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 datasets have grown exponentially [29], presenting unique challenges across various fields. This surge in data complexity necessitates robust testing methodologies to ensure the accuracy and reliability of data analysis tools and scripts. Traditional testing approaches, 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 like aerospace research, where the data's scope and diversity are exceptionally vast.

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 its 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, presenting it as an innovative and essential strategy to overcome these testing challenges, especially in scenarios involving large possible value ranges of data.

3 Method

Our methodology in tackling these challenges involved two key stages.

3.1 Literature Study

The first stage of our methodology involved a thorough literature review on property-based testing. Our objective was to gain an in-depth understanding of its theoretical foundations, including its history, key principles, and diverse applications. The process entailed 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 cases. This approach provided a solid knowledge base and included insights on handling complex datasets in DLR. These findings are complemented in the next stage of our study.

3.2 Prototyping

The second stage centered on practical application, where we developed a comprehensive guide for using Hypothesis, a property-based testing framework for Python. This guide covered critical aspects including installation, configuration, and integration with Python's pytest[4] framework. It also provided practical examples demonstrating the process of defining properties and generating test cases using Hypothesis. To validate our methodology, we conducted a case study that involved applying property-based testing to a data analysis script related to astronauts. This case study served as a real-world application, illustrating the practicality of property-based testing in complex data scenarios.

[1] https://pandas.pydata.org/, accessed: 2024-01-21
[2] https://matplotlib.org/, accessed: 2024-01-21
[3] https://www.dlr.de/en/dlr/about-us, accessed: 2024-01-21
[4] https://pytest.org/, accessed: 2024-01-21

4 Results

First, we uncovered key theoretical insights on property-based testing through our literature study. Then, we demonstrated the practical effectiveness of this approach in our prototype using Hypothesis.

4.1 Literature Study of Property-Based Testing

History The origins of property-based testing (PBT) can be traced back more than 20 years ago, even before 2000. While it had already been a subject within information technology research, as seen in Guo (1999) and Fink (1997), it gained significantly more attention with the development of QuickCheck[5] [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]. To top it off, PBT nowadays 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 as a framework for PBT [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- or post-conditions of the system [13]. To formally validate 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,

[5] https://www.cse.chalmers.se/%7Erjmh/QuickCheck/, accessed: 2024-01-21

27], where one property to test might be that after "insert[ing] a key into a valid BST, 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 test 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 in 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]. Therefore 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 (2017) 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 by hand [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 In order to classify PBT within the test pyramid, we focus on the level of unit and integration (service) testing [1, 26]. Current literature indicates that PBT has not yet been applied for testing at the highest level of either system or UI tests [26, 1], making them irrelevant for our analysis.

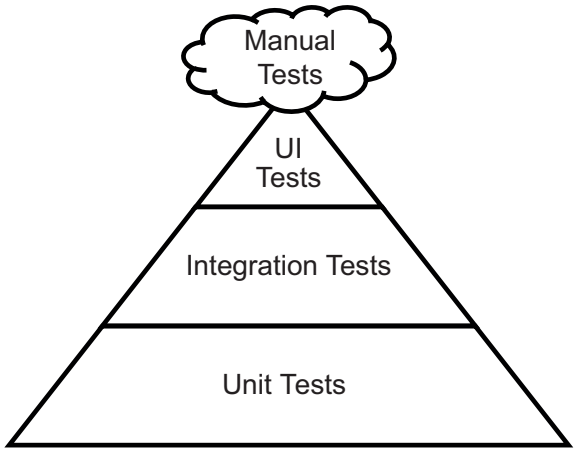

Fig. 1. Test Pyramid. Based on Vocke, H.: The Practical Test Pyramid (2018). https://martinfowler.com/articles/practical-test-pyramid.html

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[6] [16], telecom systems [3], synchronisation services [15] or automotive systems [4]. For instance, in the case of QuickREST[7], 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[8]), coq (QuickChick[9]), Scala (ScalaCheck[10]), Erlang (QuickCheck[11] and PropEr[12]), Haskell (QuickCheck[13]), OCaml (QCheck[14] and Crowbar[15]) 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[16], a Python-based PBT implementation framework. This framework has attracted significant attention recently [7, 22], is compatible with pytest and unittest[17], as well as being open source under the Mozilla Public License 2.0[18].

