracy scores in each category for the two groups, as shown in Table 5.

(3) Confidence. This analysis is the same as (1), except that we used categories corresponding to the possible confidence scores 1–5, as shown in Table 5.

(4) Task time. This analysis is the same as (1), except that we used a two-tailed paired t-test due to this measure containing continuous time data rather than categorical data. See Table 5.

(5) Tool usability. We counted the number of participants who gave a higher System Usability Scale [9] score for TerzoN vs. fast-check and divided it by the number of participants. We calculated the mean and standard deviation of the SUS scores for each treatment.

(6) Task realism. We counted the number of participants who responded in the post-survey that the tasks were realistic (vs. neutral, unrealistic) and divided the count by the number of participants.

(7) Tool comparison. We analyzed free-form post-survey data using the inductive thematic analysis procedure described by Braun and Clarke [8], which outlines six phases and emphasizes: (i) the phases are guidelines, not rules; (ii) thematic analysis is "not linear" and movement among phases is expected; and (iii) the process should not be rushed. Going beyond the recommendations of Braun and Clarke [8], the second author established replicability of the first author's result by re-coding all the data for the 14 participants using the first author's code book. We calculated inter-rater reliability using Cohen's Kappa [54], which indicated a very good [52] level of agreement ($\kappa = 0.746$).

4.7 Results

Below we report the results of the analyses described in Section 4.6.

(1) Bugs elicited. Table 4 shows that participants using TerzoN on average elicited 72% more bugs (31/42) than when using fast-check (18/42). The Fisher's exact test shown in Table 5 (Measure 1) indicates the differences in this measure are statistically significant ($p < 0.01$). The ANOVA indicates that experience with PBT was positively correlated with bugs elicited ($p < 0.01$) on the `roman` task.

(2) Bug description accuracy. Table 4 shows that participants using TerzoN described bugs more accurately (48/84) than when using fast-check (20/84). The Fisher's exact test shown in Table 5 (Measure 2) shows the differences are statistically significant ($p < 0.01$). The ANOVA indicates that experience with PBT was positively correlated with bugs elicited ($p < 0.05$) on the `roman` task.

(3) Confidence. Table 4 and Figure 3 show that participants using TerzoN on average reported 39% higher confidence (3.57/5.00) than when using fast-check (2.57/5.00). The Fisher's exact test shown in Table 5 (Measure 3) indicates the measured differences are not statistically significant ($p > 0.1$). The ANOVA indicates that TerzoN was positively correlated with confidence on the `normd` task ($p < 0.05$), as was experience with TypeScript ($p < 0.05$) on the `roman` task.

(4) Task time. Table 4 and Figure 3 show that participants using TerzoN, on average, completed tasks 16% more quickly than with fast-check. The two-tailed paired t-test shown in Table 5 (Measure 4) indicates the differences in this measure are statistically significant ($p < 0.05$). The ANOVA indicates that PBT experience was positively correlated with task time ($p < 0.05$) on the `1cm` task.

(5) Tool usability. Figure 3 shows that participants rated TerzoN (mean=72.68, SD=26.57) slightly better on the System Usability Scale [9] than fast-check (mean=69.64, SD=27.35). 8 participants gave TerzoN a higher SUS score, 2 rated both tools the same, and 4 gave fast-check a higher score.

(6) Task realism. 93% (13/14) of participants indicated the tasks in this study were realistic.

(7) Tool comparison. Our inductive thematic analysis identified seven repeated themes:

- **T1:** Organized test results can help me
- **T2:** Flexibility in specifying oracles can help me
- **T3:** Flexibility in generating inputs can help me
- **T4:** Tools should help me work more efficiently
- **T5:** Both tools were functionally similar
- **T6:** Similarity to a tool I know can help me
- **T7:** Specific tool suggestions, bug reports

