Lucid Property-Based Testing

Anonymous Author(s)

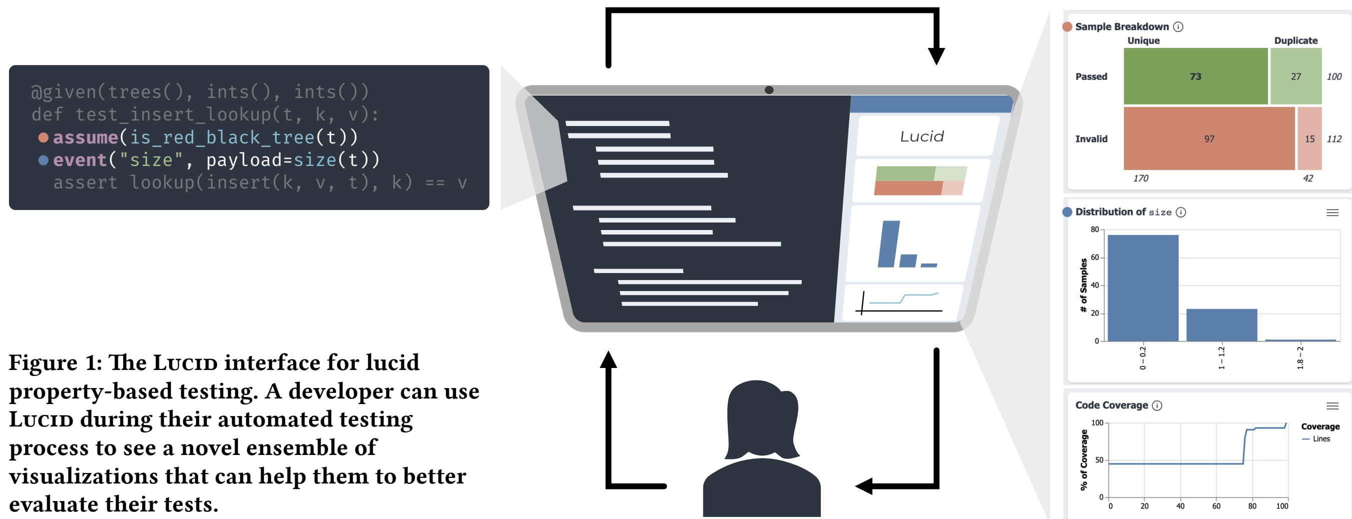


Figure 1: The Lucid interface for lucid property-based testing. A developer can use Lucid during their automated testing process to see a novel ensemble of visualizations that can help them to better evaluate their tests.

ABSTRACT

Software developers increasingly rely on automated methods to assess the correctness of their code. One such method is property-based testing (PBT), wherein a test harness generates hundreds or thousands of inputs and checks the outputs of the program on those inputs using parametric properties. Though powerful, PBT itself induces a sizable gulf of evaluation: developers need to put in nontrivial effort to understand how well the different test inputs exercise the software under test. To bridge this gulf, we propose *lucid property-based testing*, an interaction paradigm to support sensemaking for PBT effectiveness. Guided by a formative design exploration, we designed and implemented Lucid, an interface that supports lucid PBT through interactive, configurable views of test behavior with tight integrations into modern developer testing workflow. These views help developers explore global testing behavior and individual example attributes alike. To accelerate the development of powerful, interactive PBT tools, we define a standard for PBT test reporting and integrate it with a widely used PBT library. A self-guided online usability study revealed that Lucid's views lead to better assessments of software testing effectiveness.

KEYWORDS

Randomized testing, property-based testing (PBT), visual feedback, multiple program executions

ACM Reference Format:
Anonymous Author(s). 2024. Lucid Property-Based Testing. 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, 16 pages.

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

1 INTRODUCTION

Software developers work hard to build systems that behave as intended. But software is rarely 100% correct when first implemented, so developers also write tests to validate their work, detect bugs, and check that bugs stay fixed. Traditionally, these tests take the form of manually written "example-based" tests, where developers write out specific sample inputs together with expected outputs; but this process is labor intensive and can miss bugs when the sample inputs are too sparse. An alternative approach uses automated techniques to supplement or replace example-based tests. One such technique is *property-based testing* (PBT), which automatically samples many program inputs from a random distribution and checks, for each one, that the system's behavior satisfies a set of developer-provided properties. Used well, this leads to testing that is more thorough and less laborious; indeed, PBT has proven effective in identifying subtle bugs in a wide variety of real-world settings, including telecommunications software [3], replicated file and key-value stores [6, 39], automotive software [4], and other complex systems [38].

Of course, automation comes with tradeoffs, and PBT is no exception. In PBT, there are often too many automatically generated test inputs for a developer to examine every one, creating a gulf of evaluation for test suite quality. Indeed, in a recent study of the human factors of PBT [24], developers reported having difficulty understanding what was really being tested.

For example, suppose a developer is testing some mathematical function using randomly generated floating-point numbers. The developer might have a variety of questions about their test suite quality. They might ask if the distribution is broad enough (e.g., is it stuck between `0` and `1`), or too broad (e.g., does it span all possible floats, even if the function can only take positive ones). Or they may

wonder if the distribution misses corner-cases like `0` or `-1`. Perhaps most importantly, they may want to know if the data generator produces too many malformed or invalid test inputs (e.g., `NaN`) that cannot be used for testing at all. State-of-the-art PBT tooling does not give adequate tools for answering these kinds of questions: any of these erroneous situations could go unnoticed because necessary information is not apparent to the user. As a result, developers may not realize that their tests are not thoroughly exercising some important system behaviors.

This gulf of evaluation presents an opportunity to rethink user interfaces for testing. HCI has made strides in helping developers make sense of large amounts of structured program data, whether by revealing patterns that manifest in many programs [20, 22, 30, 87] or comparing the behavior of program variants [78, 81, 90]. As developers adopt PBT, it is critical to tackle the related problem of helping programmers understand a summary of hundreds or thousands of executions of a single test.

To address this problem, we propose *lucid property-based testing*, a novel interaction paradigm for supporting sensemaking and exploration of distributions of test inputs, which is informed by a review of PBT usability and evolved through iterative design with the help of experienced PBT users. We embody the lucid PBT approach in an interface called Lucid, which gives developers a novel ensemble of visualizations, fine-tuned to the PBT setting and enriched with lightweight affordances to support exploration. These provide high-level insight about the distribution of test inputs as well as various aspects of test efficiency (see Figure 1). Lucid also supports visualization and rapid drill-down into user-defined attributes of input data, taking advantage of existing hooks in PBT libraries.

To understand whether Lucid actually changes how developers understand their tests, we conducted a 40-participant, self-guided, online study. In this study, participants were asked to view test distributions and rank them according to their power to identify program defects. Compared to using a standard PBT tool, using Lucid led to better judgments about the bug-finding power of test distributions.

To encourage broad adoption of lucid property-based testing, we further define OpenLucid, a standard format for reporting results of PBT. When a PBT framework generates data in this format, its results can be viewed in Lucid and perhaps other interfaces supporting the same standard in the future. We integrated OpenLucid into the main branch of Hypothesis [50], the most widely-used PBT framework, enabling lucid property-based testing with Hypothesis and showing the way forward for other frameworks.

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

- We articulate design considerations for supporting *lucid property-based testing*, motivated by a formative study with experienced PBT users. (§4)
- We present the design of Lucid, an interface that helps developers evaluate the quality of their testing with an ensemble of visualizations fine-tuned to PBT with lightweight affordances to support exploration. (§5)
- We define the OpenLucid format for collecting and reporting PBT data to help standardize lucid PBT across different PBT frameworks. (§6)
- We evaluate Lucid in an online study, demonstrating that Lucid guides developers to significantly better assessments of test suite effectiveness. (§7)

We conclude with directions for future work (§8), including other automated testing disciplines that can benefit from lucid interfaces.

