Problems with Properties

A Study on Property-Based Testing in Industry

ANONYMOUS AUTHOR(S)

Property-based testing (PBT) allows testers to write rigorous specifications of software behavior in the form of executable code and to validate those specifications via random testing, thus offering higher confidence than simple unit tests while avoiding the cost and overhead of heavier-weight formal methods. It is much loved by academic functional programmers and in certain industrial niches. But, despite its success in these communities, PBT has not been widely adopted by industrial software engineers. Why?

We carried out a preliminary study to explore this question via semi-structured interviews with developers, finding that PBT suffers from both technical complexity (e.g., in expressing properties and random data generators) and usability concerns (e.g., failure to fit cleanly into developer workflows). While these findings largely match prior intuitions, they add considerable detail and highlight some unexpected challenges.

Building on the momentum from this initial study, we describe a larger planned study, to be carried out in partnership with Jane Street, LLC, a quantitative trading firm. This full-scale study will target a set of research questions that were refined by the preliminary study. We also discuss potential longer-term projects to continue this research effort. We hope that publicizing this research program within the HATRA community will help galvanize further research into how to make PBT more widely useful.

1 INTRODUCTION

Property-based testing (PBT) is a powerful tool for increasing confidence in software correctness. Testers write formal specifications of the behaviors that they expect from a program in the form of *properties*—functions that validate the program's correctness on a single given input—and the testing framework checks those specifications on a large number of randomly generated inputs. PBT thus occupies an attractive sweet spot between classical unit testing and full-blown formal methods, combining the low cost of the former with the rigor of the latter.

This extra rigor has practical benefits. Software engineers that we interviewed said that

> " [PBT] appealed to me as someone who makes a lot of mistakes and wants the computer to find them, "

and

> " I've found probably half a dozen major corner-case bugs [with PBT]. "

One developer gushed that PBT was "*1000 times better than the alternative*."

PBT has seen some impressive successes in industry—in telecommunications software [Arts et al. 2006], replicated file-stores [Hughes et al. 2014], cars [Arts et al. 2015], and a range of other areas [Hughes 2016]. Indeed, some correctness-conscious development teams consider PBT a core part of their toolkit [Bornholt et al. 2021].

Yet, despite its demonstrated potential, PBT has not been adopted broadly in the software industry. Python's main PBT library, Hypothesis, seen approximately 3.5 million monthly downloads, far fewer than the the 45 million downloads for the dominant testing framework, pytest. The story is similar in other languages. When a technology that has been so successful in some areas receives such a lukewarm reception elsewhere, there is clearly room for improvement! This leads us to our central question: **How can the research community make PBT more valuable for software engineers?**

We envision a research program consisting of need-finding studies to determine the challenges facing PBT's adoption and projects to address those challenges. After a brief discussion of background and related work (Section 2), we make the following contributions.

- We describe a preliminary study (the source of the above quotes) that uses semi-structured interviews to begin to understand opportunities for PBT in the software industry. The themes from the study are a compelling baseline for the challenges that PBT faces. (Section 3)
- We outline a plan for a study that expands on the preliminary study with a larger population and refined focus. Our study population will tentatively be composed of developers from Jane Street, LLC, a well-respected financial technology company. (Section 4)

To conclude, we discuss future projects in this space, outlining a research agenda that, with luck, will make PBT much more valuable to software engineers (Section 5).

2 BACKGROUND AND RELATED WORK

Property-based testing (PBT) was popularized by the QuickCheck library in Haskell [Claessen and Hughes 2000]. In PBT, users write executable functions that act as partial specifications of a function under test. For example, a tester might write the following property that specifies the correctness of a topological sorting algorithm:

```
prop_topoCorrect g = let s = toposort g in
  all (\(v, w) -> index v s < index w s) (edges g)
```

This property is just a function that takes a graph as input and asserts that topologically sorting that graph results in a node order that agrees with the graph's edge relation. Others have observed that correctness condition like this are difficult to express in traditional unit tests, since there are multiple valid outputs for any given input [Wrenn et al. 2021].

