Lucid Property-Based Testing

Supercharging Sensemaking of Plentiful Program Executions

Anonymous Author(s)

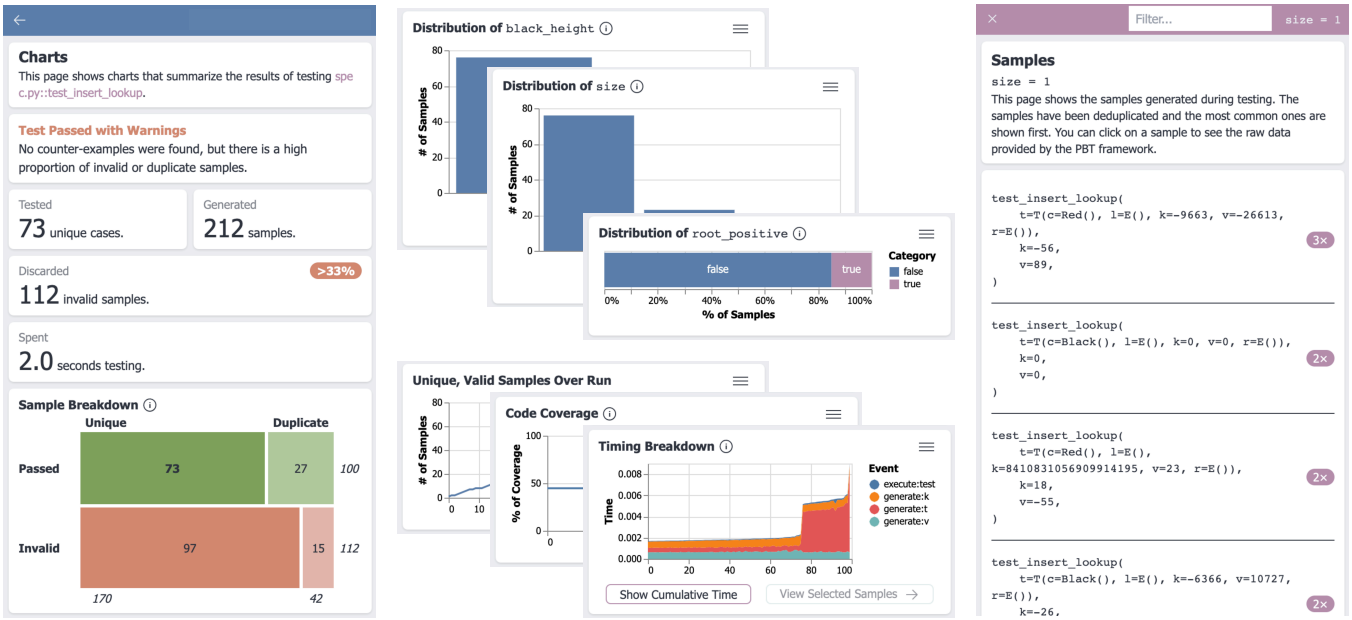


Figure 1: The Lucid interface for visual property-based testing.

ABSTRACT

Software developers increasingly rely on significant automation to assess the correctness of their code. One such method is property-based testing (PBT), wherein a test harness generates hundreds of inputs and checks the output of the program on those inputs using parametric properties. While powerful, PBT induces a sizable gulf of evaluation—developers need to put in the effort to understand what happened over hundreds of executions of their software in order to understand the thoroughness of their testing. To cross this gulf, we propose *lucid property-based testing*, an interaction paradigm to support sensemaking for PBT effectiveness. Following a formative design study, we design Lucid, an interface that supports lucid PBT with interactive, configurable views of test behavior with tight integrations into modern developer testing workflow. Lucid displays aspects of the test input distribution, code coverage metrics, timing information, and more. To accelerate the development of powerful, interactive PBT, we define a standard for PBT test reporting and integrate it into the most widely used PBT library. An online self-guided usability study revealed that use of Lucid's views leads to better-calibrated assessments of software testing effectiveness.

UIST '24, October 13–October 16, 2024, Pittsburgh, PA, USA
2024.

KEYWORDS

randomized testing, visual feedback, explorability, multiple executions, standards

ACM Reference Format:
Anonymous Author(s). 2024. Lucid Property-Based Testing: Supercharging Sensemaking of Plentiful Program Executions. In *The 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24), October 13–October 16, 2024, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 18 pages.

1 INTRODUCTION

Software developers work hard to build systems that do their jobs as intended. But getting things exactly right on the first go is rare, so developers also write tests to validate their work, flush out bugs, and check that past bugs stay fixed. Traditionally, these tests take the form of manually written "example-based" tests, where developers write out specific sample inputs together with the expected outputs, but this process is labor intensive, scales poorly as software gets more complex, and often misses subtle bugs because developers have trouble imagining the full range of inputs that are needed to thoroughly test their systems. As an alternative, developers have started reaching for automated approaches to testing to supplement or replace manual testing. In this paper, we focus on an increasingly popular form of automated testing known as *property-based testing*. PBT automatically samples many program inputs from some random distribution and checks, for each one, that the system's behavior satisfies a set of developer-provided *properties*.

The upshot—ideally—is testing that is both more thorough and less laborious than conventional methods.

Experience in both academic and industrial settings has established PBT's effectiveness at identifying subtle bugs in a wide variety of real-world settings, including telecommunications software [3], replicated file and key-value stores [7, 45], automotive software [4], and other complex systems [44]. And PBT remains an active area of interest in the programming languages and testing communities [19, 30, 32, 56, 75], which are constantly improving the effectiveness and versatility of PBT tools.

Of course, automation often presents trade-offs, and PBT is no exception. Studies by researchers in and around the HCI community [28, 29] have identified limitations of PBT tools that impact their usability in practice. One such trade-off might be called *lucidity*: in traditional example-based testing, developers can see and understand every input that is used to test the program, but in PBT there are simply too many test inputs to examine every one. Without a clear idea of which inputs were used to test the code, developers may not notice when the test suite is entirely missing critical cases and therefore not actually testing the situations they thought they were testing.

For example, suppose the programmer is testing a simple mathematical function on floating point numbers. The inputs may lie in a range that is too small (e.g., between `0` and `1`, even though the function can take any float) or too large (e.g., all floats, even though the function can only take positive ones). Or the inputs may simply be so poorly distributed that hitting important corner-cases like `0` and `-1` is very unlikely. Even worse, the data generator may produce a large amount of malformed or invalid data (e.g., `NaN`) that cannot be used for testing at all. In state-of-the-art PBT tooling, any of these erroneous situations could go unnoticed because necessary information is hidden within the operation of the framework.

This lucidity gap offers a challenge and an opportunity for user interface design. The raw information that the developer needs is theoretically available, since the PBT framework can in principle save a log of everything that happens during testing, but developers are unlikely to get much value out of such logs without help. Luckily, there is a rich tradition of user interfaces that help developers make sense of complex data during the software development process. Some existing tools help developers to understand a single execution of a single program [50, 71, 86] and others help developers to understand multiple related programs at once [24, 26, 92]. With PBT as our motivation, it is now critical that we tackle the problem of understanding multiple executions of the same program.

To characterize this new domain of program understanding and address the aforementioned UI concerns for PBT, we propose a new paradigm of *lucid property-based testing* (LPBT). In LPBT, logs of each test execution are saved, aggregated, analyzed, and presented to the developer to aid them in the testing process. We demonstrate the vision of LPBT with Lucid, a cutting edge PBT interface that embodies the principles of LPBT. Lucid provides aggregate statistics (e.g., the number of tests executed versus the number that were rejected as malformed), visualizations of aspects the data distribution (e.g., bar charts showing the sizes of the test inputs), timing information, and more (see Figure 1), all of which serve to help developers understand if their property-based tests are thoroughly testing their program. To support Lucid and encourage its broad adoption, we also propose OpenLPBT, an open format for describing PBT results. OpenLPBT creates a consistent way for a wide range of PBT frameworks to report data that was previously hidden, and it opens the door for more tools like Lucid that serve other developer needs. Together, these technologies provide developers with the lucidity they need to accurately evaluate their PBT, and the lucid property-based testing paradigm suggests a recipe that could improve other forms of automated testing as well.

After discussing background (§2) and related work (§3), we present the following contributions:

- We articulate the core principles of *lucid property-based testing* (LPBT), motivated a formative interview study with experienced PBT users. (§4)
- We present the design of Lucid, a flexible web-based interface that implements the LPBT principles and helps developers evaluate the quality of their testing. (§5)
- We describe the implementation details of Lucid. In particular, we detail the OpenLPBT format for collecting PBT data, which we define to help standardize LPBT tools across different PBT frameworks. (§6)
- We evaluate Lucid, and by extension elements of the LPBT paradigm, in an online study, and we show that Lucid is significantly better than state-of-the-art tools at helping users to distinguish between good and bad testing setups. (§7)

We conclude with directions for future work (§8), including some other automated testing disciplines that should be made "lucid."

2 BACKGROUND

In this section we give an overview of property-based testing as a whole (§2.1), describe challenges that PBT users face (§2.2), and describe the tools that PBT frameworks currently make available for understanding effectiveness (§2.3).

2.1 Property-Based Testing

In standard unit testing, developers come up with examples that demonstrate the way a function is supposed to behave, and they write one test for each. For example, this unit test checks that inserting a key `"key"` and value `42` into a red–black tree [1] and then looking up `"key"` results in the value `42`:

```
def test_insert_example():
    assert lookup(
      insert("key", 42, Empty()),
      "key"
    ) == 42
```

One could painstakingly write dozens of tests like this, checking many different example trees, keys, and values, but PBT gives a better way:

```
@given(trees(), integers(), integers())
def test_insert_lookup(t, k, v):
    assume(is_red_black_tree(t))
    assert lookup(insert(k, v, t), k) == v
```

This test, written in Hypothesis [60], a popular PBT library in Python, randomly chooses trees, keys, and values and uses those to check the same property that was tested in the original example.

[1] https://en.wikipedia.org/wiki/Red-black_tree

This single test replaces hundreds or thousands of standard unit tests, and it can actually be more thorough if the random number generator runs into examples the user might not have thought of.

2.2 Testing Process and Pitfalls

While PBT is often much more effective than standard unit testing, it is not always easy to get right. For example, if the `trees()` strategy in the above test only produces the `Empty()` tree, the test would be scarcely more useful than `test_input_example`. More subtly, the test includes the line `assume(is_red_black_tree(t))` which filters out any trees that are not valid red–black trees; if the `trees()` strategy is written naïvely, it may fail to produce inputs satisfying the filter. [2] In either of these cases, the test may appear to be thoroughly testing the red–black tree implementation, when, in fact, it is barely testing anything at all.

Recent work has shown that these pitfalls hamper real users. In *Property-Based Testing in Practice* Goldstein et al. [28] highlight the cost of limited lucidity in the PBT process. Some developers in their study described missing important bugs due to inadequate data generators: their test passed, so they shipped code into production, but the test was actually trivial because all of the values generated were too small to trigger the bug. Other developers had not experienced such a dramatic issue, but they still complained that they did not trust their test harnesses because they did not know what was happening during the test.

