Property-Based Testing in Practice

Harrison Goldstein
University of Pennsylvania
Philadelphia, PA, USA
hgo@seas.upenn.edu

Joseph W. Cutler
University of Pennsylvania
Philadelphia, PA, USA
jwc@seas.upenn.edu

Daniel Dickstein
Jane Street
New York, NY, USA
ddickstein@janestreet.com

Benjamin C. Pierce
University of Pennsylvania
Philadelphia, PA, USA
bcpierce@seas.upenn.edu

Andrew Head
University of Pennsylvania
Philadelphia, PA, USA
head@seas.upenn.edu

ABSTRACT

Property-based testing (PBT) is a testing methodology where users write executable formal specifications of software components and an automated harness checks these specifications against many automatically generated inputs. From its roots in the QuickCheck library in Haskell, PBT has made significant inroads in mainstream languages and industrial practice at companies such as Amazon, Volvo, and Stripe. As PBT extends its reach, it is important to understand how developers are using it in practice, where they see its strengths and weaknesses, and what innovations are needed to make it more effective.

We address these questions using data from 30 in-depth interviews with experienced users of PBT at Jane Street, a financial technology company making heavy and sophisticated use of PBT. These interviews provide empirical evidence that PBT's main strengths lie in testing complex code and in increasing confidence beyond what is available through conventional testing methodologies, and, moreover, that most uses fall into a relatively small number of high-leverage idioms. Its main weaknesses, on the other hand, lie in the relative complexity of writing properties and random data generators and in the difficulty of evaluating their effectiveness. From these observations, we identify a number of potentially high-impact areas for future exploration, including performance improvements, differential testing, additional high-leverage testing scenarios, better techniques for generating random input data, test-case reduction, and methods for evaluating the effectiveness of tests.

1 INTRODUCTION

Property-based testing (PBT) is a powerful tool for evaluating software correctness. The process of PBT starts with a developer deciding on a formal specification that they want their code to satisfy and encoding that specification as an executable *property*. An automated test harness checks the property against their code using hundreds or thousands of random inputs, produced by a *generator*. If this process discovers a counterexample to the property—an input value that causes it to fail—the developer is notified.

The research literature is full of accounts of PBT successes, e.g., in telecommunications software [2], replicated file [31] and key-value [8] stores, automotive software [3], and other complex systems [30]. PBT libraries are available in most major programming languages, and some now have significant user communities—e.g., Python's Hypothesis framework [37] had an estimated 500K users in 2021 according to a JetBrains survey [32]. Still, there is plenty of room for growth. Half a million Hypothesis users represent only 4% of the total Python user base, whereas the Hypothesis maintainers estimate [15] that the "addressable market" is at least 25%. (For comparison, the most popular testing framework, pytest, has 50% market share.)

To help move PBT toward wider adoption, the research community (ourselves included) needs to better understand the practical strengths and weaknesses of PBT and the places where further technical advances are required. Existing work in the software engineering literature has studied how other bug-finding tools are used in practice (see §6), but PBT offers a unique set of tools and warrants its own investigation. Accordingly, we interviewed PBT users at Jane Street, a financial technology firm that makes significant use of PBT, to learn how they use PBT, where they see its value, and in what ways they think it might be improved. Concretely, we aimed to answer two main questions:

RQ1: What are the characteristics of a successful and mature PBT culture at a software company?

RQ2: Are there opportunities for future work in the PBT space that are motivated by the needs of real developers?

The first question aims both to offer guidance for engineers and managers considering whether PBT might fit well in their organizations and to provide a basis for evaluating and comparing PBT technologies. The second question aims to help shape further research to maximize the impact of PBT.

Our findings contribute a wide range of observations about developers' experiences with PBT, adding nuance to the research community's understanding of PBT's real-world usage. Through our interviews, we gleaned several new insights about the situations in which property-based tests are deployed in practice. We found that developers use PBT mainly for testing components of complex systems, expecting the tests to provide greater confidence than conventional example-based unit tests yet still run quickly as part of their normal test suite. Interestingly, we also found that developers leverage PBT for the secondary benefit of communicating specifications: properties serve as a form of persistent documentation, demonstrating the semantics of the software to readers—a benefit less commonly discussed in the literature. Finally, we found that at Jane Street, PBT is primarily used in "high-leverage" scenarios, where properties are especially easy to identify and test. Beyond deepening our understanding of when and why developers

ICSE 2024, April 2024, Lisbon, Portugal
2024.

reach for PBT, we also found that PBT technology can be improved in several ways to better support developers. In particular, study participants reported struggling to generate distributions of test examples that they were convinced effectively exercised the property, and sometimes viewed the process of designing random data generators as a distraction. They also lamented the lack of visible feedback on the effectiveness of their testing.

From these findings, we extract a list of opportunities for future research, including understanding the nuances of PBT performance requirements, exploring better support for differential testing, and expanding the high-leverage scenarios in which PBT is most effective. We also highlight opportunities around improving languages for random data generators, designing interfaces for test-case reduction ("shrinking"), evaluating testing success, and using developer feedback to improve the testing process.

We begin with background on PBT (§2), then present our study methodology and discuss of potential threats to validity (§3). We present the study's main results (§4) and detail lessons learned in the form of observations (§5.1) and research opportunities (§5.2) before discussing related work (§6) and concluding (§7).

2 BACKGROUND

Properties are executable specifications of programs. For example, suppose a developer is working on a new implementation of binary search trees (BSTs)—tree structures where each internal node is labeled with a data value that is greater than any labels in its left subtree and less than any in its right subtree. They know that all the operations on BSTs (insert, delete, etc.) must preserve this validity condition: given a valid BST, they should always produce a valid BST. To enforce this condition, they might write the following test for the `insert` function using OCaml's Core QuickCheck library [19]:

```
let test_insert_maintains_bst () =
  QuickCheck.test
    (both int_gen (filter valid_bst tree_gen))
    (fun (x, t) -> valid_bst (insert t x))
```

The second argument to `QuickCheck.test` (on the last line) is the property: It says, given an integer `x` and a BST `t`, that `insert t x` should return a BST. The first argument to `QuickCheck.test` is a random data generator, which here generates a pair of `both` an `int` and an arbitrary BST (obtained by generating an arbitrary binary tree using `tree_gen` and using `filter` to discard trees that are not valid BSTs). `QuickCheck.test` randomly generates hundreds or thousands of pairs `(x, t)` and invoke the property to check each pair. If this check ever fails, we have discovered a bug in `insert`.

Generators like `tree_gen` are usually written using an embedded domain specific language (eDSL) provided by the PBT framework, which includes base generators like `int_gen` and combinators like `both` that can be used to create generators for complex data types from generators for their parts. The generator language is embedded in OCaml (in this case), giving generator writers access to the full power of the host language. Generators can require some careful tuning to find bugs effectively. This is often because properties have *preconditions* that define what it means for an input to be valid. For example, "`filter valid_bst tree_gen`" is a correct generator of BSTs: it simply discards trees until `tree_gen` happens to generate a valid BST. However, this means that the *majority* of the generated trees will be discarded, wasting precious generation time. A better strategy is to hand-craft a generator that only produces valid BSTs, but such generators can be nontrivial to write.

If the property ever fails during testing, the failing value is presented to the user. This value might be overly complex, with parts that are irrelevant to the failure of the property, so most PBT frameworks provide tools for test-case reduction [36], usually called *shrinking* [29] in the PBT literature.