With a property in hand, the tester then *randomly generates* inputs to the property, and checks the property for each one. If the property returns `True` for every generated input example, the tester can be relatively confident[1] that the function under test is correct. If any graph causes the function to return `False`, it is called a *counterexample* for the property and it shows that there must be something wrong with the implementation.

Successfully applying PBT requires not only good properties, but also good random generators. A bad random generator may fail to produce varied inputs that exercise different code paths, or it may generate inputs that fail to satisfy a property's *preconditions*. For example, a property concerning binary search trees (BSTs) may require that a tree passed as an input satisfies the BST invariants. The second of these failure modes is particularly problematic because it forces the tester to *rejection sample*—i.e., only test with precondition-satisfying values, discarding bad ones—which is often intractably slow. Thus, PBT ecosystems often provide abstractions for developing random generators that are tailored to the particular properties under test. The design of these abstractions is a topic of active research [Dewey 2017; Goldstein and Pierce 2022; Lampropoulos et al. 2017; Runciman et al. 2008].

We use semi-structured interviews to understand how researchers can add value to PBT for software engineers. This approach has precedent in the literature. For example, Johnson et al. use interviews to explore why software developers do not use available static analysis tools to their full effect. In a similar vein, Chattopadhyay et al. talked to developers to understand problems facing computational notebooks. Both of these studies had sample sizes comparable to our intended sample size for the full-scale study; the latter also had a survey component which we consider in Section 5.

Additionally, the population for our large-scale study includes some contributors to PBT libraries. This approach was partially inspired by prior work that explores usability of a particular tool

[1]The precise confidence is actually quite difficult to compute in general, and depends heavily on the number of samples that the tester checks, but random testing like this is surprisingly effective.

by interviewing both both producers and consumers of the tool [Dagenais and Robillard 2010; Robillard and DeLine 2011].

3 PRELIMINARY STUDY

To get an initial handle on the challenges facing users of PBT, we designed and ran a small-scale preliminary study. The core of the study was a set of short semi-structured interviews with developers who had used PBT in the past. Our goal was not necessarily to arrive at statistically significant conclusions; instead, we hoped to discover the issues impeding the use of in PBT practice. Ideally, we hoped the preliminary study would help us to refine our understanding of the challenges facing PBT, allowing us to make preliminary recommendations and focus our attention in a larger-scale study.

3.1 Study Population

We hoped to obtain a relatively cohesive signal from the interviews, so we focused our recruiting primarily on developers with experience using Hypothesis, the most well-known PBT library for an imperative language (Python) [MacIver et al. 2019]. We hoped that this choice would focus us on a group of professional developers with a wide range of backgrounds, working in an equally wide range of areas. While all of our participants did have some experience with Hypothesis, some of the participants described their experiences with other PBT libraries in other languages.

We recruited participants using our professional networks, social media, and by cold-emailing authors of public articles and blog posts about PBT. All told, we recruited and interviewed seven participants (who we refer to as P1–P7). All identified as male, and all were between the ages of 27 and 42. The participants had between 4 and 28 years writing code, and between 2 and 15 years doing so as a professional developer. Of those years, the participants reported actively using PBT tools for between 0.5 and 8 years, total . Six of our seven participants had advanced degrees, with three holding doctorates.

3.2 Interview Design and Questions

Our interview procedure was intended to feel like a casual conversation to get the participants comfortable talking their successes and failures using PBT. All of the interviews were thirty minutes long, and conducted over Zoom. The Zoom calls were all recorded, and then automatically transcribed using Otter.ai[2] for further analysis. Three team members participated in each interview, with one doing the actual interviewing, one taking notes on the participant's answers, and one taking meta-level procedural notes about the interview itself to improve procedures.

Our interviewer used a loose script containing three main prompts, each intended to spur about ten minutes of discussion, along with some optional prompts that the interviewer could choose from if there was remaining time. For all of the prompts, we took care to choose our phrasing such a way that they were not leading, but still focused the participant's attention enough that we could extract useful information from their answer.