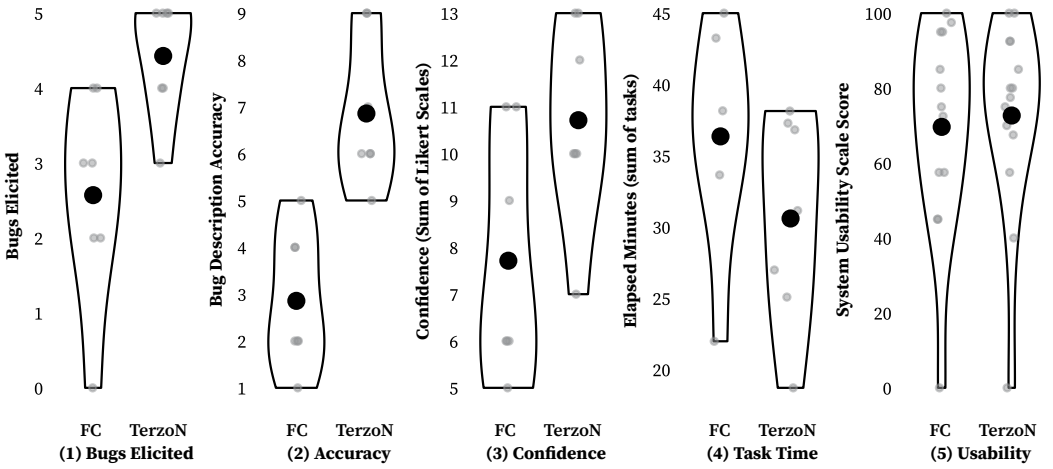

Fig. 3. Violin plots [31] showing the results by treatment for Measures 1-5. Y-axis for Measures 1–4 is the sum of three tasks for paired participants. Y-axis for Measure 5 is the SUS [9] score range. Mean and observations are indicated by black and gray dots, respectively. FC abbreviates fast-check.

Table 4. Quantitative task results summary for measures 1–4 from Table 1 ($n = 14$)

	Tasks (see Table 2)									
	`primed`		`roman`		`normd`		`lcm`		Σ	
Treatment	**Mean**	**SD**	**Mean**	**SD**	**Mean**	**SD**	**Mean**	**SD**	**Mean**	**SD**
Measure 1: Bugs elicited; range 0–1 (1=all bugs elicited)										
Fast-check	0.40	0.37	0.42	0.34	0.50	0.35	0.42	0.19	0.43	0.32
TerzoN	0.80	0.40	0.67	0.24	0.50	0.35	0.92**	0.19	0.74**	0.33
Measure 2: Bug description accuracy; range 0–1 (1=all bugs accurately described)										
Fast-check	0.25	0.22	0.25	0.14	0.25	0.18	0.21	0.09	0.24	0.16
TerzoN	0.70	0.37	0.58	0.31	0.25	0.25	0.67**	0.28	0.57**	0.35
Measure 3: Confidence; range 0–1 (1=extremely confident)										
Fast-check	0.56	0.23	0.43	0.33	0.65*	0.09	0.47	0.32	0.51	0.29
TerzoN	0.80	0.13	0.77	0.14	0.40	0.14	0.80	0.28	0.71	0.24
Measure 4: Elapsed time (in seconds); lower is better										
Fast-check	719	186	777	174	581*	197	781	213	727	207
TerzoN	360**	72	585	228	879	37	672	175	612*	235

Statistically significant differences are marked as: * ($p < 0.05$), ** ($p < 0.01$)

Table 5. Quantitative results for measures 1–4 from Table 1. FC abbreviates fast-check.

Measure 1	**FC**	**TerzoN**	Σ
Bugs Elicited	18	31	49
Bugs Not Elicited	24	11	35
Σ	42	42	84

Measure 2	**FC**	**TerzoN**	Σ
accuracy = 0	5	3	8
accuracy = 1	12	3	15
accuracy = 2	4	6	10
accuracy = 3	0	3	3
accuracy = 4	0	6	6
Σ	21	21	42

Measure 3	**FC**	**TerzoN**	Σ
confidence = 1	8	2	10
confidence = 2	2	2	4
confidence = 3	4	4	8
confidence = 4	5	8	13
confidence = 5	2	5	7
Σ	21	21	42

Measure 4	**FC**	**TerzoN**
P09 / P13 time	34m 39s	25m 05s
P10 / P15 time	45m 00s	38m 08s
P11 / P12 time	43m 15s	27m 59s
P14 / P23 time	36m 09s	37m 17s
P16 / P20 time	36m 18s	37m 49s
P17 / P19 time	22m 59s	19m 43s
P21 / P24 time	38m 09s	31m 09s
mean time	36m 21s	31m 36s

5 DISCUSSION

Our experiment investigated the three research questions introduced in Section 4. We now discuss how the answers to these questions are suggested by our results.

RQ1. Relative to standard practice, to what extent may TerzoN affect the number of bugs a user accurately identifies? TerzoN improved the accuracy of bug identification. Table 4 shows that participants using TerzoN on average elicited 72% more bugs (Measure 1, $p < 0.01$) and accurately described more than twice the number of bugs (Measure 2, $p < 0.01$) than when using fast-check.

RQ2. Relative to standard practice, to what extent may TerzoN affect the user's confidence in the test activity? Table 4 shows that participants using TerzoN reported numerically 39% higher confidence, but this result did not reach statistical significance with 14 participants.