2 BACKGROUND

We begin by describing property-based testing and reviewing what is known about its usability and contexts of use.

2.1 Property-Based Testing

In traditional unit testing, developers think up examples that demonstrate the ways a function is supposed to behave, writing one test for each envisioned behavior. For example, this unit test checks that inserting a key `"k"` and value `0` into a red–black tree [84] and then looking up `"k"` results in the value `0`:

```
def test_insert_example():
    t = Empty()
    assert lookup(insert("k", 0, t), "k") == 0
```

If one wanted to test more thoroughly, they could painstakingly write dozens of tests like this for many different example trees, keys, and values. Property-based testing offers an alternative, succinct way to express many tests at once:

```
@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 is written in Hypothesis [50], a popular PBT library in Python. It randomly generates triples of trees, keys, and values, and for each triple, checks a parameterized property that resembles a unit-test assertion. This single test specification represents an infinite collection of concrete individual tests, and using it can lead to more thorough testing (compared to a unit test suite), since the random generator may produce examples the user might not have thought of.

2.2 PBT Process and Pitfalls

At its core, the practice of PBT involves three distinct steps: defining executable properties, constructing random input generators, and reviewing the result of checking these properties against a large number of sampled inputs; challenges can arise at any of these stages. We are focused here on the third stage: helping developers review the results of testing, in part to support the (often iterative) process of refining and improving the generators constructed in the second step. For instance, in the example above, a developer might accidentally write a `trees()` generator that only produces the `Empty()` tree, in which case their property will be checked only against a single test input (over and over). Or, if the generator's strategy is not quite so broken but still too naïve, it might fail to produce very many trees that actually pass the `assume(is_red_black_tree(t))` guard.

In cases like these, developers need to remember that, although all their tests are succeeding, this does not necessarily mean their code is correct [48]: they may need to improve their generators to start seeing failing tests. Unfortunately, with conventional PBT

tools, developers may feel they don't have easy access to this knowledge [24]. While the programming languages community is continually developing better techniques for generating well-distributed inputs [27, 47, 57, 75, etc.], developers still need to be able to check that the generators they are using are actually fit for the job.

3 RELATED WORK

In this section we situate our work on lucid PBT within the larger area of programming tools research.

3.1 Current Affordances

What support do PBT frameworks provide today for developers to inspect test input distributions? We surveyed the state of practice in the most popular PBT frameworks (by GitHub stars) across six different languages: Python [50], TypeScript / JavaScript [17], Rust [19], Scala [61], Java [35], and Haskell [12]. These frameworks provide users with the following kinds of information. (A detailed comparison of framework features can be found in Appendix A.)

Raw Examples. All of these frameworks could print generated inputs to the terminal. Some (3/6) provided a flag or option to do so; the others did not provide this feature natively, although users might simply print examples to the terminal themselves.

Number of Tests Run vs. Discarded. Many frameworks (4/6) report how many examples were run vs. discarded (because they did not pass a quality filter), sometimes (2/6) hidden behind a command line flag.

Event / Label Aggregation. Many frameworks (4/6) could report aggregates of user-defined features of the data distribution—e.g., lengths of generated lists. Information about such features typically appeared in a simple textual list or table, as in this example from QuickCheck [76]:

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

I.e., among the generated lists for some test run, 7% were 7 elements long, 6% were 3 elements long, etc.

Time. One framework reported how long the test run took.

Warnings. One framework provided warnings about test distributions, in particular warning users when their generators produced a very high proportion of discarded examples.

The affordances for evaluation available in existing frameworks are situationally useful, but inconsistently implemented and incomplete. In §5 and §6 we discuss how Lucid improves on the state of the art.

3.2 Interactive Tools for Testing

Some of the earliest research on improved interfaces for testing focused on spreadsheets. Rothermel et al. [67] proposed a model of testing called "what you see is what you test" (WYSIWYT), wherein users "test" their spreadsheet by checking values that they see and marking them as correct. This approach has appeared in many domains of programming, including visual dataflow [43] and screen transition languages [7]. Complementary to WYSIWYT are features that encourage programmers' curiosity [85], for instance by detecting and calling attention to likely anomalies [56, 85].

Many of the testing tools developed by the HCI community have sought to accelerate manual testing with rich, explorable traces of program behavior [8, 15, 52, 53, 62]. These tools instrument a program, record its behavior during execution, and then provide visualizations and augmentations to source code to help programmers pinpoint what is going wrong in their code. Tools can also help programmers create automated tests from user demonstrations. For instance, Sikuli Test [10] lets application developers create automated tests of interfaces by demonstrating a usage flow with the interface and then entering assertions of what interface elements should or should not be on the screen at the end of the flow.

Recent research has explored new ways to bring users into the loop of randomized testing. One research system, NaNoFuzz [14], shows programmers examples of program outputs and helps them to notice problematic results like `NaN` or crash failures. NaNoFuzz is superficially the closest comparison available for Lucid, but the two serve different, complementary purposes. NanoFuzz's strengths reside in calling attention to failures; Lucid's strengths reside in exposing patterns in input distributions. One could imagine a user leveraging both in concert during the testing process.

3.3 Making Sense of Program Executions

In a broad sense, Lucid's aim is to help developers reason about the behavior of a program across many executions. This problem has been explored by the HCI community. Tools have been developed to reveal the behavior of a program over many synthesized examples [89], and of an expression over many loops [32, 42, 49, 72]. The problem of understanding input distributions has been of interest in the area of AI interpretability, where tools have been built to support inspection of input distributions and corresponding outputs (e.g., [33, 34]). Lucid's aim is to tailor data views and exploration mechanisms to tightly fit the concerns and context of randomized testing with professional-grade software and potentially-complex inputs (e.g., logs, trees).

Prior work has sought to help programmers make sense of similarities and differences across sets of programs. Some of these tools cluster programs on the basis of aspects, semantics, or structure [21, 23, 30, 88]. Others highlight differences in the source and/or behavior of program variants [68, 78, 81, 90]. Lucid itself does some lightweight clustering of test cases (in this case, input examples), and affordances for program differencing could be brought to Lucid to help programmers pinpoint where some instantiations of a property fail and others succeed.

3.4 Formal Methods in the Editor

Property-based testing can be seen as a kind of *lightweight formal method* [86], in that it allows programmers to specify precisely the behavior of their program and then verify that the specification is satisfied. Lucid joins a family of research projects that bring formal methods into the interactive editing experience, whether to support repetitive edits [51, 60], code search [58], program synthesis [16, 64, 82, 89], or bidirectional editing of programs and outputs [31].

4 FORMATIVE RESEARCH

Our vision of lucid PBT is informed by formative research into the user experience of PBT. Below, we describe our methods for formative research (§4.1), followed by a crystallization of user needs (§4.2) and a set of design considerations for lucid PBT (§4.3).

4.1 Methods

To better understand what developers need in understanding their PBT distributions, we collected two kinds of data.

4.1.1 Review of related work. We reviewed user needs relating to evaluating testing effectiveness as discussed at length in Goldstein et al. [24]'s recent paper on the human factors of PBT.

4.1.2 Iterative design feedback. As we developed Lucid, we continually sought and integrated feedback on its design from experienced users of PBT. We recruited 5 such users through X (formerly Twitter) and our personal networks. We refer to them as P1–P5.

For each of these users, we conducted a 1-hour observation and interview session. Each session was split into two parts. In the first part, participants showed us PBT tests they had written, described those tests, and answered questions about how they evaluate (or could evaluate) whether those tests are effective. In the second part, participants installed our then-current prototype and used it to explore the effectiveness of their own tests.[1] Study sessions were staggered throughout the design process. We altered the design to incorporate feedback after each session.

Initial prototype. All Lucid prototypes were developed as VS-Code [55] extensions. All prototypes focused on providing visual summaries of PBT data in a web view pane in the editor. The initial Lucid prototype was informed by observations from Goldstein et al. [24]'s study and the authors' experiences using and building PBT tools. The initial prototype was published as prior non-archival work, an anonymized version of which is available as supplemental material; the prototype summarized the following aspects of test data:

- "Number of Unique Inputs": New PBT users are sometimes surprised that their test harness produces duplicate data. Knowing how many unique inputs were tested is therefore one important signal of the test harness' efficiency.
- "Proportion of Valid Inputs": As discussed in §2.2, PBT test harnesses sometimes discard data that does not satisfy necessary preconditions. Developers need to know how much of the data is discarded and how much is kept.
- "Size Distribution": Testers need to keep track of the size of their inputs. It is commonly believed in the PBT community that software can be tested well by exhaustive sets of small inputs (i.e., the *small scope hypothesis* [2]), and alternatively, that large tests have a *combinatorial advantage* [37] in finding more bugs.[2] Whichever viewpoint a tester subscribes to, it is important to know the sizes of inputs.

Analysis. Interviews were automatically transcribed by video conferencing software[3], and analyzed via thematic analysis [5].

4.2 Testing Goals and Strategies

The first result of our formative research was a clarification of PBT users' goals and strategies when they were attempting to determine the effectiveness of their tests. One might imagine that testing effectiveness could be measured by the proportion of bugs found, but this is a fantastical measure: if we had it, we would know what all the bugs are and wouldn't need to do any testing! As we found in our study sessions, developers pay attention to proxy metrics to gain confidence in their test suite. Ideally, PBT tools will surface these metrics. Here, we discuss the various metrics that developers paid attention to and how they measured them.

