Property Based Testing in Rust, How is it Used?

A case study of the ‘quickcheck‘ crate used in open source repositories

Max Derbenwick[1]

Supervisor(s): Andreea Costea[1]**, Sára Juhošová** [1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Max Derbenwick
Final project course: CSE3000 Research Project
Thesis committee: Andreea Costea, Sára Juhošová, Marco Zuñiga

Abstract

Property-based testing (PBT) is a method of verifying software correctness in which a property, a statement about the behavior of the program which should always hold true, is verified with a large number of arbitrary inputs. While it is a powerful method, properties can be complex and fail on obscure inputs, making them difficult to reason about. We qualitatively analyze a large number of property-based tests (PBTs) written with *Quickcheck for Rust* from public repositories, in order to gather insights about how PBT is utilized in practice. We find that the majority of analyzed PBTs verify by comparing against a known correct implementation, composing inverse operations to form an identity, or asserting a desired contract about a system's state. These findings offer valuable direction to the development of PBT, aiming to tailor PBT frameworks to developer needs and make them more natural to work with.

1 Introduction

Verifying the correctness of software is one of the most important stages of software development. However, despite its critical role, it is normally too difficult to guarantee correctness completely. As a result, numerous different techniques exist with varying tradeoffs between effort and guarantee, including unit testing, end-to-end testing, and even formal verification. Property-based testing (PBT) is a particularly versatile type of automated testing which tests large amounts of a program's functionality with little intervention from the developer [5]. It can also guide and complement formal verification, making it suitable for even more rigorous software verification demands [4]. It works by passing large amounts of random inputs, created by *generators*, through manually defined *properties*, formal statements about a program's behavior which should always hold for a certain class of inputs [11]. Properties can be devised in many ways, with common examples including properties which verify:

- that multiple operations which should be equivalent yield the same result,
- that two inverse operations applied successively yield the original input, or
- that the result of an operation matches another implementation which is known to be correct [16].

Despite the advantages of PBT, it is often a less chosen method of software verification, as properties are difficult to devise and often perplexing to reason about [13]. The latter issue can even be exacerbated when the property fails to hold for a strange input. *Shrinkers* help to mitigate the problem of large inputs by computing smaller versions of those inputs, in an attempt to find a smaller fail case. One way to combat the obscurity of PBT to make property-based tests (PBTs) smaller, which we consider in two ways:

- reducing the amount of logic and assertions in a PBT, which we refer to as horizontal decomposition, and
- reducing the scope of the functionality being tested, which we refer to as vertical decomposition.

In order to better understand this decomposition as well as gain valuable insights into improving property-based testing, we aim to investigate how and within what scope property-based testing is used in the real world. We chose *Quickcheck for Rust* as the testing framework and language to narrow in on, for isolated and specific research. More specifically, we will describe

- the themes which emerge in real-world properties,
- how properties are implemented,
- their role in the overall correctness guarantee, and
- in which cases they use explicit generators and/or shrinking support.

Quickcheck for Rust is a testing framework based on Haskell's original *Quickcheck* framework, the first real-world implementation of PBT [2]. The Rust language is quickly gaining popularity due to its modern design and compile time guarantees [12]. Contrary to other more traditional systems languages, it also offers zero-cost memory safety, making it highly appealing from a security perspective as well [12]. All in all, its growing prevalence in the field made it a great candidate for performing relevant research. Within the Rust ecosystem, *Quickcheck* is one of the two dominant PBT libraries, alongside *Proptest*.

In order to obtain our results, we investigate real-world repositories utilizing *Quickcheck for Rust*, aiming for at least 50 individual properties used for testing. We then qualitatively analyze these properties using open coding with hashtags and constant comparison, a technique for building a consistent dataset out of qualitative data [9].

We are performing this research in parallel to four other similar research endeavors, answering the same questions for different languages and frameworks. The other languages and frameworks being investigated are *Rust and Proptest* [1], *Haskell and QuickCheck* [17], *Java and Jqwik* [15], and *Python and Hypothesis* [7]. We pay extra attention to the work being done on *Proptest*, as the relationship between it and *Quickcheck* in the Rust ecosystem is important to consider.