The pitfalls of inadequate random data generators are well known to the PBT research community . Modern research into PBT has made significant progress toward automatically providing developers with high-quality generation strategies that thoroughly test a given property [32, 57, 66, 84, etc.]. But in practice, the currently available automation is incomplete and inconsistent. Participants in *PBT in Practice* suggested that a good place to start would be to make testing feedback more readily available to the users. More than a third of study participants "said they did not think as hard as they should about testing effectiveness." Ultimately, developers still need visibility into exactly what is happening during the testing process.

Testing is an activity that takes up a great deal of developers' time; recent estimates include that it takes up 8% of a developer's day [64], and a quarter of the time in the IDE [5]. Testing often occurs in spurts—that is, developers may go long periods without doing it, but when they do it, they do it extensively [5]. Lucid is built for this kind of usage—that is, focused usage during a focused testing session.

The reason for techniques such as property-based testing is to be able to detect bugs that developers would not find by authoring one-off test examples. One of the obstacles to effective testing is that software testing is subject to confirmatory bias—that is, developers often prematurely assume their software is correct. Developer may be tempted to make such assumptions when under time pressure [79]. That said, the presence of a failing test, or even a test suite that feels like it *shouldn't* pass but does, can lead developers to be more suspicious of their software's correctness [79]. We see the role of Lucid as addressing time pressure of testing with more efficient inspection mechanisms, while easing the cost of being appropriately critical of one's test suite and thus overcoming assumptions that one's software is correct.

Achieving assurance about the behavior of code is difficult. Developers seem to generally agree that good tests should cover requirements for the software, check both normal and exceptional flow, and should try to break functionality to find bugs [52]. One of the common methods for assessing the quality of a test suite is code coverage, and in particular, how much of a program has been exercised by tests. Coverage often fails to find faults in programs [37] and, what is more, may even lead developers to overinflated confidence when writing tests without meaningful changes in testing effectiveness [58]. The design of Lucid attempts to speed up developers' ability to increase the exceptional cases that will break functionality, in particular by providing visibility into the inputs to the program and the uniqueness of those inputs.

What parts of the testing process does Lucid support? Enoiu et al. [20] propose an 8-stage model of testing that derives from classical models of problem-solving. Within this model, Lucid primarily supports developers in monitoring the progress toward a testing goal, and evaluating the result of their testing. We also anticipate that if Lucid leads a developer to realizing that their testing goal is not met, it provides clarifying information that can help the developer redefine their goals, analyze knowledge of the system and test harness, and reform their testing strategy.

2.3 Displays of PBT Output

We briefly survey the five most popular[3] PBT frameworks—Python's Hypothesis [60], TypeScript / JavaScript's fast-check [18], Rust's quickcheck [23], Scala's ScalaCheck [70], and Java's junit-quickcheck [42]—and analyze the evaluation information available in each. We also include Haskell's QuickCheck [13] as a baseline, since it was the original PBT framework.

- *Raw Examples* All frameworks surveyed had some way of dumping all generated inputs to the terminal for inspection. Some (3/6) frameworks provide this feature explicitly (e.g., via a "verbose" flag), others require users to expose the examples themselves, e.g., by printing them to the terminal.
- *Number of Tests Run vs. Discarded* Many frameworks (4/6) report counts of examples run or discarded as part of the test output. Some (2/6) provide this information by default, while others (2/6) require running in verbose mode. Surprisingly, neither fast-check nor Rust's quickcheck seemed to offer any way to report these counts.
- *Event / Label Aggregation* Many frameworks (4/6) also provided ways to emit and aggregate features of the data distribution during property execution. Users can specify strings that should be collected during test execution, and then the aggregate data is presented on the command line via a simple table:[4]

[2] This kind of filtering may also impact the test distribution. Imagine the developer carefully constructs a distribution of trees and then those trees get filtered to only the valid red–black ones. The resulting distribution will be biased towards *much* smaller tree since a higher proportion of small lists are valid.

[3] We used GitHub stars to gauge popularity; technically jsverify has more stars than junit-quickcheck, but we opted to not duplicate host languages.

[4] Borrowed from https://hackage.haskell.org/package/QuickCheck-2.14.3/docs/Test-QuickCheck.html#v:label.

```
7% length of input is 7
6% length of input is 3
5% length of input is 4
4% length of input is 6
...
```

- *Generation Time* Only Hypothesis provides some information about exactly how long (in wall-clock time) the data generation process took.
- *Warnings* Hypothesis implements warnings that tell users if their generation strategies are producing a very high proportion of discarded values.
-

The full breakdown of available feedback is shown in Appendix A.

Overall, the affordances for evaluation that exist in frameworks PBT are inconsistent. Some frameworks, for example those that aim to directly clone Haskell's QuickCheck, have a relatively consistent approach, but others may provide almost no support for understanding data distributions. Evaluation information is also inconsistently shown or hidden; sometimes information is printed to the command line by default, while other times it requires some real digging to obtain. The available information also varies depending on exactly how the tests are executed: when Hypothesis tests are run with pytest [53], the pytest interface swallows useful Hypothesis messages by default. In §5 we will discuss how our paradigm of LPBT improves on this current landscape.

3 RELATED WORK

3.1 Interactive Tools for Testing

There is a long history of testing interfaces in HCI and SE research. Some of the seminal work is in the end-user programming field, where tools have been developed to help programmers perform lightweight validation through data checking, or "What you see is what you test"—this paradigm has been explored for spreadsheets [77], visual dataflow languages [49], and screen transition languages [8]. This testing methodology has been extended with features that suggest values for a user to test, and encourage curiosity by first detecting, and then calling users' attention to, anomolies [65, 91]. These features are a useful part of the testing toolkit; WYSIWYT tends to be helpful for producing example-based tests (where we focus on randomized tests); features for anomaly detection could, if suitably robust, could be made part of a tool like Lucid.

UIs can be developed to facilitate automated testing. Sikuli [11], for instance, allows for the creation of GUI tests by capturing user actions and creating automation for using computer vision to check the behavior of GUIs as actions are automated. NanoFuzz [14], similarly to our work, displays many outputs coming from a fuzzer. Its views are quite similar to the example-based views we surface as our control, over which we show a major improvement, likely due to the bespoke visualizations that convey high-level information about output distributions and overall test success in a way that is helpful to users.

The UIST community has developed myriad tools to aid manual testing, that is, manual inspection of program behavior to determine if it is doing the right thing. Systems have been developed to support the testing of web applications [9, 72], circuit boards [16], and embedded systems application [34, 62, 63]. These systems typically work by letting a programmer record a program execution; their power often arises from exposing information about a program that is lacking in available profilers (e.g., signals on input and output pins of a circuit, network messages). These systems sometimes offer checkers that scan over profiled information to search for bugs [16, 62, 63] (e.g., particular classes of network failures, or wiring bugs). Where this prior work has largely focused on accelerating manual testing, ours focuses on the first steps to accelerating automated testing, and particular automated testing where the strongest signal of test success is the distribution of tested inputs.

Fuzz testing of "fuzzing" is closely related to PBT[5], and the fuzzing community has some interesting visual approaches to testing feedback. One of the most popular fuzzing tools, AFL++ [22], includes a sophisticated terminal user-interface with feedback on code coverage and other fuzzing statistics over time. This information is undeniably richer than what many PBT frameworks provide, but it does not give the level of insights that LPBT tools like Lucid provide.

NaNoFuzz [14] re-imagines the random testing process with the user in the loop, showing them examples of program outputs and helping them to notice problematic results like `NaN` or a crash failure. NaNoFuzz is superficially the closest comparison available for Lucid, but the two serve very different purposes. Whereas NaNoFuzz proposes a new paradigm around random testing that is ideal for those without an established workflow, the LPBT principle of **workflow integration** (see §4.3) led to a design of Lucid that builds on existing PBT workflows and helps developers to improve testing infrastructure that they already have. That said, we think NaNoFuzz and Lucid could actually work together quite well: NaNoFuzz gives great feedback on program outputs, but it still hides important insights from the user that Lucid could highlight. One could imagine a user finding a few bugs with NaNoFuzz, then loading up Lucid and realizing that there was an unexplored portion of the input space, swapping back to NaNoFuzz to find a few more bugs, etc.

TODO

- Tarantula

3.2 Making Sense of Program Executions

The purpose of Lucid is to help developers reason about the behavior of a program—in this case, their tests—across many executions. This problem has been explored in a broad sense by the HCI community. For instance, Regae [94] reveals the behavior of synthesized regular expressions by which strings from a set of example strings the regular expression matches. Tools have also been built to visualize the value a variable takes on over multiple executions of a statement [48, 59], and to expose the effect of data processing commands on each member of a data table [39, 81]. In the area of AI interpretability, tooling has been developed to support rapid inspection of input distributions and their corresponding outputs (e.g., [40, 41]). Lucid, like this prior work, reveals the behavior of a program over many executions. Its uniqueness arises from its

[5] Generally speaking, fuzzers operate on whole programs and run for extended periods of time, whereas PBT tools operate on smaller program units and run for shorter. Instead of testing logical properties, fuzzers generally try to make the program crash.

focus on test suite assessment, for substantial software, on complex test inputs (e.g., logs, trees, lists); this requires a different set of metrics, corresponding views, and linkages between those views (as we detail in §4), and the selection of control mechanisms that draw from current testing process.

Related to Lucid are tools that help programs understand the difference in program behavior across revisions of that program. For instance, tools have been designed for computational notebooks to help them both review revisions of code [36, 78], and how each revision results in different output data [21, 89]. These ideas have been additionally explored in the educational setting to help students pinpoint where their program diverges from a reference solution [87], and in the end-user programming domain to understand the difference in behavior among trigger-action programs [95]. In the future, we could see Lucid as incorporating similar interactions to these systems, in particular to help developers better understand tradeoffs between variations in how they have implemented their test harness.

Lucid also draws inspiration from prior work that has sought to help programmers make sense of large spaces of programs, whether they are student submissions to programming assignments [25, 35] or API usage examples [27, 93]. While Lucid does not cluster programs per se, it does perform some lightweight clustering of test cases—grouping together repeatedly tested inputs—as well as sorting and ordering test cases in a way reminiscent of the above techniques to support more rapid review of the space of tested inputs. If sufficient information could be derived about the structure of examples and how they are processed by the program, it might be possible to bring affordances like interactive hierarchical views [27] or semantic groups [25] into views of examples in Lucid.

3.3 Formal Methods in the Editor

This paper opens a new chapter in exploring the incorporation of formal methods into editing environments. Recent projects at UIST and beyond have explored providing powerful new interfaces to formal methods—in these interfaces, formal methods have enabled powerful affordances for repetitive text edits [61], refactoring by demonstration [69], code search [67], synthesizing programs—including regular expressions [94], Python code [17], web automation code [76], and visualizations [90]—and bidirectional programming where edits propagate bidirectionally between code and output [38]. PBT has at times been called a lightweight formal method. As with other work at UIST, this paper does the work of taking a powerful formal method and giving it the displays and control mechanisms necessary to invoke easily in the editing environment.

