Project Description

Testing plays a vital role in the robustness, security, and overall quality of modern software. It comes in many styles—unit testing, integration testing, performance testing, stress testing, accessibility testing, penetration testing, etc.—supported by diverse tools, with yet more advanced tools and methodologies always on the horizon.

One such methodology is *property-based testing* (PBT), sometimes described as "formal specification without formal verification." With PBT, a developer characterizes the desired behavior of some piece of code in the form of executable *properties*. The code is then validated against these properties by running it many times with a large number of automatically generated test inputs. This combination of rich, high-level specifications and mostly automatic validation has proven effective at identifying subtle bugs in a wide variety of settings, including telecommunications software [4], replicated file and key-value stores [60, 11], automotive software [5], and other complex systems [59]. It is used by companies including Amazon, Volvo, Stripe, Galois, and IOG, which runs the Cardano blockchain and the Ada cryptocurrency.

PBT took the functional programming world by storm following its debut in the Haskell QuickCheck framework [18]. In 2010 the QuickCheck authors received the ACM SIGPLAN "Most Influential Paper of ICFP 2000" for their paper on it; this is currently the most cited ICFP paper by a factor of 2, according to ACM's digital library. In the intervening years, PBT has spread to many other software ecosystems: Wikipedia lists QuickCheck variants in 40 languages, some with several competing frameworks [104] (Java alone has 7!). And these frameworks are popular: the developers of the Python Hypothesis framework [85, 61] estimate its user community at half a million [26, 103]. On GitHub, Hypothesis has 6.5K "stars" from developers (indicating enthusiasm) [10], Rust's quickcheck [109] has 2K, ScalaCheck [110] has 1.8K, and Clojure's test.check [20] has 1.1K. By comparison, pytest, the main Python framework for running tests, has 9.6K stars [102]—i.e., Python's PBT tool has ∼70% as many stars as Python's entire testing infrastructure.

And there is still plenty of room for growth. The 500K estimate for Hypothesis users is only 4% of all Python developers; the Hypothesis authors estimate that the "addressable market" for PBT is around 25% of the Python community, and that there remains significant room for improving its usage by existing users [26, 103]. Similarly, the list of companies using PBT, while substantial, is very far from the whole of the software industry. These gaps represent a gigantic opportunity to increase software quality and reduce software costs. A 2002 study [107] estimated that the total cost of software errors is almost $60 billion per year and suggested that $22 billion of that could be saved through better testing infrastructure; since then, the situation has only gotten worse, with a 2022 study [69] estimating that poor software quality now costs the U.S. over $2 trillion per year. Accelerating the adoption of PBT thus stands to make a significant dent in the global cost of software bugs. This is the grand challenge that we address.

Our group has already begun working to identify high-leverage ways to extend PBT's reach. In an ongoing need-finding study with PBT users at Jane Street Capital, we found consistent enthusiasm—developers called it "obviously valuable" (Participant P1), built their own frameworks for it when standard ones were not available (P8, P21), and suggested that "everyone" at the company should use it (P20); however, developers also highlighted a key opportunity for improvement—*usability*. Like many powerful tools, PBT can be difficult to apply, and developers need support to do so effectively.

The research advances required to improve the usability of PBT will require fundamental insights from both the programming languages (PL) and human-computer interaction (HCI), communities. On one side, PL provides conceptual background, mathematical underpinnings, and established tools for PBT. On the other, HCI provides a deep foundation of theory and practice for evaluating usability of systems in a rigorous and objective way, including principled methods for identifying problems, rigorous metrics, and proven approaches to tool design—where tools, here, naturally include "front end" components like data visualizers and IDE plug-ins, but also "back end" technologies like domain-specific languages for properties and generators; for the latter, HCI techniques can help designers strike the right balance between expressiveness and accessibility [21, 38]. Synergies like these led Chasins et al. [17] to argue that a research methodology combining PL and HCI hits a "sweet spot" where need-finding techniques identify pain points, motivating concrete tools that help programmers write safe, correct code.

Our team is uniquely positioned to bring PBT into this vibrant area of "PL+HCI" research. PI Head recently co-founded a new HCI group at the University of Pennsylvania and specializes in interactive

programming environments, while PI Pierce has published widely on PL topics including PBT. Our past research collaboration has led to two workshop presentations [35, 111] and to the ongoing study at Jane Street mentioned above, which informs the research and technology transfer agenda for this proposal.

We propose an interdisciplinary program of research and engineering in property-based testing, bringing to bear the combined power of PL and HCI to accelerate PBT's transition into practice. Planned research, engineering, and education activities can be grouped into five main themes (see Figure 1):

1. We will establish a solid ***foundation*** for HCI-informed research on PBT, building on our ongoing need-finding study. We will confirm and generalize this study's findings through further studies, culminating in a novel cognitive theory of PBT (§3).
2. We will explore and apply a novel "reflective" approach to ***generation***, enabling better generator tuning and counterexample shrinking and linking PBT with fuzzing. We will automate construction of reflective generators, and we will construct a platform for benchmarking these and other advances in tools for generation (§4).
3. We will empower developers to ***specify*** properties with new languages for expressing properties, tools that simplify authoring properties, and assistants that help explain properties to others (§5).
4. We will design, prototype, and evaluate novel tools for ***interacting*** with properties and generators, leveraging our advances in specification and generation to enable new ways of visualizing and understanding random distributions over test inputs and pinpointing the cause of test failures. (§6).
5. We will support the ***diffusion*** of PBT tools and methodologies from academia into industry through targeted engineering—in particular, supporting and improving open-source PBT frameworks and building a comprehensive IDE for PBT, dubbed TYCHE. We will also drive diffusion via education, developing materials for teaching software developers about high-value applications of PBT and for introducing PBT into undergraduate and masters-level computer science courses (§7).

The scale of this project—two PIs, four PhD students, and a staff engineer—is essential to the success of (1) its tightly integrated research agenda, which requires expertise across PL and HCI to make significant advances in the five interconnected themes above, and (2) its goal of broad impact on industrial practice, requiring the full-time attention of a software engineer. A detailed sketch of responsibilities and timeline can be found in the Management and Coordination supplement, but, briefly, one PhD student will build on our existing user studies to develop a clear foundational understanding of PBT from the perspective of usability and human factors, two students will develop "back end" technologies to support usable PBT workflows, and a final student will leverage these technological advances to design and develop "front end" IDE support. The research engineer will focus on transferring the products of this research into an industrial context, supporting and enriching existing open-source PBT projects with theoretically informed ideas and tools. None of these projects stands on its own; rather, each supports and informs the others to achieve both conceptual advances and significant impact on software development practice. Publication targets and other success metrics are discussed in each section below.

Two PIs is on the small side for a Large NSF proposal, but we feel this is the right size, for two reasons. First, the PIs come from quite distinct areas, so we already span a broad swath of CS in terms of perspective and expertise. Second, our success will depend on a high degree of interaction and cross-fertilization between PL and HCI; this is much easier with fewer PIs.

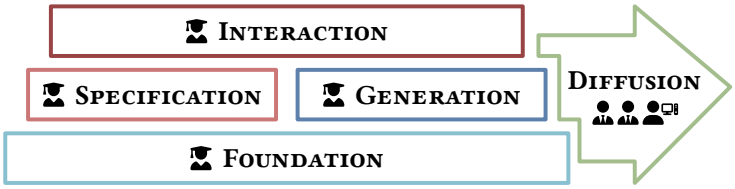

Figure 1: Overview of the proposed research. Four PhD students will advance the state of the art on four interconnected axes, supported by a research engineer and the PIs for maximum impact.

Our plans for Broadening Participation in Computing, described in a supplemental document, are focused in two areas: (1) expanding an existing NSF-REU program that brings undergraduates from underrepresented groups to Penn for summer research experiences in programming languages, and (2) increasing diversity in the TA roster for Penn's introductory computer science course, which PI Pierce co-teaches.

The rest of this Project Description describes this research and technology transfer agenda in detail. Sections §1 and §2 supply background on PBT and present preliminary findings from our ongoing study at Jane Street. Sections §3 through §7 outline plans for each of the themes listed above. Section §8 discusses the Broader Impacts of the project, and Section §9 summarizes our prior NSF-supported work.

1 Orientation: Property-Based Testing

PBT [58] is software testing method where executable functions are used as partial specifications of a component under test. For example, a developer might write the following property for an `insert` function on binary search trees, taking an arbitrary tree `t` and an integer `x` as parameters:

```
prop_insert_correct x t  =  (is_bst t ==> is_bst (insert x t))
```

That is, if the original tree is a BST, then it should remain a BST after the insertion of `x`, where `is_bst` is a function that checks whether a binary tree is arranged so that each node's label is greater than any label in its left subtree and less than any in its right subtree. In general, such a property is a function that accepts a generated test input and evaluates to `True` if the test passes and `False` otherwise. Given a property, the PBT tool generates a large number of inputs and checks that the property yields `True` for each one; any input that causes the property to fail is reported to the user as a counterexample. Designing these properties is an example of the *oracle problem* [6], which arises in any kind of automated testing where the user needs to define what it means for a program to be correct.

