NaNofuzz: A Usable Tool for Automatic Test Generation

Matthew C. Davis
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
mcd2@cs.cmu.edu

Sangheon Choi
Rose-Hulman Institute of Technology
Terre Haute, Indiana, USA
chois3@rose-hulman.edu

Sam Estep
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
estep@cmu.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

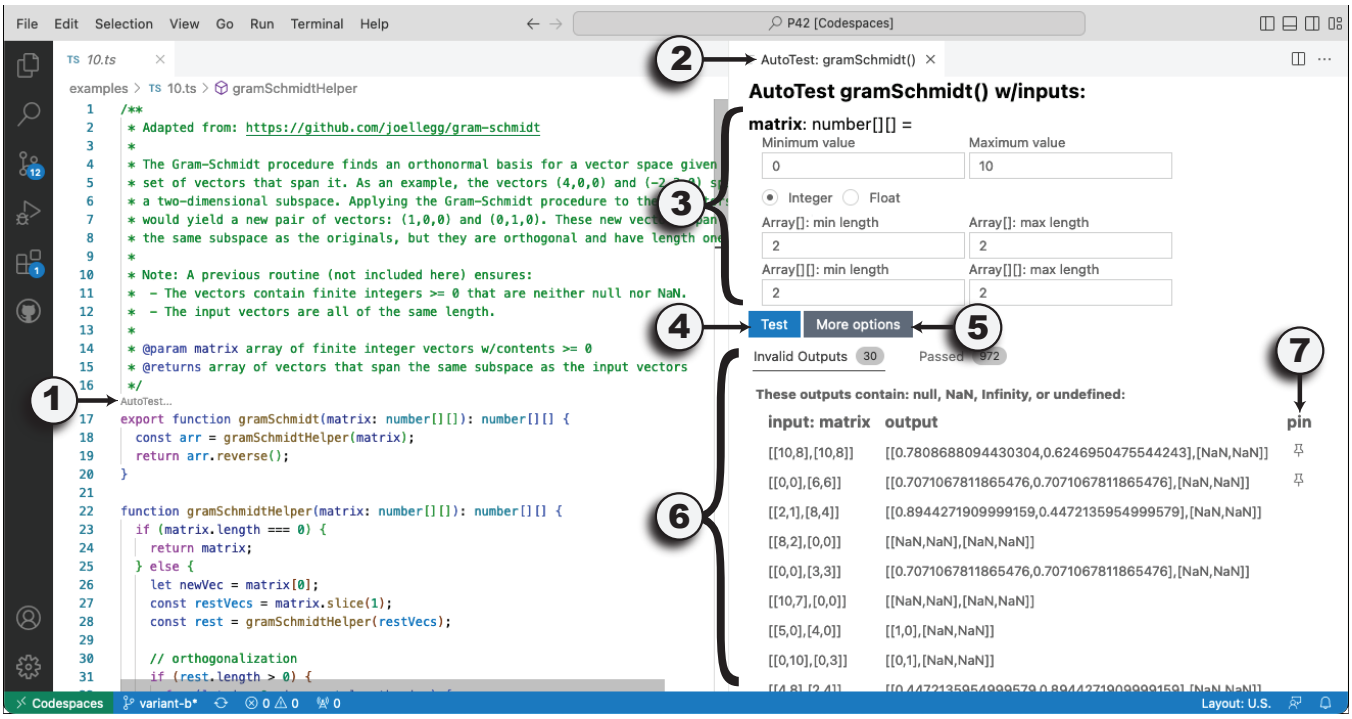

Figure 1: The NaNofuzz user interface in the Visual Studio Code IDE provides one-click test generation for TypeScript programs. Key UI elements: (1) AutoTest button above a function signature, (2) NaNofuzz testing window beside the program under test, (3) Customizable input parameters with default values, (4) Test button to start NaNofuzz, (5) Advanced options, (6) Categorized testing results with likely bugs prioritized in the display, (7) Pin button to add test cases to the test suite in Jest format.

ABSTRACT

In the United States alone, software testing labor is estimated to cost $48 billion USD per year. Despite widespread test execution automation and automation in other areas of software engineering, test suites continue to be created manually by software engineers. We have built a test generation tool, called NaNofuzz, that helps users find bugs in their code by suggesting tests where the output is likely indicative of a bug, e.g., that return NaN (not-a-number) values. NaNofuzz is an interactive tool embedded in a development environment to fit into the programmer's workflow. NaNofuzz tests a function with as little as one button press, analyses the program to determine inputs it should evaluate, executes the program on those inputs, and categorizes outputs to prioritize likely bugs. We conducted a randomized controlled trial with 28 professional software engineers using NaNofuzz as the intervention treatment and the popular manual testing tool, Jest, as the control treatment. Participants using NaNofuzz on average identified bugs more accurately ($p < .05$, by 30%), were more confident in their tests ($p < .03$, by 20%), and finished their tasks more quickly ($p < .007$, by 30%).

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616327

CCS CONCEPTS

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

KEYWORDS

Empirical software engineering, user study, software testing, human subjects, experiments, usable testing, automatic test generation

ACM Reference Format:
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 '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616327

1 INTRODUCTION

Software testing often intends to prevent bugs of various forms from affecting users and operations [30, 47, 64]. These efforts are estimated to represent 28% [6] to 50% [64] of the $174 billion USD annual [45] 2021 labor cost of US software engineering professionals. While each 2% reduction in testing effort may save $1 billion USD per year in labor[1], engineers largely create test suites manually [4, 21, 24, 35]. An important problem is therefore how to provide automation support for engineers creating test suites.

Automatic Test sUite Generation (ATUG) tools attempt to fill this gap by generating persistent test cases using various techniques and objectives (e.g., test suite size, code coverage, readability, mutants killed, etc.). However, most existing tools lack evidence that they improve a software engineer's ability to create effective test suites [24, 51], and these tools are not broadly adopted in industry [4, 51]. For example, when the ATUG tool EvoSuite was evaluated with humans, it showed a *penalty* for using it [24]. EvoSuite uses the (buggy) software under test as its test oracle; as all tests "pass," the engineer must read the generated tests to find and fix the "erroneous" tests. The negative human evaluation result implies that the *new* task of finding and fixing invalid tests may be more difficult than the *old* task EvoSuite seeks to replace: manually creating a test suite. Subsequent research has tried to improve these results by improving test suite readability [28, 42, 49, 57, 60], but evidence is unclear that these help engineers relative to manual test suite creation. We argue that a *human-focused* approach may be more appropriate for designing ATUG tools so that they improve engineers' ability to find bugs and create test suites.

We present **NaNofuzz**, a human-focused ATUG tool that is integrated into the Visual Studio Code integrated development environment (IDE). NaNofuzz allows a software engineer to test a function with as little as one click. It accomplishes this by analyzing the function signature to determine what types of inputs the function accepts. It uses heuristics to propose default value ranges for each input and allows the engineer to refine those ranges, if necessary. NaNofuzz rapidly generates test cases by sampling the input domain of the function. As NaNofuzz generates and executes test cases, it automatically categorizes and organizes the test results in the engineer's IDE using a set of heuristics to identify likely problematic output values. The test cases more likely to be eliciting bugs are more-prominently displayed to draw the engineer's attention. The engineer may then add test cases to the test suite.

We conducted a **randomized controlled human trial** with 28 professional software engineers using NaNofuzz as the intervention treatment and the popular manual testing tool, Jest [62], as the control treatment. Participants using NaNofuzz on average identified bugs more accurately ($p < .05$, by 30%), were more confident in their tests ($p < .03$, by 20%), and finished their tasks more quickly ($p < .007$, by 30%).