4 FORMATIVE STUDY

We informed our work on LPBT through a formative study with experienced PBT users. In §4.1 we give an overview of the study methods, including the the preliminary interface design that we presented to users in the study. Then, in §4.2, we discuss the goals that participants had when evaluating their test suites and the strategies that they applied to achieve those goals. Finally, §4.3 presents design considerations, based on study results and corroborated by the literature, that ultimately made up the principles of LPBT.

4.1 Methods

The PBT pitfalls discussed in §2.2 suggested to us that some kind of user interface for visualizing and understanding PBT results may help developers to better evaluate their tests. With this insight, we recruited a small group of experienced PBT users, explored the ways that they evaluate their tests, and collected their feedback on a prototype user interface.

4.1.1 Study Procedure. We conducted a small interview study to clarify the important aspects of LPBT and explore potential interface designs. We recruited 5 experienced PBT users through X (formerly Twitter) and oru personal networks (P1–P5) for 1 hour interviews, asking questions about how they evaluate testing effectiveness and collecting feedback on our early prototype of Lucid. The interviews were split into two parts. In the first part, participants showed us code with existing PBT tests, described those tests, and answered questions about how they might evaluate that those tests are effective. Then in the second part, participants installed the Lucid extension for VSCode and used it to gain visibility into their own test suites.[6]

We began the study with a prototype system that was designed based on insights from the PBT literature and then we continued to iterate on its design between interviews, based on feedback. Interviews were automatically transcribed by video conferencing software[7]. We analyzed the data via a lightweight thematic analysis [6].

4.1.2 Prototype System. The prototype that we started the study with was designed based on recommendations from prior work. *PBT in Practice* suggested that developers wanted "visualization(s) of the test [inputs] being generated... to better understand the breadth of inputs generated." As a first implementation of this insight, we created an extension to VSCode [8] that provided developers with charts that summarized the data used to test a given property. We proposed a few summary statistics early on, based on ideas from the literature:

- "Number of Unique Inputs" As a starting point, we chose to show the programmer how many of their inputs were unique and how many were duplicated. Our experience says that new PBT users are sometimes surprised that the system does actually produce duplicates from time to time, but since they are often more expensive to filter out than they would be to simply re-test, PBT frameworks often do not bother to deduplicate the test inputs. Knowing how many unique inputs were tested gives a better idea of testing effectiveness than the total number generated that frameworks sometimes provide.
- "Proportion of Valid Inputs" As discussed in §2.2, not all potential test inputs are usable for testing. Improving data generation in the face of complex preconditions is a major line of inquiry for testing researchers [12, 15, 32, 84, etc.], but even state of the art generation techniques are imperfect,

[6] P5 was prepared with properties to discuss, but they did not have the required libraries installed to run those properties, so we were not able to show them the interface running on their own code.
[7] P3's interview audio was lost due to technical difficulties, so we fell back to the notes taken during the interview.
[8] https://code.visualstudio.com/

so it is critical that developers have information about how many tests were run and how many were rejected.

- "Size Distribution" The size of inputs used to test a given program matters a lot when testing. Some testers subscribe to the *small scope hypothesis* [2], which asserts that exhaustive testing of small inputs is best for bug finding, while others believe that the *combinatorial advantage* [43] of large test inputs means that a large input can find exponentially more bugs.[9] Whichever viewpoint a tester subscribes to, keeping track of the sizes of test inputs matters a lot, and a visualization of the distribution of sizes can help the developer make better testing decisions.

These statistics and charts were gathered together in a single visualization pane that was shown in a developer's editor alongside their tests.

Besides serving as the basis for our study, this prototype implementation was presented as a demo at a major HCI conference.

4.2 Goals and Strategies

Before showing participants the prototype interface, we explored the ways they determine PBT effectiveness on their own. In a perfect world, testing effectiveness would be measured by bugs found—that is the ground truth—but this is impossible to measure perfectly (if we could, we would know what all the bugs are and wouldn't need to do any testing!). Instead, developers pay attention to proxy metrics that give them confidence in their test suite. We now discuss the various metrics that developers paid attention to and explore how they measured them.

4.2.1 Test Input Distribution. Participants understood that the value of a property is highly dependent on the distribution of inputs used to test it.

Participants reported checking that their distributions covered potential edge cases like $x = 0$, $x = -1$, or $x =$ `Integer.MAX_VALUE` (P2), which are well documented to be important cases to check when looking for bugs. They also checked that their distributions covered variations in the input space like $x = 0$, $x < 0$, and $x > 0$ (P2, P4, P5); this kind of coverage is similar to notions of "combinatorial coverage" discussed in the literature [31, 55].

Many participants (P1, P3, P4) also expressed wanting to know that their test data was realistic. They expressed that the most important bugs were the ones that users were likely to hit, so they wanted to make sure they had tested those code paths especially well. Another participant (P5) wanted their test data to be uniformly distributed across some space. They thought that this would make it easier for them to estimate the probability that there was still a bug in the program, given the number of inputs they had tested so far. Whether to test with realistic distributions or uniform ones is a topic of debate in the literature: some tools work hard to ensure uniform distributions, even for data that is very difficult to generate uniformly [12, 66], while other testing work sees uniformity as too detached from the behavior of the system under test and prefers a more holistic approach [82].

Participants used a combination of strategies to gauge these proxy metrics for test quality. Some (P1, P3, P5) tried to measure these metrics by reading through the list of examples. Two participants (P2, P5) described using evaluation tools already present in their PBT framework of choice; one participant used `events` in Hypothesis and another used `labels` in QuickCheck. The participants used these tools to understand how their code was covering different variations in the input space (e.g., how often does a particular variant of an enum appear). As we show in §2, many PBT frameworks do not provide these affordances, and those that do provide exclusively text-based interfaces that are difficult to interpret at a glance and can easily get drowned out among other terminal messages.

4.2.2 Coverage of the System Under Test. Three participants (P2, P4, P5) discussed that coverage of the system under test (e.g., line coverage, branch coverage, etc.) was also important to them as a proxy metric for test quality. While code coverage is a common way of measuring test quality, there is strong evidence that code coverage alone does not indicate testing success [51].

Two participants (P2, P4) reported actually measuring code coverage via code instrumentation, although P2 did point out the potential failings of code coverage (calling it "necessary but not sufficient").

4.2.3 Test Performance. Finally, two participants (P1, P2) discussed performance as an important proxy metric. They argued that they have a limited time in which to run tests (e.g., because the tests run every time a new commit is pushed or even every time a file is saved), so faster tests (more examples per second) naturally exercise the system better. They measured this performance with the help of tools built into the PBT framework.

Importantly, participants did not all seem confident in either the proxy metrics they cared about (P2 initially said they did not have any good evaluation metrics in mind) or in their techniques for measuring them (P4 initially claimed not to have a good technique in mind).

Besides these metrics, participants also expressed being more confident in their tests when they understood them (P3), when the test had failed previously (P3), and when a sufficiently large (for some definition of large) number of examples had been executed (P1, P4).

4.3 Design Considerations

Once participants had explored the interface in the context of their code, they provided incredibly useful feedback on the design of the tool. By combining this feedback with recommendations from the literature, we arrived at five core principles that should be true of LPBT systems.

Visual Feedback The initial recommendation from *PBT in Practice* that developers needed visual representations of their PBT results was a good one—participants reactions supported visual feedback as a core component of the interface. Most participants (P1, P2, P3, P5) mentioned Lucid's visual charts directly, stating that "having visual awareness is very interesting" (P3) and the visual charts are "a lot easier to digest than" the built in statistics output from Hypothesis (P2).

[9] The truth seems to be that each of these viewpoints is correct in some situations; a recent study [80] has a nice discussion.

Workflow Integration The prototype system was built with the assumption that "Editor Integration" may be an important component of LPBT. Indeed, if developers truly "[do] not think as hard as they should about testing effectiveness," then surely it is important to make the interface maximally available to encourage use. But while some participants validated this assumption (P1 and P2), another said they were "not always a big fan of extensions" because they use a non-VSCode IDE at work (P4). For that participant, an editor extension actually *dis*courages use. This discussion made it clear that "editor" integration is not necessarily the key; instead, LPBT interfaces should adapt to fit neatly into whatever workflows developers are already using to do their jobs.

Customizability The initial set of proposed visualizations seemed like a reasonable start, but participants suggested a slew of other visualizations that they thought might improve their testing evaluation. Many of these visualizations (e.g., code coverage and timing—see §5.2) were implemented once-and-for-all, but every testing domain is different, so developers still needed a convenient way to specify which features of the input data should be aggregated and visualized. This implied that LPBT tools should be customizable so developers can see the information that is important in their testing domain.

Explorability One commonality between almost all (P1, P2, P3, P4) participants was their curiosity while using the prototype interface. Participants would see a visualization (e.g., a categorization feature chart or a breakdown of discarded values) and wonder why the chart looked the way it did. They would then dig around in the interface, look at lists of examples, or simply ask us to try to ascertain exactly what was going on. This propensity for LPBT to surprise participants and encourage their curiosity may actually be critical in the context of testing: prior work [91] has demonstrated that fostering curiosity helps developers find errors in their code. Thus, LPBT interfaces should support this kind of curious exploration.

Standardization The above principles do not necessarily fix a particular design, and, in fact, it the community should have a wide variety of LPBT tools for different use-cases. But it would be a shame to have a wide variety of *incompatible* tools: this would lead to duplicated effort and a fragmented ecosystem. Thus, we establish standardization as a final "meta-principle" for LPBT. LPBT tools should have a consistent API so that programmers who want to add LPBT support to a new PBT framework or add a new set of visualizations can do so in a way that is compatible with existing tools.

5 DESIGN OF THE LUCID INTERFACE

The principles of LPBT from the previous section paints a detailed picture of what an LPBT interface should look like. A LPBT interface must provide rich array of **visual** information about testing effectiveness that can be easily **customized** by the user to address their particular testing domain. It must be **integrated into common workflows** so it can draw the user's attention, and it must **explorable** to encourage user curiosity. In this section, we describe Lucid, an interface that embodies these principles for the benefit of real developers. We describe the engineering details of Lucidin the next section (§6), but in this section we describe the interaction model that we imagine Lucid (§5.1) and the different visualizations that Lucid makes available (§5.2).

5.1 Interaction Model

While we have not developed a formal process model of the way Lucid users work, we present a hypothetical interaction model in order to demonstrate what using Lucid may feel like for a user.

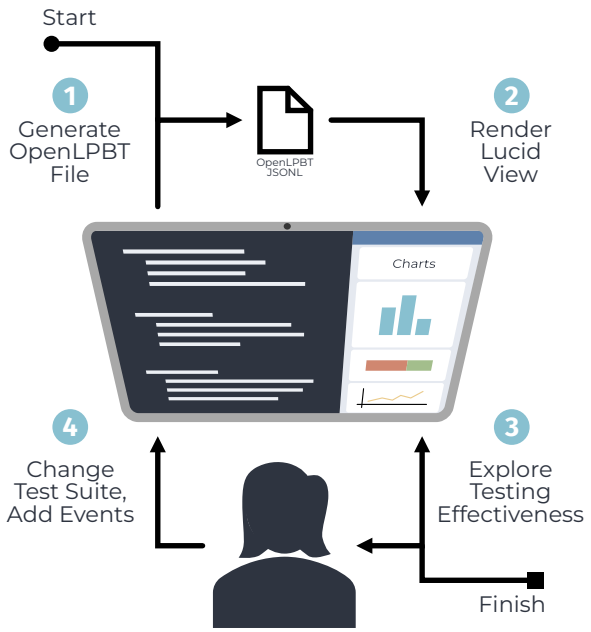


Figure 2: The high-level Lucid interaction model.

