Tyche: In Situ Analysis of Random Testing Effectiveness (Demo)

Harrison Goldstein
University of Pennsylvania
Philadelphia, PA, USA

Benjamin C. Pierce
University of Pennsylvania
Philadelphia, PA, USA

Andrew Head
University of Pennsylvania
Philadelphia, PA, USA

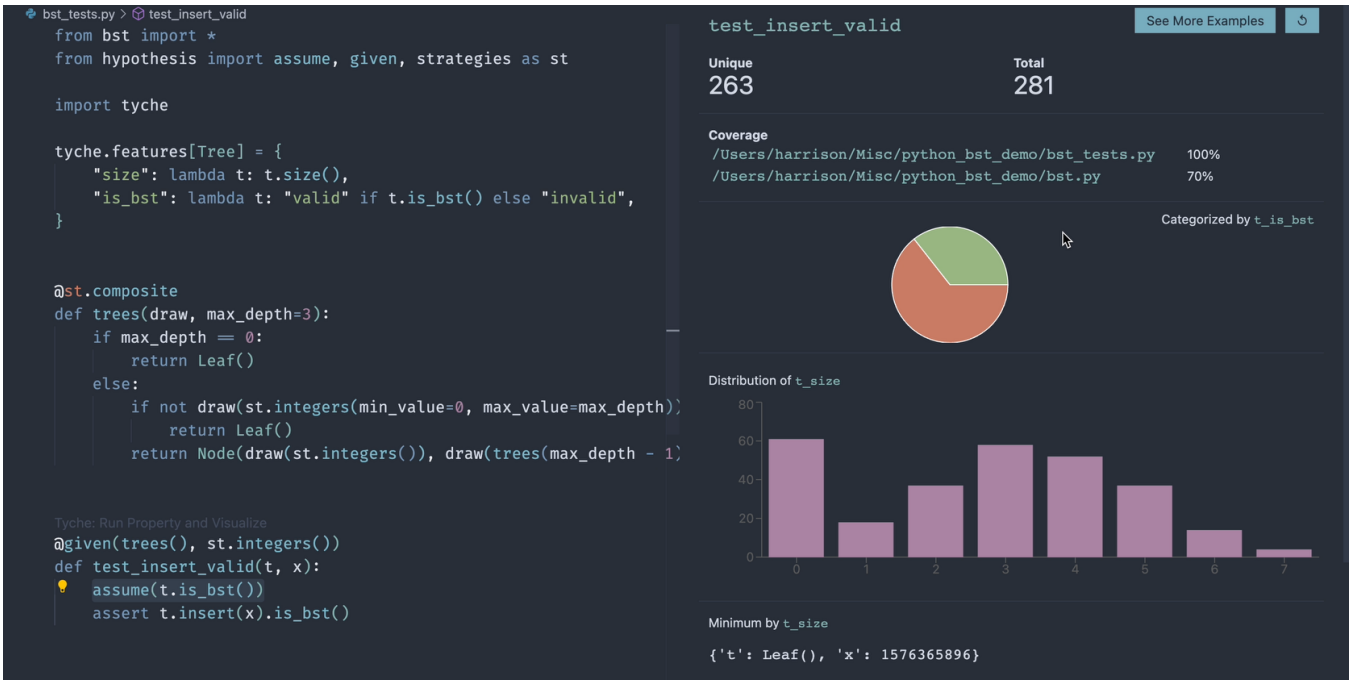

Figure 1: The Tyche property-based testing interface.

ABSTRACT

Automated testing tools have adapted to increasing program complexity by reducing the user's role in the testing process. Approaches like *property-based testing* supplement traditional unit-testing with a mode declarative approach: rather than write traditional input-output examples, the user writes executable specifications of their programs. The testing framework then exercises those specifications with randomly generated values.

However, more automated approaches to testing risk hiding too much from the user. Current property-based testing frameworks give insufficient feedback about the specific values that were used to test a given program and about the distributional trends in those values. In the worst case, this lack of visibility process may give users false confidence, encouraging them to believe their testing was thorough when, in fact, it had critical gaps.

We demonstrate Tyche, an editor extension that recovers visibility into the property-based testing process. Tyche provides an interactive interface for understanding testing effectiveness, surfacing both "pre-testing" information about test inputs and their distributions and "post-testing" information like code coverage. The extension is designed to work out of the box with tests written in Python's popular Hypothesis framework, so users can immediately start using it to improve their testing.