Our research agenda focuses mostly on *random* generation [40], the dominant approach in PBT, though many of our tools would also be applicable to alternatives like enumerative generation [108, 14]. The surprising effectiveness of random generation can be attributed to the "combinatorial nature" of large test cases—the fact that bugs can often be exposed by any test input that embodies some specific combination of features, independent of whatever other features may also be present. For example, a bug might be triggered by a particular sequence of API calls in a particular order, even when these are interleaved with other API calls. As a result, testing with large random inputs often exposes issues much faster than exhaustively enumerating small inputs. Techniques like swarm testing [39] can further amplify this effect.

To apply PBT to a system or an individual software component, the developer first defines one or more properties that they expect should always be satisfied. Then they supply *random input generators* for the values that the properties take as input—these are sometimes written by hand, but often they can be automatically generated, e.g., from the type of the input. Next, they check their properties against many generated inputs, using a test harness provided by their PBT tool. And finally, if counterexamples are discovered, they inspect them to determine the source of the bug. Each of these steps can be significantly improved for users, as we describe in the next section.

Why go to the trouble of PBT, rather than the more straightforward example-based testing that is standard across the software industry? First and foremost because a component can be tested much more thoroughly with a property plus many automatically generated examples than with a small number of examples written out by hand. But PBT is more than just thorough—it is also more general than example-based testing. For example, Wrenn et al. [127] observe that example-based testing of programs whose correctness conditions are *relational* (e.g., topologically sorting a graph, which might produce any of a number of correct results) is impossible to do faithfully; a property-based specification is a better choice in such cases. PBT is also an obvious choice if the developer already has some semi-formal specification in mind—for example if they are implementing behavior from an RFC or other design document—because it provides a clear connection between the specified behavior and the implementation. Finally, the properties required for PBT can also serve as documentation: participants in our need-finding study (P5, P21) talked at length about properties being an ideal way to communicate what a program is supposed to do.

PBT is also often compared to *fuzz testing* [129], which randomly tests software to find vulnerabilities. We discuss ideas for bringing PBT and fuzzing closer together later in the proposal (§4.2), but current fuzzers have fundamentally different goals from PBT. In general, fuzzers need to run for a long time (hours or days), they are commonly used to test whole systems "from the outside" by provoking crashes. Properties in PBT, on the other hand, can typically be checked more quickly (on the order of seconds), they express richer constraints on behavior than "does not crash," and they can be used to test both whole

systems and smaller components. Both techniques are useful, but they are applied in different ways, at different points in the development process, to achieve different goals.

With all these advantages, one might hope to find PBT on every software developer's toolbelt. But PBT poses some challenges as well, as we shall see next.

2 Motivation: A Formative Study of PBT in Industry

Our research agenda is strongly informed by preliminary findings from an in-progress need-finding study at Jane Street Capital. Our purpose in this study is to understand the usability challenges that must be addressed to boost adoption of PBT in industry. The study data consists of thirty semi-structured interviews with (1) developers who use PBT and (2) maintainers of PBT tools.

Our choice of Jane Street as a partner was motivated by our desire to understand PBT in a non-academic setting where it is broadly appreciated. At Jane Street, PBT is an established methodology, so there is a large population of people with well-informed opinions on its benefits and challenges. Additionally, Jane Street builds much of its software in OCaml, a functional language with a well-engineered PBT framework. This unified ecosystem means that developers have access to mature PBT tools, experience using them in collaborative settings, and awareness of language-level abstractions necessary for expert usage of the tools. All this makes Jane Street an ideal place for understanding the impact—and challenges—of PBT when users who have incentive to use it to full potential are provided with state-of-the-art tools. (Of course, the findings of any need-finding study are necessarily limited to the setting in which it is carried out; the understanding gleaned from this one will be deepened by further research activities described in §3.)

As of February 2023, the full complement of thirty interviews has been completed at Jane Street and a preliminary round of qualitative analysis is underway; full-scale analysis will begin later in the Spring. Findings from the study will be disseminated in a submission to a software engineering conference such as ICSE. We also carried out a smaller pilot study among Hypothesis users to prepare for the full-scale study at Jane Street; its results were presented at the 2022 HATRA workshop [35].

The final product from the ongoing study will be a fine-grained, qualitative description of how Jane Street developers use PBT, what they need from it, and how the research community can help improve it. While the full analysis of the interview data remains to be completed, a number of themes are already clear; these form the backbone of the present proposal. We describe them below, italicizing themes and referring to evidence from participants in the Jane Street study (P1–30) and the pilot study (Pilot-P1–8).

One set of themes concerned the ***generation*** of random inputs for PBT. Developers spoke highly of the *Derived Generators* that can be automatically inferred from the OCaml type system (P5 called OCaml's implementation of this "[expletive] amazing" and P30 called them a "game changer"). These generators are already quite good, but they could be better: participants identified deficiencies both small and large, the most significant being that derived generators cannot enforce semantic preconditions like `is_bst`.

When derived generators failed, participants fell back to *Bespoke Generators*, which are far more flexible but proportionally more time-consuming to build. For example, P20 successfully used a bespoke generator for XML documents to find significant bugs in their code, but reported spending "at least a day" writing it. Improving the abstractions available for authoring bespoke generators would greatly improve the usability of PBT. When a generated input turns out to be a counterexample that triggers a property violation, the developer will need to inspect that counterexample to find and fix the root problem. Developers often implemented code for *Shrinking* counterexamples to discover simpler inputs that trigger the same bug. P8 and P21, who each implemented their own PBT frameworks, both incorporated shrinkers as key components. But shrinkers need to be customized to particular kinds of data to be most effective, and they can be time-consuming to build; several developers (P16, P20 P21, P30) described constructing shrinkers as an opaque and difficult process.

A different set of themes themes revolves around properties themselves, i.e., ***specifications***. PBT is often described as a lightweight formal method, and one might therefore imagine that a common challenge would be coming up with the specifications of desired program behavior. Indeed, in our earlier pilot study [35], some respondents indicated just that: developers with less experience with PBT sometimes struggled to *Imagine Properties* or to understand what properties to test (Pilot-P1, Pilot-P3–5). By contrast, Jane Street developers on the whole reported little difficulty finding properties. Rather, most developers

applied PBT in *High-Leverage Scenarios* where properties were already available or straightforward to invent. In the words of P9, PBT is particularly easy to apply when one has "a really good abstraction with a complicated implementation." When asked to speculate, several participants (P3, P15, P20, P22) guessed that 80–100% of Jane Street developers write programs like this, where properties are easy to find and PBT is relatively easy to apply. This suggests that an effective way to boost PBT in industry would be to provide educational materials and documentation that highlight real-world applications where it is a natural fit.

We also heard *Opportunities for Better Leverage* of specifications, where PBT is not easy *yet* but could be with a bit more research effort. For example, developers in both studies (Pilot-P4–6 and Jane Street P7) complained that PBT was difficult when code was poorly abstracted. Further, more than three quarters of study participants had used a particular approach to PBT commonly called *Model-Based Testing*. P3, an author of PBT tools at Jane Street, considered better automation and tooling around model-based testing to be one of the most significant ways to improve PBT usability.

A further set of themes concerned the ***interaction*** between developers and their testing environments—especially their strategies for *Evaluating the Effectiveness* of their tests. Many wished for better ways to evaluate their generators and properties, including feedback on code coverage (P9 and P25), mutation testing [97], and help understanding the distribution of randomly generated inputs (P10, P16, P16). Worryingly, while many developers admitted they would benefit from better ways to evaluate their tests, many seemed to *Implicitly Trust the Infrastructure* that they did use. P14 actually shipped broken code because they did not realize their generator had missed important cases. Additionally, one participant (P14) saw significant benefit from *Visualizations* they had built themselves to understand their testing effectiveness.

Finally, our experience with these studies (and with the products of our own prior research!) suggests that significant engineering and pedagogical effort is needed to amplify PBT's ***diffusion*** into the broader community. Developers in both studies (Pilot-P1, Pilot-P4, JS P3 and P11) reported a dearth of *Documentation and Examples* for learning about PBT. PBT is also not taught in traditional computer science curricula at the undergraduate or masters level; making more developers aware of it will required expanded *Classroom Education*. Finally, existing tools for PBT need continual support to meet demands from growing (and increasingly sophisticated) user-bases; if PBT is to become mainstream, we should look for high-value ways to improve *Open-Source PBT Frameworks* with the insights and tools we develop, to help our work and ideas permeate the PBT world.

3 Foundation: Understanding Needs and Opportunities

The ultimate findings from the Jane Street study should give us a clear picture of the benefits and challenges of PBT in the specific context of Jane Street and other organizations with similar characteristics. But to fully understand the potential impact of PBT across the whole software industry—as well as the factors that may limit its adoption—we need to cast a wider net. In this section, we describe four planned studies that aim to produce a comprehensive, actionable *foundation* for the latter stages of this project and beyond. Building a rigorous foundation, with methods motivated by the HCI literature, is hard work, but it will significantly increase the chance that future projects achieve their goals. We plan two written surveys, one to assess the generality of these needs and obstacles and one to identify potential for adoption of PBT tools (§3.1) and two user studies, a design probe using a minimal PBT framework and an observation study to guide the design of interactive tools (§3.2). Finally, we plan to distill these study findings into a cognitive theory of property-based testing, a conceptual foundation for the design activities elsewhere in the project (§3.3). The projects in this section will culminate in an article for the Communications of the Association for Computing Machinery (CACM). To ensure fairness, ethics, accountability, and transparency (FEAT), each of our user studies will be vetted through Penn's Institutional Review Board (IRB) review process.

