How Developers Implement Property-Based Tests

Arthur Lisboa Corgozinho
Department of Computer Science
Federal University of Minas Gerais
Belo Horizonte, Brazil
arthurlisboa@dcc.ufmg.br

Marco Tulio Valente
Department of Computer Science
Federal University of Minas Gerais
Belo Horizonte, Brazil
mtov@dcc.ufmg.br

Henrique Rocha
Department of Computer Science
Loyola University Maryland
Baltimore, USA
hsrocha@loyola.edu

***Abstract*—Property-based testing (PBT) is an interesting alternative to example-based testing where the inputs are randomly generated by the testing tool. In PBT, we check properties that always hold for any input. Despite being a promising testing category, to the best of our knowledge, we still lack studies that investigate in the wild how developers are using PBT in practice. In this paper, we report the preliminary results of a study we are conducting on the usage of PBT. We created a dataset of 30 popular Python repositories using Hypothesis (a PBT tool) and selected a random sample of 86 tests. We manually analyzed these tests to understand the most commonly implemented properties and also to reveal the most used features to create them.**

***Index Terms*—Property-Based Testing, Hypothesis, Testing**

I. Introduction

Automated tests are predominantly guided by examples [9]. For instance, when implementing a unit test, a developer first selects some values to call the function under test. Then, she calls the function using these values and checks whether the returned results match the expected ones. Despite being common nowadays, example-based tests require considerable effort and expertise from developers for selecting the most effective test inputs, which is far from being a simple task.

Thus, the idea behind Property-Based Testing (PBT) is exactly to free developers from manually selecting and defining test inputs [2], [3], [5], [8]. Instead, the tested values are automatically generated by the testing framework. The ultimate goal is to allow developers to spend more time fixing bugs than defining test case inputs, examples, and scenarios. A common motto among PBT practitioners summarizes this goal: "test faster, fix more".[1]

Therefore, when using PBT it is no longer feasible to call and then check the results of the function under test since we do not know the specific values used to call this function. Instead, developers should implement tests that will check properties that always hold for the planned function behavior. As a very simple and illustrative example, a function that adds two integers should be commutative, meaning that `add(x,y)` should be equal to `add(y,x)`, for any values of `x` and `y`. As a second property, the `add` function has also a neutral element, meaning that `add(x,0)` should be equal to `x`, for any input value `x`. In summary, when using PBT, the test inputs are automatically generated by the testing framework, and, in the body of the test, developers should check "universal" properties of the function (see the code in the next listing). Since the functions are called hundreds or thousands of times by the testing framework, the assumption is that possible bugs will arise in some of such calls.

```
1  from hypothesis import given, strategies as st
2
3  def add(x,y):
4      return x + y
5
6  @given(x=st.integers(), y=st.integers()) # input generator
7  def test_add_commutative(x, y):
8      assert add(x, y) == add(y, x)
9
10 @given(x=st.integers())  # input generator
11 def test_add_zero(x):
12     assert add(x, 0) == x
```

Property-based testing was initially proposed for functional languages more than 20 years ago. For instance, QuickCheck [3] is a well-known framework for implementing such tests in Haskell. More recently, PBT is being used in Python programs, mainly due to the popularity of the Hypothesis testing framework available for this language [8]. For example, the Hypothesis repository on GitHub has almost 7K stars. However, to the best of our knowledge, we still lack studies that investigate in the wild how developers are using PBT in practice. In our view, the importance of such studies is twofold: (a) first, they can contribute to increasing the visibility of PBT among practitioners and researchers; (b) second, they can reveal the properties and features that are most used in real examples of PBTs. This last contribution is particularly important to pave the way for developers that intend to implement their first property-based tests. It could also have strategic value for framework builders, who might for example prioritize to evolve and optimize the most adopted features or even decide to deprecate the rarely used ones.

In this paper, we report the first results of a study we are conducting on the usage of PBT. Our goal is to shed light on the properties that are in fact checked in such tests (*RQ1*) and also reveal the features that are mostly used when implementing property-based tests (*RQ2*). To this purpose, we first create a dataset with 86 property-based tests implemented in Python using the Hypothesis framework. These tests were randomly selected from a curated list of 30 well-known Python projects that use PBTs, including well-known projects such as `pytorch` (a machine learning framework), `numpy` (a scientific computing package), and `polars` (a dataframe library).

[1]https://hypothesis.works

The remainder of this paper is organized as follows. In Section II, we explain the key concepts of Property-Based Tests and introduce the Hypothesis tool. Next, in Section III, we describe the study design. Our preliminary results are presented in Section IV. Threats to validity are discussed in Section V and related work in Section VI. Finally, we suggest possible directions for future work in Section VII.