4.2.1 Test Input Distribution. Participants reported checking that their distributions covered potential edge cases like $x = 0$, $x = -1$, or $x =$ `Integer.MAX_VALUE` (P2), which are widely understood as bug-triggering values. They also checked that their distributions covered regions 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 [26, 46].

Multiple participants (P1, P3, P4) wanted to know that their test data was realistic. Their justification was that the most important bugs are the ones that users were likely to hit. Another participant (P5) wanted their test data to be uniformly distributed across a space of values. They thought that this would make it easier for them to estimate the probability that there was still a bug in the program. Whether to test with realistic or uniform distributions is a topic of debate in the literature, with some tools favoring uniformity [11, 57] and others realism [73]. In either case, developers should be able to see the shape of the distribution.

Participants used a combination of strategies to review these proxy metrics of test quality. Some (P1, P3, P5) read through the list of examples. Others (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, both to understand coverage of attributes of interest (e.g., how often does a particular variant of an enumerated type appear). As we show in §2, while some PBT frameworks provide views of distributions of user-defined input features, they 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) mentioned coverage of the system under test (e.g., line coverage, branch coverage, etc.) was an important proxy metric. 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"). This view is supported by the literature, which suggests that coverage alone does not guarantee testing success [45].

4.2.3 Test Performance. Finally, two participants (P1, P2) discussed timing performance as an important proxy for testing effectiveness. They argued that they have a limited time in which to run tests (e.g.,

[1] P5 showed us older code that they no longer had the test runner for, so they only saw Lucid running on our examples.

[2] The truth seems to be that each of these viewpoints is correct in some situations; a recent study [71] has a nice discussion.

[3] P3's interview audio was lost due to technical difficulties, so we instead analyzed the notes we took during their interview.

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) will exercise the system better. They measured performance with the help of tools built into the PBT framework.

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

Our formative research further clarified what is required of usable tools for understanding PBT effectiveness. We call these requirements our five design considerations for lucid property-based testing. They are:

Visual Feedback Our goal to provide better visual feedback from testing arose from Goldstein et al. [24]'s study and was validated by participants. Most participants (P1, P2, P3, P5) appreciated the interface's visual charts, stating that the visual charts are "a lot easier to digest than" the built-in statistics printed by Hypothesis (P2). The previous section (§4.2) clarifies the specific proxy metrics that developers were interested in visualizing.

Workflow Integration Our initial prototype was built to have tight editor integration. It could be installed into VSCode in one step, and updated live as code changed. But, while some participants validated this choice (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. We therefore refocused on workflow integration instead of editor integration, and re-architected our design so that it could plug into other editing workflows.

Customizability Participants found that the default set of visualizations was a good start (P1, P2, P3, P5), but they also suggested a slew of other visualizations that they thought might improve their testing evaluation. Many of these visualizations (e.g., code coverage (P1, P3, P5) and timing (P1)—see §5.2) were integrated into Lucid. What we could not do was add views that summarized the interesting attributes of each person's data: every testing domain was different. Thus, tools should be customizable so developers can acquire visual feedback for the information that is important in their testing domain.

Details on Demand Almost all participants (P1, P2, P3, P4) expressed a desire to dig deeper into the visualizations they were presented. This means that lucid PBT interfaces should provide ways for developers to look deeper into the details of the data that is being displayed by the visual interfaces.

Standardization Participants used PBT in multiple programming languages, including Python (P1, P2, P3, P4), Java (P4), and Haskell (P5). We posit that to improve the testing experience for all of these languages and their PBT frameworks without significant duplicated effort, lucid PBT tools need to standardize the way they communicate with PBT frameworks. Since PBT frameworks largely implement the same test loop, despite superficial implementation differences, this standardization seems technically feasible.

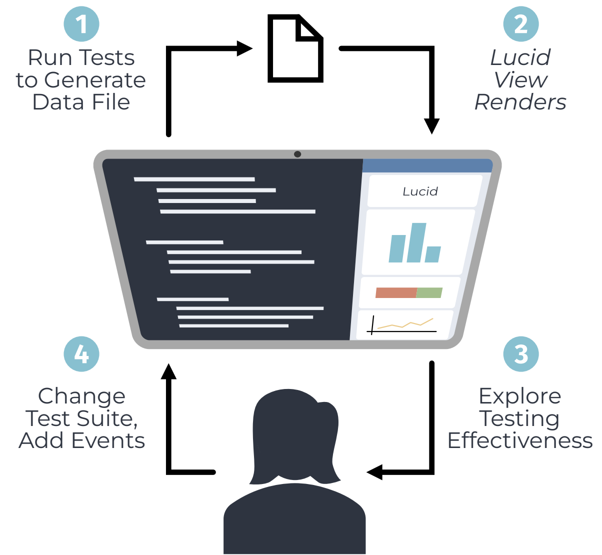


Figure 2: The Lucid interaction loop.

5 SYSTEM

In this section, we describe our vision of lucid property-based testing, bringing together the principles we identified in §4.3. We describe the interaction model that we imagine for Lucid (§5.1), Lucid's visual displays that answer PBT users' questions (§5.2), and integrations with PBT frameworks that support easy use and configuration of displays (§5.3).

5.1 Interaction Model

We envision user interactions with Lucid to follow roughly the steps outlined in Figure 2.

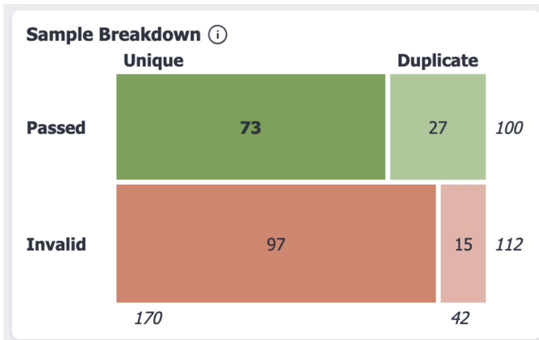
1. At the start of the loop, the developer runs their tests, and the test framework (e.g., Hypothesis) collects relevant data into a OpenLucid log (we discuss the details of this format in §6).
2. Once the data has been logged, the user sees Lucid render an interface with variety of visualizations (see §5.2).
3. 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 views to seek details 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 can customize their Lucid visualizations or make changes to their test (e.g., random generation strategies or Hypothesis parameters) before going back around the loop.

5.2 Visual Feedback

The Lucid interface presents the user with a novel ensemble of visualizations that are fine-tuned to the PBT setting and enriched with lightweight affordances to support exploration. We describe

these visualizations in the context of the kinds of questions they answer for developers.

5.2.1 How many meaningful tests were run? Perhaps the most important thing for a developer to know about a test run is how many meaningful examples were tested. Lucid communicates this information through the "Sample Breakdown" chart:

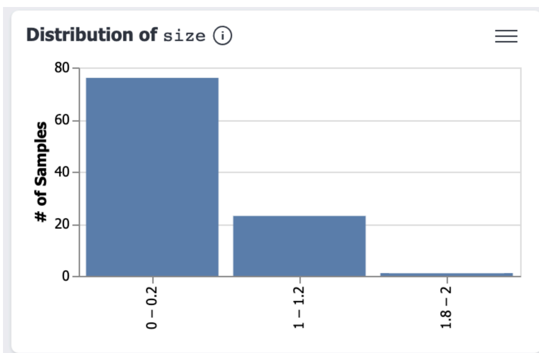


The chart communicates a high-level understanding of how many test inputs were sampled versus how many were run (because they were found to be "valid"). Ideally, the entire chart would be taken up by the dark green "Unique / Passed" bar. If the "Invalid" bars are a large portion of the chart's height or the "Duplicate" bars are a large portion of the width, the developer can see that it might be worth investing time in a generation strategy that is better calibrated for the property at hand. (If any tests had failed, there would be two more horizontal bars with the label "Failed.")

The use of a mosaic chart [28] here allows Lucid to communicate information about validity and uniqueness in a single chart. We chose this chart after feedback from study participants suggested that seeing validity and uniqueness metrics separately made it hard to tell when and how they overlapped.

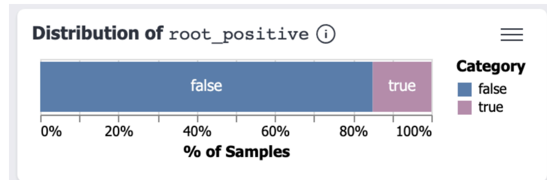
5.2.2 How are test inputs distributed? After checking the high-level breakdown of test inputs, the next questions in the user's mind will likely be about the *distribution* of inputs used to test their property. Since test inputs are structured objects (e.g., trees, event logs, ...), it is difficult to observe their distribution directly: what would it even mean to plot a distribution of trees? Instead, the developer can visualize *features* of the distribution by plotting numerical or categorical data extracted from their test inputs.

For example, the following chart shows a distribution of `sizes` projected from a distribution of red–black trees:



Charts like these give developers windows into their distributions that are much easier to interpret than either the raw examples or the statistics reported by frameworks like Hypothesis: the chart above, for example, shows that the distribution skews quite small (actually, most trees are size 0!), which is a significant problem for test success (see in §4).

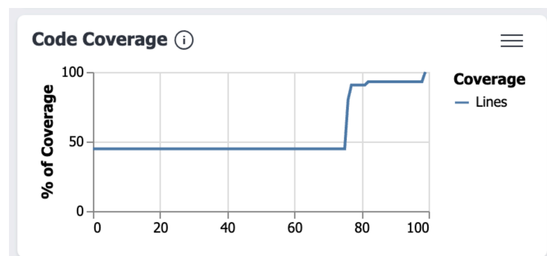
Distributions for categorical features (e.g., whether the value at the root of a red-black tree is positive or not) are displayed in a different format:



Categorical feature charts can be especially useful for helping developers understand whether there are portions of the broader input space that their tests are currently missing. In this case, the developer may want to check on why so few roots are positive—in fact, it is because an empty tree does not have a positive root, and the distribution is full of empty trees!

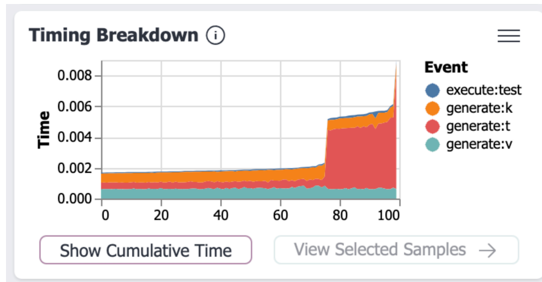
Our formative research suggested that just these two kinds of charts covered the kinds of projections that developers cared about. In fact, participants seemed concerned that adding more kinds of feature charts could be distracting; they felt they may waste time trying to find data to plot for the sake of using the charts, rather than plotting the few signals that would actually help with their understanding. In §6.3 we describe how developers can design their own visualizations outside of Lucid if needed.

5.2.3 How did the tests execute overall? The previous visualizations show information about test inputs, but developers may also have higher-level questions about what happened during testing. For example, early users, including formative research participants, asked for ways to visualize code coverage for their properties. Lucid provides the following coverage chart:



This Lucid chart shows the total code coverage achieved over the course of the test run. Note that this example is from a very small codebase, so there were really only a few disjoint paths to cover. Big jumps (around the 1st and 75th inputs) indicate inputs that meaningfully improved code coverage, whereas plateaus indicate periods where no new code was reached. As discussed in §4, code coverage is an incomplete way to measure testing success, but knowing that the first 70+ test inputs all covered the same lines suggests that the generation strategy may spend too long exploring a particular type of input.

Lucid also provides charts with timing feedback, again answering a high-level question about execution that was requested by formative research participants:



The chart above shows that a majority of inputs execute quite quickly (less than 0.002 seconds) but that some twice or three times that. For the most expensive tests, the red area, signifying the time it takes to generate trees, is the largest. While users did request this chart, we are not clear how useful it is in practice. In our evaluation study (§7.1), participants did not generally find it helpful. However, the timing data can be used to corroborate and expand on information from other charts. For example, notice how the timing breakdown above actually mirrors the `size` chart from the previous section. The combination of these charts suggests that larger trees take much longer to generate, which suggests as trade-off that a developer should be aware of.

5.2.4 *What test inputs were actually generated?*

Although much of the point of Lucid is to avoid programmers needing to sift through individual test input examples, Lucid does make those examples available, in line with the lucid PBT consideration of details on demand:

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