This paper contributes: (i) NaNofuzz, a human-focused automatic test generation tool; and (ii) an experimental human evaluation of NaNofuzz that provides evidence that NaNofuzz improves software engineers' ability to generate test cases for a test suite.

2 DEFINITIONS

This paper uses a number of terms. A **test oracle**, proposed by Howden [29], allows one to "check the correctness of test output." "Program under test" is abbreviated as **PUT**, and "software engineer" is often shortened to **engineer**. **Test suite size** measures the number of test cases or statements within the test suite. A **mutant** is a modification made to the original program through simple and intentional syntax changes that aims to create faults [31]. Mutants are **killed** when a test case detects differences between the mutant and the original code. **Code coverage** measures what code is executed during a test [65]. A **fuzzer** is a tool that exercises a PUT with randomly-determined inputs to elicit bugs [38, 40]. An **opaque box fuzzer** has no knowledge of the internals of the PUT and generates test cases based on input/output behavior [38].

3 NANOFUZZ

NaNofuzz is intended to help engineers find incorrectness during development or testing. Salient aspects of NaNofuzz are described below. We reference the interface elements shown in Figure 1.

(1) AutoTest button. NaNofuzz decorates exported TypeScript functions displayed in the IDE's editor with an "AutoTest" button. Clicking the button opens NaNofuzz in a side window **(2)**.

(2) NaNofuzz window. The testing window opens beside the function under test so that both the code and its tests are visible.

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

(4) Test button. This button starts the test, which uses a stochastic, opaque-box fuzzer with an implicit oracle that classifies the following as likely errors: runtime exceptions, non-termination within a configurable time threshold, and outputs containing null, NaN, infinity, or undefined. When the fuzzer terminates, testing results are displayed in the results grid **(6)**.

(5) Options. This button toggles the display of advanced options, which are shown in Figure 2 and allow the engineer to, e.g., adjust the fuzzer's runtime. By default, the fuzzer returns results in 3 seconds, and runs at most 1,000 tests to ensure the tool provides rapid feedback to maintain the engineer's attention. Longer testing sessions may be configured.

[1] $174 billion · 28% [6] · 2% = $0.97 billion; $174 billion · 50% [64] · 2% = $1.74 billion

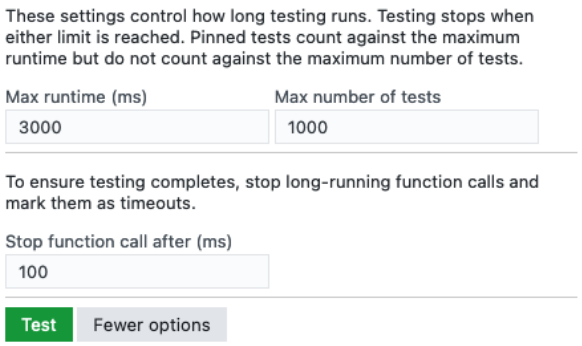

Figure 2: NaNofuzz with "More Options" Pane

(6) Results. This pane displays a set of relevant category tabs, each containing a grid of test results. The categories are: non-termination, runtime exception, erroneous outputs (e.g., null, NaN, infinity, undefined), and passed tests. Tabs containing no result are not displayed. To direct the engineer's attention to tests that are likely to be errors, the tabs are organized in the order shown above. We opted to display tests and results in table form rather than as code to minimize the engineer's reading effort.

(7) Pin button. After NaNofuzz displays test results, the engineer may add any of the generated tests to the test suite by pressing the Pin button next to the test result on the results grid **(6)**. In Figure 1, the engineer has pinned two tests. When pinned, NaNofuzz saves the unit test in Jest format to the file system. When the test button **(4)** is pressed again, all pinned tests are re-executed with their test results displayed at the top of the grid above any newly-generated tests. If a test is un-pinned, then the Jest unit test is removed from the file system.

4 SCOPE AND LIMITATIONS

NaNofuzz has a number of important limitations, but it is a useful vehicle for evaluating how a human-focused automatic testing tool might affect an engineer's ability to generate a test suite. As mentioned above, the experimental version of NaNofuzz in this study uses an implicit oracle such that it is currently not possible to specify desired result values for a test case. This version of NaNofuzz also only 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. Type references are not supported. In addition, default input ranges are determined using type-based heuristics. We plan to broaden support as NaNofuzz matures.

5 EXPERIMENT

We provide an overview of the **randomized controlled trial** in Figure 3 and the data and methods in Table 1. Sadowski and Zimmermann [58] describe software engineer productivity in terms of three dimensions—quality, satisfaction, and velocity. This paper investigates the extent to which NaNofuzz impacts three testing measures inspired by these dimensions: bug identification accuracy, confidence, and task time. This study aims to answer three variants of the question: *"Relative to standard practice (e.g., Jest), to what extent may NaNofuzz affect X?"* where the values of X are:

RQ1. X = the number of bugs an engineer accurately identifies
RQ2. X = the engineer's confidence in the test activity
RQ3. X = the engineer's time on the testing task

Hypothesis: We hypothesize that NaNofuzz improves software engineers' ability to create tests by automatically supporting two tasks that are difficult for engineers: identifying edge cases and understanding how outputs relate to inputs.

This experiment uses a **between-subjects design**. 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 more rare. One approach to reduce the number of participants required for statistical significance is to select a within-subjects design with repeated measures such that participants use both treatments to complete the same or matched tasks. Fatigue and learning effects are important problems that may often be addressed by counter-balancing the task/treatment sequence. 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 design is more appropriate, despite its offering less statistical power for the same number of participants.

In our design, participants use 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. Before the study began, we piloted the study with 9 pilot participants to refine the tasks and instructions. This study was reviewed by our Institutional Review Board.

5.1 Treatments

Jest [62] is an open-source TypeScript and JavaScript unit-testing framework. As of October 2022, Jest had 50 million monthly downloads, was used in over 3.8 million public GitHub repositories, and was used at Meta (Jest's creator), Twitter, Spotify, Airbnb, and many other companies [62]. Thus, Jest represents the state-of-the-art in manual test suite generation. Engineers largely create test suites manually rather than using ATUG tools [4, 21, 24, 35] such as EvoSuite, which one study showed had disappointing results when compared against a manual testing tool [24]. Consequently, it is important to compare ATUG tools with the tools that practitioners are actively using in practice, even if those widely-used tools lack automation.

To control for differences that we did not want to measure, we used Jest Runner [63], which allowed participants to run Jest via a GUI button in the IDE, similar to the way they run NaNofuzz. Similarly, our protocol instructed participant to open the Jest tests "to the side" as shown in Figure 4 so that both code and tests were simultaneously visible, similar to NaNofuzz. To minimize typing,