The BST example above uses OCaml's QuickCheck library, but the general approach—a module under test, a concise property, a data generator, and a shrinker—is shared by all PBT tools, from the original Haskell QuickCheck [11] to Python's Hypothesis library [37] and beyond. The power of PBT comes from the ability to test a huge number of system inputs with a single specification and generator, often uncovering edge and corner cases that the developer might not have considered. It thus hits a useful midpoint between example-based unit tests and heavier-weight formal methods, retaining the precision of traditional formal specifications and their ability to characterize a system's behavior on all possible inputs, but offering quick, best-effort validation instead of requiring developers to write formal proofs.

Of course, PBT is only one among many testing methodologies. Two primary alternatives are especially relevant to our study.

Example-based unit testing [13] (as supported by pytest, JUnit, and other frameworks) evaluates a program by testing it on individual example inputs. For each input, the developer writes down a short snippet of code that checks that the program's output is correct for this input. (We call this style "example-based unit testing" throughout the paper, rather than just "unit testing," because PBT is also typically used to test individual units within larger systems.)

Fuzz testing, invented by Miller [4, 39] and popularized by tools like AFL [56], also aims to find bugs by randomly generating program inputs. The technical foundations of fuzz testing and PBT overlap to a large (and increasing [34, 53]!) degree, but existing tools from the two communities tend to be tuned for different situations: fuzz testing is typically used in integration testing of complete systems, to detect catastrophic bugs like crash failures and memory unsafety, while PBT is used to flush out logical errors in smaller software modules.

3 METHODOLOGY

To address the research questions described in §1, we conducted an interview study of developers who use PBT in their work. This study was conducted in accordance with the ACM SIGSOFT Empirical Standards for qualitative surveys.

3.1 Population

As the setting for our study, we chose Jane Street, a financial technology firm that uses PBT extensively. We recruited 31 participants and carried out 30 interviews (one was a joint interview). Participants were recruited by a Jane Street developer who volunteered to coordinate the study: they found instances of PBT in the source tree and contacted the authors of those libraries; announced the study on an internal blog; and carried out snowball sampling by asking participants for others we should talk to. All participants had some experience with PBT, and most self-reported as having

Table 1: Participant backgrounds. Columns 3–5 reflect optional questions; participants that didn't respond to these are listed at the bottom. ⋆Measured in years. ⋆⋆Stated comfort with PBT on a Likert scale: 7 most comfortable, 1 least. †Participant indicated skepticism of PBT. ‡Pair interview; coded as one participant. (At Jane Street's request, we do not associate individuals with their teams or company divisions.)

ID	Role	SE Exp.⋆	PBT Exp.⋆	Comfort⋆⋆	
1	Tester	12	3		(6)
3	Maint.	22	17		(7)
4	Maint.	15	16		(6)
5	Tester	5	7		(6)
6	Tester	16	16		(7)
9	Tester	4	4		(5)
11	Maint.	10	7		(6)
14	Tester	8	7		(5)
15	Tester	2	2		(6)
16	Tester	8	8		(3)
18	Tester	1	1		(5)
20†	Tester	5	2		(5)
23	Tester	6	6		(5)
25‡	Testers	26	20		(5)
26	Tester	2	2		(6)
28	Tester	7	1		(5)
29	Tester	4	5		(6)
Other Testers: 2, 7, 8, 10, 12, 13, 17, 19†, 21, 22, 24, 27, 30					

positive experiences. We also explicitly asked for participants who reported neutral or negative feelings about PBT, but they were difficult to find: most felt they did not have enough experience to speak to those feelings. Two self-described PBT-critical developers participated in the study. More details about the study participants can be found in Table 1.

Most of those we interviewed were *testers* (26/30), who use PBT tools in their day-to-day work, but we also spoke to several *maintainers* (4/30) who play a role in building and maintaining Jane Street's PBT infrastructure. These two groups were given different prompts (see below), reflecting our expectation that the maintainers would be able to offer a more "global" perspective, but their responses were coded uniformly since they address the same research questions.

Participants who answered an optional background questionnaire had between 1 and 26 years of professional Software Engineering experience (median 7) and between 1 and 20 years using PBT (median 6). This wide range of experience (for PBT, almost as wide as possible, since the first paper on PBT is only 23 years old! [11]) also means we heard from developers at many different points in their careers and with differing levels of software development experience. When asked to rate their comfort with PBT on a Likert scale, these participants were overall quite comfortable with PBT (median 6 out of 7): all but one were at least "somewhat comfortable" (5 out of 7). Working with developers who are already relatively fluent users of PBT allowed us to benefit from their well-informed ideas about how to make PBT better, as well as their frustrations and challenges.

Jane Street's overall financial operations are supported by a diverse set of software engineering efforts. We spoke to developers working on core data structures, distributed systems, compilers, graphical interfaces, statistical computation, and even custom hardware. Study participants spanned fifteen teams in four main areas: Trading (2/30, across 2 teams), working directly with traders to develop technology specific to individual trading desks; Trading Infrastructure (13/30, across 5 teams), building platforms to support Jane Street's trading; Quantitative Research (5/30, across 2 teams), writing software to enable research on trading algorithms; and Developer Infrastructure (11/30, across 6 teams), designing tools and languages upon which Jane Street's applications are built. Jane Street developers primarily work in OCaml, a programming language with strong typing, good mechanisms for modularity, and a focus on performance. OCaml is thought of as a functional language, but it has strong support for imperative programming as well.

3.2 Protocol

We conducted a semi-structured one-hour interview with each participant; Goldstein and Cutler led the interviews. For testers, the prompts were:

(1) Tell us about a noteworthy time that you applied PBT.
   (a) What kinds of properties did you test?
   (b) How did you generate test inputs?
   (c) How did you evaluate the effectiveness of your testing?
   (d) What did you do to shrink your failing inputs?
(2) Which parts of the PBT process are the most difficult?
(3) What role does PBT play in your development workflow?
(4) To whom would you recommend PBT?
(5) In what contexts is PBT most useful?
(6) Is there anything that would make PBT more useful to you?

The script was designed to attain depth by encouraging reflection on real, memorable experiences with PBT. It evolved somewhat over the course of the study, allowing us to validate interesting or unexpected observations from earlier interviews.

A separate script was used with maintainers:

(1) Have you seen the type of adoption that you want from your PBT tools?
(2) How can QuickCheck be improved?
(3) What do you think it would take to get everyone at Jane Street using PBT? Would that be a good thing?
(4) What do you hope we'll learn from this study?

Time permitting, we also asked maintainers about their use of PBT.

We did not explicitly interview until saturation; we simply tried to recruit a reasonably sized group that was representative of PBT users at the company. However, as we neared the end of the study we felt we were no longer learning about new aspects or perspectives on PBT, and we saw convergence on many of our findings (as evidenced by the numbers we report in §4). Thus, we do not expect there are significant holes in our results.

Once the interviews were complete, they were transcribed using an automated transcription service [42] and analyzed to extract important themes following a thematic analysis process [7]. Goldstein and Cutler carried out an open coding pass (reading through the transcripts and assigning thematic codes as they went). The codes

we chose focused on participant goals, benefits of PBT, challenges encountered, and opportunities for improvement. Once the set of codes stabilized, they were validated by a third co-author, then clarified by the whole team to obtain a final set of clean codes. Finally, one co-author performed an axial coding pass (reading through the transcripts and codes again to make connections and ensure consistency) with the updated codebook. Our codebook is available online[1].

