

UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Isadora de Oliveira

Property-Based Testing in Python: Empirical Insights

Belo Horizonte
2026

Isadora de Oliveira

Property-Based Testing in Python: Empirical Insights

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Marco Túlio de Oliveira Valente
Co-Advisor: Henrique Santos Camargo Rocha

Belo Horizonte
2026

2026, Isadora de Oliveira.
Todos os direitos reservados

Oliveira, Isadora de.

O48p Property-based testing in Python empirical Insights / Isadora de Oliveira. – 2026.

1 recurso online (79 f. il., color.) : pdf.

Orientador: Marco Túlio de Oliveira Valente.
Coorientador: Henrique Santos Camargo Rocha.

Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.

Referências: f. 74-77.

1. Computação – Teses. 2. Engenharia de software – Teses. 3. Software – Testes – Teses. 4. Python (Linguagem de programação de computador) – Teses. I. Valente, Marco Túlio de Oliveira. II. Rocha, Henrique Santos Camargo. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da computação. IV. Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Irenquer Vismeg Lucas Cruz
CRB 6/819 - Universidade Federal de Minas Gerais – ICEX



UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Property-Based Testing in Python: Empirical Insights

ISADORA DE OLIVEIRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Orientador
Departamento de Ciência da Computação - UFMG

Documento assinado digitalmente

gov.br

HENRIQUE SANTOS CAMARGOS ROCHA
Data: 03/03/2026 15:22:39-0300
Verifique em <https://validar.iti.gov.br>

PROF. HENRIQUE SANTOS CAMARGOS ROCHA - Coorientador
Department of Computer Science - Loyola University Maryland

PROF. ANDRÉ CAVALCANTE HORA
Departamento de Ciência da Computação - UFMG

Documento assinado digitalmente

gov.br

JOAO EDUARDO MONTANDON DE ARAUJO FILHO
Data: 03/03/2026 14:08:53-0300
Verifique em <https://validar.iti.gov.br>

PROF. JOÃO EDUARDO MONTANDON DE ARAÚJO FILHO
Ciência da Computação - UFMG

Belo Horizonte, 10 de fevereiro de 2026.

For the child who wanted, back then, to be where I am now.

Acknowledgments

Agradeço aos meus familiares, amigos e professores. Agradeço em especial:

Ao Anderson, que está sempre ao meu lado, por me encorajar nos momentos de incerteza, pela paciência, pelo cuidado e pelo companheirismo que nunca faltaram ao longo dessa jornada, e por sempre encontrar uma forma de colocar um sorriso no meu rosto.

Aos meus pais, pelo exemplo de dedicação e integridade, pelo apoio e incentivo na conclusão de mais esta etapa e por sempre me aconselharem a dedicar-me aos estudos.

Aos meus irmãos, agradeço pelo companheirismo, por estarem presentes quando preciso e por nunca medirem esforços para me apoiar.

À Marina, pela singela e preciosa amizade construída ao longo dos anos, pelas desventuras em série e por celebrarmos juntas cada vitória, minha e sua, como se fossem sempre nossas. Sua presença tornou essa caminhada mais leve, mais corajosa e muito mais feliz.

Aos amigos e colegas de trabalho, pelo apoio constante, pelas conversas que tornam os dias mais leves e pelos momentos de descontração ao final das sextas-feiras. Em especial às pessoas que passaram ou ainda fazem parte de um desses times: Compras, *Task Force*, Navegação e Home & Personalização, por compartilharem essa jornada comigo.

À Mestra Cristina Mara, professora de português, a única com esse título durante todo o meu ensino fundamental, que me inspirou a pensar: “um dia, serei eu”.

Ao Prof. Felipe Cunha, a quem tenho grande estima, pelo apoio e incentivo constantes aos meus estudos. No dia em que lhe contei que havia sido aprovada, vi em você uma alegria ainda maior do que a minha.

Ao meu orientador, Prof. Marco Túlio, e ao meu coorientador, Prof. Henrique Rocha, pela dedicação, apoio e valiosos ensinamentos ao longo de todo o

mestrado.

Ao Arthur, pela colaboração e contribuições que possibilitaram a extensão do Estudo 1.

Ao grupo de pesquisa ASSERG, pelo conhecimento compartilhado.

Aos membros da banca, Prof. André Hora e Prof. João Montandon, pela disponibilidade em participar deste trabalho, pela avaliação da dissertação e pelas contribuições que certamente aprimoraram este trabalho.

A UFMG e ao PPGC, pelo suporte acadêmico e pela oportunidade de realizar este mestrado, ambiente no qual pude crescer, aprender e desenvolver minhas habilidades de pesquisa.

Aos docentes e aos funcionários do PPGCC, que direta ou indiretamente contribuíram para a minha formação. Em especial, **Cristiane, Leonardo, Larissa e Aline**, pelos momentos de estudo e pelos trabalhos desenvolvidos em grupo — foi muito gratificante aprender ao lado de vocês —, e à **Sônia**, por todo o auxílio administrativo ao longo dessa trajetória.

“No matter how hard it gets. . . I’ll never give up.”
(Kamado Tanjiro)

Resumo

Testes Baseados em Propriedades (PBT) gera automaticamente entradas de teste para validar propriedades de programas, deslocando o esforço dos desenvolvedores de escrever exemplos para especificar invariantes. Embora a técnica tenha ganhado popularidade em Python por meio da biblioteca Hypothesis, pouco se sabe sobre como os desenvolvedores a adotam e utilizam na prática. Esta dissertação apresenta três estudos empíricos. Primeiro, analisamos 367 PBTs reais de 244 projetos em Python, classificando-os em nove categorias de propriedades e quantificando o uso de construções do Hypothesis. Constatamos que as propriedades do tipo *Test Oracle* predominam (29.97%) e que os PBTs são geralmente concisos (mediana de 14 LOC), baseando-se fortemente em estratégias internas (75.20%), mas também em externas (22.62%) e personalizadas (17.17%). Em segundo lugar, estudamos 213 postagens do Stack Overflow com a *tag* PBT, revelando que os principais desafios enfrentados pelos desenvolvedores dizem respeito às estratégias de geração de dados (36.62%), especialmente para dados compostos e tabulares (24.36%). Por fim, avaliamos o Ghostwriter, o gerador automatizado de testes do Hypothesis, em 203 testes reais; apenas 18.23% foram totalmente automatizáveis, enquanto a maioria exigiu adaptação parcial (30.05%) ou se mostrou incompatível (51.72%). Em conjunto, nossos achados fornecem a maior caracterização empírica de PBT em Python até o momento, destacam as dificuldades dos desenvolvedores em adotar a técnica e expõem limitações no suporte das ferramentas atuais.

Palavras-chave: teste baseado em propriedade; python; hypothesis; ghostwriter; Stack Overflow.

Abstract

Property-Based Testing (PBT) automatically generates test inputs to validate properties of programs, shifting developers' effort from writing examples to specifying invariants. While the technique has gained popularity in Python through the Hypothesis framework, little is known about how developers adopt and use it in practice. This dissertation reports on three empirical studies. First, we analyzed 367 real PBTs from 244 Python projects, classifying them into nine property categories and quantifying their use of Hypothesis constructs. We found that *Test Oracle* properties dominate (29.97%), and that PBTs are generally concise (median 14 LOC), relying heavily on built-in strategies (75.20%), but also on external (22.62%) and internal (17.17%) ones. Second, we studied 213 Stack Overflow posts tagged with PBT, revealing that the main challenges developers face concern data generation strategies (36.62%), especially for composite and tabular data (24.36%). Finally, we evaluated Ghostwriter, Hypothesis's automated test generator, against 203 real tests; only 18.23% were fully automatable, while most required partial adaptation (30.05%) or were incompatible (51.72%). Together, our findings provide the largest empirical characterization of PBT in Python to date, highlight developers' difficulties in adopting the technique, and expose limitations of current tool support.

Keywords: property-based testing; python; hypothesis; ghostwriter; Stack Overflow.

List of Figures

3.1	Methodology used to answer RQ1	44
3.2	Distribution of code lines for all categories	48
4.1	Example question in the Strategies for Data Generation category	58
4.2	Example question in the Framework Features category	59
4.3	Example question in the General Questions category	59
4.4	Example question in the Optimization and Performance category	60
4.5	Example question in the Use of Other Decorators category	60
4.6	Example question in the Integration with External Libraries category	60
4.7	Example question in the Reporting and Results Visualization category	61
4.8	Example question in the Composite Data subcategory	62
4.9	Example question in the DataFrame and Tabular Data subcategory	62
4.10	Example question in the Primitive Data subcategory	63
4.11	Example question in the Object Data subcategory	63
4.12	Example question in the Complex and Recursive Data Structures subcategory	64

List of Tables

2.1	Some strategies provided by Hypothesis	27
3.1	Distribution of the categories in the 367 tests of the dataset	46
3.2	Frequency of built-in constructs	49
3.3	Frequency of the built-in strategies	50
3.4	Frequency of each type of strategy	50
3.5	Distribution of external strategies by project	51
3.6	Frequency of extension strategies by extended library	52
3.7	Frequency of the @settings attributes	53
3.8	Frequency of deadline values	53
3.9	Frequency of max_examples values	54
4.1	Distribution of the question categories in the 213 questions of the sample . . .	58
4.2	Distribution of the types used in the Strategies for Data Generation category .	61
5.1	Eligibility distribution across categories	67

List of Code Listings

1.1	A simple example of Property-Based Testing (PBT)	18
2.1	Example of PBT that checks the associativity of integer addition	23
2.2	Example of PBT that checks the commutativity of the add function	23
2.3	Example of falsifying that was automatically generated by Hypothesis	23
2.4	Example of PBT that checks the returns a valid permutation of the original list	24
2.5	Example of falsifying shows a missing element in the permutation	24
2.6	Example of test with an integer strategy	26
2.7	Using assume to enforce a precondition in the test domain	27
2.8	Definitions of a composite strategy	28
2.9	Using @example to fix test cases	28
2.10	Using @settings to limit the number of generated test cases	29
2.11	Setting a fixed seed to ensure reproducible test inputs	29
2.12	Example of a test in the Roundtrip category	30
2.13	Example of a test in the Test Oracle category	30
2.14	Example of a test in the Outputs Within Expected Bounds category	31
2.15	Example of a test in the Different Paths Same Destination category	31
2.16	Example of a test in the Some Things Never Change category	31
2.17	Example of a test in the Idempotence category	32
2.18	Example of a test in the Hard to Prove Easy to Verify category	32
2.19	Example of a test in the Testing for Exceptions category	33
2.20	Example of a test in the Testing for Crashes category	33
3.1	Example of a PBT in the Test Oracle category	46
3.2	Example of a PBT in the Some Things Never Change category	46
3.3	Example of a PBT in the Hard to Prove Easy to Verify category	47
3.4	Example of a PBT in the Other category	47
3.5	Example of an external strategies	51
3.6	Example of an internal strategy to generate object identifiers (OIDs)	51
5.1	Example of an Eligible test in the Roundtrip category	67
5.2	Example of an Partially Eligible test in the Roundtrip category	68
5.3	Example of an Ineligible test in the Roundtrip category	68
A.1	Example of a skeleton generated for the Roundtrip category	78
A.2	Example of a skeleton generated for the Test Oracle category	78

A.3	Example of a skeleton generated for the Different Paths Same Destination category	78
A.4	Example of a skeleton generated for the Idempotence category	79
A.5	Example of a skeleton generated for the Testing for Exception category . .	79

List of Abbreviations and Acronyms

PBT	<i>Property-Based Testing</i>
LOC	<i>Lines of Code</i>
RQ	<i>Research Question</i>
JSON	<i>JavaScript Object Notation</i>
GCD	<i>Greatest Common Divisor</i>
LLM	<i>Large Language Model</i>
MT	<i>Metamorphic Testing</i>
LDA	<i>Latent Dirichlet Allocation</i>
NLP	<i>Natural Language Processing</i>
OID	<i>Object Identifier</i>
URL	<i>Uniform Resource Locator</i>
NaN	<i>Not a Number</i>

Contents

1	Introduction	17
1.1	Motivation	17
1.2	Proposed Work	19
1.3	Contributions	20
1.4	Outline of Dissertation	21
2	Background and Related Work	22
2.1	Property-Based Tests	22
2.2	Hypothesis	24
2.2.1	The Hypothesis Framework	24
2.2.2	Structure of a Hypothesis Test	25
2.2.2.1	Using the @given decorator	26
2.2.2.2	Defining Preconditions with assume	26
2.2.2.3	Creating Complex Strategies with @composite	27
2.2.2.4	Ensuring Critical Cases with @example	28
2.2.2.5	Configuring Behavior with @settings	29
2.2.2.6	Reproducibility with Seed	29
2.3	Categories of Properties	30
2.4	Ghostwriter Tool	33
2.5	Related Work	35
2.5.1	Studies of PBT in Functional Languages	36
2.5.2	Studies of PBT in Python	36
2.5.3	Studies of Related Testing Paradigms	38
2.5.4	Studies of PBT in ML Projects	38
2.5.5	Studies of Automatic Test Generation	39
2.5.6	Empirical Evaluations Using Stack Overflow Data	40
2.5.7	Other Studies	41
2.6	Final Remarks	41
3	Characterization Study	42
3.1	Dataset	43
3.2	Study Design	43
3.3	What are the most common categories of PBT? (RQ1)	45

3.4	Size Characterization	47
3.5	What are the most commonly used built-in constructs in Hypothesis? (RQ2)	49
3.5.1	Most Common Strategies	49
3.5.2	Most Common Settings	52
3.6	Threats to Validity	54
3.7	Final Remarks	55
4	Stack Overflow Study	56
4.1	Dataset and Study Design	56
4.2	Challenges Developers Face When Using the Hypothesis Framework	57
4.2.1	Questions on Data Generation	61
4.3	Threats to Validity	64
4.4	Final Remarks	65
5	Assessing a Tool to Generate PBTs	66
5.1	Study Design	66
5.2	Results	67
5.3	Threats to Validity	69
5.4	Final Remarks	69
6	Conclusion	70
6.1	Overview	70
6.2	Lessons Learned and Recommendations	71
6.3	Contributions	72
6.4	Future Work	73
6.5	Data Availability	73
	Bibliography	74
	Appendix A Skeletons generated by Ghostwriter	78

Chapter 1

Introduction

In this chapter, we present the motivation for this master’s dissertation in Section 1.1. Following that, in Section 1.2, we delineate the design of the proposed work. In Section 1.3 we highlight the most important contributions. Finally, in Section 1.4 we outline the remaining chapters of this dissertation.

1.1 Motivation

Automated testing has become a cornerstone of modern software development to ensure better efficiency and reliability when checking software quality, as software testing is commonly used throughout the development process to identify defects and establish confidence in program correctness [34, 23]. Automated tests are predominantly guided by examples [37]. For instance, when implementing a unit test, a developer first selects some values (i.e., examples) to call the function under test. Then, she calls the function using these values and checks whether the returned results match the expected ones. Even modern testing techniques such as Mutation Testing [35, 27], Test Amplification [2], and Exploratory Test [10] rely on example-based tests.