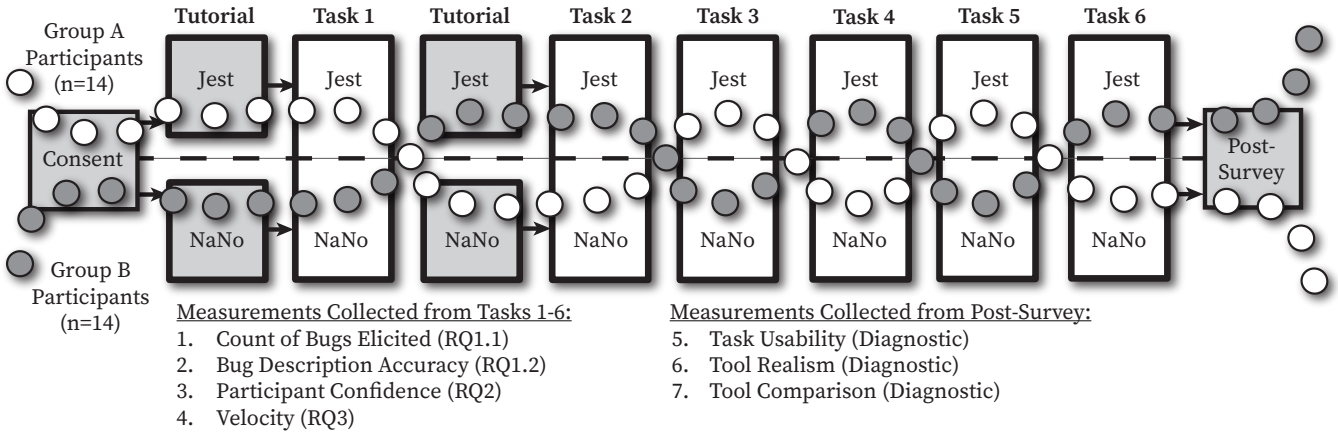

Figure 3: Visualization of study task sequence (see Section 5). Participants are randomly assigned by pairs into Group A or Group B, which determines the treatment for each task. Shaded tasks (e.g., tutorials) are not time-limited.

Table 1: Measurements, Instruments, and Methods

ID	Measurement	Instrument	Collection Step	Research Question	Productivity Dimension†	Analysis Method(s)
1	Count of Bugs Elicited	Scoring Rubric	Task	RQ1.1	Quality	ANOVA, Fisher's exact test* [53]
2	Bug Descr. Accuracy	Scoring Rubric	Task	RQ1.2	Quality	ANOVA, Fisher's exact test* [53]
3	Engineer Confidence	Likert Scale	Task	RQ2	Satisfaction	ANOVA, Fisher's exact test* [53]
4	Task Time	Elapsed Time (clock)	Task	RQ3	Velocity	ANOVA, paired t-test* [53]
5	Tool Usability	Sys. Usability Scale [9]	Post-Survey	Diagnostic	n/a	Direct comparison
6	Task Realism	Likert Scale	Post-Survey	Diagnostic	n/a	Direct comparison
7	Tool Comparison	Free-form Survey	Post-Survey	Diagnostic	n/a	Inductive Thematic Analysis [8]

†=As defined by Sadowski and Zimmermann [58]; *=two-tailed

Table 2: Programs Under Test

Task	Name	Program Found On	Lines	Error Class	Bugs
1	6.ts	Stack Overflow	3	exception	1
2	3.ts	Stack Overflow	4	NaN output	1
3	7.ts	Stack Overflow	12	divide by 0	1
4	11.ts	Rosetta Code	14	exception	1
5	14.ts	Stack Overflow	15	infinite loop	1
6	10.ts	GitHub	57	NaN output	2

we provided Jest participants a single passing Jest test case and instructed participants to copy and paste from it.

NaNofuzz is the ATUG tool we created and described in Section 3. To minimize influence on participants, we did not disclose during the session that we created NaNofuzz; rather, we characterized both tools as ones participants might have used previously. Participants were not allowed to use the internet during the study, and we called NaNofuzz "AutoTest" to obscure its origin.

5.2 Tasks

As shown in Figure 3, the experiment included two tutorials—one for each treatment—and six testing tasks. The six tasks varied: (a) the program under test (Table 2) according to our study protocol and (b) the treatment (Section 5.1) according to the participant's random group assignment and task number. In each of the six 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 provided verbal instructions from a script explaining that the goal of the task was to find inputs to the program that cause any of the following results: null, undefined, NaN, infinity, a runtime exception, or apparent non-termination. We specified which treatment to use and, if Jest, directed the participant to open the Jest test file "to the side" so that both the tests and the code under test were simultaneously visible. We reminded each participant that the comment at the top of the program specified the allowed input values. After providing the instructions and the participant indicated they were ready, we manually recorded the start time. Using Zoom, we monitored each participant's screen and audio to ensure use of the intended treatment and program. During the task, the participant tested the program using the designated

treatment and created test cases. At the end of 15 minutes or when the participant was done creating the tests, we recorded the stop time and verbally instructed the participant to complete a post-task survey where they typed up their understanding of the generalized input domains where bugs occurred and their confidence in the testing activity according to a 5-point Likert scale.

Programs Under Test. We utilized Ko et al.'s [34] suggestion to use "found" tasks: we searched GitHub, Stack Overflow, and Rosetta Code and found 14 faulty TypeScript programs that violate an implicit oracle in some circumstances due to the presence of bugs. As some programs were code fragments, we added or edited code necessary for the program to run. We added type annotations as-needed to avoid distracting IDE warnings and provided a description of each program, its allowed inputs, and its expected outputs. We ran a series of pilots with students using various combinations of the 14 buggy programs. These pilots allowed us to: (i) eliminate programs that pilot users could not finish testing in 15 minutes regardless of the treatment used, and (ii) determine the relative difficulty of the remaining programs. Based on these observations and pilot participant feedback, we selected the six programs under test described in Table 2 and sequenced them according to increasing difficulty.

Task Infrastructure. Deployment of experimental study software to remote software engineers is difficult [13]. To avoid some of these difficulties, we hosted the tasks on GitHub Codespaces [26], which provides a web-based VS Code IDE and VM such that a remote software engineer may edit, run, test, and debug code using a complete IDE inside a web browser.

Tutorials. We designed the NaNofuzz and Jest tutorials to be similar such that both have two short exercises and are roughly the same length. While participants spent a mean 93 seconds longer in the NaNofuzz tutorial, this may be due to 68% (19/28) of participants entering the study session with previous Jest experience.

5.3 Participants

Finding and recruiting a large and/or representative sample of professional software engineers is difficult [2, 5, 13, 34]. We recruited professional software engineers via LinkedIn, Twitter, 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; however, we did not offer incentives to do so.

From November 4, 2022 to January 6, 2023, the screener survey received 552 responses and automatically classified 99 responses as likely being eligible, of which 35 were humans that scheduled sessions and 28 completed the study. Four consented participants are excluded from the data set: one was unable to access their GitHub account and was unable to start the study, another did not follow protocol, and two more had to leave unexpectedly without finishing the session.

In the screener survey, potential participants self-reported: gender; professional software engineering experience; hours of coding per week; and experience with testing tools, TypeScript, Jest, and VS Code. The screener included timed questions recommended by Danilova [12] to eliminate non-programmers. We also created timed TypeScript and Jest questions, which we used to identify bots. The final page of the screener included a Google Calendar link that allowed the participant to choose a 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 the participant was flagged as needing a match. When the next participant with the same experience level scheduled a time slot, the new participant would be matched to the previous one such that one participant would be randomly assigned to group *A* and the other randomly assigned to group *B*.

Participant demographics were as follows: 5 participants identified as female, 21 as male, and 2 did not disclose; 4 participants had 10+ years of professional experience, 4 had 6–9 years, and 20 had 1–5 years; 11 participants reported spending 30+ hours coding per week, 13 reported spending 10–29 hours per week, and 4 reported exactly 5 hours per week. A table showing the demographics of each participant is provided in the supplementary materials.

5.4 Measurements

The experiment includes seven measurements as shown in Table 1:

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

(2) Bug description accuracy. After generating the tests, the participant was asked to type in the general circumstances under which each bug may be elicited. For example, suppose we had a program that throws a runtime exception for integers > 1. According to the rubric, a full score was given for stating the entire set accurately (integers>1). A half score was given if the participant's set included only part of the rubric's described subset (e.g., integers>2). Finally, a half score was deducted if the participant's set included allowed inputs that did not elicit the bug (e.g., integers<1).