3.3 Threats to Validity

While our study covered a wide variety of software teams and tasks, it should be noted that participants mostly described using a single PBT toolset, in a single language and software development ecosystem. Our findings may under-represent the usage patterns and challenges experienced by those working with other PBT toolsets in other languages. To help us develop findings that generalize beyond the Jane Street toolset, we designed our interview protocol to focus considerable attention on the conceptual and methodological aspects of PBT, rather than on details of OCaml's QuickCheck implementation or specifics of how it is used at Jane Street. (In the cases where we make observations that are toolset-specific, e.g., where we know tools outside of OCaml address some of the problems we observed, we state this explicitly.) In addition, participants were mostly experienced developers who were comfortable with PBT. This means that our study under-reports the experiences of novice developers. Finally, as discussed above, we did not measure saturation as interviews progressed, so there is a chance that more interviews may still have uncovered new insights.

Another threat comes from the inherent limitations of interview studies. For one thing, participants may not always accurately recall the particulars of their experiences with tools. We did our best to mitigate this threat by asking developers to tell us about specific experiences—a standard technique for studies like ours. Additionally, our own biases as interviewers may naturally have skewed the results; Goldstein and Cutler, who carried out the interviews, and Pierce and Head, who also participated with additional questions, are all property-based testing researchers who have a vested interest in the outcome of the study. Outcomes that show PBT in a positive light may have unintentionally been highlighted, and criticisms that we have addressed in our prior work may have received extra attention.

4 RESULTS

We now present our findings. We start by describing the benefits that PBT delivered to developers and how it fit with the other testing approaches they used. Then we describe in detail their experiences writing specifications, designing generators, debugging, and evaluating testing success, focusing both on challenges they experienced and on opportunities for improving PBT tools and workflows.

4.1 Benefits of PBT

As might be expected, the primary reason participants used PBT was to assess whether their code was correct. Participants often used PBT to validate widely used or mission-critical code. For example, one participant described using PBT for code that was "used in 50,000 places across the code base" to establish reliable behavioral contracts for library consumers (P13). Another participant used PBT to check software that interacted with information they were "sending to the [stock] exchange" (P25), which, should it contain errors, could incur serious financial costs. An advantage of PBT in these cases was its ability to explore edge cases (mentioned by 10/30) since, in the words of one participant, "the bugs are always going to happen in cases you didn't think of" (P21).

For many participants, PBT served to increase confidence that their software was robust. They described code tested with PBT as "solid" (P6, P28), and some (9/30) explicitly mentioned that PBT increased their "confidence" that the code was correct. One described using PBT when they "wanted peace of mind" (P11). Confidence was accompanied by material evidence of success: a third of participants (10/30) said their property-based tests found bugs that they had not found via other methods.

A less obvious benefit of PBT was its ability to help participants better understand their work. One participant described properties as tools that "force [the developer] to think clearly" (P30) about their code, and another pointed out that sometimes properties fail because the specification (rather than the code) is wrong (P10). In such cases, failing tests can highlight gaps in the developer's understanding of the problem space that could lead to issues later.

Outside of the testing loop, properties were useful for documentation and communication (12/30). One participant noted that properties "are part of the interface and part of the documentation" (P3) and emphasized that, unlike other documentation, properties never fall out of date: whereas a textual description in a comment might drift from the code's actual behavior, a property is repeatedly re-checked as the code changes. Many also talked about the value of PBT in the code review process (18/30) as a compact way to express what a particular program does. One remarked: "I feel like [PBT] is very nice for review…I really dread seeing 10,000 lines of tests where you just need to spend hours to read code and understand what's going on…And I think, if there is a QuickCheck test and you look at the property that's been tested, which is usually much shorter…it gives you much more confidence that the code is correct." (P19)

The benefits of PBT were such that even those who were critical of it found it to be beneficial in some situations. During the recruiting process we tried to explicitly recruit participants who said they *did not* like PBT; we found two (P19 and P20), but the criticism in their interviews turned out to be constructive suggestions that were echoed by PBT's proponents as well, and both primarily spoke about situations where PBT was valuable.

Moreover, some participants who liked PBT *really* liked it: one said that they "use QuickCheck for everything" (P13) and another asserted that "everyone should be aware of it" (P27). Yet another participant argued that "[PBT] is so useful that it's worth trying to reorganize your code into the form that it is applicable. PBT is so helpful that if you can reinvent things that way, you should." (P10) Several participants (8/30) also talked about evangelizing PBT, encouraging others to use it and increasing its adoption across the organization.

We found that this sort of evangelism benefited all parties: PBT becomes more useful as more people on a team or in an organization use it. A culture of supporting PBT can save work: P24 and P27

[1] https://doi.org/10.5281/zenodo.10407686

both noted that PBT would be easier to use if more developers in the organization provided generators for the types exported by their modules, making it much easier to test code in other modules that uses those types. In addition, treating property-based tests as documentation, as many (12/30) did, requires that others in the organization can read and understand such documentation.

4.2 Comparisons to Other Testing Approaches

The most common approach to testing at Jane Street is not PBT, but rather an example-based unit testing framework called "expect tests" [40]. Expect tests are basically conventional example-based unit tests with editor integration that helps developers create unit tests from the outputs that the system produces. With expect tests, developers add code to the system under test that that logs interesting data to the standard output channel; the expect testing framework then checks that the output matches the output string that was captured from the previous run of the system—it is up to the developer to decide which output is intended. Expect tests are integrated into Jane Street's editor workflow, and they are used pervasively in much the same way as frameworks like pytest and JUnit are used in other languages. Thus, we can use comparisons with expect tests as a proxy for comparisons with example-based tests in general where the specifics of expect tests are not relevant.

From a legibility perspective, PBT was sometimes seen as better, sometimes worse than expect tests. Expect tests were described as more transparent and "often easier to understand" (P5). P18 said "expect tests…explain what they're doing to a better degree. And it's easier for someone to come in and review my test," and P27 made a similar point. But expect tests can sometimes go past from "transparent" to just verbose. P17 complained that reading a whole output string was onerous, and another participant complained "[with expect tests] it's easy to get into a mode where you just like write a test, and you just print out a bunch of crap. And then it's really annoying to like code review that test because there's just too much stuff" (P7). Properties are more concise.

PBT and expect tests also present different strengths and weaknesses when it comes to test writing. One participant said that writing an individual expect test is "way easier than writing invariants," but they highlighted that at the scale of a whole test suite they "love not writing specific unit tests cases" (P13).

Ultimately there was no universal preference for properties or expect tests—both should be available in a developer's toolkit. We might infer from participant responses that when code is simple enough and examples communicate its behavior well enough, expect tests are an attractively lightweight option. In cases where the code is particularly difficult to get right or where writing out enough examples becomes tedious, it seems PBT may be a better choice.

One area where properties seem to be at a clear advantage over expect tests is in the confidence they provide developers. P25 remarked of Jane Street developers that "[they] go to randomized testing when [they're] not so confident of what [they've] done." As discussed above, around a third of participants talked directly about PBT building confidence, and the same proportion reported PBT finding bugs that had not been found with other methods.

Expect tests and PBT were both used to test individual software units during development. PBT was almost always described as running in Jane Street's build system, and many developers actually test their properties "locally after each edit" (P2) Participants described strict time budgets for PBT—no more than "30 seconds" (P27) and as low as "50 milliseconds" (P11)—to ensure it would not slow down the build. These time budgets serve to keep PBT from triggering the 1-minute-per-library timeout that many Jane Street developers set for their unit test suite (P7).

Alongside these fast-turnaround unit testing tools, developers also used fuzzing tools like AFL [56] and AFL++ [20] for integration testing. A third of participants (10/30) mentioned fuzzing, and a few (3/30) described using AFL alongside PBT as part of their testing process. In contrast to the strict timeouts set for expect and property-based tests, one participant described a fuzzer that was forgotten and left running for a year and a half on a spare server (P9). Others did not run fuzzing this long, but still considered it to be running "out of band" (P28), outside of the normal continuous integration and at time scales on the order of hours or days. The upshot is that fuzzing tools are given *much* longer to run than PBT tools.

