On the Evolution of Python Test Cases into Property-based Tests

Cindy Wauters
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
cindy.suzy.wauters@vub.be

Ruben Opdebeeck
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
ruben.denzel.opdebeeck@vub.be

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.be

***Abstract*—Conventionally, unit tests exercise their unit under test on predetermined input to validate its behavior. The advent of property-based testing frameworks has led to unit tests that exercise the unit under test on randomly-generated inputs, for which the unit's behavior has to satisfy developer-specified invariant properties. The promise of increased coverage and stronger validation may lead developers to evolve existing conventional unit tests into property-based ones. This paper reports on the results of an empirical study of 257 property-based unit tests (PBT) from 64 open-source repositories that have evolved from a conventional unit test. The study examines each PBT-introducing commit for change patterns, and for the type of features of property-based testing frameworks that are adopted. Next, it investigates the impact of PBT adoption by running test cases to compute changes in code coverage and test failures. Finally, the study investigates the evolution of the newly-introduced PBT by comparing its first to its most recent version. The study's findings include that 37 out of 257 evolutions required no changes to the body of the test. When changes are required, these are most likely to target setup code followed by oracle code. Over half of the studied PBTs use off-the-shelf input generators, yet see an improvement in code coverage by an average of 4 statements. Additionally, for 5 test cases, the PBT found a counterexample where the original unit test passed. Finally, 80% of the evolved test cases are still present in the current version of the project.**

***Index Terms*—Empirical study, software testing, property-based testing, code coverage**

I. Introduction

Conventionally, unit tests take an example-based form in which the unit under test is exercised on predetermined input. For a unit test that should be run with several inputs, parameterized unit testing [35] frameworks support substituting the example input by a parameter for which the developer provides a predetermined collection of values. When enumerating these input values becomes intractable, property-based testing [6] frameworks support generating them randomly according to a developer-specified *input generation strategy*. For each of the generated inputs, the property-based test (PBT) will check that the behavior of the unit under test satisfies a developer-specified *invariant property*.

The QuickCheck framework [6] for the Haskell programming language was the first to support property-based testing. Since then, more than 35 property-based testing tools [18] have been introduced for various languages, such as Java[1], Scala[2], Python[3] and JavaScript[4]. Proposing domain-specific input generation strategies and invariant properties, research initiatives have brought property-based testing to challenging application domains such as telecom software [2], distributed synchronization services [19], neural networks [33], and cyber-physical systems [13].

Despite these successes, property-based tests often constitute only a fraction of the test suite. According to a 2023 JetBrains survey [22], the most common PBT framework for Python, Hypothesis, was only used by 5% of Python developers. To bring the benefits of property-based testing to more projects, researchers have proposed tool support for creating property-based tests [16], [38]. However, there is a gap in the research when it comes to the evolution of existing test cases into property-based tests. Therefore, in this paper, we investigate these evolutions, their impact, and the maintenance of the evolved test cases. By doing so, we aim to provide valuable insights for testers, tool builders, and researchers alike. With this paper, we make the following contributions:

- We collect a dataset of 257 test case evolutions across 64 open-source GitHub repositories written in Python. Our data collection targets Python projects as, at the time of writing, it is one of the most popular programming languages [21]. We provide a replication package [40] containing the resulting dataset and the implementation of all subsequent analyses.
- An empirical analysis of the changes required to rewrite a conventional unit test into a property-based test, as well as whether test cases evolved afterwards.
- An empirical analysis of the impact of these evolutions on the system under test in terms of changes in code coverage and test failures.

The remainder of this paper is structured as follows. Section II provides a motivating example and introduces the research questions. Section III details the dataset of real-world test evolutions we will be working with. Then, Section IV, Section V, and Section VI present our research questions. For each of the three research questions, we detail the research

[1]https://github.com/jqwik-team/jqwik

[2]https://github.com/typelevel/scalacheck

[3]https://github.com/HypothesisWorks/hypothesis

[4]https://github.com/dubzzz/fast-check

```
def test_encode_decode():
    txt = "hello world"
    assert decode(encode(txt)) == txt
```

Listing 1: An example-based test for `encode` and `decode`

```
import pytest

testdata = ["hello world", "test -", "(/test)"]
@pytest.mark.parametrize("txt", testdata)
def test_encode_decode(txt):
    assert decode(encode(txt)) == txt
```

Listing 2: A parameterized unit test for `encode` and `decode`

method, the findings, and present an objective analysis of these findings. Section VII discusses the actionable insights gained from the findings across the research questions. Here we also describe the threats to validity and their mitigation. In Section VIII we position our work and compare it to current research into property-based testing.

II. Motivating Example and Research Questions

Test cases may evolve over time. To illustrate such an evolution, consider an `encode` function and its inverse `decode`. Listing 1 depicts an example-based unit test (EBT) checking that decoding the encoding of `"hello world"` results in the same string. This unit test validates the behavior of the unit under test for a single developer-provided example string.

Using Pytest [23], the most common Python testing framework, the example-based unit test from Listing 1 can be generalized into a parameterized unit test [36]. Listing 2 depicts an example generalization in which a parameter `txt` substitutes for the earlier example input. Three possible values have been provided for the parameter, thereby validating the behavior on predetermined strings that contain special characters.

Using Hypothesis [26], [27], the most popular property-based testing framework for Python, the test case can be generalized further. Listing 3 depicts the resulting property-based test, specifying that the composition of decoding after encoding is the identity operation for all strings generated using `st.text`. Hypothesis will automatically generate multiple strings, each of which will be encoded and decoded and compared to the original string. This way, the property-based unit test might uncover defects for edge cases the developer did not enumerate in the example-based nor in the parameterized version of the unit test.

```
from hypothesis import given, strategies as st

@given(st.text())
def test_encode_decode(txt):
    assert decode(encode(txt)) == txt
```

Listing 3: A property-based test for `encode` and `decode`

However, it can be difficult for developers to come up with invariant properties to test [14]. The Hypothesis Quickstart [43] hints at existing parameterized unit tests (PUTs) and existing sources of randomness in a test suite for inspiration. However, to our knowledge, no research has looked into real-world test case evolutions such as the one described above. In this paper, we answer the following research questions:

- **RQ1: How do conventional unit tests evolve into property-based tests?** In this research question, we investigate real-world commits for the changes required to rewrite a conventional example-based unit test or parameterized unit test into a property-based one.
- **RQ2: What is the impact of adopting property-based testing?** To investigate the impact of these test case evolutions, we compute the difference in code coverage before and after the rewrites as well as the test failures.
- **RQ3: How are property-based tests maintained?** To assess the effort required to maintain the resulting property-based tests, we investigate whether there are changes made to the test cases after their rewriting into a property-based test, or if they are removed.