RQ3. Relative to standard practice, to what extent may TerzoN affect the user's time on the testing task? TerzoN sped up testing tasks. Tables 4 and 5 show that participants on average completed tasks 16% faster with TerzoN than with fast-check ($p < 0.05$).

Why did TerzoN show the positive results above? When did participants encounter problems using TerzoN? Our participants compared both treatments in the post-survey, and we structure the remainder of our discussion based on the diagnostic data we collected in Measures 5 and 7.

5.1 TerzoN Generated More Tests and Organized the Test Results Effectively (T1)

Users said that TerzoN provided more test results than fast-check (P12), made test results easy to find and compare (P11, P14, P15, P19, P20), and provided a way to track interesting examples across test runs (P23, P24). For fast-check, users said that they wanted to see the outputs of more test cases beyond that of the counter-example (P13, P14, P20).

> "[TerzoN] was much easier to use. Especially, when I can see a complete list of examples of failing or passed test cases side-by-side" (P19.Q11)

> "[TerzoN's] ability to track / pin counterexamples across runs was a big plus, as it was a more UI-friendly tool than fast-check's command line output." (P23.Q11.1)

5.2 TerzoN Users Tested Effectively Using Multiple Oracles (T2)

Users said that TerzoN's ability to work with different types of oracles facilitated testing. For example: generating tests with the implicit oracle made it easy to start testing (P14, P23), it was easy to annotate the correctness of generated examples (P10, P20), and when manually annotating examples got tedious, the ability to write a property function was helpful (P10, P20).

> "[TerzoN] was helpful in the sense I could write [a property] validator, there was [implicit] aspect, and then I can always validate the results myself." (P10.Q11.1)

P12 stated that it was "easier to write tests" with TerzoN and that "fast-check requires every test to be manually written." For example, P12 elicited both bugs in the `primed` task in 4 minutes, 55 seconds using all three of TerzoN's oracles, including writing a property test. In contrast, the only fast-check user who elicited both bugs in the same task (P20) required 15 minutes to write a property test equivalent to P12's. As shown in Section 4.3, the task of writing property test code was similar for both tools. Consequently, it is possible that TerzoN's Composite Oracle and the coherently-organized test result feedback provided by TerzoN's user interface helped users think of properties to test.

The Disagree tab, which shows results where there is a disagreement between the property- and example-based oracles, appeared 25 times across 7 users (P10, P15, P19, P20, P21, P23, P24). For example, P15 generated initial tests for the `normd` task with the implicit oracle and then marked an example with a NaN output as incorrect. The user then wrote a property test, `return !(output<0)`. But this assertion is incorrect: it passes when `output=NaN` because `NaN<0` evaluates to `false`. When

the user executed the property test, TerzoN compared the property-based judgement *pass* to the example-based judgement *fail* and notified the user of the contradiction using the Disagree tab. Similarly, P24 used the implicit oracle to generate initial tests for the `roman` task and then annotated incorrect examples such as "`IV`" and "`"`". Based on the user-provided examples, TerzoN's user interface alerted the user that the property validator was passing examples that the user thought should *fail*. P24 then elicited and accurately described both bugs in the task in under 10 minutes.

5.3 Generated Inputs May Need to Satisfy Complex Constraints (T3)

P13 and P19 emphasized the need to generate inputs that satisfy complex constraints. While creating complex input generators was not a focus of our study, it is an important area of inquiry.

> "It could also be easier in both tools to specify more complex constraints that the inputs should satisfy." (P13.Q16.2)

5.4 TerzoN Made Common Tasks Efficient (T4)

Users stated that TerzoN was "better for iteration" (P23), "quick" (P15, P20), "faster" (P11), "easy" (P20, P24), required less code (P12), and required less memorization of syntax (P15, P19).

> "[TerzoN] means I have to memorize less syntax to do what I want." (P19.Q13.1)

> "[TerzoN is] GUI based, quick iterations, less code." (P12.Q13)

5.5 Both Tools Were Functionally Similar (T5)

While we point out that TerzoN has some important limitations in Section 3.5, users stated that the basic functionally of the two tools was similar (P10, P11, P13, P14, P17, P19).

