Property-Based Testing Across Four Environments in Open-Source Repositories

Sára Juhošová
Delft University of Technology
Delft, The Netherlands
S.Juhosova@tudelft.nl

Harald Toth
Delft University of Technology
Delft, The Netherlands
H.Toth@student.tudelft.nl

Antonios Barotsis
Delft University of Technology
Delft, The Netherlands
A.Barotsis@student.tudelft.nl

Andreea Costea
Delft University of Technology
Delft, The Netherlands
M.A.Costea@tudelft.nl

ABSTRACT

Property-based testing (PBT) is a valuable technique for assessing software correctness, and its adoption differs across contexts. In this work, we examine how PBT is used in open-source repositories across three languages and four frameworks: Java with jqwik, Python with Hypothesis, and Rust with proptest and QuickCheck. Our study reveals that PBT is used to test a variety of systems and usually verifies one of the following properties: adherence to a contract, equivalence with an oracle, or expected behaviour of errors. We found that developers use a combination of customised generators and post-hoc alterations to control the property input, and they write non-trivial logic to verify the properties, often with multiple assertions along various execution paths. Though these patterns are common across the four environments, notable differences emerge from the specific tooling support of each framework and the capabilities of the underlying language. Based on these similarities and differences, our findings point to potential research directions aimed at facilitating developer adoption of PBT.

1 INTRODUCTION

Most of today's developers agree that verifying the correctness of our software is an essential part of the engineering process — though how rigorous that verification should be and what techniques should be employed is still up for debate. The most well-established method for software verification is *testing* [1], a relatively cheap and time-efficient technique for detecting bugs in our programs. Its main limitation is its inability to conclusively prove the absence of bugs in a program [6]. Formal verification techniques, on the other hand, *can* prove a program's correctness with respect to some specification, but have a notoriously steep learning curve, and are usually time-consuming to employ [11].

Somewhere between conventional software testing and formal verification lies *property-based testing* [4]. Property-based testing (PBT) involves defining properties that a system under test (SUT) should satisfy, and then verifying that the property holds for a large amount of automatically generated input data. The canonical example of PBT is demonstrated on a `reverse` function (the SUT), which takes a list as input, and returns that list in reverse. Two properties can be tested for this function, each in a separate test:

(1) for all elements `x`, `reverse([x])` is equal to `[x]`
(2) for all lists `xs`, `reverse(reverse(xs))` is equal to `xs`

Despite promising to be the "middle road", PBT does not seem to be as widely adopted as expected. For example, in Python, the most wide-spread PBT framework, Hypothesis, is only used by about 4% of the user base, which is far behind the roughly 50% adoption rate of pytest, the leading general-purpose testing framework [8]. We hypothesise that this is because PBT requires significant developer involvement, and requires a more generalised way of thinking than normal software testing. Our goal is to determine whether this hypothesis is supported by data from existing PBT practices.

To this end, our study takes a fresh look at how PBT is used by analysing existing property-based tests in open-source repositories, and determining the developer involvement necessary for them. In the long term, we aim to identify which practices generalise across programming languages and PBT frameworks ("environments") and to gain insights into how the techniques and practices that work effectively in one environment can be transferred to others. We make a step towards that goal by answering the following research question:

> **RQ** How is property-based testing used across different environments in open-source repositories?

We break this question down into the following sub-questions:

- **Q1** What is property-based testing used for?
  - **Q1.1** Which systems are tested?
  - **Q1.2** What types of properties are tested?
- **Q2** How are property-based tests implemented?
  - **Q2.1** How is input generated and controlled?
  - **Q2.2** How are the properties asserted?

By answering these questions, we make the following contributions:

- a categorisation of the types of properties being written;
- a definition of the *anatomy of a property-based test*, with insights into how developers interact with this anatomy throughout the four environments; and
- a set of follow-up research questions that aim to improve the writing and comprehension of property-based tests across all environments.

2 METHODOLOGY

The goal of our work is not to report on the distribution of PBT practices in open-source repositories, but rather to identify the

variety of PBT practices being applied. This means that we employ qualitative analysis techniques which "do not aim to identify the distribution of characteristic values but rather aim to find most [...] possible values of a characteristic in the population" [12, p. 3176]. We define this approach below:

Approach: Identify the widest possible variety of PBT practices in open-source repositories.