The main three prompts were the following:

(1) Tell us about your most memorable experience applying PBT.
(2) How did you come up with the properties that you tested?
(3) Did you need to write custom generators? Or do you have an example of a project where you did?

[2]While the transcription was not perfect, it was accurate enough for our open and axial coding in Section 3.3. The quotes in this paper are all manually re-transcribed from the recorded audio.

Prompt (1) got the participant thinking about a particular experience so they could walk us through their experience; asking about a particular experience prevented prevent general, high-level pontification. Next, prompt (2) was intended to get the participant talking about one of the main two components of PBT, the properties. We hoped we might hear about problems that users had expressing properties for their code. Finally, prompt (3) asked about the other component of PBT, the random data generators that produce test inputs. Since generators can be either very simple and require only a basic use of the library's combinators, or very complex and require substantial programming to enforce complex data invariants, we in particular asked the participants to tell us about any experience they'd had writing generators which enforced complex preconditions.

Finally, we attempted to use snowball sampling to find more study participants. Although we did not successfully recruit via this method, the ensuing discussions made it clear that PBT was often not widely used or understood in our participants' companies (see Section 3.3 for more).

3.3 Thematic Analysis

With our interview transcripts in hand, we began a thematic analysis (à la [Blandford et al. 2016]) of our corpus of qualitative data. To begin, multiple authors independently conducted open codings of the first three of the seven transcripts, building individual sets of codes as well as groupings of their codes into their own broad categories. After the open coding was complete, the participating authors convened to reconcile their individual codings. We then repeated this process with the remaining four transcripts, at which point the codes had converged. Finally, one of the authors performed an in-depth axial coding pass on all seven transcripts.

Our thematic analysis uncovered a number of challenges that, if addressed, could make PBT more valuable to developers. While addressing these challenges is likely not sufficient to ensure that PBT will become broadly adopted, many of them are likely roadblocks that are necessary to address.

Challenges Designing Properties. One prominent challenge we identified was that users often struggle to design the properties that they need to test their code.

We identified a number of "levels" of property difficulties that developers faced, ranging from abstract questions about what it means for a particular program to be correct to concrete issues with implementing a specification as an executable property.

At the most abstract level developers recounted struggling to define what it means for their code to be correct. P5 said:

> " So [PBT] was useful. It was really useful. But I also think, broadly speaking, you're going to struggle to see adoption in typical software projects, because it's very hard to independently verify that your inputs and outputs are indeed correct, using tools that are not already part of the process that you want to test. "

In other words: the code does what it does. Input-output pairs are correct only by virtue of coming from the function. Of course, there are approaches to PBT that side-step this issue (e.g., testinga an application against a previous version or a reference implementation), but in general PBT does require that there be some notion of correctness available as a basis for properties.

One level more concrete, some developers noted that even when they had an idea of what correctness meant for their code, they did not always know how to express those correctness conditions in the form of properties. We call this a failure to "imagine" properties. A quintessential example of this phenomenon was described by P2: they recounted that while they were able to specify their code in terms of unit tests, they struggled to determine a general property that those unit tests implied.

Finally, at the most concrete level, some developers that have generalized notions of correctness in mind, but struggle to actually implement them in code. P6 described being unable to find "compose

points where [they could] integrate [properties] with the system." Without these clean abstraction boundaries, P6 could not cleanly express the specification they had in mind as concrete logical code. Likewise, P4 had difficulty implementing a property for a web application. Once again, the participant had an intuitive understanding of what a partial correctness property could mean, intending to compare the behavior of their function to a simpler "oracle" program that computes the same results but that is more obviously correct. Unfortunately, while this is sometimes a reasonable approach (e.g., if the function is conceptually simple, but highly optimized, it is possible to write an oracle implementation that is clearly correct), in this case the participant simply could find "no real easy way to create an oracle," and thus could not use PBT.

These levels of property challenges are much more complex than we originally expected, and suggest that there are likely multiple avenues to improving property design at different levels of abstraction.

Challenges Writing Random Data Generators. Also in line with the focus of our prompts, some participants described struggling to write custom generators that adequately exercised the code under test. For example, some participants recounted encountering situations where it was difficult to write a generator to enforce complex preconditions.