3.1 Generalizability of the Jane Street findings. Preliminary findings from the Jane Street study have already revealed a number of opportunities to improve property-based testing. To identify others and better understand which are most critical, we will conduct two surveys with broader samples of developers. These surveys aim to (1) determine which obstacles observed in the original study represent widely experienced pain points and (2) understand the potential benefits of better tools for the software industry.

The main survey aims to confirm (or perhaps refute) that the things we are learning from Jane Street generalize to other settings. We will ask developers which of the issues we found at Jane Street are ones they have also encountered, which are most severe in their experience, and which other issues they have encountered. To provide clear usage scenarios, respondents will be asked to write brief anecdotes elaborating on the most severe issues they remember. Other questions will assess how heavily respondents depend on specific features of their PBT tools that may be enabled by their ambient programming environment and language, e.g., whether their language supports Haskell-like typeclasses or OCaml-style metaprogramming, both of which are used to good effect by PBT tools in those languages. In line with typical practice in human factors research surveys in software engineering [106, 121, 89], we aim to recruit respondents with upwards of around 150 respondents per survey (and aspirationally many more). This is a significant increase in scale from the original study, requiring energy and care commensurate with that increase, but it will give us a critical opportunity to galvanize our findings. Respondents will be recruited broadly, from groups including: (1) users of the major PBT frameworks in Python, Java, Haskell, Rust, and Scala; (2) the students' and PIs' professional networks, including Twitter and Mastodon, various mailing lists, and discussion boards for developer conferences—e.g., "Yow!", where PI Pierce spoke last year [100]; and (3) Jane Street, aiming for a broader set of developers than in the original study.

A second written survey later in the project will investigate how broadly PBT may *eventually* be able to reach. To get a sense of this, we will survey "proximal" users of PBT—developers who do not use PBT currently but who might find it particularly useful. We will again recruit a broad sample of participants from varied settings (professional, open source, educational) by working with our industry contacts and recruiting over social media.

Publication target: International Conference on Software Engineering (ICSE).

3.2 PBT interaction models. Interviews and surveys are adequate for exploring usability challenges at a high level, but in order to build usable tools we also need to study the low-level interactions between users and their PBT tooling. We will conduct targeted observational studies to inform the many fine-grained design decisions involved in language design (e.g., §4.3) and interface design (§6) activities.

Understanding user interactions is a core activity of HCI research and helps to inform robust interface design. Observation studies often require significant effort to plan and perform. That said, they are indispensable for answering questions about tool design may that receive only speculative answers in interviews and surveys, like: (1) How much time are participants willing to devote to creating a property or tuning a generator when in the middle of a programming task? (2) How much space is available on a developer's screen (amidst other tools like code editors and terminals) for new live visualizations? or (3) What are developers' current strategies for solving the problems they describe, e.g., what representations seem most helpful for a programmer trying to understand the distribution of data from a generator?

We will conduct two kinds of studies. The first will be observational studies in the style of contextual inquiries [56, Chapter 3], where developers perform PBT in realistic contexts, with their own tools and tasks. The output of these studies will be detailed understanding of the task structures involved with specific PBT tasks (e.g., writing a generator, assessing test results) and, most importantly, what steps in those tasks are particularly time-consuming and effortful. Recordings and notes from these sessions will serve as metrics against which the likelihood of success of particular designs can be measured before implementation.

The second kind of study will be controlled comparisons of design alternatives. Language design, like what we plan relating to generators (§4.3, §4.4), is famously difficult to inform empirically because there are many design choices that interact in complex ways. We plan to identify the most significant patterns of variation between modern PBT tools (e.g., defining properties as simple functions vs. via DSLs, or using types to steer generation vs. fully manual generators), and perform a small number of comparative studies to gain clarity on how various choices would impact our design. Following best practices in experiment design for programming tools [66], we will keep the experimental tasks simple, train participants to control for variance in prior skill, and triangulate findings from a mixture of quantitative metrics (e.g., success, task duration, error rate, subjective reports of difficulty) and qualitative analysis (e.g., thematic analysis [9] to reveal the kinds and frequency of problematic incidents). Most of these findings will be in service of research projects described elsewhere in this proposal, but they may also be disseminated as publications if they turn out to be of broader interest.

Publication target: PLATEAU Workshop.

3.3 A cognitive theory of PBT. Theories of programming have long served a critical role in human-centered software engineering research, providing rich and memorable descriptions of programming processes and clarifying opportunities to improve tooling and training. For instance, Lawrance et al. [76] advocate for the use of *information foraging theory* [101] to describe how programmers navigate complex code bases to search for behaviors within that code. This theory has motivated a variety of tools that demonstrably improve the programming experience (e.g., [50]). Other theories that have inspired advances in programming tools include Blackwell's Attention-Investment model [8] Ko and Myers' debugging framework [67], and Ko's characterization of end-user programming barriers [68].

Recognizing the power of theory, one of the first steps in our research will be to develop a cognitive theory of PBT. The starting point will be the conceptual framework we are developing through the Jane Street study. This will be validated and refined with evidence from early observation studies we described in §3.2, leading to a provisional theory in the first two years. This theory will help us refine our ideas for tools and provide a foundation for contextualizing the questionnaires and observations described in the following sections. Conversely, evaluations of the tools we build will provide additional evidence to further refine the theory.

A successful outcome of this effort will be a theory that is simple, evocative, and validated. The theory will define the constructs involved in PBT, including the tasks involved in PBT (e.g., generation, property specification, and reviewing test output) and a developer's goals (e.g., developer time, test speed, and level of assurance). It will explain tensions between goals (e.g., more assurance often means more developer time). Furthermore, it will identify how developers make choices to achieve their goals. For instance, our Jane Street interviews showed us that some developers' testing activity might better be described as satisficing [15] rather than maximizing assurance of a system: these developers limited the time they spent writing and running tests, although more testing effort might well have revealed further bugs or achieved better coverage. A theory that brings these elements together would suggest that for such developers to achieve higher assurance, they would evidence that better assurance is possible with only modest effort.

Publication target: Foundations of Software Engineering (FSE).

The projects in this section will develop an evidence-based model for PBT that will be useful far beyond our research. Therefore we also plan to publish an article in the Communications of the Association for Computing Machinery (CACM) to communicate about PBT to a broader computer science audience.

4 Generation: Better Tools for Random Inputs

Many of the usability improvements suggested by the Jane Street study center around test input *generation*. Indeed, generators were regularly cited as one of the most challenging aspects of PBT: the existing tools related to random generation are varied and powerful, but they are not especially usable. We propose *reflective generators*, an abstraction for random generation that we have been experimenting with at Penn, as a way of unbundling generator functionality, exposing levers for automation that enable a number of new approaches to generator tooling (§4.1). We explore how reflective generators can help to connect PBT and fuzzing (§4.2) and design a user study for evaluating and improving the usability of the reflective generator language (§4.3). Finally, we discuss plans to boost generator automation (§4.4) and metrics by which to evaluate improvements in generation (§4.5). Success for these enabling technologies will be measured both by the success of the tools built on them in §6 and §7 and by top-tier publications, including performance evaluations and/or user studies as appropriate. PI Pierce is a longtime collaborator of John Hughes, one of the QuickCheck creators; John plans to collaborate on the projects in this section; a letter of collaboration appears in the supplemental documents.

Context: Why Random Generation is Hard. As discussed in the Orientation section (§1), PBT relies heavily on *random data generators*. Checking a property for many randomly generated inputs gives confidence that it holds—provided the inputs actually trigger a wide variety of program behaviors. Unfortunately, many properties that developers want to test have *preconditions* (a.k.a. *validity conditions* or *input constraints*) that restrict the set of inputs that can be used for testing. This comes up often when testing

data structures with invariants that must hold in order to apply the operations being tested. Testing such properties can be problematic, since many preconditions are difficult to satisfy randomly; if the developer is not careful, they may waste most of their time budget generating and discarding precondition-failing inputs. (Of course, some fraction of the testing budget *should* be spent on ill-formed or nonsensical inputs, but not too much, since these will not exercise much of the system's functionality; most tests should be well-formed—though perhaps unusual—instances of the sorts of inputs the system is designed to process.) Worse, even with validity accounted for, there are more insidious ways for generators to under-perform—for example, by generating many similar inputs while ignoring large parts of the input space.

Existing approaches to these issues fall on a spectrum from automatic to manual. The automatic approaches use various proxies for validity and general "interestingness" of inputs: some, like *fuzzers* [129], try to maximize readily available metrics like code coverage, while others ask users to provide their own metrics [83], and naturally some use machine learning to infer proxies for validity [34, 105]. These approaches are easy to apply and produce diverse sets of inputs, but they are rarely sufficient for testing properties with complex preconditions. Slightly more manual approaches are based on declarative representations of validity conditions: for preconditions that are primarily structural, *grammar-based fuzzing* provides a compelling solution [33, 54, 122, 123, 114], and for more complex, semantic preconditions, SMT-solvers [25, 71, 116] can be used to automatically seek out valid inputs. These tools are much better at satisfying easy to moderately complex properties but much worse at very complex or "sparse" properties. The semi-automatic category also includes tools for *example-based tuning*, a process that improves realism of inputs by mimicking user-provided examples [113]; these tools can generate realistic inputs, but they are again limited in the preconditions they can satisfy.