5.1.1 The Lucid Loop. We envision the user interaction for Lucid to follow roughly the outline in Figure 2.

(1) At the start of the loop, the test framework (e.g., Hypothesis) runs the test suite and collects relevant data into a OpenLPBT file (we discuss the details of this format in §6).
- In VSCode, the editor extension automatically watches for new data files in the workspace, while browser users must manually navigate to the data file to make sure the Lucid interface can find it.

(2) Once a data file has been either automatically or manually chosen, Lucid renders an interface with a host of rich visualizations (which we will discuss in detail in §5.2).

(3) Next, the user interacts with the interface. This may be as simple as seeing a visualization and immediately noticing that something is wrong, but they may also explore the interface to answer questions about surprising results or generate hypotheses about what might need to change in their test suite. If the user is happy with the quality of the test suite at this point, they may finish their testing session.

(4) Finally, the user may make changes to either their test setup (e.g., random generation strategies or Hypothesis settings) or they may make changes specifically to customize Lucid visualizations.

- When the user is ready, they can re-run their tests and return to (1).

5.1.2 Assumptions and Events. In step (4) of the Lucid loop, the user has the opportunity to change their test suite in ways that will customize the visualizations that are shown on the next time around the loop. These customizations come in to flavors, both of which are demonstrated in Figure 3, which is an extension of the red–black tree property shown in §2. Note that while Lucid is intended to work with any PBT framework, we describe assumptions and events in detail for Python's Hypothesis specifically. Other frameworks may choose to implement user customization in other ways that are more idiomatic for their users.

```
def test_insert_lookup(t, k, v):
    # Add numerical features tracking "size"...
    event("size", payload=size(t))
    # ... and "black_height"
    event("black_height", payload=black_height(t))

    # Add a categorical feature checking if the
    # root is positive
    event("root_positive", payload=root_pos(t))

    # This was already here, but it impacts which
    # samples are considered "valid"
    assume(is_red_black_tree(t))
    assert lookup(insert(k, v, t), k) == v
```

Figure 3: The red–black tree property from §2, extended with events.

Assumptions. The `assume` function was already called in the version of the property from the background section. Recall that the assumption filters out any tree that does not satisfy the provided Boolean check—in this case, that the generated tree is a valid red–black tree. But when used in conjunction with Lucid, assumptions also to control which inputs appear as "valid" in certain visualizations. If a user finds examples that Lucid labels "valid," but that they would prefer to exclude for testing, they can use `assume` to exclude those examples from both the testing process and the visualizations.

Events. New to the property in Figure 2 are three different calls to `event`. Before Lucid events were used in Hypothesis to mark specific paths through the property as interesting. For example, the programmer might write,

```
if some_condition:
    event("hit_condition")
```

and then Hypothesis would add a line like "`hit_condition: 42%`" to its statistics output. (In QuickCheck, these kinds of events are called "labels," and behave much the same way.) To support a richer set of test input features, we extended the Hypothesis API (with the support of the Hypothesis developers) to allow events to include "payloads" that are either numbers or strings.

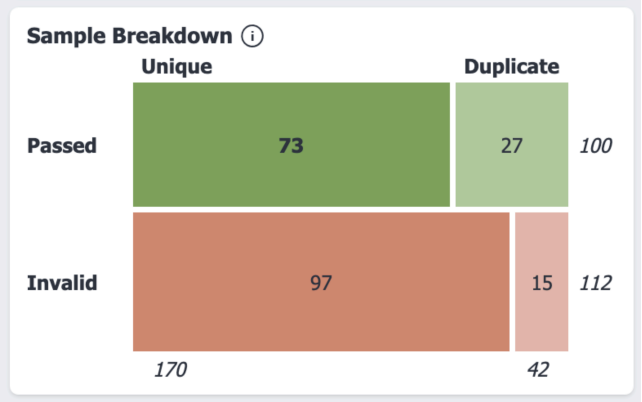
Now, when a developer adds events like the ones in the figure, it causes Lucid to generate additional visualizations that capture the breadth of events logged by Hypothesis.

By reusing Hypothesis's existing idioms for assumptions and events, we were able to hook into existing developer workflows and make them more powerful.

5.2 Visualizations

The Lucid interface presents the user with a variety of helpful visualizations.

5.2.1 Sample Breakdown.

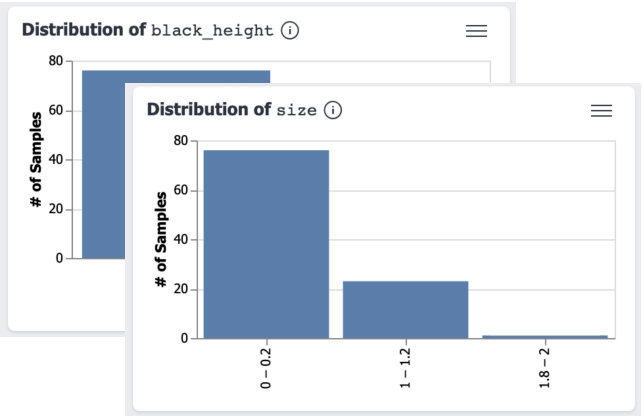


The initial prototype of Lucid told the user about the number of unique inputs and the proportion of valid ones, but feedback from P1 suggested that these two data were difficult to understand when presented separately. The mosaic chart [33] shown in the updated Lucid interface makes the relationship between uniqueness and validity clear: the four quadrants capture each of the four combinations of those states. Follow-up conversations with P1 suggested that this presentation was much clearer.

Recall that "validity" is controlled by assumptions that the developer adds into their property, so this view is customizable to adapt to domain specific notions of validity.

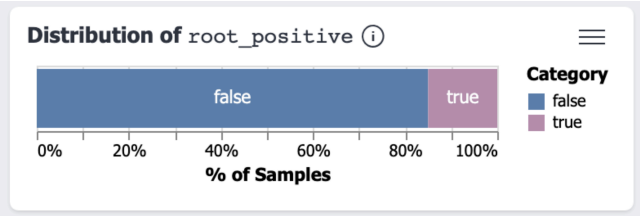
As discussed above, many of the Lucid visualizations are interactive. In the case of this mosaic chart, the user can click on any of the quadrants (as well as any of the four aggregate counts) to see the specific test inputs that contribute to that slice.

5.2.2 Feature Charts.



When users add events to their properties to capture interesting numeric features of their test inputs, the Lucid interface generates

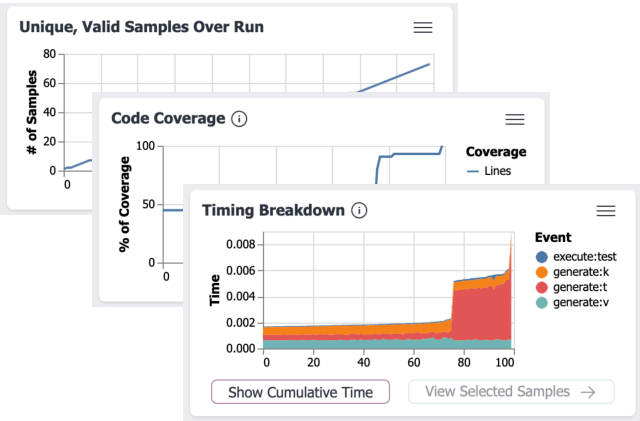
a bar chart to help visualize that data. Numerical feature charts subsume the "Size" chart that was displayed in the prototype implementation, giving developers more freedom to come up with different numerical measurements that are interesting for their domain. In the case of our running red–black tree example, the *black height* of each generated tree gives extra insight on top of the more generic size data.



If a user adds an event that captures categorical data, rather than numeric data, we present them with a different style of bar chart. (The original prototype used a pie chart for these kinds of categorical visualizations, but following we switched to a version that was easier to interpret.) These categorical feature charts can be used to visualize additional notions of validity that are not captured in the sample breakdown, but they are also especially useful for understanding the combinatorial coverage [55] of the input space. We also implement a two-dimensional version of this chart, which is similar to the sample breakdown mosaic chart, for visualizing this two-dimensional categorical. These charts were informed by conversations with P4 and P5.

Similar to the sample breakdown, the bars in feature charts can be clicked on to inspect the examples that contribute to the particular bar.

5.2.3 *Time, Code Coverage, and Unique Over Time Charts.*



In addition to the charts that can be customized via assumptions and events, we implement a few charts that present generically useful data. Each of these charts was inspired by feedback from a participant in the design study.

"Unique, Valid Samples" The original version of Lucid had simple counts of how many unique samples were used for testing, but keeping track of exactly when those samples appear is also interesting. A plateau in a chart of samples over time indicates that a property may be "saturated" and that the generator is either too narrow or that fewer samples would suffice. (Inspired by P1.)

"Code Coverage Over Time" As mentioned above, code coverage is an incomplete way to measure testing success, but it is still useful sometimes. We added a chart showing how code coverage changes over time, which gives developers a sense of at what point during testing they maximize the total lines covered. (Inspired by P2.)

"Timing Data" When properties are given limited time to run, it is important to know where that time is going. We added visualizations of the total time spent during each stage of the testing process (e.g., generation time for each argument, execution time, etc.) that helps developers notice data generators that may need to be optimized. (Inspired by P1.)

5.2.4 *Example View.*

```
test_insert_lookup(
    t=T(c=Red(), l=E(), k=-9663, v=-26613,
r=E()),                                          3x
    k=-56,
    v=89,
)
------------------------------------------------------
test_insert_lookup(
    t=T(c=Black(), l=E(), k=0, v=0, r=E()),
    k=0,                                         2x
```

While part of the point of Lucid is to avoid programmers needing to sift through test input examples, we found that there were still a few reasons for them to do so. First and foremost, Lucid's example view enables the curious exploration that we discussed as a critical principle of LPBT—the example view is usually opened by clicking on a particular area of one of the above charts, in which case it shows the data that contributes to the given chart area. This means that developers can dig into the data to answer questions about the chart looks a certain way. As discussed in §4.3, supporting this curiosity may be beneficial for bug-finding in its own right, by encouraging developers to be curious about their code's behavior. Secondarily, the example view can be used to search for particular examples to make sure they appear in the sample (e.g., important edge or corner cases). To support this, the example view features a simple search bar that can be used to filter the view.

Each example in the view shows a textual representation of the generated sample, and it can be expanded to see the full associated `example` JSON object (see §6.2). Examples are initially grouped, so that identical examples are not shown multiple times; this manifests in the "3x" nnd "2x" annotations shown in the above screenshot. Even though this is not technically a visualization, it still provides better visual feedback than the alternative. Additionally, grouped examples are sorted in decreasing order of frequency and unique examples are sorted in increasing order of size. The former order makes it easy to see what was generated very often, and the latter gives the developer the opportunity to ease into the examples rather than immediately be shown ones that are too large to read.