P7 described finding the rejection sampling approach to generation—generating arbitrary inputs and throwing out those which do not satisfy the precondition in question—too time-consuming to be useful. The standard solution would be to write a generator that enforces preconditions by construction, and the participant did highlight that 'it might be possible to make the generator smarter," but they were worried that it "may introduce a ton of complexity, which may or may not be worth it."

In another expected instance, some participants described having difficulty with controlling their generators' distributions. P6 recounted:

> " I just relied on the built in [generators]. But… [there were] some tests that were running for quite a long time. Sometimes they were missing some useful use cases, just because the data distribution targeted use cases were missed by the default generators. "

In addition to the aforementioned technical issues, we also participants participants who encountered usability problems. P3 struggled to write a generator due to a difficult-to-understand generator abstraction. They explained that the abstraction did not match their expectations and they could not "hold it in [their] head very easily." This highlights the importance of *idiomatic* generator abstractions.

Challenges Fitting PBT into Programmer Workflows. One unanticipated finding was that some participants struggled to incorporate PBT into their team's development workflows. One especially interesting issue that came up was exactly where and when properties should be checked. P4 explained that they generally only check their properties in their continuous integration (CI) system, because they are too slow to run locally. However, they also pointed out that that properties in CI are "a bit like Russian roulette, because sometimes the code breaks while you're working on something totally unrelated." Indeed, P1 agreed that they preferred to not have properties in CI for that reason. The tension created by PBT being both long-running and nondeterministic is one that we had not considered as a usability issue until these conversations, and it suggests a number of opportunities for research into improved PBT tooling and workflows.

Challenges Conceptualizing PBT. Another theme we did not necessarily anticipate hearing about was our participants' challenges with learning or teaching PBT. Multiple participants reported that for themselves or for others on their team, PBT was simply not a natural concept. P7 described that while their own programming background made PBT understandable, members of their team came

from different programming subcultures which made PBT seem like an unnatural tool to use. They continued:

> " I think people are kind of interested in the idea of property based testing whenever it comes up, but they don't reach for it naturally. "

Meanwhile, P4 expressed fear over showing their codebase's property based tests to newcomers to the team, because the newcomers may get lost in all of the new complexity. P4 also had the following complaint:

> " I don't think there is much literature on it, at least not something really accessible. "

This is something we suspected from our own experience, but that we did not necessarily expect to hear so clearly from our participants.

The Bootstrapping Challenge. Finally, some of our participants' responses underlined a far more basic and rudely circular impediment to PBT adoption: for PBT to become widely adopted, it must first become popular. This was best illustrated by P5, when asked if they thought that PBT was missing any features that could help drive adoption.

> " Well, I think the problem is really more one of popularity, more than a technical problem. If lots of people using property based testing, then more people will be adept at using it. And people will be more willing to write code that might support such a methodology of testing. "

P6 also implied a very similar bootstrapping problem when describing how their managers reacted to a property-based test that found a potentially severe bug. Despite their excitement around the success story, P6's managers were unwilling to further invest in PBT due to its relative unpopularity.

> " They realize that [PBT is] very important and very useful, but can they find somebody to write those tests and maintain those tests? That's another question. "

This bootstrapping problem is not specific to PBT. Indeed, many technologies have struggled with it . This is a helpful reminder that solving both the technical problems above and the as-of-yet hidden ones would not be sufficient to ensure broad adoption of PBT. To truly see PBT used across the software industry, researchers need to convince developers, managers, and community leaders, to invest in PBT without guarantees that it will take off.

4 PLANNED WORK: FULL-SCALE STUDY

In this section we describe an ongoing study that builds on the preliminary study from Section 3. We plan to partner with Jane Street, LLC to gain deeper insights into the challenges facing PBT and to begin to explore potential solutions

4.1 Research Questions

Our research questions for the full-scale study should again address our central question: How can the research community make PBT more valuable for software engineers? Recall that our preliminary study highlighted five categories of challenges that PBT faces: property, generator, workflow, learning, and social. We expect that learning challenges are best explored in a classroom setting (see Section 5), and that the social challenge is too intertwined with the other challenges to address head-on at this point. Accordingly, our research questions for the full-scale will focus on property, generator, and workflow challenges.