To get a wider overview of the usage of PBT in practice, we analysed the usage of four different PBT frameworks over three widely used programming languages with different type systems:

(static types)	Java →	JQWIK [JJ]
(dynamic types)	Python →	HYPOTHESIS [PH]
(ownership types)	Rust →	PROPTEST [RP]
	↪	QUICKCHECK [RQ]

With this selection of environments, we can cover property-based tests in languages with various type systems and, in the case of Rust, we can compare between frameworks from the same language.

2.1 Repository Selection

Following the approach defined at the beginning of this section, we tried to get as wide a sample of tests as possible, while still drawing from relevant repositories. For each environment, we collected a total of 60 tests from GitHub by

- identifying "popular"[1] repositories using each framework,
- manually selecting a variety of repositories, and
- randomly selecting up to twelve tests from each repository.

By selecting *up to* twelve tests, we made sure to get data from *at least* five different repositories for each environment.

2.2 Test Analysis

We began the analysis process by assigning each of four computer science researchers involved in this study with an individual environment, and having them perform *open coding* on all tests in their respective environment. Open coding is "the process of coding data inductively and comprehensively, with an open mindset, to enable emergence of information and insights, without looking to find anything specific in the data" [p. 233][10]. We demonstrate the result of this process on the synthetic example below:

```
1 fun test( @ForAll[1] @Positive[2] x[3] : Int[4] ) {
2     assume x.isOdd()[5]
3     let result = sut[6]( Tag(x)[7] )
4     assert result.isValid()[8]
5 }
```

[1] generator
[2] filter
[3] test input
[4] type hint
[5] filtering assumption
[6] system under test (SUT)
[7] SUT input
[8] assertion

After this initial phase, the team met to agree on the emerging codes and to move into the next phase of data collection, formulating a shared understanding of the "anatomy" of a property-based test across the different environments, and defining interesting *features* of a property-based test and recorded them for each test in the dataset. We did this in multiple passes over the data, since each pass revealed edge cases and new questions which required features to be introduced or redefined. For example, we began with collecting the number of assertions in each test, realised that many assertions were on different execution paths, and decided to keep a record of all tests that have "guarded" assertions, i.e., ones that are only executed if a certain condition is met.

Once we were satisfied with the collected features (described in our data repository [2]), we sat down together to *diagram* [3, p. 218–221] with our findings, keeping memos [7] about discovered relationships between the anatomy and features of the tests.

3 OBSERVATIONS

We analysed a total of 240 tests across 38 repositories. Detailed information about each repository, including links, is available in our data repository [2] under an MIT License. The repository also contains our test dataset, with permalinks and feature annotations for each analysed test. We use examples from our dataset throughout this section, denoted as [JJ14] to refer to test 14 in the data repository from JAVA/JQWIK[2].

3.1 What is property-based testing used for?

To understand what PBT is used for, we aim to identify the types of SUTs being tested and the properties being checked. In each repository we analysed, the property-based tests account for less than 1% of the total tests.

Q1.1: Which systems are tested?

To identify the system under test in each property-based test, we used a few heuristics, including the structure of the test, the names of the variables in the test, or even the title of the test itself. Despite these helpers, it was still often difficult to pinpoint the precise SUT. Our analysis yielded the following types of systems under test:

→ *standalone functions*, (`parse_f64(..)` in [RP3]);
→ *composed functions*, (`deserialize(serialize(..))` in [JJ35]);
→ *mutating functions*, (`dt.replace(..)` in [PH12]);
→ *data structures*, (`IndexMap` in [RQ1]); and
→ *entire components*, (`Cluster` in [JJ11]).

Q1.2: What types of properties are tested?

The tests we encountered can be divided into three categories:

(1) those that verify that an SUT adheres to certain contracts,
(2) those that compare the SUT to some oracle, and
(3) those that define error and error recovery expectations.

(1) Contracts. Property-based tests that verify whether a SUT honours its contracts were the most common type of test in our dataset. We take the definition of contracts from the design-by-contract programming paradigm [13], where a POSTCONDITION expresses properties that are ensured by the SUT if certain preconditions are met by the input [RP1]:

[1] Determined by a combination of the number of downloads and GitHub stars.