The paper is presented in the following structure. First, the full methodology of the data collection is given in chapter 2. Then, the results are explained and presented visually in chapter 3. This is followed by a deeper discussion and analysis of the results in chapter 4. Chapter 5 then discusses threats to validity and responsible research considerations made during the research. Finally, chapter 6 draws final conclusions about PBT's role in software development with Rust, as well as future work to be done.

2 Methodology

In this section we detail our methodology of data collection, including both the selection of repositories and PBTs as well as the qualitative analysis methods used to build the dataset.

2.1 Repository Selection

In order to select PBTs for analysis, we first selected a set of public repositories which are representative of common practices in the Rust development ecosystem. We selected candidate repositories based on being reverse dependencies of *Quickcheck*, and having at least one PBT.

We then ranked candidate repositories such that the highest ranked ones would form this representative set. Both GitHub stars and number of downloads were considered, and we ultimately chose the latter. While GitHub stars often indicate a repository's popularity, they demand action from the user and are not required for usage of the repository. On the other hand, the number of downloads more accurately represents the extent to which the repository is used in other projects, which we believe to be a better indicator of a repository's contribution to the Rust ecosystem.

Searching reverse dependencies of Rust repositories is trivial using crates.io, where the reverse dependencies of *Quickcheck* can be listed and sorted by number of downloads. We selected repositories sequentially from this list, analyzing as many as possible within the scope of the project.

Accounting for Rust Workspaces

Rust offers a feature called workspaces, which allow for multiple *crates*, the Rust term for a package or library, to be bundled together as a single unit [14]. Up until this point, we have assumed that crates are analogous to repositories when browsing *Quickcheck*'s reverse dependencies. However, in practice, a crate may be only part of a repository which contains multiple crates. Thus, as we intend to analyze repositories, we selected every repository containing a selected crate, and skipped any selected crate belonging to a repository we have already fully analyzed.

2.2 Repository Analysis

Before analyzing specific PBTs, we collected basic information about each repository to give additional context to the dataset. The information collected includes

- the number of GitHub stars,
- the number of published versions,
- the number of reverse dependencies, and
- the number of tests, separated into traditional tests and PBTs.

The number of published versions and the number of reverse dependencies were both collected through crates.io. The number of traditional (non-PBT) tests was calculated through the following *grep* command:

```
grep -r '^\s*#\[test\]' .  | wc -l
```

Note that this is a rather naive approach to test counting, as it only counts the number of `#[test]` attributes, which does not account for macro-generated tests. Rust does not provide a way to count tests out of the box, and more advanced automated test counting is beyond the scope of this project. However, as macro-generated tests are infrequent, this heuristic provides a general order of magnitude of the number of traditional tests, which is sufficient for the research.

On the other hand, PBTs implemented with *Quickcheck* do not have the `#[test]` attribute, and thus were counted manually. Rust supports declarative macros, which allow similar code snippets to be repeatedly generated with small differences, including *Quickcheck* PBTs. Thus, as our analysis of PBTs is much more precise, we count them in two separate ways:

- the number of uniquely written PBTs, including tests which occur inside of macro definitions, and
- the number of real PBTs in the test suite after macro expansion.

With the repository information collected, we then selected PBTs from the repository to analyze. Note that these PBTs correspond to the first mentioned counting method: uniquely written PBTs, even within macros. To ensure that a diverse set of repositories could be analyzed within the scope of the project, the number of PBTs to analyze per repository was limited to ten. If a repository has more than this amount, ten PBTs were selected at random to analyze.

Note that in reality these rules were not followed exactly. A limit on PBT count was only imposed after more thoroughly analyzing a single repository, and the decision to keep the additional data was deemed more valuable than removing it for exactly equal representation. Similarly, some repositories fell slightly short of ten PBTs analyzed due to time constraints.