Despite being widely used, example-based tests require effort and expertise from developers for selecting the most effective test inputs, which is not a simple task. Thus, the idea behind Property-Based Testing (PBT) is to free developers from manually selecting test inputs [7, 8, 14, 25]. Instead, these values are randomly generated by the testing framework. The ultimate goal is to allow developers to spend more time fixing bugs than defining test case inputs. A common motto among PBT practitioners summarizes this goal: “test faster, fix more.”¹

Therefore, when using PBT, we do not verify a specific result of the function under test, since we do not know the exact input values used by the test, which are automatically generated by the testing framework. Instead, developers implement tests

¹<https://github.com/Hultner/Test-faster-fix-more>

that check properties that should hold for any possible input value. A simple example is shown in Listing 1.1, assuming a function `add(x,y)`. The first property-based test verifies the commutative property of addition, that is, the result of `add(x,y)` should be the same as the one returned by `add(y,x)`. The second test verifies the neutral element property, that is, `add(x,0)` should always return `x`, for any input `x`. Since the tests are called hundreds or even thousands of times by the testing framework using randomly generated inputs, the assumption is that existing bugs may be detected 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())
7 def test_add_commutative(x, y):
8     assert add(x, y) == add(y, x)
9
10 @given(x=st.integers())
11 def test_add_zero(x):
12     assert add(x, 0) == x
```

Listing 1.1: A simple example of Property-Based Testing (PBT)

Property-based testing was first proposed for functional languages more than 20 years ago. For instance, QuickCheck [8] is a well-known framework for implementing such tests in Haskell. More recently, PBT has gained traction in Python, mainly due to the popularity of the Hypothesis framework [25]. As of February 2026, the Hypothesis repository on GitHub has nearly 8.5k stars. In a survey conducted by JetBrains with more than 30,000 Python developers in 2024, it ranked sixth among the most widely used Python testing frameworks, being mentioned by 3% of respondents.² The frameworks with the highest number of mentions are aimed at unit testing, such as `pytest` (53%) and `unittest` (23%). Therefore, at least according to this data, Hypothesis currently represents the leading framework for implementing property-based testing in Python.

Nevertheless, despite this growing interest, there is still a lack of studies investigating how developers use PBT in practice. A notable exception is the study by Goldstein et al. [16], published in 2024, which conducted an empirical investigation of PBT based on 30 interviews at Jane Street, a financial company. The study’s goal was to understand how developers apply PBT to test OCaml programs and to identify benefits, limitations, and opportunities for advancing the technique. In addition, we chose a more popular programming language than OCaml and, consequently, we also consider different tools (Hypothesis and Ghostwriter).

Thus, in our view, more studies on the use of PBT remain important for two key reasons: (i) they can increase its awareness among practitioners and researchers; and (ii) they can reveal the properties most frequently checked in real implementations, as well as the challenges developers face when specifying them. The latter contribution is

²<https://lp.jetbrains.com/python-developers-survey-2024/>

particularly valuable for those implementing their first property-based tests. It may also have strategic value for framework builders, who could, for example, prioritize widely adopted features or deprecate rarely used ones.

1.2 Proposed Work

As stated previously, the motivations of this work is to increase awareness of PBT by filling the gap in the existing academic literature and providing the largest empirical characterization of PBT in Python to date. To achieve this goal, our work is divided into three main empirical studies focused on the Hypothesis framework: a characterization study, a Stack Overflow study, and an evaluation of a tool to generate PBTs.

- *Characterization Study:* The short paper by Corgozinho et al. [9] analyzed 86 property-based tests from real Python systems to answer two questions: (1) Which properties are most frequently tested by PBTs? and (2) Which Hypothesis constructs are most often used?

In this master dissertation, we extended the dataset used by Corgozinho et al. [9] to 367 property-based tests (an increase of over 300%). In the mentioned paper, the classification of properties checked by the tests was conducted by a single classifier. On the other hand, in this master dissertation, the properties were evaluated by two independent classifiers, improving the reliability of the classification process. This extended dataset also enabled us to identify two new PBT categories: *Testing for Exceptions* and *Testing for Crashes*. We further analyzed and quantified the Hypothesis constructs (such as `@given`, `strategies`, and `assume`) most frequently used in our dataset. Finally, we examined the distribution of lines of code (LOC) per PBT category, showing that PBTs are generally simple tests, with a median of 14 LOC.

- *Stack Overflow Study:* In a second and novel part of this master dissertation, we analyze a set of 213 Stack Overflow questions about PBTs. Our goal is to shed light on the challenges developers face when using this type of test and, consequently, provide feedback for improvements and adaptations in current PBT frameworks. We found that the most common questions are related to strategies for input data generation, accounting for 36.62% of the questions analyzed. Within this topic, questions about generating composite data (such as tuples, dictionaries, and other similar data structures) are the most frequent, representing 37.18% of the cases.

- *Assessing a Tool to Generate PBTs*: Finally, in a third and also novel study, we evaluate a test generation tool called Ghostwriter, which accompanies the Hypothesis framework. For this purpose, we used a dataset of 203 tests that check properties whose generation can, in principle, be automated with Ghostwriter. We then verified whether the tool would be able to generate these tests. An important conclusion is that the vast majority of real-world property-based tests are too complex and therefore incompatible with the testing templates produced by Ghostwriter. In fact, the tool would automate the generation of only 37 tests from our sample (18%), while for another 61 tests (30%) its assistance would be partial.

1.3 Contributions

We highlight the following contributions from the three studies presented in this master dissertation:

- The largest empirical characterization of PBT in Python to date, providing an in-depth analysis of developer adoption of the technique.
- A characterization study of 367 real PBT tests extracted from 244 Python projects, classifying them into nine property categories (including the two new proposed categories: *Testing for Exceptions* and *Testing for Crashes*). The quantitative results of this analysis show that the *Test Oracle* category is the most common (29.97%), that the tests are generally concise (median of 14 LOC), and that the use of Hypothesis strategies is predominant: 75.20% are built-in, but external (22.62%) and internal (17.17%) strategies were also observed.
- An investigation of PBT adoption challenges through the analysis of 213 posts on Stack Overflow, which revealed that the main difficulty for developers is in data generation strategies, especially for composite and tabular data.
- Evaluation of Ghostwriter, Hypothesis’s automated test generation tool, reveals that only 18.23% of actual tests would be fully automatable, thus exposing the limitations of current tool support.

In summary, the studies present practical recommendations for broadening the adoption of PBT. For beginners, we suggest to start with more accessible categories—such as *Roundtrip* and *Different Paths, Same Destination*—given their relative simplicity

and the extensive support provided by Ghostwriter. Regarding Hypothesis, we highlight the need for richer documentation on composite strategies and tools for addressing performance. Finally, for future research, we recommend investigating the use of Large Language Model (LLMs) for generating complex PBTs and expanding empirical analyzes to additional frameworks, such as `quickCheck` and `jqwik`.

1.4 Outline of Dissertation

The remainder of this dissertation is organized as follows:

- Chapter 2 provides an overview of PBT, describing the categories of properties that can be verified in this type of testing. It also includes a step-by-step guide on how to use the Hypothesis framework. We introduce the Ghostwriter tool, highlighting its role in the automatic generation of tests, and present a discussion of Related Work.
- Chapter 3 presents the Characterization Study, which aims to answer two research questions: (RQ1) *What are the most common categories of PBT?* and (RQ2) *What are the most commonly used built-in constructs in Hypothesis?*.
- Chapter 4 describes the Stack Overflow Study, in which we analyze questions asked by developers about PBT (RQ3). Our goal is to identify the main challenges faced in practice when using Hypothesis. This analysis helps us understand recurring difficulties and provides insights into the gaps between PBT theory and its real-world use in software development.
- Chapter 5, we evaluate the Ghostwriter tool, investigating its ability to automatically generate property-based tests (RQ4).
- Chapter 6 concludes the master dissertation with final considerations and suggestions for future work.

Chapter 2

Background and Related Work

We begin this chapter in Section 2.1 by introducing the fundamental concepts of Property-Based Testing (PBT), along with practical examples that illustrate its effectiveness in uncovering subtle bugs through automated input generation. In Section 2.2, we describe the Hypothesis framework—the main tool used throughout this dissertation—detailing its architecture, data generation and reduction mechanisms, and syntactic constructs, such as decorators and strategies. We then present the categories of properties typically verified by PBTs in Section 2.3. Following this, Section 2.4 introduces the Ghostwriter tool, which supports automated test generation. Section 2.5 analyzes related work on the use of PBT in academic and industrial contexts, identifying existing gaps that motivated our empirical studies. Finally, we provide concluding remarks in Section 2.6.

2.1 Property-Based Tests

Property-Based Testing (PBT) is a testing methodology that gained popularity through the family of QuickCheck libraries, originally developed in Haskell [25]. Unlike the traditional testing approach, which relies on concrete, manually specified examples to verify input–output pairs, PBT focuses on defining high-level properties that must hold across a wide range of inputs [25] [20] [24].

In this way, the focus of PBT shifts from simple verification to the active falsification of the specification. The central premise of PBT is that, instead of proving software correctness, the testing tool actively attempts to refute the properties defined by the developer, searching for counterexamples that cause the test to fail [20].

In contrast, in traditional unit testing, developers create examples that demonstrate how a function should behave, writing one test for each expected behavior [18]. Given these advantages, the application of PBT has proven to be powerful, as evidenced by its use in highly critical and complex industrial environments, such as those of companies like Jane Street [16] and Amazon [5].

However, for this methodology to be effective, the quality of data generation is a critical factor. The successful application of PBT requires not only well-defined properties but also well-designed random generators. A random generator is a program that defines the input space used to exercise the property, and not all generators are equally useful. A poor generator may fail to produce sufficiently diverse inputs to explore the possible code paths, or it may generate inputs that do not satisfy a property's preconditions (and thus are not useful for testing) [17].

We present below three practical examples implemented using the Hypothesis framework, a mature and widely adopted library for PBT in Python. Although Section 2.2 details the internal mechanisms of Hypothesis, these examples focus on the structure of a PBT and its ability to detect counterexamples.

Example 1: The test verifies the associative property of addition. Hypothesis automatically generates three integers (a , b , c). The assertion (line 7) validates the invariance of the result with respect to grouping order: $(a + b) + c$ must be equal to $a + (b + c)$, testing this property across a wide input domain.

```

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

```

Listing 2.1: Example of PBT that checks the associativity of integer addition

Example 2: In this second example, we demonstrate the effectiveness of PBT in identifying counterexamples. For this purpose, we deliberately introduce a bug into the `add` function. The failure occurs whenever the parameter x is greater than 10,000 (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)

```

Listing 2.2: Example of PBT that checks the commutativity of the add function

Thus, Hypothesis detects the following failure in this test:

```

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

```

Listing 2.3: Example of falsifying that was automatically generated by Hypothesis

The failure occurs because `add(10001, 1)` subtracts the input values (due to our bug) and therefore returns 10,000.

Example 3: The following test checks the behavior of the `sample(xs, len(xs))` function call (line 6). It should return a permutation of the original list `xs`, including all elements, with none missing and none duplicated. Therefore, sorting both the original list and the sampled list must yield the same result (line 7).

```
1 from hypothesis import given, strategies as st
2
3 @given(st.randoms(use_true_random=False))
4 def test_can_sample_from_whole_range(rnd):
5     xs = list(map(str, range(10)))
6     ys = rnd.sample(xs, len(xs))
7     assert sorted(ys) == sorted(xs)
```

Listing 2.4: Example of PBT that checks the returns a valid permutation of the original list

However, suppose that the `sample` function has a bug that causes it to always return permutations with one element missing. In this case, our test would fail as follows:

```
1 E AssertionError: assert ['0', '1', '2...'4', '5', ...] == ['0', '1', '2...'4', '5', ...]
2 E Right contains one more item: '9'
3 E Use -v to get more diff
4 E Falsifying example: test_can_sample_from_whole_range(
5 E   rnd=HypothesisRandom(generated data),
6 E )
```

Listing 2.5: Example of falsifying shows a missing element in the permutation

In this specific case, the element 9 is present in the original list `xs` but missing from the sampled list `ys`, which causes the assertion `sorted(ys) == sorted(xs)` to fail.

2.2 Hypothesis

2.2.1 The Hypothesis Framework

Hypothesis is a property-based testing library widely used in the Python language, with at least 200K direct downloads from the PyPI package¹ and it is used by thousands of open source projects.² We selected Hypothesis for this study because it is a mature and robust tool that incorporates the key constructs necessary for the implementation

¹<https://pypistats.org/packages/hypothesis>

²<https://github.com/HypothesisWorks/hypothesis/network/dependents>

of PBTs in Python, a language that boasts a rich ecosystem of scientific software, and Hypothesis is useful for ensuring its correctness [25].

The effectiveness of Hypothesis in the context of PBT is demonstrated by its architecture, which integrates data generation with a failure processing mechanism. This integration allows for: (1) the efficient exploration of the input domain, including edge cases, and (2) the automatic simplification of falsifying examples to aid in debugging.

1. *Data Generation*: The library employs a declarative syntax that allows developers to specify the properties that must hold true for all inputs within a given domain. Hypothesis then randomly selects and generates various inputs—including edge cases that can be easily overlooked in manual testing.³
2. *Shrinking Mechanism*: In addition to data generation, Hypothesis features an automatic shrinking mechanism that facilitates debugging by presenting a minimal failing example for each distinct failure [20]. When a property fails, the framework does not report the original complex example. Instead, it initiates an iterative process to minimize the falsifying example (as seen in Section 2.1), simplifying it to the smallest and most understandable dataset that still reproduces the bug. This shrinking process is fundamental because it simplifies the debugging process for the developer, mitigating one of the difficult points in PBT adoption: the complexity of defining generators for more elaborate data types [16].

Next, we detail the syntactic structure and the main components used for constructing a PBT using the framework.

2.2.2 Structure of a Hypothesis Test

The construction of a PBT test in Hypothesis is based on a declarative syntax, utilizing specific decorators and function calls that define the input domain and the properties to be verified.

³<https://hypothesis.readthedocs.io/en/latest/>

2.2.2.1 Using the `@given` decorator

Hypothesis's workflow begins with its primary decorator, `@given`, which acts as a bridge between the test function and the library's data generation engine. This decorator accepts one or more objects called strategies.

Strategies are parameterized data generators that define and produce a variety of inputs for the test function's arguments. Hypothesis provides a vast collection of strategies for primitive types (such as `integers`, `text`), complex data structures (`lists`, `dictionaries`), and for composing custom data types. Hypothesis generates each set of inputs (an example) and passes it to the test function.

In Listing 2.6, by using `st.integers()` as the parameter for the `@given` decorator, Hypothesis is instructed to generate integers for the parameter `s`. When this test is run, Hypothesis instantly fails because negative numbers violate the specified property. The shrinking mechanism will be activated, ensuring that the reported falsifying example is the value `-1`, as it is the simplest input that violates the property.⁴

```
1 from hypothesis import given, strategies as st
2
3 @given(st.integers())
4 def positive(s):
5     assert s >= 0
```

Listing 2.6: Example of test with an integer strategy

The Table 2.1 below describes some strategies provided by Hypothesis.

2.2.2.2 Defining Preconditions with `assume`

In many scenarios, a property is only valid under certain preconditions. The `assume` command is used to discard generated examples that don't satisfy a condition, without causing the test to fail.

If `assume(condition)` returns `false`, the current input example is discarded, and Hypothesis attempts to generate a new value until the example limit is reached. The use of `assume` is important for restricting the property's domain to valid inputs in the context of the software under test, ensuring that the test focuses only on scenarios where the property is indeed valid.