[2] In digital form, clicking these references will lead to the implementation on GitHub.

	term	definition	component of test in Section 2.2
(1)	**test data**	the data with which a PBT's parameters are instantiated (if a PBT has multiple parameters, "test data" refers to all of them as a tuple)	`x`
	test domain	the set of all test data with which a PBT can be instantiated	all positive `Int`s
	generator	a function which produces the test domain (every PBT with parameters has a generator)	`@ForAll Int`
	custom generator	a generator which requires intervention from the developer (i.e., is not what the framework automatically provides by default)	`@ForAll @Positive Int`
(2)	**post-hoc filter**	a condition based on the test data which interrupts the execution of a test before the call to the SUT (used in the body of the test)	`x.isOdd()`
	SUT input	the data that is passed as input into the SUT	`Tag(x)`
	property domain	the set of all inputs that can be passed to the SUT for the property under test to hold	`@ForAll x: Int. Tag(x)`
	post-hoc constructor	a function from the test domain to the property domain (used in the body of the test)	`Tag`
(3)	**SUT**	the system under test	`sut(...)`
(4)	**assertion**	the component that causes a PBT to fail if (a part of) the property is not satisfied	`result.isValid()`

Table 1: Components of a PBT defined before the test is initialised (1), and before (2) and after (4) the call to the SUT (3).

```
assert sut(x) >= 0
```

The most commonly-occurring contract pattern in our dataset (as well as in Java/jqwik and Rust/proptest separately) was RoundTrip [RP46]. This contract dictates that when a function and its inverse are applied to an input, the result should be equal to the original input:

```
assert sutI(sut(x)) == x
```

An intuitive example of such a function pair is `encode` and `decode` [JJ51], or the `reverse` function from the introduction and itself.

A popular contract pattern, especially in Java/jqwik, was Mutation, which verified that an object was mutated correctly [RQ2]:

```
list.add(x)
assert list.contains(x)
```

The tests that we encountered with this pattern could often be considered to be checking an object's Invariant [PH29]:

```
assert list.size() <= list.capacity
```

Finally, a contract pattern we noted mainly in Python/Hypothesis was FunctionProperty. These tests verified that a function satisfies certain mathematical properties, e.g., additivity [PH4]:

```
assert sut(x) + sut(y) == sut(x + y)
```

(2) Oracles. Test oracles can be used as a sort of reference implementation, to which the SUT can compare all of its outputs:

```
assert sut(x) == oracle(x)
```

This can, for example, be useful when comparing a new, optimised version to a trusted version of some algorithm. We encountered tests which used different libraries as oracles [PH18] or compared two custom implementations to each other [RQ35], but also ones which were implemented *within the test itself* [PH13]. Usually, this was done when the expected output could be trivially computed, or could branch out into only a few cases:

```
assert sut(x) == (2 * x) + 18
```

(3) Errors. About 10% of the tests we analysed contained assertions about error-related behaviour. These tests either verified that an error was or was not thrown [RQ5] or made sure that a program could recover from an error [JJ12]:

```
assert sut(x) throws SUTException
assert sut(y) !throws SUTException
```

3.2 How are property-based tests implemented?

We identified four categories of PBT components[3] in open-source repositories, which we call *the anatomy of a property-based test*:

(1) those that act before the test is initialised,
(2) those that act before the call to the SUT,
(3) the system under test (SUT) itself, and
(4) those that act after the call to the SUT.

A list of these components, along with their definitions, is presented in Table 1, with references to examples from our tests in Section 2.2.

During our analysis, we observed several aspects related to the implementation of the analysed tests, summarised as follows:

O1 *Developers implement non-trivial testing logic.* Many of the tests we analysed went far beyond textbook examples, and contained, for example, a custom input generator for filtering or data construction [JJ24], an involved test oracle implementation [RP31], or properties that span numerous assertions [PH11].

O2 *Components exhibit limited modular separation.* The definition of the property domain is often conflated with the implementation of the test logic [PH41], and assertions are often conditioned on either the test data or the SUT's output [PH3].

O3 *The tests are not always easily comprehensible.* Regardless of the environment, we found that tests adhering to a clean, textbook structure are easy to comprehend. However, many of the tests we analysed use complex setups or assertion logic, making it difficult to distinguish between pre- and post-conditions, to identify which values flow into the SUT, or even to determine what the SUT precisely was [RP2].