RQ1. What support do testers need when writing properties? What at what "level" do their property difficulties fall?

RQ2. What kinds of generators do testers need to exercise their properties? Do they have specific precondition and/or distributional requirements?

RQ3. How does PBT fit into testing workflows? Could it fit better?

RQ4. What concrete improvements could be made to modern PBT systems to improve effectiveness and usability?

It should be clear how **RQ1**, **RQ2**, and **RQ2** relate to property, generator, and workflow challenges respectively. In addition to these questions, we also consider **RQ4**, which more directly asks how existing systems might be able to be improved. This final question will help to keep us focused, reminding us that participants likely have the context and knowledge to suggest solutions to some of the challenges we have identified and that we should not waste the opportunity to ask their opinions.

4.2 Study Population

As in the preliminary study, our primary tool will be interviews with software developers. We will tentatively partner with Jane Street, LLC, a large financial technology firm, to carry out our full-scale study. We plan to interview 30 Jane Street employees about their experiences using PBT.

Jane Street has a number of qualities that makes it an ideal place to carry out these kinds of inquiries. To start, we are aware that Jane Street developers use a variety of testing tools, including PBT. This means we have a place to start when asking questions, since developers will likely have seen PBT before, but there will be room to explore the trade-offs between PBT and other options. Additionally, Jane Street developers are famous for writing almost all of their code in OCaml, a mostly-functional programming language with strong support for static typing and modularity. A unified ecosystem like this will allow us to control for a number of potentially confounding factors; all of the developers we talk to will have access to the same quality of PBT tools and access to programming abstractions that may make testing easier or harder.

Of course, carrying out a study at a single firm has potential drawbacks as well. The most obvious issue is that our results may not generalize. Indeed, we cannot guarantee that our results apply outside of the OCaml ecosystem (and other ecosystems like it). However, Jane Street is home to a diverse array of software systems, including but not limited to: trading systems, quantitative algorithms, networked systems, and hardware description code [Minsky 2022]. We hope that the breadth of these programming tasks will mean that the software engineers that we talk to will come to the discussion with a wide variety of experiences.

Stakeholders. Our initial discussions with Jane Street leadership made it clear that there are really two populations at Jane Street that we can learn from. The population we initially planned to talk to was Jane Street developers who have used PBT in their work at the firm. These developers can tell us about how PBT helped them, what challenges they might have faced, and what techniques they use instead of PBT to check that their code is correct. But we realized that we can also learn a ton by talking to PBT *stakeholders* at Jane Street—i.e., the folks who build and maintain the PBT systems that Jane Street uses. Since we expect that stakeholders may also help us to refine our developer interview script, we plan to interview stakeholders first.

4.3 Interview Plan

Each interview will be allotted a 1 hour slot, to account for a more detailed interview script than the one in the preliminary study. As mentioned above, we will start by interviewing stakeholders. Our prompts will primarily set the stage for our developer interviews and establish background that will help us to interpret our results, but stakeholders may also be able to help us answer **RQ4** (and to a lesser extent other research questions) directly. We will pick the stakeholder's brains about the most promising opportunities to support users of PBT, phrasing our prompts to get our participants thinking creatively.

After talking to stakeholders, we will begin the developer interviews. The script for these interviews will depend on our conversations with stakeholders, but at a high level we will aim to answer our research questions as directly as possible. Much like the preliminary study, we will have our participants tell us about a specific experience with PBT and ask them about the properties and generators that they used (hopefully giving us insights about **RQ1** and **RQ2**). We will also ask about whether or not they are using PBT on their *current* project (and if not, why not). Finally, we will ask questions addressing **RQ3** and **RQ4** directly.