2.3 PBT Analysis

For each PBT, we investigated both the test itself and and the system under test (SUT), in order to gain a general understanding of the functionality being tested and how the test was implemented. Then, we qualitatively analyzed the PBT using open coding techniques such as hashtag coding and constant comparison [9], as well as answering a set of questions about the PBT which seek to describe its behavior and implementation. These questions include

- how many assertions the PBT makes,
- whether its assertions are functionally independent,
- how many times it invokes the SUT, and
- whether they use a custom generator and/or shrinker.

In the spirit of open coding, the dataset started off with a small number of only baseline questions and codes inspired by a guide on creating properties [16], which was slowly expanded as new discoveries were made. The central part of this baseline dataset is a set of built-up hashtag codes categorizing a test's *intention*, which characterizes the way in which the test composes operations of the SUT in order to form a property which holds. Throughout the data collection, the following intentions were identified and incorporated:

- **DifferentPaths:** Verifies the equivalence of two transformations that should give identical results when performed on the same input.
- **RoundTrip:** Verifies that an operation composed with its inverse when applied to an input gives the same output.

```rust
fn reverse<T>(list: Vec<T>) -> Vec<T> {...}

quickcheck! {
    fn reverse_ident(list: Vec<i32>) -> bool {
        list == reverse(reverse(list.clone()))
    }
}
```

Figure 1: A PBT testing the `reverse` function

- **Invariant:** Verifies the invariance of a component of a state which should not change when the state is transformed in some way.
- **Idempotence:** Verifies that applying an idempotent operation n>1 times is equivalent to applying it once.
- **StructuralInduction:** For recursive problems, verifies that the result returned from an operation upholds some recursive contract.
- **HardToProveEasyToVerify:** Verifies the correctness of the result returned by the SUT with some 'easy' verification algorithm.
- **TestOracle:** Compares the result returned by the SUT to the one returned by an oracle, which is a known correct implementation.
- **StateContract:** Verifies that, after some operations, the state of the SUT upholds a contract defined by or derived from its specification.
- **KnownOutput:** Given some set of inputs, the output can be trivially derived from the input, requiring no more than a few cases.
- **NoErrors:** Ensures that the given operation does not throw some kind of error on any input.

After the first analysis of each PBT, we performed a second iteration in which all gaps in the dataset generated from later discoveries were revisited and bridged. For the full set of questions asked for each PBT, refer to [Appendix reference].

PBT Analysis Example

Figure 1 shows an example of a PBT to be analyzed. The `reverse` function, whose implementation is hidden, takes an arbitrary list as input and returns a reversed version. The PBT `reverse_ident` verifies that for any list of integers, reversing the list twice is equivalent to the original list. This PBT would be coded as follows:

- **Assertion count:** 1
- **Calls to SUT:** 2
- **Intention:** *RoundTrip*
- **Input type(s):** List(Numerical)
- **Custom generator:** No

Note that this is a small snippet of this PBT's coding, and that codings in the dataset contain many more fields and a deeper analysis.