(3) Engineer confidence. At the end of each task, the participant reported their degree of confidence in identifying the inputs under which the bugs are elicited using a 5-point Likert scale.

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

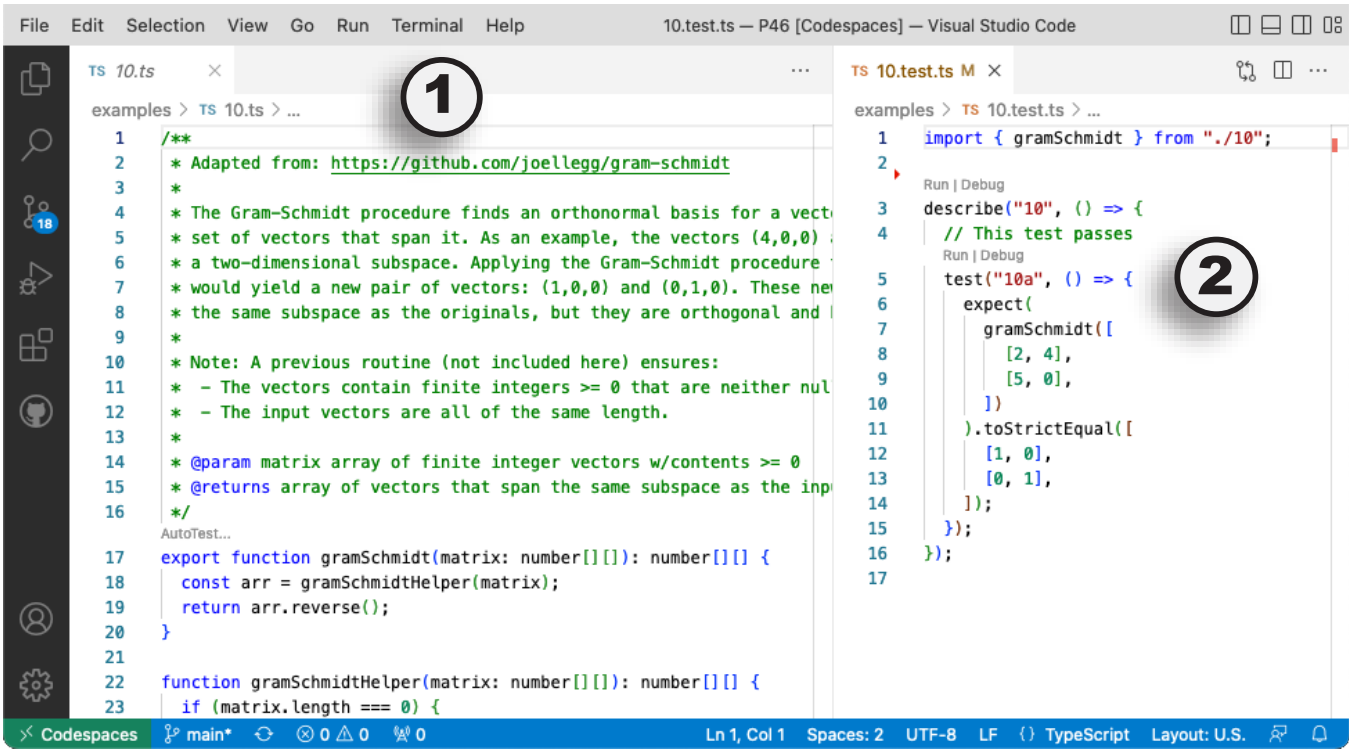

Figure 4: VS Code Editor (1) with Jest Tests (2) Opened "to the side."

(5) Tool usability. The System Usability Scale (SUS) [9] 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 the treatments': (i) effectiveness in testing, (ii) ease of use, (iii) fit with the participant's test and debug workflow, and (iv) opportunities for improvement.

5.5 Data Analysis

As shown in Table 1, this mixed-methods experiment quantitatively analyzed task data to answer the research questions. A diagnostic explanatory analysis was also designed to help explain or explore the primary quantitative results. We discuss the analysis for each measurement from Figure 3 and Table 1 below.

(1) Count of bugs elicited. We used a randomized matched pair design (see Section 5.3) based on years of professional experience. From 28 total participants, this yielded 14 pairings. Within each pairing, we created two pseudo-participants, one by taking all the data from the pairing using Jest, and another by taking all the data from the pairing using NaNofuzz. 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 are a total of seven bugs per participant for this measure, so for this measure we first allocated $7 \cdot 14 = 98$ possible tests, and then within the two groups (Jest and NaNofuzz), 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 Jest, and experience with VS Code. For this measure, task six has twice as many bugs as tasks 1-5, so we divided its raw score by 2 so that scores across tasks have the same range.

(2) Bug description accuracy. This analysis is the same as (1), except we used categories corresponding to the recorded accuracy values of -1, 0, 1, 2, and 3. We counted up the number of accuracy scores in each category for the two groups, as shown in Table 4.

(3) Engineer 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 4, and we did not need to normalize the range of task 6.

(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, and we did not need to normalize the range of task 6.

(5) Tool usability. We counted the number of participants who gave a higher System Usability Scale [9] score for NaNofuzz vs. Jest

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 qualitatively using the inductive thematic analysis procedure described by Braun and Clarke [8], which calls attention to Frith and Gleeson [25] as a "particularly good example of an inductive thematic analysis." We therefore used Frith and Gleeson as a model for our data analysis. Braun and Clarke [8] describes 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] and Frith and Gleeson [25], the third author established replicability of the second author's result by re-coding the data for five random participants using the second author's codes. This resulted in a substantial [36] level of inter-rater reliability ($\kappa = 0.6962$).

6 RESULTS

Below we report the results of the data analyses described in Section 5.5. Table 3 summarizes measures 1–4 of Figure 3. Figure 5 plots results for measures 1–5. Table 4 shows the two-tailed Fisher's exact tests for measures 1–3. Table 5 shows the paired t-test for measure 4.

(1) Count of Bugs Elicited. Table 3 shows that participants using NaNofuzz, on average, detected more bugs (92%) than when using Jest (80%). The two-tailed Fisher's exact test shown in Table 4 (Measure 1) indicates the differences in this measure are statistically significant ($p < 0.05$). The ANOVA raises no statistically-significant alternative hypotheses.

(2) Bug description accuracy. Table 3 shows that participants using NaNofuzz, on average, described bugs more accurately (1.57/2.00) than when using Jest (1.18/2.00). The two-tailed Fisher's exact test shown in Table 4 (Measure 2) indicates the differences in this measure are statistically significant ($p < 0.007$). The ANOVA indicates that VS Code experience was positively correlated with task time ($p < 0.05$) on task 2.

(3) Engineer confidence. Table 3 and Figure 5 show that participants using NaNofuzz, on average, reported higher confidence on the Likert scale (3.70/5.00) than when using Jest (3.08/5.00). The two-tailed Fisher's exact test shown in Table 4 (Measure 3) indicates the measured differences are statistically significant ($p < 0.03$). The ANOVA raises no statistically-significant alternative hypotheses.

(4) Task time. Out of 168 tasks (28 participants · 6 tasks per participant), 13 Jest tasks (15%) and 2 NaNofuzz tasks (2%) were stopped due to running out of time. Table 3 and Figure 5 show that participants usually completed tasks more quickly with NaNofuzz than with Jest. The two-tailed paired t-test shown in Table 5 indicates the differences in this measure are statistically significant ($p < 0.0001$). The ANOVA indicates that VS Code experience was positively correlated with task time ($p < 0.007$) on task 4.