4.3 Writing Specifications

The previous sections have dealt with PBT at a high level; we now begin a deeper dive into the specifics of the PBT process and how developers described actually using PBT tools. The first step in this process is defining one or more properties to test. While participants did describe struggling with this stage, many ultimately developed powerful strategies for finding properties.

Many participants (16/30) said the process of writing specifications slowed their progress. P4 summed it up like this: "I think the most common failure mode is actually not knowing what properties to test." Challenges ranged from systems that seemed not to have properties at all—P1 talked about "a server that serves queries…and has some…nuanced behavior" that is not compatible with "logical properties"—to systems whose properties were hard to articulate in the form of QuickCheck tests: "With [example-based unit tests], I kind of look at it, I can just make a snap judgment as to whether this is okay or not. Trying to formalize that judgment sometimes can be very difficult." (P2)

Challenges in articulating properties can come from the way code is written. P7 explained that mutable state (e.g., a hidden variable that may change between calls to the same function) gets in the way of writing properties: "there's…this hidden state component…that kind of makes it harder for me to think about what are the right laws." P22 also pointed out that "integrating the outside world and…dealing with the interaction of very large and complicated systems" makes it less clear how to "meaningfully test with PBT." (These findings are not surprising. It is well known that code that interacts with its environment is also a challenge for testing in general, not just PBT: *stateful code* uses mutable data structures such as hash tables, which might introduce differences between runs of the same test if the state is not reset properly; and *effectful code* might rely on external files, databases, or networks, which may not be safe to access over and over during testing.)

Despite these difficulties, the developers we spoke to were generally enthusiastic about PBT. One might therefore wonder: Did they

simply forge ahead regardless, or did they have specific techniques for circumventing the difficulties of writing specifications?

What we found was that developers often succeeded in applying PBT by being opportunistic in their choice of *when* to apply it. Rather than try to apply PBT to every program at all times, participants looked for situations where it offered *high leverage*: more confidence for less work. Some participants described this in general terms as "go[ing] for the low hanging fruit" (P2) or finding places where "the properties are sort of obvious" (P28), but many enumerated specific situations where they would reach for PBT.

Classical Properties (11/30). Certain forms of properties are familiar from the PBT literature and from library documentation for PBT frameworks. These include mathematical properties (e.g., the commutativity of addition), properties drawn from CS theory (e.g., repeated sorting has no effect), and properties that naturally fall out of a data structure's invariants (e.g., the ordering condition of a BST). These properties may be more accessible because they are naturally top-of-mind.

Round-Trip Properties (11/30) are also common in the literature; we heard about them so much more often than the other classical cases that they seem worth calling out on their own. These properties check that a pair of functions are inverses of one another—for example, parsing and pretty-printing or encoding and decoding functions. This situation is easy to notice and easy to test since the properties are incredibly succinct (you call one function, then its inverse, and then check that the final result matches the original input), so round-trip properties are a popular choice.

Catastrophic Failure Properties (7/30). Rather than write logical specifications, some participants used PBT to try to provoke catastrophic failures such as assertion failures and uncaught exceptions. In general, these kinds of properties provide less confidence in code quality (there can still be logical errors, even if the code does not crash), but they help to rule out worst-case scenarios and they are easy to write. (These kinds of specifications are identical to the ones often used for fuzzing; the line between the techniques is blurry, but since the participants were using PBT tools—including complex generators—and testing small units of software, we still consider this relevant for our purposes.)

Differential Properties (17/30). A "differential property" (in sense of differential testing [25]) compares the system under test to a reference implementation of the same functionality that serves as a specification; these are often also called model-based properties (c.f. model-based testing [54]), especially when the reference implementation is designed to be an abstract model of the original code. Differential properties were by far the most widely implemented kind of property. Differential properties were described as a "natural place to use property based testing" (P3)—indeed, one participant remarked that PBT was challenging in a particular situation in part because a good reference implementation was not available (P1).

The common theme of the high-leverage scenarios described above is the availability of a succinct abstraction that can be used in writing a property. These PBT scenarios were summed up by P9 with the following mantra: "[PBT is] most useful when…you have a really good abstraction with a complicated implementation." We discuss how these high-leverage scenarios should be taken into account to accelerate research in PBT in §5.

4.4 Generating Test Data

Participants had several goals for generating inputs. First and foremost, many participants (17/30) talked about needing to generate values that satisfy some precondition. This can be extremely important for effective testing: if too few of the generated values satisfy the property's precondition, then testing will fail to exercise the code in interesting ways; in the worst case, it may even report false positives. A few of the preconditions mentioned by participants are easy to satisfy randomly—for example non-empty strings or lists—but most are quite hard to satisfy: valid postal addresses, well-structured XML documents, red-black trees, and syntactically valid S-expressions are extremely unlikely to be generated by a naïve random generator. Participants also described test data requirements going beyond precondition validity. Some (5/30) said they wanted their test data to be "realistic"—i.e., similar to the distributions of data the application was likely to see in the real world. Others described heuristics for the distribution of the input data, for example generating lists or trees of a "reasonable size" (P9).

Participants described a few different ways that they generated inputs. Most talked about either handwritten random generators (19/30), which are the default approach in libraries like QuickCheck, or *derived* generators (19/30), which are inferred based on the types of the data to be generated (e.g., the type `int list` implies a generic generator for lists of integers).

The need for handwritten generators seemed to be a source of friction for many participants. P6 said that writing generators for data that satisfies property preconditions is "the biggest annoyance with trying to use QuickCheck" and many participants thought of writing generators as "tedious" (6/30) or "high-effort" (7/30). P2 described the task of writing generators as intruding into their development process and contributing to a general perception of PBT as "high cost and low value." Since PBT was usually described as an integral part of the development process, rather than as a separate quality-assurance task, it makes sense that requiring detours into a lengthy generator design process might make PBT less desirable.

Besides requiring significant effort to use, available tools for writing generators by hand do not easily enable developers to produce precondition-satisfying values that are also well distributed in the way they would like. Writing a precondition-satisfying generator on its own can be a large task—whole papers have been written about generators for a single complex data type [43]—and accounting for distributional considerations adds even more friction. P13 said, "there's a tension in…all these handwritten generators between 'I want kind of coverage of everything' and 'I want coverage that is realistic for most inputs.'" As a solution, P13 suggested that one might be able to carefully combine two different handwritten generators to achieve these competing goals, but thought "that way lies madness". Other participants had an idea of the kinds of values they wanted to generate more or less frequently, but they were not sure how to get there: "What should [the probabilities in my generator] look like?…I'm sure if I studied probability and statistics and fully understood how the QuickCheck generating system worked, I could give better guesses. But all my guesses…they're not educated guesses. They're just random…And that's a little bit of a mental strain." (P20)

In Jane Street's ecosystem, derived generators are enabled via the `ppx_quickcheck` library, which provides a preprocessor that runs before the compiler and synthesizes generator code from type information. Ideally, this approach is totally automatic, providing a generator without any additional user input, and it was described fondly by participants. Many included it in their workflows (19/30), and one called it "f***ing amazing" (P5). P13 went so far as to suggest that "you shouldn't really be handwriting generators, you should be changing the structure of your type [to improve generation]." (For example, if a function being tested takes an `int` flag but the only valid values are `0`, `1`, and `2`, then replacing `int` with an appropriate enumerated type causes `ppx_quickcheck` to derive a better generator.) Others (7/30) leveraged a combination of derived and handwritten generators, using derived generators as a foundation on which to build more complex generator programs.