```
test_insert_lookup(
    t=T(c=Black(), l=E(), k=0, v=0, r=E()),
    k=0,                                       2×
    v=0,
```

Each example in the view shows a textual representation of the generated sample that can be expanded to see metadata like execution duration and code coverage for the individual example. Examples are grouped, so that identical examples are only shown once; this manifests in the "3x" and "2x" annotations shown in the above screenshot. This grouping aligns better with the lucid PBT idea of visual feedback, and it cuts down on clutter.

The main way a user reaches the example view is by clicking on one of the selectable bars of the sample breakdown or feature distribution charts. The user can dig into the data to answer questions about why a chart looks a certain way (e.g., if they want to explore why so few of the red–black tree's root nodes are positive). Secondarily, the example view can be used to search for particular examples to make sure they appear in the sample (e.g., important corner cases that indicate thorough testing).

5.3 Reactivity and Customizability

The visualizations provided by Lucid are reactive and customizable, allowing them to integrate neatly into the developer's workflow as dictated by the lucid PBT design considerations.

5.3.1 *Reactivity.*

Reactivity has been incorporated into an astonishing variety of programming tools (see the review by Rein et al. [66]). It is a common feature of many modern developer tools—two modern examples are Create React App [13], which reloads a web app on each source change and pytest-watch [65], one of many testing harnesses that live-reruns tests upon code changes. When run as a VSCode extension, Lucid automatically refreshes the view when the user's tests re-run. When used in conjunction with a test suite watcher (e.g., pytest-watch, which reruns Hypothesis tests when the test file is saved) this yields an end-to-end experience with "level 3 liveness" on Tanimoto's hierarchy of levels of liveness [79].

5.3.2 *Customizability.*

Specifically, in step (4) of the Lucid loop, the user can tweak their testing code in ways that change the visualizations that are shown the next time around the loop.[4]

Assumptions. As discussed in §2 with the red–black tree example, developers make assumptions about on what inputs are valid for their property. Concretely, this happens via the Hypothesis `assume` function; for example:

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

The `assume` function 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. In the sample breakdown, inputs that break assumptions are shown as "Invalid."

Events. Hypothesis also has a feature called "events" to label property executions interesting. For example, the programmer might write:

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

and then Hypothesis would output "`hit_condition: 42%`." To support richer visual displays of features, we extended the Hypothesis API (with the support of the Hypothesis developers) to allow events to include "payloads" that correspond to the numerical and categorical features in the feature charts above. Adding an event to the above property gives:

