Some Problems with Properties

A Study on Property-Based Testing in Industry

HARRISON GOLDSTEIN, University of Pennsylvania, USA
JOSEPH W. CUTLER, University of Pennsylvania, USA
ADAM STEIN, University of Pennsylvania, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA
ANDREW HEAD, University of Pennsylvania, USA

Property-based testing (PBT) is a popular testing methodology from the functional programming community that is beginning to see broader use. For testers who already think in terms of formal specifications of the behavior of their code, PBT provides an easy way to check those specifications; for those who do not yet, PBT's successful track record offers a good reason to start. However, like any up-and-coming technology, PBT has plenty of room to grow and improve. In particular, the challenges and opportunities of applying PBT in an industrial setting are not clearly understood.

We present results from a preliminary study that we conducted to investigate this question. Drawing on interviews with seven professional users of a contemporary PBT tool, we identify several areas sorely in need of attention if PBT tools are to better meet developers where they are. Namely, developers encountered obstacles envisioning properties to test, defining generators, and integrating PBT into their development workflow. We detail participants' challenges, highlighting directions where further research and tool-building efforts could mitigate the challenges.

We conclude with an agenda for further deepening understanding of the pitfalls and potentials of PBT in industry. We offer a brief overview of research activities we plan to undertake in the coming years, including an expanded interview study to be carried out in partnership with Jane Street, LLC, a large software company that leads in its use of functional programming tools. We hope that advertising 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 behavior 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. For testers who care about formal specification but don't want to pay the hefty cost of full-scale verification, PBT offers a simple and effective middle ground.

Even for testers who don't care about formal specification in its own right, there is mounting evidence that PBT a powerful bug-finding strategy. It has been used to find critical bugs in a range of real-world software, including telecommunications software [Arts et al. 2006], replicated file-stores [Hughes et al. 2014], cars [Arts et al. 2015], and more [Hughes 2016]. Some industrial development teams already consider PBT a core part of their toolkit [Bornholt et al. 2021]. And our interviews with software developers about PBT have yielded comments consistent with this positive perspective, with one recounting that they had "found probably half a dozen major corner-case bugs [with PBT]" and another gushing that PBT was "1000 times better than the alternative [testing options]."

Despite its demonstrated potential, PBT still has plenty of room to grow. It is a relatively new technology that has yet to become established across industry. If we want to increase adoption of

Authors' addresses: 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; Adam Stein, University of Pennsylvania, Philadelphia, PA, USA, steinad@seas.upenn.edu; 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.

PBT, there are likely challenges that need to be overcome. This leads us to our central question: **How can the research community make PBT more useful for real-world software developers?**

In this paper, we describe the first steps in a research program that aims to find and address challenges facing PBT's adoption. Our immediate focus is a pair of need-finding studies that, when complete, will provide the community with a much more robust model of available opportunities to improve PBT. After a brief discussion of background and related work (Section 2), we offer the following contributions.

- We describe a small ($N = 7$) preliminary study using semi-structured interviews to begin to understand opportunities for PBT in the software industry (Section 3). The themes from this study offer an initial model of the challenges that PBT faces.
- We outline plans for a future study with a larger population ($N \approx 30$) and a focus refined by the conceptual model of the preliminary study. The study population will be drawn from the developers at Jane Street, LLC, a well-respected quantitative trading firm (Section 4).

In closing, we discuss some possible further projects in this space, outlining a research agenda that, we hope, will make PBT even more valuable to software developers of the future (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 to specify 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 an ordinary function that takes a graph as input and asserts that topologically sorting that graph results in a node ordering that agrees with the graph's edge relation. Others have observed that correctness conditions like this are difficult to express with traditional unit tests, since there are multiple valid outputs for any given input [Wrenn et al. 2021].

Given such a property, the testing framework *randomly generates* input graphs and checks that property holds for each one. If any graph causes the function to return `False`, then we have a *counterexample* for the property, showing that there is something wrong with the implementation. If it returns `True` for every generated input, we have increased confidence that the function under test is correct. (The appropriate increase in confidence is, of course, difficult to estimate precisely, since PBT, like unit testing, is an incomplete process. But standard techniques like delta debugging [Zeller and Hildebrandt 2002] and coverage measurements [Lampropoulos et al. 2019] can help.)

