Property-Based Testing of Data Analysis Scripts

A Focus on Hypothesis for Python

Jean-Sebastian de Wet, Jan-Philipp Kiel, and Pascal Mager

University of Cologne, Cologne, Germany
Group Criterion

Abstract. This report explores property-based testing (PBT) as a method to enhance the reliability of data analysis scripts, particularly for research at the German Aerospace Center (DLR) using Python. It outlines challenges with traditional testing in scenarios with diverse data values, emphasising the need for innovative testing approaches. The literature study thoroughly covers PBT, including its history, key concepts, advantages and disadvantages, as well as its classification within the test pyramid. It then focuses on a prototype using Hypothesis for Python, detailing its functionalities, integration with pytest, and unique features. Real-world application is demonstrated through code examples showcasing how Hypothesis strengthens the reliability of data analysis scripts. The report concludes that while PBT shows promise in enhancing software testing, further studies are needed to assess its full potential in research-intensive settings like DLR.

Keywords: Property-Based Testing · Data Analysis Scripts · Hypothesis · Python · pytest · Reliability · Test Pyramid · Code Examples · DLR Research.

1 Introduction

In the evolving landscape of data analysis, the complexity and volume of data has grown exponentially [29], creating unique challenges across various fields. This increase in data complexity requires robust testing methodologies to ensure the accuracy and reliability of data analysis tools and scripts. Traditional testing approaches, which are primarily based on specific input-output cases, often fall short in addressing the dynamic and unpredictable nature of modern datasets. These limitations are particularly evident in specialised fields such as aerospace research, where the volume and variety of data is exceptionally large.

The structure of this report begins with a brief description of the applied methodology, followed by the presentation of results. These results are then discussed in detail, placed in the context of the existing literature and used to derive implications for future works.

2 Background

Within the German Aerospace Center (DLR), the reliability of data analysis scripts is a cornerstone of successful research outcomes. DLR researchers frequently use Python [18], along with powerful libraries like pandas[1] and Matplotlib[2], for complex data manipulations and visualisations. This introduces significant testing challenges. DLR, being at the forefront of aerospace research and development, deals with an enormous range of data variables, from satellite imagery to flight dynamics.[3] This variety and complexity of data make testing particularly challenging. This report explores the adoption of property-based testing (PBT), presenting it as an innovative and essential strategy to overcome these testing challenges.

3 Method

Our methodology involved two key stages.

3.1 Literature Study

The first stage of our methodology involved a thorough literature review on PBT. Our objective was to gain an in-depth understanding of its theoretical foundations, including its history, key principles, and diverse applications. The process consisted of systematic searches in academic databases and online repositories, using targeted keywords such as 'property-based testing' and 'automatic test case generation'. We focused on selecting literature from various domains, particularly papers highlighting PBT's ability to create diverse test inputs. This approach provided a solid knowledge base and included insights on handling complex datasets at the DLR.

3.2 Prototyping

The second stage centered on practical application through the use of Jupyter Notebooks[4]. We developed a comprehensive guide (`tutorial.ipynb`) for using Hypothesis[5]—a PBT framework for Python. This guide covered critical aspects including installation, configuration, and integration with pytest[6]. It also provided concrete examples demonstrating the process of defining properties and generating test inputs using Hypothesis. To validate our methodology, we conducted a case study (`data_analysis.ipynb`) that involved applying PBT to a data analysis script on astronauts. This case study served as a realistic application, illustrating the potential of PBT in complex data scenarios.

[1] pandas is a Python library offering data manipulation and analysis tools
[2] Matplotlib is a Python library for creating visualisations
[3] https://www.dlr.de/en/dlr/about-us, accessed: 2024-01-21
[4] Jupyter Notebooks is a web-based interactive computing platform for data analysis
[5] https://hypothesis.works/, accessed: 2024-01-21
[6] pytest is a Python testing framework

4 Results

First, we uncovered key theoretical insights about PBT through our literature study presented in this report. Second, we demonstrated the practical effectiveness of Hypothesis in our prototype by using code examples from the Jupyter Notebooks included in the `code` folder of this repository.

4.1 Literature Study of Property-Based Testing

History The origins of PBT can be traced back more than 20 years ago, even before the year 2000. While it had already been a subject within information technology research, as seen in Guo et. al. [11] as well as Fink and Bishop [9], it gained significantly more attention with the development of QuickCheck[7] [27, 11, 9, 22, 13]. Initially focusing on research questions related to topics such as automation of test input generation and automated techniques in general [9] and directing the automated input generator towards values with a higher probability of failure [20], recent papers have focused on the implementation of different frameworks or platforms and techniques for PBT application [23, 13, 27, 7]. Moreover, PBT currently enjoys wide-ranging support in different programming languages including automation capabilities [5, 23, 13, 8, 27], as well as the application within many different Python projects using Hypothesis [7].