To illustrate the application of preconditions, we modified the property so that it is tested only with even numbers. In Listing 2.7, if an odd number is generated, it

⁴https://github.com/pschanely/CrossHair/crosshair/examples/hypothesis/bugs_detected/simple_strategies.py#L7

Table 2.1: Some strategies provided by Hypothesis

Strategies	Descriptions
<code>integers()</code>	Generates integers number
<code>floats()</code>	Generates floats numbers
<code>booleans()</code>	Generates booleans (true or false)
<code>text()</code>	Generates unicode strings
<code>list()</code>	Generates a list with elements from the strategy passed to it. Example: <code>st.lists(st.integers())</code> generates a list of integers
<code>tuples()</code>	Generates tuples of a fixed length <code>st.tuples</code> . Example: <code>st.tuples(st.integers, st.floats())</code> generates tuples with two elements, first element is integer and second is a float
<code>one_of()</code>	Generates from any of the strategies passed to it, <code>st.one_of(st.integer() st.floats())</code> generates either integers or floats
<code>builds()</code>	Generates instances of a class by specifying for each argument. Example: <code>st.build(Person, name=st.text(), age=st.integers())</code> generates a text for name and integer for age
<code>just()</code>	Generates the exact value passed to it. Example: <code>st.just("a")</code> generates the exact string "a"
<code>sampled_from()</code>	Generates a random value from a list. Example: <code>st.sampled_from(["a", 1])</code> is roughly equivalent to <code>st.just("a" st.just(1))</code>
<code>none()</code>	Generates None. Useful for parameters that can be optional. Example: <code>st.integers() st.none()</code>

is silently discarded and replaced until an even value is found. Thus, the precondition is made explicit through the `assume` command, while the property (invariant) is validated with `assert`.

```

1 from hypothesis import given, strategies as st
2
3 @given(st.integers())
4 def positive(s):
5     assume(s % 2 == 0) # Precondition: only even numbers
6     assert s >= 0

```

Listing 2.7: Using `assume` to enforce a precondition in the test domain

2.2.2.3 Creating Complex Strategies with `@composite`

Although Hypothesis provides a wide collection of built-in strategies (`st.integers()`, `st.text()`, `st.list()`), it is necessary to test more complex scenarios. To test cases where

data structures depend on specific domain constraints, we can create custom strategies using the `@composite` decorator. A `@composite` strategy allows sequential and conditional generation of input data, where one strategy can depend on the value generated by another.

In Listing 2.8, we demonstrate the definition of a composite strategy to generate a valid range (`min_val` and `max_val`), ensuring that the starting value is never greater than the ending value.

```

1 from hypothesis import given, strategies as st, composite
2
3 @composite
4 def valid_range(draw):
5     min_val = draw(st.integers(max_value=100))
6     max_val = draw(st.integers(min_value=min_val, max_value=200))
7     return (min_val, max_val)
8
9 @given(r=valid_range())
10 def positive(r):
11     assert r[0] <= r[1]
```

Listing 2.8: Definitions of a composite strategy

The use of `draw` (lines 4-6) allows the generation of `max_value` to be directly conditioned on the value generated for `min_val`. Applying `@composite` to enforce a precondition, rather than relying on runtime checks (e.g., `assume`), results in a more efficient testing process. Instead of discarding numerous invalid examples, the strategy generates only valid data, focusing the effort on finding counterexamples within the relevant domain.

2.2.2.4 Ensuring Critical Cases with `@example`

Although Hypothesis is effective at intelligently and randomly generating cases, high-quality PBT requires guarantees of coverage for specific and known values, especially in edge cases.

The `@example` decorator allows the injection of explicit values before random generation, ensuring that critical inputs are always tested. The arguments for `@example` must match the number and types of arguments defined by `@given`.

In Listing 2.9, the test is first executed with `s = 0` and `s = 1`. Hypothesis then generates the remaining 98 examples in a pseudo-random manner.

```

1 from hypothesis import given, strategies as st, example
2
3 @example(s=0) # Ensures that zero is always tested
4 @example(s=1) # Ensures that one is always tested
5 @given(st.integers())
6 def positive(s):
7     assert s >= 0
```

Listing 2.9: Using `@example` to fix test cases

2.2.2.5 Configuring Behavior with @settings

The execution behavior of a test can be adjusted using the `@settings` decorator. It configures parameters such as the number of generated examples, timeout, or verbosity settings for the output.

The `@settings` decorator should be applied before the `@given` decorator. In Listing 2.10 below, we reduce the maximum number of test examples to 10, a useful adjustment when execution time is a relevant factor:

```
1 from hypothesis import given, settings, strategies as st
2
3 @settings(max_examples=10) # Runs only 10 examples instead of the default 100.
4 @given(st.integers())
5 def positive(s):
6     assert s >= 0
```

Listing 2.10: Using `@settings` to limit the number of generated test cases

2.2.2.6 Reproducibility with Seed

To ensure reproducibility in test generation, Hypothesis allows the definition of a `seed`. Since PBT relies on pseudo-randomness, using a `seed` guarantees that the same sequence of inputs is reproduced in subsequent runs. This feature is useful for investigating bugs and ensuring that the minimal failing case reported by shrinking can be consistently reached.

```
1 import hypothesis
2
3 hypothesis.seed(42) # Fix the seed to ensure the order of examples
```

Listing 2.11: Setting a fixed seed to ensure reproducible test inputs

With `hypothesis.seed(42)`, the execution becomes deterministic: the same sequence of inputs is generated in subsequent runs, enabling consistent debugging.

2.3 Categories of Properties

In this section, we describe the key types of properties that can be checked by PBTs. Five of these categories are described in an online document cited by the Hypothesis documentation,⁵ while another—*Output Within Expected Bounds*—is mentioned in an article by Hatfield-Dodds et al. [20]. Additionally, *Idempotence* is discussed in the context of Ghostwriter (Section 2.4). The remaining two—*Testing for Exceptions* and *Testing for Crashes*—were identified and proposed by us based on our experience and findings during this study. Next, we present a description of each category:

There and Back Again (aka Roundtrip): This category applies when a function produces a result that is immediately accessed and returned by another function (e.g., writing a value to a file and then reading it back).

Thus, a distinctive feature of this category is the combination of an operation with its inverse, which should return the same value as the initial input. Another example is the process of serialization and deserialization of a string. Consider Listing 2.12, where a string `xs` is encoded to JSON using `hyperjson.dumps(xs)` and subsequently decoded back to its original format using `hyperjson.loads`. The test ensures that the value obtained after deserialization is identical to the original value (`xs`).⁶

```
1 @given(st.text())
2 def test_text(xs):
3     assert hyperjson.loads(hyperjson.dumps(xs)) == xs
```

Listing 2.12: Example of a test in the Roundtrip category

Test Oracle: This category applies when a well-known algorithm is used as an oracle to check the implementation of a novel or untested algorithm (e.g., testing a new sorting algorithm against a default implementation such as QuickSort).

Another example is shown in Listing 2.13. Consider the `square_root_mod_prime` function, which returns a valid square root modulo a prime number. In the assertion `assert calc * calc % prime == square` (line 5), the `square` acts as the Test Oracle. Instead of comparing against a pre-calculated value, the oracle uses the invariant to verify if the result (`calc`) is correct, ensuring that when it is squared and reduced modulo the prime, it reproduces the original square value.⁷

```
1 @given(st_num_square_prime())
2 def test_square_root_mod_prime(self, vals):
3     square, prime = vals
4
5     calc = square_root_mod_prime(square, prime)
```

⁵<https://fsharppforfunandprofit.com/posts/property-based-testing-2/>

⁶https://github.com/mre/hyperjson/tests/test_hypothesis.py#L21

⁷https://github.com/tlsfuzzer/python-ecdsa/src/ecdsa/test_numbertheory.py#L405

```
6 assert calc * calc % prime == square
```

Listing 2.13: Example of a test in the Test Oracle category

Outputs Within Expected Bounds: This category involves verifying whether the results of a function stay within expected limits. These bounds can be either computational or logical (e.g., if a function returns a price, the value should be non-negative).

Another example is shown in Listing 2.14. Consider a function `get_beta` (the Beta angle, crucial for the solar illumination of satellites), which verifies that this angle—physically defined as never exceeding $\pm 90^\circ$ —always returns a value between `-90` and `90` (line 3). The `@given(datetimes())` decorator ensures that this verification is carried out for numerous dates and times, validating the integrity and accuracy of the calculation’s physical limits.⁸

```
1 @given(datetimes())
2 def test_get_beta_always_between_m_90_and_90(non_sun_synchronous, when_utc):
3     assert -90 <= non_sun_synchronous.get_beta(when_utc) <= 90
```

Listing 2.14: Example of a test in the Outputs Within Expected Bounds category

Different Paths, Same Destination: This category applies when a combination of functions should produce the same result regardless of the order in which they are applied (e.g., a commutative property).

Consider Listing 2.15, which verifies the commutativity of the compatible function. Two annotations, `s1` and `s2`, are randomly generated, and the assertion ensures that the order doesn’t affect the result (line 3). This validates that the compatibility operation between annotations is independent of the order of the arguments.⁹

```
1 @given(s1=st_annotation, s2=st_annotation)
2 def test_commutativity_of_union_compatibility(s1, s2):
3     assert compatible(s1, s2) == compatible(s2, s1)
```

Listing 2.15: Example of a test in the Different Paths Same Destination category

Some Things Never Change: This category applies when an invariant is preserved between input and output of a transformation (e.g., the size of a list should not change after sorting).

Another example is shown in Listing 2.16. The function `util.roll_1d` performs a cyclic shift on an array by a given shift. The assertion `self.assertEqual(len(post), len(array))` (line 4) ensures that the array length is preserved after the operation.¹⁰

```
1 @given(get_array_1d(min_size=1), st.integers())
2 def test_roll_1d(self, array: np.ndarray, shift: int) -> None:
3     post = util.roll_1d(array, shift)
```

⁸https://github.com/satellologic/orbit-predictor/tests/test_predictors.py#L73

⁹https://github.com/mesalock-linux/mesapy/rpython/annotator/test/test_model.py#L173

¹⁰https://github.com/static-frame/static-frame/static_frame/test/property/test_util.py#L137

```

4 self.assertEqual(len(post), len(array))
5 self.assertEqualWithNaN(array[-(shift % len(array))], post[0])

```

Listing 2.16: Example of a test in the Some Things Never Change category

Idempotence: This category applies when repeated applications of an operation preserve the result, producing the same outcome as a single application (e.g., $f(f(x)) = f(x)$), ensuring stability.

An example is shown in Listing 2.17. Consider the test that generates random values and assigns them to the Amount setter twice on the same object. The first assignment sets the value; the second must not alter the state. The assertion `after_first_set == after_second_set` (line 10) ensures that repeating the assignment does not change the state beyond the first execution.¹¹

```

1 @given(st.integers())
2 def test_setter_is_idempotent(value: int):
3     obj = Dollar(0)
4     obj.Amount = value
5     after_first_set = obj.Amount
6
7     obj.Amount = value
8     after_second_set = obj.Amount
9
10    assert after_first_set == after_second_set

```

Listing 2.17: Example of a test in the Idempotence category

Hard to Prove, Easy to Verify: This category applies when a result can be easily verified, even if the method for computing it is more complex (e.g., checking the output of a tokenization algorithm is simpler than implementing the algorithm itself, since verification only requires concatenating the returned tokens).

Another example is shown in Listing 2.18. The `extended_euclid` (line 4) function is a variation of the Euclidean algorithm that returns the Greatest Common Divisor (GCD) `d` along with the Bézout coefficients `(x, y)`. The first two assertions (lines 5-6) check whether `d` divides both `a` and `b`, while the last one validates Bézout's identity,¹² confirming the correctness of both the GCD and the coefficients.¹³

```

1 @given(st.integers(min_value=1, max_value=1000), st.integers(min_value=1, max_value=1000))
2 def test_extended_euclid(a, b):
3     x, y, d = extended_euclid(a, b)
4     assert a % d == 0
5     assert b % d == 0
6     assert a * x + b * y == d

```

Listing 2.18: Example of a test in the Hard to Prove Easy to Verify category

Testing for Exceptions: This category includes tests that check whether the function under test throws a specific exception when called with invalid inputs or abnormal conditions.

¹¹<https://fsharpforfunandprofit.com/posts/property-based-testing-5/>

¹²<https://www.geeksforgeeks.org/engineering-mathematics/bezouts-identity-bezouts-lemma/>

¹³https://github.com/HPAC/matchpy/tests/test_utils.py#L34

Prior studies indicate that exceptional behaviors are common in practice and frequently under-tested [11, 21, 26].

Consider Listing 2.19. The test verifies that `decode_float` is resilient to invalid inputs. When the function encounters data that cannot be decoded, the test ensures that the correct error handling is applied. Specifically, it asserts that the exception raised, `BlackboxProtobufException` (line 5), is appropriate for the decoding context and is not mistakenly identified as another type of error, such as `EncoderException` (line 6).¹⁴

```
1 @given(buf=st.binary(max_size=100), pos=st.integers(max_value=200))
2 def test_decode_float(buf, pos):
3     try:
4         fixed.decode_float(buf, pos)
5     except BlackboxProtobufException as exc:
6         assert not isinstance(exc, EncoderException)
7         pass
```

Listing 2.19: Example of a test in the Testing for Exceptions category

Testing for Crashes: This category includes tests executed with random inputs for the sole purpose of checking whether the system crashes.

An example is shown in Listing 2.20. Note that there are no explicit assertions; the test only passes each generated string to the function (line 3) and verifies that it can process any input without crashing or raising exceptions.¹⁵

```
1 @given(import_string=st.text())
2 def test_fuzz_strip_syntax(import_string):
3     parse.strip_syntax(import_string=import_string)
```

Listing 2.20: Example of a test in the Testing for Crashes category

2.4 Ghostwriter Tool

Ghostwriter is a tool designed to assist in writing tests with Hypothesis. Its sole purpose is to generate a template of test functions, thus simplifying the adoption of Property-Based Tests in software projects.¹⁶ The tool can generate tests for the following categories: *Roundtrip*; *Test Oracle*; *Different Paths, Same Destination*; *Idempotence*; and *Testing for Exceptions*.

Roundtrip: To generate tests for this category, testers should use the `roundtrip()` function. Suppose a program includes `encoded` and `decoded` functions for converting a string into bytes

¹⁴https://github.com/nccgroup/blackboxprotobuf/lib/tests/py_test/test_exceptions.py#L114

¹⁵https://github.com/PyCQA/isort/tests/unit/test_parse.py#L60

¹⁶<https://hypothesis.readthedocs.io/en/latest/reference/integrations.html>

and then decoding it back. By calling:

```
hypothesis.extra.Ghostwriter.roundtrip(encoded, decoded)
```

Ghostwriter generates the following test:

```
1 @given(text=st.text())
2 def test_roundtrip_encoded_decoded(text):
3     bytes = encoded(text)
4     value = decoded(bytes)
5     assert text == value
```

Test Oracle: When using Ghostwriter, one way to generate tests for this category is through the `equivalent()` function. Suppose a program with two sorting functions: `quicksort` and `my_sort`. The first is a well-known, widely tested implementation, while the second is an alternative version that needs verification. Thus, `quicksort`¹⁷ serves as an oracle for testing `my_sort`. Specifically, the following Ghostwriter call:

```
hypothesis.extra.Ghostwriter.equivalent(quicksort, my_sort)
```

generates the test:

```
1 @given(lst=st.nothing())
2 def test_equivalent_quicksort_my_sort(lst):
3     result_quicksort = quicksort(lst)
4     result_my_sort = my_sort(lst)
5     assert result_quicksort == result_my_sort
```

It is noteworthy that `st.nothing()` is a placeholder generated by Ghostwriter. It should be replaced with a valid strategy like `st.lists(st.integers())` in practical usage.