III. Dataset

In order to study real-world unit test evolutions, we first need a dataset of open-source repositories that contain at least one property-based unit test (PBT) that started as either an example-based unit test (EBT) or as a parameterized unit test (PUT). Our data collection will start from Python repositories that have a dependency on the Hypothesis [27] framework for property-based testing. While other empirical studies have collected such repositories before (e.g., [32], [39]), we collect our own to ensure the dataset contains ample instances of test case evolutions.

1) Data collection: We first use the GitHub API to find open-source Python repositories that have a dependency on Hypothesis. In order to exclude toy projects, we only include projects that have at least three stars. We also filter out larger projects with more than 10 stars. While these projects can be valuable to study, they are also difficult for an outsider to understand and reason about. We do not consider those projects suitable for a deep manual investigation, as required for this empirical study section. In total, we considered 1388 repositories that have a dependency on Hypothesis and satisfy our inclusion criteria.

Next, we use PyDriller [34] to find all PBTs in a repository that were once either an EBT or a PUT, and are now a PBT. To this end, we rely on the presence of `@given` decorators in a file with a PBT. The GitHub API alone does not suffice, as noted by prior empirical studies into real-world Hypothesis usage (e.g., [8], [32], [39]), due to the existence of other libraries named Hypothesis. The `@given` decorator in a test file, in contrast, denotes the entry point of a Hypothesis-based PBT. If none exists, the repository does not contain a PBT, and is excluded. If we find at least one PBT in a file, we investigate all previous versions of the file. In case the PBT exists in the initial commit of the file, PBTs have always existed in that

	# LOC Python	# evolved test cases	# commits
Mean	7575	4.02	923
Std Dev.	17147	6.66	1996
median	2607	2	294
1st Quar.	1178	1	121.5
3rd Quar.	5346	4	680.5
Min	164	1	15
Max	109516	39	10274

TABLE I: Characteristics of the 64 repositories investigated in this paper. Lines of Code computed by Cloc [10].

file, and we do not consider this file relevant for investigating test case evolution. In case there is a previous version of the file in which the PBT did not exist, while it does in a newer version, we manually investigate the last version before the introduction with the first version after the introduction. We perform this step manually to ensure that we also retrieve evolutions of tests that have been renamed. Additionally, we take note of whether the transformation in question is from an EBT to a PBT, or from a PUT to a PBT. We recognize PUTs as unit tests adorned with the `@pytest.mark.parametrize` decorator. This renders their recognition purely syntactic.

2) Characteristics of the dataset: Out of the 1388 Hypothesis-dependent GitHub repositories initially discovered, 575 have at least one PBT present. Upon filtering for test case evolutions, we identified 64 repositories that feature at least one evolution from an EBT or PUT into a PBT —amounting to 257 PBT *before-after mappings*. For each PBT, we have a mapping from the test *before the PBT introduction* to the test *after the PBT introduction*. For a minority of PBTs, the test before its introduction was split into multiple PBTs. On the other hand, some tests were merged into a single PBT. In these cases, we consider the same PBT multiple times (each time with a different mapping). Our 257 before-after mappings therefore consist of 252 unique tests before PBT introduction and 250 unique PBTs. Of the total 257 test case evolutions, 182 test cases follow the EBT to PBT evolution, whereas 75 follow the PUT to PBT evolution. Table I presents summary statistics of the distribution of the data across the most recent version of the repositories. Included are the lines of code in Python (blank lines and comments not included), the number of investigated test case evolutions, and the total number of commits per repository.

IV. RQ1: How do conventional unit tests evolve into property-based tests?

RQ1 investigates how PBTs can evolve out of conventional test cases. To this end, we manually investigate the 257 aforementioned real-world before-after mappings in the dataset.

A. Research Method

1) Changes to the test case body: First, we consider *how* test cases are updated when they evolve into a PBT. To this end, we categorize all changes to the *body* of the test case into the following categories (four per type):

- Changes to the *oracle* code: non-semantic changes, semantic changes, additional oracles, removal of oracles.
- Changes to the *setup* code: non-semantic changes, semantic changes, additional setup code, removal of setup code.

To delimit *test oracle* code, we follow the definition of Barr et al. [3]. It is the part of the code that decides whether the system under test behaves as expected. This does not necessarily mean that only (nor all) the code in an assert statement is part of the oracle code. Code that defines the oracle is included too.

As *setup code*, we consider those parts of the body of the test case that create or retrieve values through which the behavior of the system under test will be validated. In other words, it sets up the system under test. This is similar to the definition of a test fixture by Meszaros [28]. This is sometimes also known as test context. However, we only consider the code inside the body of the test case.

As a *non-semantic* change to the body of a test case, we consider straightforward refactorings such as changing a constant to a value generated by the input generation strategy. For *semantic changes*, in contrast, we consider anything that changes the semantics of a test case (e.g., extra calculations within existing code). An addition or removal of setup code means any lines added removed, that can be considered setup code. An addition or removal of oracle code is the addition of, for example, a new assert statement, or the removal of an old one.

A test case can evolve into a PBT through several of these types of changes. To illustrate, consider Listing 4 which depicts one of the test cases in our dataset before and after evolving into a PBT. Line 2, as well as the `r.tolerance` in line 3 is considered the setup code: these lines define what will be tested later. The assert statement, excluding `r.tolerance`, is oracle code. This test case changes in two different ways. First, a non-semantic change to the setup code: on line 10 and 11, we see a renaming of `r` to `eu`. Additionally, hardcoded values `R`, `3`, and `2` are replaced with the parameters

```
1  def test_tolerance_sign_EBT():
2      r = R.from_min_max(3, 2)
3      assert r.tolerance == 0.2
4
5
6  @given( unit=st.sampled_from((U, I, P, R)),
7          a=st.floats(allow_nan=False),
8          b=st.floats(allow_nan=False))
9  def test_tolerance_sign_PBT(unit, a, b):
10     eu = unit.from_min_max(a, b)
11     assert eu.tolerance >= 0
```

Listing 4: A real test case before and after becoming a PBT. Some simplifications applied for readability. Case taken from https://github.com/wese3112/eecalpy

that will now be generated data. Second, a semantic change in the oracle code: as the values are not hardcoded anymore, an exact expected value is not known anymore, and the oracle needs to be updated. The change from == to >= makes this a semantic change.