Key Concepts PBT is a method supporting the formal verification of a software [5, 9, 13, 25], focusing on validating high-level or general properties of the software [9, 13, 7]. Test cases within this method are typically formulated using logical descriptions of the expected behavior of the software [5, 9, 13, 20, 7], including pre- and post-conditions of the system [13]. To formally approximate the system's behavior, a single test case is executed multiple times with randomly generated input to search for counterexamples that violate specific properties or cause a software crash, thereby invalidating said property [5, 20, 23, 8, 25, 7]. Data generators are used for random input generation, which can be adjusted based on a certain domain's needs [5, 20, 23, 8]. Through automated execution of tests with random input, PBT tries to approximate the validity of a property by subjecting it to numerous instantiations within a given input range; otherwise, the property is falsified [9, 8, 7, 25]. To further elaborate on the properties, which represent the desired behaviour in terms of input and output of given tested functions through specifications [5, 9, 20], some examples can be given. To start with a simple one, think of a function that adds two numbers (A, B) and returns the sum of both numbers (A+B). You can define a test which asserts that for any given input for either A and B, the function will return the addition of both numbers. Another simple example would be a sort-function. Assume a sort-function sorts a list, the invariant would be, that sorting an already sorted or the same list twice, will result in a sorted and the same list [7]. A common example which is frequently used for showcasing PBT are binary search trees [7,

[7] QuickCheck is a PBT tool for the Haskell programming language

27], where one property to test might be that after "insert[ing] a key into a valid BST [binary search tree], it should maintain its validity" [27]. In the context of PBT you would then use logic expressions for your tests and use a random input generator to check whether the given property is violated in any case within a certain input range. It is also possible to apply it to software security concerns such as authentication [9] or for verifying the "correctness of hardware [and] external software involved" [5]. In summary, PBT allows for the approximation of a system's invariants formal verification [9, 8, 7].

Advantages and Disadvantages As previously mentioned, PBT supports the formal validation of software by testing specified properties using randomly generated input for each test. However, describing all expected behavior of a system in a logical style is often less feasible [5, 17]. In contrast, by applying PBT the required endeavour for formal validation can be lowered [14, 5, 25]. Moreover, specifications used for PBT might also improve cooperation between software developers and testers in larger projects, as the language used for defining tests is easier to grasp compared to abstract proofs [5, 20]. Besides this, PBT can also be applied to tests in several contexts [16], such as interfaces [16, 10, 19], e.g. by testing invariants regarding the responses of requested URLs of REST-APIs [16]. Other potential domains of application include telecom systems [3], file synchronisation services [15] and databases [4]. Despite its wide applicability and advantages regarding formal validation, PBT reduces engineering effort in terms of defining individual test cases and input parameters [5, 20, 7] and may also incentivise developers to design code that can be easily expressed by properties [5]. More specifically, when compared to manually written tests like in unit-testing, PBT allows the software engineer to put more emphasis on ensuring and restoring correctness of the software and less on "defining test case inputs, examples, and scenarios" [7], turning it into a much less mundane effort [20]. It therefore reduces costs related to testing, including costs induced by changes made to the software [5, 20]. Furthermore it allows for validating a software based on a much larger range of inputs [20, 7] and even more creative or sophisticated inputs [4]. As such PBT complements traditional testing techniques by unveiling yet unknown bugs within even well tested systems [4, 15, 3] and in general, is useful for finding bugs within the implementation of a software and its specifications [5, 9, 20, 25, 6, 7].

Although PBT offers quite some advantages, it does not come without any disadvantages. Due to the randomness of the input generator provided by tools, the chances of finding more specific bugs is reduced, depending on the portion of erroneous inputs of the entire input range, thus potentially failing to unveil errors [20, 23, 8, 27]. It should be noted that processing a large number of tests can lead to reduced efficiency [8, 27], as it involves attempting to approximate a formal proof using numerous randomly selected scenarios [9, 8, 25]. Löscher and Sagonas [20] gave quite a good fictional example using a "system of network nodes" [20]. They tried to falsify the property that for any input scenarios (graphs created), the longest of the shortest paths "between the sink and other nodes [...]

should not exceed 21 hops" [20]. Even after 100000 tests they were not able to falsify the property, which could be done manually [20]. A possible solution to this problem is implied by the usage of individually conceptualised data generators [20, 8, 27, 25, 6] or targeted property-based testing [20]. While constraining data generators in order to cancel a test by using pre-conditions as a form of domain knowledge represents an option to reduce computation effort [20, 8, 27], targeted property-based testing achieves this by guiding the input generator [20]. The idea is to increase the chance of generating inputs causing the violation of a property, by applying search techniques [20]. However, the first technique of implementing specified data generators comes with its own challenges regarding the development [20, 8, 27], resulting in a reduced attractiveness of PBT.