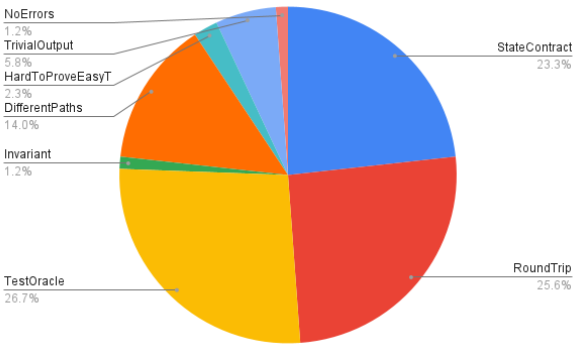

Figure 2: The distribution of test intentions in the dataset

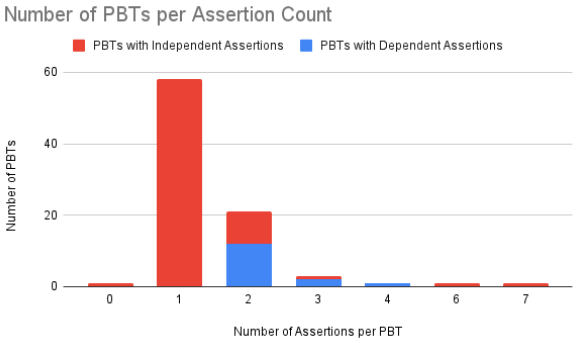

Figure 3: The number of PBTs observed for each count of assertion

3 Findings

In the end, we analyzed a total of 86 PBTs from 13 repositories. In this section, we report our most important and relevant findings from the gathered data. The full dataset from which these findings were derived is published [8].

Firstly, the test counts for each analyzed repository are shown in Table 1. As expected, we see that traditional testing is most prevalent, and that property-based tests fall short by almost an order of magnitude in most cases. On average from this dataset, property-based tests make up 24.76% of the total number of tests in a repository.

Proceeding to the PBTs themselves, the distribution of test intentions across the dataset is shown in Figure 2. We can see that *TestOracle*, *RoundTrip*, and *StateContract* occurred roughly as often as one another, and collectively make up over 75% of the analyzed PBTs. On the other hand, *Idempotence* and *StructuralInduction*, both of which are predetermined intentions derived from prior research [16], did not appear at all in the dataset. Furthermore, few anomolies were observed in the test intentions, with only *Invariant* and *NoErrors* being created as single-instance intentions.

Beyond test intentions, we have also grouped PBTs by how many assertions they make, and whether these assertions would be independent enough to meaningfully perform horizontal decomposition. Figure 3 shows this grouping in terms of number of PBTs observed for each assertion count, further classified into dependently- and independently-asserting

Repository	Regular Test Count	PBT Count	Expanded PBT Count
indexmap	106	33	33
time	506	61	61
regex	398	5	5
itertools	140	203	203
memchr	920	24	115
byteorder	32	3	30
http	132	1	1
h2	55	1	1
crc32fast	1	5	5
flate2	62	5	5
num-bigint	205	47	47
unicode-segmentation	9	5	5
bumpalo	78	19	19

Table 1: The number of tests per analyzed repository

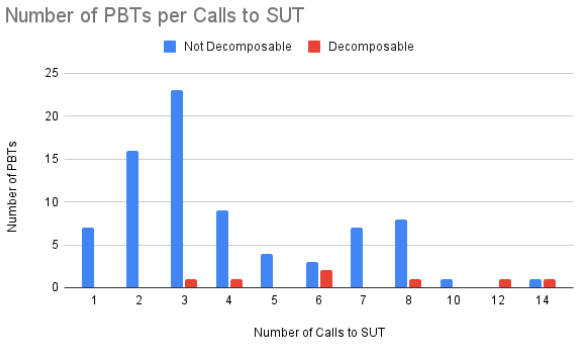

Figure 4: The number of properties for each count of SUT calls

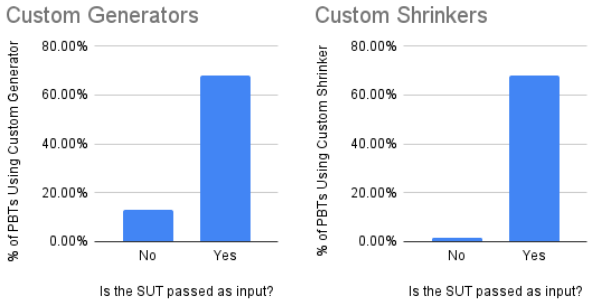

Figure 5: How often custom generators and shrinkers are used when the SUT is passed as input

PBTs. The Figure demonstrates that PBTs heavily favor a small number of assertions, and that a significant number of PBTs with more than one assertion could be split into sub-properties.