```
def test_insert_lookup(t, k, v):
    event("size", payload=size(t))
    assume(is_red_black_tree(t))
    assert lookup(insert(k, v, t), k) == v
```

These user events correspond to feature charts: the one shown here generates the `size` chart shown in the previous section.

By reusing Hypothesis's existing idioms for assumptions and events, Lucid hooks into existing developer workflows and makes them more powerful. The same pattern also applies to other PBT frameworks.

[4] While Lucid works with many PBT frameworks, we describe these customizations 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.

```
{
  line_type: "example",
  run_start: number,
  property: string,
  status: "passed" | "failed" | "discarded",
  representation: string,
  features: {[key: string]: number | string}
  coverage: ...,
  timing: ...,
  ...
}
```

Figure 3: The OpenLucid line format.

6 IMPLEMENTATION

In this section, we outline the implementation of the Lucid interface. We begin with the mechanics of the system itself (§6.1), but the most interesting part is the standardized OpenLucid format that PBT frameworks use to send data to Lucid (§6.2). In §6.3 we explain how the Lucid architecture makes it easy to extend the lucid PBT ecosystem.

6.1 UI Implementation

At the implementation level, Lucid is a web-based interface that is easy to integrate into existing PBT frameworks.

6.1.1 React Application. Lucid is a React [80] web application that consumes raw data about the results of one or more PBT runs and produces interactive visualizations to help users make sense of the underlying data. The primary way to use Lucid is in the context of an extension for VSCode that shows the interface alongside the tests that it pertains to, but it is also available as a standalone webpage 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.)

The mosaic chart described in §5.2.1 is implemented with custom HTML and CSS, but all other charts and visualizations are generated with Vega-Lite [70]. Vega-Lite has good default layout algorithms for the types of data we care about.

6.1.2 Framework Integration. As discussed in §5.1, we worked with the Hypothesis developers to make a few small changes to enable Lucid; other PBT tools require similar changes. The Hypothesis developers added callback to capture data on each test run, and it was easy to use this callback to produce data for Lucid. This data is dumped in the OpenLucid format, which we discuss in §6.2.

In Hypothesis specifically, we also adapted the `event` function to have a richer API, described in §5.3.2.

6.2 OpenLucid Data Format

Lucid uses an open standard that PBT frameworks can use to integrate with lucid PBT tools.

OpenLucid is based on JSON Lines:[5] each line in 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 represents a single test case. The JSON schema in Figure 3 defines the format of a single example line. Each example has a `run_start` timestamp, used to group examples from the same run of a property and disambiguate between multiple runs of data that are stored in the same file. The `property` field names the property being tested (extracted from language internals) and the `status` field says whether this example `"passed"`, meaning the property passed, or `"failed"` indicating that the example is a counterexample to the property, or `"discarded"` meaning that the value did not pass assumptions. The `representation` is a human-readable string describing the example (e.g., as produced by a class's `__repr__` in Python). Finally, the `features` reflect the data collected for user-defined events.

The full format includes a few extra fields, including some human-readable details (e.g., to explain why a particular value was discarded), optional fields naming the particular generator that was used to produce a value, and a freeform metadata field for any additional information that might be useful in the example view.

6.3 Expanding the Ecosystem

The clean divide between Lucid and OpenLucid means that PBT frameworks require only the modest work of implementing OpenLucid to get access to the visualizations implemented by Lucid, and conversely that front ends other than Lucid will work with any PBT tool that implements OpenLucid.

6.3.1 Supporting New Frameworks. Supporting a new PBT framework is as simple as extending it with some lightweight logging infrastructure. Framework developers can start small: supporting just five fields—`type`, `run_start`, `property`, `status`, and `representation`—is enough to enable a substantial portion of the features of lucid PBT. After that, adding `features` will enable user control of visualizations; `coverage` and `timing` may be harder to implement in some programming languages, but worthwhile to support the full breadth of Lucid charts.

So far, support for OpenLucid exists in Hypothesis, Haskell QuickCheck, and OCaml's base-quickcheck. Our minimal Haskell QuickCheck implementation is an external library comprising about 100 lines of code and took an afternoon to write.

6.3.2 Adding New Analyses. Basing OpenLucid 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 easy to process the data with other tools. For example, getting started visualizing OpenLucid data in a Jupyter notebook requires two lines of code

```python
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 cannot be generated by adding an assumption or event, they can simply load the data into a notebook and start building their own analyses.

In the open-source community, we also expect that developers may find entirely new use-cases for OpenLucid data that are not tied to Lucid. For example, OpenLucid data could be used for reporting testing performance to other developers or managers (this use-case that was mentioned by participants during our formative exploration).

[5] https://jsonlines.org

929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986

7 EVALUATION

In this section, we evaluate our approach to lucid PBT. §7.1 presents an online self-guided study to assess Lucid's impact on users' judgments about the quality of test suites. §7.2 describes the concrete impact that lucid PBT has already had through identifying bugs in the Hypothesis testing framework itself.

7.1 Online Study

We chose this study to validate what we saw as the most critical question about the design: whether the kinds of visual feedback offered by Lucid led to improved understanding of test suites. We regarded this question as most critical because we had relatively less confidence in the effectiveness of visual feedback for helping find bugs than in other aspects of the Lucid design—indeed, it is a tall order for *any* kind of feedback to provide an effective proxy for the bug-finding power of tests. (By contrast, we felt our choices around customizability, workflow integration, details on demand, and standardization were already on solid ground—these choices were more conservative, and had previously received positive feedback from developers and PBT tool builders.)

Accordingly, we designed a study to address the following research questions:

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

RQ2 Which aspects of Lucid best support sensemaking about test results?

To go beyond qualitative feedback alone, we designed the study to support statistical inference about whether we had improved judgments about test distributions. This led us a self-guided, online usability study that centered on focused usage of Lucid's visual displays. The study allowed us to collect sufficiently many responses from diverse and sufficiently-qualified programmers to support the analysis we wanted.

