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

Abstract. This paper explores property-based testing as a method to ensure data analysis scripts' reliability, especially in DLR research using Python, Pandas, and Matplotlib. It outlines challenges with traditional testing in scenarios with diverse data values, emphasizing the need for innovative testing strategies. The paper thoroughly covers property-based testing, including its history, key principles, use cases, and integration in the test pyramid. It then focuses on Hypothesis for Python, a powerful tool for property-based testing, discussing its use, integration with pytest, and unique features. Real-world application is demonstrated with code examples, highlighting how property-based testing, especially with Hypothesis, strengthens data analysis scripts' reliability. The paper concludes by summarizing key findings and emphasizing the crucial role property-based testing, like Hypothesis, plays in boosting researchers' confidence with unknown data.

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 specialized fields like aerospace research, where the data's scope and diversity are exceptionally vast.

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 [17], along with its powerful libraries like Pandas[1] and Mat-

[1] https://pandas.pydata.org/, accessed: 21.01.2024

plotlib[2], for complex data manipulations and visualizations. 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 paper 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 phases.

3.1 Literature Study

The first phase 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 that emphasized the capability of property-based testing in generating a wide range of test cases, an essential feature for managing complex datasets typical in DLR research.

3.2 Prototyping

The second phase 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.

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.

[2] https://matplotlib.org/, accessed: 21.01.2024
[3] https://www.dlr.de/en/dlr/about-us, accessed: 21.01.2024
[4] https://pytest.org/, accessed: 21.01.2024

4.1 Overview of Property-Based Testing

History of Property-Based Testing The origins of property-based testing (PBT) can be traced back more than 20 years ago, even before 2000. Although it had already been a topic within information technology research like in Goldreich (1998) and Fink (1997), it gained much more attention through the development of QuickCheck[5] [27, 9, 21, 12]. Beginning with research questions concerning topics like automation of test input generation and automated techniques in general [9] and guiding the automated input generator towards values with higher probability of failure [19], more recent papers deal with the implementation of different frameworks or platforms and techniques of PBT-application [23, 12, 27, 7]. To top it off, PBT already enjoys wide-ranging support in different programming languages including automation capabilities [5, 23, 12, 8, 27], as well as the application within many different python projects [7].

How Property-Based Testing Works PBT is a method enabling the formal verification of a software [5, 9, 12, 25], with the key concept of validating high-level or general properties of a software [9, 19, 12, 7]. Test cases used within the application of this method, are usually formulated using logical descriptions of a software's expected behaviour [5, 9, 12, 19, 7]. More explicitly, these tests may include pre- or post-conditions of the system [12]. In order to provide formal validation of the system's behaviour, a single test case is executed many times with randomly generated input in search for counterexamples resulting in violation of a specific property or even a crash of a software, therefore invalidating said property [5, 19, 23, 8, 25, 7]. For randomly generating input, data generators are used [5, 19, 23, 8] which can be adjusted according to "domain-specific knowledge" [5]. Through this automated execution of tests with random input PBT tries to approximate the validity of a certain property, as it has to withstand the check using many different instantiations within a given input range or otherwise the property is falsified [9, 8, 7, 25]. To further elaborate on the properties, which represent the desired behaviour in terms of input / output of given tested functions through specifications [5, 9, 19], 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). Hier ggfs. Beispiel für sortierter Liste einfügen –> Beinhaltet "Invariante" 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 [7]. Another example which is frequently used for showcasing PBT are binary search trees [7, 27]. In PBT "one desirable property [could be], that if we [you] insert a key into a valid BST, then it should remain a valid BST" [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. Other possible use cases could be software-security related as example in the case of authentication [9] or "the correctness of hardware [and] the external soft-

[5] https://www.cse.chalmers.se/ rjmh/QuickCheck/, accessed: 21.01.2024

ware involved" [5]. Long story short, PBT allows for the formal verification of a system's invariants [9, 8, 7].