We additionally categorized PBTs by the number of times in which they invoke the SUT. Following this, we labeled the PBTs by which could reasonably undergo a naive approach to vertical decomposition wherein we simply split SUT invocations into separate properties where possible. The results of this analysis are shown in Figure 4. We can see that the majority of analyzed PBTs are not decomposable in this way, but that it becomes slightly more probable when having more than a few SUT invocations.

3.1 Additional Findings

On top of the general results, we identified a number of patterns which are worth discussing and offer valuable insights into how *Quickcheck* is being used.

In the dataset, we found that 29.07% of PBTs use custom generators for its input, and only 20.93% of PBTs use custom shrinkers. However, we found a strong correlation between using custom generators/shrinkers and passing the SUT as input to the test. That is, generated SUT instances often use custom generators and shrinkers. This correlation is justified by Figure 5, which shows the percentage of PBTs using custom generators and shrinkers, firstly when the SUT is not passed as input, and secondly when it is.

Secondly, we found a strong correlation between a test's intention, and whether or not it makes assumptions. We define assumptions in property-based testing as referring to conditions on the input which are required to hold in order for the test to be valid. If an assumption does not hold, the test simply returns a dismissive `true`, or in some other way discards the test without failing. We found that while a total of 23.26% of analyzed PBTs make assumptions, these tests belong entirely to the *TestOracle* and *RoundTrip* intentions, with a majority concentration in *TestOracle*. This is displayed visually in Figure 6, which shows the percentage of PBTs which make assumptions, bucketed per intention.

4 Discussion

From the repositories analyzed and data gathered, we found that PBT remains a more seldom used testing method, even in repositories which make significant use of it. Of the PBTs analyzed, we saw that the majority fall into a small number of intention categories. Examples of PBTs that fall into these categories are as follows:

TestOracle:

- `check_against_baseline` - Checks a fast crc32

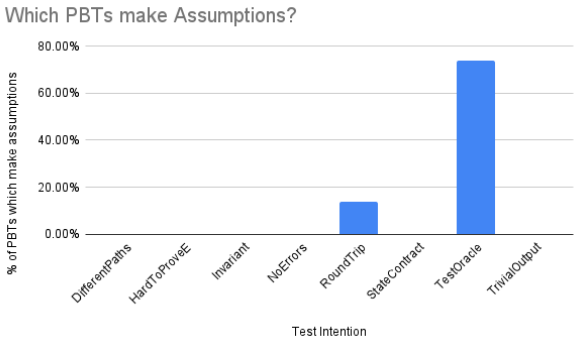

Figure 6: How often PBTs make assumptions per test intention

checksum implementation (the SUT) against a known slow implementation.

- `test_shr_unsigned` - Tests a big integer shift-right operation (the SUT) against the default system implementation.

RoundTrip:

- `quickcheck_signed_conversion` - Converts a big integer to and back from a string of some radix.
- `hpack_fuzz` - Encodes and then decodes an HTTP header.

StateContract:

- `insertion_order` - Ensures the insertion order of elements is maintained in the SUT's state.
- `prop_state_read_write_pattern_ids` - Ensures that IDs added to the SUT are able to be read back.

PBTs with descriptions such as those are frequent in the dataset, and all have in common that they are simple and apparent use cases of PBT, without complex logic. While most PBTs in the dataset share this simplicity, we observed that chasing it seems to evoke property intentions which fit more naturally to the SUT. Of course, it is commonplace to implement inverse operations, and SUTs often claim to guarantee a property about their state, making *StateContract* tests a natural choice. The prevalence of test oracles is particularly interesting, and we believe it to be directly tied to the types of repositories which tend to utilize *Quickcheck*. The repositories analyzed include custom allocators, fast implementations of low-level and cryptographic operations, and widely applicable utility libraries. Many of these problems have slow or limited reference implementations, which make good test oracles for certain inputs.

We observed a similar tendency for simplicity in horizontal and vertical PBT complexity. We found that the analyzed PBTs preferred to make fewer assertions and fewer calls to the SUT. The assertion count is generally decreasing, while the number of SUT calls has a mode of three, as multiple SUT calls are usually necessary to create a viable property. Little correlation is found between the assertion count and potential for horizontal decomposition, and likewise between the number of calls to the SUT and the potential for vertical decomposition. However, it does remain that a significant number of PBTs from the dataset can be horizontally decomposed, and that a select few can be vertically decomposed using the naive approach of splitting independent SUT invocations into different tests.