Different Paths, Same Destination: When using Ghostwriter, one way to generate tests for this category is through the `binary_operation()` function. Suppose a function `add(a, b)` that takes two numbers and returns their sum. Then, we want to verify whether our implementation satisfies the associative property of addition. By calling:

```
hypothesis.extra.Ghostwriter.binary_operation(add, associative=True, commutative=False, identity=None)
```

Ghostwriter generates the following test:

```
1 @given(lst=st.nothing())
2 @given(a=st.nothing(), b=st.nothing(), c=st.nothing())
3 def test_associative_binary_operation_add(a, b, c):
4     left = add(a, add(b, c))
5     right = add(add(a, b), c)
6     assert left == right
```

By defining `associative=True`, we instruct Ghostwriter to generate tests for the associative property of the `add` function. Similarly, `commutative=False` and `identity=None` exclude tests for the commutative and identity properties.

¹⁷<https://www.geeksforgeeks.org/dsa/python-program-for-quicksort/>

Idempotence: When using Ghostwriter, one way to generate tests for this category is with the `idempotent()` function. Suppose we want to test a function `normalize()`, which takes a string, removes all whitespace, and converts all letters to lowercase. Then, by calling:

```
hypothesis.extra.Ghostwriter.idempotent(normalize)
```

Ghostwriter generates the following test:

```
1 @given(data=st.nothing())
2 def test_idempotent_normalize(data):
3     result = normalize(data)
4     repeat = normalize(result)
5     assert result == repeat
```

Testing for Exceptions: When using Ghostwriter, one way to generate tests for this category is through the `fuzz()` function. Suppose we want to test whether a function `divide(x, y)` raises a `ValueError` when `y` is zero. By calling:

```
hypothesis.extra.Ghostwriter.fuzz(divide, except_=(ValueError,)).
```

Ghostwriter generates the following test:

```
1 @given(x=st.nothing(), y=st.nothing())
2 def test_fuzz_divide(x, y):
3     try:
4         divide(x, y)
5     except ValueError:
6         reject()
```

Finally, it is worth noting that the Ghostwriter does not support generating tests for the following categories: *Output Within Expected Bounds*; *Some Things Never Change*; *Hard to Prove, Easy to Verify*; and *Testing for Crashes*.

2.5 Related Work

We organized the discussion of related work into six parts: studies using functional languages; studies using Python (in generic domains); studies comparing PBT with related testing paradigms; studies using Python with a focus on machine learning projects; studies of empirical evaluations of test effectiveness; and studies of empirical evaluations based on Stack Overflow.

2.5.1 Studies of PBT in Functional Languages

Arts et al. [4] 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.¹⁸ 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, whereas in our study we analyzed the PBTs implemented in 244 well-known Python projects.

Goldstein et al. [16] conduct a qualitative study in an industrial context at the financial technology company Jane Street, investigating the use of PBT through 30 interviews with developers who use the QuickCheck tool. The study identifies four recurring types of properties employed in practice, including the *Roundtrip* property, which is also analyzed in our study. About one-third of the participants reported that PBTs uncovered bugs not detected by other testing methods, reinforcing their potential to identify subtle faults in complex code. Despite these benefits, the authors highlight important challenges, such as the difficulty of deciding which properties to test and the cognitive cost of writing specifications, which leads many developers to adopt PBT opportunistically, restricting its use to properties considered obvious. In contrast, our study covers a broad set of property categories, offering support to help developers identify and select the properties to be tested.

2.5.2 Studies of PBT in Python

The study by Corgozinho et al. [9] investigated the practical implementation of PBT in Python projects, using the popular Hypothesis framework. The research analyzed a sample of 86 tests extracted from 30 prominent Python repositories (including PyTorch, NumPy, and Polars) with the goal of characterizing the most commonly verified properties (RQ1) and the most used features of the framework (RQ2). Their preliminary results revealed the predominance of two property categories, covering 70.9% of the analyzed tests: *Roundtrip* (37.2%) and *Test Oracle* (33.7%). Additionally, the most employed

¹⁸<https://www.quviq.com/products/>

Hypothesis feature is the input generation strategy (present in 65.1% of the tests), notably the generation of integer values (34.8%), followed by the use of the `@settings` decorator for configuration (25.5%) and the `assume()` method to validate data preconditions (15.5%). These findings provide insights for developers and for the evolution of PBT tools, by signaling the most adopted testing approaches and functionalities in practice. Our Study 1 (detailed in Section 3) extends this investigation by Corgozinho et al. [9] by analyzing a substantially larger sample of 367 PBTs in Python projects. To ensure classification reliability, the manual analysis and categorization of these tests were conducted by two reviewers, which allowed us to achieve methodological consensus and address the research questions (RQ1 and RQ2) with greater robustness, depth, and scope.

An empirical study evaluates the use and effectiveness of PBT in Python using Hypothesis. Conducted by Ravi et al. [29], this study proposes a taxonomy consisting of twelve property categories and analyzes their fault-detection capabilities through mutation testing. Their categorization is based on the assertions present in each test function, whereas our study performs categorization at the test level; however, the authors observed that a test function contains, on average, 1.65 distinct categories, indicating that PBT tests rarely combine multiple property types within a single function. They also identified recurring patterns, such as tests focused on exception raising, a pattern that is likewise observed in our dataset. Although their study provides important insights into the effectiveness of different PBT categories, it primarily focuses on evaluating error-detection performance. In contrast, our work investigate developers' usage patterns, practical challenges, and tool-support limitations, taking a broader perspective on the practical use of PBT in real-world projects. Moreover, our results complement existing findings by confirming similar trends, such as the prevalence of Roundtrip properties (13.90% in our study versus 13.84% in theirs), while also providing new evidence on practices, adoption, and tool-support limitations that were not examined in prior work.

Hatfield-Dodds et al. [20] proposed four categories of properties relevant to property-based testing in scientific software, demonstrating how categories like *Outputs Within Expected Bounds*, *Roundtrip*, *Differential Testing*, and *Metamorphic* can help reveal bugs in well-known libraries such as NumPy and Astropy. While their work illustrates the implementation of these categories through specific examples, it does not quantify the practical occurrence of each. In our study, we adopted two of these categories, *Outputs Within Expected Bounds* and *Roundtrip*, to classify the tests in our dataset. Moreover, our study quantifies the occurrence of eight categories (as presented in Subsection 2.3) in a dataset of 367 PBTs implemented in open-source projects.

2.5.3 Studies of Related Testing Paradigms

Metamorphic Testing (MT) [6, 22, 28] is a software testing approach introduced in 1998 that eliminates the need for a traditional *Test Oracle* of expected results. Instead of checking expected outputs, this approach verifies metamorphic relations which essentially are inherent properties of the function under test that must hold across a range of inputs, including randomly generated ones. Metamorphic Testing is similar to property-based testing, as both verify immutable properties over random inputs. One key difference is that PBT provides its own customizable input generation strategies, whereas MT offers no such facility, requiring developers to implement the generation strategies manually. Another difference is that MT can use random inputs to generate multiple example-based test cases, thereby increasing the total number of test cases. Moreover, MT does not “shrink” failing inputs, which may make it harder to isolate faulty examples.

Both Differential Testing and PBT are designed to find subtle bugs, especially in complex systems where traditional examples would be insufficient [16]. Differential Testing uses the same input to run two versions of a software system: a base version and a test version. The base version is a verified/tested version of the system, while the test version is the modified one. The outputs from both runs are then meticulously compared to find abnormalities that could lead to potential bugs, all using the same input [19]. PBT, on the other hand, validates whether a system property or invariant remains true for a wide range of automatically generated inputs [14]. While the article [19] focuses on the rapid adoption of the technique by hundreds of teams at Google, it doesn’t mention the ability to ‘shrink’ the inputs that cause a failure. This functionality is a central aspect of PBT that simplifies debugging, making failure analysis more straightforward compared to other techniques.

2.5.4 Studies of PBT in ML Projects

Wauters et al. [36] focus on PBTs in ML projects, whereas our study analyzes individual PBTs across diverse domains, including web systems, libraries, and general-purpose software. Despite this difference, both studies highlight the relevance of automatic data generation: Wauters et al. observed positive developer perceptions, and our Stack Overflow analysis shows most questions relate to data generation in Hypothesis. They report

complex strategies as dominant, while our characterization shows integers and booleans among the most used. Additionally, `hypothesis.extra` (NumPy) is the most frequently used extension in our dataset. Finally, both studies note PBTs are often applied to conventional code rather than ML-specific components, aligning partially with our findings.

In the study by Durelli et al. [12], the technique is applied to test data generation for machine learning models. The authors propose a method that uses decision trees to intelligently create test data. This allows them to explore how the machine learning model behaves in various situations, helping to find errors that common tests might not detect.

2.5.5 Studies of Automatic Test Generation

Several studies have investigated software testing practices through large-scale empirical evaluations based on controlled experiments and quantitative metrics. Kracht et al. [23] analyzed automatically generated and manually written test suites from 100 open-source Java projects, assessing test quality using branch coverage and mutation scores. Their study followed a standardized experimental protocol, with multiple executions under fixed computational budgets and statistical analyses to compare different tools.

Similarly, Shamshiri et al. [31] conducted a large-scale evaluation of automated unit test generation using 357 real faults from the Defects4J dataset. They measured test effectiveness in terms of fault detection and applied repeated executions and statistical tests to ensure result reliability. While these studies rely primarily on automated metrics and controlled experimental settings, our work adopts a different empirical perspective. In contrast, we combine manual classification by independent evaluators, analysis of developer discussions, and tool evaluation to provide a more comprehensive view of testing practices in real-world settings.

Etemadi et al. [13] propose an LLM-based approach for the automatic generation of PBTs aimed at protecting cyber-physical systems (CPSs). The work introduces the ChekProp tool, which generates PBTs at design time from documentation, Python source code, and unit tests, and reuses them at runtime as mechanisms for monitoring unsafe states. The tests are implemented using Hypothesis and follow a structure composed of generators, a test body, and assertions. Although this structure is similar to that of Ghostwriter analyzed in our study, ChekProp is specific to CPSs and relies heavily on existing artifacts, whereas Ghostwriter is a general-purpose tool applicable across different domains. The authors report a high recovery of properties compared to those defined

manually, indicating the potential of LLMs to reduce human effort in critical domains.

Aichernig et al. [3] investigate the use of LLMs to generate code and PBTs in Scala based on an Event-B-inspired model, in the context of a bridge traffic control system. Unlike more guided approaches, the authors delegate the near-complete generation of code and tests to ChatGPT, exploring two modeling strategies. The results reveal significant difficulties, such as faulty properties, violations of preconditions, and substantial effort required to understand and correct the generated code after multiple iterations. Taken together, these studies show that while LLMs are promising for supporting PBT generation, their effectiveness strongly depends on the domain, the level of guidance provided, and the complexity of the model, which motivates our empirical analysis of Ghostwriter's use in real-world Python projects.

2.5.6 Empirical Evaluations Using Stack Overflow Data

The study by Tahaei et al. [33] also focuses on analyzing the developer community on Stack Overflow, with the primary goal of understanding privacy-related challenges. The authors adopted a mixed-methods approach, using Latent Dirichlet Allocation (LDA) topic modeling on 1,733 questions and a detailed qualitative analysis of a sample of 315 questions. This approach led to the identification of key themes such as "privacy policies" and "access control." This methodology is similar to our study, which also uses an empirical analysis of Stack Overflow questions to gain insights into developer difficulties. Their rigorous approach, which includes inter-rater validation with a Kappa of 0.77, reinforces the reliability of their findings. However, while the study by Tahaei et al. [33] focuses on privacy challenges, our study, which analyzed 213 questions, concentrates on the main challenges in PBT. Finally, the study by [33] also included an analysis of accepted answers, where a single researcher classified them into categories, such as those that provided a solution, those that included links to official documentation, or those with no reference to any external resources.

Abdellatif et al. [1] investigate the challenges and topics of interest encountered by chatbot developers, an area that demands specialized expertise in Machine Learning and conversational design. The work utilizes Stack Overflow as its primary data source to provide insights that bridge the knowledge gap regarding the practical problems of chatbot development. The authors employ a Topic Modeling approach (specifically LDA) to identify and classify the themes discussed on Stack Overflow, such as conversation

design, Natural Language Processing (NLP), and framework integration. Beyond mapping the topics, the research examines their popularity and difficulty, thereby characterizing the current development landscape. Ultimately, the study offers a taxonomy of practical problems, aiming to guide the research community, framework providers, and companies in offering more effective support to these developers.

2.5.7 Other Studies

Sun et al. [32] use PBT to validate user privacy functionalities in social media. Through the PDTDroid tool, privacy specifications are converted into formal properties, enabling automatic detection of inconsistencies, such as data leaks. Meanwhile, Xiong et al. [38] employ PBT to detect functional bugs that do not necessarily result in application crashes in Android apps. An example is when, upon removing a tag from OmniNotes, another tag in the same note or in the tag list is incorrectly altered or truncated. To address this, the authors developed the Kea tool, which combines a property description language with automated exploration strategies to verify the expected behavior of applications.

2.6 Final Remarks

This chapter provides an overview and discusses the related work concerning the core themes addressed in this dissertation. We begin by describing the theoretical foundations of PBT, explaining its principles, workflow, and essential constructs within the Hypothesis framework. Following this, we present a detailed tutorial on the syntax and functionality of Hypothesis. We also discuss the main categories of verifiable properties through PBT, illustrated with examples from our dataset (Section 3.1). Finally, we review prior studies that contextualize the growing adoption of PBT in both academia and industry. These concepts form the groundwork for the next chapter, where we empirically characterize PBT practices in real-world Python projects.

Chapter 3

Characterization Study

This chapter presents the first empirical study of this dissertation, which aims to characterize the use of PBT in real-world Python projects. In particular, we seek to answer two research questions:

RQ1: *What are the most common categories of PBT?* This question is important because it can guide the adoption of PBT by revealing the properties most frequently tested. For example, developers can begin implementing PBTs for functions in their systems that exhibit these same properties. In other words, since such properties are more common, identifying suitable functions tends to be easier.

RQ2: *What are the most commonly used built-in constructs in Hypothesis?* As with RQ1, the goal is to support developers in taking their first steps toward adopting PBT. By built-in constructs, we mean those provided by Hypothesis, such as strategies (`@given`) and other decorators. Specifically, we aim to identify the constructs most commonly used in practice. This allows developers to focus on a core subset of constructs, rather than learning the entire set supported by Hypothesis.

The remainder of this chapter is organized as follows. The methodology and the presentation of results to answer these RQs are structured in the following sections. Section 3.1 describes the construction of the dataset, detailing the collection and selection of the sample. Section 3.2 describes the study design and the methodology adopted to answer the two previously described research questions (RQ1 and RQ2). Section 3.3 the classification results and the quantitative distribution of the identified categories. Section 3.4 examines the size and complexity of the tests, while Section 3.5 analyzes the usage patterns of Hypothesis strategies and configuration constructs. Section 3.6 discusses the threats to validity. Finally, Section 3.7 provides the final remarks and transitions to the next study, which addresses the challenges developers face when adopting PBT in practice.