Advantages and Disadvantages of Property-Based Testing As already mentioned, PBT allows to formally validate the correctness of a software by testing specified properties using randomly generated input for each test. However, describing all of a system's expected behaviour in a logical style is paired with reduced feasibility [5, 16]. By applying PBT, required endeavour for formal validation can be lowered [13, 5, 25]. Moreover, specifications used for PBT might also improve cooperation between software engineers (SE) and software testers in larger projects, as the language used for defining tests is easier to grasp compared to abstract proofs [5, 19]. Besides this, PBT can also be applied to test in "multiple domains" [15] - to name a few examples next to the given examples of the previous chapter: interfaces [15, 10, 18], e.g. by testing invariants regarding the responses of requested URLs of REST-APIs [15]. Other possible domains are telecom systems [3], file synchronisation services [14] and databases [4]. Despite its wide applicability and advantages regarding formal validation, it lowers engineering effort in terms of defining individual test cases as well as input parameters [5, 19, 7] and might as well offer incentives for SEs to design the code to be easily expressed by properties [5]. More specifically, when compared to manually written tests like in unit-testing, PBT allows the SE to put more emphasis on ensuring and restoring correctness of the software and less on "defining test case inputs, examples, and scenarios" [7], which is also less "mundane" [19]. It therefore reduces costs related to testing including change induced costs [5, 19]. Furthermore it allows for validating a software based on a much larger range of inputs [19] [Hypothesis for Software Testing Research] 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, 14, 3] and in general, is useful for finding bugs within the implementation of a software and its specifications [5, 9, 19, 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 reduces depending on the portion of erroneous inputs of the entire input range and therefore might fail to unveil errors [19, 23, 8, 27]. Not to mention that an enormous amount of tests processed implies reduced efficiency [8, 27], as you try to approximate a formal proof using many randomly picked scenarios [9, 8, 25]. Löscher (2017) gave quite a good fictional example using a "system of network nodes" [19]. 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" [19]. Even after "100000 tests" [19] they were not able to falsify the property, which could be done "by hand" [19]. A possible solution to this problem is implied by the usage of individually conceptualised data generators [19, 8, 27, 25, 6] or targeted property-based testing [19]. While constraining data generators by using domain knowledge in order to reduce the e.g. by using pre-conditions to cancel

a test [19, 8, 27], targeted property-based testing tries to guide the input generator "with search techniques towards values, that have a higher probability of falsifying a property" [19]. Especially the first mentioned technique comes with its own challenges caused by the required development of your own generators, resulting in a reduced attractiveness of PBT [19, 8, 27].

Position in the Test Pyramid In order to position PBT at a level of the testing pyramid, we focus on the level of unit and integration (service) testing [1, 26]. To the best of our knowledge, PBT has not been applied to test the highest lvl of either system or UI tests [26, 1], thus the highest lvl of the testing pyramid is ignored in discussing the position of PBT within the test pyramid. The relevant levels are now shortly summed up.

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" [integration testing]. In other words, "testing is performed in isolation from other components" [integration testing] and focuses on "example-based" [7] testing of individual parts – the "smallest parts" [1] – of a system [11, 7].

Within integration tests, every component of a system is integrated with each other if needed, including relevant external components [1, 11, 26]. The goal is to ensure that interaction between the given components works correctly and are therefore properly integrated [11, 1].

Although most of the mentioned use cases of chapter 2 (how does PBT work) relate to individual components in the means of functions such as adding numbers, inserting nodes in binary search trees and the sorted list [müssen hier die indirekten Zitate rein?], PBT also offers possible application surrounding the physical and external components of a software [5]. Coming back to the authentication-functionality provided by a system, you can not only test the functionality of authentication but also the integration with said authentication services [9]. Furthermore PBT has already been applied in many different cases of testing RESTful APIs in the context of OpenAPIs [15], telecom systems [3], synchronisation services [14] or AUTOSAR software [4]. In the case of QuickREST you can use the response codes in order to differentiate between invalid and URL-requests, which in the end lead to revealing a yet unknown "underspecification" [15] of a used OpenAPIS documentation [15]. Whereas in the case of AUTOSAR the testing of correct request processing unveiled a yet unknown bug in terms of task prioritisation [4]. Last but not least, entire crashes of addressed components can be perceived as well [3].

Therefore the method of PBT can be located within the level of unit and integration testing. It can be very well applied to testing individual components of a system, but is not limited to it, because testing the integration of components is also possible and is used for both in practice.

Tools and Programming Languages As already mentioned, PBT enjoys wide ranging support in many different programming languages [5, 27]. It is sup-

ported by Java (Quicktheories[6]), coq (QuickChick[7]), Scala (ScalaCheck[8]), Erlang (QuickCheck[9] and PropEr[10]), Haskell (QuickCheck[11]), OCaml (QCheck[12] and Crowbar[13]) to name a few examples [20, 23, 25, 2, 24, 6]. Obviously many of these tools were inspired by QuickCheck, being the tool popularising PBT. However in this work we focus on Hypothesis[14], a PBT implementing framework for Python. It is a framework receiving much attention recently [7, 21], which is compatible with pytest, unittest[15] and "probably many others", while being open source "under the Mozilla Public License 2.0".[16]

4.2 Introduction to Hypothesis for Python

Overview of Hypothesis Hypothesis is a Python library for Property-Based Testing. The focus is on having an easy way to test code with a wide variety of inputs. The main difference when writing test cases in Hypothesis, versus traditional unit tests, is that instead of giving concrete input, performing the function, and asserting something about the result, 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 were not thought 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 [22].