7.1.1 Study Population. We recruited study participants both from social media users on X (formerly Twitter) and Mastodon and from graduate and undergraduate students in the computer science department of a large university, aiming to recruit a diverse set of programmers ranging from relative beginners with no PBT experience to experts who may have some exposure.

In all, we recruited 44 participants. 4 responses were discarded because they did not correctly answer our screening questions, leaving 40 valid responses. All but one of these reported that they were at least proficient with Python, with 12 self-reporting as advanced and 9 as expert. Half reported being beginners at PBT, 13 proficient, 6 advanced, and 0 experts. Almost all participants reported being inexperienced with the Python Hypothesis framework; only 7 reporting being proficient. To summarize, the average participant had experience with Python but not PBT, and if they did know about PBT it was often not via Hypothesis.

When reporting education level, 4 participants had a high school diploma, 15 an undergraduate degree, and 20 a graduate degree. The majority of participants (24) described themselves as students; 7 were engineers; 3 were professors; 6 had other occupations. 28 participants self-identified as male, 5 as female, 2 as another gender, and 5 did not specify.

We discuss the limitations of this sample in §7.1.5.

7.1.2 Study Procedure. We hypothesized that Lucid would improve a developer's ability to determine how well a property-based test exercises the code under test—and therefore, how likely it is to find bugs. At its core, our study consisted of four "tasks," each presenting the participant with a PBT property plus three sets of sampled inputs for testing that property, drawn from three different distributions. The goal of each task was to rank the distributions, in order of their bug-finding power, with the help of either Lucid or a control interface that mimicked the existing user 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 included Hypothesis's warnings (e.g., when $< 10\%$ of the sample inputs were valid). 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; Lucid was re-labeled "Charts" and the control was labeled "Examples," to reduce demand characteristics.

The distributions that participants had to rank were chosen carefully; one distribution was the best we could come up with, one had a one to two clear flaws, and one was intended to be very low quality. To establish a ground truth, we benchmarked each trio of input distributions using a state-of-the-art tool called Etna [71]. 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 a collection of synthetic bugs to be injected into a particular bug-free program, and Etna does the work of measuring how quickly (on average) a generator is able to trigger a particular bug with a particular property. Prior work has shown that test quality as measured by mutation testing is well correlated with the power of tests to expose real faults [41]. These ground truth measurements agreed with the original intent of the generators, with the best ones finding the most bugs, followed by the flawed ones, followed by the intentionally bad ones.

The study as experienced by the user is summarized in Figure 4. We started by providing participants some general instruction on PBT, since we did not require that participants had worked with it before. After some screening questions to ensure that participants had understood the instruction, we presented the main study tasks. In each, the participants ranked three test distributions based on how likely they thought they were to find bugs. Each of the four tasks was focused on a distinct property and data structure:

- *Red–Black Tree* The property described in §2.1 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 realistic name server [83] behaves the same as a simpler model implementation.

These tasks were designed to be representative common PBT scenarios: red–black trees are a standard case study in the literature [69, 71], topological sort has been called an ideal pedagogical example for PBT [59], programming language implementations are

987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044

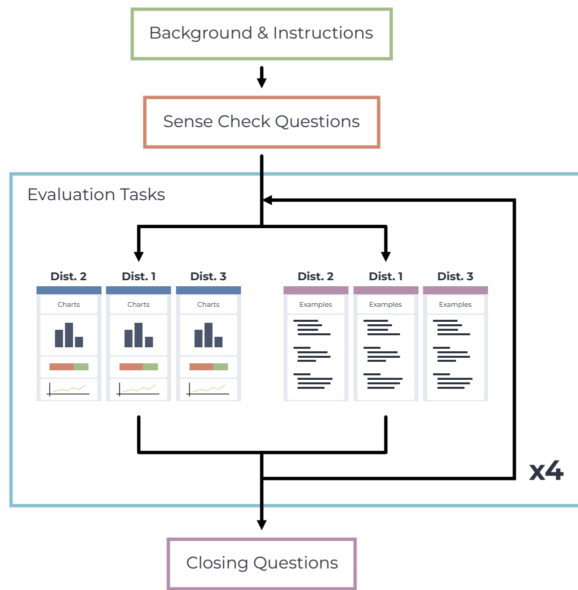


Figure 4: The procedure of the self-guided, online usability study.

a common PBT application domain [63], and name servers capture some of the challenges of PBT on systems with significant internal state [38].

To counterbalance potential biases due to the order that different tasks or conditions were encountered, we randomized the participants' experience in three ways: (1) two tasks were randomly assigned Lucid, while the other two received the control interface, (2) tasks were shown to users in a random order, and (3) the three distributions for each task were shown in a random order.

Four participants took over an hour to complete the study; 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 took less than 15 minutes. Participants took about 3 minutes on average ($\sigma = 2.5$) to complete each task.

7.1.3 Results. To answer **RQ1**, whether or not Lucid helps developers to predict test suite bug-finding power, we analyzed how well participants' rankings of the three distributions for each task agreed with the true rankings as determined by mutation testing. Given a participant's ranking, for example $D_2 > D_1 > D_3$, we compared it to the true ranking (say, $D_1 > D_2 > D_3$) by counting the number of correct pairwise comparisons—here, for example, the participant correctly deduced that $D_1 > D_3$ and $D_2 > D_3$, but they incorrectly concluded that $D_2 > D_1$, so this counted as one *incorrect comparison*.[6]

Figure 5 shows the breakdown of incorrect comparisons made with and without Lucid, separated out by task. To assess whether Lucid impacts correctness, we performed a one-tailed Mann-Whitney U test [54] for each task, with the null hypothesis that Lucid does not lead to fewer incorrect comparisons. The results appear in Table 1. 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, meaning that participants were better at assessing testing effectiveness with Lucid than without. Furthermore, a majority of participants got a completely correct ranking for all 4 of the tasks with Lucid, while this was only the case for 1 of the tasks without it. (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; see §7.1.4.) Despite this difference in accuracy, participants took around the same time with both treatments; the mean time to complete a task with Lucid was 183 seconds ($\sigma = 125$), verses 203 seconds ($\sigma = 165$) for the control. These results support answering **RQ1** with "yes."

Table 1: Values for Mann-Whitney U test measuring Lucid's impact on incorrect comparisons. All sample sizes were between 18 and 22, totaling 40, depending on the random variation in the way conditions were assigned; r is common language effect size, m is median number of incorrect comparisons.

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

To answer **RQ2** we used a post-study survey, asking participants for feedback on which of Lucid's visualizations they found useful. The vast majority of participants (37/40) stated that Lucid's "bar charts" were helpful. (Unfortunately, we phrased this question poorly: we intended for it to refer only to feature charts, but participants may have interpreted it to include the mosaic chart as well.) Additionally, 20/40 participants found the code coverage visualization useful, 17/40 found the warnings useful, and 14/40 found the listed examples useful. Only 4/40 found the timing breakdown useful; we may need to rethink that chart's design, although it may also simply be that the tasks chosen for the study did not require timing data to complete. These results suggest that the customizable parts of the interface—the feature and/or mosaic charts—were the most useful, followed by some of the more general affordances.

To get a sense of participants' overall impression of Lucid, we also asked "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 core observations above, we also made a couple of other smaller observations.

[6]This metric is isomorphic to Spearman's ρ [74] 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 counting incorrect comparisons to be the most intuitive way of conceptualizing the data.