> "They both use & support property-based testing & seems to be able to be integrated into any project pretty easily." (P19.Q15)

> "I could see myself getting used to both tools for testing." (P13.Q15)

5.6 Users Who Rated Fast-check as More Usable Still Performed Better With TerzoN (T6)

It is not surprising that some users rated fast-check as more usable than TerzoN given that fast-check is designed to look and function like testing tools that software engineers use daily. However, users were able to work with TerzoN after a brief (mean=10m 44s) tutorial, and twice the number of users (8/14 vs. 4/14) gave TerzoN a higher SUS score than fast-check after using both tools. In the post-survey data, the 4 users (P10, P13, P16, P17) who favored fast-check said they did so at least partially due to its similarity to tools they were used to. However, the 4 users who favored fast-check on average still found more bugs and completed tasks in less time with TerzoN.

> "I am more familiar with the way fast-check works, where you write a test and run it and get results in the terminal." (P13.Q13.1)

> "It is easier and faster for me to use fast-check because of a smaller learning curve. Although, the difference is not much." (P17.Q15)

5.7 TerzoN Needs Further Improvement (T7)

Users made various suggestions about how TerzoN might be improved, e.g., allowing use of TerzoN outside Visual Studio Code (P15), improved typing of the PUT input and output variables within the property validator functions (P14, P23), storing TerzoN's validator functions in a separate file (P16), and automatically grouping and categorizing failing results (P17).

> "[TerzoN] was pretty great, except for the few things such as the arguments i.e., `r.in` and `r.out`. Except that, everything was good." (P14.Q16.2)

6 THREATS TO VALIDITY OF THE STUDY

This study has a number of limitations that might be addressed in future studies.

Internal Validity. To ensure that the study duration fit within a length of time that might be acceptable to professional engineers, we limited the maximum time for each task to fifteen minutes. This limit stopped participants in 17 tasks (6 with TerzoN, 11 with fast-check). Given this limit affected fast-check tasks more often than TerzoN tasks, it is possible that accuracy for fast-check tasks might be higher had no time limit been imposed. As our study design required participants to use both tools, it is possible that effects from this specific combination of tools was not controlled.

External Validity. We adopted Ko et al.'s [34] guidance to use "found" tasks to improve the realism of our tasks and the validity of our results; however, it is possible that the tasks we selected from Rosetta Code and RunJS were neither realistic nor representative. For instance, task-based user studies such as ours must select tasks that are sufficiently brief so as to fit within the time that a professional software engineer is available. While we are not aware of reasons why it may be the case, it is possible that the Composite Oracle might exhibit different properties with programs larger than those used in our study. During the post-survey, we asked participants to what extent the tasks they encountered were realistic, and 93% (13/14) indicated the study tasks were similar to tasks they might encounter outside the study. Our sample of professional software engineers may be dissimilar to the overall population of software engineers in important ways that are not quantified. Due to procedural difficulties with securing approval to recruit participants outside of the United States and Canada, our sample does not include software engineers from other important geographies. The TerzoN prototype used in this study possesses capability necessary to evaluate the hypothesis but lacks some important features that we discuss in Section 3.5. Tools with more functionality may be more difficult to use [42], and a tool with more features may provide a different effect.

Construct Validity. Practical limits of time and recruiting of professional software engineers [13, 34] required us to prioritize the most vital research questions; consequently, our study design evaluated TerzoN's particular user interface and instantiation of the Composite Oracle all at once. However, we cannot apportion the effects among these two contributions. **Effectiveness (RQ1)** investigated the number of bugs a user accurately identified while generating the tests and was measured in two parts. **RQ1.1** measured how many known bugs the user's tests elicited, which is an important quality component: test suites that find actual bugs may be considered more effective. **RQ1.2** measured the quality of the testing activity by assessing the extent to which a user could accurately describe the bugs elicited. Other measures of quality exist that we did not assess, such as code coverage, test suite size, and mutants killed. However, it was not feasible to test *all* measures of quality in study sessions of limited duration. **Confidence (RQ2)** investigated the participant's confidence in the testing activity. Sadowski and Zimmermann [59] explain that "satisfaction may be impacted by the real or perceived effectiveness of a user's personal work." We measured self-reported confidence as a proxy variable for satisfaction, but it is not clear to what extent a user's confidence may relate to overall satisfaction. **Task time** (**RQ3**) investigated the time a user required to test the PUT, which was measured as time elapsed from the beginning to the end of the testing task.

7 RELATED WORK

Randoop [48], NaNofuzz [14], and IntelliTest [41] use an implicit oracle and generate persistent test cases. Unlike Randoop and NaNofuzz, TerzoN can *also* evaluate correctness. IntelliTest runs in an IDE and can evaluate correctness. Unlike IntelliTest, TerzoN uses a Composite Oracle that can identify inconsistent test assertions and was evaluated with humans in a randomized controlled human trial. PBT tools such as Hypothesis [38] and fast-check [19] can evaluate correctness. However,

TerzoN differs by its use of a Composite Oracle that can identify inconsistent test assertions and that *also* incorporates implicit and example-based oracles.