Applying PBT successfully demands not only good properties, but also good *random generators*. A random generator is a program that defines the space of inputs that are used to exercise the property, and not all generators are equally useful. A bad generator may fail to produce sufficiently varied inputs to exercise all possible code paths, or it may generate inputs that fail to satisfy a property's *preconditions*. For example, a property about binary search trees (BSTs) may only behave properly on trees that satisfy the BST invariants. The latter failure mode is particularly problematic because automatic testing frameworks deal with it using *rejection sampling*—generating values that may or may not satisfy preconditions and discarding the ones that do not—which is often intractably slow. Thus, PBT ecosystems often provide abstractions for developing random generators custom 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, etc.].

Our studies use semi-structured interviews to understand how researchers can add value to PBT for software developers, an approach with ample precedent in the literature. For example, one study used interviews to explore why software developers do not use available static analysis tools to their full effect [Johnson et al. 2013], and another talked to developers to understand problems facing computational notebooks [Chattopadhyay et al. 2020]. Both of these studies used sample sizes comparable to our intended full-scale study; the latter also had a survey component, which we consider in Section 5.

The planned population for our large-scale study also includes some developers and contributors to PBT libraries. This approach was partially inspired by prior work that explores usability of a particular tool by interviewing both producers and consumers of the tool [Dagenais and Robillard 2010; Robillard and DeLine 2011].

3 PRELIMINARY STUDY

To begin getting a handle on the challenges facing users of PBT, we designed and ran a preliminary study in the Spring of 2022. The core of the study was a set of short semi-structured interviews with developers who had used PBT in the past. The goal in this small-scale study was not to arrive at statistically significant conclusions; rather, we hoped to clarify some of the issues impeding the use of PBT in practice and sharpen our ideas for a later, larger study.

3.1 Study Population

To help obtain a cohesive signal from the interviews, we focused our recruiting primarily on professional developers with experience using Hypothesis, the best-known PBT library for an imperative language (Python) [MacIver et al. 2019]. While all of our participants had used Hypothesis, some also described 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; we refer to them below as P1–P7. All identified as male, and all were between the ages of 27 and 42. Six of the seven had advanced degrees, with three holding doctorates. They had between 4 and 28 (median 14) years of experience writing code and between 2 and 15 years doing so professionally. The participants reported actively using PBT tools for between 0.5 and 8 years, total.

3.2 Interview Design and Questions

Our interview procedure was designed to feel like a casual conversation, to get the participants comfortable talking about successes and failures using PBT. Each interview was thirty minutes long, conducted over Zoom. These Zoom calls were recorded, then automatically transcribed using Otter.ai for further analysis. (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.) 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 notes about the interview itself to improve procedures.

The interviewer followed a loose script comprising three main prompts, each intended to spur about ten minutes of discussion, along with some optional prompts that the interviewer could use if time allowed. For all of the prompts, we strove to avoid leading phrasing while still focusing participants' attention enough to extract useful information from their answers.

The three main prompts were:

(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?

Prompt (1) got the participant thinking about a particular experience, preventing general, high-level pontification. Prompt (2) was intended to get the participant talking about one of the main two components of PBT, the properties. We hoped to hear about problems that users had experienced in expressing properties for their code. Finally, prompt (3) asked about the other aspect of PBT—the random generators for test inputs. Testers write a range of generators, from very simple ones that require only a basic use of the library's combinators to complex ones that require substantial programming, so we were sure to focus the conversation on generators that 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; we discuss this further below.

3.3 Thematic Analysis

We analyzed the interview transcripts following the *thematic analysis* methodology described by Blandford et al. [2016]. The analysis began with *open coding* of 3 of the 7 transcripts. Three authors ("coders") independently coded the transcripts, determining an initial set of descriptive codes and categories of those codes, focusing primarily on the obstacles informants encountered. The codes were refined through discussion among the three coders. Then the remaining four transcripts were independently analyzed by the coders with the refined codes, with each author further revising the codes as needed. A preliminary set of codes was then distilled through additional discussion among the three coders.

A comprehensive axial coding was then performed by one of the authors using the agreed-upon set of codes. Another author reviewed the analysis, providing suggestions and pointing out inconsistencies. The analysis was revised to incorporate this feedback, and then validated by another author once again.

What emerged were five categories of obstacles that developers encountered, relating to the design of properties, the definition of generators, learning PBT, integrating PBT into team workflows, and achieving critical mass in the use of the tools. In Section 5, we describe how these challenges can help shape the research and design of usable PBT tools. Below, we detail these challenges, referring to particular participants by pseudonyms P1–7. Quotes from participants are lightly edited for clarity, removing filler words and backtracking. We note that, given the small scale of this preliminary study, some of the obstacles are reflective of singular experiences, and the set of obstacles may not be complete. This highlights the need for expanded studies of the sort we describe in Section 4 to confirm and deepen our understanding of the usability of PBT tools.