ACM Reference Format:
Harrison Goldstein, Benjamin C. Pierce, and Andrew Head. 2023. Tyche: In Situ Analysis of Random Testing Effectiveness (Demo). In *The 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23 Adjunct), October 29-November 1, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3586182.3615788

1 INTRODUCTION

Property-based testing (PBT) frameworks, like the Hypothesis [6] library in Python, allow users to specify their program's behavior and check that the program behaves according to that specification. To write a property in Hypothesis, the user simply writes a function that performs some operation on the system under test: if this function raises an exception, the property is judged to have failed; otherwise the property passes. Properties succinctly and declaratively capture aspects of program correctness, with users specifying what

UIST '23 Adjunct, October 29-November 1, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0096-5/23/10.
https://doi.org/10.1145/3586182.3615788

(e.g., make sure this function maintains a particular invariant) to test but not exactly how to test it (e.g., check that invariant with `x`, `y`, `z` specific inputs).

Unfortunately, in practice this declarative approach presents a rather leaky abstraction—programmers cannot always rely on the system to figure out how best to test a given property. For example, many properties come with *preconditions*: if the input data does not fit a certain form, the test will be discarded and the property will pass trivially. In such cases, Hypothesis can struggle to find bugs. Most PBT frameworks have some way to mark a test as "discarded," and they print a warning if too many tests are thrown away, but these messages are easy to miss. Worse, even a generator that exclusively produces inputs that satisfy the property's precondition might still fail to find bugs because its *distribution* is not interesting or varied enough. If the user is not paying attention to these subtleties, they may incorrectly conclude that their code is bug-free.

There is a fundamental interface design problem here. While modern testing interfaces succeed well at communicating test failures, they fail to communicate *what it means when testing "succeeds"*. Developers need interactive tools that allow them to analyze and understand the effectiveness of passing properties, improve test distributions that miss the mark, and gain well-founded confidence that their final test set of properties covers the cases they care about.

Our solution is a tool called Tyche. After a brief discussion of related work (§2), we describe the following contributions:

- We present Tyche, an editor extension that provides *in-situ* visibility into PBT effectiveness, with affordances for inspecting input distributions, exploring input examples, and evaluating line coverage. (§3)
- We demonstrate Tyche on a common PBT case study and show that it can help users improve their testing. It gives feedback about inadequate testing and responds in real time as the user updates their testing code. (§4)
- We show that Tyche also works on real-world code bases "out of the box," applying it in the context of a significant Python project pulled as-is from GitHub. (§5)

We conclude with a brief discussion of future work (§6).

2 RELATED WORK

Property-based testing was popularized by QuickCheck [3] in Haskell. Since then, similar libraries have appeared in most major programming languages. Hypothesis is one of the most widely used.

The vast majority of PBT frameworks work from the command line. Hypothesis, for example provides `hypothesis[cli]`, which runs property-based tests and prints results to the terminal, and it integrates with `pytest` [5], so properties can be checked as part of an existing test suite. In both cases, but the latter especially, information can get lost: test output is often hidden or de-emphasized when tests pass.

The adjacent "fuzzing" community has some more interesting approaches to testing feedback. For example, AFL++ [4] includes a sophisticated TUI with feedback on code coverage and other fuzzing statistics over time.

3 TYCHE: AN INTERFACE FOR PBT FEEDBACK

Tyche an extension to the Visual Studio Code (VSCode) editor that provides a new paradigm for interacting with and testing properties. (See the screenshot in Figure 1.) The user invokes Tyche by clicking on a button that the extension places above each property declaration in the user's currently open file. When clicked, the extension runs the property: if the test fails Tyche displays the failing example; but, more interestingly, if it passes Tyche displays a wealth of information about the testing process that the user can use to decide if the property is being tested properly. The current version of Tyche integrates with Hypothesis.

3.1 Features

Tyche provides three main modes of feedback to the user.

Aggregated Test Input Trends. If a user is unsure that their test input distribution is properly tuned to trigger interesting program behaviors, the first step is to display aspects of the distribution for user inspection. Tyche accomplishes by extracting numerical *features* from the test inputs and displaying their trends in pre-defined charts. For example, the user might extract the size or height of a tree structure.

Individual Test Inputs. High level trends give the usee a broad idea of what their input data looks like, but as users of PBT we also find that it is helpful to simply spot-check individual examples. The user can drill down into the trend charts by clicking on bars or pie-chart segments, bringing them to an example view that is filtered to those values that fall into the given bar or segment. Alternatively, the user can simply page through examples in the order they were generated, allowing them to quickly notice potential issues.

Code Coverage. Black-box notions of testing effectiveness like the ones above, which do not require examining the program's execution or having the source code available, are simple and effective. But in informal conversations during the initial design process, we found that users badly wanted coverage information to supplement the test input visualizations. Tyche displays coverage information in two forms. Users can get a high-level view via a simple table, summarizing the percentage of theoretically-coverable lines that are actually covered during the execution of the property. Then, to dig deeper, users can toggle per-line coverage information: a line is highlighted one color if the line was covered, highlighted another color if it was not covered, and un-highlighted it is not coverable at all. All of this information comes Python's `coverage` library [1].