[6] OpenAPI is a specification for defining and documenting RESTful APIs
[7] QuickREST is a library for testing and building OpenAPI-described RESTful APIs
[8] https://github.com/quicktheories/QuickTheories, accessed: 2024-01-21
[9] https://github.com/QuickChick/QuickChick, accessed: 2024-01-21
[10] https://scalacheck.org/, accessed: 2024-01-21
[11] http://www.quviq.com/products/erlang-quickcheck/, accessed: 2024-01-21
[12] https://proper-testing.github.io/, accessed: 2024-01-21
[13] https://hackage.haskell.org/package/QuickCheck, accessed: 2024-01-21
[14] https://github.com/c-cube/qcheck/, accessed: 2024-01-21
[15] https://github.com/stedolan/crowbar, accessed: 2024-01-21
[16] https://hypothesis.works/, accessed: 2024-01-21
[17] https://docs.python.org/3/library/unittest.html, accessed: 2024-01-21
[18] https://hypothesis.works/products/, accessed: 2024-01-21

4.2 Prototype of Data Analysis Scripts using Hypothesis for Python

Overview Hypothesis is a Python library for Property-Based Testing. The aim is to have a simple manner in which software can be tested. In traditional unit testing the focus is on performing the function and asserting something about the result. With Hypothesis, the focus is on trying to make the assertions hold for all data matching a certain specification.

The advantages are that Hypothesis helps to discover edge cases and hidden bugs which one might not think of during typical testing. It also helps to make the tests more robust, seeing as a wide variety of inputs, some even random, are used. Furthermore, it can help save time, by encompassing many traditional unit test cases into one PBT test case. And lastly, Hypothesis also integrates into other testing frameworks such as pytest and nose. In terms of disadvantages, Hypothesis might lead to longer execution times of test suites. This is due to a Hypothesis test case generating many more subordinate test cases and running them. Although Hypothesis tries to reproduce the input that caused the test case to fail, this might not always work and it can therefore sometimes be challenging to understand why a test case failed. Lastly, one typically has less control when using PBT in comparison to a typical unit test.[19]. The majority of these advantages and disadvantages correspond with the general advantages and disadvantages of PBT that can be found within the academic literature.

Strategies and Data Generation One of the main concepts of Hypothesis is the idea of search strategies. Search strategies refer to how Hypothesis will try to generate input for the test function. Or rather, what it will use to 'search' for bugs. Hypothesis has plenty of different search strategies that can be used. For instance, it can generate text, floats, integers, boolean values, a value of a given set of values and values that match a given regex. Furthermore, these search strategies can be refined by input parameters. This can, for instance, limit the size of the floats generated or allow only dates between date ranges.

The format of using a search strategy in Hypothesis is consistent. Typically, an object representing a search strategy is instantiated using the syntax `st.(type)`, where `type` is the specific search strategy to use (e.g., `text`, `float`, `date`). The instantiation can be further refined by specifying parameters. The parameters can for instance, limit the dates to a certain range or only generate floats up to a certain size. By default, Hypothesis will try to create 100 different tests

Hypothesis Usage In order to use Hypothesis, one first needs to install the Hypothesis package. This can be done by using the command `'pip install hypothesis'`. The specific setup used is as follows:

- Operating System: Ubuntu 22.04.3 LTS
- Python Version: 3.10.12

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

- pip Version: 22.0.2
- Hypothesis Version: 6.92.2

However, Hypothesis officially tries to support the latest version of Python.[20]

Consider the following code extract:

```
from hypothesis import given
import hypothesis.strategies as st

@given(st.integers())
def test_builtin_abs(x: int) -> None:
    assert abs(x) >= 0
    assert abs(x) == (x if x >= 0 else -x)

test_builtin_abs()
```

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

The main way in which the Hypothesis test is annotated, is using the `given` decorator. The `given` decorator takes a search strategy object and uses this to populate the parameters of the function. In this case, integers are generated to be used as input to the x parameter of the function.

This function then asserts that certain properties hold for each value of x generated. In this example, it tests that the built-in absolute value function of Hypothesis works as intended.

Another key aspect of Hypothesis is being able to create new, unique search strategies. The following code block is an example of this:

```
from hypothesis.strategies import composite

PI = 3.14159

@composite
def custom_input_generator(draw) -> tuple[float, str]:
  decimal = draw(st.floats(max_value=PI))
  text = draw(
    st.text(alphabet=st.characters
    (whitelist_categories=['Lu']),
  min_size=2, max_size=5))
  return decimal, text

@given(custom_input_generator())
def test_custom_input_generator
```

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

```
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 is used to specify a function that generates a custom search strategy. The function works by combining existing search strategies. So in the above, the float search strategy is used to generate floats up until a max value of π. Then, the text search strategy is used to generate text that consists of only upper-case letters and is between 2 and 5 characters long. The test function simply tests that the search strategy generation function generates output of the correct form.

In order to improve test robustness, it can be useful to increase the number of tests generated. Furthermore, generating large amounts of random test cases helps to approximate formal verification. The number of test cases that are generated can be changed with the `settings` decorator and the `max_samples` property:

```
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`

Globally this can be done as follows in the beginning of the Python test file:

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

Code Example 4. Specifying Test Case Amount from `tutorial.ipynb`

Lastly, another useful feature of Hypothesis is the ability to specify a seed. When using a seed, the test cases generated by Hypothesis will always be identical. This eliminates the random element. Seeds can be helpful to debug code, because Hypothesis will try to determine and present the specific seed that caused the program to fail. It might also be useful to have, in addition to randomized tests, specific seeds in the test suite in order to improve testing reliability. A seed can be set using the `seed` decorator and a specific seed value.