Challenges Imagining Properties. One common challenge for developers was to identify, design, and articulate properties that would meaningfully test their code.

To begin with, developers seemed to have trouble identifying use-cases for PBT that took advantage of the unique benefits of PBT over alternative approaches. While participants did write properties, often times they either under-specified the behavior of the system (for instance, simply testing that a program does not crash), or over-specifying it (for instance, comparing to a behaviorally complete *model* of the program under test, which might need to be separately implemented) (P1, P3–5). These participants were not testing properties as is conceptualized in

conventional PBT, which either undermined the power of their tests or led to tests that could be brittle or time-consuming to implement.

In other cases, developer simply did not know how to express the notion of correctness of their program as a property, which we call a failure to "imagine" a property. P1 described this as the problem of "formulating the right property in the first place," a difficulty they had experienced despite prior successes using PBT according to an over-specified model-based approach. P2 described circumstances in which this challenge to imagine properties might arise: they were testing a complex numerical computation involved in financial risk modeling. While they anticipated that properties could help them test the validity of the computation better than their existing suite of unit tests, they emphasized that determining such properties was not at all straightforward.

Once a developer can imagine a property, the next challenge is to implement it. Implementation challenges were of several sorts (P4–6). One is common to software design generally—that module boundaries must be present around the functionality to be tested. P6 described this as the challenge of finding "compose points where you can integrate [PBT] with the system." Refactoring code to be testable can be tedious at best and prohibitively time-consuming at worst. P6 reported that, for them, often the costs of restructuring code outstripped the benefits, leading them not to use PBT where it might otherwise have been useful.

Challenges Writing Random Data Generators. Developers also faced obstacles writing data generators for their property-based tests. Some developers reported that they lacked support for generating the kinds of distributions of input data that they felt were necessary for their tests.

P7 recounted testing an application where the inputs were network access control lists (ACLs) with many different validity conditions, each given by a different hardware vendor. In this situation, P7 found that the *rejection sampling* approach to generation—generating arbitrary ACLs and throwing out invalid ones—was too slow to be useful. The standard solution would be to write a generator that enforces preconditions by construction. Indeed, the participant did note that "it might be possible to make the generator smarter." However, they continued that it "may or may not be worth it," due to the complexity of generating access control lists that are valid for *all* of the vendors.

In another expected finding, some participants described difficulties with controlling the distribution over test cases that their generators produce. As P6 recounted, poor test-case distributions can lead to long testing times and missed bugs as the generator repeatedly tries similar inputs and misses importantly different ones.

> " 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. "

Besides these technical issues, participants encountered usability problems working with generator tools. P3 struggled to write a generator in Hypothesis' generator DSL. They explained that the (purely-functional style) abstraction did not match their expectations, given that Python is an imperative language, and they could not "hold it in [their] head very easily." This highlights the importance of *idiomatic* generator abstractions.

Challenges Integrating PBT into Programmer Workflows. One unanticipated finding was that some participants struggled to incorporate PBT into their team's development process. One issue that came up was exactly where properties should be checked in the development pipeline. 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, because property-based testing finds bugs at random, running PBT in CI is "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.

Challenges Learning PBT. For some developers, the process of learning PBT proved a challenge in and of itself. While developers learned about PBT concepts from presentation materials, library documentation, and blog posts, they sometimes found these sources inaccessible or otherwise insufficient for learning how to apply PBT in practice. For example, P4 described finding it challenging to understand the circumstances in which to use PBT after having watched a presentation about it, and P1 expressed a desire for the library documentation to include a comprehensive corpus of examples to use as starting points for their own tests:

> " The hard part is knowing when you have a property that you can test and how to write that test. What would make it easier? I almost feel like more examples, you know, like… just examples upon examples upon examples, so that you can just read through them and eventually hit one that, you know, hit a couple that click, and then those are clicked in your mind. And if you see that again, you can apply [them]? "

Developers also recounted trepidation at the idea of explaining PBT to their organization's newcomers, fearing that it represented yet another unfamiliar tool to learn in their already complex stack.

> " It's a bit scary, because every time I have to show the code base to a newcomer, there is plenty of complicated things in the code base… and when you talk about all those things, which are complicated or interconnected, you then go on, 'By the way, there is this Hypothesis thing, which is really a bit more complicated'… so, **laughs** … I don't know if when I talk about that people are too lost. "

The Critical Mass Problem. Finally, some of our participants' responses underlined a far more basic and somewhat 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 use 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 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, but this is a helpful reminder that solving both the technical problems above and the as-of-yet hidden ones is necessary, but likely not 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.