3.1 Dataset

To answer the proposed RQs, we retrieved a list of projects that depend on Hypothesis using the `github-dependents-info` feature from GitHub.¹ This feature provides information about GitHub repositories that depend on a given project. In the case of Hypothesis, the resulting list included 367 repositories. However, after an initial inspection, we found projects that import Hypothesis but do not use the library in any file. To discard such false positives, we implemented a Python script to search for real usages of Hypothesis in the selected repositories. For this task, we used the PyGithub² and GitPython³ libraries to retrieve the repositories' URLs and clone them, respectively.

As a result we selected 244 repositories that contain code depending on Hypothesis. Our final dataset includes widely used projects such as PyTorch (a machine learning framework), NumPy (a scientific computing package), and Polars (a dataframe library). In these 244 projects, we identified 7,612 property-based tests. However, since our analysis required an extensive manual effort, we chose to evaluate a random sample of 367 test cases (covering 108 of the 244 projects). The sample size was computed using the standard statistical procedure for estimating population proportions, assuming a 95% confidence level and a 5% margin of error.

3.2 Study Design

To answer RQ1, we followed the process described in Figure 3.1. Essentially, two authors independently and manually classified the 367 tests selected for the study—more than four times the number considered in our preliminary study. This classification was conducted in two rounds. In the first round, the authors focused on classifying the tests using only three popular categories from our initial study: *Roundtrip*, *Test Oracle*, and *Output Within Expected Bounds*. Tests that did not fit into any of these categories were labeled as *Other*. We chose to begin with popular and more easily identifiable categories to foster a shared understanding of PBTs and the proposed classification criteria between the two authors. The authors then met to compare their results, discuss disagreements, and reach a consensus. In total, 177 tests were classified into the initially selected categories. In the second round, the authors followed a similar process to classify the remaining tests

¹<https://github.com/nvuillam/github-dependents-info>

²<https://pygithub.readthedocs.io>

³<https://gitpython.readthedocs.io>

into the other categories, again working independently and holding a second and final consensus meeting. The Kappa coefficient ⁴ computed after the two rounds of classification was 0.54, indicating a moderate level of agreement between the authors. However, this result can be considered satisfactory given the complexity involved in analyzing real-world tests developed by third parties, which often depend on multiple functions and modules. It is also worth noting that the consensus steps required at least 20 hours of discussion across both rounds.

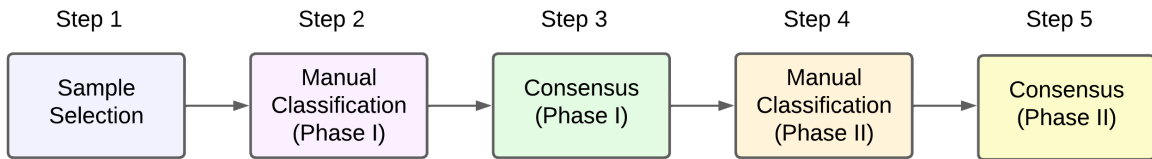


Figure 3.1: Methodology used to answer RQ1

After completing the analysis of RQ1, we also conducted an exploration of the size of the PBTs in our dataset, measured in lines of code (LOC). This analysis was performed by category and is reported in Section 3.4.

To address RQ2, the first author conducted a careful analysis of all tests in our dataset. The goal was to identify the most frequently used built-in constructs from Hypothesis. Specifically, the author searched for and counted the usage of the following constructs:

- *strategies* (such as `@integers`, `@floats`, and `@text()`), which are used to define how test inputs are generated. For example, `@given(x=integers(), y=integers())` generates a wide range of integer pairs to test a function like `add(x, y)`.
- *@settings*, which is used to configure test parameters, such as the number of times a test should be executed. For example, `@settings(max_examples=200)` increases the number of generated test cases from the default value to 200.
- *assume()*, which is used to specify conditions that input values must satisfy. For example, `assume(y != 0)` can be used to skip test cases where `y` is zero in a division operation.
- *@example*, which is used to ensure that a specific input example is always tested. For example, `@example(x=0, y=1)` guarantees that the test is executed with `x = 0` and `y = 1`, regardless of the generated data.

⁴Cohen’s Kappa coefficient measures the level of agreement between two raters while accounting for agreement that could occur by chance [15]. It ranges from -1 to 1, where 0 indicates chance-level agreement and higher values indicate stronger agreement.

- *note()*, which is used to add extra information when a failure occurs. For example, `note(f"Result of divide(x, y): result")` can help track the output that led to the failure.

Summary of methodological changes compared to previous study [9]: we expanded the dataset from 86 to 367 property-based tests (an increase of over 300%); whereas in the short paper, the classification of properties was conducted by a single classifier, in this dissertation, the properties are evaluated by two independent classifiers, thereby improving the reliability of the classification process. Additionally, the analysis of Stack Overflow posts related to PBTs (Chapter 4) and the effectiveness of the Ghostwriter tool for automatically generating PBTs (Chapter 3) are entirely new.

Summary of methodological changes compared to previous study [9]: we expanded the dataset from 86 to 367 property-based tests (an increase of over 300%); whereas in the short paper, the classification of properties was conducted by a single classifier, in this dissertation, the properties are evaluated by two independent classifiers, thereby improving the reliability of the classification process. In addition, categories reported in the previous study—such as *Solve a Smaller Problem First*, *The More Things Change, the More They Stay the Same*, *Model-Based Testing*, and *Metamorphic Properties*—were not observed in our expanded dataset, suggesting differences in dataset composition or evolution in PBT usage. Additionally, the analysis of Stack Overflow posts related to PBTs (Chapter 4) and the effectiveness of the Ghostwriter tool for automatically generating PBTs (Chapter 3) are entirely new.

3.3 What are the most common categories of PBT? (RQ1)

Table 3.1 shows the classification results for the 367 PBTs. The most common category is *Test Oracle*, accounting for nearly 30% of the tests. It is followed by *Some Things Never Change* (17.44%), *Hard to Prove, Easy to Verify* (16.08%), and *Roundtrip* (13.90%).

The four least common categories are *Testing for Exceptions* (4.90%), *Testing for Crashes* (4.36%), *Output Within Expected Bounds* (4.36%) and *Idempotence* (a category for which we could not find any PBT instance).

Next, we present examples of PBTs from each of the three most popular categories in our study. First, in the Listing 3.1, we show an example of *Test Oracle*, extracted from

Table 3.1: Distribution of the categories in the 367 tests of the dataset

Categories	#	%
Test Oracle	110	29.97
Some Things Never Change	64	17.44
Hard to Prove, Easy to Verify	59	16.08
Roundtrip	51	13.90
Different Paths, Same Destination	24	6.54
Testing for Exceptions	18	4.90
Testing for Crashes	16	4.36
Output Within Expected Bounds	16	4.36
Idempotence	0	0.00
Other	9	2.45
Total	367	100.00

the `pola-rs/polars` project:⁵

```

1 @given(value=strategy_datetime_ms)
2 def test_datetime_ms(value: datetime) -> None:
3     result = pl.select(pl.lit(value, dtype=pl.Datetime("ms"))["literal"])[0]
4     expected_microsecond = value.microsecond // 1000 * 1000
5     assert result == value.replace(microsecond=expected_microsecond)

```

Listing 3.1: Example of a PBT in the Test Oracle category

This test targets a function called `pl.Datetime("ms")`, which converts a datetime value to millisecond precision. Since the original datetime already includes microseconds, the test computes an expected value by truncating the microseconds to the nearest lower millisecond using integer division. The `assert` statement then verifies that the value returned by the library (`result`) matches this expected value. Because the expected output is computed using a standard Python expression, the test was classified as a *Test Oracle*.

Next, in Listing 3.2, we present an example from the *Some Things Never Change* category, extracted from the `dbrattli/Expression` project.⁶ This test checks whether the function `block.of_seq` preserves the length of the input list `xs`—that is, whether the length remains unchanged after the transformation.

```

1 @given(st.lists(st.integers()))
2 def test_block_len(xs: List[int]):
3     ys = block.of_seq(xs)
4     assert len(xs) == len(ys)

```

Listing 3.2: Example of a PBT in the Some Things Never Change category

Our third example, shown in Listing 3.3, belongs to the *Hard to Prove, Easy to Verify* category. It was extracted from the `google/caliban` project.⁷ This test verifies the

⁵https://github.com/pola-rs/polars/py-polars/tests/parametric/test_lit.py#L27

⁶https://github.com/dbrattli/Expression/tests/test_block.py#L180

⁷https://github.com/google/caliban/tests/caliban/util/test_util.py#L120

behavior of a function that merges two dictionaries, `m1` and `m2`. First, it calls the `merge` function (line 3) to produce a new dictionary, `merged`. Then, it checks whether each key from `m2` has been correctly inserted into `merged` (lines 5–6). Finally, it ensures that all key-value pairs from `m1` are present in `merged`, unless a key was overwritten by `m2`.

```

1 @given(st.dictionaries(st.text(), st.text()), st.dictionaries(st.text(), st.text()))
2 def test_merge(m1, m2):
3     merged = u.merge(m1, m2)
4     # Every item from the second map should be in the merged map.
5     for k, v in m2.items():
6         assert merged[k] == v
7
8     # Every item from the first map should be in the merged map, OR,
9     # if it shares a key with m2, m2's value will have bumped it.
10    for k, v in m1.items():
11        assert merged[k] == m2.get(k, v)

```

Listing 3.3: Example of a PBT in the Hard to Prove Easy to Verify category

As can be seen in Table 3.1, 2.54% of the tests do not fit into any of the proposed categories, as they do not validate properties that can be generalized. Listing 3.4 shows an example of such a test, which generates random values for a case⁸ object but checks a very specific condition—such as whether a value is equal to a given number.

```

1 @given(case=strategy)
2 @settings(deadline=None, suppress_health_check=list(HealthCheck), phases=[Phase.generate])
3 def test(case):
4     assert "\x00" not in case.query["q1"]["foo"]
5     assert len(case.query["q2"]["foo"]) == 4
6     assert int(case.query["q2"]["foo"]) % 2 == 0

```

Listing 3.4: Example of a PBT in the Other category

Summary: After analyzing 367 PBTs, we found that the most common category by far is *Test Oracle* (29.97%), followed by *Some Things Never Change* (17.44%). The least common categories are *Testing for Crashes* (4.36%), *Output Within Expected Bounds* (4.36%) and *Idempotence* (0%). While this should not be the sole criterion for selecting which PBTs to implement first, we believe this information can be helpful for those beginning to adopt property-based testing.

3.4 Size Characterization

To provide more information and context for PBT implementers, we also calculated the distribution of lines of code in the tests classified under each of the categories.

⁸https://github.com/schemathesis/schemathesis/test/specs/openapi/test_examples.py#L806

Figure 3.2 shows boxplots of the resulting distributions. When analyzed collectively (All), the median test size is 14 LOC. In other words, PBTs are not large tests in terms of lines of code. The Other category has the highest median at 30 LOC, while the *Different Paths, Same Destination* category has the lowest median at 8 LOC. On the other hand, some categories have tests with significantly divergent sizes. For example, the *Test Oracle* category has a mean of 34.08 LOC and a standard deviation of 38.60 LOC. In fact, the mean is greater than the median in all of the categories analyzed, suggesting a right-skewed distribution.

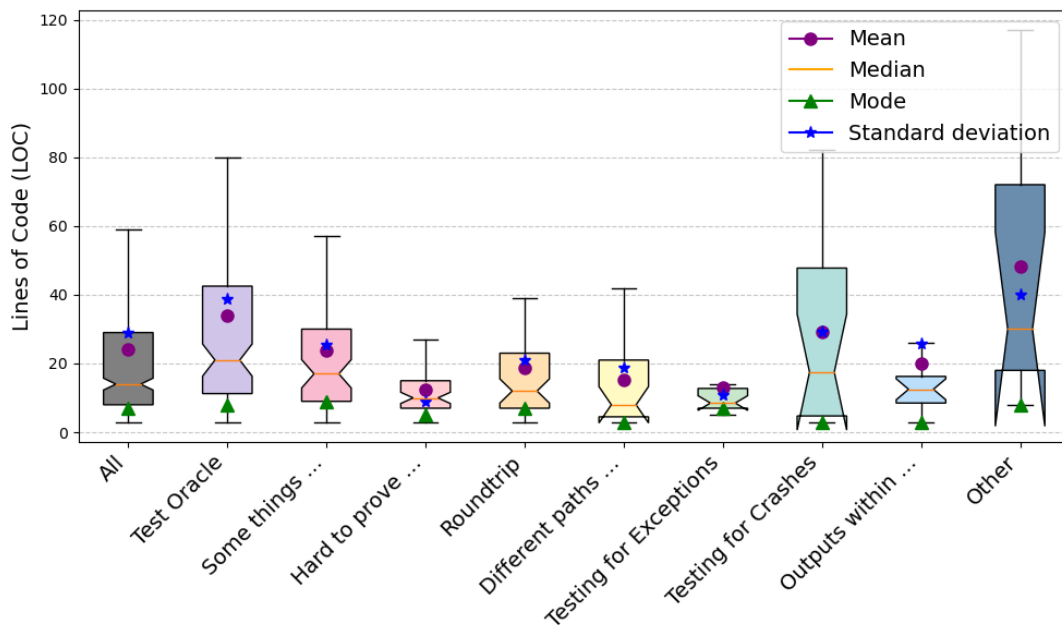


Figure 3.2: Distribution of code lines for all categories

Summary: *Different Paths, Same Destination* (median = 8 LOC) and *Testing for Exceptions* (median = 8.5 LOC) have smaller tests, while *Testing for Crashes* (median = 17.5 LOC) and *Test Oracle* (median = 21 LOC) have larger and more variable ones.

3.5 What are the most commonly used built-in constructs in Hypothesis? (RQ2)

As shown in Table 3.2, among the 367 PBTs, the most frequently used Hypothesis construct is *built-in strategies*—the input generation strategies provided by Hypothesis—appearing in 75.20% of the tests. The next most common construct is `@settings`, used to configure test parameters, as detailed below. By contrast, the least used construct is `note()`, which appears in only 0.82% of the tests. It is important to note that a single test may employ multiple constructs simultaneously, which explains why the total percentages in Table 3.2 exceed 100%.

Table 3.2: Frequency of built-in constructs

Constructs	#	%
built-in strategies	276	75.20
@settings	111	30.25
assume()	26	7.08
@example	19	5.18
note()	3	0.82

3.5.1 Most Common Strategies

Since these are the most popular constructs, we further investigated the strategies used in the PBTs from our dataset. The results are presented in Table 3.3. As shown, the `integers()` strategy is the most frequently used, accounting for 31.60% of occurrences. Next is `sampled_from()`, which generates random values from an existing collection, with 16.04%, followed by `booleans()` at 14.39%. The analysis also revealed less common strategies, such as `dictionaries()` and `fixed_dictionaries()`, which generate dictionaries with fixed keys, together representing 2.59%, and `one_of()`, which selects values from multiple strategies, at 1.89%.

An interesting observation is that only 75.20% of the tests use *built-in strategies* (as mentioned in Table 3.2). To better understand this, we investigated the strategies employed in the remaining tests, since it would be unusual for a test not to use any strategy. Our analysis revealed three other types of strategies, which we classify as:

Table 3.3: Frequency of the built-in strategies

Strategies	#	%
integers()	134	31.60
sampled_from()	68	16.04
booleans()	61	14.39
lists()	40	9.43
text()	33	7.78
floats()	31	7.31
binary()	13	3.07
dictionaries() / fixed_dictionaries()	11	2.59
data()	9	2.12
one_of()	8	1.89
builds()	6	1.42
tuple()	6	1.42
from_regex()	4	0.94
Total	424	100.00

external (i.e., implemented by external projects), internal (i.e., implemented within the same project as the PBTs under analysis), and extension strategies (i.e., implemented at the `hypothesis.extra` module but to generate data for external libraries). Table 3.4 presents the frequency of each. As can be seen in this table, the categories are not disjoint and, therefore, the percentages add up to more than 100%. For example, we have 40 tests that use both built-in and external strategies.