II. Property-Based Testing

Property-Based Testing (PBT) is a random automated testing technique that can be used to check whether key properties of a function are valid in a given input domain [3], [7]. Unlike example-based testing, PBT relies on a set of automatically generated input values for asserting the properties of a function under test. Thus, the focus shifts from checking function results produced by manually selected inputs to the verification of function properties under random inputs [8].

A PBT testing tool executes the tests multiple times with random inputs based on an input generator specification. It also checks whether a given function property—specified by the test developer— holds for such inputs. If this property does not hold, then there is a failure, and the tool attempts to reduce it into simple inputs to make it clear and easy to reproduce. Hypothesis [8] is an open-source tool for implementing property-based tests in Python. We selected Hypothesis for conducting our study because it is a robust tool that has all the functionalities for implementing PBT.

Example 1: In Hypothesis, the `@given` decorator is used to specify the inputs that will be automatically generated for a test case. This decorator requires a data generation parameter, referred to as a *strategy*. Next, we show an example of using a strategy that generates random integer parameters which are used to assert the associative property of the addition:

```
1  from hypothesis import given, strategies as st
2
3  @given(a=integers(), b=integers(), c=integers())
4  def test_associativity(a, b, c):
5      result1 = (a + b) + c
6      result2 = a + (b + c)
7      assert result1 == result2
```

Example 2: In this second example, we deliberately introduce a major bug in the `add` function. The buggy result will occur whenever the `x` parameter is greater than `10000` (lines 4-5).

```
1  from hypothesis import given, strategies as st
2
3  def add(x,y):
4      if (x > 10000):    # bug
5         return x - y    # bug
6      return x + y
7
8  @given(x=st.integers(), y=st.integers())
9  def test_add_commutative(x, y):
10     assert add(x, y) == add(y, x)
```

Thus, Hypothesis detects the following failure in this test:

```
1  Falsifying example: test_add_commutative(
2  E           x=1,
3  E           y=10001,
4  E       )
```

The failure happens because `add(1, 10001)` returns the correct value (`10002`). However, `add(10001, 1)` will subtract the parameter values (due to our bug) and therefore returns `10000`.

III. Study Design

A. Research Questions

Our primary goal with the study is to characterize the usage of Property-Based Testing in real Python projects. In particular, we intend to answer two research questions:

RQ1: What are the most common properties checked in PBT? Our intention is to categorize and reveal the types of properties that Python developers check using Hypothesis. We hope our results can help new developers that plan to use PBT in their projects. For example, when searching for functions to be tested using PBT, they can focus on the ones that exhibit the properties revealed in this research question.

RQ2: What are the most commonly used features of Hypothesis? As usual in software tools, among the many features supported by Hypothesis, we assume there is a core set that is used by most tests. Thus, by revealing such common features, we also intend to help novices to start using PBT in their projects. In other words, they can start by implementing PBTs that use such features.

The study was carried out in three steps. First, to create a dataset for our investigation, we relied on GitHub repositories that use Python and Hypothesis (Section III-B). Then, we selected a random sample of PBTs in this dataset. Finally, we performed a manual analysis of the selected tests in order to answer the proposed research questions (Section III-C). We provide a replication package with the dataset and scripts in our GitHub repository.[2]

B. Dataset

First, we retrieved a list of projects that depend on Hypothesis using the `github-dependents-info` feature from GitHub.[3] This feature provides information about GitHub projects that depend on a given project. In the case of Hypothesis, the resulting list has 367 repositories. However, in a manual inspection, we found projects that import Hypothesis but do not use the library in any source code file. Thus, to discard such "false positives", we implemented a Python script that searches for code snippets that denote a real dependency on Hypothesis in the source code files of the selected repositories. For this task, we used PyGithub[4] and GitPython[5] libraries to retrieve the repositories' URLs and to clone them, respectively. Finally, we selected the first 30 projects—ranked by the number of stars—that contain the code snippets that mark a dependency on Hypothesis. In this way, our final dataset includes widely-used Python projects, such as `pytorch` (a machine learning framework), `numpy` (a scientific computing package), and `polars`

[2]https://github.com/arthurrlisboa/how-developers-implement-pbt
[3]https://github.com/nvuillam/github-dependents-info
[4]https://pygithub.readthedocs.io
[5]https://gitpython.readthedocs.io

(a dataframe library). In these 30 projects, we identified 763 property-based tests. However, for this emerging results paper, we decided to evaluate a random sample of 86 test cases, which provides a confidence level of 95% and a margin of error of 10% to our results.