How to Use Hypothesis To use Hypothesis, start by installing the Python package. This can be done by using the command `pip install hypothesis`. In this project, the following setup was used:

[6] https://github.com/quicktheories/QuickTheories, accessed: 21.01.2024
[7] https://github.com/QuickChick/QuickChick, accessed: 21.01.2024
[8] https://scalacheck.org/, accessed: 21.01.2024
[9] http://www.quviq.com/products/erlang-quickcheck/, accessed: 21.01.2024
[10] https://proper-testing.github.io/, accessed: 21.01.2024
[11] https://hackage.haskell.org/package/QuickCheck, accessed: 21.01.2024
[12] https://github.com/c-cube/qcheck/, accessed: 21.01.2024
[13] https://github.com/stedolan/crowbar, accessed: 21.01.2024
[14] https://hypothesis.works/, accessed: 21.01.2024
[15] https://docs.python.org/3/library/unittest.html, accessed: 21.01.2024
[16] https://hypothesis.works/products/, accessed: 21.01.2024

– Operating System: Ubuntu 22.04.3 LTS
– Python Version: 3.10.12
– pip Version: 22.0.2

However, officially, Hypothesis tries to support the latest version of Python [22].

Consider the following code snippet:

```
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)
```

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. A search strategy object refers to how the input for the function should be generated. 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 case, it tests that the built-in absolute value function of Hypothesis works as it is intended to work.

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
        (generated_input: tuple[float, str]) -> None:
    decimal, text = generated_input
    assert decimal <= PI
    assert len(text) >= 2 and len(text) <= 5
    assert text.isupper()
```

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 PI. 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.

4.3 Main Concepts and Features of Hypothesis

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 always the same. 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.

Data Analysis Applications and Benefits Hypothesis can be used to test data analysis scripts. In particular, Hypothesis can be really 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.

4.4 Application of Hypothesis in Data Analysis

Code Examples The Data Analysis script that Hypothesis will be applied to, is a script that was provided by the DLR [28]. 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))
```

This is an auxiliary function, that is 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)
```

The test generates random dates between the 1st of January 1920 and the current date and then it 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. This what the function looks like:

```
  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["time_in_space_D"] =
  df["time_in_space"].astype("timedelta64[D]")
  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)
```

Please note that the line

```
df["time_in_space_D"] = df["time_in_space"].astype("timedelta64[D]")
```
(1)

from the original script has been altered to

```
df["time_in_space"] / pd.Timedelta(days=1)
```
(2)

in the revised version. Hypothesis is well-suited to test even a complicated function like the above. The focus will be on generating appropriate inputs to the above.

```
  @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))
    )
)

birthdate_str = birthdate.strftime(
  "%Y-%m-%dT00:00:00Z")
date_of_death_str = date_of_death.strftime(
  "%Y-%m-%dT00:00:00Z") if date_of_death else None

return {
    "astronaut": astronaut,
    "astronautLabel": astronautLabel,
    "birthdate": birthdate_str,
    "birthplaceLabel": birthplaceLabel,
    "sex_or_genderLabel": sex_or_genderLabel,
    "time_in_space": time_in_space,
    "date_of_death": date_of_death_str
}
```

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 as well as by specifiying a min and max value. The sex or gender label is then generated using st.sampled from which then allows Hypothesis to generate input using the values male and "female". The time in space is then generated using st.integers() and constraint 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. This will result in the following output.

Illustrative Cases 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 by Hypothesis:

```
  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'}],
)
```

The specific can be traced back to :

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

....
```

And it is due to a Python integer being to large to be converted to a C long. The data preparation script can therefore be improved, by checking beforehand that the integers are within a specified 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 (January 1, 1970). Therefore, if numpy tries to convert the date of death, '2263-01-01T00:00:00Z', the script will crash. This shows yet another example of how the data script can be made more robust. Specifically, be checking that the dates are within date ranges that pandas/Numpy can process.

5 Discussion

6 Conclusion

- Summarize the key points discussed in the paper.
- Emphasize the importance of property-based testing, particularly with tools like Hypothesis, in enhancing the reliability of data analysis scripts.

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. 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
12. 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
13. 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
14. 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
15. 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
16. 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
17. 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
18. 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
19. 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

20. MacIver, D.: Quickcheck in every language (Apr 2016), https://hypothesis.works/articles/quickcheck-in-every-language/
21. 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
22. MacIver, D.R.: Hypothesis 6.96.2 documentation. https://hypothesis.readthedocs.io/en/latest/ (2024), accessed: 2024-01-10
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/