Classification within the Test Pyramid To position PBT within the test pyramid as depicted in Figure 1, our focus was on the levels of unit and integration (service) testing [1, 26]. Existing literature indicates that PBT has not yet been applied for testing at the highest level of either system or user interface (UI) tests [26, 1], leaving these levels outside the scope of our analysis.

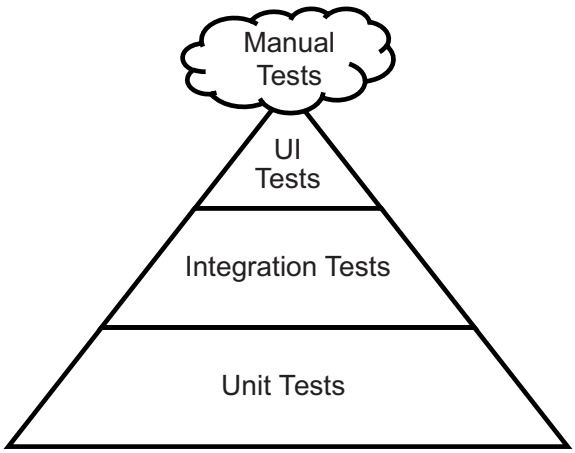

Figure 1. Test Pyramid. Based on Vocke, H.: The Practical Test Pyramid (2018). https://martinfowler.com/articles/practical-test-pyramid.html

Beginning with unit testing, "developers perform unit testing to ensure that each component correctly implements its design and is ready to be integrated into a system of components" [12]. In other words, each function or module of a system is tested in an isolated manner. Furthermore it emphasised on "example-based" [7] testing of individual features or functions—also referred to as the "smallest parts" [1] of a system [12, 7].

Within integration tests, as the name suggests, every component of a system is integrated with each other, including relevant external components [1, 12, 26].

The goal is to ensure that interaction between the given components works correctly and therefore, the combined set of functionality [12, 1].

Although the majority of the use cases explaining how PBT works focus on individual components, such as functions for adding numbers, inserting nodes in binary search trees, and managing sorted lists, it also offers potential applications concerning the physical and external aspects of software [5]. Referring back to the example regarding authentication, one can test not only the authentication functionality of a system but also the integration of its respective authentication services [9]. Additionally, PBT has been utilised in various scenarios such as testing RESTful APIs within the context of OpenAPIs[8] [16], telecom systems [3], synchronisation services [15] or automotive systems [4]. For instance, in the case of QuickREST[9], response codes can be used to differentiate between invalid and valid URL requests, ultimately revealing previously unknown 'underspecification' in the utilised OpenAPI documentation [16]. Similarly, in the context of automotive systems, testing correct request processing unveiled a previously unknown bug related to task prioritisation [4]. Lastly, complete crashes of addressed components can also be observed [3].

Consequently, the PBT method can be applied to both unit and integration testing. In practical terms, PBT is commonly employed for unit testing and is well suited for this purpose, as its highlighted advantages demonstrate complementary effects compared to simple example-based unit testing. However, its applicability is not restricted solely to unit testing. Existing literature has shown that it also extends to testing the integration of various components.

Tools and Programming Languages As previously mentioned, PBT is widely supported across various programming languages [5, 27]. It is supported by Java (QuickTheories[10]), coq (QuickChick[11]), Scala (ScalaCheck[12]), Erlang (PropEr[13]), Haskell (QuickCheck[14]), OCaml (QCheck[15] and Crowbar[16]) to name a few examples [21, 23, 25, 2, 24, 6]. Obviously many of these were inspired by QuickCheck, being the tool popularising PBT. However, this study focuses on Hypothesis, a Python-based PBT implementation framework. This framework has recently attracted significant attention [7, 22], is compatible with pytest and unittest[17], as well as being open source under the Mozilla Public License 2.0[18].

[8] OpenAPI is a specification for defining and documenting RESTful APIs
[9] QuickREST is a library for testing and building OpenAPI-described RESTful APIs
[10] https://github.com/quicktheories/QuickTheories, accessed: 2024-01-21
[11] https://github.com/QuickChick/QuickChick, accessed: 2024-01-21
[12] https://scalacheck.org/, accessed: 2024-01-21
[13] https://proper-testing.github.io/, accessed: 2024-01-21
[14] https://hackage.haskell.org/package/QuickCheck, accessed: 2024-01-21
[15] https://github.com/c-cube/qcheck/, accessed: 2024-01-21
[16] https://github.com/stedolan/crowbar, accessed: 2024-01-21
[17] unittest is Python's built-in testing framework
[18] https://hypothesis.works/products/, accessed: 2024-01-21

4.2 Prototype using Hypothesis for Python

Overview Hypothesis is a Python library for property-based testing, aiming to simplify software testing. Contrary to traditional unit testing, which concentrates on executing a function and asserting specific outcomes, Hypothesis shifts the focus towards ensuring that assertions hold across all data conforming to a predefined specification.