(5) Tool usability. Figure 5 shows that 96% (27/28) of participants rated NaNofuzz (mean=87.86, SD=11.05) higher on the System Usability Scale [9] than Jest (mean=75.98, SD=14.47).

(6) Task realism. 86% (24/28) of participants indicated the tasks in this study were realistic.

(7) Tool comparison. Our inductive thematic analysis identified seven repeated themes in participants' qualitative statements about Jest and NaNofuzz. The themes are:

- **T1:** Automation can reduce human cognitive effort required for creating test cases.
- **T2:** Automation can reduce manual labor for creating tests.
- **T3:** Flexibility is valuable—when I need it.
- **T4:** I need to specify what correctness means.
- **T5:** Building a test suite can require iteration and exploration.
- **T6:** Understanding many test outputs helps me understand the program behavior and be more confident.
- **T7:** Intuitive tool design can reduce barriers.

We discuss the implications of these themes in the next section.

7 DISCUSSION

Our experiment investigated the three research questions introduced in Section 5—RQ1, RQ2, and RQ3. We now discuss how the answers to these questions are suggested by our results.

RQ1. Relative to standard practice, to what extent may NaNofuzz affect the number of bugs an engineer accurately identifies? NaNoFuzz improved the accuracy of bug identification. Table 3 shows that participants on average found 15% more bugs (Measure 1) and described bugs 30% more accurately (Measure 2) when using NaNofuzz than when using Jest.

However, task 6's large input domain made it less likely for NaNofuzz to randomly generate inputs that elicited task 6's two bugs. Of the participants who used NaNofuzz on task 6, 50% (7/14) elicited both bugs in the task, 21% (3/14) elicited one bug, and 29% (4/14) elicited no bugs at all. A successful strategy some participants adopted was narrowing NaNofuzz' input ranges to generate smaller matrices that were more likely to elicit the bugs. Still, three participants (P33, P43, P44) ran NaNofuzz multiple times, did not observe obvious bugs, and incorrectly concluded that no bugs were present. Finding unlikely buggy inputs is a common problem with fuzzers, and incorporating additional input generation guidance (e.g., code coverage) into NaNofuzz may support higher accuracy in these situations. Displaying or visualizing in the user interface how much of the input domain has been explored by NaNofuzz may help engineers better decide when to stop looking for bugs.

RQ2. Relative to standard practice, to what extent may NaNofuzz affect the engineer's confidence in the test activity? NaNofuzz improved engineers confidence. Table 3 shows that participants were on average 20% more confident (Measure 3) when using NaNofuzz than when using Jest.

The higher confidence that NaNofuzz instills could negatively affect accuracy in some cases. For instance, P14 quickly elicited the bug in task 1 using NaNofuzz and self-rated a high confidence level but then described the bug incompletely. This pattern repeats for P34 task 2 and P31 task 3. One solution may be for NaNofuzz to

Table 3: Quantitative Task Results Summary for Measures 1-4 from Table 1 (n = 28)

Tool	Task 1 Mean	Task 1 SD	Task 2 Mean	Task 2 SD	Task 3 Mean	Task 3 SD	Task 4 Mean	Task 4 SD	Task 5 Mean	Task 5 SD	Task 6 Mean	Task 6 SD	All Tasks Mean	All Tasks SD
Measure 1: Bugs elicited (1.00=all bugs found); higher is better														
Jest	0.93	0.27	**1.00**	0.00	1.00	0.00	1.00	0.00	0.36	0.50	0.50	0.39	0.80	0.38
NaNofuzz	**1.00**	0.00	0.93	0.27	1.00	0.00	1.00	0.00	**1.00**	0.00	**0.61**	0.45	**0.92**	0.25
Measure 2: Bug description accuracy (2.00=all bugs described accurately); higher is better														
Jest	1.29	0.99	1.71	0.47	1.71	0.61	1.64	0.63	0.21	0.43	0.50	0.44	1.18	0.43
NaNofuzz	**1.86**	0.36	**1.93**	0.27	**1.93**	0.27	**1.93**	0.27	**1.21**	0.80	**0.54**	0.54	**1.57**	0.69
Measure 3: Engineer confidence (Likert Scale); higher is better														
Jest	3.14	1.35	3.43	1.22	3.29	0.99	4.36	0.74	2.29	1.07	2.00	1.04	3.08	1.31
NaNofuzz	**4.21**	0.70	**4.07**	0.83	**4.14**	0.66	4.36	0.84	**3.07**	1.14	**2.36**	1.15	**3.70**	1.15
Measure 4: Time of test suite creation (elapsed seconds); lower is better														
Jest	585	224	526	199	622	214	297	113	585	255	628	200	540	230
NaNofuzz	**284**	139	**421**	150	**318**	118	**233**	77	**402**	199	**622**	222	**380**	199

Bold = Superior mean score

Table 4: Two-tailed Fisher's Exact Test for Measures 1-3 from Table 1

Measure 1	Jest	NaNo	Σ
bug elicited	74	86	160
bug not elicited	24	12	36
Σ	98	98	196

Measure 2	Jest	NaNo	Σ
accuracy = −1	1	1	2
accuracy = 0	20	6	26
accuracy = 1	19	15	34
accuracy = 2	44	61	105
accuracy = 3	0	1	1
Σ	84	84	168

Measure 3	Jest	NaNo	Σ
confidence = 1	13	6	19
confidence = 2	17	7	24
confidence = 3	16	14	30
confidence = 4	26	36	62
confidence = 5	12	21	33
Σ	84	84	168

Table 5: Two-tailed Paired t-test for Measure 4 from Table 1

Measure 4	Jest	NaNo
P12 / P15 time	59m 52s	42m 38s
P14 / P16 time	55m 32s	35m 26s
P18 / P19 time	61m 46s	47m 49s
P20 / P21 time	49m 23s	28m 58s
P22 / P24 time	42m 13s	33m 30s
P23 / P33 time	58m 37s	42m 39s
P25 / P27 time	58m 09s	43m 20s
P26 / P44 time	61m 57s	29m 44s
P30 / P45 time	48m 25s	38m 15s
P31 / P46 time	48m 44s	32m 45s
P32 / P34 time	46m 21s	42m 53s
P35 / P36 time	42m 40s	28m 52s
P37 / P47 time	59m 58s	46m 12s
P42 / P43 time	66m 53s	38m 42s
mean time	54m 19s	37m 59s

look for additional, adjacent examples of a likely bug once one is found—and then to summarize these findings for the engineer.

RQ3. Relative to standard practice, to what extent may NaNofuzz affect the engineer's time on the testing task? NaNofuzz sped up testing tasks. Table 6 shows participants on average completed tasks 30% faster with NaNofuzz than with Jest.

Why did NaNofuzz show positive results? When did participants encounter problems using NaNofuzz? Our participants compared both treatments in the post-survey, and we structure the remainder of this section using the seven themes presented in Section 6 and then we discuss some our own thoughts.

T1: Automation can reduce human cognitive effort required for creating test cases. The process of testing involves a number of tasks that are cognitively difficult: to create a test case in Jest, the engineer must think of inputs to test, and then determine what the appropriate output might be. By automatically generating inputs and providing a categorized list of test results for the engineer to choose from, NaNofuzz may benefit from the advantages of recognition over recall [44] to improve **task time (RQ3)**.