Furthermore, it is also possible that the test case changed beyond recognition when it was rewritten into a PBT. During our manual investigation, we mark such test cases as replaced, rather than updated, if we cannot recognize it within the PBT anymore. Some large-scale studies into test evolution have used thresholds on automatically-computed similarity metrics instead (e.g., 66% used in [5] and [11]), but these make less sense when applied to small test cases instead of large files. We therefore rely on human judgement of the labelers instead. If the labelers cannot identify similar code in the test case before and after the evolution, and especially in their assert statements, the test is tagged as different test cases.

The labeling was performed individually by the first and second authors of this paper. After a first round of labeling, we assessed inter-rater agreement using Cohen's Kappa [7] calculated for each category, obtaining a mean Kappa of 0.475, signifying "moderate agreement" [25]. To avoid systematic disagreements, the two labelers discussed and resolved disagreements on a random sample of 20% of the entire dataset to establish a shared understanding. For instance, a recurring disagreement was the presence of setup code in assert statements. After resolving these differences, the labelers individually performed a second round of labeling on the remaining 80%, resulting in a mean Kappa of 0.727, signifying "substantial agreement" [25]. The remaining disagreements were discussed among the two labelers to obtain consensus.

2) Features of the property-based testing framework that are adopted: Property-based testing frameworks often offer features to customize a test case. The `@example` decorator of Hypothesis, for example, enables specifying that a value needs to be tested on every run of the test case. This can be used with known edge cases that the developer is aware of. The `@settings` decorator lets developers specify test budgets, the maximum number of examples to be generated, etc. It is also possible to reproduce test failures by re-using the random input generation seed that found the counterexample (e.g., the input for which the property did not hold).

We track the use of these features at the time the PBT was introduced. By doing so, we aim to discover whether developers are using them from the onset. We use the following categorization according to the Hypothesis API [42]:

- Strategies: `@given` to generate inputs. The next paragraph categorizes the input strategies specified through this decorator further.
- Explicit inputs: `@example`, which enable the tester to provide inputs to test each run.
- Reproducing inputs: `@reproduce_failure` and `@seed`, to reproduce the data from previous test failures.
- Settings: `@settings`, to impose test budgets, a maximum number of examples to generate, etc.
- Control: `assume`, `@note`, `@event`, `target`, used in the test case to modify how it is treated by the test runner, or to further document it.

3) Input generation strategies: As the input generation strategy is an important part of every PBT, we also categorize the strategies chosen at migration time as follows:

- Simple: strategies that generate standard values (e.g., integers, floats, strings), and collections thereof (e.g., lists, tuples). Strategies that compose standard values into a user-defined data type are classified in the next category instead.
- Custom, with reuse: strategies that are custom to the project, but reused across multiple test cases.
- Custom, no reuse: strategies that are custom to the project, but *not* reused across multiple test cases
- Other: usage of third-party libraries for input generation. For example, to generate NumPy values.

We assign each test case one of these four categories. Custom strategies encompass cases where developers define their own, complex, input generators for the test cases. We also consider any usage of `builds` and `from_type` in this category, as developers use them to generate custom user-defined data types. When a test generates both values of standard and user-defined data types, we consider its generation strategy custom overall.

B. Results

1) Changes to the test case body: We first discuss the changes that conventional unit tests undergo when rewritten into a property-based test. Figure 1 shows the results for both EBTs and PUTs that evolved into a PBT.

For test cases that evolved from EBTs in our dataset, it is especially common (117/182) for the setup code to be changed in a non-semantic way (for example, replacing a hardcoded value to instead refer to the input generation strategy, as shown in Listing 4). The second most common change is for the oracle code to be updated with a non-semantic change. Other common changes are additions of oracle code, semantic changes to oracle code, and removal of oracle code or setup code. Additionally, 12% of cases changed beyond recognition.

Tests that evolve from PUTs are a bit different. Here, we see fewer changes in the test cases overall. The most common change is the addition of oracle code, followed by the addition of setup code, and semantic changes to the oracle code. Notably, 35/75 test case evolutions from PUTs in our dataset required no change at all in the body of the test case (in comparison to 2 stemming from EBTs). Many PUTs might already be testing a property, especially if they contain no hardcoded expected values in the parameter. This means that evolving them into a property-based test can be less labor-intensive for developers.

2) Features of the property-based testing framework that are adopted: The results are shown in Table II. All tests specified at least one input generation strategy, as expected from a PBT. Hypothesis itself sees this decorator as the entry point of a PBT. Control (specifically `assume`), was