Additionally, we noticed that Rust/QuickCheck was the most "minimalist" environment with respect to implementation. We hypothesise that this is because it is a port of the original QuickCheck

[3] Any distinct part of a test that plays a specific role in the test's execution or setup.

framework for Haskell, which emphasises simplicity and minimal boilerplate [4]. Additionally, Rust has the richest type system and can thus provide the most default information to a generator.

Q2.1: How is input generated and controlled?

Though most frameworks we examined offer extensive default generator strategies, almost all the tests we analysed had some form of developer intervention in the property domain creation. Customisation of default generators was fairly common, and typically involved implementing a separate generator or composing a default generator with additional filters or construction.

What was interesting though, is that despite custom generators already deviating from the default, *the test domain and the property domain often differed.* We tracked how data flowed from the test input up to the SUT call, and noticed that, across all environments, developers often implement post-hoc alterations to change the values that flow into the SUT.

Post-hoc constructors transform test inputs into ones that can be consumed by the SUT, and are commonly defined across all environments. For example, [JJ40] generates a normal double and builds a `AtomicDouble`, and [RQ4] generates a list and then loops through it to populate a map.

Post-hoc filters were used to ensure that a property's pre-condition was met by the generated inputs. This was done by using the framework's built-in way to discard the test [RQ17], making the test pass by default [RQ42], or modifying the input to match the pre-condition [RQ40]. Interesting cases of post-hoc filters included one that could only be applied *after the call to the SUT* [RQ50] and one which called an auxiliary function which had a post-hoc filter with the same condition as was in the custom generator [RP15]. The tests we analysed for Java/jqwik used notably less post-hoc filtering.

Q2.2: How are the properties asserted?

We encountered a large variety of complexity in how properties were asserted. While some tests had only a single assertion that captured the entire property [JJ34], many used a combination of assertions — sometimes to verify a combination of properties [RP34]. These assertions were on the same execution path [JJ37], guarded behind conditions based on the test input [PH58] or the SUT output [RQ5], hidden in auxiliary functions [PH14], or repeated many times in a loop [RP2]. Python/Hypothesis even used assertions for type checking [PH44].

Although auxiliary functions supporting assertion checking were found across all environments, they appeared most frequently in Python/Hypothesis tests. For instance, in the NumPy library, developers implemented helper functions such as `arrayAllClose` to compare two array-like objects for structural and element-wise equality within a given relative or absolute tolerance [PH44]. These auxiliary functions were often reused throughout the testing code. We hypothesise that some of these functions are necessary due to the lack of static types in Python.

Depending on the framework, assertions were implemented with an explicit assertion [JJ50], as a boolean expression to be returned from the test [RQ26][JJ17], or as an assertion error thrown if a condition was not met [JJ20]. Some tests even used a combination of these, making it more difficult to comprehend the test logic [RQ6].

4 THREATS TO VALIDITY

Because we are using qualitative research methods, our own experience with PBT had likely made its way into the interpretation of the dataset. To reduce the likelihood of a misinterpretation that would pose a threat to the validity of our results, we cross-checked every feature label between at least two researchers, and discussed any discrepancies until we reached agreement.

Additionally, due to our test selection strategy, our dataset is largely biased towards tests from popular open-source repositories. While the data aligns with our approach to identify the widest possible variety of PBT practices, we cannot generalise any distributions of these practices to a wider selection of tests. Future work could select a subset of the features we examined, and replicate the study with stratified or random sampling to provide such information.

5 RELATED WORK

Unlike this work, which studies patterns across multiple environments, most existing studies focus on PBT within a single one. Etna [15] is the exception, providing an evaluation platform for PBT approaches in both Coq and Haskell. Most other existing studies focus on Python's Hypothesis:

Tyche [9] is used to study how developers interact with the framework when offered visual feedback on their tests' effectiveness;
Corgozinho et al. [5] identifies the most commonly implemented properties and the features used to construct them;
Ravi and Coblenz [14] conducted a larger-scale empirical analysis which refines this property taxonomy and concludes that properties related to exception-raising, inclusion, and type checking are most effective at detecting bugs; and
Wauters and De Roove [16] analyse the challenges that developers face when using PBT for machine learning projects.

Finally, Goldstein et al.'s [8] interviews with OCaml developers reveal common challenges in writing property-based tests and show that developers prefer using PBT in "high-leverage" situations where it offers clear benefits over conventional unit testing.