Rothermel et al. [55, 56, 57] performed foundational work in testing spreadsheets and evaluated "What You See Is What You Test" (WYSIWYT), a code-free usable testing interface for end-users. Fisher et al. [21, 22] expanded this work by creating "Help Me Test" (HMT), which used random and search-based techniques. Similar to TerzoN, these tools display tests and results as elements within a graphical user interface, but do so within the context of spreadsheets and not within an IDE.

Fraser et al. [24] evaluated the EvoSuite ATUG with human software engineers and reported that participants found *fewer* bugs with EvoSuite, despite the tool generating test suites that had higher code coverage. EvoSuite used the PUT as the oracle; consequently, the test cases EvoSuite generated could not evaluate correctness. The designers of EvoSuite expected users to manually find and fix any incorrect test cases, but EvoSuite did not provide any affordances to support users in this task. The EvoSuite study used qualitative data to suggest that poor readability of test cases generated by EvoSuite may have caused the study's negative result, and this finding gave focus to research that aimed to improve test case readability (e.g., [29, 44, 49, 58, 60]). Unlike EvoSuite, TerzoN detects bugs in current programs, provides support for finding contradictions among test cases, and has shown positive results relative to a state-of-the-art testing tool, fast-check, in a randomized human trial with professional software engineers.

Ng et al. [45] observed that one of the top barriers reported for automatic testing tools is poor usability. Li et al. [36] called for improved usability of random testing tools. Prado and Vincenzi [50] and Arcuri [3] observed that ATUGs like EvoSuite are relatively unused in industry and that tool designers often prioritize technical or secondary measures over the tool's performance with users. Rojas et al. [53] observed users and found the need for improved EvoSuite usability and for it to be integrated into the IDE. In a recent industry blog post [61], James Sowers questioned many interaction aspects of current testing tools such as Jest. With the goal of creating highly usable testing tools, we designed TerzoN in reaction to the findings of the prior work as well as our own observations of users interacting with various automatic test generation tools.

8 IMPLICATIONS FOR TESTING RESEARCHERS AND TOOL DESIGNERS

This study has a number of important implications for researchers and tool designers. TerzoN evaluated favorably to fast-check on tasks where fast-check might be expected to have an advantage, and this empirical result may point to the potential benefits of a new class of tool, **Composite Test Generators**, which, in the example of TerzoN, incorporate various types of oracles and provide a coherent user interface that allows users to assert correctness in various ways. Such tools might benefit users by, e.g., (i) allowing users to continue refining a set of tests even when a property to test is not coming to mind, (ii) calling attention to likely-incorrect properties by comparing them to other (e.g., human-annotated) oracle judgments, and (iii) coherently organizing and displaying all test results, including full details of passing tests. Continuing testing beyond the first failure by default may also provide more feedback that helps users generate and debug their tests more efficiently.

Our empirical evaluation provides further support for prior assertions that empirical software engineering testing researchers might achieve more impactful results by evaluating their tools with users [14, 24] and by considering user efficiency and satisfaction *in addition to* effectiveness when designing and evaluating tools [14]. Our experience designing TerzoN for this study implies that some popular strategies for finding the simplest failing input might require adaptation to be integrated with a Composite Oracle due to each input potentially being evaluated by multiple oracles. The Composite Oracle's hierarchical nature might also facilitate optimizations for large-scale testing, such as short-circuit, lazy, and parallel evaluation of judgments.

9 FUTURE WORK

TerzoN includes a particular instantiation of the Composite Oracle that integrates 3 types of oracles; however, other instantiations and other types of oracles are possible. Code and tests are now being generated by **Large Language Models (LLMs)** [51, 65], and our study's most experienced participant said he uses LLMs to write his test cases (P21.Q16). However, LLM-generated code can be incorrect in various [28, 37] and/or subtle ways, and users similar to participant P21 may benefit from a Composite Oracle that compares LLM-based oracle judgments to those of other oracles and alerts the user to contradictions. We would like to understand how users might interact with or benefit from such a Composite Oracle that incorporates LLM-based oracle judgments.

TerzoN is available on the Visual Studio Code Marketplace [15], and we plan to use TerzoN in real-world situations outside our study. As these situations may require more complex input generation techniques and guidance, we plan to investigate how TerzoN might provide more expressiveness and control while retaining the ability to immediately start testing with an automatically synthesized input generator function. We would also like users to be able to add their own example test inputs more easily. Usability might vary with additional features [42], and it will be important to evaluate how additional complexity may affect usability [59]. Future versions of TerzoN may support *other* IDEs, *other* languages, and *other* types of oracles, in which case further studies would be required.