Table 3.4: Frequency of each type of strategy

Strategies	#	%
built-in strategies	276	75.20
external strategies	83	22.62
internal strategies	63	17.17
extension strategies	13	3.54

As mentioned, *external strategies* are provided by modules or packages outside Hypothesis, including general-purpose libraries. An example is shown in Listing 3.5, which illustrates the `anything()` strategy from the project `pfun`, a Python library for functional programming. This strategy generates arbitrary values of any type supported by the library, thereby providing the test with diverse input data. In the example, two instances of `anything()` are used for the first and second parameters (line 3), ensuring the evaluation of different value combinations.⁹

⁹https://github.com/suned/pfun/tests/test_maybe.py#L60

```

1 from pfun.hypothesis_strategies import anything, lists, maybes, unaries
2
3 @given(anything(), anything())
4 def _test_nothing_inequality(self, first, second):
5     assume(first != second)
6     assert Just(first) != Just(second)

```

Listing 3.5: Example of an external strategies

We also analyzed the projects responsible for implementing the external strategies, as shown in Table 3.5. The project that implements most external strategies was `caffe2`, a deep learning framework developed by Facebook, appearing in 42.17% of the tests that use external strategies. The next most frequent projects are `janitor`, a utility library for data preprocessing in `pandas` and `Static-Frame`, an alternative to `pandas`, both accounting for 8.43% of the external strategies in our dataset.

Table 3.5: Distribution of external strategies by project

Project	#	%
caffe2	35	42.17
janitor	7	8.43
static-frame	7	8.43
pfun	5	6.02
hypothesis-gufunc	5	6.02
ivy	4	4.82
PyTorch	4	4.82
brownie	3	3.61
electionguard-tools	3	3.61
other	10	12.05
Total	83	100.00

Internal strategies refer to data generation strategies implemented within the project’s codebase, either as internal modules or file-scope variables. An example is the `st_oid` function shown in Listing 3.6, which is part of the *gecko-dev* project. This strategy generates Object Identifiers (OIDs), which are sequences of numbers, and ensures their validity by defining rules for the first two numbers and the total length of the sequence.¹⁰

```

1 @st.composite
2 def st_oid(draw, max_value=2**512, max_size=50):
3     """
4     Hypothesis strategy that returns valid OBJECT IDENTIFIERS as tuples
5
6     :param max_value: maximum value of any single sub-identifier
7     :param max_size: maximum length of the generated OID
8     """
9     first = draw(st.integers(min_value=0, max_value=2))

```

¹⁰https://github.com/mozilla/gecko-dev/third_party/python/ecdsa/ecdsa/test_der.py#L379

```

10  if first < 2:
11      second = draw(st.integers(min_value=0, max_value=39))
12  else:
13      second = draw(st.integers(min_value=0, max_value=max_value))
14  rest = draw(st.lists(st.integers(min_value=0, max_value=max_value),
15                  max_size=max_size))
16  return (first, second) + tuple(rest)

```

Listing 3.6: Example of an internal strategy to generate object identifiers (OIDs)

Finally, *extension strategies* refer to the use of the `hypothesis.extra` module, which provides strategies to extend Hypothesis with support for external libraries. While `hypothesis.strategies` offers generators for basic Python types, `hypothesis.extra` focuses on data types from popular frameworks, such as NumPy¹¹ and Django¹², which are not part of the standard library. In other words, these strategies are implemented by the Hypothesis developers, similar to the built-in strategies. However, instead of generating Python types, they support tests requiring types from other libraries, as shown in Table 3.6. For this reason, we classified them as a separate category.

Table 3.6: Frequency of extension strategies by extended library

Extended Library	#	%
<code>hypothesis.extra (numpy)</code>	11	84.62
<code>hypothesis.extra (django)</code>	1	7.69
<code>hypothesis.provisional</code>	1	7.69
Total	13	100.00

3.5.2 Most Common Settings

We also analyzed the usage of the `@settings` decorator, used to define test parameters or attributes, as summarized in Table 3.7. The `deadline` attribute, which sets the maximum execution time of a test, is the most common, appearing in 81.08% of tests. The `max_examples` attribute, which controls the maximum number of generated examples, follows at 48.65%. Less frequent attributes include `@suppress_health_check`, which suppresses performance warning messages (16.22%), and `verbosity`, which adjusts the level of detail in the log output (13.51%). Finally, the `parent` attribute, used to inherit the configuration from a parent test profile, is the least used, appearing in only 0.90% of cases.

¹¹<https://github.com/numpy/numpy>

¹²<https://github.com/django/django>

Table 3.7: Frequency of the @settings attributes

Attributes	#	%
deadline	90	81.08
max_examples	54	48.65
suppress_health_check	18	16.22
verbosity	15	13.51
parent	1	0.90

After identifying the prevalence of the `deadline` and `max_examples` attributes, we conducted a detailed analysis of their configurations to understand how they are used in practice. As shown in Table 3.8, among the 90 tests that configure the `deadline` attribute, the vast majority (72.22%) assign it the value `None`. This choice effectively disables the execution deadline, indicating a trend toward tests without time constraints, where example generation can proceed indefinitely or until other limits are reached (such as the maximum number of examples). It is also worth noting that, in the absence of explicit configuration, Hypothesis applies a default `deadline` of 200 milliseconds. The second most common configuration sets the value to 10,000 milliseconds (16.67%), suggesting that when developers do enforce a time limit, it is typically much more permissive than the default.

Table 3.8: Frequency of deadline values

deadline	#	%
None	65	72.22
1000	3	3.33
2000	2	2.22
2500	1	1.11
10000	15	16.67
20000	1	1.11
30000	1	1.11
datetime	2	2.22
Total	90	100.00

For the `max_examples` attribute, which defines the maximum number of examples generated per test, the analysis revealed a distinct pattern compared to the default value set by Hypothesis (100 examples). As shown in Table 3.9, the most commonly requested number of examples is 10, appearing in 36.84% of tests. The second most common is 20 examples, found in 14.04% of the analyzed tests.

Table 3.9: Frequency of `max_examples` values

<code>max_examples</code>	#	%
2	3	5.26
3	3	5.26
5	3	5.26
10	21	36.84
20	8	14.04
50	3	5.26
external	2	3.51
other values	14	24.56
Total	57	100.00

Summary: As expected, strategies are the most frequently used Hypothesis construct (75.20%), followed by `@settings`. We also identified three other types of strategies: external (22.62%), internal (17.17%), and extensions (3.54%). Regarding test parameters, the most common are `deadline` (81.09%), which sets the maximum execution time for tests, and `max_examples` (48.65%), which defines the maximum number of test cases automatically generated by Hypothesis.

3.6 Threats to Validity

In this first study, we used a sample of 367 PBTs, which prevents us from drawing more general claims. We also focused on only one language (Python) and only one testing framework (Hypothesis). However, the most relevant threat lies in the manual classification adopted to answer RQ1. We analyzed PBTs from real systems, which often involve complex logic. Therefore, our classification is subject to errors and questionable decisions. To limit such errors, the classification was carried out by two authors and complemented with two consensus meetings. In addition, our categories were, for the most part, extracted from documents widely referenced by PBT practitioners. On the other hand, counting the constructs used in the implementation of PBTs—as performed in order to answer RQ2—is an objective task and, for this reason, it was performed by only one author. Even so, since it was conducted manually, we consider it may be subject to small errors.

3.7 Final Remarks

This study offered the largest empirical characterization of PBT in Python to date. Our results revealed that *Test Oracle* (29.97%) and *Roundtrip* (13.90%) are the most prevalent categories, and that PBTs are typically concise (median of 14 LOC), relying primarily on Hypothesis’s `built-in` strategies. These findings expose the developers’ preferences and patterns in adopting the PBT paradigm. The insights gathered here set the stage for the following chapter, which investigates the difficulties developers face when applying these concepts in practice through discussions on Stack Overflow.

Chapter 4

Stack Overflow Study

In this chapter, we analyze Stack Overflow questions to identify the key challenges developers face when using the Hypothesis framework. Our goal is to answer the following research question:

RQ3: *What are the primary challenges developers face when using the Hypothesis framework?* Through this question, our goal is to leverage content from a forum like Stack Overflow to identify the main challenges developers face when using Hypothesis. Our aim is to help new PBT developers (by alerting them to common points of confusion) and also assist framework developers (providing them with consolidated information about the questions and challenges faced by users who have chosen to adopt this type of testing).

The remainder of this chapter is organized as follows. Section 4.1 describes the dataset construction process and the methodology used in this study. Section 4.2 presents and discusses the results, addressing RQ3 in depth. Section 4.3 details the threats to validity of our analysis. Finally, Section 4.4 provides the concluding remarks of this chapter and introduces the connection to the next one, which evaluates the automation of Property-Based Test generation through the Ghostwriter tool.

4.1 Dataset and Study Design

The first author of this paper retrieved and analyzed key Stack Overflow questions related to the use of Hypothesis. After a few trial searches, she decided to include only questions tagged as *python-hypothesis* (209 questions) or *property-based-testing* (147 questions). However, among the 147 questions tagged as *property-based-testing*, 120 were considered false positives, as they concerned other testing frameworks or programming languages. Specifically, these 120 questions also had at least one of the following tags:

fscheck, quickcheck, scalacheck, jqwik, kotest, haskell, scala, c++, java, kotlin, typescript, javascript, rust, clojure, go, erlang, ocaml, f#, functional-programming

Therefore, they were discarded. This left 236 questions: 209 with the *python-hypothesis* tag and 27 with the *property-based-testing* tag. However, 20 questions had both tags. Thus, in terms of unique questions, 216 remained. Among these unique questions, three were also discarded (one due to copyright infringement and two that, despite having the *python-hypothesis* tag, were not related to the topic). Consequently, the final dataset is composed of 213 questions.

As a second task, the first author conducted an analysis to identify patterns that would allow the questions to be grouped based on the nature of the problems reported by users, resulting in the inductive identification of seven distinct categories: *Strategies for Data Generation*, *Framework Features*, *General Questions*, *Optimization and Performance*, *Use of Decorators*, *Integration with External Libraries and Reporting*, and *Results Visualization*.

Then, the first author and co-advisor independently performed the manual classification of the questions. Each question was carefully assigned to a category considering the context and the nature of the problem described. Finally, the authors met to discuss and resolve the cases where there was disagreement in the classification. In such cases, the final decision was guided by the objective of the question, i. e., what the user was trying to clarify. The Kappa coefficient computed at the end of the process was 0.80, indicating a substantial level of agreement between the classifiers.

As we will report next, 78 questions (36.6%) were about data generation strategies. For this reason, the two authors decided to further detail this classification and also to re-analyze and classify the types of data developers intended to generate. Thus, independently, both authors classified the data generation strategies discussed in each of these 78 questions. They also held a second consensus meeting to reach the final classification. The Kappa coefficient computed at the end of the process was 0.93, indicating a very high level of agreement between the authors.

4.2 Challenges Developers Face When Using the Hypothesis Framework

Table 4.1 presents the final list of challenges identified in the 213 Stack Overflow questions related to the Hypothesis library. The category *Strategies for Data Generation* stood out prominently, accounting for 36.62% of the analyzed questions. This was followed by *Framework Features* (22.54%), *General Questions* (12.21%), and *Optimization and Performance* (9.39%). The categories with the fewest questions were *Use of Deco-*

rators (7.51%), *Integration with External Libraries* (6.57%), and *Reporting and Results Visualization* (5.16%).

Table 4.1: Distribution of the question categories in the 213 questions of the sample

Categories	#	%
Strategies for Data Generation	78	36.62
Framework Features	48	22.54
General Questions	26	12.21
Optimization and Performance	20	9.39
Use of Decorators	16	7.51
Integration with External Libraries	14	6.57
Reporting and Results Visualization	11	5.16
Total	213	100.00

Next, we discuss and provide examples of questions from each of the categories in Table 4.1.

Strategies for Data Generation (78 questions): This category includes questions about custom data generation strategies or regarding the adaptation of existing strategies for specific tests. For example, in the question shown in Figure 4.1, the developer asks whether Hypothesis can generate data for postal codes, phone numbers, and e-mail addresses.

Can hypothesis be used to generate data of a specific type (ie. zip codes or phone numbers)?

For example, I can define a test to generate integers, but I am expecting this test to conform to valid zip codes as well. Can I do this? Or, perhaps, more complex conform to non-US zip codes, which are all integers, but others (say, Canada) are not?

The same type of thing would be useful for text fields that are expected to conform to some type of mask (ie. email address).

Source: <https://stackoverflow.com/questions/44681295>

Figure 4.1: Example question in the Strategies for Data Generation category

In Subsection 4.2.1, we provide a detailed analysis of the types of data developers attempt to generate with Hypothesis but face problems or unintended behaviors.

Framework Features (48 questions): These questions are related to the behavior of Hypothesis and its features. Consider the question in Figure 4.2, where the user seeks a way to “allow” certain errors in tests, such as those that only occur when processing specific strings.

Python hypothesis: How to assert errors in test function

Simple example is a division function which works for integers >0 :

(...) in this simple example I could simply exclude 0 because I know the error will happen. In other functions, especially the ones working with strings, it would be difficult to exclude all things where I know that an error is thrown. In many cases though the error is ok, e.g. when using null bytes in string functions. Is it possible to somehow 'allow' certain types of errors?

Source: <https://stackoverflow.com/questions/68001322>

Figure 4.2: Example question in the Framework Features category

General Questions (26 questions): These questions concern both comparisons between Hypothesis and other testing tools, as well as general, conceptual doubts about PBT. Consider the question in Figure 4.3, where the user raises a broad issue regarding the application of PBT.

When to use Property-Based Testing?

I am trying to learn Property-Based Testing(PBT). I think I know how to implement it but when should I apply PBT? For example in this case I am trying to compare if the function `getCurrentName()` returns the expected name. Should I randomize this test?

@Test

```
public void getNameTest()  
    assertEquals(nameProxy, proxyFoto.getCurrentName());
```

Source: <https://stackoverflow.com/questions/70333238>

Figure 4.3: Example question in the General Questions category

Optimization and Performance (20 questions): This category covers questions related to improving test performance, reducing execution time, and optimizing resource usage. Consider the question in Figure 4.4, where the developer looks for a more efficient way to generate matrices with independent columns, avoiding the computational overhead of Hypothesis's `assume` function.

Use of Other Decorators (16 questions): This category includes questions related to decorators provided by Hypothesis other than `@given`, such as `@composite`, `@seed`, `@example`, and others. Consider the question in Figure 4.5 where the user wants to understand how to use the `@seed` decorator to reproduce a specific test:

Integration with External Libraries (14 questions): This category concerns questions about integrating Hypothesis with other testing libraries, plugins, or tools. Consider

Generate matrix with independent columns

I am trying to generate matrices with independent columns. At the moment I am using “assume” which works, but requires a lot of computation:

(code example)

Is there a better way to directly create a matrix X with independent columns?

Source: <https://stackoverflow.com/questions/73424597>

Figure 4.4: Example question in the Optimization and Performance category

How to use @seed in hypothesis?

I’m trying to use @seed here: (URL from Hypothesis Documentation)

But when I include it before @given, I get the error `NameError: name 'seed' is not defined at runtime`.