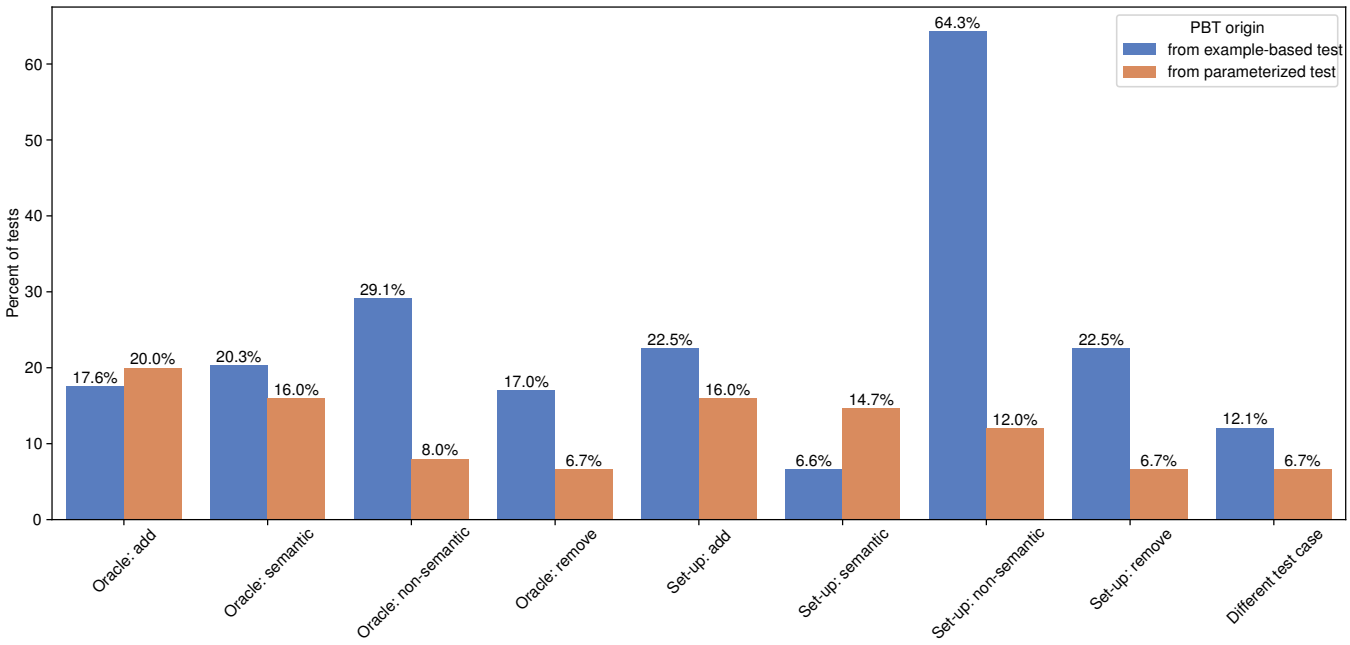

Fig. 1: The changes example-based (blue) and parameterized (orange) unit tests undergo when evolving into a property-based test. Each test case can undergo multiple changes from different categories.

	From EBT		From PUT	
Feature	# of test cases	%	# of test cases	%
Strategies	182	100	75	100
Explicit inputs	4	2.20	2	2.67
Reproducing inputs	0	0	0	0
Settings	11	6.04	8	10.67
Control	2	1.10	14	18.67

TABLE II: Features initially adopted from PBT framework.

the second most common feature, followed by `@settings` (mostly imposing deadlines on the test cases) and explicit inputs (`@example`). The fact that we see no PBT using failure-reproducing features is not surprising, as it is the first PBT-version of the test cases. They might not yet have run enough to want to reproduce a specific failure-inducing input.

Compared to test cases that evolved from EBTs, test cases that evolved from PUTs are more likely to adopt these features of the PBT framework. One reason for this could be that the developers are already more familiar with more advanced testing frameworks.

Corgozinho et al. [8] also performed a preliminary study into commonly used features across 86 PBTs. Their dataset does not consider the initial PBT introduction, and contains more well-established, big projects that make use of property-based testing. In their dataset, they observe a higher feature adoption rate compared to ours. This could be because developers evolving a PBT from an existing test might be more novice users of the framework, or because the test cases did not yet have the time to evolve and be tailored to the project. Therefore, we also investigate how the evolved test cases are maintained in Section VI-B2.

3) Input generation strategies: Table III depicts the results for our analysis of the input generation strategies used. The majority of PBTs that evolve from an EBT have a simple input generation strategy for standard data types. However, for 76 of the 120 simple cases (a majority), more refined configurations are used of these simple input generators (such as min/max bounds for integers). 60 of the EBTs that evolved into PBTs now have a custom input generation, of which 24 are reused multiple times across different test cases.

When considering PBTs that have evolved from a PUT, just over half of the PBTs have a simple input generation strategy. Of those 43 PBTs, 35 refined configurations. All custom input generation strategies are reused across multiple test cases. It is common for evolved PUTs from the same project to already share the same input parameter before their evolution into a PBT. In our dataset, the use of libraries for input generation is not common (only two instances). The only used library was the Hypothesis NumPy extension[5].

Several test cases already contained some form of randomization before being rewritten into a PBT. Afterwards, this randomization is still reflected in the input generation strategy. For example, if a test case previously contained a statement `a`

[5]https://hypothesis.readthedocs.io/en/latest/reference/strategies.html#hypothesis-numpy

	From EBT		From PUT	
	#	%	#	%
Custom, with reuse	36	19.78	32	42.67
Custom, no reuse	24	13.19	0	0
Simple	120	65.94	43	57.33
Other	2	1.10	0	0

TABLE III: Input generation strategies used within PBTs evolved from an EBT or a PUT.

`= np.random.randint(1, 10)`, a strategy of the test case would become `@given(st.int(1, 10))`.

RQ1 Non-semantic changes to the setup code are especially common in evolutions stemming from EBTs (117/182). For evolutions from PUTs, it is more common to require no changes in the body of the test case at all (35/75). The addition of settings is the most common feature (19/257) early on in the adoption of the PBT framework. Most test cases use simple input generation strategies (163/257).

V. RQ2: What is the impact of adopting property-based testing?

With research question two, we aim to find the impact of introducing PBTs. We consider commit messages, code coverage, and test failures of the 257 test evolutions.

A. Research Method

1) PBT-introducing commits: We investigate and report on 73 commit messages across 64 repositories that evolve at least one conventional test case into a PBT. We aim to uncover reasons as to why the developer decided to evolve their test case, and their potential opinions on this change.

2) Code coverage: We first explore whether code coverage improves once a conventional test case has evolved into a PBT. Improving coverage can be a goal for some PBT-introducing developers, even though it does not necessarily improve test suite effectiveness [20]. Moreover, research has proposed variants of PBT with that explicit aim (i.e., coverage-guided PBT [24], [29], [30]).

We run each test case with statement coverage before and after the introduction of Hypothesis while measuring coverage using Coverage.py [4], a well-established Python library for test coverage. We only compute the coverage for each individual test case, instead of for the entire test suite as a whole. For relative coverage (i.e., in percentage), we normalize the executed statements by those in the files executed by the test case rather than by those in the entire project. We opted for this design as coverage across all statements of the entire project would be very low for individual test cases. We also report on coverage in absolute numbers. This can be useful for test cases that lead to the execution of several files. Note that, by default, coverage.py considers all executed Python files including the files that define the test case. For projects that define behavior in their test files, these should indeed be considered. For others, including test files can lead to a wrong conclusion when the test cases themselves have shrunk or increased in size. We therefore report both on the absolute and relative coverage twice, once with the test files included and once with the test files excluded.

We were able to run 219 of the test cases before PBT-introduction, and 219 test cases after PBT-introduction. However, not all were able to obtain code coverage. In the end, we were able to run and obtain code coverage for 211 test case evolutions before and after the introduction of the PBT (meaning a total of 422 test cases). To ensure the test cases ran, some minimal changes to the code were needed. For example, one test case contained a broken import. Upon correcting this import, the test case did run. However, we made sure to not change the semantics of the system under test. In some cases, tests failed but still resulted in some code coverage. Their results are also included.

3) Test failures: Code coverage is not the only way to determine whether tests are effective. During the data collection of Section V-A2, not all test cases passed, and some test oracles threw an assertion error. For some test cases, this happened before their evolution into PBT, whereas for others, the PBT did not pass. We investigate how often this occurs, and the failed assertions in each.

B. Results

1) PBT-introducing commits: First, we report on our analysis of the commit messages of the PBT-introducing commits.