This offers several advantages, inlcuding the identification of edge cases and hidden bugs not typically anticipated in standard testing procedures. The test robustness is enhanced by utilising a wide variety of inputs, some of which are randomly generated. Additionally, it can consolidate multiple traditional unit test cases into a single PBT test case, potentially saving time. Hypothesis also seamlessly integrates with other testing frameworks, such as pytest and nose[19].

However, there are drawbacks to consider. The execution times for test suites may increase due to Hypothesis generating and executing numerous subordinate test cases. While Hypothesis attempts to reproduce the input leading to test failures, understanding the cause of failure can sometimes be challenging. Moreover, PBT typically offers less control compared to traditional unit tests.[20] These pros and cons align with the general advantages and disadvantages of PBT as discussed in our literature study.

Strategies and Data Generation A central concept within Hypothesis is the utilisation of search strategies. These strategies refer to how Hypothesis will try to generate inputs for test functions. Or rather, wha it will use to search for bugs. For that, Hypothesis is equipped with a diverse set of search strategies, enabling it to generate various types of data, including text, floats, integers, boolean values, values from specified sets, and values conforming to given regular expressions. Furthermore, it is possible to refine these strategies using input parameters. Such refinements can include constraints like limiting the range of generated float values or specifying permissible date ranges.

The implementation of search strategies within Hypothesis adheres to a consistent format. Typically, a search strategy object is instantiated using the syntax `st.(type)`, where `type` indicates the desired search strategy, such as `text`, `float`, or `date`. This instantiation process allows for further refinement through the specification of parameters. These parameters can impose constraints, for example, restricting dates to a specified range or generating floats with a maximum size limit. By default, Hypothesis is configured to generate a series of 100 distinct tests for each defined strategy.

Installation In order to use Hypothesis, the package must first be installed using Python's package installer pip. This can be done by executing the command `pip install hypothesis`. The specific configuration used in this study is described as follows:

[19] nose is a Python testing framework that extends the built-in unittest library
[20] https://hypothesis.readthedocs.io/en/latest/, accessed: 2024-01-10

- Operating System: Ubuntu 22.04.3 LTS
- Python Version: 3.10.12
- pip Version: 22.0.2
- Hypothesis Version: 6.92.2

It is worth noting that the Hypothesis team is officially committed to supporting the latest version of Python.[21]

Usage To understand the use of Hypothesis, consider the following basic test shown in Code Example 1 below. The complete Code Examples 1–5 can be found in the provided `tutorial` Jupyter Notebook.

```
1  from hypothesis import given
2  import hypothesis.strategies as st
3
4  @given(st.integers())
5  def test_builtin_abs(x: int) -> None:
6      assert abs(x) >= 0
7      assert abs(x) == (x if x >= 0 else -x)
8
9  test_builtin_abs()
```

Code Example 1. Basic Test from `tutorial.ipynb`

The primary method of annotating a Hypothesis test involves the `given` decorator (line 4). This decorator accepts a search strategy object, utilising it to supply values for the function's parameters. In the presented case, integers are generated as inputs for the parameter x in line 5.

The test function then asserts that certain properties hold for each generated value of x. In this specific example, the test validates the correct functionality of Python's built-in absolute value function (lines 6–7).

Another feature is the ability to create new, specialised search strategies. The following Code Example 2 illustrates the creation of complex inputs:

```
1  from hypothesis.strategies import composite
2
3  PI = 3.14159
4
5  @composite
6  def custom_input_generator(draw) -> tuple[float, str]:
7    decimal = draw(st.floats(max_value=PI))
8    text = draw(
9      st.text(alphabet=st.characters
10         (whitelist_categories=['Lu']),
```

[21] https://hypothesis.readthedocs.io/en/latest/, accessed: 2024-01-10

```
11            min_size=2, max_size=5))
12    return decimal, text
13
14  @given(custom_input_generator())
15  def test_custom_input_generator
16        (generated_input: tuple[float, str]) -> None:
17    decimal, text = generated_input
18    assert decimal <= PI
19    assert len(text) >= 2 and len(text) <= 5
20    assert text.isupper()
```

Code Example 2. Complex Inputs from `tutorial.ipynb`

The `composite` decorator (line 5) is utilised to define a function for generating a customised search strategy, achieved by combining existing strategies. In the example above, the float search strategy is used to generate floats with a maximum value of π in line 7. Then, the text search strategy is used to create strings consisting solely of uppercase letters (line 10), with a length contraint of 2–5 characters (line 11). The test function (line 14–20) then asserts that the generated decimal and text adhere to the specified structure and constraints.

Improving the robustness of tests can be achieved by increasing the number of test cases generated as shown in Code Example 3 below:

```
1   from hypothesis import settings
2
3   @settings(max_examples=100)
4   @given(st.integers())
5   def test_builtin_abs(x: int) -> None:
      ...
9
```

Code Example 3. Specifying Test Case Amount from `tutorial.ipynb`

This approach not only helps to improve test coverage, but also approximates formal verification by generating a large amount of random input. The test quantity generated can be adjusted using the `settings` decorator in line 3.