In our findings, we demonstrated a significant correlation between whether or not the SUT is passed as input, and the presence of custom generators and shrinkers. In the data, we found that the majority of non-SUT input consists of primitives, for which *Quickcheck* has built-in generators. Not only does this closely align with the low-level operations being implemented by these repositories, but also supports the aforementioned correlation. As the SUT is nearly always a non-primitive type defined within the repository (but not always, as Rust supports remote implementation on types), a custom generator would be required in order to supply it as input to PBTs. Furthermore, *Quickcheck* states that the default shrinking behavior of a custom arbitrary type is not to shrink at all [3]. Thus, programmers wishing to pair a custom generator with any kind of shrinking must do so manually. This supports the tight relationship between SUT input and both custom generators and shrinkers, and by extension the relationship between the presence of custom generators and shrinkers themselves.

Finally, we previously showcased a correlation between the test intention and whether that test makes assumptions, and found that test oracle PBTs make extensive use of them. We believe that this is due to the nature of the test oracles, which we found to often be naive implementations of the same problem, but with speed or input space limitations. Thus, many of these tests used assumptions to fit the input space to what is viable for the test oracle. For example, the aforementioned `test_shr_unsigned` test uses a system shift-right implementation which cannot accept nearly as large of inputs as the SUT. It is also worth noting that while *Quickcheck* does not support input space filtering aside from assumptions, we observed many tests limit numerical input space by using numerical datatypes with smaller range as input.

4.1 Framework Comparison

First a foremost, a number of differences were identified between *Quickcheck* and *Proptest*, and how they are used by Rust repositories. *Proptest* is usually known for its versatility and feature-richness, including an adaptable shrinking model that requires little programmer intervention [6]. On the other hand, Quickcheck is known for being signficiantly faster, especially regarding shrinking [6]. Of course this comes at a cost, and indeed *Quickcheck's* stateless shrinking model requires more frequent manual intervention and does not necessarily obey constraints imposed by the generator unless done manually [10]. This hypothesis is strongly supported by our data, where we found that *Proptest* PBTs are generally more complex, and used in higher level repositories than *Quickcheck*. Furthermore, we found that *Quickcheck* repositories tend to have far more PBTs, with an average of 31.69 PBTs per repository, while *Proptest* repositories have an average of only 9.25 PBTs per repository. This seems to confirm the claim that speed is far less of a concern when using *Quickcheck*.

4.2 The *NoErrors* Intention

The *NoErrors* intention is particularly interesting due to its applicability. This intention was employed only once in the dataset, and was in fact created to fit that one test. This shows us that this is a seldom utilized use case of PBT. The test which was labeled with this intention is described as follows:

- `vec_resizes_causing_reallocs` - Continuously invokes 'realloc' with different parameters, and simply verifies that they do not panic (throw an error), as no other assertions are made by this test.

In fact, this is a universally applicable type of property, as any SUT can be invoked with many arbitrary inputs, with the only meaningful property being 'does not crash.' Arguably, this method can be quite powerful; with enough inputs, all program-crashing bugs can be detected. However, such tests like this have a number of limitations:

- The intended behavior may be to crash in certain cases, and thus the input space would have to adequately filtered.
- Unintended runtime errors which crash the program are only a small subset of all bugs, and all others would be undetected by this type of test.

Given that this test intention is so infrequent, we conclude that the speed penalty and effort required to implement such a test to be seldom worth it. These tests would have to adequately filter inputs to those which should not crash the SUT, while still providing enough inputs to feasibly trigger unintended runtime errors. That being said, it makes the most sense in 'resilient' systems, which should be able to withstand any input without failing, such as the allocator implementation in which our test was found.