Source: <https://stackoverflow.com/questions/73424597>

Figure 4.5: Example question in the Use of Other Decorators category

the question in Figure 4.6, where the user wants to understand why installing support for Django succeeds, while installing support for Pandas fails.

pip install hypothesis(pandas) says hypothesis3.82.1 does not provide the extra 'pandas'

pip install hypothesis - django - seemed to work and hypothesis.extra has django but not pandas. Any idea what is going on with the pip install for pandas and numpy extras?

Source: <https://stackoverflow.com/questions/53221061>

Figure 4.6: Example question in the Integration with External Libraries category

Reporting and Results Visualization (11 questions): This category covers questions related to the visualization of data generated by Hypothesis during test execution, as illustrated in the question shown in Figure 4.7.

How to see the output of Python's hypothesis library

When using the hypothesis library and performing unit testing, how can I see what instances the library is trying on my code?

(code example)

The question is: how do I print / see the some_number variable, generated by the library?

Source: <https://stackoverflow.com/questions/53072398>

Figure 4.7: Example question in the Reporting and Results Visualization category

4.2.1 Questions on Data Generation

Given the predominance of posts regarding *Strategies for Data Generation*, we chose to deepen our investigation to better understand the specific types of data users aim to generate. The results of this second classification are presented in Table 4.2. As shown, the *Composite Data* subcategory is the most frequent, accounting for 37.18% of all data generation questions. It is followed by *DataFrame and Tabular Data* (24.36%) and *Primitive Data* (19.23%).

Table 4.2: Distribution of the types used in the Strategies for Data Generation category

Subcategories	#	%
Composite Data	29	37.18
DataFrame and Tabular Data	19	24.36
Primitive Data	15	19.23
Object Data	8	10.26
Complex and Recursive Data	7	8.97
Total	78	100.00

Next, we describe these five subcategories of data generation posts:

Composite Data (29 questions): This subcategory refers to questions about generating composite structures or collections, such as tuples, dictionaries, lists, and arrays. Consider Figure 4.8, where the user wants to generate dictionaries with values of different types (e.g., int, string, among others). Although `st.lists` provides the `unique_by` parameter to enforce variety within lists, `st.dictionaries` does not offer a similar mechanism. The core challenge, therefore, is finding a way to ensure that generated dictionaries contain values of distinct types.

Hypothesis search strategy for dictionaries with different types of values

I am trying to generate dictionaries containing different python types as values using the hypothesis module. For lists I can do this simply using the expression

(code example)

But for dictionaries, there is no `unique_by` for the values.

Is there an easy way to generate 'a': int, 'b': str, .. (at least two different python types in `dict.values()`)?

Source: <https://stackoverflow.com/questions/59107181>

Figure 4.8: Example question in the Composite Data subcategory

DataFrame and Tabular Data (19 questions): This subcategory focuses on questions about generating structured data in tabular format, often using libraries such as pandas. Consider Figure 4.9, where the user is generating a `pandas.DataFrame` containing float columns and a date index. The issue is that the generated columns frequently contain only NaN (*Not a Number*) values. Thus, the user needs to ensure two conditions: (1) No column should consist exclusively of NaN values; (2) The `DataFrame` must not be empty.

Python hypothesis dataframe assume column not exclusively consisting of NaN values

I am producing a `pd.DataFrame` using the hypothesis library like so:

(code example) + (table example)

I need to make sure that an individual column doesn't exclusively consist of NaN values. Also, I want to avoid to create an empty `pd.DataFrame`.

Source: <https://stackoverflow.com/questions/78576317>

Figure 4.9: Example question in the DataFrame and Tabular Data subcategory

Primitive Data (15 questions): This subcategory covers questions about generating basic data types and their simple variations, such as integers, floats, text, booleans, and dates. Consider Figure 4.10, where the user is generating random dates with `strategies.dates()`, but needs the result returned as a string in the format "dd.mm.yyyy" instead of a `datetime` object.

Object Data (8 questions): This subcategory involves generating objects with specific attributes and behaviors. Consider the question in Figure 4.11, where the user is able

Hypothesis.strategies generate string from date

My code generates date as `datetime.date` object, but I need it in string format (01.01.2020). So I need to convert it like

```
random_date.strftime("%d.%m.%Y")
```

But I can't find any way to do that. Is it possible to generate string from date in hypothesis?

Source: <https://stackoverflow.com/questions/66704166>

Figure 4.10: Example question in the Primitive Data subcategory

to manually create instances of the `Address` class but expects Hypothesis to automatically construct such objects from annotated types (e.g., `street: str`).

Randomizing a user dataclass with pytest and hypothesis

According to the docs, it seems like it should be possible to use an auto-generated address builder:

(code example)

But when I try the following options, neither work:

```
st.register_type_strategy(Address)
@given(Address)
```

Source: <https://stackoverflow.com/questions/73770201>

Figure 4.11: Example question in the Object Data subcategory

Complex and Recursive Data Structures (7 questions): This subcategory covers the generation of hierarchical and recursive structures, such as trees and nested lists. Consider the question in Figure 4.12, where the user wants to generate boolean expressions following the structure `((A and B) or C)`. They believe that a recursive function could help, but their current implementation produces only single-level combined expressions.

Summary: The analysis of the 213 questions revealed that the main difficulties developers face are concentrated in the *Strategies for Data Generation* category, which accounts for 36.62% of our dataset. Within this category, the subcategory *Composite Data* is the primary source of questions, representing 37.18% of the data generation posts. Other categories that also present significant challenges are *Framework Features* (22.54%) and *General Questions* (12.21%).

How can I generate a boolean expression recursively in Python Hypothesis?

I am new to Python's Hypothesis library and property based testing in general. I want to generate arbitrarily nested policy expressions with the following grammar: ((A and B) or C)

I am feeling that the recursive strategy is what I want, but I'm having a hard time understanding how to use it. The code I have only seems to generate one "level" of expression.

(code example)

How might I generate arbitrarily nested policy expressions using Hypothesis?

Source: <https://stackoverflow.com/questions/52207646>

Figure 4.12: Example question in the Complex and Recursive Data Structures subcategory

4.3 Threats to Validity

Similar to the first study, the classification of Stack Overflow posts is subject to errors and debatable decisions. To mitigate such risks, the classification was performed by two authors. The categories, however, were proposed solely by the first author. The co-advisor, nonetheless, consistently concurred with this proposal and did not identify the need to introduce additional categories. It is worth emphasizing that this classification of Stack Overflow posts was simpler and more objective than that conducted in the first study. This observation is supported by the Kappa coefficients, which were 0.54 (RQ1) and 0.80 (RQ3, as reported in this section).

It is also important to note that, in this study, after a data-cleaning step, we analyzed 213 questions, which represent the valid questions about PBTs implemented using Hypothesis available on Stack Overflow. However, we cannot generalize our results as representative of the concerns of all developers using PBT, since the study was restricted to the Stack Overflow environment.

4.4 Final Remarks

The Stack Overflow analysis revealed that developers frequently struggle with defining effective data generation strategies (36.62%), especially for composite data structures such as tuples, dictionaries, lists, and arrays, which alone account for 24.36% of the questions. Challenges related to Hypothesis decorators and integration with external libraries were also common. These findings highlight the gap between theoretical understanding and the practical application of PBT. Building upon these insights, the next chapter evaluates Ghostwriter, an automated test generation tool, as a possible means to mitigate some of these challenges.

Chapter 5

Assessing a Tool to Generate PBTs

In this chapter, we present our third study, which uses the Ghostwriter tool (for automatic test generation, as described in Subsection 2.4) to address the following research question:

RQ4: *Is the Ghostwriter tool capable of generating PBTs effectively?* Our goal is to verify whether developers who already use, or intend to use, PBTs can rely on Ghostwriter to automate test generation. To answer this question, we build on the results of RQ1 and organize our analysis by PBT categories.

The remainder of this chapter is organized as follows. Section 5.1 describes the methodology adopted in this study. Section 5.2 addresses RQ4 and presents the results of our analysis. Section 5.3 discusses the threats to validity. Finally, Section 5.4 concludes the chapter and closes the set of empirical studies presented in this dissertation.

5.1 Study Design

Of the nine categories proposed in Subsection 3.1, Ghostwriter is only capable of generating tests for four categories from our dataset: *Roundtrip*, *Test Oracle*, *Different Paths*, *Same Destination*, and *Testing for Exceptions*. Therefore, these are the only categories analyzed and considered when answering RQ4.

Initially, the first author carefully evaluated the 203 tests from these four categories, which accounted for 55.31% of the 367-test dataset used to answer RQ1. The remaining tests were excluded because they belong to categories not supported by Ghostwriter. In this evaluation, the author essentially assessed whether the implementation of the selected tests was compatible with, and followed the same structure as the code skeletons generated by Ghostwriter, as previously detailed in Subsection 2.4.

The tests were marked as follows:

- *Eligible for automatic generation:* a test is considered eligible when it fully matches

the code skeleton assumed by Ghostwriter.

- *Partially eligible for automatic generation:* a test is considered partially eligible when it matches the code skeleton assumed by Ghostwriter but also includes additional commands, thus requiring adjustments and improvements.
- *Ineligible for automatic generation:* a test is considered ineligible when it definitively does not match the code skeleton assumed by Ghostwriter.

5.2 Results

The results of RQ4 are presented in Table 5.1. As shown, 105 tests (51.72%) were classified as *ineligible* for automatic generation. This highlights that a tool like Ghostwriter faces significant limitations when used to support the generation of tests in real-world scenarios, such as those in our dataset. In contrast, 37 tests (18.23%) were classified as *eligible*, meaning they could be successfully generated by Ghostwriter. Finally, 61 tests (30.05%) were classified as *partially eligible*.

Table 5.1: Eligibility distribution across categories

Category	Eligible	Partially Eligible	Ineligible
Test Oracle	15	26	69
Roundtrip	12	22	17
Different Paths, Same Destination	9	9	6
Testing for Exceptions	1	4	13
Total	37	61	105

Next, we present three examples of tests from the *Roundtrip* category, illustrating different generation outcomes. Listing 5.1 shows a test classified as eligible for automatic generation by Ghostwriter.¹

```

1 @given(longs)
2 def test_unsigned_roundtrip(self, x):
3     x = abs(x)
4     rx = r_uint(x) # will wrap on overflow
5     assert rbigint.fromrarith_int(rx).toint() == rx
6     rx = r_ulonglong(x) # will wrap on overflow
7     assert rbigint.fromrarith_int(rx).toulonglong() == rx

```

Listing 5.1: Example of an Eligible test in the Roundtrip category

¹https://github.com/mozillazg/pypy/rpython/rllib/test/test_rbigint.py#L1676

When comparing this test with the skeleton generated by Ghostwriter's `roundtrip()` function (presented in Subsection 2.4), we can see that its structure exactly matches what Ghostwriter generates. Therefore, we conclude that this test could have been generated by Ghostwriter.

On the other hand, Listing 5.2 presents an example of a test classified as partially eligible.²

```

1 @given(x=strategies.input_map["double"])
2 def test_double_inverse(x):
3     encoded = fixed.encode_double(x)
4     decoded, pos = fixed.decode_double(encoded, 0)
5     assert pos == len(encoded)
6     if math.isnan(x):
7         assert math.isnan(decoded)
8     else:
9         assert decoded == x

```

Listing 5.2: Example of an Partially Eligible test in the Roundtrip category

Comparing this test with the skeleton generated by Ghostwriter's `roundtrip()` function, we can see that the essential structure is present. For example, the test declares variables whose values are obtained from different methods (lines 3–4), and then an assertion compares both values. However, the test also includes additional commands, such as an `if/else` statement and a second assertion that checks the `math.isnan` function (lines 5–9), which are not part of the skeleton generated by Ghostwriter. Thus, we consider this test to be only partially generable by Ghostwriter.

Finally, the test in Listing 5.3 illustrates an example of a test not eligible for automatic generation by Ghostwriter.³

```

1 @given(st.lists(subtitles()))
2 def test_parsing_content_with_blank_lines(subs):
3     for subtitle in subs:
4         # We stuff a blank line in the middle so as to trigger the "special"
5         # content parsing for erroneous SRT files that have blank lines.
6         subtitle.content = subtitle.content + "\n\n" + subtitle.content
7
8     reparsed_subtitles = srt.parse(srt.compose(subs, reindex=False, strict=False))
9     subs_eq(reparsed_subtitles, subs)

```

Listing 5.3: Example of an Ineligible test in the Roundtrip category

When comparing this test with the skeleton generated by Ghostwriter's `roundtrip()` function, we notice that the typical skeleton structure is missing. For example, although the skeleton requires an assertion to verify the expected behavior, no explicit assertion appears in the code. Instead, this check is encapsulated within the `subs_eq` function. This distinction leads us to conclude that this test could not have been generated by Ghostwriter.

²https://github.com/nccgroup/blackboxprotobuf/lib/tests/py_test/test_fixed.py#L135

³https://github.com/cdown/srt/tests/test_srt.py#L305

Summary: The analysis of 203 tests revealed that only 18.23% could have been automatically generated by Ghostwriter, while 30.05% were classified as *Partially Eligible* and 51.72% as *Ineligible*. These results highlight significant limitations in Ghostwriter’s ability to generate tests in practical scenarios. However, performance varied across categories: *Different Paths*, *Same Destination* showed the highest success rate (37.5%), whereas *Test Oracle* and *Testing for Exceptions* had the highest proportions of ineligible tests with 62.73% and 72.22%, respectively.

5.3 Threats to Validity

The classification of tests as eligible, partially eligible, or ineligible was performed manually by a single author, which may introduce subjectivity. However, the author always tried to base the decision on the template generated by the Ghostwriter tool. If the test followed the template, it could be classified as eligible (only minor changes to the template) or partially eligible (significant changes to the template, including the addition of statements). Otherwise, that is, if the template could not be identified in the code, the result of the classification was ineligible. Therefore, we believe that this criterion, which was carefully applied to all 203 tests, helps to reduce the chances of misclassification.

5.4 Final Remarks

The evaluation of Ghostwriter demonstrated that only a small portion of real-world Property-Based Tests can be fully automated, with just 18.23% of the tests generated successfully. Most tests required partial adaptation (30.05%) or were incompatible with the tool’s generation patterns (51.72%), revealing significant limitations in current automation support. Despite these constraints, Ghostwriter remains a valuable resource for beginners, as it helps them grasp the structure of basic PBT categories. The insights from this study contribute to future improvements in automated PBT generation and conclude the empirical analyzes presented in this dissertation.

Chapter 6

Conclusion

In this chapter, we synthesize the main findings from the three empirical studies and discuss their implications for the use and evolution of PBT in Python. We highlight key lessons, practical recommendations, and directions for future research.

6.1 Overview

This dissertation aimed to provide empirical insights into the practical application of PBT through the analysis of real tests, developer questions, and support tools. By examining a random sample of 367 tests, the results showed that the most common category is *Test Oracle*, followed by *Some Things Never Change* and *Hard to Prove, Easy to Verify*. We also identified two new categories not previously addressed: *Testing for Exceptions* and *Testing for Crashes*. The tests analyzed were generally short, with a median of 14 LOC. Moreover, we found that strategies are the most frequently used Hypothesis constructs, especially `integers`, `lists`, and `booleans`. Our second study focused on the analysis of 213 Stack Overflow questions on PBT. This study revealed that the greatest challenges are related to data generation (36.6%), particularly composite structures (e.g., lists, arrays, tuples) and tabular data (e.g., dataframes). Other important challenges uncovered by the questions were related to framework features (22.54%). Finally, in a third study, we investigated the capabilities of the Ghostwriter tool in generating PBT. The tool has its limitations, and based on our experiments, it would only be able to generate 18% of the tests automatically. However, about 30% more tests could be partially generated. Some test categories are better suited for auto-generation than others; for instance, *Different Paths, Same Destination* showed 37.5% of successfully auto-generated tests and another 37.5% of partially generated ones.