To apply this setting globally at the beginning of a Python file, the following approach in Code Example 4 can be used:

```
1   settings.register_profile("default", max_examples=100)
2   settings.load_profile("default")
```

Code Example 4. Global Settings from `tutorial.ipynb`

Another useful feature of Hypothesis is its capability to specify a seed, ensuring reproducibility of test cases as demonstrated in the following Code Example 5:

```
1  from hypothesis import given, seed
2  import hypothesis.strategies as st
3
4  @seed(30)
5  @given(st.integers())
6  def test_builtin_abs(x: int) -> None:
   ...
11
```

Code Example 5. Specifying Seeds from `tutorial.ipynb`

The `seed` decorator (line 4), along with a designated seed value, is used to set a seed. By setting a seed, Hypothesis generates a consistent set of test cases, thereby eliminating the element of randomness. This is particularly helpful for debugging purposes, as Hypothesis can identify and present the specific seed that led to a failure. Additionally, incorporating specific seeds into a test suite, alongside randomised tests, can enhance testing reliability.

Integration with pytest The integration of Hypothesis with pytest is seamless. To initiate the testing process, simply run the command `pytest`, which will automatically detect and run the Hypothesis tests. This automatic detection is exemplified in the included `tutorial` Jupyter Notebook, where it can be observed that pytest accurately identifies Hypothesis tests. However, it is crucial to recognise that although pytest registers a Hypothesis test as a singular test entity, in reality, it executes multiple underlying tests.

For acquiring more in-depth information regarding Hypothesis, the optional flag `--hypothesis-show-statistics` can be utilised. As shown in the shortened output below, this flag displays detailed information specific to Hypothesis:

```
==================== Hypothesis Statistics ====================
test.py::test_builtin_abs:
  - during generate phase (0.02 seconds):
    - Typical runtimes < 1ms, of which < 1ms in data generation
    - 100 passing, 0 failing, 0 invalid examples
  - Stopped because settings.max_examples=100
```

Data Analysis Scripts Hypothesis is a powerful tool for testing data analysis scripts, especially for validating data preparation and cleaning. Its strength lies in its ability to simulate a wide range of inputs, which increases the robustness of the functions. In the following case study, we examine the application of Hypothesis to an exemplary data analysis script on astronauts provided by Stoffers and Schlauch [28] from the DLR. The complete Code Examples 6–10 can be found in the included `data_analysis` Jupyter Notebook.

The first function covered is `calculate_age`, as shown in the Code Example 6 below. It serves as an auxiliary function in the data preparation script.

```
1
   ...
7  def calculate_age(born):
8    today = date.today()
9    return today.year - born.year -
10     ((today.month, today.day) < (born.month, born.day))
```

Code Example 6. Calculate Age from `data_analysis.ipynb`

It calculates the current age by subtracting the year of birth from the present year (line 9) and adjusts for birthdays not yet occurred in this year (line 10).

The testing of this function is demonstrated in the following Code Example 7, using the `test_calculate_age` function:

```
1  @given(st.dates(min_value=date(1920, 1, 1),
2                  max_value=date.today()))
3  def test_calculate_age(born: date) -> None:
4    age = calculate_age(born)
5    assert age >= 0
6       and age <= (born.today().year - born.year)
7
8  test_calculate_age()
```

Code Example 7. Test Calculate Age from `data_analysis.ipynb`

In this test, the `given` decorator generates random dates between the 1st of January 1920 (line 1) and today (line 2). The function then asserts that the calculated age is non-negative (line 5) and does not exceed the maximum possible age based on the year of birth (line 6). This ensures the correct functionality of the `calculate_age` function.

The next function `prepare_data_set` is arguably the most important in the script, being responsible for preparing the data sets as shown in Code Example 8:

```
1  def prepare_data_set(df):
2    df = rename_columns(df)
3    df = df.set_index("astronaut_id")
4
5    # Set pandas dtypes for columns with date or time
6    df = df.dropna(subset=["time_in_space"])
7    df["time_in_space"] = df["time_in_space"].astype(int)
8    df["time_in_space"] = pd.to_timedelta(df["
       time_in_space"], unit="m")
9    df["birthdate"] = pd.to_datetime(df["birthdate"])
```

```
10      df["date_of_death"] = pd.to_datetime(df["
            date_of_death"])
11      df.sort_values("birthdate", inplace=True)
12
13      # Calculate extra columns from the original data
14      df["time_in_space_D"] = df["time_in_space"] / pd.
            Timedelta(days=1)
15      df["alive"] = df["date_of_death"].apply(is_alive)
16      df["age"] = df["birthdate"].apply(calculate_age)
17      df["died_with_age"] = df.apply(died_with_age, axis=1)
18      return df
    ...
46
```

Code Example 8. Prepare Data Set from `data_analysis.ipynb`

Hypothesis is well suited for testing even complex functions. In the above example, the function transforms an astronaut dataset by renaming columns (line 2), managing missing values and data types (lines 6–10), and adding analytical columns (lines 14–17).