For both handwritten and derived generators, Jane Street developers recognized opportunities for tool improvements. P4 described the process of hand-writing a generator for a mutable data structure as "overwhelming," and P6 suggested that libraries could do a better job supporting that use case. As for improving PPX-derived generators, P26 wanted better support for generating values of *generalized algebraic data types* (GADTs), special types in OCaml and some other languages that can express fine-grained properties of data.

4.5 Understanding Failure

Debugging is often tricky, but participants described ways that tracking down bugs detected by PBT can be especially so. One participant described their debugging process: after finding a large, unwieldy counter-example with PBT, "you have to pull that failure into a separate sort of regression test…which runs the same infrastructure but does it with much more detail…Having done that, it is still sometimes unclear—like what about this example actually causes us to fail?" (P1) This is an instance of a broader problem: random test generators often produce failing examples that are too large to make sense of during debugging. To help developers understand failing examples, PBT frameworks offer shrinkers that transform large inputs into smaller inputs that (presumably) introduce the same bug. One participant in our study deemed shrinking "necessary" (P15), and two who implemented their own ad-hoc PBT frameworks (P8 and P21) insisted that shrinking was one of the most important features they implemented.

That said, participants found it difficult to use the shrinking functionality in QuickCheck, which resembles shrinking in many PBT frameworks. In such frameworks, shrinking is achieved by having users manually write functions that incrementally reduce a large value (e.g., a string, list, or other more complex data structure) to a smaller one that triggers the same failure. Participants saw writing shrinkers as an undesirable activity: one plainly stated, "I hate writing shrinkers" (P4). One challenge of writing shrinkers was writing them in a way that preserved important non-trivial invariants: "It's easy to write a shrinker that accidentally does not preserve some invariant, and that makes your test fail." (P13) This led one participant to ask for a "generic solution [for shrinking], rather than having the user write shrinkers" (P10).

A few participants also described wanting more information from the shrinking process—for example, the progressively smaller intermediate values that were found during shrinking. P1 said "the shrinker gets it right…but [it's] still useful to see the evolution."

4.6 Understanding Success

Property-based testing not only requires a developer to understand test failures; it also requires developers to understand whether *passing* a test actually means the software is correct. If the inputs provided by the test harness fail to adequately exercise buggy code, a property-based test may pass misleadingly. One participant described this situation, recalling a bug in published code that their property-based tests failed to catch because the generator was "not generating the case that [they] had in mind" (P19). P14 worried they could not trust their generators because they had "no idea what it's generating." (P14)

But, while participants understood that, in principle, tests could pass erroneously, 11 of them—more than a third—said they did not think as hard as they should about testing effectiveness, or whether they had adequately tested their software with their generators. Some (3 of the 11) reported seeing their property catch a couple of bugs and deciding that they did not have to improve their properties any further; the rest trusted that the generators provided by OCaml's QuickCheck library or the ones they derived or wrote themselves would be good enough. While this group is a minority of our sample, even a signal of this size is striking: PBT tools such as derived generators should make it easy for developers to get started; they should not (but evidently sometimes do) discourage developers from being critical of their test suite.

On the other hand, several participants described techniques for validating their PBT tests:

Mutation Testing (7/30). Some participants intentionally added bugs to their code and checked that their tests successfully found those bugs. This technique is standard in the testing literature [44, 47] and used in testing benchmarks such as Magma [27] and Etna [50].

Example Inspection (8/30). An even simpler way to assess a generator's distribution is to look at a handful of examples that the generator produces; one participant (P26) even designed a small utility to graphically render one example at a time, to make them easier to understand at a glance.

Code Coverage (2/30). Participants evaluated the code coverage achieved by their tests and used code coverage measurements as an indication that their properties were thoroughly exploring the space of program behaviors.

Property Coverage (1/30). Another participant measured coverage not of the system under test, but of the property itself. This is a weaker measurement, since it does not say anything about the system under test, but it is much easier to make because it can be measured without complex tooling.

A couple of participants (2/30) compensated for gaps in their property-based tests by supplementing them with example-based unit tests. In these cases, example-based tests were written to test complementary functionality that could not be easily tested with properties. As one participant put it, "it's kind of not a good idea to use QuickCheck in isolation" (P6), as doing so limits the breadth of software behaviors that can be tested.

How might PBT frameworks provide better support these (and other) techniques that give insight into how thoroughly properties

have been tested? P15 described their ideal interaction with PBT tools, where the tools give "insight into what [PBT] is doing…I think a combination of knowing what it's doing and having more control of the space that it's exploring would be interesting." Participants desired greater awareness of what their tools were doing: P16 called this "visibility", and P30 called it "inspectability." Many participants wanted greater visibility to better understand the breadth of inputs generated. P18, for instance, desired "visualization of the test [inputs] being generated". Others wished to see details such as "the answer [of the function under test] on a smaller set of examples" (P9), "statistics on…edge cases" (P18), statistics describing generated inputs such as "the average length of [a generated] list" (P25), and code coverage (P9).

4.7 Tradeoffs

As the earlier sections indicate, using PBT is not without costs. From coming up with properties to evaluating testing success, participants found friction in many steps of PBT. Participants described seeing themselves as employing PBT more often if its "overhead" could be lowered (P26), or if there was less "effort necessary to integrate [it] into [their] workflow" (P10).

Amidst these costs, PBT was often seen as worth the effort. For some participants, development of properties was unremarkable. For others with more involved implementation, it was still worth the costs: in the words of P6,"[It was] a huge slog, but I was like 'this needs to be right,' and…I'm glad I did it." P17 describes the tradeoffs of using PBT as follows: "doing PBT is both putting in more effort and saving oneself effort as well."

5 LESSONS LEARNED

In this section we answer our research questions, setting a course for upcoming PBT research and tool development that is grounded in findings from the study. Our recommendations cut across the PBT process, and establish new goals and priorities for different areas of PBT research. We first answer **RQ1** with observations of PBT's practice that offer a reality check to PBT researchers and toolbuilders (§5.1), then we answer **RQ2** with a technical and empirical research agenda motivated by our study (§5.2).

5.1 The State of Practice

Recall that **RQ1** asks, "What are the characteristics of a successful and mature PBT culture at a software company?" Based on the results of our study, we answer: A mature PBT culture considers PBT a tool to be opportunistically applied in situations of high leverage to gain confidence in software and document its behavior; developers try to keep PBT "out of their way," expecting it to work quickly and easily. In this section, we synthesize the results from the previous section into observations that expand on these ideas. Some observations point forward to §5.2, where we present ideas for future research based on our findings.

OB1: *Property-based testing is being used successfully to build confidence in complex systems.* The broader research community sometimes situates PBT as a lower-effort, lower-reward alternative to formal proofs of correctness, but that perspective underestimates PBT as a practical tool for improving software quality. In §4.1, we show that the developers at Jane Street found PBT to be a valuable part of their testing toolbox, using it to gain confidence when they were unsure of code's correctness, especially where correctness was of critical importance. PBT gave them confidence that their software was operating correctly, finding bugs in code that had not been found via other methods.

OB2: *Properties are used as devices for communicating specifications.* Tests are generally a valuable form of documentation, and properties are no exception. Jane Street developers use them as executable evidence that code satisfies a specification, which has significant advantages over traditional documentation that might go stale as the code changes. As shown in §4.1 and §4.2, properties were deemed especially useful for communication during code review: when properties were available, they were considered a strong signal that the code was correct, and when unavailable reviewers sometimes asked the submitter to write some. These auxiliary uses for properties are worth keeping in mind for PBT framework designers; for example, some property languages try to make properties easier to

Observations

OB1	Property-based testing is being used successfully to build confidence in complex systems.
OB2	Properties are used as devices for communicating specifications.
OB3	Property-based tests need to be *fast*; they are expected to perform as well as other unit tests.
OB4	Property-based testing is used opportunistically in high-leverage scenarios where properties are readily available; developers rarely go out of their way to write subtle specifications.
OB5	Developers see writing generators as a distraction, preferring to use derived generators.
OB6	Developers may not interrogate properties that do not find bugs, even in cases where they acknowledge they should.