6.2 Lessons Learned and Recommendations

The systematic analysis of PBTs in open-source projects and the investigation of developer discussions in Stack Overflow reveal several opportunities for developers, tool maintainers, and researchers. For example, to start adopting PBTs effectively, we recommend beginning with the most frequent categories. However, although *Test Oracle* is the most popular category (29.97%), it tends to have tests that are more complex and that have higher LOC (median = 21). Thus, a more accessible starting point might be the *Roundtrip* category, which is the fourth most frequent (13.90%) and can also be easily automated by Ghostwriter (23.53%). Another option is the *Different Paths, Same Destination* category, which, although less popular (6.54%), presents tests with lower LOC (median = 8) and a higher success rate in automatic generation (37.50%). In summary, prioritizing categories with smaller and popular tests might help to master key PBT concepts before advancing to more relevant challenges.

The majority of developers' questions on Stack Overflow are related to data generation strategies (36.62%). This aligns directly with the fact that strategies are the most used constructs of Hypothesis in real projects. Specifically, generating more complex structures, such as dictionaries and lists (i.e., *Composite Data*), presents a significant challenge. Thus, resources like the official Hypothesis documentation and practical examples found on Stack Overflow are invaluable for building this knowledge base.

The study also highlights areas for improvement in the Hypothesis framework and its documentation. As mentioned before, the difficulty in generating *Composite Data*, evidenced by the large number of questions on Stack Overflow about the topic (37.18%), suggests the need for more detailed examples and guides. In addition, the recurrence of questions about *Optimization and Performance* (9.39%) indicates an opportunity to offer more guidance and tools that help developers write more efficient PBTs.

The evaluation of the Ghostwriter tool suggests that the automatic generation of PBTs is more effective for categories like *Different Paths, Same Destination* (37.50% of success rate). However, more complex tests, such as those in the *Test Oracle* category, require significant intervention from developers. Future research could focus on developing techniques to automatically generate tests for all categories, including the simplest ones (whose success rate can still be improved) and, of course, for the more complex ones. Finally, it is important to mention that Ghostwriter currently does not generate tests for at least four categories (*Some Things Never Change*, *Hard to Prove*, *Easy to Verify*, *Outputs Within Expected Bounds*, and *Testing for Crashes*).

However, overall, the results obtained with Ghostwriter were not very promising. For instance, in our sample, even after removing tests from categories that are not supported by Ghostwriter, only 18.23% could be generated by the tool. Therefore, we believe

that these results may motivate research on new tools for PBT generation that are not based on templates, such as tools based on AI models, including Large Language Models (LLMs). These tools generally show very promising performance for generating traditional tests, such as unit tests [30], and it may therefore be interesting to also evaluate their use for generating PBTs.

6.3 Contributions

This master’s dissertation makes several empirical and methodological contributions to the understanding and practice of PBT in Python. Through three complementary studies, we provide a comprehensive view of how developers apply, discuss, and automate PBT using the Hypothesis framework.

First, we conducted the largest empirical characterization of Property-Based Tests in Python to date, analyzing 367 real tests from 244 open-source projects. This study extended prior work by introducing two new categories (*Testing for Exceptions* and *Testing for Crashes*) and offered quantitative insights into the typical size, structure, and usage patterns of Hypothesis constructs. These findings form an empirical foundation for future studies on PBT adoption and test design in Python ecosystems.

Second, we performed a systematic investigation of developer challenges by analyzing 213 Stack Overflow posts. This analysis revealed that difficulties in defining effective data generation strategies remain a major barrier to PBT adoption, especially when dealing with composite or tabular data. The study highlights practical areas where improved documentation, educational resources, and framework support could significantly benefit the developer community.

Third, we evaluated the Ghostwriter tool, an automated generator for property-based tests integrated into Hypothesis. By assessing its applicability to 203 real-world tests, we found that only 18.23% of them could be fully automated, exposing limitations in the current tool design. This analysis provides valuable feedback for tool developers and motivates future research on test generation.

Overall, this dissertation contributes both empirical evidence and actionable insights to advance PBT research and practice. It bridges the gap between theoretical understanding and real-world usage, identifies the main obstacles to broader adoption, and outlines concrete opportunities for improving automation and developer experience in PBT.

6.4 Future Work

As future work, we suggest evaluating other widely adopted PBT frameworks beyond Hypothesis, such as `quickcheck` for Haskell and Erlang and `jqwik` for Java. This would allow for comparative analyses across programming languages and software domains, helping to validate the generality of the categories and challenges identified in this study. In addition, future work could analyze the type of requirement validated by PBTs, (e.g., data manipulation, JSON file processing), complementing the RQ1 category-level view with semantic-level analysis of functional testing. This analysis can help explain the difference in the complexity of properties and the effectiveness of tool support across domains. Furthermore, we also propose investigating new solutions for the automatic generation of PBTs. A promising direction is the use of LLMs, exploring techniques capable of overcoming the challenges observed with Ghostwriter by learning from large code corpora to generate complex and realistic test cases. Another relevant direction could investigate *Partially Eligible*, or *Ineligible* generated examples, to better understand why Ghostwriter fails these cases. This could support the creation of a catalog of antipatterns and practices that hinder automation. Finally, we suggest broadening the analysis to additional forums and developer communities to capture diverse needs and practices, and to explore the application of PBT in different domains, such as machine learning models and Android applications.

6.5 Data Availability

We publicly provide the complete dataset and replication package at <https://doi.org/10.5281/zenodo.17117043>, including spreadsheets for Study 1 (Characterization), Study 2 (Stack Overflow), and Study 3 (Ghostwriter Assessment).

Bibliography

- [1] Ahmad Abdellatif, Diego Costa, Khaled Badran, Rabe Abdalkareem, and Emad Shihab. Challenges in Chatbot Development: A Study of Stack Overflow Posts. In *17th International Conference on Mining Software Repositories (MSR)*, pages 174–185, 2020.
- [2] M. Abdi, H. Rocha, S. Demeyer, et al. Small-Amp: Test Amplification in a Dynamically Typed Language. *Empirical Software Engineering (EMSE)*, 27(128), 2022.
- [3] Bernhard K Aichernig and Klaus Havelund. AI-assisted programming with test-based refinement. In *International Conference on Bridging the Gap between AI and Reality (AISoLA)*, pages 385–411, 2023.
- [4] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing Telecoms Software with Quviq QuickCheck. In *2006 ACM SIGPLAN Workshop on Erlang*, pages 2–10, 2006.
- [5] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, et al. Using Lightweight Formal Methods to Validate a key-value Storage Node in Amazon S3. In *28th Symposium on Operating Systems Principles (SOSP)*, pages 836–850, 2021.
- [6] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Computing Surveys (CSUR)*, 51(1), January 2018.
- [7] 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 *15th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, pages 84–97, 2022.
- [8] Koen Claessen and John Hughes. QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279, 2000.
- [9] 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, 2023.

- [10] Jarbele C. S. Coutinho, Wilkerson L. Andrade, and Patricia Machado. Implementing Exploratory Testing in an Agile Context: A Study Based on Design Science Research. In *8th Brazilian Symposium on Systematic and Automated Software Testing (SAST)*, page 67–76, 2023.
- [11] Francisco Dalton, Márcio Ribeiro, Gustavo Pinto, Leo Fernandes, Rohit Gheyi, and Balduino Fonseca. Is Exceptional Behavior Testing an Exception? An Empirical Assessment using Java Automated Tests. In *24th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 170–179, 2020.
- [12] Vinicius HS Durelli, Ricardo Monteiro, Rafael S Durelli, Andre T Endo, Fabiano C Ferrari, and Simone RS Souza. Property-based Testing for Machine Learning Models. In *Symposium on Systematic and Automated Software Testing (SAST)*, pages 39–48. SBC, 2024.
- [13] Khashayar Etemadi, Marjan Sirjani, Mahshid Helali Moghadam, Per Strandberg, and Paul Pettersson. LLM-based property-based test generation for guardrailng cyber-physical systems. In *International Conference on Bridging the Gap between AI and Reality (AISoLA)*, pages 18–46, 2025.
- [14] George Fink and Matt Bishop. Property-Based Testing: a New Approach to Testing for Assurance. *ACM SIGSOFT Software Engineering Notes (SEN)*, 22(4):74–80, 1997.
- [15] Joseph L Fleiss and Jacob Cohen. The Equivalence of Weighted Kappa and the Intra-class Correlation Coefficient as Measures of Reliability. *Educational and Psychological Measurement*, 33(3):613–619, 1973.
- [16] Harrison Goldstein, Joseph W Cutler, Daniel Dickstein, Benjamin C Pierce, and Andrew Head. Property-Based Testing in Practice. In *IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1–13, 2024.
- [17] Harrison Goldstein, Joseph W Cutler, Adam Stein, Benjamin C Pierce, and Andrew Head. Some Problems with Properties, Vol. 1. <https://harrisongoldste.in/papers/hatra2022.pdf>, 2022.
- [18] Harrison Goldstein, Jeffrey Tao, Zac Hatfield-Dodds, Benjamin C Pierce, and Andrew Head. Tyche: Making Sense of Property-Based Testing Effectiveness. 2024.
- [19] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. Perception and Practices of Differential Testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 71–80, 2019.

- [20] Zac Hatfield-Dodds. Falsify Your Software: Validating Scientific Code with Property-Based Testing. In *19th Python in Science Conference (SciPy)*, pages 162–165, 2020.
- [21] Andre Hora and Gordon Fraser. Exceptional Behaviors: How Frequently Are They Tested? In *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 70–79, 2025.
- [22] Upulee Kanewala and Tsong Yueh Chen. Metamorphic Testing: A Simple Yet Effective Approach for Testing Scientific Software. *Computing in Science and Engineering (CiSE)*, 21(1):66–72, January 2019.
- [23] Jeshua S Kracht, Jacob Z Petrovic, and Kristen R Walcott-Justice. Empirically evaluating the quality of automatically generated and manually written test suites. In *14th International Conference on Quality Software (SWQD)*, pages 256–265, 2014.
- [24] Andreas Löscher and Konstantinos Sagonas. Targeted Property-Based Testing. In *26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 46–56, 2017.
- [25] David R MacIver, Zac Hatfield-Dodds, et al. Hypothesis: A New Approach to Property-Based Testing. *Journal of Open Source Software (JOSS)*, 4(43):1891, 2019.
- [26] Diego Marcilio and Carlo A Furia. How Java Programmers Test Exceptional Behavior. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 207–218, 2021.
- [27] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation Testing Advances: an Analysis and Survey. *Advances in Computers*, 112:275–378, 2019.
- [28] Zedong Peng, Upulee Kanewala, and Nan Niu. Contextual Understanding and Improvement of Metamorphic Testing in Scientific Software Development. In *15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021.
- [29] Savitha Ravi and Michael Coblenz. An Empirical Evaluation of Property-Based Testing in Python. *Proceedings of the ACM on Programming Languages (PACMPL)*, 9(OOPSLA2):3897–3923, 2025.
- [30] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2024.

-
- [31] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211, 2015.
- [32] Jingling Sun, Ting Su, Jun Sun, Jianwen Li, Mengfei Wang, and Geguang Pu. Property-Based Testing for Validating User Privacy-Related Functionalities in Social Media Apps. In *32nd ACM International Conference on the Foundations of Software Engineering (FSE)*, pages 440–451, 2024.
- [33] Mohammad Tahaei, Kami Vaniea, and Naomi Saphra. Understanding Privacy-Related Questions on Stack Overflow. In *Conference on Human Factors in Computing Systems (CHI)*, pages 1–14, 2020.
- [34] Marco Tulio Valente. *Software Engineering: A Modern Approach*. Independent, Belo Horizonte, 2024.
- [35] Sten Vercammen, Markus Borg, and Serge Demeyer. Validation of Mutation Testing in the Safety Critical Industry through a Pilot Study. In *2023 IEEE International Conference on Software Testing (ICST), Verification and Validation Workshops (ICSTW)*, pages 334–343, 2023.
- [36] Cindy Wauters and Coen De Roover. Property-based Testing within ML Projects: an Empirical Study. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 648–653, 2024.
- [37] Titus Winters, Tom Manshreck, and Hyrum Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O’Reilly Media, Inc., 2020.
- [38] Yiheng Xiong, Ting Su, Jue Wang, Jingling Sun, Geguang Pu, and Zhendong Su. General and Practical Property-Based Testing for Android Apps. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 53–64, 2024.

Appendix A

Skeletons generated by Ghostwriter

To improve the readability of this work, we chose not to show the skeletons generated by Ghostwriter in Section 2.4. However, to make this study self-contained, in this appendix we present the code of the skeletons generated by Ghostwriter for the following categories:

Roundtrip:

```

1 from hypothesis import given, strategies as st
2
3 # TODO: replace st.nothing() with an appropriate strategy
4
5 @given(data=st.nothing())
6 def test_roundtrip_encoded_decoded(data):
7     value0 = encoded(data=data)
8     value1 = decoded(data_encoded=value0)
9     assert data == value1, (data, value1)

```

Listing A.1: Example of a skeleton generated for the Roundtrip category

Test Oracle:

```

1 from hypothesis import given, strategies as st
2
3 # TODO: replace st.nothing() with an appropriate strategy
4
5 @given(data=st.nothing())
6 def test_equivalent_reverse1_reverse2_reverse3(data):
7     result_reverse1 = reverse1(data=data)
8     result_reverse2 = reverse2(data=data)
9     result_reverse3 = reverse3(data=data)
10    assert result_reverse1 == result_reverse2, (result_reverse1, result_reverse2)
11    assert result_reverse1 == result_reverse3, (result_reverse1, result_reverse3)

```

Listing A.2: Example of a skeleton generated for the Test Oracle category

Different Paths, Same Destination:

```

1 from hypothesis import given, strategies as st
2
3 # TODO: replace st.nothing() with an appropriate strategy
4
5 add_operands = st.nothing()
6
7 @given(a=add_operands, b=add_operands, c=add_operands)
8 def test_associative_binary_operation_add(a, b, c):
9     left = add(a=a, b=add(a=b, b=c))
10    right = add(a=add(a=a, b=b), b=c)
11    assert left == right, (left, right)

```

Listing A.3: Example of a skeleton generated for the Different Paths Same Destination category

Idempotence:

```
1 from hypothesis import given, strategies as st
2
3 # TODO: replace st.nothing() with an appropriate strategy
4
5 @given(data=st.nothing())
6 def test_idempotent_normalize(data):
7     result = normalize(data=data)
8     repeat = normalize(data=result)
9     assert result == repeat
```

Listing A.4: Example of a skeleton generated for the Idempotence category

Testing for Exception:

```
1 from hypothesis import given, reject, strategies as st
2
3 # TODO: replace st.nothing() with appropriate strategies
4
5 @given(x=st.nothing(), y=st.nothing())
6 def test_fuzz_divide(x, y):
7     try:
8         divide(x=x, y=y)
9     except ValueError:
10        reject()
```

Listing A.5: Example of a skeleton generated for the Testing for Exception category