10 CONCLUSION

In this paper, we formally defined the Composite Oracle and presented TerzoN, an Automatic Test sUite Generator (ATUG) that surfaces within its user interface a particular instantiation of the Composite Oracle that incorporates property-based, example-based, and implicit oracles. With TerzoN, a user may generate an initial set of tests using an implicit oracle with a single button click, refine the tests by annotating the correctness of particular execution examples, and write snippets of code that define properties of the program that judge current and future execution examples. Because TerzoN's Composite Oracle is a composition of multiple types of oracle that assert correctness in various ways, TerzoN *also* compares judgments among the oracle types and alerts the user when, e.g., a property-based test and a human-annotated example test yield conflicting results.

To empirically evaluate whether TerzoN's particular Composite Oracle and user interface, together, might benefit users, we conducted a randomized controlled human trial with 14 professional software engineers where TerzoN was the intervention treatment and the popular industry tool, fast-check, was the control treatment. Participants using TerzoN elicited 72% more bugs ($p < 0.01$), accurately described more than twice the number of bugs ($p < 0.01$) and tested 16% more quickly ($p < 0.05$) relative to fast-check. This positive empirical result may point to potential future impacts of Composite Test Generators that report full test details and allow users to assert correctness in various ways using a coherent user interface. We hope that our contributions and empirical results spur much-needed further research into testing tool usability, interventions that help users generate test suites more effectively and efficiently, and appropriate evaluations of each intervention's success.

11 DATA AVAILABILITY

We provide the data and materials to reproduce our results as supplementary material [16], except for video and audio data, which we are unable to share due to participant privacy and IRB restrictions.

ACKNOWLEDGMENTS

This work was supported in part by a CyLab seed funding award and by NSF grants 1910264, 2150217, and 2339775. Any opinions, findings, or conclusions expressed in this material are those of the authors and do not necessarily reflect those of the sponsors.

REFERENCES

[1] Bilal Amir and Paul Ralph. 2018. There is no random sampling in software engineering research. In *Proceedings of the 40th International Conference on Software Engineering: companion proceeedings*. 344–345.

[2] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001. https://doi.org/10.1016/j.jss.2013.02.061

[3] Andrea Arcuri. 2018. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering* 23, 4 (2018), 1959–1981.

[4] Sebastian Baltes and Paul Ralph. 2022. Sampling in software engineering research: a critical review and guidelines. *Empirical Software Engineering* 27, 4 (April 2022), 94. https://doi.org/10.1007/s10664-021-10072-8

[5] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[6] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2017. Developer testing in the ide: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering* 45, 3 (2017), 261–284.

[7] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 836–850. https://doi.org/10.1145/3477132.3483540

[8] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.

[9] John Brooke et al. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

[10] Stephen Cass. 2024. Top Programming Languages 2024 - IEEE Spectrum. https://spectrum.ieee.org/top-programming-languages-2024

[11] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279.

[12] Anastasia Danilova. 2022. *How to Conduct Security Studies with Software Developers.* Ph. D. Dissertation. Universitäts- und Landesbibliothek Bonn.

[13] Matthew C. Davis, Emad Aghayi, Thomas D. Latoza, Xiaoyin Wang, Brad A Myers, and Joshua Sunshine. 2023. What's (Not) Working in Programmer User Studies? *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 120 (jul 2023), 32 pages. https://doi.org/10.1145/3587157

[14] Matthew C. Davis, Sangheon Choi, Sam Estep, Brad A. Myers, and Joshua Sunshine. 2023. NaNofuzz: A Usable Tool for Automatic Test Generation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1114–1126. https://doi.org/10.1145/3611643.3616327

[15] Matthew C. Davis, Amy Wei, Sangheon Choi, and Sam Estep. 2024. NaNofuzz - Visual Studio Marketplace. https://marketplace.visualstudio.com/items?itemName=penrose.nanofuzz [Online; accessed 2024-09-01].

[16] Matthew C. Davis, Amy Wei, Brad A. Myers, and Joshua Sunshine. 2024. Replication package for TerzoN study. https://figshare.com/s/9acd5b6c1033d53840d1

[17] Nicolas Dubien. 2024. dubzzz/fast-check. https://github.com/dubzzz/fast-check original-date: 2017-10-30T23:41:11Z.