3.2 Implementation

The VSCode extension is split into two main components: the extension itself and a *webview* that displays the input examples and aggregated statistics. The extension is written in TypeScript using the standard VSCode extension API; the webview is a React app, also written in TypeScript. The two components communicate via browser messages.

The extension provides code lenses that launch the webview. When clicked, the extension runs the selected property in the background by calling an external Python script; the output from that script is parsed as JSON and used to generate the various

visualizations provided by the webview. In principle, this interaction model could be adapted fairly easily to work with other languages and PBT frameworks, although the current version is specialized to work with Hypothesis.

4 CASE STUDY

We demonstrate Tyche's core features on a common PBT example: binary search trees (BSTs). BSTs are a useful example because they are well understood but nontrivial to test. We show how Tyche helps the user to notice and address three common difficulties that might arise when testing an unfamiliar data structure.

The first such issue is the need for preconditions: BSTs require their data to be ordered, but a naïve data generator will not produce very many ordered trees. Tyche alerts the user of this issue with a chart that indicates the proportion of the values that pass the BST precondition, and the user can dig deeper to see that the few values that do pass the precondition are trivial trees.

After fixing this issue by writing a better generator, the user might still not be happy. Generators sometimes require *tuning* to ensure that they produce a wide and interesting distribution of inputs. Tyche is helpful again; this time, the user can explore the live-updating bar charts to gauge which changes to the generator result in the kind of data distribution they are looking for.

Finally, once the inputs to the property look good, the user can use Tyche to validate that the property gets good code coverage over the system under test. If they find that certain lines are uncovered, they can either continue to tune the generator or add further properties that exercise the un-covered code.

5 REAL WORLD

We also demonstrate Tyche in the real world. We chose to use `bidict` [2], a popular Python library for bidirectional mappings that is used extensively in industry, to demonstrate our tool. The library already uses Hypothesis, but has not been adapted at all to work with Tyche. By adding just two lines of code—one to import the `tyche` library and one specify a feature extractor—we can immediately jump in and start analyzing the properties as they run.

6 CONCLUSION AND FUTURE WORK

We are excited to share this demo with the UIST community and get feedback on our interface design. This project is still early stage: while the current version of Tyche already seems useful, we hope a future version can really change the way that users think about and interact with PBT.

We plan to add a number of new affordances to Tyche.

Debugging Test Failures. The current version of Tyche simply prints an error message if the property fails. For the moment this is fine, as we are currently concerned with validating passing tests, but moving forward we hope to include tools for debugging failing tests as well. In particular, we would like to integrate our tool with Hypothesis's "explain mode." Tyche could include Tarantula-style [7] coverage feedback and present counterexamples to users in a way that lets them easily manipulate and explore them.

Bidirectional Distribution Tuning. Tyche visualizations already aid users when tuning their random data generators by providing live feedback of how changes impact the distribution. But what if the user could simply click and drag a Tyche bar chart to change their generator's distribution? This kind of bidirectional editing may be possible with the help of other ongoing work in the PBT space, and it may make PBT much more accessible to novices who do not have experience writing data generators by hand.

However, beyond affordances, our biggest goal moving forward is to put this interface in front of real users. The current design is informed by our own use of PBT and PBT tools, but developers who use Hypothesis in their day-to-day work will likely have different ideas about which parts of Tyche are most useful and which parts need to change.

We plan to collaborate with the maintainers of Hypothesis to recruit for and carry out an observational user study, evaluating the design of Tyche and looking for opportunities to make it better. The exact parameters of the study is still up in the air, and feedback will be welcome!

REFERENCES

[1] Ned Batchelder. 2023. coverage: Code coverage measurement for Python. https://github.com/nedbat/coveragepy

[2] Joshua Bronson. 2023. bidict. https://bidict.readthedocs.io/en/main/

[3] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, Montreal, Canada, 268–279. https://doi.org/10.1145/351240.351266

[4] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++} : Combining Incremental Steps of Fuzzing Research. https://www.usenix.org/conference/woot20/presentation/fioraldi

[5] holger krekel. 2023. pytest: helps you write better programs — pytest documentation. https://docs.pytest.org/en/7.4.x/

[6] David R MacIver, Zac Hatfield-Dodds, and others. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891. https://joss.theoj.org/papers/10.21105/joss.01891.pdf

[7] W. Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. 2007. Effective Fault Localization using Code Coverage. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, Vol. 1. 449–456. https://doi.org/10.1109/COMPSAC.2007.109 ISSN: 0730-3157.