"[NaNofuzz] is really useful for surfacing edge cases without me needing to think of them." (P45.Q15)

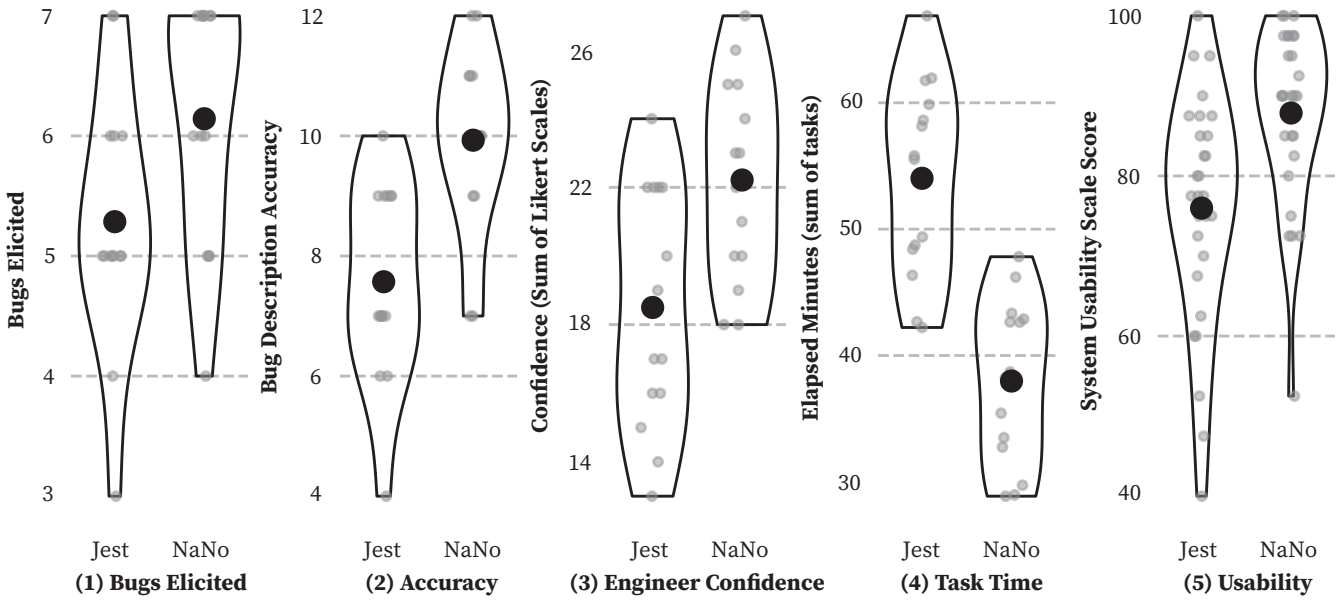

Figure 5: Violin plots showing the distribution of results by treatment for Measures 1-5. Y-axis for Measures 1-4 shows the sum of six tasks for paired participants. Y-axis for Measure 5 is the SUS [9] score range. The mean is indicated by a black dot. Grey dots indicate observations. Wider areas of the plot indicate a larger number of observations.

"[With Jest] if I didn't know what was wrong with the program I had to sit and stare at the source code or maybe try a few random guess inputs to get unstuck." (P25.Q11)

T2: Automation can reduce manual labor for creating tests. Manual test generation tools such as Jest often require significant manual labor to type up each test case, including repetitive boilerplate testing code that can result in a low signal-to-noise ratio. For example, Figure 4 shows a ten line Jest test case, but only half the lines contain any meaningful information for testing. While these burdens might seem small, participants noticed this difference, which may contribute to improved **task time (RQ3)**.

"[Jest] can be most cumbersome to use when there are a lot of similar edge cases that need to be tested, providing those edge cases can take a lot of effort to set up vs [NaNofuzz] making them for you." (P25.Q16)

"[NaNofuzz] helped me enumerate a large set of cases quickly. It would be useful in scenarios where state space explosion is relevant and the behavior of the program is complex and unpredictable." (P34.Q11)

T3: Flexibility is valuable—when I need it. No tool is always best. Some participants said they would use both tools when testing.

"Would use both in combination" (P22.Q11.3)

"I like [NaNofuzz] more, but still think that both are better situationally." (P30.Q11)

T4: I need to specify what correctness means. NaNofuzz in the form we evaluated does not allow the engineer to specify an explicit oracle such that the PUT's correctness relative to a specification may be evaluated. Some participants noticed this limitation and suggested it be addressed in future versions of NaNofuzz.

"The option to enforce a ground truth value helps to check both the undefined, NaN, etc errors as well as logic errors." (P35.Q11)

"Jest makes way more sense to test exact semantics one cares about." (P14.Q15)

T5: Building a test suite can require iteration and exploration. Traditional tools like Jest that require significant manual effort to operate can make it difficult to explore a PUT's behavior during testing, especially with a new or unfamiliar program. NaNofuzz' automation supports exploration of the relationship among a PUT's inputs and outputs, which may contribute to NaNofuzz' improved performance on **bugs accurately identified (RQ1)**.

"[NaNofuzz] was useful in dynamically verifying my assumptions and exploring options I wasn't aware of" (P22.Q11.2)

"[W]hen I wanted to test a wide range of values or wanted to do some quick overview [NaNofuzz] was more useful" (P21.Q13.2)

T6: Understanding many test outputs helps me understand the program behavior and be more confident. Simply having more test results available made many participants feel more confident in their testing, either because they better-understood the function's behavior or because the higher number of test cases was more likely to find unexpected errors. These aspects may contribute to NaNofuzz' improved performance on **engineer confidence (RQ2)**.

"I was not certain I had generated enough inputs manually using Jest." (P16.Q11)

"[NaNofuzz gave me] confidence that I had exercised lots of possible inputs for the functions under test. With Jest I felt like I was fishing around trying to find degenerate cases." (P26.Q11)

T7: Intuitive tool design can reduce barriers. We saw in Section 6 that participants rated NaNofuzz as more usable on the System Usability Scale, and the free-form feedback provides insight into some reasons that NaNofuzz might be considered more usable. For instance, NaNofuzz outputs have a higher signal-to-noise ratio than Jest, and NaNofuzz only requires minimal information to test while Jest expects engineers to write testing code, much of which is boilerplate.

"[NaNofuzz] had a really simple UI while Jest displayed a lot of information that wasn't necessary for me to understand how to go about fixing the output or figuring out what inputs were wrong." (P08.Q13)

"It was easier to click [NaNofuzz'] test button and then just change the input parameters." (P20.Q15)

8 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. We found that this limit stopped participants in 15 tasks (2 with NaNofuzz, 13 with Jest), or 9% of the total tasks encountered. Of the stopped Jest tasks, 4 received a perfect score, so it is not reasonable to expect that the time limit reduced these scores. Given this limit affected Jest tasks more often than NaNofuzz tasks, it is possible that accuracy for 9 Jest tasks might be higher had no time limit been imposed. Despite using an unambiguous rubric, it is possible that human error reduced the reliability of RQ1's scoring. To address this risk, the third author randomly sampled four sessions and independently scored them by re-watching the recorded session and agreed with the original scoring. While our study design randomized treatment sequence, the individual task programs were presented in a fixed order, which did not control for fatigue effects. As our study design required participants to use both tools, it is possible that effects from this combination of tools are not controlled. For instance, engineers may have been more confident on Jest tasks if they exclusively used Jest without seeing NaNofuzz.