Only eight commit messages mention anything positive or negative about Hypothesis or property-based testing (beyond simply stating that they have started using it). One commit message mentions improving code coverage to find more errors across the code. Two commit messages mention the new PBTs failing (a bug was found, but not yet fixed). Finally, five commit messages use the words *better*, *trickier*, or *improved*.

Four commit messages also mentioned bug fixes of some sort, meaning the PBT was introduced at the same time as a bug fix. This can indicate that the PBT at introduction might have found a bug that the initial conventional test case did not, after which the developer fixed the bug. Alternatively, the developer may also be introducing bug fixes and PBTs at the same time to ensure their most recent bug fix is tested more in the future. We did not find any evidence in the commit message supporting either hypothesis, but Section V-B3 looks into test case failures before and after the commit.

2) Code coverage: To verify whether there is a difference in coverage before and after the evolution into PBT, we use a Wilcoxon signed-rank test [41] as our data is non-parametric and paired (code coverage of a test case before and after the PBT-introducing commit). We consider a p-value of <0.05 statistically significant. The test results are listed in Table IV. For all 4 cases, we observe a statistically significant difference.

First, we look at the change in number of statements covered. Figure 2 shows box plots for the distribution of the absolute statement coverage before and after the test case

	Stmt (all)	% (all)	Stmt (no test)	% (no test)
p-value	0.006	2.497×10^{-20}	1.477×10^{-13}	0.045

TABLE IV: p-values of change in coverage, in terms of absolute and relative statement coverage.

	total # ran tests	# passing	# failing	with coverage
Before PBT	219	199	20	211
After PBT	219	207	12	211

TABLE V: Number of passing and failing test cases. We could not obtain coverage for all tests (as shown by the last column).

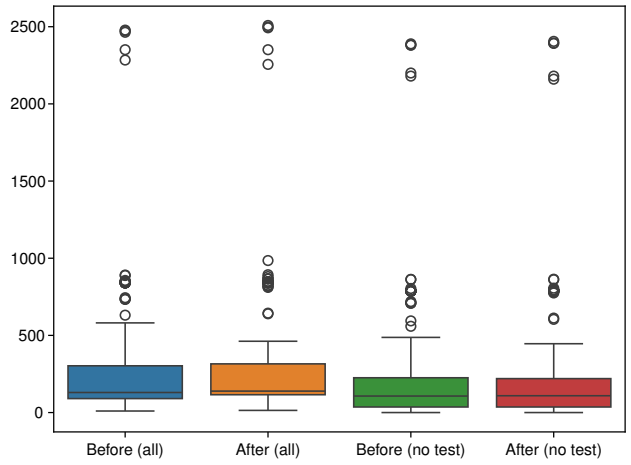

Fig. 2: Statements covered by test cases before and after PBT-introduction. The first two plots show coverage across all files, the second two without test files included.

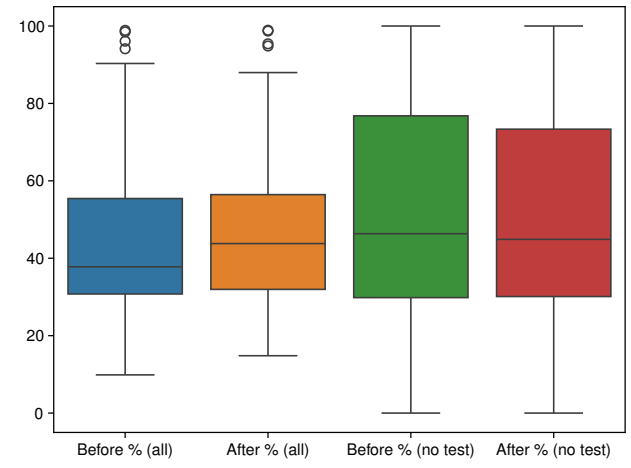

Fig. 3: Coverage percentage by test cases before and after PBT-introduction. The first two plots show coverage across all files, the second two without test files included.

coverage Δ	# Stmt (all)	% (all)	# Stmt (no test)	% (no test)
Mean	+13.09	+2.31	+4.01	-0.82
Std Dev.	27.79	6.06	16.81	5.5
median	+11	+0.57	+0	+0
1st Quar.	+2	-0.06	+0	-1.26
3rd Quar.	+17	+2.46	+3	+0.28
Min	-78	-10.09	-75	-21.90
Max	+223	+32.61	+77	+32.52

TABLE VI: Changes in code coverage, both in terms of number of statements covered and change in the coverage percentage, for both all files and all files without test files included.

evolution, taking into account all executed files (denoted "all" in the plot) and all executed files without test files (denoted "no test"). While we see a slight improvement in median and quartiles when all executed files are taken into account, the change becomes much more subtle when excluding the executed test files. One reason could be that property-based test cases execute more lines of test code. Even if the tests themselves are not necessarily longer, the inclusion of input generators can add more test code to cover —especially for projects that have defined custom ones in test files.

Figure 3 on the other hand shows the relative statement coverage in percentage. As mentioned before, by default, coverage.py only takes into consideration files that have at least one executed line of code, meaning files that are never covered by the test case are not considered. Here we also see an improvement when taking into account test files, but once they are removed from the data, we notice a slight *decrease* in code coverage on average.

Table VI shows the deltas for each of the four measurements. While the mean absolute number of covered statements without test files included does improve slightly (4.01), the relative coverage percentage goes down slightly. This could be due to extra files being discovered, or more code being introduced during the PBT-introducing commit. Most noticeably, one project's covered statements decreases by 75. Upon further investigation, the test case in question contained many assert statements. While the conventional test case before PBT-introduction passed, Hypothesis found a counterexample on the first assert, thereby stopping the test case and not executing the subsequent assert statements (leading to a drastic decrease in coverage). Additionally, 26 test cases from the same repository all had one additional change when evolved into PBT: the removal of a call to the logging function (which was three statements), leading to a decrease of coverage of indeed 3 statements for these 26 test cases.

3) Test failures: Table V shows the number of tests failed before and after PBT-introduction. Two test cases did not pass as a conventional unit test, nor after having been rewritten into a PBT. For 18 test cases, the initial conventional test did not pass, but the updated PBT did pass. In another ten cases, the initial test passed but the resulting PBT did not.