C. Property Categories

To answer our research questions, the first author manually analyzed the 86 test cases by reviewing the source code of PBT tests using the GitHub code search tool. As a result, he classified each one into one out of ten categories. Eight of these categories are described in a blog post that is cited by the Hypothesis official documentation.[6] The two remaining ones (*Outputs within expected bounds* and *Metamorphic properties*) are mentioned in a paper by Hatfield-Dodds [6]. Next, we describe each category:

Different paths, same destination. When a combination of functions should generate the same result regardless of the order of application (e.g., commutative property of addition).

There and back again (roundtrip). When a function produces a result that is immediately accessed and returned by another function (e.g., write a value in a file followed by reading it). A distinctive feature of this category is the combination of an operation with its inverse, resulting in the same value as the initial input. (e.g., encoding a string followed by decoding it)

Some things never change. When an invariant is preserved after a transformation (e.g., the size of a list after sorting).

The more things change, the more they stay the same. When calling a function twice is the same as calling it once (e.g., filtering distinct elements from a list; after we retrieve the distinct elements once, further calls return exactly the same elements since there are no more repeated elements).

Solve a smaller problem first. When a larger problem can be divided into smaller parts and a property that is true for those smaller parts should also be true for the whole (e.g., in a list with only positive elements, the head of the list should be a positive number and the tail should hold the same property).

Hard to prove, easy to verify. When we can easily verify a result, despite the method for computing it being more complex (e.g., checking the results of a tokenization algorithm is more simple than implementing the algorithm. The reason is that we only need to concatenate the returned tokens).

The test oracle (differential testing). When a well-known algorithm is used as an oracle for checking the implementation of a novel and untested algorithm (e.g., testing a novel sorting algorithm against a default sorting method, such as QuickSort).

Model-based testing. A variant of the test oracle category, when a simplified model of the system under test is used. During testing, the same stimuli are applied to both the system and the model and their states are compared.

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

Outputs within expected bounds. When we verify the expected limits of the result. These bounds can be either computational or logical in nature (e.g., if a function returns a price the value should be non-negative).

Metamorphic properties. Concerns the validation of processes that cannot be directly verified regarding their final results. Instead, this approach focuses on variations that occur in the test input data and how these variations affect subsequent results (e.g., a time precision test when adding a small time variation before converting to another scale is practically equivalent to performing the conversion first and then the addition). This category is particularly valuable when the results can be affected by undisclosed or uncontrollable external factors.

IV. Preliminary Results

A. What are the most common properties checked in PBT?

Table I shows the most common categories after our manual classification and the percentage of tests for each one.

TABLE I
Properties tested using PBT

Property Category	#	%
Different paths, same destination	0	0.0%
Roundtrip (there and back again)	32	37.2%
Some things never change	6	6.9%
The more things change, ...	0	0.0%
Solve a smaller problem first	0	0.0%
Hard to prove, easy to verify	0	0.0%
Test oracle	29	33.7%
Model-based testing	3	3.4%
Outputs within expected bounds	5	5.8%
Metamorphic properties	3	3.4%
Other	8	9.3%

As shown in Table I, *roundtrip* and *test oracle* are the most popular categories in our sample, having been found in 61 tests (70.9%). This suggests that developers are more familiar with such properties and find them easier to implement and use. The following listing shows an example of a test that checks a *roundtrip* property in the `nylas/sync-engine` project.[7] In line 5, we can observe that a value is encoded and then immediately decoded.

```
1  # This will run the test for a bunch of randomly-chosen
       values of sample_input.
2  @hypothesis.given(str, bool)
3  def test_blobstorage(config, sample_input, encrypt):
4      config['ENCRYPT_SECRETS'] = encrypt
5      assert decode_blob(encode_blob(sample_input)) ==
           sample_input
```

The following listing shows a *test oracle* being used in a property-based test from the PyTorch project.[8] As can be seen in line 22, this test verifies the implementation of the `tile()` method using as reference the corresponding method in

[7]https://github.com/nylas/sync-engine/blob/b91b94b9a0033be4199006eb234d270779a04443/tests/security/test_blobstorage.py#L7