The most manual—and most flexible—solutions use hand-built generators, written in a custom-designed domain-specific language (DSL). In Haskell, where PBT was first popularized, such DSLs are commonly implemented using *monads* [88], an elegant design pattern for expressing effectful (here, random and stateful) computations in a pure, stateless underlying language. While monadic DSLs are not needed to express generators in impure languages, some frameworks (e.g., in OCaml) still choose to use monadic abstractions for their generator DSLs.

Monadic generators can implement random data producers of arbitrary complexity (e.g., for random Haskell programs [99]); in this sense, they are strictly more expressive than representations like grammar-based generators. Yet monadic generators are syntactically constrained in a way that isolates the probabilistic code and prevents usage errors (like passing the wrong random seed around).

To further improve monadic generators, it helps to re-frame generators as *parsers of random choices*. The usual intuition is that a generator operates by making a series of random choices; equivalently, we can think of it as being *given* some random sequence of choices and simply following those choices to produce a value. This shift of perspective has been used as the basis for implementations of PBT tools [85, 28], but we were the first to make it formal using *free generators* in our paper, *Parsing Randomness* [37].

Context: Reflective Generators. Building on the free generator ideas described above, we are exploring a powerful generalization based on monadic generators called *reflective generators*. We began thinking about the basic concept during a past NSF project (see §9), and we will submit a first paper to ICFP 2023.

What's special about reflective generators is that they can be run *backward*. If running a generator in the usual way can be seen as parsing a sequence of choices into a value, then running it backward should take that value and produce a sequence of choices that would generate it—i.e., reflective generators can "reflect" on a given test and analyze the choices that they could have made to generate it. The mathematical machinery that makes reflective generators work is somewhat complex,[1] but, like free generators, the syntax that users see remains close to that of normal monadic generators.

Running a reflective generator backward is not simply a matter of remembering the choices it made going forwards and replaying them in reverse order (as some PBT frameworks already do [85, 41]). For

[1]Reflective generators are both monads and *partial profunctors*, implementing bidirectional programming in the style of Xia et al. [128]. This approach to bidirectional programming is related to lenses [31], but it hides much of the complexity of bidirectional program composition in the bind operation of the monad. The result is an elegant programming experience where both directions of the computation can be written at once, in a type-safe way.

one thing, a reflective generator can reflect on inputs it did not actually produce—all that's required is that it *could* have produced them. For another, the choices can be structured in different ways (as bit strings, higher-level choice sequences, choice trees, etc.) if increased structure reveals information that an analysis algorithm (like the ones discussed below) can use.

Reflective generators have myriad uses. Here are a few. **Example-Based Tuning.** It is often helpful to start by testing with "realistic" inputs that trigger common-case behavior in the program; one way to ensure this is to tune the generator so it produces values that are similar to some user-supplied values deemed realistic. Existing tools make good use of this example-based approach to tuning [113], but they do not work with generators as powerful as monadic generators. We implement a similar algorithm using reflective generators; using it, we can (1) reflect on realistic values to obtain sets of choices, and (2) run the generator with *new choice weights* informed by the choices that we saw. Our evaluation shows that reflective generators approximate the performance of the algorithm from [113] but work with a larger class of generators. **Reflective shrinkers.** Reflective generators can also be used as a tool for analyzing and manipulating the structure of generated inputs. Inspired by the test-case reduction algorithms in Hypothesis [84], we implemented validity-preserving *shrinking* of values to find smaller counterexamples and speed up debugging with no additional effort from the user. Hypothesis shrinks test inputs by shrinking the random choices (conceptually, coin flips) that produce those inputs. There are many benefits to this approach: shrinking can be implemented once-and-for-all, and it can leverage the generator code to ensure that shrunk inputs remain valid with respect to property preconditions. But shrinking the random choices requires that the choices is actually available, which is not always the case. Reflective generators can be used to recover the random choices and thus enable the Hypothesis shrinking algorithm on any valid input (not just those that we know the choices for). We can, for example, automatically shrink inputs provided by the developer or gleaned from bug reports. **Other data producers.** Reflective generators can be freely "reinterpreted": the same code that specifies random choices can also be used to *enumerate choices* or make *dynamically guided choices*. This means that as new strategies for guided random generation and enumeration become available, they can be used to improve reflective generators.

4.1 Theory of reflective generators. The work on reflective generators has begun to produce research results, but the theory around reflective generators is still far from settled. In particular, there are open questions about exactly which primitives are ideal for the generator language. The iteration in our current paper seems to give users too much power, potentially hurting usability (see §4.3), but limiting that power may have unanticipated technical implications. We will explore a variety of language formulations in the context of examples to find an appropriate balance. Additionally, reflective generators are currently only ergonomic in purely functional languages; we will build on prior work [12, 24] to explore a form for reflective generators that is appropriate in languages like Python, Java, and Rust.

Regarding shrinking, the shrinking algorithm used in Hypothesis is powerful, but not obviously optimal. Hypothesis shrinkers reduce the input randomness to the *shortlex smallest* choice sequence—that is, they favor sequences that make the generator make as few choices as possible, and where each choice is "minimal." This is a useful heuristic, but it is not directly related to the user's understanding of the size of the shrunk inputs. Ideally, we'd like some reflective shrinkers to *always* produce inputs that are smaller by some natural metric on their type, not on the random choices that produced them; we will state and properties of a generator that imply this property.

Finally, there we will explore the relationship between reflective generators, which are implemented in terms of monads, and *algebraic effects*. Algebraic effects may make it possible to simplify the representation of reflective generators, making them more modular and composable.

Beyond theory, reflective generators offer a powerful foundation enabling a range of more practical innovations. We discuss these in the remainder of the section.

Publication target: International Conference on Functional Programming (ICFP).

4.2 Fuzzing with reflective generators. *Fuzzers* like AFL [129] are based on principles similar to the ones behind PBT: in particular, they use randomization to exercise a range of program behaviors. Fuzzers are popular because they are both effective and inherently easy to use: the developer need only point the fuzzer at an executable binary and wait. But, without help, fuzzers are not very good at finding bugs when certain program paths are hard to reach (e.g., if they require the program's input to satisfy some

complex precondition). Our ultimate goal is a unification of PBT and fuzzing that combines the powerful automation potential of reflective generators with the usability of fuzzers.

Some existing projects have already worked to combine PBT and fuzzing. For example, the FuzzChick framework in Coq [72] uses code coverage as guidance for PBT, and HypoFuzz uses a similar approach in Python [41]. These projects are demonstrably powerful, but neither benefits from the years of expertise poured into industrial-strength fuzzers; Crowbar, on the other hand, does [28]. Crowbar re-interprets the output of AFL [129], one of the best-established fuzzers: instead of letting AFL generate the program input, it instead uses AFL to generate a sequence of choices that a generator then parses to get a program input. Reinterpreting the AFL output in this way does require the user to write a generator, which is more effort than is required for standard fuzzing techniques, but the result is a system that is much more likely to achieve thorough testing.

We admire Crowbar, and think the idea can be pushed even further by building a variant of Crowbar on top of reflective generators. The idea of crowbar is to start with a classic fuzzing setup, which tries to make the system under test crash by passing it a variety of semi-random inputs. But instead of the fuzzer "working against" the parser, in the sense that the parser's job is to reject invalid inputs and the fuzzer's job is to get past it, Crowbar's generators are monadic and can generate inputs that always pass the parser. Our generators will be monadic for the same reason but also *reflective*.

Why a reflective generator? To start, because its backward interpretation can be used to help seed the fuzzer. Modern fuzzers often ask the user for a number of *seeds*, input examples that the fuzzer can start from, to ensure that the fuzzer does not spend ages exploring inputs that have no hope of getting beyond the parser in the system under test and exercising other parts of the system. Normally these seeds are easy for the user to write down—they are simply example program inputs—but it is infeasible to ask the user to write down the sequences of choices that result in their seed values. This is one excellent use for a backward interpretation. The user can write down their seeds—either as values in the program, or as text that can be parsed by the program's parser—and ask the reflective generator to get the choices that lead to those seeds.

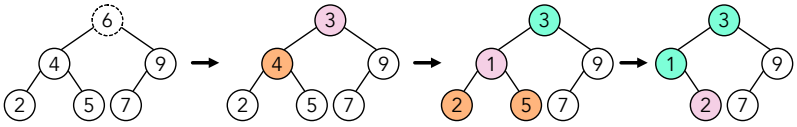

Figure 2: Validity-preserving mutation of a binary search tree, maintaining the BST invariant. Mutating the root node from 6 to 3 invalidates the 4 in the left-hand subtree; the generator 4 with a new random label, 1, then throws away its left subtree (because 1 is the minimum element of the label range) and relabels 5 to 2.

Reflective generators can also support validity-preserving *mutation*. Fuzzing algorithms operate by mutating "interesting" values so as to explore the behavior of the program in a space "around" those values. But mutation can be tricky in scenarios where values are subject to complex validity constraints, since purely random mutation often produces invalid values. Reflective generators can help with this by (1) reflecting on a particular value to obtain a sequence of choices, (2) mutate those choices, and (3) run the generator forward with the new choices, *correcting any that would lead to an invalid value on the fly.* Figure 2 illustrates how this algorithm can mutate a binary search tree while maintaining validity, using no bespoke BST-specific code.

Publication target: Programming Language Design and Implementation (PLDI).

4.3 Usability of reflective generators. Reflective generators are unavoidably a bit more complex than standard QuickCheck generators. The current design of the reflective generator language is a best guess at what users will find most usable (based on our own experience), but we can do better than guessing!