The following `astronaut_data` function in Code Example 9 is designed to generate appropriate inputs for the previous `prepare_data_set` function:

```
1   @st.composite
2   def astronaut_data(draw) -> dict:
3     astronaut = draw(st.from_regex(r"http://www\.wikidata
          \.org/entity/Q\d+", fullmatch=True))
4     astronautLabel = draw(st.from_regex(r"[A-Z][a-z]+ [A-
          Z][a-z]+", fullmatch=True))
5     birthdate = draw(st.dates(min_value=date(1920, 1, 1),
           max_value=date(2030, 12, 31)))
6     birthplaceLabel = draw(st.from_regex(r"[A-Z][a-z]+",
          fullmatch=True))
7     sex_or_genderLabel = draw(st.sampled_from(["male", "
          female"]))
8     time_in_space = draw(st.integers(min_value=1,
          max_value=900))
9     date_of_death = draw(st.one_of(
10         st.none(),
11         st.dates(birthdate + timedelta(days=1),
12                  max_value=date(2030, 12, 31))
13         )
14    )
      ...
40
```

Code Example 9. Astronaut Data from `data_analysis.ipynb`

The `astronaut_data` function creates an dictionary with various inputs (lines 1–2). Specifically, it generates an astronaut link and label using `st.from_regex` search strategies (lines 3–4), and a birthdate within a specified range using the `st.dates` strategy (line 5). The sex or gender label is generated using `st.sampled_from`, drawing from male and female values (line 7). The time in space is generated using `st.integers` within a given range (line 8). Additionally, the date of death is generated using `st.one_of` to choose between two strategies: `st.none` for the possibility of no death date (line 10), and `st.dates` to ensure a date after the birthdate (line 11) and within a realistic timeframe (line 12). This combination of strategies in Hypothesis enables the generation of complex and realistic test data. Through the use of regular expressions, specified date ranges and diverse sampling methods, comprehensive inputs for data analysis scripts are ensured.

Identifying Issues Hypothesis also proves valuable in identifying potential bugs. By removing the maximum value constraint of the time in space strategy (line 8 in Code Example 9), Hypothesis exposes an issue that causes the script to crash, as indicated by the following error message:

```
OverflowError: Python int too large to convert to C long
Falsifying example: test_prepare_data_set(
    data=[{'astronaut': 'http://www.wikidata.org/entity/Q0',
      'astronautLabel': 'Aa Aa',
      'birthdate': '2000-01-01T00:00:00Z',
      'birthplaceLabel': 'Aa',
      'sex_or_genderLabel': 'male',
  >>  'time_in_space': 18446744073709551616,  <<
      'date_of_death': '2000-01-02T00:00:00Z'}],)
```

This overflow error can be traced back to line 7 of the `prepare_data_set` function presented in detail in Code Example 8. The corresponding line is highlighted in Code Example 10 below:

```
7 df["time_in_space"] = df["time_in_space"].astype(int)
```

Code Example 10. Trace Overflow Error from `data_analysis.ipynb`

The error occurs when a Python integer exceeds the size limits of a C[22] long data type, leading to overflow during interactions with C-based extensions or libraries. The data preparation script can therefore be improved by ensuring that processed integers are within an error-free range.

[22] C is a low-level language used by Python for core functions and library interaction

Another illustrative example occurs when the maximum date range constraint in the date of death strategy (line 12 in Code Example 9) is removed, resulting in the following error:

```
pandas._libs.tslibs.np_datetime.OutOfBoundsDatetime:
Out of bounds nanosecond timestamp: 2263-01-01T00:00:00Z,
at position 0
Falsifying example: test_prepare_data_set(
    data=[{'astronaut': 'http://www.wikidata.org/entity/Q0',
      'astronautLabel': 'Aa Aa',
      'birthdate': '2000-01-01T00:00:00Z',
      'birthplaceLabel': 'Aa',
      'sex_or_genderLabel': 'male',
      'time_in_space': 1,
  >>  'date_of_death': '2263-01-01T00:00:00Z'}],)  <<
```

This out of bounds error can be traced back to line 10 of the `prepare_data_set` function detailed in Code Example 8. The corresponding line is showcased in Code Example 11 below:

```
10 df["date_of_death"] = pd.to_datetime(df["date_of_death"])
```

Code Example 11. Trace Out Of Bounds Error from `data_analysis.ipynb`

The error occurs due to the limitations of the NumPy[23] `datetime64` data type, which can only represent dates within a certain range, typically from '1677-09-21' to '2262-04-11'. This range limitation stems from its use of 64-bit integers to encode dates as nanoseconds elapsed since the Unix epoch. Attempting to convert a date outside this range, such as '2263-01-01', exceeds the maximum value that `datetime64` can represent, resulting in an error. This issue underscores the importance of including checks in data scripts to ensure that dates fall within the supported range of the underlying data type.