[8]https://github.com/pytorch/pytorch/blob/7cef7195f616f75bf25a48cf5692f704d35ac4b2/caffe2/python/operator_test/tile_op_test.py#L18

an external library (NumPy). The purpose of this method is to create a new array by repeating an input array a specific number of times.

```
@given(M=st.integers(min_value=1, max_value=10),
   K=st.integers(min_value=1, max_value=10),
   N=st.integers(min_value=1, max_value=10),
   tiles=st.integers(min_value=1, max_value=3),
   axis=st.integers(min_value=0, max_value=2),
   **hu.gcs)
@settings(deadline=10000)
def test_tile(self, M, K, N, tiles, axis, gc, dc):
    X = np.random.rand(M, K, N).astype(np.float32)

    op = core.CreateOperator(
        'Tile', ['X'], 'out', tiles=tiles, axis=axis,
    )

    def tile_ref(X, tiles, axis):
        dims = np.asarray([1, 1, 1], dtype=int)
        dims[axis] = tiles
        tiled_data = np.tile(X, dims)
        return (tiled_data,)

    # Check against numpy reference
    self.assertReferenceChecks(gc, op, [X, tiles, axis],
                               tile_ref)
    # Check over multiple devices
    self.assertDeviceChecks(dc, op, [X], [0])
    # Gradient check wrt X
    self.assertGradientChecks(gc, op, [X], 0, [0])
```

Four categories (*some things never change*, *model-based testing*, *outputs within expected bounds*, and *metamorphic properties*) have a lower number of occurrences, ranging from three to six tests. This may suggest that these categories are not as widely known or useful as the previously described ones. In the following listing[9], we show an example for *some things never change* of a test that validates whether, after serialization, the response remains encoded in base64, if the body continues to be a string, and if, upon decoding the body, it remains an instance of bytes.

```
@given(body=st.binary(), content_type=st.sampled_from(
    BINARY_TYPES))
def test_handles_binary_responses(body, content_type):
    r = Response(body=body, headers={'Content-Type':
     content_type})
    serialized = r.to_dict(BINARY_TYPES)
    # A binary response should always result in the
    # response being base64 encoded.
    assert serialized['isBase64Encoded']
    assert isinstance(serialized['body'], six.string_types)
    assert isinstance(base64.b64decode(serialized['body']),
          bytes)
```

Furthermore, it is important to highlight that four categories (*different paths, same destination*, *the more things change, the more they stay the same*; *solve a smaller problem first*; and *hard to prove, easy to verify*) were not found in our sample, with zero occurrences. This indicates a possible difficulty for developers in identifying these properties in their programs.

Finally, we were not able to classify eight tests into any category. Interestingly, six tests call a function in a way that results in an exception or invalid state, which is then checked by the test. This may suggest, therefore, a novel property category that is relatively common in PBTs, although not described in the literature. Thus, we plan to better investigate this new category in our future work.

> *Summary:* Although the literature reports ten distinct categories of properties that can be checked using PBT, in our analysis, we found that two categories are by far the most common ones: *roundtrip* (used by 37.2% of the tests in our sample) and *test oracle* (used by 33.7%). Thus, both account for 70.9% of the categories checked by the property-based tests in our study.

B. What are the most commonly used features of Hypothesis?

Table II shows the Hypothesis features that are most used in our sample of PBTs. The columns show the number and percentage of tests where each feature was found.[10]

TABLE II
USE OF HYPOTHESIS FEATURES

Feature	Description	#	%
strategies	Module for input generation strategies	56	65.1%
@settings	Decorator for test configuration	22	25.5%
assume()	Validates preconditions in test data	13	15.5%
@example	Decorator to ensure that a specific example is always tested	4	4.6%
note()	Adds values to execution output on minimal failure	1	1.1%

As expected, most tests use the input generation strategies provided by Hypothesis (65%). Test configurations are also used by many tests (25%). The `settings` decorator is used for defining testing parameters (e.g., explicitly setting the number of times Hypothesis should run a given test). On the other hand, there is a lower usage of `assume()` (15%). This feature enables developers to define conditions the input values generated by Hypothesis should have; if such conditions are not satisfied, the execution of the test is aborted. We also found tests that use the `example` (4 tests) and `note` (one test) features. The `example` decorator is used to execute a test on a specific input, thus allowing developers to cover edge cases or reproduce a failure. The `note` feature allows developers to add additional information in the messages produced by Hypothesis in the case of failures.

Since strategies are widely used in our sample, we also performed a finer-grained analysis on this feature. For each strategy supported by Hypothesis, Table III, shows the number and the percentage of tests using it. Since a test can use more than one strategy, the percentages add up to more than 100%. Most strategies are self-explanatory, generating a data type related to its name (e.g., `integer()` generates integer values). The `sampled_from()` strategy allows developers to specify a sequence or list of values which the strategy will randomly select values from. As we can see in Table III, `integers()` is the

[9]https://github.com/aws/chalice/blob/79838b02dc330cfe549823899d2662ded0538015/tests/unit/test_app.py#L1819

[10]We selected these features from Hypothesis documentation. The first author manually marked, and counted the features used in each test.

most popular strategy appearing in more than one-third of the analyzed tests. The second and third most popular strategies are `floats()` and `sampled_from()`, respectively. Moreover, it is noteworthy that PBT can be used for text inputs, as `text()` is also present in some tests. Finally, the least used strategies in our sample are `times()` and `timedeltas()`.