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

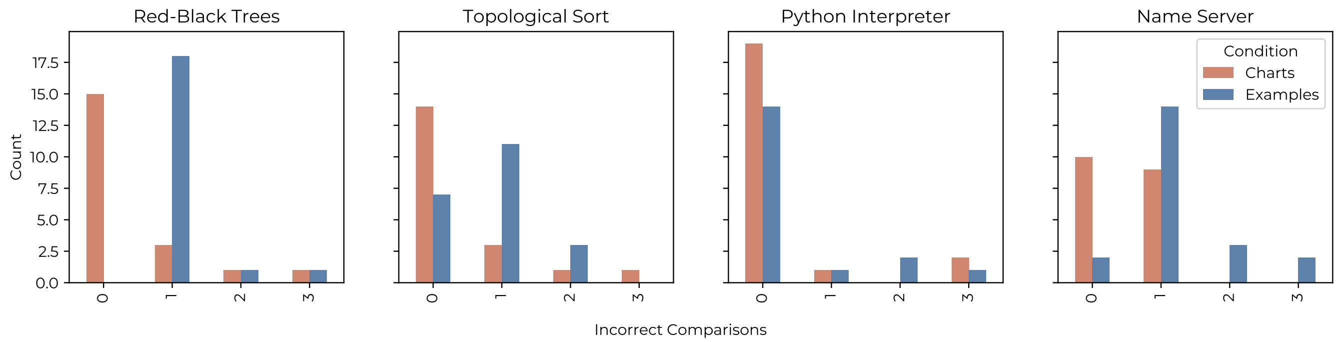


Figure 5: Distribution of errors made for each task, where each incorrect relative ranking between two test suites counts as one error. When using Lucid, significantly fewer errors were made for Red Black Trees, Topological Sort, and Name Server. = Lucid, = Control.

Confidence. Alongside each ranking, we asked developers how confident they were in it, on a scale from 1–5 ("Not at all" = 1, "A little confident" = 2, "Moderately confident" = 3, "Very confident" = 4, "Certain" = 5). We found that reported confidence was significantly higher with Lucid than without on two tasks (Red–Black Tree and Topological Sort), as computed via a similar one-sided Mann-Whitney U test to the one before ($p < 0.01$ and $p = 0.03$ respectively), with no significant difference for the other tasks. However, these results have no clear interpretation. When we computed Spearman's ρ [74] between the confidence scores and incorrect comparison counts, we found no significant relationship; in other words, participants' confidence was not, broadly, a good predictor of their success.

Non-significant Result for "Python Interpreter" Task. As mentioned above, the Python Interpreter task seems to have been too easy; participants made very few mistakes 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 distributions, Hypothesis clearly displayed a warning that "< 10% of examples satisfied assumptions," an obvious sign of something wrong. Conversely, for the best distribution of the three, Hypothesis showed a wide variety of values for the `variable_uses` event, which was only ever 0 for the other two distributions. 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.) This result shows that Hypothesis's existing tools can be quite helpful in some cases: in particular, they seem to be useful when the distributions have big discrepancies that make a visual difference (e.g., adding significant volume) in the statistics output.

7.1.5 Limitations. We are aware of two significant limitations of the online study: sampling bias and ecological validity.

The sample we obtained was appropriate from the perspective of prior experience and general level of education, but it under-represents important groups with regards to both gender and occupation. For gender, prior work has shown that user interfaces often demonstrate a bias for cognitive strategies that correlate with gender [9, 77], so a more gender-diverse sample would have been more informative for the study. For occupation, we reached a significant portion of students and proportionally fewer working developers. Many of those students are in computer science programs and therefore will likely be developers someday, but software developers are ultimately the population we would like to impact so we would like to have more direct confirmation that Lucid works for them.

The other significant limitation is ecological validity. Because this study was not run *in situ*, aspects of the experimental design may have impacted the results. For example, study participants did not write the events and assumptions for the property themselves; if they would have been unable to do so in practice, the outcomes may have been very different. Additionally, participants saw snippets of code, but they were not intimately familiar with, nor could they inspect, the code under test. In a real testing scenario, a developer's understanding of their testing success would depend in part on their understanding of the code under test itself. We did control for other ecological issues: for example, we used live instances of Lucid in an iframe to maintain the interactivity of the visual displays, and we developed tasks that span a range of testing scenarios. We discuss plans to evaluate Lucid *in situ* in §8.1

7.2 Impact on the Testing Ecosystem

Since Lucid is an open-source project that is beginning to engage with the PBT community, we can also evaluate its design by looking at its impact on practice. The biggest sign of this so far is that Lucid has led to 5 concrete bug-fixes and enhancements in the Hypothesis codebase itself. As of this writing, Hypothesis developers have found and fixed three bugs—one causing test input sizes to be artificially limited, another that badly skewed test input distributions, and a third that impacted performance of stateful generation strategies—and two long-standing issues pertaining to user experience: a nine-year-old issue about surfacing important feedback about the `assume` function and a seven-year-old issue asking to clarify terminal error messages. All five of these issues are threats to developers' evaluation of their tests; the problems were found and fixed after study participants and other Lucid users noticed deficiencies in their test suites that turned out to be library issues.

The ongoing development of Lucid has the support of the Hypothesis developers, and it has also begun to take root in other parts of the open-source testing ecosystem. One of the authors was

contacted by the developers of PyCharm, an IDE focused on Python specifically, to ask about the OpenLucid format. They realized that the coverage information therein would provide them a shortcut for code coverage highlighting features that integrate cleanly with Hypothesis and other testing frameworks.

8 CONCLUSIONS AND FUTURE WORK

Lucid property-based testing rethinks the PBT process as more interactive and empowering to developers. Rather than hide the results of running properties, which may lead to confusion and false confidence, the OpenLucid protocol and interfaces like Lucid give developers rich insight into their testing process. Lucid PBT tools provide visual feedback, integrate with developer workflows, provide hooks for customization, show details on demand, and work with other tools in the ecosystem to provide a standardized way to evaluate testing success. Our evaluation of the lucid PBT paradigm shows that it helps developers to tell the difference between good and bad test suites; its demonstrated real-world impact on the Hypothesis framework backs up those conclusions.

Moving forward, we see a number of directions where further research would be valuable.

8.1 Evaluation in Long-Term Deployments

Our formative research and online evaluation study have provided evidence that lucid PBT is usable, but there is more to explore about the design of lucid PBT and Lucid. In particular, as Lucid is deployed over longer periods of time in real-world software development settings, we are excited to assess its continued impact.

8.2 Improving Data Presentation for Lucid PBT

As the Lucid project evolves, we plan to add new visualizations and workflows to support developer exploration and understanding.

Code Coverage Visualization. The visualization we provide for displaying code coverage over time was not considered particularly important by study participants: it may be useful to explore alternative designs.

One path forward is in-situ line-coverage highlighting, like that provided by Tarantula [40]. Indeed, it would be easy to implement Tarantula's algorithm, which highlights lines based on the proportion of passed versus failed tests that hit that line, in Lucid (supported by OpenLucid). In cases where no failing examples are found, each line could simply be highlighted with a brightness 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 how code is exercised for different parts of the input space. To address these questions, we plan to experiment with visualizations that cluster test inputs based on the coverage that they have in common. This would let developers answer questions like "which inputs could be considered redundant in terms of coverage?" and "which inputs cover parts of the space that are rarely reached?"

Mutation Testing. In cases where developers implement mutation testing for their system under test, we propose incorporating that information into Lucid for better interaction support. Recall that in §7.1, we used mutation testing, via the Etna tool, as a ground truth for test suite quality; mutation testing checks that a test suite can find synthetic bugs or "mutants" that are added to the test suite. Etna is powerful, but its output is not interactive: there is no way to explore the charts it generates, nor can you connect the mutation testing results with the other visualizations that Lucid provides. Thus, we hope to add optional visualizations to Lucid, inspired by Etna, that tell developers how well their tests catch mutants.