6 IMPLEMENTATION OF THE LUCID INTERFACE

In this section we briefly outline the implementation details of the Lucid interface. We begin with the mechanics of the system itself

(§6.1), but the most interesting part of the implementation is the standardized OpenLPBT data format that PBT frameworks use to send data to Lucid (§6.2). In §6.3 we explain how the implementation of OpenLPBT and Lucid make it easy to extend the LPBT ecosystem.

6.1 Lucid Implementation Details

At the implementation level, Lucid is a web-based interface that is enabled by a few simple changes to PBT frameworks and supported by existing developer tooling.

6.1.1 React Application. Lucid is a React [88] web application that consumes raw data about the results of a PBT run and produces interactive visualizations that help users make sense of the underlying data. The primary way to use Lucid is in the context of an extension for VSCode, which puts the interface alongside the tests that it pertains to, but it is also available in a browser as a static HTML page to support workflow integration for non-VSCode users. (When running as an extension, Lucid is still fundamentally a web application; VSCode can render web applications in an editor pane.)

6.1.2 Framework Integration. As discussed in §5.1, we worked with the Hypothesis developers to make a few small changes to the framework that enable Lucid. The main change was adding a hook to capture data on each test run, which was relatively simple to do in Hypothesis. The data from all of these implementations is captured in the OpenLPBT format, which we will discuss in §6.2.

In Hypothesis specifically, we also adapted the `event` function to have a richer API, which we describe in §5.1.

6.1.3 Live Launcher. When run as a VSCode extension, Lucid can automatically watch and refresh the underlying data files to streamline the experience for the user. In an ideal workflow, a developer sets up their test suite to watch their code files (e.g., via pytest-watch, which reruns Hypothesis tests when the test file is saved) and then Lucid watches the files that Hypothesis outputs its data to. The end-to-end experience is live feedback that updates every time code is saved. This was the original intent of **workflow integration**; the Lucid interface just shows up and provides feedback, even if the developer did not think to check for it.

6.2 The OpenLPBT Data Format

Since standardization is a core principle of LPBT, we define an open standard that PBT frameworks can use to communicate with lucid PBT tools. Our format OpenLPBT, is based on JSON Lines. Each line of the file is a JSON object that corresponds to one *example*. An example is the smallest unit of data that a test might emit; each example represents a single test case.

The abridged JSON schema in Figure 4 defines the format of a single example line. Each example has a `run_start` timestamp; this is used to group examples from the same run of a property and allow multiple runs of data to be stored in the same file. The `property` field names the property being tested and the `status` field says whether this example `"passed"` meaning the property passed error, `"failed"` indicating that the example disproves the property, or `"discarded"` meaning that the value was not valid for testing. An example's `representation` is a human-readable string describing the example. Finally, an example's `features` reflect the

```
{
  line_type: "example",
  run_start: number,
  property: string,
  status: "passed" | "failed" | "discarded",
  representation: string,
  features: Record<number | string>,
  coverage: ...,
  timing: ...,
  ...
}
```

Figure 4: The OpenLPBT line format.

customizable events that users used to project generated values to either strings or numbers, for the purpose of visualizations.

The full format includes a few extra fields, including some human-readable descriptions (e.g., to explain why a particular a was discarded), optional fields naming the particular generator that was used to produce a value, and a freeform metadata field. A guide to using the format can be found online.[10]

6.3 Extending the Ecosystem

The clean divide between OpenLPBT and Lucid means that new frameworks are free to implement OpenLPBT and get access to Lucid and that alternatives to Lucid can automatically work with any framework that implements OpenLPBT.

6.3.1 Supporting New Frameworks. Supporting a new framework is as simple as adding some lightweight logging infrastructure after a given property is run. Frameworks can start small: including only five fields—`type`, `run_start`, `property`, `status`, and `representation`—is enough to gain a substantial portion of the benefits of LPBT. After that, we highly recommend adding `features` to enable user control of visualizations; `coverage` and `timing` may be harder to implement in some programming languages, but users do find these features valuable when available.

We have currently implemented support for OpenLPBT in Hypothesis, Haskell QuickCheck, and OCaml's base-quickcheck. The Hypothesis implementation is deeply embedded in the framework, but our minimal QuickCheck implementation was implemented as external library in about 100 lines of code.

6.3.2 Adding New Analyses. Basing OpenLPBT on JSON Lines and making each line a mostly-flat record means that processing the data is very simple. This simplifies the Lucid codebase, but it also makes it extremely easy to process the data with other tools. For example, getting started with OpenLPBT data in a Jupyter notebook requires two lines of code

```
import pandas as pd
pd.read_json(<file>, lines=True)
```

This means that if a developer starts out using Lucid but finds that they need a visualization that is not a straightforward assumption

[10] See https://hypothesis.readthedocs.io/en/latest/observability.html. Some field names been changed to clarify the explanations in the paper.

or event, they can simply load the data into a notebook and start building their down analyses.

In the open source community, we also expect that developers may find entirely new use-cases for OpenLPBT data that are not tied to Lucid its embodiment of LPBT. For example, we envision that OpenLPBT data could be used for reporting that communicates testing performance to other developers or managers, long-term analysis that measures meta-properties of test quality over time, and profiling that looks for specific kinds of performance problems by cross-referencing LPBT data with other performance data. We are excited to see what the community does with this standardized and readily available source of testing information.

As PBT researchers with an interest in a broad range of PBT frameworks, we are well-positioned to propose a format like OpenLPBT. We have buy-in from the Hypothesis team that this format works for their framework, and we have demonstrated that the same layout also works for capturing relevant data in frameworks like Haskell's QuickCheck and OCaml's base-quickcheck. As Lucid grows and develops (see §8.2), as we built new tools for LPBT, and as we build integrations with more and more frameworks, we hope that this standard format will raise the bar and level the playing field around PBT framework usability.

7 EVALUATION

In this section, we evaluate the Lucid interface and, to some extent, the principles of LPBT themselves. In §7.1 we describe our main evaluation, an online study that we developed to quantitatively measure the impact of the Lucid interface on users' ability to evaluate the quality of test suites. Then in §7.2 we describe other impact that LPBT has had, including early tool adoption and bugs found in the Hypothesis testing framework.

7.1 Online Study

We designed the principles of LPBT, and Lucid specifically, to give developers a better understanding of the quality of their test suites. We were reasonably confident that the Lucid interface was usable in general—we iterated the design based on feedback from the design study, and we checked back in with participants when we made a change that they did not directly suggest—but we still did not know if Lucid actually achieved its goal of improving developer understanding. Specifically, we still had the following research questions:

RQ1 Does Lucid help developers to predict the bug-finding power of a given test suite?

RQ2 Does Lucid change developers' level of confidence in a given test suite?

RQ3 Do developers perceive Lucid as being useful to determining the difference between good and bad test suites?

Since these questions are about concrete impact on developer performance, we wanted to answer them with an appropriately-scaled quantitative study.

7.1.1 Study Population. In order to achieve statistically significant quantitative results, we chose to examine these research questions using a wide-reaching online survey instrument. We recruited from graduate and undergraduate student populations in the computer science department of a large university and social media users on X (formerly Twitter) and Mastodon, with the hopes of recruiting a diverse set of programmers ranging from relative beginners with no PBT experience to experts who may have some exposure to it.

We were ultimately able to recruit 44 participants. 4 responses were discarded because they did not correctly answer our sense check questions, leaving 40 valid responses.

Participant Prior Experience. All but one participant reported that they were at least proficient with Python, with 12 self-reporting as advanced, and 9 as experts. Half of the participants reported being beginners at PBT, 13 proficient, 6 advanced, and 0 experts. Almost all participants reported being beginners with Hypothesis (the Python PBT framework that we used in this study), with 7 reporting as proficient. To summarize, the average participant had experience with Python, but likely not PBT, and if they did know about PBT it was often not via Hypothesis.

Participant Demographics. 4 participants reported having a high school diploma, 15 an undergraduate degree, and 20 a graduate degree. The majority of our participants (24) reported that they were students; 7 were engineers; 3 were professors; 6 had other occupations. 28 participants self-identified as male, 5 as female, and 2 as another gender; 5 did not specify their gender. We discuss this sample skew in §7.1.5.

7.1.2 Study Design. At its core, our survey consisted of four "tasks." Each task presented the participant with a PBT property and three different random generation strategies for that property. The goal of each task was to rank the random generation strategies, in order of their bug-finding power, with the help of either Lucid or a control interface that mimicked the existing experience of Hypothesis. Concretely, the control interface consisted of Hypothesis's "statistics" output and a list of pretty-printed test input examples. The statistics output does display Hypothesis's warnings, although they may not have been as prominent as they would have been in real use. Both interfaces were styled the same way and embedded in HTML iframes, so participants could interact with them as they would if the display were visible in their editor.

Before the study, we benchmarked each trio of generation strategies using Etna [80]. Etna greatly simplifies the process of *mutation testing* as a technique for determining the bug-finding power of a particular generation strategy. The programmer specifies synthetic bugs that they would like injected into a bug-free program, and Etna does the work of determining how quickly (on average) a generator is able to trigger a particular bug with a particular property. Prior work has shown that test performance on mutation tests is be correlated with test performance on real faults [47], so this mutation testing process gave us a ground-truth ranking for each task that we could compare survey answers against.

The full survey workflow is summarized in Figure 5. We started by providing participants some general background on PBT, since we did not require that participants had worked with it before, and some more specific background on the challenges of writing high-quality PBT test suites that was relevant to the task at hand. Then, after some sense-check questions to ensure that participants had understood the background, we presented them with the four tasks.

- *Red–Black Tree* The property above about the `insert` function for a red–black tree implementation.
- *Topological Sort* A property checking that a topological sorting function works properly on directed acyclic graphs.
- *Python Interpreter* A property checking that a simple Python interpreter behaves the same as the real Python interpreter on straight-line programs.
- *Name Server* A property checking that a name server[11] behaves the same as a simpler model implementation.

We randomized the participants' experience in three ways: (1) two tasks were randomly assigned Lucid and the other two received the control interface, (2) tasks were shown in a random order, and (3) strategies were shown in a random order. This randomization mitigated potential biases due to the order that different tasks or conditions were encountered.

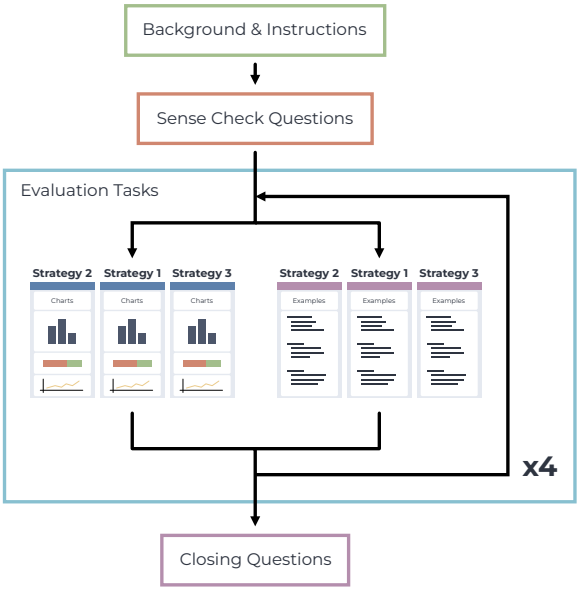


Figure 5: The flow of the quantitative analysis survey.