In terms of the *before*, one project with one evolution had a test case that resulted in an assertion error. The PBT-introducing commit also contained a bug fix. Another project had three failing test cases that returned a type error. After the introduction of the PBT, all three passed. However, the PBT-introducing commit does not introduce a bug fix (meaning the bug was in the test code itself). Another project had two failing test cases, with an attribute error and a type error. For both, the PBT versions did pass. In one of the bigger projects in our dataset, with 16 evolved test cases, ten tests did not pass before the evolution. In three cases this was because of an assertion error (with the other seven not passing due to a value not being defined). All ten of these test cases passed after being updated to a PBT. These results suggest that people are introducing PBTs when fixing known errors, perhaps to ensure more thorough testing of those parts of the system.

When looking at the failing tests cases *after* the introduction of PBTs, we found five test cases for which Hypothesis reported a counterexample for which the original test case passed. Further, as mentioned in Section V-B1, some commits did contain additional bug fixes. This indicates that PBTs can uncover bugs not found by conventional test cases validating the same behavior, thus improving bug detection. For the other 5 non-passing PBTs, we found other errors (such as ValueErrors), even though the original test case passed before.

RQ2 8 PBT-introducing commit messages convey an intent to improve test cases to improve coverage or find bugs. We observed a statistically significant change in statement coverage, with on average an increase of 4 statements. We also found evidence of PBTs being introduced along with a fix for a bug in the code, and of PBTs finding counterexamples where the original corresponding conventional test passed.

VI. RQ3: How are property-based tests maintained?

We consider the most recent version of the 250 unique PBTs from the in total 257 test case evolutions, and investigate the changes (or removals) they have been subject to since.

A. Research Method

For each test case, we compare the version resulting from the PBT-introducing commit to the version in the most recent commit in its repository. We consider the following:

1) Do Property-based Test Cases Change?: We investigate whether the 250 PBTs in our dataset change over time, for instance in response to changes in the system under test. For each PBT, we manually compare against the same test case in the most recent version of the GitHub repository. For each of our test cases, we consider the following:

- No changes to the test case. The rest of the project and other test cases might have changed, but the investigated PBT has not.
- Removal of the test case. In this case, we also investigate possible reasons why.
- Updates to the test case encompassing both semantic and non-semantic changes.

We make one exception for 39 test cases originating from one repository. In the most recent version of that repository, most Python code has been removed, and is scheduled to be added at a later stage. Because of this drastic rewrite affecting the latest version in the repository, we instead consider its most recent version with the test cases still present.

Additionally, we report on the survival rate of the test cases by looking at the number of commits since introduction, or the number of commits between introduction and removal.

2) Are more parts of the PBT framework adopted?: As developers become more acquainted with property-based testing in their projects, they might discover features unbeknownst to them before. Therefore, similar to Section IV-A2, we investigate whether the most recent version of their PBT uses more features of the PBT framework. We use the same categories of features from the Hypothesis API [42] to classify the additions. As all PBTs already had an input generation strategy when they were introduced, we investigate whether this strategy has been updated in the meantime.

B. Results

1) Do Property-based Test Cases Change?: To answer this question, we compare the most recent commit in the GitHub repository to the PBT-introducing commit. Table VII shows the number of commits the test cases have survived (for those that are still present in the current version of their project, i.e., the number of commits between introduction and the most current commit), or how many repository commits they survived before their deletion. Most test cases in our dataset (110) are unchanged. Their corresponding repositories have also enjoyed the fewest commits since the PBT-introducing one (with 99 on average, with a standard deviation of 205). Interestingly, one test case survived 1728 repository commits without undergoing any changes itself. A total of 90 test cases did undergo changes since their introduction, both semantics-preserving and semantics-affecting ones. We observe that these test cases also survived more commits, with on average 642 commits and a higher standard deviation. This is expected as PBTs need to co-evolve with their system under test.

Finally, 50 PBTs were removed from the projects since their initial introduction. These test cases also survived fewer commits; 352 on average. However, one test case survived 3573 commits to the repository before being removed. Reasons cited for the deletion of these test cases were removal of the functions they tested (5), unexpected behavior (3), and the tests being slow (1).

2) Are more parts of the PBT framework adopted?: Finally, we consider the changes made to PBTs in terms of the parts of the PBT framework that are adopted. Table VIII describes the results. In 65 test cases, the input generation strategies were updated after their introduction. One repository in particular mentions doing so to improve the running time of the test cases. Only one repository added an explicit input to test for on each run. Additionally, one test case added a `seed` decorator

	No changes	Changes	Removal
Mean	99	642	352
Std Dev.	205.31	1450.79	967.80
median	51	146	34
1st Quar.	48	64	17
3rd Quar.	63	146	71
Min	1	17	7
Max	1728	5886	3573

TABLE VII: Number of commits since the introduction of the PBT. Data split for the test cases that did not change at all, for those with changes, and for those removed in the meantime.

Feature	# of test cases
Change in strategies	65
Addition of explicit inputs	1
Addition of reproducing inputs	1
Addition of settings	19
Addition of control	0
Addition of exceptions	0

TABLE VIII: Changes in used PBT features in the most recent versus the initial PBT-introducing commit of the test case.

to reproduce inputs. 19 of the test cases eventually got some form of `settings` added, which doubles their total number.

RQ3 Most PBTs (200/250) in our dataset have survived since their introduction, with 90 having gone through changes. Removed test cases survived on average 352 commits to the repository. 65 test cases saw a change in their input generation strategy since their introduction. Finally, 19 test cases received additional settings.

VII. Discussion

We now distil the actionable insights from our findings, for software testers, tool builders, and researchers alike. We also discuss the threats to the validity of our findings.

A. Actionable insights

For software testers: As evidenced by our findings for RQ2 (Section V-B), evolving well-chosen unit tests into property-based ones can increase their code coverage. We moreover found evidence of such real-world evolutions finding counterexamples where the original test passed without any failures, leading to a deeper validation of the system under test. We even found instances of these evolutions being conducted alongside bug fixes, most likely to test for regressions more thoroughly. Our findings should motivate practitioners to consider unit tests that have missed edge cases in the past as candidates to be evolved into property-based ones.

Prior studies have found that implementing input generation strategies for a PBT can be tedious and effort-intensive [14]. Nonetheless, our findings for RQ1 (Section IV-B) indicate that many evolved PBTs in our dataset use but a simple, built-in input generation strategy. Practitioners may find it easier to evolve unit tests that exercise the unit under test on inputs of standard data types (e.g., integers) into a PBT with such a strategy. Our findings should motivate practitioners to prioritize their generalization efforts on those unit tests first.