Our preliminary study identified a number of important growth opportunities for PBT. In Section 5 we discuss our plans for follow-on work which will aim to capitalize on these opportunities. But before doing so, we must confirm and refine the findings of the preliminary study, as well as identify barriers to adoption that we may have missed: this requires a larger scale study.

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 in industrial settings and to begin to explore potential solutions.

4.1 Research Questions

The research questions for the full-scale study will again address our central question: How can the research community make PBT more valuable for software developers?

Our preliminary study highlighted five sorts of challenges that PBT faces: what might be called *property*, *generator*, *workflow*, *learning*, and *social* challenges. We expect that learning challenges are best explored in a classroom setting (see Section 5), and that social challenges are too intertwined with challenges of other sorts to study directly. Accordingly, our research questions for the full-scale study will focus on property, generator, and workflow challenges.

RQ1. What support do developers need to help them imagine properties?
RQ2. What kinds of generators do testers need to exercise their properties effectively? Do they have specific precondition and/or distributional requirements?
RQ3. What aspects of the developer workflow around PBT need improvement?
RQ4. What concrete changes could be made to modern PBT systems to improve effectiveness and usability?

RQ1, **RQ2**, and **RQ3** relate to property, generator, and workflow challenges, respectively. **RQ4** more directly asks how existing systems might be improved. This final question will help to keep us focused, reminding us that participants likely have the context and knowledge not only to identify challenges but also to suggest solutions!

4.2 Study Population

As in the preliminary study, our primary tool will be interviews with developers. We will partner with Jane Street, LLC, a large financial technology firm, interviewing around 30 Jane Street employees about their experiences using PBT.

Jane Street has a number of qualities that makes it an ideal place for this study. To start, Jane Street developers use a variety of testing tools, including PBT. This gives us a place to start when asking questions, since participants will likely have seen PBT before, and it means that will be score for exploring trade-offs between PBT and other forms of testing. Additionally, Jane Street developers famously build almost all of their systems in OCaml, a mostly functional programming language with strong support for static typing and modularity. This unified ecosystem 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 PBT tools and the same libraries, language-level programming abstractions, house coding rules, etc. (which might make testing easier or harder).

Naturally, carrying out a study at a single firm has potential drawbacks as well. The most obvious is that our results may not generalize: We cannot guarantee that our findings will apply outside of the OCaml ecosystem (and other ecosystems like it). However, Jane Street is home to a diverse array of software systems, including 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 developers 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 two distinct populations at Jane Street that we can learn from. The group we had initially planned to talk to was developers who have used PBT in their work at the firm. These can tell us about how PBT helped them, what challenges they faced, and what techniques they use instead of PBT to check that their code is correct. But we 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 themselves. Since talking with stakeholders may also help us to refine our developer interview script, we will interview them first.

4.3 Interview Plan

Each interview will be allotted a one 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 opportunities to support users of PBT, phrasing our prompts to encourage creative thinking.

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 refine our understanding of the challenges facing PBT. We were able to confirm long-held impressions of the challenging aspects of PBT, including those pertaining to imagining properties and writing generators, and we were also able to highlight challenges in previously unexplored parts of the process like the developer workflow. 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 on the results of the full-scale study, our preliminary study suggests a few directions that may make sense.

Observations of Practice and Surveys. The data we collect from interviews should reflect developers' *impressions* of their work, but it may miss important details that were simply not memorable. Accordingly, we hope to follow up with one or more *behavioral studies*. Such a study might include observing developers completing a pre-determined task or carrying out more realistic contextual inquiries [Beyer and Holtzblatt 1999]. Both of these approaches would give us the opportunity to see a more unfiltered view of the challenges that developers face with PBT. We may also carry out a larger-scale survey, with the goal of establishing quantitative results to support the quantitative ones from our interview studies.

Education. Our preliminary study reminded us that PBT builds on concepts that are not always comfortable for developers. Prior work has explores ways to close this knowledge gap [Nelson et al. 2021; Wrenn et al. 2021]; we expect there are further education challenges that are worth exploring. We plan to investigate these by incorporating PBT into a required course at [a large university].

Concrete Tools. In the course of our interviews, it may become clear that the solutions to some of the challenges facing PBT are within easy reach. For example, we hope some of the workflow challenges mentioned in Section 3 can be addressed via standard HCI methods and that some of the generator challenges are amenable to known PL techniques.

Indeed, while the bulk of this report has been about need-finding, we do not intend to to stop there. Rather, we plan to take the insights gained through interviews (and observations, surveys, etc.) and apply them to build compelling tools for real-world 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

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

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).

Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.