Four participants completed the study in over an hour; we suspect this is because they started, got up for a while, and then returned to the study. Of the rest, participants took 32 minutes on average ($\sigma = 12$) to complete the study. Only one participant took less than 15 minutes. Additionally, participants took about 3 minutes on average ($\sigma = 2.5$) to complete each task. Given the size of the tasks, we think these numbers are a good signal that participants were generally taking the survey seriously.

7.1.3 Results. We now answer our research questions based on the data collected from users.

To answer **RQ1**, we measured how well participants' rankings agree with the true rankings determined based on mutation testing. Given a participant ranking, for example $S_2 > S_1 > S_3$, we compared it to the true ranking (say, $S_1 > S_2 > S_3$) by counting the number of correct pairwise comparisons. For these two rankings, the participant correctly deduced that $S_1 > S_3$ and $S_2 > S_3$, but they incorrectly concluded that $S_2 > S_1$, so this counted as one *incorrect comparison.*[12]

Figure 6 shows the breakdown of incorrect comparisons made with and without Lucid, separated out by task. We performed a one-tailed Mann-Whitney U test , with the null Hypothesis that Lucid does not lead to fewer incorrect comparisons. The results follow:

	U	p	r	m_{Lucid}	m_{Examples}
Red–Black Trees	65	< 0.01	0.84	0	1
Topo. Sort	127	0.01	0.68	0	1
Python Interp.	182	0.26	-	0	0
Name Server	91	< 0.01	0.77	0	1

(All sample sizes were between 18 and 22, depending on the random variation in the way conditions were assigned.) For three of the four tasks (all but *Python Interpreter*), participants made significantly fewer incorrect comparisons when using Lucid, with strong common-language effect sizes, r.[13] For *Python Interpreter*, participants overwhelmingly found the correct answer with both conditions—in other words, the task was simply too easy—but precisely why it was too easy is interesting. We discuss this in §7.1.4. These results indicate that we can conclusively answer **RQ1** with "yes."

Alongside each ranking, we asked developers to rate their confidence in their answer on a scale from 1–5[14] Unfortunately, this did not help us to answer **RQ2** as we had hoped. We tried computing Spearman's ρ [83] between the confidence scores and incorrect comparison counts, and we found no significant relationship. In other words, participants' confidence was not, broadly a good predictor of their success. We did find that participants' reported confidence was significantly higher for two of the four tasks with Lucid than without (as computed via a similar one-sided Mann-Whitney U test to the one before), but if confidence does not predict success it is unclear how to interpret those results. Confidence is well-known to be difficult to measure and interpret , so in the end **RQ2** may just not have been a useful question to ask.

Finally, we used a post-study survey to answer **RQ3**. We asked participants "Which view [Lucid or the control] made the difference between test suites clearer?" with five options on a Likert scale. All but one participant said Lucid made the differences clearer, with 35/40 saying Lucid was "much clearer."

7.1.4 Discussion. Overall the online study was an encouraging evaluation of Lucid's impact on developer understanding. In addition to the quantitative observations above, we also made a couple of other useful observations in the course of our analysis.

First, we asked participants for feedback on which of Lucid's visualizations they found useful and elicited free-text feedback on the design of the interface. We were unable to draw strong

[11] https://en.wikipedia.org/wiki/Name_server

[12] This metric is isomorphic to Spearman's ρ [83] in this simple case. Making 0 incorrect comparisons equates to $\rho = 1$, making 1 is $\rho = 0.5$, 2 is $\rho = -0.5$, and 3 is $\rho = -1$. We found incorrect comparisons to be more intuitive to explain and work with, so we focus on that instead.

[13] This corresponds to the probability that randomly sampled Lucid participant will make fewer errors than a control participant, computed as $r = \frac{U_1}{n_1 * n_2}$.

[14] Specifically, we asked "How confident are you in your answer?" and users responded "Not at all" = 1, "A little confident" = 2, "Moderately confident" = 3, "Very confident" = 4, "Certain" = 5.

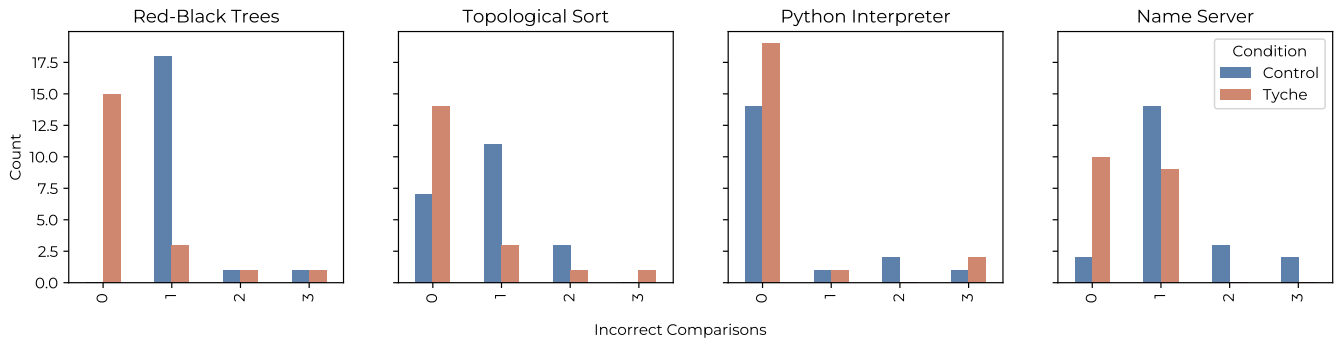


Figure 6: Distribution of incorrect comparisons made for each task, where each incorrect relative ranking between two test suites counts as one error. Significantly fewer incorrect comparisons were made for Red Black Trees, Topological Sort, and Name Server. = Lucid, = Control.

conclusions about any of the results, but one interesting note is that the vast majority of participants (37/40) stated that Lucid's "bar charts" were helpful. Unfortunately we phrased the question poorly: we intended for that to refer only to *feature* charts, but on further inspection participants may have interpreted that to include the mosaic chart as well. Even so, such a strong majority finding one or the other of those features useful suggests that the core of the interface added value.

Second, we should address the *Python Interpreter* task. We mention above task was too easy; participants made very few mistakes on it across the board. We propose that this is, at least in part, because the existing statistics output available in Hypothesis was already good enough. For the worst of the three strategies, Hypothesis clearly displayed a warning that "< 10% of examples satisfied assumptions," which is an obvious sign that something is wrong. Conversely, for the best of the three, Hypothesis showed a wide variety of values for the `variable_uses` event, which was only ever 0 for the other two strategies. Critically, the list displayed was *visually longer*, so it was easy to see the variety at a glance. We show an example of what the user saw in Appendix B. We think this result shows both that Hypothesis's existing tools were helpful in some cases and that the direction LPBT takes is a good one. The existing tools were able to help developers as much as Lucid did, but part of the reason they were successful was that, in these cases, they displayed data that was meaningful at a glance and visually interpretable.

7.1.5 Limitations. The online study has two potential limitations: sampling bias and ecological validity.

The sample we obtained was appropriate from the perspective of prior experience and general level of education, but we think it was sub-optimal for both gender and occupation. For gender, prior work has shown that people of different genders interact differently with user interfaces [10, 85], so we wish we could have gotten a more gender-diverse sample. For occupation, we reached a significant portion of students and proportionally fewer working developers. While many of those students are in computer science programs and therefore will likely be developers someday, we are curious to see know if working developers might have performed differently.[15]

Ecological validity is another limitation. Because this study was not run *in situ*, there are aspects of the experimental design that may have impacted the results. We tried hard to avoid these issues (e.g., we used live instances of Lucid in an iframe to maintain the full interactiveness of the real tool and we developed tasks that span a range of testing scenarios) but in the future we would like to study how developers use Lucid in practice. We discuss this more in §8.1

7.2 Other Impact

As Lucid is an open source project that has already engaged with the PBT community in small ways, we can also evaluate its design, in part, by looking at the impact it has had on software practice. The biggest measure of this is that Lucid has led to concrete bug-fixes and performance enhancements in the Hypothesis codebase itself. These Hypothesis changes came after study participants and other early users of Lucid noticed various issues that seemed to have nothing at all to do with their own test suites. On further inspection, these issues were either true bugs (e.g.,) or performance glitches (e.g.,) that have since been raised to the Hypothesis developers and fixed. This concrete impact means that Lucid has already helped to make thousands of developers' tests more efficient, and we hope that this pattern will continue.

Additionally, one of the authors was contacted by the developers of PyCharm, an IDE focused on Python specifically, to ask about the OpenLPBT format. They realized that the coverage information therein would provide them a shortcut for code coverage highlighting features that they had been meaning to implement for a while. This speaks to the versatility of OpenLPBT and the value provided by such open standards.

Finally, the Lucid extension has had over 100 installs on the VSCode extension store to date, and has been independently posted on Lobst.rs (a relatively small, software engineering focused social

[15]Interestingly, we may been able to ameliorate both of these shortcomings to some degree if we had been able to keep our survey on social media for longer; we were forced to take it down after a couple of days due obvious automated spamming. Lesson learned.

media website). This is still quite small engagement, but we are encouraged that the extension has already found some early users.

8 CONCLUSION AND FUTURE WORK

Our new paradigm of lucid property-based testing rethinks the PBT process as one that is fundamentally more interactive and empowering to developers. Rather than hide the data used to test properties, which may lead to confusion and false confidence, the OpenLPBT protocol and interfaces like Lucid give developers a rich understanding of their testing process. The online evaluation of this paradigm showed that it helps developers to tell the difference between good and bad generation strategies, and its demonstrated real-world impact on the Hypothesis framework backs up those conclusions. We hope LPBT continues to make PBT better for real developers.

Moving forward, we are excited about a few different directions.

8.1 Long-Term Deployment Evaluation

We are encouraged from the results from both our formative design study and our online evaluation study, but we think there is still more to explore about the design of LPBT and Lucid. In particular, as Lucid sees use in real-world software development settings, we are curious to explore its impact over time. Ideally, we would like to find a software development team at a medium or large software company that is interested in trying Lucid and follow their experience. We would ask the team to use Lucid as-needed and record their experience (either via note-taking or telemetry) as they go. At the end of a 3–6 month period, we would interview members of the team to ask how Lucid changed their PBT process and if they had ways they would change the interface or the paradigm more generally.

8.2 Evolving the Lucid Interface

As the Lucid project evolves, we see exciting opportunities to add new visualizations and workflows that support developer exploration and understanding.

Code Coverage Visualization. The visualization we provide displaying code coverage over time is a reasonable start, but it was not rated particularly highly by online study participants and in our experience it seems there is room for improvement.

One path forward is in-situ line coverage highlighting, like that provided by Tarantula [46]. Indeed, it would be easy to implement Tarantula's algorithm in Lucid (supported by OpenLPBT) in cases where the test framework finds failing examples. In cases where no failing examples are found, each line could simply be highlighted with a hue proportional to the number of times it was covered.