We will evaluate and redesign the surface syntax of reflective generators with the help of real users, in a style informed by prior work in the HCI literature [66]. We will recruit a small group of users, teach them to use reflective generators via a short introductory "README," and ask them to implement a number of PBT generators inspired by the research literature and real-world examples. We will both observe and interview the users to identify points of friction and ensure a clear understanding of their impressions. With this information in hand, we will identify potential changes to the API and test those changes with

a new group of users. The ultimate measure of success (again, evaluated through user interviews) is whether our language for reflective generators is usable by non-expert PBT practitioners.

Publication target: Human Aspects of Types and Reasoning Assistants (HATRA).

4.4 Generator automation. Even with an ergonomic and learnable language for reflective generators, the less code the user has to write the better. Both of our interview studies revealed that thinking about generators slows users down and makes PBT more challenging. But we should not compromise: any automation should help users obtain *high quality*, *reflective* generators for use throughout the testing process.

As a first step, we will adapt existing tools for type-based generator automation [87] to work with reflective generators. In general, type-based generators are only useful for testing properties with easily satisfiable preconditions, but this is a good starting point for many developers. Beyond that, we will consider techniques for *interactively* constructing reflective generators that maintain complex preconditions.

Some users may already have a standard QuickCheck-style generator that they would like to use in situations that require a reflective generator. We will assist those users with tools that automatically synthesize backward annotations required to make the generator bidirectional. We will experiment with both conflict-driven program synthesis [30] and solver-aided synthesis [120] to see which more successfully generates annotations for realistic generators. We plan to collaborate with professor Hila Peleg at the Technion, who has experience with both PBT and program synthesis. A letter of collaboration from Prof. Peleg appears in the supplemental documents.

Going one step further, we will explore techniques that can synthesize an entire reflective generator from whole cloth. There are a variety of attempts to do this for standard generators: Lampropoulos et al. provides two solutions, one based on on inductive relation specifications of preconditions [75] and the other based on an extended language for properties [71], while Steinhöfel et al. infer a generator from constraints using Z3 [116, 23]. But none of these tools fully solves the problem of automated constrained generation in general, and none of them include the tools necessary to produce reflective generators.

Pushing forward with generator automation could also extend the reach of PBT into new domains. Constrained generation is a major difficulty of the QuickChick ecosystem [98], and automated generator solutions could be used to test program logics [63, 79] to improve usability of PBT in contexts like proof assistants. More generally, generator automation could make PBT viable in any situation where properties already exist but preconditions are hard to satisfy.

Publication target: International Conference on Functional Programming (ICFP).

4.5 Generator benchmark suite. Most papers in the existing PBT literature use small case studies, showing that certain bugs in certain systems are caught more quickly with their tool than existing ones. For theoretically-oriented work, this may be sufficient to demonstrate the interest of an idea, but such evaluations are hard to interpret from the perspective of a would-be user. We can do better.

We have begun work on a robust empirical evaluation framework for test input generation strategies including infrastructure for easily and extensibly running experiments to understand testing effectiveness. By "easily," we mean that we will take on the burden of collecting data and analyzing the results. When complete, this evaluation framework will be able to evaluate a given tool based on (1) the degree to which it is able to achieve high code coverage quickly, and (2) the speed with which it finds bugs that have been pre-seeded in example programs. By "extensibly," we mean that in addition to the two languages (Haskell and OCaml/Coq), multiple frameworks (QuickCheck, SmallCheck, QuickChick, etc.), and numerous workloads that we currently support, we will design the infrastructure so that users can easily add new languages, frameworks, and workloads. Some initial comparisons using this framework will soon be submitted as an experience report to ICFP 2023, but many crucial features (e.g., coverage measurement) have not yet been implemented.

Our second contribution will codify a substantial library of case-studies and examples as *benchmarks for PBT*. Similar suites of benchmarks already exist in the fuzzing literature [42], but those benchmarks are not organized around the particular challenges that PBT users face. In particular, few of the benchmarks deal with the kinds of complex preconditions that PBT tools are built to handle. We want to establish a set of challenging tasks that can serve as a polestar for future improvements to PBT generators and bug-finding strategies (including our own!).

Past and proposed work on this topic are joint with Leonidas Lampropoulos and his group at Maryland. A letter of collaboration from Prof. Lampropoulos appears in the supplemental documents, and PI Pierce has a long history of successful projects with Prof. Lampropoulos [71, 36, 73, 75, 74, etc.].

Publication target: International Conference on Functional Programming (ICFP).

5 Specification: Widening the On-Ramp

To bring PBT to the people, we need to help developers with envisioning and writing specifications. Our the Jane Street study suggested that developers struggle to write properties about their programs, sometimes due to poorly abstracted code and other times simply because they fail to imagine what properties they might want to test. We propose to address the first of these issues with a property language that operates over program traces (§5.1) and the second with mixed-initiative interactions where the user and the computer collaborate to write a property (§5.2). Further, we will automate one specific high-leverage scenario, model-based testing (§5.3). Finally, we will explore how to generate automatic explanations of properties, for documentation and education (§5.4). Success in these tasks will again be measured indirectly, through the tools described in §6 and §7, and directly in terms of publications.

5.1 Properties over program traces. A common pain point for developers in our interview studies was that their code is not necessarily organized in a way that is conducive to PBT. PBT works best on components with clear boundaries and does not easily apply to programs with poorly encapsulated global state, or with leaky or complex abstraction boundaries.

Rather than write properties that are tied to the structure of the program, we will explore how to define properties over user-defined event logs. Consider the case of P7, who was testing a system, let's call it "`Inner`," that was difficult to test because its most interesting behavior arose only when interacting with a particular calling component, "`Outer`". `Outer` took in simple inputs and used them to construct much more complex inputs to `Inner`. Thus, `Inner` could not be tested with realistic inputs without the complex apparatus of `Outer` to produce those inputs. The developer cited this as a circumstance where a PBT is difficult to apply, because the abstraction boundary between the components was too fuzzy.

We will develop a tool that allows developers to express properties that reach into the state of multiple interacting software components. Take again the example of `Inner`. A solution to P7's problem could be to drive the test through `Outer` but write properties about `Inner` by monitoring the internal values that appear while it runs. For example, suppose `Inner` processes a stream of messages and its internal state includes a queue of processed items, `processed`, and an `overflow` flag. Useful properties might include:

```
prefix_of (processed, next (processed))       (* 1. Processing is monotonic *)
msg.length < 100 ==> never (overflow = true) (* 2. Can handle 100 bytes    *)
eventually (processed.length = msg.length)    (* 3. whole message processed *)
```

Is it possible to check such behaviors with built-in assertion statements? We posit that an assertion-based approach is non-ideal. On the one hand, assertions would require nontrivial code to save previous values for properties like (1) above, and likely involves brittle macros or meta-programming to remove such auxiliary code at release time. Furthermore, it is difficult to imagine using the assertion approach to checking properties (2) or (3) at all, because it would require assertions cognizant of the "end" of a computation, information that may not be available to the assertion statement.

Instead, we propose tooling where properties can be defined over logged values. The first key component of the tool is the invocation of logging for specific values using lightweight annotations within modules like `Inner`. The second component is a language for writing properties over the annotations, capable of capturing concepts like `next` and `never` from the example properties above. This language will need to take inspiration from temporal logics, for example linear temporal logic (LTL).

Temporal logics have been used for PBT in prior work [91], but adapting that work to this situation will require some finesse. For one thing, the prior work assumes a fixed source of logical events, and thus does not have to support arbitrary predicates over the kinds of values that may appear in a program trace. For another, LTL is thought by some to be difficult to use [38]; we hope to choose a simplified set of

```
module StringFns : sig                        module Set : sig
  val reverse : string -> string                type 'a t     val empty : 'a t
  val drop_n  : int -> string -> string         val mem   : 'a t -> 'a -> bool
  val split   : string -> string * string       val add   : 'a t -> 'a -> 'a t
```

Figure 3: Some module interfaces we would like to test automatically.

temporal connectives that are intuitive for users. The end goal of this task is a tool that compiles properties written in some friendly temporal logic into lower-level properties over log traces, which are validated by providing random inputs to an outer system *containing* the component(s) of concern.

Publication target: Object Oriented Programming, Systems, Languages, and Applications (OOPSLA).

5.2 Mixed-initiative property specification. Even if a developer believes, in the abstract, that PBT should useful to them, they may still have trouble writing concrete properties in situations that actually come up in their work. We will explore how programmers can engage in an iterative loop to compose properties in collaboration with their tools, focusing on first encounters with PBT in an educational setting.

We will design a novel mixed-initiative [1] workflow where students and their code editor work together to arrive at meaningful properties. Prior work has contributed techniques that automatically extract properties from programs [2, 77, 19, 112, etc.]. Our aim is to explore how student and tool can collaborate to arrive at properties that are not just *accurate* descriptions of correct system behavior, but also capture *important* aspects of behavior. To achieve this aim, it will require designing interactions for developers to specify desired properties directly, plus extensions to property extractors that incorporate this information. We envisage interactions that help students steer a property extractor by (1) selecting specific functions or procedures that they want to check, and then indicating the kinds of properties they wish to test by (2) writing unit test cases or REPL assertions that represent special cases of more general properties. This interaction would be mixed-initiative, permitting a user to refine generated properties by providing additional test cases that should be satisfied or rejected. Such interactions would be prototyped on top of an open source property generator tool like QuickSpec [19], using user inputs to steer generation, and evaluated in Penn's upper division Haskell course (CIS 5520).