6 CONCLUSION & FUTURE WORK

We found that PBT is used to test a large variety of systems, from standalone functions to entire components, and verifies three types of properties: adherence to a contract, equivalence with an oracle, and expected error behaviour. Developers combine customised generators with post-hoc alterations to control the property domain of their tests, and they write non-trivial logic to verify the properties, often with multiple assertions along various execution paths. Our preliminary results lead to these follow-up research questions:

- How do language features influence PBT framework design and the comprehension of property-based tests?
- What are the benefits and limitations of having a test domain that is different from the property domain?
- How does post-hoc alteration affect the effectiveness of a property-based test?

- Which PBT design choices lead to a more "ergonomic" way of writing property-based tests?

We believe that by investigating these questions, we can improve the effectiveness and usability of PBT across all environments.

ACKNOWLEDGMENTS

This paper is based on the work of Antonios Barotsis, Max Derbenwick, David de Koning, and Ye Zhao, conducted in partial fulfillment of the requirements for their Bachelor of Science degree in Computer Science and Engineering at the Delft University of Technology in June 2025. Their works are available online in the university's repository.

REFERENCES

[1] Maurício Aniche. 2022. *Effective Software Testing: A developer's guide.* Manning.
[2] Antonios Barotsis, Harald Toth, Sára Juhošová, and Andreea Costea. 2025. Property-Based Testing Across Different Environments in Open Source Repositories. https://doi.org/10.4121/08ecb1c5-fb8a-4a3d-8c91-a96bc7f28ec7
[3] Kathy Charmaz. 2014. *Constructing Grounded Theory* (2 ed.). Sage Publications.
[4] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP).* ACM, 268–279. https://doi.org/10.1145/351240.351266
[5] Arthur Lisboa Corgozinho, Marco Tulio Valente, and Henrique Rocha. 2023. How Developers Implement Property-Based Tests. In *International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 380–384. https://doi.org/10.1109/ICSME58846.2023.00049 ISSN: 2576-3148.
[6] Edsger W. Dijkstra. 1972. The humble programmer. *Commun. ACM* 15, 10 (1972), 859–866. https://doi.org/10.1145/355604.361591
[7] Barney G. Glaser. 1978. *Theoretical sensitivity: Advances in the methodology of grounded theory.* Sociology Press.
[8] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *International Conference on Software Engineering.* ACM, 1–13. https://doi.org/10.1145/3597503.3639581
[9] Harrison Goldstein, Jeffrey Tao, Zac Hatfield-Dodds, Benjamin C. Pierce, and Andrew Head. 2024. Tyche: Making Sense of PBT Effectiveness. In *Symposium on User Interface Software and Technology (UIST '24).* ACM, 1–16. https://doi.org/10.1145/3654777.3676407
[10] Rashina Hoda. 2024. *Qualitative Research with Socio-Technical Grounded Theory: A Practical Guide to Qualitative Data Analysis and Theory Development in the Digital World.* Springer Cham. https://doi.org/10.1007/978-3-031-60533-8
[11] Sára Juhošová, Andy Zaidman, and Jesper Cockx. 2025. Pinpointing the Learning Obstacles of an Interactive Theorem Prover. In *International Conference on Program Comprehension (ICPC).* IEEE, 159–170. https://doi.org/10.1109/ICPC66645.2025.00024
[12] Jorge Melegati, Kieran Conboy, and Daniel Graziotin. 2024. Qualitative Surveys in Software Engineering Research: Definition, Critical Review, and Guidelines. *IEEE Transactions on Software Engineering* 50, 12 (Dec. 2024), 3172–3187. https://doi.org/10.1109/TSE.2024.3474173
[13] Bertrand Meyer. 1992. Applying 'design by contract'. *Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279
[14] Savitha Ravi and Michael Coblenz. 2025. An Empirical Evaluation of Property-Based Testing in Python. 9, OOPSLA2 (2025), 412:3897–412:3923. https://doi.org/10.1145/3764068
[15] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C. Pierce, and Leonidas Lampropoulos. 2023. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). 7, ICFP (2023), 218:878–218:894. https://doi.org/10.1145/3607860
[16] Cindy Wauters and Coen De Roover. 2024. Property-based Testing within ML Projects: an Empirical Study. In *International Conference on Software Maintenance and Evolution (ICSME).* 648–653. https://doi.org/10.1109/ICSME58944.2024.00067 ISSN: 2576-3148.