[18] Nicolas Dubien. 2024. fast-check. https://www.npmjs.com/package/fast-check

[19] Nicolas Dubien. 2024. fast-check official documentation | fast-check. https://fast-check.dev/

[20] Eduard Enoiu and Robert Feldt. 2021. Towards Human-Like Automated Test Generation: Perspectives from Cognition and Problem Solving. In *2021 IEEE/ACM 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 123–124. https://doi.org/10.1109/CHASE52884.2021.00026

[21] Marc Fisher, Mingming Cao, Gregg Rothermel, Curtis R Cook, and Margaret M Burnett. 2002. Automated test case generation for spreadsheets. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*. IEEE, 141–151.

[22] Marc Fisher, Gregg Rothermel, Darren Brown, Mingming Cao, Curtis Cook, and Margaret Burnett. 2006. Integrating automated test generation into the WYSIWYT spreadsheet testing methodology. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15, 2 (2006), 150–194.

[23] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (Feb. 2013), 276–291. https://doi.org/10.1109/TSE.2012.14

[24] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 4 (2015), 1–49.
[25] GitHub. 2023. GitHub Codespaces. https://github.com/features/codespaces. [Online; accessed 2023-07-21].
[26] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3597503.3639581
[27] Harrison Goldstein, Joseph W Cutler, Adam Stein, Benjamin C Pierce, and Andrew Head. 2022. Some Problems with Properties. In *Proc. Workshop on the Human Aspects of Types and Reasoning Assistants (HATRA)*.
[28] Google DevOps Research and Assessment. 2024. *2024 State of DevOps Report*. Technical Report 2024. Google.
[29] Giovanni Grano, Simone Scalabrino, Harald C Gall, and Rocco Oliveto. 2018. An empirical investigation on the readability of manual and generated test cases. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 348–3483.
[30] Alex Groce, Todd Kulesza, Chaoqiang Zhang, Shalini Shamasunder, Margaret Burnett, Weng-Keen Wong, Simone Stumpf, Shubhomoy Das, Amber Shinsel, Forrest Bice, and Kevin McIntosh. 2014. You Are the Only Possible Oracle: Effective Test Selection for End Users of Interactive Machine Learning Systems. *IEEE Transactions on Software Engineering* 40, 3 (March 2014), 307–323. https://doi.org/10.1109/TSE.2013.59
[31] Jerry L. Hintze and Ray D. Nelson. 1998. Violin Plots: A Box Plot-Density Trace Synergism. *The American Statistician* 52, 2 (May 1998), 181–184. https://doi.org/10.1080/00031305.1998.10480559
[32] ISO. 2018. Ergonomics of human-system interaction—Part 11: Usability: Definitions and concepts ISO 9241–11: 2018 (en).
[33] Pankaj Jalote. 2008. *A concise introduction to software engineering*. Springer Science & Business Media.
[34] Amy J Ko, Thomas D LaToza, and Margaret M Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141.
[35] Jeshua S Kracht, Jacob Z Petrovic, and Kristen R Walcott-Justice. 2014. Empirically evaluating the quality of automatically generated and manually written test suites. In *2014 14th International Conference on Quality Software*. IEEE, 256–265.
[36] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers.. In *USENIX Security Symposium*. 2777–2794.
[37] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2024. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3597503.3608128
[38] David R. MacIver, Zac Hatfield-Dodds, and many other contributors. 2019. Hypothesis: A new approach to property-based testing. https://doi.org/10.21105/joss.01891
[39] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.
[40] Cyrus R. Mehta and Nitin R. Patel. 1983. A Network Algorithm for Performing Fisher's Exact Test in r × c Contingency Tables. *J. Amer. Statist. Assoc.* 78, 382 (1983), 427–434. https://doi.org/10.2307/2288652 Publisher: [American Statistical Association, Taylor & Francis, Ltd.].
[41] Microsoft. 2023. Overview of Microsoft IntelliTester. https://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual/. [Online; accessed 2023-01-27].
[42] Brad A Myers. 1994. Challenges of HCI Design and Implementation. *Interactions* 1, 1 (jan 1994), 73–83. https://doi.org/10.1145/174800.174808
[43] Brad A Myers, Amy J Ko, Thomas D LaToza, and YoungSeok Yoon. 2016. Programmers are users too: Human-centered methods for improving programming tools. *Computer* 49, 7 (2016), 44–52.
[44] Mathieu Nassif, Alexa Hernandez, Ashvitha Sridharan, and Martin P Robillard. 2021. Generating unit tests for documentation. *IEEE Transactions on Software Engineering* (2021).
[45] Sebastian P Ng, Tafline Murnane, Karl Reed, D Grant, and Tsong Yueh Chen. 2004. A preliminary survey on software testing practices in Australia. In *2004 Australian Software Engineering Conference. Proceedings.* IEEE, 116–125.
[46] Jakob Nielsen. 1994. Enhancing the explanatory power of usability heuristics. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. 152–158.
[47] Gerard O'Regan. 2019. Fundamentals of Software Testing. In *Concise Guide to Software Testing*. Springer, 59–78.
[48] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, Minneapolis, MN, USA, 75–84.

https://doi.org/10.1109/ICSE.2007.37 ISSN: 0270-5257.