Publication target: User Interface Software and Technology (UIST).

5.3 Model-based properties for modules. One surprising finding from the Jane Street study was that it is quite common for developers to build (or already have) a *model implementation* of the code they are testing and want to test that the two implementations behave the same. This is a well-documented approach to PBT [59], but it is not well publicized, nor is it supported as well as it could be by existing tooling. With this in mind, we plan to build a tool to automate model-based testing for systems written in languages with strong *module systems*, like OCaml, Standard ML, etc. [86]. We will start in OCaml because our relationship with Jane Street makes it easy to rapidly get feedback on our findings and tools.

Model-based testing is almost trivial in the case where the component under test and the model are both pure functions. But when the code under test is a *collection* of functions organized into a module, things get much more interesting. Testing even a simple module requires orchestrating multiple calls to the different functions in its signature. For example, testing the signature `StringFns` in Figure 3 requires wiring together calls the functions in the signature in a well typed way—e.g., testing `drop_n` requires an integer and a string, each of which must be either generated or obtained from a call to a different function in the signature (e.g., `split`). We will build on prior work [59] to generate well-typed sequences of function calls that can be used to compare module implementations.

Testing modules gets even hairier when they contain *abstract types*, as is the case with `Set`. Now some of the types in the signature cannot be generated on their own, but must be obtained by calling other functions in the signature (e.g., `create` and `add`). The `Set` signature is also *polymorphic*, so a concrete instance must be chosen for the element type `'a` before testing; there is one significant piece of work on this problem [57], but it is not solved in general.

To address these problems, we will take a foundational approach, breaking down modules into their constituent features and developing automation heuristics from first principles. This will involve identi-

fying a subset of modules, e.g., potentially those that use generation-unfriendly types like GADTs, where automation is *not* possible; highlighting this subset will provide useful context for future work that might aim to simplify these cases in other ways (e.g., by putting a human in the loop). For the modules where our automation does apply, we will provide a totally automatic solution.

The work in this project will ultimately provide a recipe for automating a significant swath of module testing situations, massively lowering the barrier to entry for PBT in languages with OCaml-like module systems. It will also provide a basis that others can build on in other languages: abstract and polymorphic signatures are features of many other modularity mechanisms (e.g., Java interfaces, Rust traits, Haskell type classes, etc.), so solutions we develop for OCaml should transfer.

Publication target: International Conference on Functional Programming (ICFP) and/or Principles of Programming Languages (POPL).

5.4 Automatic explanations for properties. Like any code, properties can be hard to understand, particularly by programmers who did not write them. Can we design tools to help explain them?

We will explore how to design useful in-situ explanations of properties, for example explanations that could be viewed as just a few lines in tooltips and inline annotations in the editor and which clarify the meaning of an opaque property. This involves several research challenges.

A first challenge is to simply address the right aspects of complexity; at the minimum, properties can be confusing because of their use of complex data (e.g., binary search trees, even logs), structural complexity (as is the cases for properties involving many predicates), or use of specialized syntax (as happens for advanced properties incorporating specialized operators in frameworks like QuickSpec). We know all of these manifest in some properties, but the frequencies are unclear. We will characterize complexity as it arises in the properties developers write by conducting a content analysis [70] of code bases using PBT.

A second challenge is understanding how to generate explanations that will help a programmer understand properties at-a-glance. Several approaches may work well here. **Input-Output Examples.** A first approach is to generate example input-output pairs demonstrating the behavior that a property tests. This approach is motivated by other recent efforts to generate input-output examples for instructional purposes (e.g., [32, 22]). We will leverage reflective generators and shrinkers to generate generating understandable inputs with potential complex structure, finding both positive examples and negative (i.e., failing) examples of program's behavior. **Plain Language Explanations.** PI Head has prior work developing rule-based textual explainers of DSLs like CSS selectors and Unix commands [43]; a similar approach might work in this context. The challenge in this research is understanding patterns of textual explanations that both "speak the language" of a reader, and which can be reliably generated from static analysis of the code. **Large Language Models.** Large language models have recently made strides in being able to explain source code; PI Head is exploring such technologies for general-purpose programming languages, and this approach may work well for frameworks like Hypothesis where properties are Python functions.

The above efforts will contribute a characterization of complexity in property specifications, techniques for generating readable examples and descriptions for properties, and comparisons in usability studies.

Publication target: IEEE Visual Languages and Human-Centric Computing (VL/HCC).

6 Interaction: PBT at Users' Fingertips

In this section, we describe a sequence of research efforts to design and evaluate novel modes of *interaction* for PBT. These projects will contribute new techniques for (1) assessing whether their generators are generating sufficient and appropriate inputs (§6.1), (2) altering their generators to achieve better distributions (§6.2), and (3) understanding testing-provoked failures involving complex inputs (§6.3). We also describe our plans to design mixed-initiative [1] interactions by which a developer could tailor a raw counterexample output by a PBT tool into the source code for a unit test they would wish to maintain (§6.4).

Each of these projects will be developed following an artifact-driven [124] HCI research methodology. In this style of research, contributions come in the form of insights into task structure arising from close observational interviews; the creation of novel interaction primitives that map closely to these tasks; the refinement of technical methods (like those from §4 and §5) to support proof-of-concept implementation; and the lessons for interaction and algorithm design that arise from their evaluation with human users.

Success will be measure by the interactions' ability to reduce testing and debugging time and increase effectiveness. PI Head has extensive experience with this style of work (see [43, 117, 44, 48, 45, 46, 47]).

6.1 Evaluating data distributions. Unlike conventional unit testing, where developers write concrete test cases, PBT automatically generates tests by drawing inputs from some automatically generated distribution. The success of PBT thus depends on the quality of this distribution—whether most of its probability mass is on tests that are both sufficiently "interesting" and sufficiently diverse.

The challenge is that it is by no means easy for a developer to determine if a distribution is a "good" distribution. One common strategy is to inspect a small number of inputs by hand; though this strategy obviously leaves developers without an understanding of the breadth of what is tested by the tests. Despite advances in HCI in visualizing input data distributions generally, (e.g., for machine learning model training [52] and [53]) and visualizing program sequences of program values (e.g., [64]), there remains no fitting solution for the PBT case. We aim to remedy that.

More concretely, PBT poses the unique challenge that test inputs are programmatically generated and can be of unbounded structural complexity (e.g., lists, trees, other algebraic data types, sequences of API calls). Consider an example from a participant in our formative study, who wanted to generate realistic logs of input data, where each log entry included at least a timestamp and an event type. Ideally, the developer would be able, by examining the generator's distribution, to answer questions like: Are the generated log inputs long enough? And are the event sequences realistic?

Our research efforts will focus on contributing the interaction primitives and unifying environments necessary to answer questions like these. We anticipate making several innovations to bring this about. A mockup of these innovations appears in Figure 4.

The first research innovation will be the design of live displays of generated values. These displays will show aggregate data views, including aggregate statistics (Figure 4.2) and visualizations of the distribution of important summary features of the data (Figure 4.3). The displays will be live [118] and real-time. Noting the success of other successful live functional programming environments in industry and academia (e.g., [81, 93]), we anticipate liveness will provide visibility and control (i.e., the ability to change or halt) long-running, opaque data generation processes. Visualizations will be generated using recommendation rules as done in related work in exploratory data analysis [78, 125, 126]. Unique to our setting, the tool will leverage type-based automation and user-defined functions to extract features to visualize. For example, consider the `log` type from above. A developer might be interested in the log's `length`, field accessors like `event_type`, `id`, and `timestamp`, filters like `is_empty`, and aggregators like `max_by`. Such features can be generated automatically for common data types.

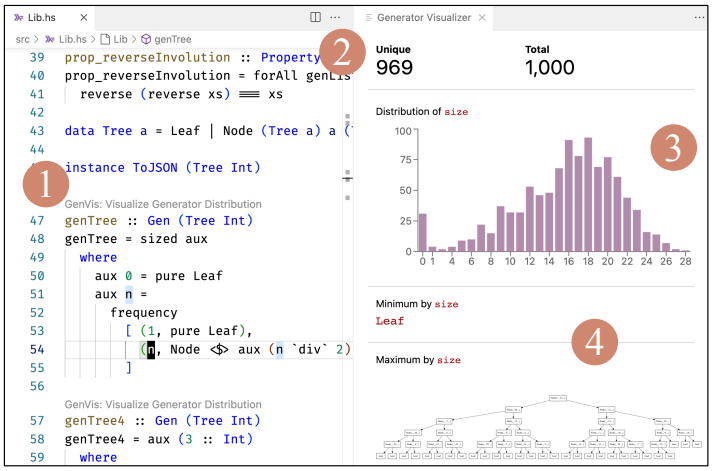

Figure 4: A tool for evaluating data distributions, showing generator source code (1), summary statistics (2), aggregate visualizations (3), and views of input instances (4). Iteration would be supported with live adjustment of generator code and refinement of filters used in the visualizations.

A second innovation will be environmental support for rapid customization with user-defined functions. If there are features that the developer notices should be extracted but that the system cannot come up with itself (e.g., `ids_unique`), the developer can write them alongside their property specifications; the interface will automatically load those features into the display.