Here is an example:

```
from hypothesis import given, seed
import hypothesis.strategies as st

@seed(30)
@given(st.integers())
def test_builtin_abs(x: int) -> None:
...

```

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

Integration with pytest The integration with pytest is automatic. It is sufficient to run pytest in the traditional sense, as pytest will automatically recognize the Hypothesis tests. In the `tutorial.ipynb` notebook this is illustrated. It can be seen that pytest correctly identified the Hypothesis test. However it is important to note that pytest registers a Hypothesis test as a single test. In actuality, multiple tests are run. By using the optional `'--hypothesis-show-statistics'` flag, one can get more detailed information specific to Hypothesis.

The returned output could look like this:

```
====================== Hypothesis Statistics ======================
test.py::test_builtin_abs:

  - during generate phase (0.02 seconds):
    - Typical runtimes: < 1ms, of which < 1ms in data generation
    - 100 passing examples, 0 failing examples, 0 invalid examples

  - Stopped because settings.max_examples=100
```

Application for Data Analysis Scripts Hypothesis can be used to test data analysis scripts. In particular, Hypothesis can be helpful to ensure that the data preparation and data cleaning steps are properly tested. This is due to Hypothesis being able to simulate a wide range of inputs, which can then be used to ensure that the functions are robust. In the following, we will apply Hypothesis to an exemplary astronaut data analysis script [28] from the DLR.

The first function that will be examined is the `calculate_age` function:

```
...
def calculate_age(born):
  today = date.today()
  return today.year - born.year - ((today.month, today.
      day) < (born.month, born.day))
```

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

This is an auxiliary function in the data preparation script. It takes a date object and calculates the current age by considering the time that has passed since the given date. This function can be tested as follows:

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

test_calculate_age()
```

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

The test generates random dates between the 1st of January 1920 and the current date. It then confirms that the `calculate_age` function works as intended.

The next function that will be tested is arguably the most important function in the script, the function that is used to prepare the data sets:

```
def prepare_data_set(df):
  df = rename_columns(df)
  df = df.set_index("astronaut_id")

  # Set pandas dtypes for columns with date or time
  df = df.dropna(subset=["time_in_space"])
  df["time_in_space"] = df["time_in_space"].astype(int)
  df["time_in_space"] = pd.to_timedelta(df["
      time_in_space"], unit="m")
  df["birthdate"] = pd.to_datetime(df["birthdate"])
  df["date_of_death"] = pd.to_datetime(df["
      date_of_death"])
  df.sort_values("birthdate", inplace=True)

  # Calculate extra columns from the original data
```

```
    df["time_in_space_D"] = df["time_in_space"] / pd.
        Timedelta(days=1)
    df["alive"] = df["date_of_death"].apply(is_alive)
    df["age"] = df["birthdate"].apply(calculate_age)
    df["died_with_age"] = df.apply(died_with_age, axis=1)
    return df
...

```

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

Hypothesis is well suited to test even a complicated function like the above. The focus will be on generating appropriate inputs for the `prepare_data_set` function:

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

```

Code Example 9. Prepare Data Set from `data_analysis.ipynb`

This function generates a wide variety of inputs. Specifically, it generates an astronaut link and label using regex search strategies. It then generates a birthdate that falls within a certain range using the `st.dates` strategy and specifying the date range. The `sex_or_genderLabel` is then generated using `st.sampled` which then allows Hypothesis to generate input using the values `male` and `female`. The time in space is then generated using `st.integers` and confined to a given range. Then the date of death is used by combining two strategies, `st.none` and `st.dates`, of which one will be selected. The birthdate and, if present, death

date are then transformed into the corresponding format that the data preparation script expects. This shows how Hypothesis can be used to generate complex inputs which can then be used to test data analysis scripts.

Discovering Issues Hypothesis can also be very helpful in finding potential bugs and making data analysis scripts more robust. In particular, by removing the max value of the time in space strategy, Hypothesis will trigger the data analysis script to crash. The following output is returned:

```
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 specific error can be traced back to line 7:

```
1
   ...
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) #
       This line caused the error
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)
   ...
60
```

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

This error occurs when a Python integer exceeds the size limits of a C 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.

As for another example, by removing the max date range the following error will be produced:

```
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'}],)
```

The `datetime64[ns]` data type in pandas/NumPy is limited to the range of dates from `'1677-09-21'` to `'2262-04-11'` because it stores dates as 64-bit integers representing nanoseconds since the Unix epoch (1 January 1970). Therefore, if NumPy tries to convert the date of death, the script will crash. This shows yet another example of how the data script can be made more robust. Specifically, by checking that the dates are within range that pandas/NumPy can process.

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

Moreover, with the help of 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 tailored 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 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 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 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/