Finally, a study by Goldstein et al. [14] reports that PBT adopters have difficulty identifying the properties to test for. Using existing test cases, rather than creating new ones, might help. This is especially true for parameterized unit tests (PUT), and especially those that do not list expected outputs for every given input, as those might already be testing a property (but not yet randomly generating input). Indeed, for 35/75 PBTs that evolved from a PUT (Section IV-B), no changes to the body of the test case were required at all. This suggests that PUTs can be considered a stepping stone towards, or even an intermediate step in a planned introduction of PBTs.

For tool developers: Recent years have seen the introduction of tool support for creating PBTs. From Hypothesis' own ghostwriter[6], over using LLMs to write PBTs [13], [38], to tools that aim to aggregate similar test cases into one PBT [31]. Tools that take EBTs and transform them into PUTs also exist [37]. However, there remains a need for tools that support developers in evolving existing, dissimilar, test cases into PBTs. Our findings for RQ1 (Section IV-B), and our supporting dataset, can inspire tool builders with examples of real-world transformations that can be partially automated.

As shown in Section IV-B2, not all features of the property-based testing framework are immediately adopted in newly-evolved PBTs. In Section VI-B2 we found that features such as PBT settings are often included only later in the life of a PBT. We also saw that inside some PBTs, developers were using `if`-statements, where an `assume` might have been more appropriate. Therefore, future tools could not only help with the evolution of test cases into PBTs, but also aid with the introduction of features to optimize existing PBTs.

For researchers: There are still few empirical work that compares conventional unit tests to PBTs. Ours is, to our knowledge, the first to do so in a one-to-one comparison of test cases before and after their evolution into property-based ones. Our comparison on test failures and code coverage has been insightful, and could inspire other one-to-one comparisons on other metrics. For instance, mutation score (recently used in prior work [32], see Section VIII) has not yet been explored in a one-to-one comparative setting.

B. Threats to validity

Following standard practice in empirical software engineering, we discuss potential threats to the validity of our results. For each threat, we describe the applied mitigation strategies.

Construct validity: We considered test case evolutions that are found in open-source GitHub repositories, which can raise quality concerns. We mitigated this threat by applying strict inclusion criteria, resulting in 575 high-quality candidate repositories with at least one PBT. As most of those PBTs did not result from an evolution, we were left with fewer test case

[6]https://hypothesis.readthedocs.io/en/latest/reference/integrations.html#ghostwriter

evolutions to study, but the study subjects still stem from 64 different repositories, ensuring diversity.

Further, while false negatives cannot be excluded, all test case evolutions were manually inspected by both the first and the second author, ensuring the dataset is free of false positives. The same goes for the before-after mappings for each test case evolution, which were created by the first author and inspected and confirmed by the second author.

Manual labeling of test case evolutions was also used to answer RQ1. We mitigated the threat of subjective bias by having multiple labelers and multiple labeling rounds until a substantial inter-rater agreement was reached. To answer RQ2, we used tools to execute test cases and compute their code coverage. To mitigate the threat of measurement bias introduced by the tooling, we only used mature open-source tools that are well-established within the community.

Internal validity: For RQ2, we measure changes in code coverage induced by the evolution into a PBT. However, the introduction of a PBT is not always the only change in a commit. If additional changes are present, the total number of statements may change, which can confound the observed coverage differences and lead to incorrect conclusions. To mitigate this threat, we manually inspected commits exhibiting large coverage changes. In addition, we report both absolute and relative coverage, computed with and without the test files included, as test files are more likely to change due to the introduction of PBTs.

External validity: Finally, our study focuses on Python test cases that use the Hypothesis framework. Hypothesis is the most widely recognized PBT framework for Python, and Python is ranked among the most popular programming languages. Nevertheless, our findings may not generalize to other programming languages nor to other PBT frameworks.

VIII. Related Work

Several studies have investigated the introduction and maintenance of various types of test cases, from tests in general [1], [9], [44] over GUI tests [5] to performance tests [11]. A few studies have also looked into changes in code coverage induced by software patches, e.g., [12] and [17]. However, our work is the first to study how PBTs can evolve from conventional unit tests, and to study the impact on code coverage —rather than changes in code coverage over time or induced by changes to the system under test.

Zooming in on other studies targeting property-based testing, Corgozinho et al. [8] sampled 86 PBTs from GitHub, and categorized each of them manually according to the testing patterns they adhere to, which stem from an influential blog post[7]. They also study the use of Hypothesis features in those PBTs, similar to us. However, while they investigate bigger, established projects, we investigate the adoption of these features in test cases that were previously not a PBT. We also study whether those features are adopted later on, while the PBT matures.

[7]https://fsharpforfunandprofit.com/posts/property-based-testing-2/

Goldstein et al. [14] interviewed 30 developers from a fintech company, to find insights about how people experience PBT in practice. Another paper by Goldstein et al. [15] reports that developers experience difficulties in finding properties to test for. Our research looks at open-source projects rather than the industry, and investigates existing test cases.

A study by Wauters and De Roover [39] looked into the use of PBTs by 28 open-source machine learning projects. The study investigated common positive and negative sentiments expressed in GitHub commits and code comments, which parts of a project were tested, as well as the complexity of data generation strategies. Our study also investigates data generation strategies but does not focus on ML-intensive projects that often require more complex data.

Ravi and Coblenz [32] recently performed a large-scale empirical study on PBTs in 426 Python programs. Their study uses a data flow analysis to automatically categorize real-world PBTs into 12 categories. Additionally, they used mutation testing to investigate how many mutations a project's existing PBTs can find compared to the project's existing conventional tests. Our research looks at the impact of evolving test cases, and compares a test case before its evolution into a PBT to the resulting PBT. This means we compare two test cases that are equivalent in terms of the behavior they are supposed to validate, and we do so one-to-one.

Unlike our study, the aforementioned papers do not investigate the evolution of PBTs, but rather consider snapshots of PBTs at one point in time.

IX. Conclusion and Future Work

This paper studied the evolution of property-based unit tests from conventional ones, the impact of such PBT-introducing evolutions, and how the resulting PBTs are maintained afterwards. We found that the evolution of parameterized unit tests into a PBT commonly requires no updates to their test body at all. For example-driven unit tests, in contrast, the evolution commonly required non-semantic changes to the test body. These account for the introduction and usage of a randomly generated input value. The input generation strategy of PBTs that evolved from existing unit tests is often simple. Impact-wise, we found that code coverage can improve when unit tests are replaced by PBTs. More importantly, we found evidence of bugs found through or alongside the introduction of these PBTs. Most of the PBTs in our dataset are still present in the most recent version, with almost half having gone through some maintenance during their life time. The adoption of additional PBT features is not uncommon either.