5 Threats to Validity and Responsible Research

Despite the unparalleled rigor of this research, it is of course still subjective in nature, and there are a number of biases worth mentioning.

5.1 Reproducibility of Methodology

In general, we designed the methodology to be specific and rigorous so that it is reproducible. However, due to random PBT selection in large repositories, results may differ when repeating the methodology. Furthermore, inconsistencies in the application of the methodology also hinder authentic reproducibility. Namely, as we stated previously, we were not able to follow the 'ten PBTs per repository' rule exactly, leading to slight over- and under-representation in some repositories. Nevertheless, a large number of PBTs were analyzed, and thus we believe that the statistical significance of our findings is likely to persist in a second iteration, given enough PBTs.

It is also worth noting that the qualitative coding process is subjective, and bias on part of the researchers is always possible. Knowledge of the language, *Quickcheck*, and the technology specific to the repositories analyzed, all make room for potential bias and misinterpretations. These biases can influence the resulting dataset, and threaten the reproducibility of the methodology as a whole. We aimed to circumvent this bias through multiple iterations of analysis per PBT, and qualitative coding techniques such as constant comparison [9].

5.2 Biases in PBT Representation

Beyond the reproducibility, the potential for systemic bias also exists in the methodology. For example, we chose to only analyze most downloaded repositories, as they represent a large part of the current Rust ecosystem. However, they are not exhaustive, and a large subset of developers and practices may not have been encountered with this approach. In particular, three of the repositories we analyzed are primarily maintained by *BurntSushi*, the same author of the *Quickcheck* library itself. This fact introduces a degree of bias, as a large portion of the data now not only comes from the same author, but also represents an informed and idealistic use of *Quickcheck*, that may not be representative of how the remainder of the community uses it.

5.3 Software Licenses

In this research, we analyzed source code from public repositories. To ensure that the research was performed legally, we verified that all analyzed repositories were licensed under terms that permit analysis for research purposes, including the MIT license and Apache 2.0 license. We adhered to the conditions of these licenses, and ensured that no proprietary or confidential code was included.

6 Conclusions and Future Work

In the end, we have unveiled a number of insights into property-based testing with Rust and *Quickcheck*. With this information, we answer our original questions:

- **What themes emerge in real-world properties?** We found that most properties in *Quickcheck* are based on test oracles, composition of inverse operations, or verifying desired contracts about the SUT's state.
- **How are properties implemented?** We saw that most properties are rather small, making few assertions and calls to the SUT. Most larger properties exist because of dependent assertions or to avoid redundancy. We also observed that PBTs usually pass only primitives or the SUT as input, making assumptions to filter the input space.
- **What is PBT's role in correctness guarantees?** As expected, we saw that PBT still plays a small role when compared to other testing forms, making up on average only 24.76% of the total tests in a repository. Nevertheless, repositories using *Quickcheck* make substantial use of PBT with an average of 31.69 PBTs per repository, with substantial coverage of the SUT even on their own.
- **When do PBTs use explicit generators/shrinkers?** We found that custom generators are used primarily when the SUT is passed as input to the test, as primitive inputs tend to be used in most other cases. Because *Quickcheck's* default behavior with custom arbitraries is

not to shrink, we found as well that custom generators are most often coupled with custom shrinkers.

We additionally showed that in many cases, decomposition of properties, both horizontally and vertically, would be feasible. As stated earlier, doing so could make properties more granular and easier to reason about. However, *Quickcheck* can make this somewhat more difficult for complex input types, as it does not shrink by default, leading to obscure fail cases.

6.1 Future Work

The motivation of this research was to give a clearer direction to the future of PBT by identifying the most important aspects of how it is used in practice. Thus, the results of this research open several opportunities for further development of PBT and PBT analysis.

Firstly, the biases of this research could be tackled by repeating the methodology for a wider variety of PBTs with a richer repository selection method. Doing so could reveal insights about PBTs in both high and low profile projects, as well as a wider variety of domains. Perhaps there exist some higher level repositories which also use *Quickcheck*.