A third innovation will be interaction primitives for drilling down into individual inputs from aggregate views by selecting marks in visualizations (e.g., choosing a bar of length "10" to preview individual inputs with that length). Prior HCI tools (e.g., [52]) provide this feature generally. What is unique in our

domain is to provide suitable representations of complex inputs that will be easy to understand. Some general solutions might be to pretty-print inputs, provide interactive object browsers like those available in JetBrains [62], to allow a developer to explore an object using a built-in REPL, and to produce DOT [29] representations of common kinds of inputs (i.e., lists, trees) for at-a-glance comprehension of larger inputs (Figure 4.4). We will try each of these approaches and compare their effectiveness in pilot studies.

A fourth innovation is the synchronization of above views with live feedback on what code is exercised by the tests. Building on the design pattern whereby a tool show which lines of code are currently executing [13, 95, 16], our tools will colorize code to show how frequently each line is executed while running tests, both in aggregate and for individual tests. Altogether, the above activities will be a set of interrelated interaction primitives that serve as building blocks for understanding distributions of complex input data. Success of this (and the next two projects) will be determined by the novel interactions' ability to reduce testing time and achieve desirable input distributions.

Publication target: User Interface Software and Technology (UIST).

6.2 Tuning data distributions. Tuning generators to produce "interesting" values more often can be challenging, requiring significant trial and error. Purely automated approaches like those we describe in §4.1 provide part of the solution, as do code coverage-based generation techniques (e.g. [129]). That said, even these techniques fall short of the kinds of inputs a developer would wish to produce.

Building on our work in §4.1, we envision new ways of specifying desired distributions through interaction. One innovation we will work on is to support tuning by manipulating the visualizations from §6.1. The goal is to enable a kind of bidirectionality for PBT programming that has seen success in other areas of programming research (e.g., [49, 65, 94, 92]). A first approach is via input filtering: a developer could select a region (e.g., a bar) in an aggregate visualization and request that inputs from that region are discarded before testing. This approach is flexible but limited, as it does not change the underlying generator.

Therefore, we will design additional interactions that leverage the unique ability of reflective generators (§4.1) to change the underlying generator parameters. As developers interact with a visualization, the reflective generator will map interactions to choices in the generator. The generator code will be displayed side-by-side with visualizations of the distribution, and the user will be able to manipulate both sides. Developers will be able to interact directly with the visualization to indicate, for example, that they would like to see more inputs like one that has already been generated, or that they would like an input similar to a generated input, but different in a way that they are demonstrating. This project and that in §6.1 will together yield new paradigms for understanding and changing input data distributions.

Publication target: User Interface Software and Technology (UIST).

6.3 Interactive shrinking and debugging. As with any kind of testing, one challenge with PBT is understanding why a given counterexample triggers a failure. We will explore new interactions that take advantage of the machinery available from PBT tooling to help programmers understand failures.

Our first effort will be to design interactive input shrinkers. Typical automated shrinking approaches (e.g., [58, 3], and our own proposed techniques in §4.1) can get stuck at local minima far from the global minimum, without realistic ways for users to get them unstuck. But developers can see shrinking options that a shrinker may not. Could interaction help a developer shrink a stubbornly large input?

We can take inspiration here from HCI research on incremental code simplification (e.g., [82, 45, 55, 51]), but new ideas are needed when the simplification target is code with recursion and/or complex data. We propose a semi-automatic interaction paradigm where a developer performs some simplifications manually in an interactive object viewer (like those in modern debuggers like JetBrains IDE [62]) while a shrinker attempts further simplifications in the background. The system would display the valid ways an input can be altered, determined using reflective generators, and automatically shrink after each action.

We also see opportunities to help developers find the root causes of failures in a PBT setting. A current approach is implemented by Hypothesis' `explain` module, which points users to lines that are always run by failed tests, but never by successful tests. But we found this technique to be inconsistent, even on examples provided in the Hypothesis documentation. Could a tool highlight divergences in other more useful ways? PI Head was on the team that designed views of trace divergences for student programmers debugging failed submissions versus a reference program [117], but this technique is not quite right for PBT, where is no correct program to compare to. Instead, we need to show divergences in a way that

(1) points programmers to control decisions that vary between a failure-inducing input and a set of very similar success-inducing ones (using the reflective generator framework to assess similarity); (2) shows what lines will be executed next following the control decision by the gamut of comparison inputs; and (3) supports rapid navigation between control decisions in the case of greatly diverging traces. We will iteratively design, implement, and evaluate a tool supporting such interactions.

Publication target: User Interface Software and Technology (UIST).

6.4 Counterexamples as regression tests. One pain point reported by informants in our Jane Street interviews is that it requires considerable work to transform a (repaired) counterexample detected by their PBT tools into a regression test, even though much of this work feels mechanical.

Much of this work could be automated away. After all, a repaired counterexample is just an input to some component plus (as specified in the property) a way of checking that the produced output matches expectations. In theory, this should easily be transformable into a unit test. In practice, there are numerous issues to deal with. Consider: (1) an input may be so large it belongs in an external file; (2) an input may be passed through so many operations that a tester wishes they were split up into independent lines, and perhaps even tested individually; or (3) a developer may wish to remove a precondition check from a property (consider the left side of `prop_insert_correct` in §1). In all of these cases (and many others), automation cannot know a developer would wish to make these choices beforehand, but it can at least make them readily available to the developer.

We will design a mixed-initiative [1] interaction flow that supports creating unit tests from counterexamples. In this flow, a developer's editor creates a provisional unit test. The developer is guided through a sequence of decisions corresponding to different ways they may wish to "write out" the test case. Each decision results in a re-rendered test (e.g., where inputs are moved to a file, operations are split up, etc.) As with PI Head's prior work with similar tools [45], innovations in this work come from identifying a small number of powerful decisions and sequencing them in an order that is easy for developers to follow. Decision points will be chosen following observations of developers creating test cases from counterexamples; we will prioritize hard-to-automate decisions—ones that are time-consuming or vary by context.

Publication target: Foundations of Software Engineering (FSE).

7 Diffusion: Advancing Open-Source Tools

The activities described in the previous sections are focused on fundamental research to advance understanding of PBT and explore new approaches to both foundations and tool design. But research alone is not enough to reach the broad impact we seek. Accordingly, our agenda includes plans for increasing the *diffusion* of PBT with both educational efforts and well-engineered tools embodying our findings.

We begin with activities for ensuring that our foundational advances have an impact on real users, with the help of a research engineer (§7.1). A major part of this effort will be building and promoting TYCHE, an integrated development environment for PBT (§7.2). Next, we discuss plans to support open-source frameworks for PBT (§7.3). Finally, our education-focused activities include developing a catalog of high-leverage PBT use-cases (§7.4) and undergraduate- and masters-level course materials for PBT (§7.5).

These projects will also result in a number of developer-focused talks. We will bring them to industry conferences like "Yow! Lambda Jam," "Strangeloop," and "Lambda World."

7.1 Beyond research software. The research projects described above have the potential to dramatically improve the power and usability of PBT, but only if their products are attractive to real users. There is an unfortunate tendency for research software to remain just that—missing critical documentation that new users need to get started and lacking a clear maintenance schedule that would give software companies the confidence to adopt. To avoid this fate for our projects, the PhD students will work closely with our research engineer. The code for those projects will be designed with learnability and maintainability in mind, and upon completion it will be handed off to appropriate owners in the open source community.

Concretely, we expect the research engineer to be most involved in the following projects. *PBT interaction models* (§5.4): Helping PhD 1 to build and maintain a minimal PBT framework. *Theory of reflective generators* (§4.1): Implementing and maintaining versions of reflective generators in a number of languages

and PBT ecosystems, with PhD 2. *Mixed-initiative property specification* (§5.2) and *Counterexamples as regression tests* (§6.4): Assisting PhDs 3 and 4 with building tools for property specification. Most importantly, the research engineer will help to build and maintain TYCHE, the IDE for PBT designed by PhD 4.

7.2 Tyche: An IDE for PBT. The projects in §6 will explore a wealth of new modes of interaction for PBT. Their potential impact will be magnified by integrated them into a system that is attractive to would-be users. This is the inspiration for TYCHE, an IDE extension for PBT. TYCHE will first serve as a platform for building and testing all the functionality designed in §6. Promising features from research will constitute its core. TYCHE will be developed as a Visual Studio Code extension, taking advantage of VSCode's status as the most widely used editor according to a 2022 survey [115] and its support for every language with mature PBT tooling. To maintain focus, we will concentrate our Efforts on making TYCHE especially effective for one widely used PBT framework—probably Hypothesis, since its developers have expressed interest in including TYCHE in the main Python language extension for VSCode. But we will make the code of TYCHE language-agnostic wherever possible, to allow other language communities to use it with their PBT frameworks.

The research engineer will lead feature integration, ensuring features come together into a seamless whole. They will also develop TYCHE to serve as a platform for further innovation as it will have been for our own: extension points will be added for others to create and evaluate techniques for understanding and creating properties and generators. In this way, TYCHE will provide a pipeline for bringing research on PBT algorithms and interaction models to developers craving better tools.

We will make TYCHE known to the developer community through talks at developer conferences and by disseminating information via our contacts in industry and the open source community. Our goal is for TYCHE to reach 60,000 downloads by the end of the project. At the time of writing, this would put TYCHE in the top 10 extensions tagged "Testing" in the Visual Studio Code extension marketplace that are not either full language servers or Large Language Model interfaces [119].