External Validity. While we adopted Ko et al.'s [34] recommendation to use "found" tasks to improve task realism, it is possible that the tasks we selected from GitHub, Rosetta Code, and Stack Overflow are neither realistic nor representative. During the post-survey, we asked participants to what extent the tasks they encountered were realistic, and 86% (24/28) indicated the study tasks were similar to tasks they might encounter while programming 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 NaNofuzz prototype used in this study possesses capability necessary to evaluate the hypothesis but lacks some important features that we discuss in Section 4. Tools with more functionality may be harder to use [41], and NaNofuzz' present simplicity may provide a different effect than that of a more-comprehensive tool. The bugs in our tasks were elicited using the same implicit oracle, which is not the case with many testing tasks. Due to this limitation, we propose in Section 10 human studies of gradual oracle specification using a future and more feature-rich version of NaNofuzz. During the participant sessions, we did not model some aspects of industrial software engineering such as interruptions, since that might introduce variance that would be difficult to control and that may obscure the effect we intended to measure with a limited number of participants.

Construct Validity. **Quality (RQ1)** investigates the number of bugs an engineer accurately identifies while generating the test suite, and is measured in two parts. **RQ1.1** measures how many known bugs the test suite elicits, which is an important quality component: test suites that find actual bugs may be considered higher quality. **RQ1.2** measures the quality of the testing activity by assessing the extent to which an engineer can 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. **Satisfaction (RQ2)** investigates the participant's confidence in the testing activity. Sadowski and Zimmermann [58] explain that a software engineer's "satisfaction may be impacted by the real or perceived effectiveness of their personal work." We measure self-reported confidence as a proxy variable for satisfaction on each given task, but it is not clear to what extent an engineer's confidence may relate to overall satisfaction with the testing activity. **Task time** (**RQ3**) investigates how quickly a software engineer may generate a test suite. This was measured as time elapsed from the beginning to the end of the task.

9 RELATED WORK

Anand et al. [3] stated that "test case generation is among the most labour-intensive tasks in software testing." Yet, test suites are often created manually [4, 21, 24, 35] despite decades of prior work to automate aspects of test suite generation.

The Randoop [48] and IntelliTest [39] fuzzers are similar to NaNofuzz in that they use an implicit oracle and generate persistent test cases for a test suite. Like NaNofuzz, IntelliTest is integrated into an IDE. NaNofuzz differs by prioritizing speed and ease of use over code coverage; further, NaNofuzz was evaluated with humans in a randomized controlled trial.

Mutation testing originated in the 1970s [10, 18, 19, 32, 46]. DeMillo and Offutt [20] described adding mutation testing to the Mothra and Godzilla testing systems and reported promising lab results. Subsequent improvements were reported (e.g., [31, 50]); yet, mutation testing often uses the PUT as its oracle, so the tests these tools generate neither detect bugs that presently exist nor assert correctness. NaNofuzz differs from mutation testing tools by not mutating the PUT and by using an implicit oracle that detects common types of bugs. Like mutation testing tools, NaNofuzz is unable to evaluate a PUT's correctness relative to a specification.

Ahlgren et al. [1] and Bornholt et al. [7] provided evidence that metamorphic test generation tools continue to find important use

cases, particularly in problems where an oracle may be unknowable or complex. Segura et al. [59] surveyed the metamorphic testing literature, but excludes related work on property-based testing (e.g., QuickCheck [11] and its progeny). However, many engineers find such testing tools difficult to use, and an important theme of Bornholt et al. [7] was that removing usability barriers was important for engineer adoption. Goldstein et al. [27] provided further evidence that practitioners find that property-based testing tools have significant usability barriers, which may be limiting the application of metamorphic testing tools to an artificially-small set of use cases. NaNofuzz differs from these tools in that it uses an implicit oracle and does not currently allow an engineer to assert additional relationships between inputs and outputs.

Rothermel et al. [54, 55, 56] performed foundational work in testing spreadsheets and evaluated "What You See Is What You Test" (WYSIWYT), a usable testing interface for end-users that did not require writing code. Fisher et al. [22, 23] expanded this work by creating "Help Me Test" (HMT), an ATUG tool for spreadsheets that used random and search-based techniques. Similar to NaNofuzz, 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 EvoSuite, a state-of-the-art search-based ATUG tool, with human software engineers and reported that human software engineers found fewer bugs with EvoSuite, despite the tool generating test suites that have higher code coverage. Like mutation testing tools, EvoSuite uses the PUT as the oracle, so its tests neither detect current bugs nor assert correctness. The study used qualitative data to suggest that poor readability of test cases represented as code may be partially at fault for its negative result. This finding gave focus to recent research that aimed to improve test case readability (e.g., [28, 42, 49, 57, 60]). Unlike EvoSuite, NaNofuzz detects bugs in current programs and has shown positive results relative to manual test generation in a randomized human trial with professional software engineers.

Ng et al. [43] observed that one of the top barriers reported for automatic testing tools is their difficulty of use. Li et al. [37] called for improved usability of random testing tools. Prado and Vincenzi [51] and Arcuri [4] observed that ATUG tools are relatively unused in industry and that tool designers often prioritize technical or secondary measures over the an engineer's productivity. Rojas et al. [52] 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. We designed NaNofuzz in reaction to the findings of the prior work.

10 IMPLICATIONS AND FUTURE WORK

This study's findings imply that empirical software engineering researchers may achieve more impactful results by prioritizing usability aspects such as velocity and satisfaction *in addition to* test suite quality when designing or evaluating ATUG tools.

This study provides important evidence that fuzzing can provide productivity benefits to software engineers, even with a simple stochastic opaque-box algorithm such as the one used by NaNofuzz. More sophisticated fuzzers may provide *further* benefits if provided to engineers in a usable way within a development environment.

Reducing the number of tests that need to be created manually might also allow engineers to re-direct efforts towards testing software more rigorously than may be practical today. Additionally, end-user developers are often ignored by ATUG tool designers, but these developers represent a large user base that is particularly sensitive to the benefit received (e.g., bugs found relative to time invested) [33]. Due to this aspect, usable ATUG tools may also have a great impact on the quality of end-user developed software.

We have released NaNofuzz to the Visual Studio Code Marketplace [14] and plan to use it in real-world situations outside our study. As these situations may require testing with explicit oracles, we plan to investigate how NaNofuzz might adopt aspects of testing tools that allow specifying an explicit oracle. This extension of NaNofuzz was also suggested by participants (Section 6, T4). Usability might vary with additional feature complexity [41], and it is important to explore how additional complexity may affect engineer productivity [58] when building a test suite while simultaneously refining an explicit oracle. Additional formative human studies on engineers using metamorphic testing tools may provide insights into the barriers these engineers encounter and help researchers identify human-centered solutions to these barriers. Future versions of NaNofuzz may also support *other* IDEs and *other* languages beyond Visual Studio Code and TypeScript.

11 CONCLUSIONS

This paper presents NaNofuzz, a usable automatic test generation tool that runs within an engineer's IDE and offers a simple set of interactions for generating a test suite. NaNofuzz provides automation support for finding edge cases, generating test cases for a test suite, and categorizing test results. We evaluated NaNofuzz in a randomized controlled human trial with 28 professional software engineers using Jest as the control treatment. Participants using NaNofuzz on average identified bugs more accurately ($p < .05$, by 30%), were more confident in their tests ($p < .03$, by 20%), and finished their tasks more quickly ($p < .007$, by 30%). Given the estimated $47 billion USD annual cost of testing in the United States, these findings suggest that prioritizing testing tool usability may lead to significant productivity gains for software engineers, as well as allow for more-rigorous software testing. We hope that this study motivates further research into usable test suite generation tools that help engineers efficiently and confidently generate effective test suites, along with appropriate evaluations of their success.

12 DATA AVAILABILITY

The study data, analysis, tasks, participant demographics, and protocol are submitted with this paper as supplementary material [17] and via our study repository [16]. NaNofuzz is MIT-licensed and available via GitHub [15] and the VS Code Marketplace [14].