Research Opportunities

RO1	Understand time constraints for property-based tests.
RO2	Improve support for differential and model-based testing.
RO3	Make more testing scenarios high leverage.
RO4	Streamline the process of writing well-distributed, precondition-satisfying generators.
RO5	Improve interfaces for shrinking.
RO6	Improve tools for evaluating testing effectiveness.
RO7	Connect evaluation of testing effectiveness and generator improvement.

Figure 1: Summary of major outcomes.

write by providing terse syntax and expressive defaults, but those features may cause problems if they hurt readability.

OB3: *Property-based tests need to be* fast*; they are expected to perform as well as other unit tests.* The PBT literature is not always clear about exactly when in the software engineering process they expect properties to be written, but in §4.2 we observe that PBT's niche is testing module-level code during development. This is in contrast with fuzz testing, which, when used by Jane Street developers, is treated as a separate step and run outside of the standard testing workflow. This discrepancy makes sense in view of the differing goals that we observed for PBT and fuzzing: PBT is used for module-level tests with often-complex logical specifications, while fuzzing is used for integration-style tests of whole applications, to check for relatively basic (e.g., assertion failure or uncaught exception) errors. (Look ahead to RO1 in §5.2.)

Properties live alongside other unit tests, so they are expected to run as quickly as other unit tests. Knowing this may change priorities for some PBT researchers. If users are only testing their properties for 50 milliseconds, research advances that increase input generation or property execution speed might make the difference between bugs being found or not. Conversely, approaches that make generators more thorough but significantly slower may not be used unless developers are given reason to accept longer execution times.

OB4: *Property-based testing is used opportunistically in high-leverage scenarios where properties are readily available; developers rarely go out of their way to write subtle specifications.* Conventional wisdom in the PBT community sometimes assumes that developers decide to use PBT and then try to think of a specification, but the study participants generally did the opposite: they saw an obvious testable property and then decided to use PBT. We call situations with these readily available properties "high-leverage" testing scenarios.

The high-leverage scenarios varied—we are not confident that we have documented all of the ones available in practice—but a few are described in §4.3. The most popular was differential or model-based testing, which compares the code under test to some other available implementation. (We are not surprised our participants found these techniques useful, after all both differential and model-based testing have rich literatures on their own, but we did not expect to see them so seamlessly incorporated into PBT workflows.) Other high-leverage properties include round-trip properties that check an inverse relationship between functions and catastrophic failure properties that make sure a program does not fail completely.

What does this opportunistic use of PBT mean for researchers and tool builders? Frameworks can optimize for and automate these scenarios—tools like Hypothesis Ghostwriter [16] have already begun incorporating common PBT scenarios into an automation framework—improving the common case and accelerating PBT use even further. High-leverage scenarios should also be incorporated into PBT benchmarks like Etna [50] to make sure that the performance of PBT algorithms is evaluated in a way that reflects their use in real-world scenarios. (Look ahead to RO2 and RO3 in §5.2.)

OB5: *Developers see writing generators as a distraction, preferring to use derived generators.* Since PBT is often done in the midst of development, developers are reluctant to slow down and write a generator; the task was seen as both difficult and time-consuming. Instead, as seen in §4.4, many participants opted to test with generators derived from the types of the data that their programs operate on.

This has two implications. First, it means that ongoing work towards improving generator automation is critical. After all, as easy as current derived generators are to use, they are not always sufficient. The well-liked `ppx_quickcheck` library, for example, cannot derive generators for properties with complex preconditions. Broadening applicability of derived generators lowers barriers to using PBT. Second, it means that developers of manually written generator languages must carefully consider any added friction. Languages that make generators more difficult to write, perhaps because doing so enables desirable new features, should be cautious—these features may be unlikely to see adoption.

Separately, languages and tools for writing generators should provide a wealth of sensible defaults, and they should optimize the user experience around common patterns. Participants indicated that there was room for improvement in all of these areas, at least in OCaml's QuickCheck. (Look ahead to RO4 in §5.2.)

OB6: *Developers may not interrogate properties that do not find bugs, even in cases where they acknowledge they should.* A passing property sometimes means that a program is free of bugs, but it may also mean that the input values are not the right ones to *trigger* a latent bug. Developers acknowledged this fact, and they had expectations around the kinds of input values that they wanted when testing their properties. (Besides satisfying property preconditions, they wanted them to be realistic, well-distributed in the space, interesting enough to cover corner and edge cases, and more.) But, as shown in §4.6, when it came time to decide if their generators met expectations, developers did not analyze them closely. Developers of PBT frameworks, and especially PBT automation, should keep this in mind: automated tools for generating test inputs risk giving developers a false sense of security. They must ensure that their tools test thoroughly, because testers may not. (Look ahead to RO6 in §5.2.)

5.2 Research Opportunities

Now we tackle **RQ2**: "What opportunities exist for future work in the PBT space, motivated by the needs of real developers?" Based on our results, we conclude: Exciting avenues for PBT research include further studies into performance and usability of PBT generators, broader and deeper support of high-leverage PBT scenarios, better tools for shrinking, and better visibility into the testing process. This section discusses research opportunities in software engineering, programming languages, and human-computer interaction research that will unlock the yet-unrealized potential of PBT. Some research opportunities point backward to §5.1, where we synthesize observations that inform these ideas.

RO1: *Understand time constraints for property-based tests.* Future studies should more thoroughly explore how long developers across the software industry actually budget for running PBT. In this study, we heard about numbers between 50 milliseconds and 30 seconds; that difference is massive, and tools that support PBT cannot make optimal decisions around optimization without clearer data. Furthermore, future research should develop a sense of how long is "enough" for common kinds of properties and software so

tools have the data to argue for larger time budgets where required. Moreover, developers of tools that combine ideas from PBT and fuzzing need to carefully examine their performance and recognize that many PBT users set strict timeouts that may preclude the overhead of measuring code coverage while running tests. This is encouraging for tools like Crowbar [17] and HypoFuzz [26] that allow developers to transition between short-running PBT and longer-running fuzzing with the same properties. (See OB3 in §5.1.)

RO2: *Improve support for differential and model-based testing.* As the most popular high-leverage scenario for PBT, differential testing seems particularly interesting as a topic of further research. Differential testing has a rich literature, as discussed in §6, but in the context of PBT there are still advances within reach. For example, in languages like OCaml with rich module structures, researchers should aim to increase automation around differential testing and produce a test harness for comparing modules without requiring any manual setup; Hypothesis Ghostwriter has begun to incorporate similar ideas with Python's classes. (See OB4 in §5.1.)

RO3: *Make more testing scenarios high leverage.* Some testing situations are just barely outside the realm of "high-leverage" scenarios, and improved tooling could make the difference. For example, P5 described a technique that they used to test a poorly abstracted module with an overly complicated interface: instead of writing normal properties, they instrumented the module with log statements, wrote properties about what those logs should look like, and then tested the module via an external interface that hides many of its internal details. Generalizing and operationalizing this technique—e.g., perhaps with temporal logic in the style of Quickstrom [41]—would allow for better testing leverage in cases where poor abstraction prevents traditional PBT.