Secondly, an extension of this research could be done that investigates specifically the relationship between PBTs and their respective SUTs. This could reveal some better insights related to vertical decomposition, as the goal is to better isolate components of the SUT when building properties. We, on the other hand, investigated vertical decomposition at only a surface level. We determined the potential for vertical decomposition by analyzing the feasibility of splitting the literal SUT invocations into multiple properties without losing functionality.

Finally, this research could motivate an attempt at designing a method for horizontal and/or vertical property decomposition. Using the insights we have gathered, a number of common cases of PBTs could be investigated, and decomposition could be generalized across them. Of course, this is a substantial undertaking on its own which would require far more than an understanding of PBT's usage in practice, but nevertheless such an understanding offers valuable direction to the endeavor.

References

[1] Antonios Barotsis. *Property-Based Testing in Open-Source Rust Projects: A Case Study of the proptest Crate*. Bachelor Thesis, Delft University of Technology, 2025.

[2] Valentin Bogad. The properties of quickcheck, hedgehog and hypothesis. https://seelengrab.github.io/articles/The%20properties%20of%20QuickCheck,%20Hedgehog%20and%20Hypothesis/, Jan 2024.

[3] BurntSushi. quickcheck. https://docs.rs/quickcheck/latest/quickcheck/index.html, n.d.

[4] Zilin Chen, Christine Rizkallah, Liam O'Connor, Partha Susarla, Gerwin Klein, Gernot Heiser, and Gabriele Keller. Property-based testing: Climbing the stairway to verification. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2022, page 84–97, New York, NY, USA, 2022. Association for Computing Machinery.

[5] Jesper Cockx. An introduction to property-based testing with quickcheck. https://jesper.sikanda.be/posts/quickcheck-intro.html, Dec 2020.

[6] Proptest contributors. Differences between quickcheck and proptest. https://proptest-rs.github.io/proptest/proptest/vs-quickcheck.html, n.d.

[7] David de Koning. *Property-Based Testing in Practice using Hypothesis: In-depth study on how developers use Property-Based Testing in Python using Hypothesis*. Bachelor Thesis, Delft University of Technology, 2025.

[8] Max Derbenwick, Harald Toth, David de Koning, Antonios Barotsis, Ye Zhao, Andreea Costea, and Sára Juhošová. Property-based testing in the wild! 4TU.ResearchData, 2025. doi: 10.4121/368f63ab-10fc-4603-a15a-bde25e72e778.

[9] Rashina Hoda. *Qualitative Research with Socio-Technical Grounded Theory*. Springer Cham, Cham, Switzerland, September 2024.

[10] David MacIver. Integrated vs type based shrinking. https://hypothesis.works/articles/integrated-shrinking/, Dec 2016.

[11] David Maclver. What is property based testing? https://hypothesis.works/articles/what-is-property-based-testing/, May 2016.

[12] Yuliia Panasenko and Mykhailo Maidan. Rust market overview: reasons to adopt rust, rust use cases, and hiring opportunities. https://yalantis.com/blog/rust-market-overview/, Jun 2025.

[13] Nataly Rocha. Diving into property-based testing with js - part 3. https://www.stackbuilders.com/insights/diving-into-property-based-testing-with-javascript-part-3/, Apr 2024.

[14] Carol Nichols Steve Klabnik and Chris Krycho. *The Rust Programming Language*. No Starch Press, 2022.

[15] Harald Toth. *Property-Based Testing in the Wild!: Exploring Property-Based Testing in Java: An Analysis of jqwik Usage in Open-Source Repositories*. Bachelor Thesis, Delft University of Technology, 2025.

[16] Scott Wlaschin. Choosing properties for property-based testing. https://fsharpforfunandprofit.com/posts/property-based-testing-2/, Dec 2014.

[17] Ye Zhao. *Property-Based Testing in the Wild!: A Study of QuickCheck Usage in Open-Source Haskell Repositories*. Bachelor Thesis, Delft University of Technology, 2025.