Line highlighting is useful for asking questions about particular parts of the codebase, but developers may also have questions about different parts of the input space. For this, we propose visualizations that cluster test inputs based on the coverage that they have in common. This would let developers answer questions like "which inputs cover uncommon parts of the space" and "which inputs have particularly high coverage."

In general, we think that diving deeper into unique ways that PBT interacts with code coverage visualization is an exciting direction for future research.

Mutation Testing Feedback. Mutation testing [73] attempts to better approximate the true measure of test suite quality—its ability to find bugs—by checking that it can find synthetic bugs or "mutants" that are added to the test suite. Many tools already exist to assist with mutation testing in general [74], but PBT is unique in that its ability to find bugs is nondeterministic. Developers not only want to know whether or not a test suite *can* find injected bugs, they want to know how long it takes on average, which bugs are found less often, etc. Luckily, there is already prior work on displaying the results of mutation testing for PBT in the form of a benchmarking tool called Etna [80]. The "task bucket" charts that Etna generates during benchmarks would be a great addition to the Lucid interface, and this added dimension of visual feedback would be a powerful way for developers to gain even more confidence (or, indeed, less confidence) in their PBT setups.

Comparing Data Over Time. Informal conversations we have had with potential users of Lucid have suggested that developers want ways to compare between visualizations at different points in time. This might be in the short term, because a developer wants to closely inspect what happened when they made a chance, or in the long term, because a developer wants to see how their testing has evolved over the course of months or years.

Interestingly, the design of the online evaluation study accidentally foreshadowed an effective design: allow two instances of Lucid to run side-by-side so the user can compare between them. This is simple to implement and provides little additional overhead or learning for developers, so we are excited to build this feature and explore how it is used.

8.3 Further Interaction Paradigms

We designed Lucid to support existing developer workflows and provide additional insights into potential test suite shortcomings, but participants in the formative study (P1, P4) did speculate about ways that Lucid could help developers to adjust their strategies after they notice something is wrong.

Direct Manipulation of Lucid Diagrams. In a perfect world, a developer who notices that something is wrong with a Lucid chart could simply manipulate that chart to look more like what they want, press a button, and receive an updated generation strategy that captures their desires. Unfortunately, this is intractable in general: if the programmer asked a generator of Boolean formulas for more that are satisfiable, this direct manipulation problem would reduce to 3-SAT. Still, there may be situations in which direct manipulation of a Lucid visualization is actually possible.

One idea would be to use probabilistic programming languages like Dice , which make it possible to view and manipulate the distribution of certain classes of probabilistic programs efficiently. This might allow, for example, manipulation of size distributions for trees, even if it cannot handle some harder tasks. In a totally different part of the literature, reflective generators [30] make it possible to tune a PBT generator to mimic a provided set of examples. If a developer thinks a particular bar of a chart should be larger, a reflective generator may be able to tune a generator to expand on the examples represented in that bar. We are eager to see how both of these insights can be integrated into PBT user interfaces.

Manipulating Strategy Parameters in Lucid. If direct manipulation of PBT results is too difficult, we also try to provide tools within Lucid for manipulating the parameters of different generation strategies. For example, if a generation strategy takes a `max_value` as an input, we could provide developers with a slider that lets them change that value and monitor the way the Lucid visualizations change. These kinds of interactions are far more tractable than direct manipulation of charts, but they may also pose difficulties around efficiency since each new value requires running a program hundreds of times. We are currently exploring if techniques for self-adjusting computation [1] could address this problem.

8.4 LPBT for Education

So far we have considered LPBT a paradigm that helps experienced software developers gain a better understanding of their testing effectiveness, but it could just as easily help beginners to understand what is going on "under the hood" when they are learning PBT for the first time. This is especially important because prior work has argued that teaching PBT can be a stepping stone to teaching formal methods more broadly [54, 68].

To reap these benefits, we plan to make Lucid available as a learning aid when teaching PBT to undergraduate students in a large university course. We will track students' use of the tool and collect feedback about how LPBT affects students' perception of testing effectiveness. Ultimately we hope that having tools like Lucid around from the beginning of one's PBT journey will give developers better-calibrated instincts with which to gauge their testing effectiveness.

REFERENCES

[1] Umut A. Acar. 2009. Self-adjusting computation: (an overview). In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM '09)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/1480945.1480946

[2] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. 2002. Evaluating the "Small Scope Hypothesis". (2002).

[3] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing telecoms software with quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang (ERLANG '06)*. Association for Computing Machinery, New York, NY, USA, 2–10. https://doi.org/10.1145/1159789.1159792

[4] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 1–4. https://doi.org/10.1109/ICSTW.2015.7107466

[5] 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.

[6] Ann Blandford, Dominic Furniss, and Stephann Makri. 2016. Analysing Data. In *Qualitative HCI Research: Going Behind the Scenes*, Ann Blandford, Dominic Furniss, and Stephann Makri (Eds.). Springer International Publishing, Cham, 51–60. https://doi.org/10.1007/978-3-031-02217-3_5

[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 (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 836–850. https://doi.org/10.1145/3477132.3483540

[8] Darren Brown, Margaret Burnett, Gregg Rothermel, Hamido Fujita, and Fumio Negoro. 2003. Generalizing WYSIWYT visual testing to screen transition languages. In *IEEE Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings. 2003.* IEEE, 203–210.

[9] Brian Burg, Richard Bailey, Amy J Ko, and Michael D Ernst. 2013. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. 473–484.

[10] Margaret Burnett, Simone Stumpf, Jamie Macbeth, Stephann Makri, Laura Beckwith, Irwin Kwan, Anicia Peters, and William Jernigan. 2016. GenderMag: A Method for Evaluating Software's Gender Inclusiveness. *Interacting with Computers* 28, 6 (2016), 760–787.

[11] Tsung-Hsiang Chang, Tom Yeh, and Robert C Miller. 2010. GUI testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1535–1544.

[12] Koen Claessen, Jonas Duregård, and Michal H. Palka. 2015. Generating constrained random data with uniform distribution. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S0956796815000143

[13] 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 (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, Montreal, Canada, 268–279. https://doi.org/10.1145/351240.351266

[14] Matthew C. Davis, Sangheon Choi, Sam Estep, Brad A. Myers, and Sunshine. 2023. NaNoFuzz: A Usable Tool for Automatic Test Generation.

[15] Kyle Thomas Dewey. 2017. *Automated Black Box Generation of Structured Inputs for Use in Software Testing*. University of California, Santa Barbara. https://api.semanticscholar.org/CorpusID:190004227

[16] Daniel Drew, Julie L Newcomb, William McGrath, Filip Maksimovic, David Mellis, and Björn Hartmann. 2016. The toastboard: Ubiquitous instrumentation and automated checking of breadboarded circuits. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. 677–686.

[17] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unifed Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM. Paper 315.

[18] Nicolas Dubien. 2024. fast-check. https://fast-check.dev/

[19] Segev Elazar Mittelman, Aviel Resnick, Ivan Perez, Alwyn E. Goodloe, and Leonidas Lampropoulos. 2023. Don't Go Down the Rabbit Hole: Reprioritizing Enumeration for Property-Based Testing. In *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium (Haskell 2023)*. Association for Computing Machinery, New York, NY, USA, 59–71. https://doi.org/10.1145/3609026.3609730

[20] Eduard Enoiu, Gerald Tukseferi, and Robert Feldt. 2020. Towards a model of testers' cognitive processes: Software testing as a problem solving approach. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 272–279.

[21] Rebecca Faust, Carlos Scheidegger, and Chris North. 2023. Aardvark: Comparative Visualization of Data Analysis Scripts. In *2023 IEEE Visualization in Data Science (VDS)*. IEEE, 30–38.

[22] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++} : Combining Incremental Steps of Fuzzing Research. https://www.usenix.org/conference/woot20/presentation/fioraldi

[23] Andrew Gallant. 2024. BurntSushi/quickcheck. https://github.com/BurntSushi/quickcheck original-date: 2014-03-09T07:29:09Z.

[24] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Transactions on Computer-Human Interaction* 22, 2 (March 2015), 7:1–7:35. https://doi.org/10.1145/2699751

[25] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. 22, 2 (2015), 7:1–7:35.

[26] Elena L. Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. 2018. Visualizing API Usage Examples at Scale. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3174154

[27] Elena L. Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. 2018. Visualizing API Usage Examples at Scale. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM. Paper 580.

[28] Harrison Goldstein, Joseph W Cutler, Daniel Dickstein, Benjamin C Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *International Conference on Software Engineering (ICSE)*.

[29] Harrison Goldstein, Joseph W Cutler, Adam Stein, Benjamin C Pierce, and Andrew Head. 2022. Some Problems with Properties, Vol. 1. https://harrisongoldste.in/papers/hatra2022.pdf

[30] Harrison Goldstein, Samantha Frohlich, Meng Wang, and Benjamin C. Pierce. 2023. Reflecting on Random Generation. In *Proceedings of ACM Programming Languages*. Seattle, WA, USA. https://doi.org/10.1145/3607842

[31] Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. 2021. Do Judge a Test by its Cover. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Nobuko Yoshida (Ed.). Springer International Publishing, Cham, 264–291. https://doi.org/10.1007/978-3-030-72019-3_10

[32] Harrison Goldstein and Benjamin C. Pierce. 2022. Parsing Randomness. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022), 128:89–128:113. https://doi.org/10.1145/3563291

[33] J. A. Hartigan and B. Kleiner. 1981. Mosaics for Contingency Tables. In *Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface*, William F.

Eddy (Ed.). Springer US, New York, NY, 268–273. https://doi.org/10.1007/978-1-4613-9464-8_37

[34] Björn Hartmann, Scott R Klemmer, Michael Bernstein, Leith Abdulla, Brandon Burr, Avi Robinson-Mosher, and Jennifer Gee. 2006. Reflective physical prototyping through integrated design, test, and analysis. In *Proceedings of the 19th annual ACM symposium on User interface software and technology*. 299–308.

[35] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Conference on Learning at Scale*. ACM, 89–98.

[36] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM. Paper 270.

[37] Hadi Hemmati. 2015. How effective are code coverage criteria?. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 151–156.

[38] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 281–292.

[39] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM. Paper 532.

[40] Fred Hohman, Andrew Head, Rich Caruana, Robert DeLine, and Steven M. Drucker. 2019. Gamut: A Design Probe to Understand How Data Scientists Understand Machine Learning Models. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM. Paper 579.

[41] Fred Hohman, Kanit Wongsuphasawat, Mary Beth Kery, and Kayur Patel. 2020. Understanding and Visualizing Data Iteration in Machine Learning. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM. Paper 50.

[42] Paul Holser. 2024. pholser/junit-quickcheck. https://github.com/pholser/junit-quickcheck original-date: 2010-10-18T22:33:36Z.

[43] John Hughes. 2007. QuickCheck Testing for Fun and Profit. In *Practical Aspects of Declarative Languages (Lecture Notes in Computer Science)*, Michael Hanus (Ed.). Springer, Berlin, Heidelberg, 1–32. https://doi.org/10.1007/978-3-540-69611-7_1

[44] John Hughes. 2016. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). Springer International Publishing, Cham, 169–186. https://doi.org/10.1007/978-3-319-30936-1_9

[45] John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. 2016. Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 135–145. https://doi.org/10.1109/ICST.2016.37

[46] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. Association for Computing Machinery, New York, NY, USA, 273–282. https://doi.org/10.1145/1101908.1101949

[47] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 654–665.