5 Discussion

Our structured literature study on PBT has provided in-depth insights into this approach as an alternative to traditional testing methods. We identified the key aspects of PBT and compared the advantages and disadvantages with those of conventional methods, some of which have been addressed in existing research. Furthermore, the literature has explored the suitability of PBT across different levels of the test pyramid.

Through developing a prototype using Hypothesis for Python, we have illustrated the concepts from the literature. Hypothesis fundamentally supports the core functions of PBT. For instance, it facilitates the generation of random inputs for automated test execution while also enabling the implementation of

[23] NumPy is a core Python library for numerical operations used by pandas

specific data generators. This capability approximates the formal verification of essential software components.

Moreover, using this prototype, Hypothesis was applied to an exemplary data analysis script at the DLR. In our case study, we successfully generated complex inputs for tests, identifying ways to make the script more robust as demonstrated in the date-range example. Besides direct analysis functions, Hypothesis enables extensive testing of the input data. It can be customised similar to unit tests and integrated with pytest, supplementing or replacing the traditionally labor-intensive and numerous unit tests. This enhances the reliability of analysis scripts and is generally accessible even to less experienced software developers.

PBT fundamentally is a method that relies less on technical know-how and more on logical specifications of systems, thereby likely fostering collaboration between software developers and testers. However, the known drawbacks of PBT were also evident. Specifically, the increased effort required to develop specialised input generators and a potential reduction in testing efficiency should be noted.

The applicability of Hypothesis in the context of the DLR is therefore conceivable. However, the scope of this work is limited as it only represents a relatively simple use case. We have demonstrated the basic functionality of Hypothesis and its application to an exemplary analysis script. In order to fully assess its suitability as a complement to established testing methods and its potential benefits for the DLR, further structured and comprehensive use cases need to be investigated. This will lead to a more definitive understanding of the role and value of PBT in improving software testing practices in research-intensive environments such as the DLR.

6 Conclusion

In this study, we have explored and applied the method of PBT, enriching our understanding with insights from existing research and an introduction to the Hypothesis framework. It presents notable advantages, by offering capabilities for automated and randomised generation of test inputs to test general properties of systems. PBT proposes a way to supplement traditional testing methods, significantly enhancing the reliability of developed systems.

In addition, our prototype demonstrated a potential application for PBT in the context of data analysis scripts at the DLR. This practical example implies the possible benefits and utility of PBT in real-world research settings.

However, it is important to acknowledge the limitations of our study, primarily due to the narrow scope of our prototype. The results we have presented here are preliminary and need to be further validated by applying PBT to a broader range of test cases and larger-scale projects. This will enable a more comprehensive understanding of the effectiveness of PBT and its potential to transform software testing practices, particularly in complex and data-intensive research environments like those at the DLR. This future work will be crucial in fully realising the benefits of PBT for enhancing software reliability and confidence in research outcomes.

References

1. Aniche, M.: Effective software testing. Manning Publications Co, Shelter Island, NY, 1 edn. (2022), includes bibliographical references and index
2. Arts, T., Castro, L.M., Hughes, J.: Testing erlang data types with quviq quickcheck. In: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG. ICFP08, ACM (Sep 2008). https://doi.org/10.1145/1411273.1411275
3. Arts, T., Hughes, J., Johansson, J., Wiger, U.: Testing telecoms software with quviq quickcheck. In: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang. ICFP06, ACM (Sep 2006). https://doi.org/10.1145/1159789.1159792
4. Arts, T., Hughes, J., Norell, U., Svensson, H.: Testing autosar software with quickcheck. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE (Apr 2015). https://doi.org/10.1109/icstw.2015.7107466
5. Chen, Z., Rizkallah, C., O'Connor, L., Susarla, P., Klein, G., Heiser, G., Keller, G.: Property-based testing: Climbing the stairway to verification. In: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering. SLE '22, ACM (Nov 2022). https://doi.org/10.1145/3567512.3567520
6. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming. ICFP00, ACM (Sep 2000). https://doi.org/10.1145/351240.351266
7. Corgozinho, A.L., Valente, M.T., Rocha, H.: How developers implement property-based tests. In: 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE (Oct 2023). https://doi.org/10.1109/icsme58846.2023.00049
8. Elazar Mittelman, S., Resnick, A., Perez, I., Goodloe, A.E., Lampropoulos, L.: Don't go down the rabbit hole: Reprioritizing enumeration for property-based testing. In: Proceedings of the 16th ACM SIGPLAN International Haskell Symposium. Haskell '23, ACM (Aug 2023). https://doi.org/10.1145/3609026.3609730
9. Fink, G., Bishop, M.: Property-based testing: a new approach to testing for assurance. ACM SIGSOFT Software Engineering Notes **22**(4), 74–80 (Jul 1997). https://doi.org/10.1145/263244.263267
10. Francisco, M.A., López, M., Ferreiro, H., Castro, L.M.: Turning web services descriptions into quickcheck models for automatic testing. In: Proceedings of the twelfth ACM SIGPLAN workshop on Erlang. ICFP'13, ACM (Sep 2013). https://doi.org/10.1145/2505305.2505306
11. Guo, R., Reddy, S.M., Pomeranz, I.: Proptest: a property based test pattern generator for sequential circuits using test compaction. In: Proceedings of the 36th annual ACM/IEEE Design Automation Conference. DAC99, ACM (Jun 1999). https://doi.org/10.1145/309847.310019
12. Hartmann, J., Imoberdorf, C., Meisinger, M.: Uml-based integration testing. In: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis. ISSTA00, ACM (Aug 2000). https://doi.org/10.1145/347324.348872
13. Honarvar, S., Mousavi, M.R., Nagarajan, R.: Property-based testing of quantum programs in q#. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. ICSE '20, ACM (Jun 2020). https://doi.org/10.1145/3387940.3391459