Other testing situations might be supported better as well. Potential opportunities include improving PBT support for code with mutable state (e.g., by saving memory snapshots for repeatable tests) and code that interacts with the environment (e.g., via robust integration with tools for *mocking* [38]). Increasing the scenarios in which PBT works out of the box will naturally make it higher-value for developers. (See OB4 in §5.1.)

RO4: *Streamline the process of writing well-distributed, precondition-satisfying generators.* This has been a research goal for the PBT community since the beginning. Our study clarifies some promising paths forward, including improving tools for automated tuning, generalizing unified languages for defining generators alongside properties, and supporting alternatives to randomness as first-class.

There are two main options for tuning generator distributions. Actively tuned generators modify their distribution live, in response to feedback, whereas pre-tuned generators compute distributions ahead of time. Actively tuned generators are the norm in the fuzzing literature [20] and have been ported to PBT in many forms [17, 23, 34, 48], but they spend precious testing time on tuning analysis, making them less useful in time-constrained PBT scenarios. Pre-tuned generators can be much faster, but they currently require too much programmer effort. For example, reflective generators [22] (which build on example-based tuning *à la* the *Inputs from Hell* approach [51]) automate the process of generating realistic test inputs, but this automation is only possible if a reflective generator is already available. Moving forward, the community should continue to look for ways to use current active-tuning strategies for pre-tuning (e.g., by saving inputs to be run later) and ways to make pre-tuned generators easier to use (e.g., with interfaces that help developers write complex generators).

When it comes to precondition-satisfying generators, many seek to unify languages for writing generators with ones for writing preconditions. One approach, used in the QuickChick PBT framework [46], uses inductive relations as a language for both properties and generators [35, 45]. This approach works well in the Coq proof assistant, but complex inductive relations are not expressible in mainstream languages or accessible to non-specialists. Researchers should consider ways to generalize these results. Alternatively, dedicated languages such as ISLa [53] use a common language that is closer to the logical connectives one might use in a standard programming language; more research should be done to evaluate if this approach can be applied in common PBT scenarios. Whatever dual-purpose language is chosen, this is a compelling path forward. (See OB5 in §5.1.)

RO5: *Improve interfaces for shrinking.* It is clear from the study results that shrinkers would be far more useful if they were both more automated and more informative. As we mention above, we recommend that existing frameworks improve automation by incorporating internal test-case reduction [36, 52], which uses generators to aid in shrinking, where possible. Internal shrinking has the added benefit of always producing valid values, which is difficult to achieve otherwise.

Participants also asked to see more intermediate examples from the shrinking process. Indeed, there are cases where the smallest failing example is *not* the most helpful one: e.g., if the shrinker outputs the tuple `(0, 0)`, one might conclude that any tuple of integers triggers the bug, but it may be that the bug is only found if the first component is actually `0`. This example motivated Hypothesis to begin developing a tool that will give users a variety of tools for exploring failing tests. Debugging cases using shrinking might mean showing many shrunk examples to the user, following related work in the model-finding literature [14, 18], or even providing control over the shrinking process (e.g., with novel interactions allowing the user to click on components of a value to shrink only those components).

RO6: *Improve tools for evaluating testing effectiveness.* Participants reported ad-hoc, piecemeal approaches to understanding their testing effectiveness, and they asked for better ways to visualize testing feedback. As a simple first step, tools should always announce counts of discarded test cases (i.e., ones that failed the property's precondition) so the developer can catch problems early. Many PBT tools provide some way to aggregate statistics while a property is running (see Haskell QuickCheck's `label` and `collect` functions) but OCaml's QuickCheck hides output when tests succeed, which obscures that information, and more should be done around the usability and legibility of these kinds of aggregations. Going further, participants desired (1) better interfaces for scanning through examples of generated values, (2) better ways of visualizing generated distributions, and (3) better integration of code and branch coverage information. These could significantly improve developers' understanding of how thoroughly their code has been tested. (See OB6 in §5.1.)

RO7: *Connect evaluation of testing effectiveness and generator improvement.* How might a developer clearly express their test data goals *and* ensure that the generator achieves them? Future tools could tighten the feedback loop wherein a developer evaluates and then expresses how to improve their generator at the same time. For instance, a future user interface might display a generator's distribution in some graphical form like a bar chart and allow the developer to directly manipulate the distribution by dragging bars up and down. Alternatively, the developer might be able to click on those annotations to request that the distribution try harder to cover a particular line or balance a particular branch, in the style of AFLGo [10]. In an ideal world, an insufficient generator could be caught and fixed in a few clicks, without allowing bugs slip by.

6 RELATED WORK

We focus, in this section, on related work exploring the usability of testing and formal methods tools. Prior work on PBT, testing, and formal methods usability has made important observations about the challenges of specification and bug finding, but it has lacked the depth and domain focus to paint a clear picture of PBT, its usage, and opportunities for improvement.

Property-based testing. As a precursor to this study, our group did a smaller-scale pilot study with developers using Hypothesis [21]. The full-scale study is far more in-depth, and presents more detailed and nuanced findings, although talking to Python developers did raise a few concerns that were less prevalent at Jane Street, especially when it comes to difficulty coming up with specifications.

A study analyzing open-source libraries using Hypothesis [12] evaluated the kinds of properties that developers test in practice, and found significant overlap with our "high-leverage" testing scenarios. In particular, they found that both round-trip and differential or model-based testing are overrepresented in real-world tests.

An experience report from Amazon [9] described *differential testing* as a major use-case of PBT and reported that developers used shrinking tools for debugging and *mutation testing* to ensure testing effectiveness. Our study confirms these patterns and explores further PBT use-cases (§4.3), insights around shrinking (§4.5), and concerns about testing effectiveness (§4.6).

Other studies highlight a more narrow set of specific challenges faced by developers using PBT. One experience report describing PBT use at DropBox [31] cited usability problems when testing timing-dependent code. The report found that sequences of timed operations resist shrinking, often remaining unwieldy, and that timing dependence caused tests to be flaky. An education-focused study using PBT [55] observed that the PBT community lacks good motivating examples. While the former concern only appeared in passing in our study (when discussing PBT's handling of stateful software), the latter—a dearth of good motivating examples—might be ameliorated by our characterization of *high-leverage* testing scenarios in §5.

Testing more generally. Beyond PBT, there is considerable work studying usability software of testing in general. Our study sheds light into how these common testing issues manifest in PBT specifically. Two studies of developers who use IDEs [5, 6] conclude that testing, especially Test Driven Development, is not as prevalent as conventional wisdom would suggest. Based on these studies, Beller et al. coined the term "Test *Guided* Development" to describe the strategy that programmers actually use. Our study agrees with the idea that developers use testing to guide their thinking during development (§4.1). We also corroborate challenges that another study found around integration testing: Greiler et al. [24] found that while unit testing is common, integration testing is difficult and often left to the the software's users; in our study, developers struggled to use PBT for integration testing because it is difficult to write specifications of entire systems' behaviors. (§4.3).

Our study also suggests that PBT addresses testing difficulties raised in the literature. Aniche et al. [1] observed that developers try to write relatively "random" test cases, with the hope of accidentally stumbling on bugs; this is related to developers' goals for PBT generators, discussed in §4.4, although PBT has the advantage of automating this process in many cases. Another study [25], focusing on differential testing, found PBT to be a valuable tool in a developer's toolbox, providing a coherence check for important code; however, they also found some problems with differential testing systems, such as naïve sampling algorithms that failed to trigger bugs and large counter-examples that were difficult to reason about. Our study suggests new tools may address these problems.