5 LOOKING FORWARD

Our preliminary study is compelling evidence that talking to developers can help us to refine our understanding of the challenges facing PBT. We are excited to complete the full-scale study and uncover high-leverage opportunities to improve PBT for industry users. While our future research plans depend strongly on the results of the full-scale study, our preliminary study suggests a few directions that may make sense. We outline those potential projects here.

5.1 Observations of Practice

The data we collect from interviews will reflect developers' *impressions* of their work, but it may miss important details that were simply not memorable. Accordingly, we hope to eventually follow up with one or more behavioral studies, observing developers in practice.

There are a few options for exactly what such a behavioral study might look like: we may observe developers completing a pre-determined task, or we may opt for a more realistic contextual inquiry [Beyer and Holtzblatt 1999]. Setting developers a task would make the study straightforward to run and recruit for, but the choice of task may limit the breadth of the observations we are able to make. A contextual inquiry would likely be more informative, but it would be much more difficult to recruit for, since we would need to find developers with realistic PBT work that they can do in front of us. We will use the outcomes of the full-scale interview study, as well as feedback from the community, to decide how to proceed.

5.2 Quantitative Surveys

The full-scale study will be far larger than our preliminary study, but our target is still only 30 total participants. We would need a far larger population if we wanted to draw statistically significant quantitative conclusions. Interviewing hundreds or thousands of developers would be incredibly expensive, but distributing a survey to a population of that size would be much more reasonable. Thus, we propose a wide-reaching survey to confirm or refute the qualitative outcomes of our full-scale study.

5.3 Education

Our preliminary study pointed out that one major challenge for PBT is that it builds on concepts that are not necessarily comfortable for the average developer. Prior work has looked at ways to close this knowledge gap [Nelson et al. 2021; Wrenn et al. 2021], but we expect that there are more education challenges around PBT that are worth exploring. We plan to incorporate PBT into a major-required course at a large university.

5.4 Concrete Tools

In the course of our interviews, it will likely become clear that the solutions to some of the challenges facing PBT are within reach. For example, we are hopeful that some of the workflow challenges mentioned in Section 3 can be addressed via standard HCI methods, and that some of the generator challenges are within reach with PL ideas.

While the bulk of this report has been about need-finding, we have no intention to stop there. We want to take the insights gained through interviews (and potentially observations and surveys), and apply them in the real world to build compelling tools for software developers.

REFERENCES

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*. 2–10.

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)*. IEEE, 1–4.

Hugh Beyer and Karen Holtzblatt. 1999. Contextual design. *interactions* 6, 1 (1999), 32–42.

Ann Blandford, Dominic Furniss, and Stephann Makri. 2016. Qualitative HCI research: Going behind the scenes. *Synthesis lectures on human-centered informatics* 9, 1 (2016), 1–115.

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

Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. 2020. What's wrong with computational notebooks? Pain points, needs, and design opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–12.

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

Barthélémy Dagenais and Martin P Robillard. 2010. Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. 127–136.

Kyle Thomas Dewey. 2017. *Automated Black Box Generation of Structured Inputs for Use in Software Testing*. University of California, Santa Barbara.

Harrison Goldstein and Benjamin C. Pierce. 2022. Parsing Randomness: Unifying and Differentiating Parsers and Random Generators. arXiv:2203.00652 [cs.PL]

John Hughes. 2016. Experiences with QuickCheck: testing the hard stuff and staying sane. In *A List of Successes That Can Change the World*. Springer, 169–186.

John Hughes, Benjamin Pierce, Thomas Arts, and Ulf Norell. 2014. Mysteries of Dropbox. (2014).

Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 114–129. http://dl.acm.org/citation.cfm?id=3009868

David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891.

Yaron Minsky. 2022. Signals and Threads. Web. https://signalsandthreads.com/

Tim Nelson, Elijah Rivera, Sam Soucie, Thomas Del Vecchio, John Wrenn, and Shriram Krishnamurthi. 2021. Automated, Targeted Testing of Property-Based Testing Predicates. *arXiv preprint arXiv:2111.10414* (2021).

Martin P Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (2011), 703–732.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, Andy Gill (Ed.). ACM, 37–48. https://doi.org/10.1145/1411286.1411292

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