14. Hritcu, C., Lampropoulos, L., Spector-Zabusky, A., de Amorim, A.A., Dénès, M., Hughes, J., Pierce, B.C., Vytiniotis, D.: Testing noninterference quickly. Journal of Functional Programming **26** (2016). https://doi.org/10.1017/s0956796816000058
15. Hughes, J., Pierce, B.C., Arts, T., Norell, U.: Mysteries of dropbox: Property-based testing of a distributed synchronization service. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE (Apr 2016). https://doi.org/10.1109/icst.2016.37
16. Karlsson, S., Causevic, A., Sundmark, D.: Quickrest: Property-based test generation of openapi-described restful apis (2019). https://doi.org/10.48550/ARXIV.1912.09686
17. Koopman, P., Achten, P., Plasmeijer, R.: Model Based Testing with Logical Properties versus State Machines, pp. 116–133. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-34407-7_8
18. von Kurnatowski, L., Schlauch, T., Haupt, C.: Software development at the german aerospace center: Role and status in practice. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. ICSE '20, ACM (Jun 2020). https://doi.org/10.1145/3387940.3392244
19. Lamela Seijas, P., Li, H., Thompson, S.: Towards property-based testing of restful web services. In: Proceedings of the twelfth ACM SIGPLAN workshop on Erlang. ICFP'13, ACM (Sep 2013). https://doi.org/10.1145/2505305.2505317
20. Löscher, A., Sagonas, K.: Targeted property-based testing. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA '17, ACM (Jul 2017). https://doi.org/10.1145/3092703.3092711
21. MacIver, D.: Quickcheck in every language (Apr 2016), https://hypothesis.works/articles/quickcheck-in-every-language/
22. MacIver, D., Hatfield-Dodds, Z., Contributors, M.: Hypothesis: A new approach to property-based testing. Journal of Open Source Software **4**(43), 1891 (Nov 2019). https://doi.org/10.21105/joss.01891
23. Padhye, R., Lemieux, C., Sen, K.: Jqf: coverage-guided property-based testing in java. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA '19, ACM (Jul 2019). https://doi.org/10.1145/3293882.3339002
24. Papadakis, M., Sagonas, K.: A proper integration of types and function specifications with property-based testing. In: Proceedings of the 10th ACM SIGPLAN workshop on Erlang. ICFP '11, ACM (Sep 2011). https://doi.org/10.1145/2034654.2034663
25. Paraskevopoulou, Z., Hriţcu, C., Dénès, M., Lampropoulos, L., Pierce, B.C.: Foundational Property-Based Testing, pp. 325–343. Springer International Publishing (2015). https://doi.org/10.1007/978-3-319-22102-1_22
26. Radziwill, N., Freeman, G.: Reframing the test pyramid for digitally transformed organizations (2020). https://doi.org/10.48550/ARXIV.2011.00655
27. Shi, J., Keles, A., Goldstein, H., Pierce, B.C., Lampropoulos, L.: Etna: An evaluation platform for property-based testing (experience report). Proceedings of the ACM on Programming Languages **7**(ICFP), 878–894 (Aug 2023). https://doi.org/10.1145/3607860
28. Stoffers, M., Schlauch, T.: Astronaut analysis (3 2021). https://doi.org/10.5281/zenodo.5018166, https://doi.org/10.5281/zenodo.5018166
29. Taylor, P.: Amount of data created, consumed, and stored 2010-2020, with forecasts to 2025 [infographic]. Statista (Nov 2023), https://www.statista.com/statistics/871513/worldwide-data-created/