Formal methods. PBT is sometimes described as a "lightweight formal method." Indeed, a recent position paper [49] argued that testing techniques like PBT and fuzzing were important steps on the way towards more formal verification. Formal methods as a field have seen similar calls for usability improvements. A report out of the Naval Research Lab [28] says, "to be useful to software practitioners, most of whom lack advanced mathematical training and theorem proving skills, current formal methods need a number of additional attributes, including more user-friendly notations, completely automatic (i.e., pushbutton) analysis, and useful, easy to understand feedback." This report was published in 1998, but, as our study shows, some of their usability criteria are still not adequately met by modern PBT tools. A more contemporary account agrees that "the user experience of formal methods tools has largely been understudied" [33] and calls for better education and tools for writing and understanding specifications.

7 CONCLUSION

Our study reveals that, even after two decades of active exploration—and, increasingly, exploitation—of PBT, there is still much to learn about how it is being used and what challenges it faces in practice. We contribute a wealth of observations about PBT's use in an industrial setting, along with well-founded ideas for future research in PBT that that we, with the help of the broader community, hope to pursue in the coming years.

ACKNOWLEDGMENTS

We would like to thank John Hughes, Hila Peleg, Zac Hatfield-Dodds, and Shriram Krishnamurthi for their input and feedback on drafts of this paper. Also, thank you to Ron Minsky and others at Jane Street for being open to this kind of collaboration. We appreciate the support of the University of Pennsylvania's PLClub and Penn HCI. This work was financially supported by NSF awards #1421243, *Random Testing for Language Design* and #1521523, *Expeditions in Computing: The Science of Deep Specification.*

REFERENCES

[1] Maurício Aniche, Christoph Treude, and Andy Zaidman. 2022. How Developers Engineer Test Cases: An Observational Study. *IEEE Transactions on Software Engineering* 48, 12 (Dec. 2022), 4925–4946. https://doi.org/10.1109/TSE.2021.3129889

[2] 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

[3] 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

[4] Karen Barrett-Wilt. 2021. The trials and tribulations of academic publishing – and Fuzz Testing. https://www.cs.wisc.edu/2021/01/14/the-trials-and-tribulations-of-academic-publishing-and-fuzz-testing/

[5] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2019. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Transactions on Software Engineering* 45, 3 (March 2019), 261–284. https://doi.org/10.1109/TSE.2017.2776152

[6] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 179–190. https://doi.org/10.1145/2786805.2786843

[7] 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

[8] 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 *SOSP 2021*. https://www.amazon.science/publications/using-lightweight-formal-methods-to-validate-a-key-value-storage-node-in-amazon-s3

[9] 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

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020 event-place: Dallas, Texas, USA.

[11] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (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

[12] Arthur Corgozinho, Marco Valente, and Henrique Rocha. 2023. How Developers Implement Property-Based Tests. In *Conference: 39th International Conference on Software Maintenance and Evolution (ICSME 2023)*.

[13] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 201–211. https://doi.org/10.1109/ISSRE.2014.11 ISSN: 2332-6549.

[14] Natasha Danas, Tim Nelson, Lane Harrison, Shriram Krishnamurthi, and Daniel J. Dougherty. 2017. User Studies of Principled Model Finder Output. In *Software Engineering and Formal Methods (Lecture Notes in Computer Science)*, Alessandro Cimatti and Marjan Sirjani (Eds.). Springer International Publishing, Cham, 168–184. https://doi.org/10.1007/978-3-319-66197-1_11

[15] Zac Hatfield Dodds. 2022. current maintainer of Hypothesis (https://github.com/HypothesisWorks/hypothesis). Personal communication.

[16] Zac Hatfield Dodds and David R. MacIver. 2023. Ghostwriting tests for you — Hypothesis 6.82.0 documentation. https://hypothesis.readthedocs.io/en/latest/ghostwriter.html

[17] Stephen Dolan and Mindy Preston. 2017. Testing with crowbar. In *OCaml Workshop*.

[18] Tristan Dyer, Tim Nelson, Kathi Fisler, and Shriram Krishnamurthi. 2022. Applying cognitive principles to model-finding output: the positive value of negative information. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 79:1–79:29. https://doi.org/10.1145/3527323

[19] Carl Eastlund. 2015. Quickcheck for Core. https://blog.janestreet.com/quickcheck-for-core/

[20] 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

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

[22] 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

[23] 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

[24] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. 2012. Test confessions: A study of testing practices for plug-in systems. In *2012 34th International Conference on Software Engineering (ICSE)*. 244–254. https://doi.org/10.1109/ICSE.2012.6227189 ISSN: 1558-1225.

[25] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. 2019. Perception and Practices of Differential Testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 71–80. https://doi.org/10.1109/ICSE-SEIP.2019.00016

[26] Zac Hatfield Dodds. 2023. HypoFuzz. https://hypofuzz.com/

[27] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2021. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (June 2021), 49:1–49:29. https://doi.org/10.1145/3428334

[28] Constance Heitmeyer. 1998. *On the Need for Practical Formal Methods*. Technical Report. https://apps.dtic.mil/sti/citations/ADA465485 Section: Technical Reports.

[29] 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

[30] 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

[31] 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

[32] JetBrains. 2021. Python Developers Survey 2021 Results. https://lp.jetbrains.com/python-developers-survey-2021/

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

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

[35] 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.

[36] David R. MacIver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer (Tool Insights Paper). In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 13:1–13:27. https://doi.org/10.4230/LIPIcs.ECOOP.2020.13 ISSN: 1868-8969.

[37] 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

[38] Tim Mackinnon, Steve Freeman, and Philip Craig. 2000. Endo-testing: unit testing with mock objects. *Extreme programming examined* (2000), 287–301.

[39] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (dec 1990), 32–44. https://doi.org/10.1145/96267.96279

[40] Minsky. 2015. Testing with expectations. https://blog.janestreet.com/testing-with-expectations/

[41] Liam O'Connor and Oskar Wickström. 2022. Quickstrom: property-based acceptance testing with LTL specifications. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 1025–1038. https://doi.org/10.1145/3519939.3523728

[42] Otter.ai. 2023. Otter.ai - Voice Meeting Notes & Real-time Transcription. https://otter.ai/

[43] Michał H. Pałka, 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.

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

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

[46] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *Interactive Theorem Proving (Lecture Notes in Computer Science)*, Christian Urban and Xingyuan Zhang (Eds.). Springer International Publishing, Cham, 325–343. https://doi.org/10.1007/978-3-319-22102-1_22

[47] Goran Petrovic and Marko Ivankovic. 2018. State of Mutation Testing at Google. In *Proceedings of the 40th International Conference on Software Engineering 2017 (SEIP)*.

[48] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly generating diverse valid test inputs with reinforcement learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1410–1421. https://doi.org/10.1145/3377811.3380399

[49] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. 2020. Towards making formal methods normal: meeting developers where they are. http://arxiv.org/abs/2010.16345 arXiv:2010.16345 [cs].

[50] 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

[51] 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.

[52] Jacob Stanley. 2017. Hedgehog will eat all your bugs. https://hedgehog.qa/

[53] 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

[54] Mark Utting and Bruno Legeard. 2010. *Practical Model-Based Testing: A Tools Approach*. Elsevier.

[55] John Wrenn, Tim Nelson, and Shriram Krishnamurthi. 2021. Using Relational Problems to Teach Property-Based Testing. *The art science and engineering of programming* 5, 2 (Jan. 2021). https://doi.org/10.22152/programming-journal.org/2021/5/9

[56] Michał Zalewski. 2022. American Fuzzy Lop (AFL). https://github.com/google/AFL original-date: 2019-07-25T16:50:06Z.