[49] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C Gall. 2016. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th international conference on software engineering*. 547–558.

[50] Marllos Paiva Prado and Auri Marcelo Rizzo Vincenzi. 2018. Towards cognitive support for unit testing: A qualitative study with practitioners. *Journal of Systems and Software* 141 (2018), 66–84. https://doi.org/10.1016/j.jss.2018.03.052

[51] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn. 2023. CAT-LM Training Language Models on Aligned Code And Tests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 409–420. https://doi.org/10.1109/ASE56229.2023.00193

[52] Darrel A. Regier, William E. Narrow, Diana E. Clarke, Helena C. Kraemer, S. Janet Kuramoto, Emily A. Kuhl, and David J. Kupfer. 2013. DSM-5 Field Trials in the United States and Canada, Part II: Test-Retest Reliability of Selected Categorical Diagnoses. *American Journal of Psychiatry* 170, 1 (Jan. 2013), 59–70. https://doi.org/10.1176/appi.ajp.2012.12070999 Publisher: American Psychiatric Publishing.

[53] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2015. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *Proceedings of the 2015 international symposium on software testing and analysis*. 338–349.

[54] Robert Rosenthal and Ralph L Rosnow. 2008. *Essentials of behavioral research: Methods and data analysis*.

[55] G. Rothermel, L. Li, and M. Burnett. 1997. Testing strategies for form-based visual programs. In *Proceedings The Eighth International Symposium on Software Reliability Engineering*. 96–107. https://doi.org/10.1109/ISSRE.1997.630851

[56] G. Rothermel, L. Li, C. DuPuis, and M. Burnett. 1998. What you see is what you test: a methodology for testing form-based visual programs. In *Proceedings of the 20th International Conference on Software Engineering*. 198–207. https://doi.org/10.1109/ICSE.1998.671118 ISSN: 0270-5257.

[57] Karen J Rothermel, Curtis R Cook, Margaret M Burnett, Justin Schonfeld, Thomas RG Green, and Gregg Rothermel. 2000. WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation. In *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*. IEEE, 230–239.

[58] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. 2020. DeepTC-Enhancer: Improving the readability of automatically generated tests. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 287–298.

[59] Caitlin Sadowski and Thomas Zimmermann. 2019. *Rethinking productivity in software engineering*. Springer Nature.

[60] Novi Setiani, Ridi Ferdiana, and Rudy Hartanto. 2022. Understandable Automatic Generated Unit Tests using Semantic and Format Improvement. In *2022 6th International Conference on Informatics and Computational Sciences (ICICoS)*. 122–127. https://doi.org/10.1109/ICICoS56336.2022.9930600

[61] James Somers. 2023. What if writing tests was a joyful experience? https://blog.janestreet.com/the-joy-of-expect-tests/. [Online; accessed 2023-01-22].

[62] Facebook Open Source. 2023. Jest - Delightful Javascript Testing. https://jestjs.io/. [Online; accessed 2024-07-08].

[63] Tristan Teufel and contributors. 2022. Jest Runner. https://github.com/firsttris/vscode-jest-runner. [Online; accessed 2022-11-10].

[64] Priyadarshi Tripathy and Kshirasagar Naik. 2011. *Software testing and quality assurance: theory and practice*. John Wiley & Sons.

[65] Vasudev Vikram, Caroline Lemieux, Joshua Sunshine, and Rohan Padhye. 2024. Can Large Language Models Write Good Property-Based Tests? https://doi.org/10.48550/arXiv.2307.04346 arXiv:2307.04346 [cs].

[66] John Wrenn, Tim Nelson, and Shriram Krishnamurthi. 2021. Using Relational Problems to Teach Property-Based Testing. *The art science and engineering of programming* 5, 2 (2021).

[67] Michał Zalewski. 2014. american fuzzy lop. https://lcamtuf.coredump.cx/afl/. https://lcamtuf.coredump.cx/afl/ [Online; accessed 2023-06-27].

Received 2024-09-13; accepted 2025-04-01; revised 12 September 2024