Longitudinal Comparisons of Testing Effectiveness. Informal conversations with potential industrial users of Lucid suggest that developers want ways to compare visualizations of the same system at different points in time—either short term, to inspect the results of changes—or longer term, to understand how testing effectiveness has evolved over time. These comparisons would make it clear if changes over time have improved test quality, or if there have been significant regressions.

Interestingly, the design of the online evaluation study accidentally foreshadowed a design that may be effective: allowing two instances of Lucid, connected to different instances of the system under test, to run side-by-side so the user can compare them. Since developers were able to successfully compare two distributions side-by-side with Lucid in the study, we expect they will also be able to if presented the same thing in practice. This is simple to implement and provides good value for little conceptual overhead on developers.

8.3 Improving Control in Lucid PBT

Lucid is currently designed to support *existing* developer workflows and provide insights into test suite shortcomings. But participants in the formative research (P1, P4) did speculate about some ways that Lucid could help developers to adjust their random generation strategies after they notice something is wrong.

Direct Manipulation of Distributions. When a developer notices, with the help of Lucid, that their test input distribution is sub-par, they may immediately know what distribution they would prefer to see. In this case, we would like developers to be able to change the distribution via *direct manipulation*—i.e., clicking and dragging the bars of the distribution to the places they should be, automatically updating the input generation strategy accordingly. One potential way to achieve this would be to borrow techniques from the probabilistic programming community, and in particular languages like Dice [36]. Probabilistic programming languages and random data generators are actually quite closely related, and the potential overlap is under-explored. Alternatively, *reflective generators* [25] can 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.

Manipulating Strategy Parameters in Lucid. Occasionally direct manipulation as discussed above will be computationally impossible to implement; in those cases Lucid could still provide tools to help developers easily manipulate the parameters of different generation strategies. For example, if a generation strategy takes a `max_value` as an input, Lucid could render a slider that lets the developer change that value and monitor the way the visualizations change, resembling interactions already appearing in HCI programming tools (e.g., [29, 44]). Of course, running hundreds of tests on every slider update may be slow; to speed it up, we propose incorporating

ideas from the literature of self-adjusting computation [1], which has tools for efficiently re-running computations in response to small changes of their inputs.

8.4 Lucid *Automated* Testing

The ideas behind lucid PBT may also have applications beyond the specific domain of PBT. Other automated testing techniques—for example fuzz testing ("fuzzing")—could also benefit from enhanced understandability. Fuzzing is closely related to PBT[7], and the fuzzing community has some interesting visual approaches to communicating testing success. One of the most popular fuzzing tools, AFL++ [18], includes a sophisticated textual user-interface giving feedback on code coverage and other fuzzing statistics over the course of (sometimes lengthy) "fuzzing campaigns." But current fuzzers suffer from the same usability limitations as current PBT frameworks, hiding information that could help developers evaluate testing effectiveness. We would like to explore adapting Lucid and expanding OpenLucid to work with fuzzers and other automated testing tools, bringing the benefits of the lucid design methodology to an even broader audience.

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] 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

[6] 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

[7] 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.

[8] Brian Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 473–483.

[9] 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.

[10] 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.

[11] 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

[7] 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 times. Instead of testing logical properties, fuzzers generally try to make the program crash.

[12] 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

[13] Create React App [n. d.]. Retrieved March 23, 2024 from https://github.com/facebook/create-react-app

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

[15] 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.

[16] 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.

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

[18] 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

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

[20] 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

[21] 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.

[22] 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

[23] 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.

[24] 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)*.

[25] 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

[26] 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

[27] 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

[28] 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

[29] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 2008. Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 91–100.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–31. https://doi.org/10.1145/3428208 arXiv:2005.09089 [cs].

[37] 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

[38] 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

[39] 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

[40] 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

[41] 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.

[42] 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.

[43] 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.

[44] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. mage: Fluid moves between code and graphical work in computational notebooks. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 140–151.

[45] 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.

[46] 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

[47] 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.

[48] 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.

[49] 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.

[50] 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

[51] 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.

[52] 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.

[53] 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.

[54] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* (2010), 1–1. Publisher: Wiley Online Library.

[55] Microsoft. 2024. Visual Studio Code. https://code.visualstudio.com/

[56] 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.

[57] 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

[58] 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.

[59] 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].

[60] 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.

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

[62] 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.

[63] Micha\l H. Pa\lka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615 event-place: Waikiki, Honolulu, HI, USA.

[64] 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.

[65] pytest-watch [n. d.]. Retrieved March 23, 2024 from https://github.com/joeyespo/pytest-watch

[66] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming* 3, 1 (2018).

[67] 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.

[68] 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.

[69] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. *ACM SIGPLAN Notices* 44, 2 (Sept. 2008), 37–48. https://doi.org/10.1145/1543134.1411292

[70] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan. 2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030 Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[71] 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

[72] 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.

[73] 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.

[74] 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..

[75] 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

[76] Donald Stewart, Koen Claessen, Nick Smallbone, and Simon Marlow. 2024. Test.QuickCheck — hackage.haskell.org. https://hackage.haskell.org/package/QuickCheck-2.14.3/docs/Test-QuickCheck.html#v:label

[77] 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.

[78] 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.

[79] Steven L. Tanimoto. 1990. VIVA: A Visual Language for Image Processing. 1, 2 (1990), 127–139.

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

[81] 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.

[82] 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.

[83] Wikipedia. 2024. Name server — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Name%20server&oldid=1215654110

[84] Wikipedia. 2024. Red–black tree — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Red%E2%80%93black%20tree&oldid=1215636980

[85] 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

[86] J. Wing, D. Jackson, and C. B. Jones. 1996. Formal Methods Light. *Computer* 29, 04 (apr 1996), 20–22. https://doi.org/10.1109/MC.1996.10038

[87] 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

[88] 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.

[89] 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.

[90] 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 TEST EVALUATION AFFORDANCES IN EXISTING FRAMEWORKS

Table 2: Breakdown of existing test evaluation affordances 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

suite1.py::test_evaluate_equiv_to_python:
- during generate phase (182.90 seconds):
  - Typical runtimes: ~ 69-314 ms, of which ~ 68-311 ms in data generation
  - 52 passing examples, 0 failing examples, 948 invalid examples
  - Events:
    - 5.20%, variable_uses: 0
- Stopped because settings.max_examples=100, but < 10% of examples satisfied assumptions

```
test_evaluate_equiv_to_python(
    p=
    <START>

    0
    <END>,
)
```

suite2.py::test_evaluate_equiv_to_python:
- during generate phase (19.88 seconds):
  - Typical runtimes: ~ 20-703 ms, of which ~ 19-694 ms in data generation
  - 100 passing examples, 0 failing examples, 7 invalid examples
  - Events:
    - 93.46%, variable_uses: 0
- Stopped because settings.max_examples=100

```
test_evaluate_equiv_to_python(
    p=
    <START>

    0
    <END>,
)

test_evaluate_equiv_to_python(
```

suite3.py::test_evaluate_equiv_to_python:
- during generate phase (14.89 seconds):
  - Typical runtimes: ~ 35-229 ms, of which ~ 33-225 ms in data generation
  - 100 passing examples, 0 failing examples, 2 invalid examples
  - Events:
    - 21.57%, variable_uses: 2
    - 16.67%, variable_uses: 3
    - 13.73%, variable_uses: 0
    - 10.78%, variable_uses: 1
    - 10.78%, variable_uses: 4
    - 6.86%, variable_uses: 8
    - 5.88%, variable_uses: 10
    - 5.88%, variable_uses: 5
    - 3.92%, variable_uses: 6
    - 0.98%, variable_uses: 7
    - 0.98%, variable_uses: 9
- Stopped because settings.max_examples=100