TABLE III
USE OF HYPOTHESIS STRATEGIES

Strategy	#	%
integers()	30	34.8%
floats()	14	16.2%
sampled_from()	12	13.9%
text()	9	10.4%
lists()	4	4.6%
booleans()	3	3.4%
binary()	3	3.4%
times()	1	1.1%
timedeltas()	1	1.1%

Summary: Strategies are—without surprise—the most common Hypothesis feature detected in our study, used by 65.1% of the tests. However, other features are also common, such as settings decorators (25.5%) and `assume()` methods (15.5%). Regarding the strategies, they are used to generate inputs of most types, but integers are the most common ones (34.8%).

V. THREATS TO VALIDITY

We based the study on a sample of public Python repositories, which therefore may not be fully representative of the whole population of Python projects, including commercial ones. Second, we investigated only property-based tests implemented using Hypothesis. Despite being a popular tool nowadays, our findings might be different if other tools or languages are considered. Additionally, our results are based on a manual classification performed by a single person. Therefore, they might be subjected to errors or biased interpretations.

VI. RELATED WORK

Arts et al. [1] present a case study on the usage of PBT in an industrial implementation of the Megaco protocol (a pattern for controlling media gateways in telecommunications networks). In this case, the tests were implemented using the Quviq QuickCheck tool, which is a property-based testing tool for Erlang.[11] The article describes the input generators used by the tests, the features that are tested, and the methodology employed in the tests. The authors also commented that PBT was able to find significant failures that required non-trivial fixes. However, this study is based on a single application, while in our study we analyzed the PBTs implemented in 30 well-known Python projects.

[11]http://www.quviq.com/products/erlang-quickcheck

Hatfield-Dodds [6] presents four categories of properties that can be checked using PBT. He also provides examples of usage from two scientific applications in Python (Numpy and Astropy). These property categories include *Outputs within expected bounds*, *Metamorphic properties*, *Differential testing*, and *Round-trip properties*. Thus, in our study, we are considering more categories (ten vs four categories) and more systems (thirty vs two projects). Moreover, we manually and carefully classified the category of a sample of tests (RQ1) and the features used in their implementation (RQ2), which is not performed in Hatfield-Dodds' study.

VII. CONCLUSION AND FUTURE WORK

In this paper, we described an ongoing study on the usage of property-based tests in Python. After carefully selecting a sample of 86 tests from 30 well-known projects, we revealed that two properties are predominantly implemented by such tests, namely roundtrip and test oracles. These properties were found in 71% of the tests currently analyzed in the study. We also revealed the most common features that are used in such tests, namely generator strategies (65%), setting decorators (25%), and assume methods (15%). These findings might be useful to developers when implementing their first PBTs. They might also provide valuable insights to tool builders.

Our roadmap for future work includes the following working units: (a) extending our sample to evaluate more projects; (b) conducting a qualitative study with developers to strengthen our current findings; (c) characterize the type of bugs that are detected by PBT and contrast the results with bugs detected by other alternative testing approaches, such as snapshot testing [4]; (d) conduct large-scale quantitative studies on the use of features of PBT libraries.

ACKNOWLEDGMENTS

We were supported by grants from CNPq and FAPEMIG.

REFERENCES

[1] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with Quviq QuickCheck. In *ACM SIGPLAN Workshop on Erlang*, page 2–10, 2006.

[2] Z. Chen, C. Rizkallah, L. O'Connor, P. Susarla, G. Klein, G. Heiser, and G. Keller. Property-based testing: Climbing the stairway to verification. In *15th ACM SIGPLAN International Conference on Software Language Engineering*, page 84–97, 2022.

[3] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming*, page 268–279, 2000.

[4] V. G. Cruz, H. Rocha, and M. T. Valente. Snapshot testing in practice: Benefits and drawbacks. *Journal of Systems and Software*, 1:1–7, 2023.

[5] G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Enginnering Notes*, 22(4):74–80, 1997.

[6] Z. Hatfield Dodds. Falsify your Software: validating scientific code with property-based testing. In *19th Python in Science Conference*, pages 162–165, 2020.

[7] A. Löscher and K. Sagonas. Targeted property-based testing. In *26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 46–56, 2017.

[8] D. R. MacIver, Z. Hatfield-Dodds, and M. O. Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1–3, 2019.

[9] T. Winters, T. Manshreck, and H. Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly, 2020.