One avenue for future work is investigating the long-term impact of PBT-introducing test case evolutions. Another is developer support in the form of a tool that identifies candidate unit tests for such an evolution, and for partially automating the necessary test changes.

Acknowledgements

This research was partially funded by the Research Foundation Flanders (FWO) Grant No. 1SHFI24N and by the Cybersecurity Research Program Flanders (CRPF).

References

[1] Maurício Aniche, Christoph Treude, and Andy Zaidman. How developers engineer test cases: An observational study. *IEEE Transactions on Software Engineering*, 48(12):4925–4946, 2022.

[2] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, page 2–10, New York, NY, USA, 2006. Association for Computing Machinery.

[3] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[4] Ned Batchelder and Contributors to Coverage.py. Coverage.py: The code coverage tool for Python. https://web.archive.org/web/20251223093400/https://github.com/coveragepy/coveragepy. Accessed on 20 december 2025.

[5] Laurent Christophe, Reinout Stevens, Coen De Roover, and Wolfgang De Meuter. Prevalence and maintenance of automated functional tests for web applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 141–150, 2014.

[6] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[7] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[8] Arthur Lisboa Corgozinho, Marco Tulio Valente, and Henrique Rocha. How Developers Implement Property-Based Tests . In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 380–384, Los Alamitos, CA, USA, October 2023. IEEE Computer Society.

[9] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211, 2014.

[10] Albert Danial. cloc: v1.92. https://doi.org/10.5281/zenodo.5760077, December 2021.

[11] Sergio Di Meglio, Luigi Libero Lucio Starace, Valeria Pontillo, Ruben Opdebeeck, Coen De Roover, and Sergio Di Martino. Performance testing in open-source web projects: Adoption, maintenance, and a change taxonomy. In *Proceedings of the 41st IEEE International Conference on Software Maintenance and Evolution (ICSME 2025)*. IEEE, sep 2025. 41st IEEE International Conference on Software Maintenance and Evolution (ICSME 2025), ICMSE ; Conference date: 07-09-2025 Through 12-09-2025.

[12] S. Elbaum, D. Gable, and G. Rothermel. The impact of software evolution on code coverage information. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 170–179, 2001.

[13] Khashayar Etemadi, Marjan Sirjani, Mahshid Helali Moghadam, Per Strandberg, and Paul Pettersson. Llm-based property-based test generation for guardrailing cyber-physical systems. In *International Conference on Bridging the Gap between AI and Reality*, pages 18–46. Springer Nature Switzerland Cham, 2025.

[14] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[15] Harrison Goldstein, Joseph W Cutler, Adam Stein, Benjamin C Pierce, and Andrew Head. Some problems with properties. In *Proc. Workshop on the Human Aspects of Types and Reasoning Assistants (HATRA)*, 2022.

[16] Harrison Goldstein, Jeffrey Tao, Zac Hatfield-Dodds, Benjamin C. Pierce, and Andrew Head. Tyche: Making sense of pbt effectiveness. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, UIST '24, New York, NY, USA, 2024. Association for Computing Machinery.

[17] Michael Hilton, Jonathan Bell, and Darko Marinov. A large-scale study of test coverage evolution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 53–63, New York, NY, USA, 2018. Association for Computing Machinery.

[18] John Hughes. *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*, pages 169–186. Springer International Publishing, Cham, 2016.

[19] John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. Mysteries of dropbox: Property-based testing of a distributed synchronization service. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 135–145, 2016.

[20] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 435–445, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Paul Jansen. Tiobe index for december 2025. https://web.archive.org/web/20251218235222/https://www.tiobe.com/tiobe-index/. Accessed on 20 december 2025.

[22] JetBrains. Python developers survey 2023 results. https://web.archive.org/web/20250920050020/https://lp.jetbrains.com/python-developers-survey-2023/. Accessed on 10 december 2025.

[23] Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, and Florian Bruhin. pytest 9.0.0. https://web.archive.org/web/20250906051102/https://github.com/pytest-dev/pytest/, 2004. Version 9.0.0 Contributors include Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, Florian Bruhin, and others.

[24] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. Coverage guided, property based testing. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[25] J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

[26] David R. MacIver and Alastair F. Donaldson. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:27, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[27] David R MacIver, Zac Hatfield-Dodds, et al. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.

[28] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[29] Agustín Mista and Alejandro Russo. Mutagen: Reliable coverage-guided, property-based testing using exhaustive mutations. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 176–187, 2023.

[30] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 398–401, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Hila Peleg, Dan Rasin, and Eran Yahav. Generating tests by example. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 406–429, Cham, 2018. Springer International Publishing.

[32] Savitha Ravi and Michael Coblenz. An empirical evaluation of property-based testing in python. *Proc. ACM Program. Lang.*, 9(OOPSLA2), October 2025.

[33] Mohammad Rezaalipour and Carlo A. Furia. An annotation-based approach for finding bugs in neural network programs. *Journal of Systems and Software*, 201:111669, 2023.

[34] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 908–911, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. *SIGSOFT Softw. Eng. Notes*, 30(5):253–262, September 2005.

[36] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, page 253–262, New York, NY, USA, 2005. Association for Computing Machinery.

[37] Deepika Tiwari, Yogya Gamage, Martin Monperrus, and Benoit Baudry. Proze: Generating parameterized unit tests informed by runtime data.

In *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 166–176, 2024.

[38] Vasudev Vikram, Caroline Lemieux, Joshua Sunshine, and Rohan Padhye. Can large language models write good property-based tests?, 2024.

[39] Cindy Wauters and Coen De Roover. Property-based Testing within ML Projects: an Empirical Study . In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 648–653, Los Alamitos, CA, USA, October 2024. IEEE Computer Society.

[40] Cindy Wauters, Ruben Opdebeeck, and Coen De Roover. Replication package on the evolution of python test cases into property-based tests. https://doi.org/10.6084/m9.figshare.31424447, Feb 2026.

[41] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[42] Hypothesis Works. Api reference - hypothesis 6.148.3 documentation. https://web.archive.org/web/20251223092313/https://hypothesis.readthedocs.io/en/latest/reference/api.html. Accessed on 10 december 2025.

[43] Hypothesis Works. When to use hypothesis and property-based testing - hypothesis 6.148.3 documentation. https://web.archive.org/web/20251223092910/https://hypothesis.readthedocs.io/en/latest/tutorial/introduction.html#when-to-use-hypothesis-and-property-based-testing. Accessed on 11 december 2025.

[44] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. Mining software repositories to study co-evolution of production & test code. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 220–229, 2008.