ACKNOWLEDGMENTS

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

REFERENCES

[1] John Ahlgren, Maria Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Erik Meijer, Silvia Sapora, and Justin Spahr-Summers. 2021. Testing Web Enabled Simulation at Scale Using Metamorphic Testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 140–149. https://doi.org/10.1109/ICSE-SEIP52600.2021.00023

[2] 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. https://doi.org/10.1145/3183440.3195001

[3] 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

[4] 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. https://doi.org/10.1007/s10664-017-9570-9

[5] 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

[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. https://doi.org/10.1109/TSE.2017.2776152

[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. https://doi.org/10.1191/1478088706qp063oa

[9] John Brooke et al. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7. https://doi.org/10.1201/9781498710411

[10] Timothy A Budd, Richard J Lipton, Richard DeMillo, and Frederick Sayward. 1978. The design of a prototype mutation system for program testing. In *Managing Requirements Knowledge, International Workshop on*. IEEE Computer Society, 623–623. https://doi.org/10.1109/AFIPS.1978.195

[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. https://doi.org/10.1145/351240.351266

[12] Anastasia Danilova. 2022. *How to Conduct Security Studies with Software Developers*. Ph. D. Dissertation. Universitäts-und Landesbibliothek Bonn. https://hdl.handle.net/20.500.11811/10063

[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, and Sam Estep. 2022. NaNofuzz - Visual Studio Marketplace. https://marketplace.visualstudio.com/items?itemName=penrose.nanofuzz. [Online; accessed 2022-11-20].

[15] Matthew C Davis, Sangheon Choi, and Sam Estep. 2022. NaNofuzz: a fast and easy-to-use automatic test suite generator for Typescript that runs inside VS Code. https://github.com/nanofuzz/nanofuzz. [Online; accessed 2022-11-20].

[16] Matthew C Davis, Sangheon Choi, and Sam Estep. 2022. nanofuzz-study. https://github.com/nanofuzz/nanofuzz-study.

[17] Matthew C Davis, Sangheon Choi, and Sam Estep. 2023. Reproduction Package for Article "NaNofuzz: A Usable Test Suite Generation Tool". https://doi.org/10.1145/3580413

[18] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41. https://doi.org/10.1109/C-M.1978.218136 Conference Name: Computer.

[19] Richard A DeMillo. 1989. Completely validated software: Test adequacy and program mutation (panel session). In *Proceedings of the 11th International Conference on Software engineering*. 355–356. https://doi.org/10.1145/74587.74634

[20] Richard A DeMillo and A Jefferson Offutt. 1993. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2, 2 (1993), 109–127. https://doi.org/10.1145/151257.151258

[21] 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

[22] 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. https://doi.org/10.1145/581339.581359

[23] 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. https://doi.org/10.1145/1131421.1131423

[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. https://doi.org/10.1145/2699688

[25] Hannah Frith and Kate Gleeson. 2004. Clothing and embodiment: Men managing body image and appearance. *Psychology of men & masculinity* 5, 1 (2004), 40. https://doi.org/10.1037/1524-9220.5.1.40

[26] GitHub. 2022. GitHub Codespaces. https://github.com/features/codespaces. [Online; accessed 2022-11-21].

[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] 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. https://doi.org/10.1145/3196321.3196363

[29] William E Howden. 1978. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering* 4 (1978), 293–298. https://doi.org/10.1109/TSE.1978.231514

[30] Pankaj Jalote. 2008. *A concise introduction to software engineering*. Springer Science & Business Media. https://doi.org/10.1007/978-1-84800-302-6

[31] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678. https://doi.org/10.1109/TSE.2010.62

[32] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sept. 2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[33] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad A Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The State of the Art in End-User Software Engineering. *ACM Comput. Surv.* 43, 3, Article 21 (apr 2011), 44 pages. https://doi.org/10.1145/1922649.1922658

[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. https://doi.org/10.1007/s10664-013-9279-3

[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. https://doi.org/10.1109/QSIC.2014.33

[36] J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (1977), 159–174. https://doi.org/10.2307/2529310 Publisher: [Wiley, International Biometric Society].

[37] 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*. USENIX Association, 2777–2794.

[38] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[39] Microsoft. 2023. Overview of Microsoft IntelliTester. https://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual/. [Online; accessed 2023-01-27].

[40] Barton P Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44. https://doi.org/10.1145/96267.96279

[41] Brad A Myers. 1994. Challenges of HCI Design and Implementation. *Interactions* 1, 1 (jan 1994), 73–83. https://doi.org/10.1145/174800.174808

[42] Mathieu Nassif, Alexa Hernandez, Ashvitha Sridharan, and Martin P Robillard. 2022. Generating unit tests for documentation. *IEEE Transactions on Software Engineering* 48, 9 (2022), 3268–3279. https://doi.org/10.1109/TSE.2021.3087087

[43] 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. https://doi.org/10.1109/ASWEC.2004.1290464

[44] Jakob Nielsen. 1994. Enhancing the Explanatory Power of Usability Heuristics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, Massachusetts, USA) *(CHI '94)*. Association for Computing Machinery, New York, NY, USA, 152–158. https://doi.org/10.1145/191666.191729

[45] Bureau of Labor Statistics. 2022. Occupational Outlook Handbook, Software Developers, Quality Assurance Analysts, and Testers. https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm. [Online; accessed 2022-10-06].

[46] Jeff Offutt. 2011. A mutation carol: Past, present and future. *Information and Software Technology* 53, 10 (Oct. 2011), 1098–1107. https://doi.org/10.1016/j.infsof.2011.03.007

[47] Gerard O'Regan. 2019. *Fundamentals of Software Testing.* Springer International Publishing, 59–78. https://doi.org/10.1007/978-3-030-28494-7_3

[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* (Austin, Texas) *(ICSE '16).* Association for Computing Machinery, New York, NY, USA, 547–558. https://doi.org/10.1145/2884781.2884847

[50] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. 2019. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software* 157 (2019), 110388. https://doi.org/10.1016/j.jss.2019.07.100

[51] 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

[52] 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* (Baltimore, MD, USA) *(ISSTA 2015).* Association for Computing Machinery, New York, NY, USA, 338–349. https://doi.org/10.1145/2771783.2771801

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

[54] 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

[55] 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.

[56] Karen J. Rothermel, Curtis R. Cook, Margaret M. Burnett, Justin Schonfeld, T. R. G. Green, and Gregg Rothermel. 2000. WYSIWYT Testing in the Spreadsheet Paradigm: An Empirical Evaluation. In *Proceedings of the 22nd International Conference on Software Engineering* (Limerick, Ireland) *(ICSE '00).* Association for Computing Machinery, New York, NY, USA, 230–239. https://doi.org/10.1145/337180.337206

[57] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. 2021. DeepTC-Enhancer: Improving the Readability of Automatically Generated Tests. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20).* Association for Computing Machinery, New York, NY, USA, 287–298. https://doi.org/10.1145/3324884.3416622

[58] Caitlin Sadowski and Thomas Zimmermann. 2019. *Rethinking productivity in software engineering.* Springer Nature. https://doi.org/10.1007/978-1-4842-4221-6

[59] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875

[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/

[62] Facebook Open Source. 2022. Jest - Delightful Javascript Testing. https://jestjs.io/. [Online; accessed 2022-11-08].

[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. https://doi.org/10.1002/9780470382844

[65] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.* 29, 4 (dec 1997), 366–427. https://doi.org/10.1145/267580.267590

Received 2023-02-02; accepted 2023-07-27