[48] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 737–745.

[49] Marcel R Karam and Trevor J Smedley. 2001. A testing methodology for a dataflow based visual programming language. In *Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments (Cat. No. 01TH8587)*. IEEE, 280–287.

[50] Amy J. Ko and Brad A. Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. Association for Computing Machinery, New York, NY, USA, 151–158. https://doi.org/10.1145/985692.985712

[51] Pavneet Singh Kochhar, David Lo, Julia Lawall, and Nachiappan Nagappan. 2017. Code Coverage and Postrelease Defects: A Large-Scale Study on Open Source Projects. *IEEE Transactions on Reliability* 66, 4 (Dec. 2017), 1213–1228. https://doi.org/10.1109/TR.2017.2727062 Conference Name: IEEE Transactions on Reliability.

[52] Pavneet Singh Kochhar, Xin Xia, and David Lo. 2019. Practitioners' views on good software testing practices. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 61–70.

[53] holger krekel. 2023. pytest: helps you write better programs — pytest documentation. https://docs.pytest.org/en/7.4.x/

[54] Shriram Krishnamurthi and Tim Nelson. 2019. The Human in Formal Methods. In *Formal Methods – The Next 30 Years (Lecture Notes in Computer Science)*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer International Publishing, Cham, 3–10. https://doi.org/10.1007/978-3-030-30942-8_1

[55] D. Richard Kuhn, James M. Higdon, James Lawrence, Raghu Kacker, and Yu Lei. 2012. Combinatorial Methods for Event Sequence Testing. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche (Eds.). IEEE Computer Society, 601–609. https://doi.org/10.1109/ICST.2012.147

[56] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage guided, property based testing. *PACMPL* 3, OOPSLA (2019), 181:1–181:29. https://doi.org/10.1145/3360607

[57] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2017. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30. https://dl.acm.org/doi/10.1145/3158133 Publisher: ACM New York, NY, USA.

[58] J Lawrance, Steven Clarke, Margaret Burnett, and Gregg Rothermel. 2005. How well do professional developers test with code coverage visualizations? an empirical study. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE, 53–60.

[59] Sorin Lerner. 2020. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–7.

[60] David R MacIver, Zac Hatfield-Dodds, and others. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891. https://joss.theoj.org/papers/10.21105/joss.01891.pdf

[61] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 291–301.

[62] Will McGrath, Daniel Drew, Jeremy Warner, Majeed Kazemitabaar, Mitchell Karchemsky, David Mellis, and Björn Hartmann. 2017. Bifröst: Visualizing and checking behavior of embedded systems across hardware and software. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 299–310.

[63] William McGrath, Jeremy Warner, Mitchell Karchemsky, Andrew Head, Daniel Drew, and Bjoern Hartmann. 2018. Wifröst: Bridging the information gap for debugging of networked embedded systems. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 447–455.

[64] André N Meyer, Earl T Barr, Christian Bird, and Thomas Zimmermann. 2019. Today was a good day: The daily life of software developers. *IEEE Transactions on Software Engineering* 47, 5 (2019), 863–880.

[65] Robert C Miller and Brad A Myers. 2001. Outlier finding: Focusing user attention on possible errors. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*. 81–90.

[66] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching processes for QuickCheck generators. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, Nicolas Wu (Ed.). ACM, 1–13. https://doi.org/10.1145/3242744.3242747

[67] Aaditya Naik, Jonathan Mendelson, Nathaniel Sands, Yuepeng Wang, Mayur Naik, and Mukund Raghothaman. 2021. Sporq: An interactive environment for exploring code using query-by-example. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 84–99.

[68] Tim Nelson, Elijah Rivera, Sam Soucie, Thomas Del Vecchio, John Wrenn, and Shriram Krishnamurthi. 2021. Automated, Targeted Testing of Property-Based Testing Predicates. *The Art, Science, and Engineering of Programming* 6, 2 (Nov. 2021), 10. https://doi.org/10.22152/programming-journal.org/2022/6/10 arXiv:2111.10414 [cs].

[69] Wode Ni, Joshua Sunshine, Vu Le, Sumit Gulwani, and Titus Barik. 2021. recode: A lightweight find-and-replace interaction in the ide for transforming code by example. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 258–269.

[70] Rickard Nilsson. 2024. ScalaCheck. https://scalacheck.org/

[71] Stephen Oney and Brad Myers. 2009. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 105–108. https://doi.org/10.1109/VLHCC.2009.5295287 ISSN: 1943-6106.

[72] Stephen Oney and Brad Myers. 2009. FireCrystal: Understanding Interactive Behaviors in Dynamic Web Pages. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE, 105–108.

[73] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. 2018. Mutation Testing Advances: An Analysis and Survey. *Advances in Computers* (Jan. 2018). http://dx.doi.org/10.1016/bs.adcom.2018.03.015

[74] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. In *Advances in Computers*, Atif M. Memon (Ed.). Vol. 112. Elsevier, 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015

[75] Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022. Computing correctly with inductive relations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 966–980. https://doi.org/10.1145/3519939.3523707

[76] Kevin Pu, Rainey Fu, Rui Dong, Xinyu Wang, Yan Chen, and Tovi Grossman. 2022. SemanticOn: Specifying content-based semantic conditions for web automation programs. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–16.

[77] Gregg Rothermel, Lixin Li, Christopher DuPuis, and Margaret 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*. IEEE, 198–207.

[78] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. 2018. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. In *Proceedings of the Conference on Computer-Supported Cooperative Work and Social Computing*. ACM. Article 150.

[79] Iflaah Salman, Pilar Rodriguez, Burak Turhan, Ayşe Tosun, and Arda Güreller. 2020. What leads to a confirmatory or disconfirmatory behavior of software testers? *IEEE Transactions on Software Engineering* 48, 4 (2020), 1351–1368.

[80] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C Pierce, and Leonidas Lampropoulos. 2023. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). *Proc. ACM Program. Lang.* 7 (2023). https://doi.org/10.1145/3607860

[81] Nischal Shrestha, Titus Barik, and Chris Parnin. 2021. Unravel: A fluent code explorer for data wrangling. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 198–207.

[82] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2020. Inputs from Hell: Learning Input Distributions for Grammar-Based Test Generation. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2020.3013716 Publisher: IEEE.

[83] C Spearman. 1904. The Proof and Measurement of Association between Two Things. *American Journal of Psychology* 15 (1904), 72–101. Publisher: University of Illinois Press, etc..

[84] Dominic Steinhöfel and Andreas Zeller. 2022. Input invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 583–594. https://doi.org/10.1145/3540250.3549139

[85] Neeraja Subrahmaniyan, Laura Beckwith, Valentina Grigoreanu, Margaret Burnett, Susan Wiedenbeck, Vaishnavi Narayanan, Karin Bucht, Russell Drummond, and Xiaoli Fern. 2008. Testing vs. code inspection vs. what else? Male and female end users' debugging strategies. In *Proceedings of the SIGCHI Conference on human factors in computing systems*. 617–626.

[86] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Björn Hartmann. 2017. TraceDiff: Debugging unexpected code behavior using trace divergences. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 107–115. https://doi.org/10.1109/VLHCC.2017.8103457 ISSN: 1943-6106.

[87] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Björn Hartmann. 2017. TraceDiff: Debugging unexpected code behavior using trace divergences. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE, 107–115.

[88] Jordan Walke. 2024. React. https://react.dev/

[89] April Yi Wang, Will Epperson, Robert A DeLine, and Steven M Drucker. 2022. Diff in the loop: Supporting data comparison in exploratory data analysis. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–10.

[90] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. 2021. Falx: Synthesis-powered visualization authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.

[91] Aaron Wilson, Margaret Burnett, Laura Beckwith, Orion Granatir, Ledah Casburn, Curtis Cook, Mike Durham, and Gregg Rothermel. 2003. Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '03)*. Association for Computing Machinery, New York, NY, USA, 305–312. https://doi.org/10.1145/642611.642665

[92] Litao Yan, Elena L. Glassman, and Tianyi Zhang. 2021. Visualizing Examples of Deep Neural Networks at Scale. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3411764.3445654

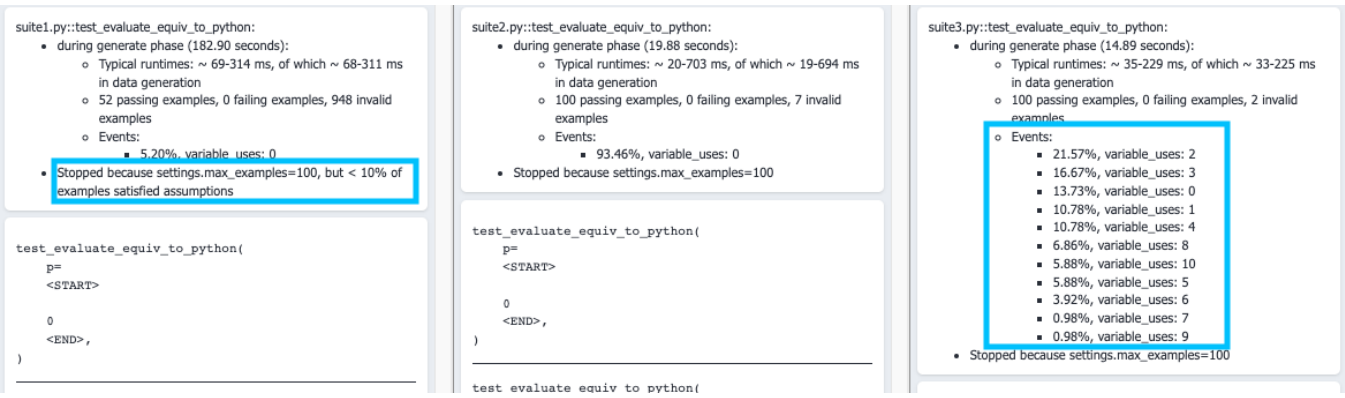
[93] Litao Yan, Elena L Glassman, and Tianyi Zhang. 2021. Visualizing examples of deep neural networks at scale. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–14.

[94] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. 2020. Interactive program synthesis by augmented examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 627–648.

[95] Valerie Zhao, Lefan Zhang, Bo Wang, Michael L Littman, Shan Lu, and Blase Ur. 2021. Understanding trigger-action programs through novel visualizations of program differences. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–17.

A TABLE OF "LUCIDITY" FEATURES IN EXISTING FRAMEWORKS

Table 1: Breakdown of existing lucidity features in popular PBT frameworks.

	Hypothesis	fast-check	quickcheck (Rust)	ScalaCheck	junit-quickcheck	QuickCheck (Haskell)
# Tests Run	✓			✓	✓	✓
# Tests Discarded	✓			✓	✓	✓
Events / Labels	✓			✓	✓	✓
Generation Time	✓					
Warnings	✓					

B CONTROL VIEW FOR PYTHON INTERPRETER STUDY TASK