7.3 Nurturing the PBT ecosystem. The existing ecosystem of open-source PBT tooling is broad and varied. Some projects are vibrant, while others, including some with great ideas and implementations, do not have the developer bandwidth to address user needs. Additionally, existing frameworks have not been designed with the benefit of the insights and tools that we will build in this project, leaving (we believe) large opportunities for improvement. Thus, we will allocate a significant part of the engineer's time to supporting and improving open-source PBT frameworks. To make best use of limited resources, we will choose existing PBT frameworks as targets and use the findings and software products from our research to boost their usability and adoption.

We will start with Python's Hypothesis framework, since it is well-established and yet has significant room for growth. We are in contact with Zac Hatfield-Dodds, the main maintainer; a letter of collaboration from Mr. Hatfield-Dodds appears in the supplemental documents. The Hypothesis team is especially excited about integrating ideas like reflective generators (§4.1) and interaction modes like distribution visualization (§4) into Python's VSCode environment. The success of this collaboration will be measured by the Hypothesis team's willingness to include our tools in future releases and by uptake in the community.

Besides Hypothesis, we will choose two additional projects to contribute to in smaller ways, prioritizing great projects in popular programming languages that have significant room for growth. Candidates include Scala's ScalaCheck, OCaml's Crowbar, which has great ideas and some attractive low-hanging fruit [27], and projects like `jqwik` in Java or `proptest` in Rust, which are embedded in massive language communities that could benefit from better tools for PBT. We will work closely with the developers and maintainers of these projects to ensure that our input is well received by the community.

7.4 "When to Specify It!". Besides engineering efforts, we plan to disseminate PBT via educational materials. Our interviews at Jane Street suggest that the developers there have a tacit understanding of several "no-brainer" situations where PBT is an obvious choice, where properties are easy to find and PBT provides more thorough testing than other techniques—many developers seemed to limit their use of PBT to such situations. While experienced PBT users do also apply it in high-cost / high-benefit situations, it appears that focusing educational efforts on easy cases may be the best way to drive adoption.

We will produce resources to help developers understand high-leverage situations for PBT. We will begin with an effort tailored to the academic community: a survey paper with the aspirational title "When to Specify It!" in homage to John Hughes's widely-viewed tutorial "How to Specify It!". This survey will document a range of high-leverage scenarios identified in our formative research, including cases like (1) "these two functions (e.g., a parser and a printer) should round-trip," (2) "this data structure (e.g. a set, map, etc.) should obey algebraic laws," (3) "this stateful module should uphold an invariant," (4) "these two programs should behave the same" (e.g., because one is an optimized version of the other), and (5) "this program should not crash." This list will be expanded based on follow-up surveys with broader communities of developers (§3.1), a comprehensive review of case studies that appear in the academic literature, and an examination of open source projects across a variety of different software ecosystems using PBT (e.g., QuickCheck/Haskell, Hypothesis/Python, Quickcheck/OCaml).

We will distill these findings into media directly tailored for developers. First, we will write approachable developer documentation in partnership with our industry collaborators, to be read among the first resources of any developer documentation on PBT tools. We will work with our industry collaborators to disseminate this documentation and incorporate it into blogs and tool manuals. Finally we will package the findings as a talk for one or more professional conferences.

7.5 Curricula for PBT. Education can act as a force multiplier to drive adoption of new technologies in industry. We will develop materials for integrating PBT into early undergraduate courses—initially targeting introductory data structures courses—as well as higher-level courses for masters students. As noted in §7.4, testing well-defined data abstractions is one of the high-leverage scenarios for using PBT, making data structures a powerful anchor for demonstrating the power of PBT. Building on recent PBT teaching in data structures courses [127, 90], we will integrate PBT into Penn's CIS 1210 course on data structures (which PI Pierce regularly teaches). We will evolve the CIS 1200 curriculum to emphasize themes in PBT from our formative research—good circumstances for using PBT, methods for writing useful generators, properties as documentation, and scenarios where PBT is suitable (see §7.4). We will also include PBT in Penn's CIS 5730, a masters-level software engineering course. We will evaluate the impact of these new instructional approaches on students' ability to leverage PBT in final projects, disseminating our findings in publications at computer science education venues. The instructional materials we develop will be made available for instructors at other institutions to use in their own courses.

Publication target: Koli Calling International Conference on Computing Education Research.

8 Broader Impacts

Broad impact is central to our research agenda, and many of the specific tasks we have described—particularly those grouped under Diffusion (§7)—aim for broad impacts. Here we recapitulate those aims and sketch our plans for mentoring, diversity, and broadening participation in computing.

Impact on industry. The main broad impact from the proposed work will be a significant increase in the use of PBT across the software industry within the time frame of the grant, driven by more powerful and usable tools. Specifically, the research engineer will work to bridge the gap between the foundational insights and prototype tools coming from the four PhD students and the needs and priorities of industrial developers. Specific activities led or amplified by the research engineer are discussed in §7.1 and §7.3.

Educational Impact. A second, supporting arena for broad impact will be the development of educational materials for both students and professional developers. Educational activities will be coordinated with the rest of the work, and are integral to the project's overall goal of making PBT a standard testing methodology. Specific pedagogical threads within the project are described in §7.4 and §7.5, but the tasks in §3, §4.3, §5.2, §5.4, and §6.1 will also influence education. The work in §7.5 will generate course materials that we will share publicly; we will also report our experiences in a paper for a CS-ed-focused conference.

An ancillary educational goal is to publicize and communicate the benefits of PBT to the broader computer science *research* community. As part of these efforts, we will write an article on PBT for the Communications of the Association for Computing Machinery (CACM).

Mentoring and Diversity. The majority of the requested funding will support formative research experiences and mentoring for graduate students. We also plan to work with 10 undergraduate students via an

REU program (more in the next paragraph) along with other undergraduate and masters students working for credit; they, too will benefit from the research experience. Each graduate student will lead multiple facets of the project, including co-supervising undergraduate and masters researchers.

The PIs will recruit students for this project with a mind towards making the research area reflective of diversity in the US and Pennsylvania. The PIs have already made inroads broadening participation of women in their groups (1/3 of Pierce's direct PhD mentees and 2/3 of PI Head's are women). That said, a particular area of focus going forward will be increasing representation of other underrepresented groups, including Black and Latinx students. Pierce (along with Penn colleagues Zdancewic and Weirich) was recently informed that he will receive funding for an NSF REU program that will involve 24 undergraduates (either per year for three+ years), selected specifically with an eye to diversity; for the present project, we are requesting funds to add two more per year specifically to work on PBT topics, many of which are well suited to undergraduate interns. See the BPC supplementary document for more details.

Benefits to Society. The project's goals will also be advanced by open-source distribution of tools built during the project. Key systems will be engineered and documented to a standard that makes them immediately useful to engineers and students; the projects around improved generation and shrinking, PBT over logs, and evaluating data distributions will be particular targets for widespread dissemination. The remaining projects will be published under a permissive open source license to serve as models for similar tools in other programming languages and environments.

This project also presents excellent opportunities for strengthening collaborations between university researchers and industrial advocates of PBT. Our ongoing user study at Jane Street has been carried out with the enthusiastic support of their developers and management, and we hope to continue using Jane Street as a testbed for prototypes of our tools. We are also in active discussions with the developers of Hypothesis, who are keenly interested in the results of this proposal. We plan to establish connections with the developers and user communities of popular PBT tools in languages like Java, Rust, and Scala.

Longer term, better testing means better software. As software systems have grown to the gigantic scale seen today, good testing methodologies and tools (unit testing tools, test-first design methods, etc.) have come to play an ever more crucial role. Adding a powerful new testing tool to programmers' toolbelts will further streamline this aspect of the development process, leading to software of every sort that is more robust, more reliable, and less expensive.

9 Results from Prior NSF Support

PI Pierce: (NSF 1955565) "Collaborative Research: SHF: Medium: Bringing Python Up to Speed" ($437,999, 7/2020–6/2023), with co-PIs Michael Hicks (Maryland) and Emery Berger (Amherst). The project aimed to dramatically increase the performance and correctness of applications written in Python by developing novel techniques for performance analysis, optimization, run-time systems, property-based random testing, concolic execution, and program synthesis. It developed both novel performance analysis tools and optimizations and novel automatic testing frameworks. These were largely tailored to and implemented for Python, but applicable in other, similar languages. **Intellectual Merit.** The project involved work on both performance measurement (mostly at Amherst and Maryland) and PBT (mostly at Penn and Maryland). Specific threads of work involving Penn included building an early version [**?**] of the Reflective Generators described in §4.1, carrying out the pilot study of PBT in Python mentioned in the motivation section above [35], and building on the idea of freer monads from functional programming to develop "free generators," which unify parsing and generation [37], presented a principled automatic testing framework for application-layer protocols [80], and developed and released a freely available mutation testing framework for Python, called `pytest-mutagen` [96], and applied ideas from combinatorial testing, a widely studied testing methodology, to modify the distributions of random test-case generators so as to find bugs with fewer tests [36]. **Broader Impacts.** Project results and open-source software products are being used to increase the performance and correctness of Python applications. Educational impact has included training graduate and undergraduate students, including a female PhD student, Jessica Shi. **Publications (involving Penn):** [**?**, 36, 37, 35, 80]. **Research Products (involving Penn)** [96].

PI Head has not previously received NSF support.